# LA GUIDA COMPLETA
# C++

# Indice


**Prefazione** **XV**

**PARTE PRIMA LE BASI DEL C++: IL LINGUAGGIO C** **1**

Capitolo 1 **Una panoramica sul linguaggio C** **3**
1.1 Le origini del linguaggio C 3
1.2 Il C è un linguaggio di medio livello 4
1.3 Il C è un linguaggio strutturato 5
1.4 Il C è un linguaggio per programmatori 7
1.5 L'aspetto di un programma C 9
1.6 La libreria e il linker 10
1.7 Compilazione separata 12
1.8 Le estensioni di file: .c e .cpp 12

Capitolo 2 **Le espressioni** **15**
2.1 I cinque tipi di dati principali 15
2.2 Modificare i tipi principali 16
2.3 Nomi degli identificatori 18
2.4 Le variabili 19
2.5 I modificatori di accesso 25
2.6 Specificatori di classe di memorizzazione 27
2.7 Inizializzazione delle variabili 34
2.8 Le costanti 35
2.9 Gli operatori 38
2.10 Le espressioni 56

Capitolo 3 **Le istruzioni** **61**
3.1 La verità e la falsità in C e C++ 62
3.2 Le istruzioni di selezione 62

3.3 Le istruzioni di iterazione 74
3.4 La dichiarazione di variabili nelle istruzioni di selezione e iterazione 85
3.5 Le istruzioni di salto 86
3.6 Le espressioni 92
3.7 I blocchi 93

Capitolo 4 **Gli array e le stringhe** **95**
4.1 Gli array monodimensionali 95
4.2 La generazione di un puntatore a un array 97
4.3 Come passare un array monodimensionale a una funzione 98
4.4 Le stringhe 99
4.5 Gli array bidimensionali 102
4.6 Gli array multidimensionali 108
4.7 L'indicizzazione dei puntatori 109
4.8 L'inizializzazione degli array 111
4.9 L'esempio del tris (tic-tac-toe) 114

Capitolo 5 **I puntatori** **119**
5.1 Che cosa sono i puntatori? 119
5.2 Variabili puntatore 120
5.3 Gli operatori per i puntatori 121
5.4 Espressioni con puntatori 122
5.5 Puntatori e array 127
5.6 Indirizzamento multilivello 129
5.7 Inizializzazione di puntatori 131
5.8 Puntatori a funzioni 133
5.9 Le funzioni di allocazione dinamica del C 136
5.10 Problemi con i puntatori 138

Capitolo 6 **Le funzioni** **143**
6.1 La forma generale di una funzione 143
6.2 Regole di visibilità delle funzioni 144
6.3 Gli argomenti delle funzioni 145
6.4 Gli argomenti di main(): argc e argv 150
6.5 L'istruzione return 154
6.6 Ricorsione 160
6.7 Prototipi di funzioni 162
6.8 Dichiarazione di elenchi di parametri di lunghezza variabile 165
6.9 Dichiarazione di parametri con metodi vecchi e nuovi 165
6.10 Elementi implementativi 166





Capitolo 7 **Strutture, unioni, enumerazioni e tipi definiti dall'utente** **169**
7.1 Le strutture 170
7.2 Gli array di strutture 174
7.3 Il passaggio di strutture alle funzioni 175
7.4 I puntatori a strutture 177
7.5 Gli array e strutture all'interno di altre strutture 181
7.6 I campi bit 182
7.7 Le unioni 185
7.8 Le enumerazioni 188
7.9 Uso di sizeof per assicurare la trasportabilità del codice 191
7.10 La parola riservata typedef 193

Capitolo 8 **Operazioni di I/O da console** **195**
8.1 Un'importante nota applicativa 196
8.2 La lettura e la scrittura di caratteri 196
8.3 La lettura e la scrittura di stringhe 199
8.4 Le operazioni di I/O formattato da console 202
8.5 La funzione printf() 203
8.6 La funzione scanf() 211

Capitolo 9 **Operazioni di I/O da file** **219**
9.1 Operazioni di I/O C e C++ 219
9.2 Stream e file 220
9.3 Gli stream 220
9.4 I file 221
9.5 Principi di funzionamento del file system 222
9.6 fread() e fwrite() 235
9.7 fseek() e operazioni di I/O ad accesso diretto 237
9.8 fprint() e fscanf() 239
9.9 Gli stream standard 240

Capitolo 10 **Il preprocessore e i commenti** **245**
10.1 Il preprocessore 245
10.2 La direttiva #define 246
10.3 La direttiva #error 249
10.4 La direttiva #include 250
10.5 Le direttive per compilazioni condizionali 250
10.6 La direttiva #undef 254
10.7 Uso di defined 255
10.8 La direttiva #line 256
10.9 La direttiva #pragma 256
10.10 Gli operatori del preprocessore # e ## 257

10.11 Le macro predefinite 258
10.12 I commenti 259

**PARTE SECONDA ● IL LINGUAGGIO C++ 261**

Capitolo 11 **Panoramica del linguaggio C++ 263**
11.1 Le origini del C++ 263
11.2 Che cos'è la programmazione a oggetti 265
11.3 Elementi di base del linguaggio C++ 268
11.4 C++ vecchio stile e C++ moderno 275
11.5 Introduzione alle classi C++ 279
11.6 L'overloading delle funzioni 284
11.7 L'overloading degli operatori 287
11.8 L'ereditarietà 288
11.9 I costruttori e i distruttori 293
11.10 Le parole riservate del C++ 297
11.11 La forma generale di un programma C++ 297

Capitolo 12 **Le classi e gli oggetti 299**
12.1 Le classi 299
12.2 Le strutture e le classi 303
12.3 Le unioni e le classi 305
12.4 Le funzioni friend 307
12.5 Le classi friend 312
12.6 Le funzioni inline 313
12.7 Definizione di funzioni inline all'interno di una classe 316
12.8 I costruttori parametrizzati 317
12.9 I membri static di una classe 320
12.10 Quando vengono eseguiti i costruttori e i distruttori? 327
12.11 L'operatore di risoluzione del campo d'azione 329
12.12 La nidificazione delle classi 330
12.13 Le classi locali 330
12.14 Il passaggio di oggetti a funzioni 331
12.15 La restituzione di oggetti 334
12.16 L'assegnamento di oggetti 335

Capitolo 13 **Gli array, i puntatori, gli indirizzi e gli operatori di allocazione dinamica 337**
13.1 Gli array di oggetti 337
13.2 I puntatori a oggetti 341
13.3 Verifiche di tipo sui puntatori C++ 343
13.4 Il puntatore this 343



13.5 I puntatori a tipi derivati 345
13.6 I puntatori ai membri di una classe 348
13.7 Gli indirizzi 351
13.8 Questione di stile 359
13.9 Gli operatori di allocazione dinamica del C++ 360

Capitolo 14 **Overloading di funzioni, costruttori di copie e argomenti standard 371**
14.1 Overloading delle funzioni 371
14.2 Overloading delle funzioni costruttore 373
14.3 I costruttori di copie 377
14.4 Ricerca dell'indirizzo di una funzione modificata tramite overloading 381
14.5 L'anacronismo della parola riservata overload 383
14.6 Gli argomenti standard delle funzioni 383
14.7 Overloading di funzioni e ambiguità 390

Capitolo 15 **Overloading degli operatori 395**
15.1 Creazione di una funzione operator membro 396
15.2 Overloading di operatori tramite funzioni friend 403
15.3 Overloading di new e delete 409
15.4 Overloading di alcuni operatori particolari 418
15.5 Overloading dell'operatore virgola 425

Capitolo 16 **L'ereditarietà 429**
16.1 Controllo dell'accesso alla classe base 429
16.2 Ereditarietà dei membri protected 432
16.3 Ereditarietà da più classi base 436
16.4 Costruttori, distruttori ed ereditarietà 437
16.5 Accesso alle classi 445
16.6 Classi base virtuali 448

Capitolo 17 **Funzioni virtuali e polimorfismo 453**
17.1 Le funzioni virtuali 453
17.2 L'attributo virtual viene ereditato 458
17.3 Le funzioni virtuali sono gerarchiche 459
17.4 Le funzioni virtuali pure 462
17.5 Uso delle funzioni virtuali 464
17.6 Il binding anticipato e il binding ritardato 467

Capitolo 18 **I template 469**
18.1 Funzioni generiche 469
18.2 Uso delle funzioni generiche 478

18.3 Classi generiche 482
18.4 Le parole riservate typename ed export 493
18.5 La potenza dei template 494

Capitolo 19 **Gestione delle eccezioni** **497**
19.1 Principi di gestione delle eccezioni 497
19.2 Gestione delle eccezioni per classi derivate 506
19.3 Opzioni della gestione delle eccezioni 507
19.4 Le funzioni terminate() e unexpected() 513
19.5 La funzione uncaught_exception() 515
19.6 Le classi exception e bad_exception 515
19.7 Applicazioni della gestione delle eccezioni 516

Capitolo 20 **Il sistema di I/O C++: le basi** **519**
20.1 Operazioni di I/O C++ vecchie e nuove 520
20.2 Gli stream del C++ 520
20.3 Le classi per stream C++ 520
20.4 Operazioni di I/O formattato 522
20.5 Overloading di << e >> 535
20.6 Creazione di funzioni di manipolazione 544

Capitolo 21 **Operazioni di I/O su file in C++** **549**
21.1 L'header <fstream> e le classi per i file 549
21.2 L'apertura e la chiusura di un file 550
21.3 La lettura e la scrittura di un file di testo 553
21.4 Le operazioni di I/O binarie e non formattate 555
21.5 Altre forme della funzione get() 561
21.6 La funzione getline() 561
21.7 Rilevamento della fine del file 563
21.8 La funzione ignore() 565
21.9 Le funzioni peek() e putback() 566
21.10 La funzione flush() 566
21.11 L'accesso diretto ai file 566
21.12 Lo stato delle operazioni di I/O 571
21.13 Personalizzazione delle operazioni di I/O sui file 573

Capitolo 22 **L'identificazione run-time dei tipi e gli operatori cast** **577**
22.1 L'identificazione run-time dei tipi (RTTI) 577
22.2 Gli operatori di conversione cast 587
22.3 L'operatore dynamic_cast 587





Capitolo 23 **Namespace, funzioni di conversione e altri argomenti avanzati** **599**
23.1 I namespace 599
23.2 Lo spazio dei nomi std 609
23.3 Creazione di funzioni di conversione 611
23.4 Funzioni membro const e mutable 614
23.5 Funzioni membro volatile 617
23.6 Costruttori espliciti 617
23.7 Uso della parola riservata asm 619
23.8 Specifiche di linking 620
23.9 Operazioni di I/O su array 621
23.10 Uso di array dinamici 626
23.11 Uso di I/O binario con stream basati su array 628
23.12 Riepilogo delle differenze esistenti fra C e C++ 628

Capitolo 24 **Introduzione alla libreria STL** **631**
24.1 Introduzione all'uso della libreria STL 632
24.2 Le classi container 635
24.3 Funzionamento generale 636
24.4 I vettori 637
24.5 Le liste 647
24.6 Le mappe 658
24.7 Gli algoritmi 664
24.8 Uso degli oggetti funzione 674
24.9 La classe string 682
24.10 Commenti finali sulla libreria STL 694

**PARTE TERZA LA LIBRERIA DI FUNZIONI STANDARD** **695**

Capitolo 25 **Le funzioni di I/O basate sul C** **697**

Capitolo 26 **Le funzioni per stringhe e caratteri** **721**

Capitolo 27 **Le funzioni matematiche** **733**

Capitolo 28 **Le funzioni per le date, le ore e la localizzazione** **741**

Capitolo 29 **Le funzioni di allocazione dinamica della memoria** **749**

Capitolo 30 **Le funzioni di servizio** 753

Capitolo 31 **Le funzioni per caratteri estesi** 767
31.1 Le funzioni di classificazione per caratteri estesi 768
31.2 Le funzioni di I/O per caratteri estesi 770
31.3 Funzioni per stringhe di caratteri estesi 772
31.4 Funzioni di conversione per stringhe di caratteri estesi 773
31.5 Funzioni per array di caratteri estesi 773
31.6 Funzioni per la conversione di caratteri multibyte ed estesi 774

**PARTE QUARTA LA LIBRERIA DI CLASSI STANDARD DEL C++** 775

Capitolo 32 **Le classi di I/O del C++ standard** 777
32.1 Le classi di I/O 777
32.2 Gli header di I/O 780
32.3 I flag di formattazione e i manipolatori di I/O 780
32.4 I tipi del sistema di I/O del C++ standard 782
32.5 Overloading degli operatori < e > 784
32.6 Le funzioni di I/O di utilizzo generale 784

Capitolo 33 **Le classi container STL** 799

Capitolo 34 **Gli algoritmi STL** 823

Capitolo 35 **Iteratori, allocatori e oggetti funzione STL** 843
35.1 Gli iteratori 843
35.2 Gli oggetti funzione 854
35.3 Gli allocatori 860

Capitolo 36 **La classe string** 863
36.1 La classe basic_string 863
36.2 La classe char_traits 872

Capitolo 37 **Le classi per numeri** 875
37.1 La classe complex 875
37.2 La classe valarray 879
37.3 Gli algoritmi numerici 893



Capitolo 38 **Le classi per la gestione delle eccezioni** 899
38.1 Le eccezioni 899
38.2 La classe auto_ptr 901
38.3 La classe pair 903
38.4 La localizzazione 904
38.5 Altre classi interessanti 905

**PARTE QUINTA APPLICAZIONI C++** 907

Capitolo 39 **Integrazione delle nuove classi: una classe personalizzata per le stringhe** 909
39.1 La classe StrType 910
39.2 Le funzioni costruttore e distruttore 912
39.3 Operazioni di I/O di stringhe 913
39.4 Le funzioni di assegnamento 914
39.5 Il concatenamento 916
39.6 Sottrazione di sottostringhe 918
39.7 Gli operatori relazionali 920
39.8 Funzioni varie 921
39.9 L'intera classe StrType 922
39.10 Uso della classe StrType 931
39.11 Creazione e integrazione di nuovi tipi 933
39.12 Un esercizio 933

Capitolo 40 **Un analizzatore di espressioni realizzato con tecniche a oggetti** 935
40.1 Le espressioni 936
40.2 L'elaborazione delle espressioni: il problema 937
40.3 Analisi di un'espressione 938
40.4 La classe parser 939
40.5 Sezionamento di un'espressione 940
40.6 Un semplice parser di espressioni 943
40.7 Aggiunta delle variabili 949
40.8 Controllo della sintassi in un parser a discesa ricorsiva 959
40.9 Realizzazione di un parser generico 960
40.10 Alcune estensioni da provare 967

**Indice analitico** 969

# Prefazione

Questa è la seconda edizione della *Guida completa C++*. Negli anni trascorsi dalla realizzazione della prima edizione, il linguaggio C++ è stato sottoposto a numerose modifiche. Forse la modifica più importante è stata la standardizzazione del linguaggio. Nel novembre del 1997, il comitato ANSI/ISO, incaricato del compito di standardizzare il linguaggio C++, ha prodotto lo standard internazionale per il linguaggio. Questo evento ha concluso un processo lungo e talvolta controverso. Come membro del comitato ANSI/ISO per la standardizzazione del linguaggio C++, l'autore ha seguito tutti i progressi di questo processo di standardizzazione, partecipando a ogni dibattito e discussione. Alle battute finali del processo di sviluppo dello standard, vi era un serrato dialogo quotidiano via e-mail a livello mondiale in cui sono stati esaminati i pro e i contro di ogni singolo argomento per giungere a una soluzione finale. Anche se questo processo è stato più lungo e stressante di quanto chiunque potesse immaginare, i risultati sono decisamente all'altezza delle aspettative. Ora esiste uno standard per quello che senza ombra di dubbio è il linguaggio di programmazione più importante del mondo.

Durante la fase di standardizzazione sono state aggiunte nuove funzionalità al C++. Alcune sono relativamente piccole mentre altre, come l'introduzione della libreria STL (Standard Template Library) hanno implicazioni che influenzeranno il corso della programmazione negli anni a venire. Il risultato di queste aggiunte è stato una notevole estensione delle possibilità del linguaggio. Ad esempio, grazie all'aggiunta della libreria per l'elaborazione numerica, ora il C++ può essere utilizzato più comodamente nei programmi che svolgono una grande quantità di calcoli matematici. Naturalmente, le informazioni contenute in questa seconda edizione riflettono lo standard internazionale del linguaggio C++ così come è stato definito dal comitato ANSI/ISO, includendo tutte le nuove funzionalità introdotte.

## Le novità di questa seconda edizione

La seconda edizione della Guida completa C++ è stata notevolmente estesa rispetto all'edizione precedente. Questo si nota anche nella lunghezza del volume che è praticamente raddoppiata! Il motivo principale di ciò è che la seconda edizione analizza in modo più esteso la libreria delle funzioni standard e la libreria delle classi standard. Quando è stata realizzata la prima edizione, nessuna di queste due librerie era sufficientemente definita da consigliarne l'introduzione nel volume. Ora che la fase di standardizzazione del linguaggio C++ è terminata, è stato finalmente possibile aggiungere una descrizione di questi argomenti.

Oltre a queste aggiunte, la seconda edizione include anche una grande quantità di materiale nuovo un po' in tutto il volume. La maggior parte delle aggiunte è il risultato delle funzionalità introdotte nel linguaggio C++ fin dalla preparazione dell'edizione precedente. Sono stati particolarmente estesi i seguenti argomenti: la libreria STL (Standard Template Library), l'identificazione run-time dei tipi (RTTI), i nuovi operatori di conversione cast, le nuove funzionalità dei template, i namespace, il nuovo stile degli header e il nuovo sistema di I/O. Inoltre è stata sostanzialmente modificata la parte riguardante l'implementazione di new e delete e sono state discusse molte nuove parole riservate.

Onestamente, chi non abbia seguito attentamente l'evoluzione del linguaggio C++ negli ultimi anni, rimarrà sorpreso della sua crescita e delle funzionalità che gli sono state aggiunte; non è più lo stesso buon vecchio C++ che si usava solo qualche anno fa.

## Il contenuto della guida

Questo volume descrive in dettaglio tutti gli aspetti del linguaggio C++, a partire dal linguaggio che ne costituisce la base: il linguaggio C. Il volume è suddiviso in cinque parti:

- Le basi del C++: il linguaggio C
- Il linguaggio C++
- La libreria di funzioni standard
- La libreria di classi standard del C++
- Applicazioni C++

La Parte prima fornisce una trattazione completa del sottoinsieme del linguaggio C++, costituito dal linguaggio C. Come molti lettori sanno, il linguaggio C++ si basa sul C. È proprio il C che definisce le caratteristiche di base del C++, fin dai suoi elementi più semplici come i cicli for e le istruzioni if. Inoltre il C definisce la natura stessa del C++ come nel caso della struttura a blocchi dei programmi, dei puntatori e delle funzioni. Poiché molti lettori conoscono già il linguaggio C e hanno raggiunto un'elevata produttività in tale linguaggio, la scelta di discutere il sottoinsieme C in una parte a sé stante ha lo scopo di evitare al programmatore C di dover incontrare ripetutamente informazioni che conosce già. Dunque il programmatore esperto in C potrà semplicemente consultare quelle sezioni del volume che discutono le funzionalità specifiche del linguaggio C++.



La Parte seconda descrive in dettaglio le estensioni che il C++ ha apportato al C. Fra di esse vi sono le funzionalità a oggetti come le classi, i costruttori, i distruttori e i template. In pratica la Parte seconda descrive tutti quei costrutti specifici del linguaggio C++ ovvero assenti in C.

La Parte terza descrive la libreria delle funzioni standard e la Parte quarta esamina la libreria delle classi standard, inclusa la libreria STL (Standard Template Library). La Parte quinta mostra due esempi pratici di applicazione del linguaggio C++ e della programmazione a oggetti.

## Un libro per tutti i programmatori

Questa Guida completa C++ è dedicata a tutti i programmatori C++, indipendentemente dalla loro esperienza. Naturalmente il lettore deve essere quanto meno in grado di creare un semplice programma. Per tutti coloro che si trovano ad apprendere l'uso del linguaggio C++, questo volume potrà affiancare efficacemente qualsiasi Guida di apprendimento e costituire un'utile fonte di risposte. I programmatori C++ più esperti troveranno particolarmente utili le parti che si occupano delle funzionalità aggiunte in fase di standardizzazione.

## Programmazione in Windows

Il C++ è il linguaggio perfetto per Windows ed è completamente a suo agio nella programmazione in tale ambiente operativo. Ciononostante, nessuno dei programmi contenuti in questo volume è un programma per Windows. Si tratta in tutti i casi di programmi a console. Il motivo è facile da spiegare: i programmi per Windows sono, per loro stessa natura, estesi e complessi. La quantità di codice necessario per creare anche solo la semplice struttura di un programma per Windows occupa dalle 50 alle 70 righe. Per scrivere un programma per Windows che sia utile per illustrare le funzionalità del C++ sono necessarie centinaia di righe di codice. In poche parole, Windows non è l'ambiente più appropriato per descrivere le funzionalità di un linguaggio di programmazione. Naturalmente è possibile utilizzare un compilatore Windows per compilare i programmi contenuti in questo volume

poiché il compilatore creerà automaticamente una sessione a console nella quale eseguire il programma.

## Il codice sorgente nel Web

Il codice sorgente di tutti i programmi di questo volume è disponibile gratuitamente nel Web all'indirizzo `http://www.osborne.com`. Prelevando questo codice si eviterà di dover digitare manualmente gli esempi.

## Ulteriori studi

La Guida completa C++ è solo uno dei volumi scritti da Herbert Schildt. Ecco un elenco parziale dei volumi realizzati da questo autore, tutti editi da McGraw-Hill Libri Italia.

Chi volesse sapere qualcosa di più sul linguaggio C++, troverà particolarmente utili i seguenti volumi.

88 386 0351-0 H. Shildt, *Guida completa C++*
88 386 0332-4 H. Shildt, *Windows 95 Programmazione in C e C++*
88 386 3407-6 H. Shildt, *Guida al linguaggio C++*

Per quanto riguarda il linguaggio C, che sta alla base del C++, si consiglia la lettura dei seguenti volumi.

88 386 0340-5 H. Shildt, *Guida completa C 2ª ed.*
88 386 0175-5 H. Shildt, *Arte della programmazione in C*

Per sviluppare programmi per il Web è utile consultare:

88 386 0416-9 P. Naughton, H. Shildt, *Guida completa Java*

Infine, per la programmazione per Windows, si rimanda a:

88 386 0455-X H. Shildt, *Programmazione Windows NT4*
88 386 0397-9 H. Shildt, *MFC Programmazione Windows*

Parte prima

# LE BASI DEL C++: IL LINGUAGGIO C

In questo volume la descrizione del linguaggio C++ viene suddivisa in due parti. La Parte prima si occupa delle funzionalità che il C++ ha in comune con il suo progenitore, il C. Infatti il linguaggio C rappresenta un *sottoinsieme* del C++. La Parte seconda descrive le funzionalità specifiche del C++. Insieme, queste due parti, descrivono dunque il linguaggio C++. Come forse molti sanno, il C++ si basa sul linguaggio C. In pratica si può dire che il C++ include l'intero linguaggio C e (tranne lievi eccezioni), tutti i programmi C sono anche programmi C++. Quando fu inventato il linguaggio C++, venne impiegato come base il linguaggio C al quale vennero aggiunte molte nuove funzionalità ed estensioni con lo scopo di garantire il supporto della programmazione orientata agli oggetti (OOP). Questo non significa che gli aspetti che il C++ ha in comune con il C siano stati abbandonati ma, a maggior ragione, il C standard ANSI/ISO costituisce il *documento di partenza* per lo Standard Internazionale per il C++. Pertanto, la conoscenza del linguaggio C++ implica una conoscenza del linguaggio C.

In un volume come questa Guida completa, il fatto di suddividere il linguaggio C++ in due parti (le basi C e le funzionalità specifiche del C++) consente di ottenere tre vantaggi.

1. Si delinea con chiarezza la linea di demarcazione esistente fra C e C++.
2. I lettori che già conoscono il linguaggio C potranno facilmente trovare informazioni specifiche sul linguaggio C++.
3. Viene fornito un modo per discutere quelle funzionalità del linguaggio C++ che sono più legate al sottoinsieme costituito dal linguaggio C.

Comprenderne la linea di divisione esistente fra C e C++ è importante poiché si tratta in entrambi i casi di linguaggi molto utilizzati e dunque è molto probabile che prima o poi venga richiesto di scrivere o eseguire la manutenzione di codice C e C++. Quando si lavora in C si deve sapere esattamente dove finisce il C e dove inizia il C++. Molti programmatori C++ si troveranno talvolta a scrivere codice

che deve rientrare nei limiti stabiliti dal "sottoinsieme C". Questo accade particolarmente nel campo della programmazione di sistemi e della manutenzione di applicazioni preesistenti. Conoscere la differenza fra C e C++ è parte integrante della propria esperienza di programmatore C++ professionale.

Una buona comprensione del linguaggio C è insostituibile anche quando si deve convertire del codice C in C++. Per svolgere l'operazione in modo professionale, è necessario conoscere in modo approfondito anche il linguaggio C. Ad esempio, senza una conoscenza approfondita del sistema di I/O del C è impossibile convertire in modo efficiente dal C al C++ un programma che esegua notevoli operazioni di I/O.

Molti lettori conoscono già il linguaggio C. Il fatto di discutere le funzionalità C in apposite sezioni può aiutare un programmatore C a ricercare con facilità e rapidità le informazioni riguardanti il C senza perdere tempo a leggere informazioni già note. Naturalmente in questa Parte prima sono state elencate anche alcune differenze marginali fra il C e il C++. Inoltre il fatto di separare le basi C dalle funzionalità più avanzate e orientate agli oggetti del linguaggio C++ consentirà di concentrarsi sulle funzionalità avanzate perché tutti gli elementi di base saranno stati trattati in precedenza.

Anche se il linguaggio C++ contiene l'intero linguaggio C, quando si scrivono programmi C++ non vengono utilizzate molte delle funzionalità fornite dal linguaggio C. Ad esempio, il sistema di I/O del C è disponibile anche in C++ ma quest'ultimo linguaggio definisce nuove versioni a oggetti. Un altro esempio è rappresentato dal preprocessore. Il preprocessore è molto importante in C; molto meno in C++. Il fatto di discutere le funzionalità C nella Parte prima evita dunque di congestionare di dettagli la parte rimanente di questo volume.

**SUGGERIMENTO** *Il sottoinsieme C descritto nella Parte prima costituisce la base del linguaggio C++ e il nucleo fondamentale su cui sono costruite le funzionalità a oggetti del linguaggio C++. Tutte le funzionalità descritte in questa Parte prima fanno parte del linguaggio C++ e dunque sono disponibili all'uso.*

**NOTA** *La Parte prima di questo volume è stata adattata da La guida completa C (McGraw-Hill Libri Italia - 1995 - ISBN 0340-5). Chi fosse particolarmente interessato al linguaggio C troverà tale volume molto interessante.*

Capitolo 1

# Una panoramica sul linguaggio C


1.1 Le origini del linguaggio C
1.2 Il C è un linguaggio di medio livello
1.3 Il C è un linguaggio strutturato
1.4 Il C è un linguaggio per programmatori
1.5 L'aspetto di un programma C
1.6 La libreria e il linker
1.7 Compilazione separata
1.8 Le estensioni di file: .c e .cpp


Conoscere il C++ significa conoscere le forze che hanno portato alla sua creazione, le idee che gli hanno dato il suo aspetto e i "caratteri" che ha ereditato. Pertanto la storia del C++ non può che partire dal C. Questo capitolo presenta una panoramica del linguaggio di programmazione C, le sue origini, il suo utilizzo e la sua filosofia. Poiché il C++ si basa sul C, questo capitolo presenta anche un'importante prospettiva storica sulle radici del C++. Molti degli elementi che hanno reso il linguaggio C++ quello che è hanno la loro origine nel linguaggio C.

## 1.1 Le origini del linguaggio C

Il C fu inventato e implementato per la prima volta da Dennis Ritchie su un sistema DEC PDP-11 che impiegava il sistema operativo Unix. Il C è il risultato di un processo di sviluppo che è partito da un linguaggio chiamato BCPL. Il BCPL, sviluppato da Martin Richards, influenzò un linguaggio chiamato B, inventato da Ken Thompson. Il B portò allo sviluppo del C negli anni '70.

Per molti anni, lo standard de facto del C fu la versione fornita con il sistema operativo Unix versione 5. Il linguaggio fu descritto per la prima volta nel volume *The C Programming Language* di Brian Kernighan e Dennis Ritchie. Nell'estate del 1983 venne nominato un comitato con lo scopo di creare uno standard ANSI (American National Standards Institute) che definisse il linguaggio C una volta

per tutte. Il processo di standardizzazione richiese sei anni (molto più del previsto). Lo standard ANSI C fu infine adottato nel dicembre del 1989 e le prime copie si resero disponibili all'inizio del 1990. Lo standard venne anche adottato dall'ISO (International Standards Organization) ed ora è chiamato Standard C ANSI/ISO. Per semplicità si userà semplicemente il termine Standard C. Oggi, tutti i principali compilatori C/C++ seguono lo Standard C. Inoltre, lo Standard C è anche alla base dello Standard C++.

## 1.2 Il C è un linguaggio di medio livello

Il C è considerato da molti un linguaggio di medio livello. Questo non significa che il C sia meno potente, più difficile da utilizzare o meno evoluto rispetto a un linguaggio ad alto livello come il BASIC o il Pascal, né che il C abbia la natura complicata del linguaggio Assembler (con tutti i problemi derivanti). Piuttosto, significa che il C è un linguaggio che riunisce i migliori elementi dei linguaggi ad alto livello con le possibilità di controllo e la flessibilità del linguaggio Assembler. La Tabella 1.1 mostra la posizione del C nello spettro dei linguaggi per computer.

Essendo un linguaggio di medio livello, il C consente la manipolazione di bit, byte e indirizzi, gli elementi su cui si basa il funzionamento di un computer.

Nonostante ciò, il codice C è anche molto trasportabile. Con trasportabilità si intende la facilità di adattare su un sistema un software scritto per un altro computer o sistema operativo. Ad esempio, se è possibile convertire con facilità un programma scritto per il DOS in modo che possa essere utilizzato sotto Windows, significa che tale programma è trasportabile.

Tutti i linguaggi di programmazione di alto livello prevedono il concetto di tipo di dati. Un tipo di dati definisce una gamma di valori che una variabile è in grado di memorizzare insieme a un gruppo di operazioni che possono essere eseguite su tale variabile.

**Tabella 1.1** Come si posiziona il C nel mondo dei linguaggi.

| | |
|---|---|
| Alto livello | Ada<br>Modula-2<br>Pascal<br>COBOL<br>FORTRAN<br>BASIC |
| Medio livello | Java<br>C++<br>C<br>FORTH |
| Basso livello | Macro assembler<br>Assembler |



I tipi di dati più comuni sono gli interi, i caratteri e i numeri reali. Anche se il C prevede cinque tipi di dati di base, non si tratta di un linguaggio fortemente tipizzato, come il Pascal o l'Ada. Il C consente quasi ogni conversione di tipo. Ad esempio, è possibile utilizzare liberamente in un'espressione i tipi carattere e intero.

A differenza dei linguaggi ad alto livello, il C non esegue verifiche di errore al momento dell'esecuzione (verifiche run-time).

Ad esempio, nulla vieta di andare per errore a leggere oltre i limiti di un array. Questo tipo di controlli deve pertanto essere previsto dal programmatore. Analogamente, il C non richiede una compatibilità stretta di tipo fra un parametro e un argomento.

Come il lettore può pensare sulla base di precedenti esperienze di programmazione, un linguaggio di alto livello richiede normalmente che il tipo di un argomento sia (in forma più o meno forte) lo stesso del tipo del parametro che riceverà l'argomento. Questo non è il caso del C. In C un argomento può essere di qualsiasi tipo che possa essere ragionevolmente convertito nel tipo del parametro. La conversione di tipo viene eseguita automaticamente dal C.

La peculiarità del C consiste nella possibilità di manipolare direttamente i bit, i byte, le word e i puntatori. Questo lo rende adatto alla programmazione di software di sistema, in cui queste operazioni sono molto comuni.

Un altro aspetto importante del C è la presenza di solo 32 parole chiave (27 derivanti dallo standard "de facto" Kernighan e Ritchie e 5 aggiunte dal comitato di standardizzazione ANSI), che sono i comandi che formano il linguaggio C. Normalmente i linguaggi di alto livello hanno molte più parole chiave. Come confronto, si può ricordare che la maggior parte delle versioni di BASIC conta più di 100 parole chiave!

## 1.3 Il C è un linguaggio strutturato

In altre esperienze di programmazione, il lettore può aver sentito parlare di strutturazione a blocchi applicata a un linguaggio per computer. Anche se il termine non si applica in modo stretto al C, si parla normalmente del C come di un linguaggio strutturato. In effetti il C ha molte analogie con altri linguaggi strutturati, come l'ALGOL, il Pascal e il Modula-2.

**NOTA** *Il motivo per cui il C (e il C++) non è, tecnicamente, un linguaggio strutturato a blocchi, è il seguente: i linguaggi strutturati a blocchi consentono la dichiarazione di procedure o funzioni all'interno di altre procedure o funzioni. Tuttavia, poiché in C questo non è consentito non può, formalmente, essere chiamato linguaggio strutturato a blocchi.*

La caratteristica che distingue un linguaggio strutturato è l'isolabilità del codice e dei dati. Questa è la capacità del linguaggio di suddividere e nascondere dal resto del programma tutte le informazioni e le istruzioni necessarie per eseguire una determinata operazione. Un modo per ottenere ciò consiste nell'uso di subroutine che impiegano variabili locali (temporanee). Utilizzando variabili locali, è possibile scrivere subroutine realizzate in modo tale che gli eventi che avvengono al loro interno non provochino effetti collaterali in altre parti del programma. Questa possibilità semplifica la condivisione di sezioni del codice fra più programmi C. Se si sviluppa una funzione ben isolata, tutto quello che si deve sapere sulla funzione è cosa essa faccia e non come lo faccia. Occorre ricordarsi che un uso eccessivo di variabili globali (variabili note all'intero programma) può dare origine a bug (errori) provocati da effetti collaterali. Chiunque abbia programmato in BASIC conosce bene questo problema.

**NOTA** *Il concetto di isolamento è notevolmente impiegato in C++. In particolare, in C++ è possibile controllare esattamente quali parti del programma debbano avere accesso a quali altre parti.*

Un linguaggio strutturato dà molte possibilità. In particolare accetta direttamente numerosi costrutti di ciclo, come while, do-while e for. In un linguaggio strutturato, l'uso del goto è proibito o sconsigliato e non costituisce di certo la forma più comune di controllo del programma (come nel caso del BASIC standard e del FORTRAN tradizionale). Un linguaggio strutturato consente di inserire le istruzioni in qualunque punto di una riga e non prevede un forte concetto di campo (come alcune vecchie implementazioni del FORTRAN).

Ecco alcuni esempi di linguaggi strutturati e non strutturati:

| NON STRUTTURATI | STRUTTURATI |
|---|---|
| FORTRAN | Pascal |
| BASIC | Ada |
| COBOL | Java |
| | C++ |
| | C |
| | Modula-2 |

I linguaggi strutturati sono in genere più moderni. Infatti, una caratteristica tipica dei vecchi linguaggi di programmazione è l'assenza di strutture. Oggi, pochi programmatori penserebbero di utilizzare un linguaggio non strutturato per realizzare programmi professionali.



**NOTA** *Le nuove versioni di molti vecchi linguaggi di programmazione hanno tentato di introdurre elementi di strutturazione. Un esempio è rappresentato dal BASIC. Tuttavia le caratteristiche di base di tali linguaggi non possono essere completamente dissimulate poiché si tratta di linguaggi sviluppati fin dall'inizio senza avere tenere in considerazione le funzionalità della programmazione strutturata.*

Il principale componente strutturale del C è la *funzione*: una subroutine a sé stante. In C, le funzioni sono i mattoni su cui si basa tutta l'attività di un programma. Esse consentono di definire e codificare in modo distinto le varie operazioni svolte da un programma e quindi consentono di creare programmi modulari. Dopo aver creato una funzione, è possibile utilizzarla in varie situazioni senza temere di veder sorgere effetti collaterali in altre parti del programma. La possibilità di creare funzioni a sé stanti è estremamente critica specialmente nei grandi progetti in cui il codice realizzato da un programmatore non deve interferire accidentalmente con quello prodotto da un altro programmatore.

Un altro modo per strutturare e isolare il codice C prevede l'uso di blocchi di codice. Un *blocco di codice* è formato da un gruppo di istruzioni connesse logicamente che viene considerato come una singola unità. In C, è possibile creare un blocco di codice inserendo una sequenza di istruzioni fra una coppia di parentesi graffe. In questo esempio,

```
if (x  < 10)  {
   printf("troppo basso, riprova\n");
   scanf("%d", &x);
}
```

se x è minore di 10 vengono eseguite entrambe le istruzioni che si trovano dopo l'if e fra parentesi graffe. Queste due istruzioni, insieme alle parentesi graffe, rappresentano un blocco di codice. Si tratta di unità logiche: non è possibile eseguire un'istruzione senza eseguire anche l'altra. I blocchi di codice consentono di implementare molti algoritmi con chiarezza, eleganza ed efficienza. Inoltre, aiutano il programmatore a concettualizzare meglio la vera natura dell'algoritmo implementato.

## 1.4 Il C è un linguaggio per programmatori

Sorprendentemente, non tutti i linguaggi di programmazione sono comodi per un programmatore. Basta considerare i classici esempi di linguaggi per non programmatori, come il COBOL e il BASIC. Il COBOL non è stato progettato per migliorare il lavoro dei programmatori, né per aumentare l'affidabilità del codice pro-

dotto e neppure per incrementare la velocità di realizzazione del codice. Piuttosto, il COBOL è stato progettato, in parte, per consentire ai non programmatori di leggere e presumibilmente (anche se difficilmente) comprendere il programma. Il BASIC fu creato essenzialmente per consentire ai non programmatori di programmare un computer per risolvere problemi relativamente semplici.

Al contrario, il C è stato creato, influenzato e testato sul campo da programmatori professionisti. Il risultato finale è che il C dà al programmatore quello che il programmatore desidera: poche restrizioni, pochi motivi di critiche, strutture a blocchi, funzioni isolabili e un gruppo compatto di parole chiave. Utilizzando il C, si raggiunge quasi l'efficienza del codice assembler ma utilizzando una struttura simile a quella dell'ALGOL o del Modula-2. Non è quindi una sorpresa che il C e il C++ siano con facilità diventati i linguaggi più popolari fra i migliori programmatori professionisti.

Il fatto che sia possibile utilizzare il C al posto del linguaggio Assembler è uno dei fattori principali della sua popolarità fra i programmatori. Il linguaggio Assembler utilizza una rappresentazione simbolica del codice binario effettivo che il computer esegue direttamente. Ogni operazione del linguaggio Assembler corrisponde a una singola operazione che il computer deve eseguire. Anche se il linguaggio Assembler fornisce ai programmatori tutto il potenziale per eseguire questi compiti con la massima flessibilità ed efficenza, si tratta di un linguaggio notoriamente difficile per quanto riguarda lo sviluppo e il debugging di un programma. Inoltre, poiché il linguaggio Assembler non è strutturato, il programma finale tende a essere molto ingarbugliato: una complessa sequenza di salti, chiamate e indici. Questa mancanza di strutturazione rende i programmi in linguaggio Assembler difficili da leggere, migliorare e mantenere. Ma c'è di peggio: le routine in linguaggio Assembler non sono trasportabili fra macchine dotate di unità di elaborazione (CPU) diverse.

Inizialmente, il C fu utilizzato per la programmazione di software di sistema. Un *programma di sistema* è un programma che fa parte del sistema operativo del computer o dei suoi programmi di supporto. Ad esempio, sono considerati programmi di sistema i seguenti software:

- sistemi operativi
- interpreti
- editor
- compilatori
- programmi di servizio per la gestione di file
- ottimizzatori prestazionali
- programmi per la gestione di eventi in tempo reale

Mano a mano che crebbe la popolarità del C, molti programmatori iniziarono a usarlo per realizzare tutti i loro programmi, sfruttandone la trasportabilità e l'efficienza. quando venne creato, il linguaggio C rappresentava un notevole passo in avanti nel campo dei linguaggi di programmazione. Naturalmente anche il linguaggio C++ ha tenuto fede a questa tradizione.

Con la nascita del linguaggio C++, molti pensarono che l'esperienza del C come linguaggio a sé stante si sarebbe conclusa. Tale previsione si è rivelata errata. Innanzitutto non tutti i programmi richiedono l'applicazione delle tecniche di programmazione a oggetti fornite dal C++. Ad esempio i programmi di sistema vengono in genere sviluppati in C. In secondo luogo, in molte situazioni viene ancora utilizzato codice C e dunque vi è un notevole lavoro di estensione e manutenzione di questi programmi. Anche se il C è ricordato soprattutto per il fatto di aver dato origine al C++, rimane pur sempre un linguaggio molto potente che verrà ampiamente utilizzato negli anni a venire.

## 1.5 L'aspetto di un programma C

La Tabella 1.2 elenca le 32 parole chiave che insieme alla sintassi formale del C, formano il linguaggio di programmazione C. Di queste, 27 sono state definite dalla versione originale del C. Le altre cinque sono state aggiunte dal comitato ANSI e sono: enum, const, signed, void e volatile. Naturalmente tutte queste parole riservate fanno parte anche del linguaggio C++.

Inoltre, molti compilatori hanno aggiunto numerose parole chiave che consentono di sfruttare al meglio un determinato ambiente operativo. Ad esempio, molti compilatori comprendono parole chiave che consentono di gestire l'organizzazione della memoria tipica della famiglia di microprocessori 8086, di programmare con più linguaggi contemporaneamente e di accedere agli interrupt. Ecco un elenco delle parole chiave estese più comunemente utilizzate:

| | | | |
|---|---|---|---|
| asm | _cs | _ds | _es |
| _ss | cdecl | far | huge |
| interrupt | near | pascal | |

Il compilatore può inoltre prevedere altre estensioni che aiutano a sfruttare tutti i vantaggi di un determinato ambiente operativo.

Tutte le parole chiave del linguaggio C (e C++) devono essere scritte in lettere minuscole. Inoltre le lettere maiuscole e le lettere minuscole sono considerate differenti: else è una parola chiave mentre ELSE non lo è. In un programma non è possibile utilizzare una parola chiave per altri scopi (ovvero come una variabile o un nome di funzione).

Tutti i programmi C sono formati da una o più funzioni. L'unica funzione che deve essere obbligatoriamente presente si chiama main(). la prima funzione che viene richiamata quando inizia l'esecuzione del programma. In un programma C ben realizzato, main() contiene uno schema dell'intero funzionamento del pro-

gramma. Questo schema è formato da una serie di chiamate a funzioni. Anche se main() non è una parola chiave, deve essere trattata come se lo fosse. Ad esempio, non si può cercare di usare main() come nome di variabile poiché con ogni probabilità si confonderebbe il compilatore.

L'aspetto generale di un programma C è illustrato nella Figura 1.1, in cui le indicazioni da f1() a fN() rappresentano le funzioni definite dall'utente.

## 1.6 La libreria e il linker

In senso tecnico, è possibile creare un utile e funzionale programma C o C++ costituito unicamente dalle istruzioni create dal programmatore. Tuttavia, questo è molto raro in quanto né il C né il C++ forniscono metodi per eseguire operazioni di input e output (I/O), per svolgere operazioni matematiche complesse o per manipolare i caratteri. Il risultato è che molti programmi includono chiamate alle varie funzioni contenute nella *libreria standard.*

Tutti i compilatori C++ sono dotati di una libreria di funzioni standard che eseguono le operazioni più comuni. Lo Standard C++ specifica un gruppo minimo di funzioni che devono essere supportate da tutti i compilatori. Tuttavia, un determinato compilatore può contenere molte altre funzioni. Ad esempio, la libreria standard non definisce alcuna funzione grafica ma il compilatore ne includerà probabilmente più di una.

La libreria standard C++ può essere suddivisa in due parti: la libreria delle funzioni standard e la libreria delle classi. La libreria delle funzioni standard è ereditata dal linguaggio C. Il linguaggio C++ supporta l'intera libreria di funzioni definita dallo Standard C. Pertanto nei programmi C++ sono disponibili tutte le funzioni standard C.

**Tabella 1.2_** L'elenco delle parole chiave del C ANSI.

| | | | |
|---|---|---|---|
| auto | double | int | struct |
| break | else | long | switch |
| case | enum | register | typedef |
| char | extern | return | union |
| const | float | short | unsigned |
| continue | for | signed | void |
| default | goto | sizeof | volatile |
| do | if | static | while |



```
Dichiarazioni globali
tipo restituito main(elenco parametri)
{
      sequenza istruzioni
}
tipo restituito f1(elenco parametri)
{
      sequenza istruzioni
}
tipo restituito f2(elenco parametri)
{
      sequenza istruzioni
}
.
.
.
tipo restituito fN(elenco parametri)
{
      sequenza istruzioni
}
```

**Figura 1.1** La forma generale di un programma C

Oltre alla libreria di funzioni standard, il linguaggio C++ definisce anche una propria libreria di classi. La libreria di classi offre delle routine a oggetti utilizzabili dai programmi. Inoltre definisce la libreria STL (Standard Template Library) che offre soluzioni pronte all'uso per un'ampia varietà di problemi di programmazione. La libreria di classi e la libreria STL verranno discusse più avanti in questo volume. Nella Parte prima verrà utilizzata solo la libreria delle funzioni standard poiché è l'unica definita anche in C.

Gli implementatori del compilatore C hanno già scritto la maggior parte delle funzioni di utilizzo generale che il programmatore si troverà a utilizzare. Quando si richiama una funzione che non fa parte del programma, il compilatore C prende nota del suo nome. In seguito, il linker riunisce al codice scritto dal programmatore il codice oggetto che si trova nella libreria standard. Questo processo è chiamato *linking*. Alcuni compilatori C sono dotati di un proprio linker mentre altri utilizzano il linker standard fornito insieme al sistema operativo.

Le funzioni contenute nella libreria sono in formato *rilocabile*. Questo significa che gli indirizzi di memoria delle varie istruzioni in codice macchina non

devono essere definiti in modo assoluto: devono essere conservate solo le informazioni di offset (scostamento). Quando il programma esegue il linking con le funzioni contenute nella libreria standard, questi offset di memoria consentono di creare gli indirizzi che verranno effettivamente utilizzati. Vi sono molti manuali tecnici che descrivono questo processo in dettaglio. In questa fase, non vi è però alcun bisogno di conoscere in profondità l'effettivo processo di rilocazione per iniziare a programmare in C o in C++.

Molte delle funzioni di cui il programmatore avrà bisogno nella scrittura dei programmi sono già contenute nella libreria standard. Queste funzioni possono essere considerate i mattoni che il programmatore può unire per formare un programma. Se il programmatore si troverà a scrivere una funzione che pensa di utilizzare più volte, potrà inserire anch'essa nella libreria. Alcuni compilatori consentono infatti di inserire nuove funzioni nella libreria standard; altri consentono invece di creare nuove librerie aggiuntive. In ogni caso, il codice di queste funzioni potrà essere utilizzato più volte.

## 1.7 Compilazione separata

La maggior parte dei programmi C più piccoli si trova contenuta in un unico file sorgente. Tuttavia, mano a mano che cresce la lunghezza del programma, cresce anche il tempo richiesto dalla compilazione. Pertanto, il C/C++ consente di suddividere un programma su più file che possono essere compilati separatamente. Dopo aver compilato tutti i file, ne viene eseguito il linking, includendo anche tutte le routine della libreria, per formare un unico file di codice oggetto completo. Grazie alla compilazione separata, è possibile fare modifiche a un file sorgente del programma senza dover ricompilare l'intero programma. Su progetti non proprio banali, questo consente di risparmiare una grande quantità di tempo. Per informazioni sulle strategie per la compilazione separata, consultare la documentazione del compilatore C/C++.

## 1.8 Le estensioni di file: .c e .cpp

I programmi della Parte prima di questo volume sono, naturalmente, programmi C++ e possono essere compilati utilizzando un moderno compilatore C++. Tuttavia si tratta anche di programmi C compilabili con un compilatore C. Pertanto se si devono scrivere programmi C, quelli illustrati nella Parte prima possono essere considerati buoni esempi. Tradizionalmente, i programmi C usano l'estensione .c e i programmi C++ usano l'estensione .cpp. Un compilatore C++ utilizza tale estensione per determinare quale tipo di programma sta compilando. Questa distinzione è dunque importante poiché il compilatore suppone che ogni programma che usa l'estensione .c sia un programma C e che ogni programma che usa l'estensione .cpp sia un programma C++. Se non viene indicato esplicitamente, per i programmi della Parte prima possono essere utilizzate entrambe le estensioni. Al contrario i programmi contenuti nelle parti successive devono avere l'estensione .cpp.



Un'ultima annotazione: anche se il C è un sottoinsieme del C++, i due linguaggi presentano alcune lievi differenze e in alcuni casi è necessario compilare un programma C come un programma C (usando l'estensione .c). Nel volume tutte queste situazioni verranno indicate in modo esplicito.

Capitolo 2

# Le espressioni


2.1 **I cinque tipi di dati principali**
2.2 **Modificare i tipi principali**
2.3 **Nomi degli identificatori**
2.4 **Le variabili**
2.5 **I modificatori di accesso**
2.6 **Specificatori di classe di memorizzazione**
2.7 **Inizializzazione delle variabili**
2.8 **Le costanti**
2.9 **Gli operatori**
2.10 **Le espressioni**


Questo capitolo esamina l'elemento di base del linguaggio C e del C++: l'*espressione*. Le espressioni C/C++ sono sostanzialmente più generali e più potenti rispetto a quelle della maggior parte degli altri linguaggi di programmazione. Esse sono costituite dagli elementi "atomici" del C: i *dati* e gli *operatori*. I dati possono essere rappresentati da variabili o costanti. Come la maggior parte dei linguaggi di programmazione il C/C++, consente di utilizzare vari tipi di dati ed è dotato di un'ampia varietà di operatori.

## 2.1 I cinque tipi di dati principali

In C vi sono cinque tipi di dati principali: caratteri, numeri interi, numeri in virgola mobile, numeri in virgola mobile doppi e non-valori (rispettivamente char, int, float, double e void). Tutti gli altri tipi di dati utilizzati in C si basano su questi cinque tipi. Le dimensioni e i valori memorizzabili in questi tipi di dati possono variare a seconda del microprocessore e del compilatore impiegati. Tuttavia, nella maggior parte dei casi, un carattere è contenuto in un byte. Le dimensioni di un intero equivalgono in genere alle dimensioni di una word nell'ambiente di esecuzione del programma. Per la maggior parte degli ambienti a 16 bit, come il DOS o Windows 3.1, un intero occupa 16 bit. Negli ambienti a 32 bit, come Windows NT, in genere un intero occupa 32 bit. In ogni caso è sconsigliabile basarsi su

queste semplici indicazioni, specialmente se si vuole fare in modo che i propri programmi possano essere trasportabili da un ambiente a un altro. È importante comprendere che sia gli standard C e C++ indicano solamente una gamma di valori minimi che un determinato tipo di dati deve contenere e non le dimensioni in byte.

**NOTA** *Il C++ aggiunge ai cinque tipi di dati principali del C i tipi bool e wchar_t che verranno discussi nella Parte seconda di questa guida.*

Il formato esatto dei valori in virgola mobile dipende dall'implementazione. Gli interi corrispondono generalmente alle dimensioni naturali di una word nel computer. I valori di tipo char sono normalmente utilizzati per contenere i valori definiti dal set di caratteri ASCII. I valori che non rientrano in questo intervallo possono essere gestiti in modo diverso dalle varie implementazioni di C.

Gli intervalli di valori utilizzabili nei tipi float e double dipendono dal metodo utilizzato per rappresentare i numeri in virgola mobile. Qualunque sia il metodo, questo intervallo è piuttosto esteso. Lo Standard C specifica che l'intervallo minimo per un valore in virgola mobile vada da 1E-37 a 1E+37. Il numero minimo di cifre di precisione per ognuno dei tipi in virgola mobile si trova elencato nella Tabella 2.1.

**NOTA** *Lo Standard C++ non specifica valori di estensione o di intervallo minimi per i tipi predefiniti ma stabilisce solo alcuni requisiti minimi. Ad esempio, lo Standard C++ dice che un* `int` *deve avere dimensioni naturali rispetto all'architettura dell'ambiente di esecuzione. In ogni caso l'intervallo di valori deve essere uguale o maggiore rispetto a quanto indicato dallo Standard C. Ogni compilatore C++ specifica l'estensione e l'intervallo di valori consentiti da un tipo nel file header <climits>.*

Il tipo void dichiara esplicitamente che una funzione non restituisce alcun tipo di valore o consente di creare puntatori generici. L'uso di questi oggetti verrà discusso nei prossimi Capitoli.

## 2.2 Modificare i tipi principali

Se si esclude il tipo void, i tipi di dati principali possono essere dotati di vari modificatori. Il *modificatore*, che riceve la dichiarazione di tipo, altera il significato del tipo base per adattarlo con più precisione alle varie situazioni. L'elenco dei modificatori è il seguente:

signed
unsigned
long
short



È possibile applicare i modificatori signed, short, long e unsigned al tipo intero e i modificatori signed e unsigned al tipo carattere. Inoltre, il modificatore long può essere applicato anche al tipo double.

La Tabella 2.1 mostra tutte le combinazioni di tipi di dati validi, indicando le dimensioni approssimative in bit e l'intervallo minimo richiesto (questi valori sono validi anche per le implementazioni di C++). Si ricordi che la tabella mostra solo l'intervallo minimo che questi tipi devono avere secondo quanto specificato dallo Standard C/C++ e non un intervallo tipico. Ad esempio, nel caso di computer che impiegano l'aritmetica con complemento a 2 (praticamente tutti i computer), un intero avrà un intervallo compreso almeno fra 32767 e -32768.

L'uso del modificatore signed sugli interi è consentito ma è ridondante in quanto la dichiarazione standard dell'intero prevede un numero dotato di segno. L'uso più importante del modificatore signed è in congiunzione con il tipo char in implementazioni in cui char sia considerato senza segno.

**Tabella 2.1** Tutti i tipi di dati definiti dal C ANSI.

| TIPO | DIMENSIONI | INTERVALLO MINIMO APPROSSIMATIVO IN BIT |
|---|---|---|
| char | 8 | da -127 a 127 |
| unsigned char | 8 | da 0 a 255 |
| signed char | 8 | da -127 a 127 |
| int | 16 | da -32767 a 32767 |
| unsigned int | 16 | da 0 a 65535 |
| signed int | 16 | come `int` |
| short int | 16 | come `int` |
| unsigned short int | 16 | da 0 a 65535 |
| signed short int | 16 | come `short int` |
| long int | 32 | da -2.147.483.647 a 2.147.483.647 |
| signed long int | 32 | come `long int` |
| unsigned long int | 32 | da 0 a 4.294.967.295 |
| float | 32 | sei cifre di precisione |
| double | 64 | dieci cifre di precisione |
| long double | 80 | dieci cifre di precisione |

La differenza fra interi con segno (signed) e senza segno (unsigned) è il modo in cui viene interpretato il bit alto dell'intero. Se si specifica un intero signed, il compilatore C genera codice che assume che il bit alto di un intero venga utilizzato come *bit di segno*. Se questo bit è 0, il numero è positivo mentre se questo bit è 1, il numero è negativo.

In generale, i numeri negativi sono rappresentati utilizzando un approccio di tipo *complemento a due*, che inverte tutti i bit del numero (tranne il bit del segno), aggiunge 1 al numero e imposta il bit del segno a 1.

Gli interi con segno sono importanti per un gran numero di algoritmi ma hanno un'ampiezza assoluta pari alla metà dei loro fratelli unsigned. Ad esempio, questa è la rappresentazione del numero 32767:

```
0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
```

Se il bit alto fosse impostato a 1, il numero sarebbe stato interpretato come -1. Se invece si dichiara questo numero unsigned int, impostando il bit alto a 1, il numero verrà interpretato come 65535.

## 2.3 Nomi degli identificatori

In C e in C++ i nomi delle variabili, delle funzioni, delle etichette e degli altri oggetti definiti dall'utente sono chiamati *identificatori*. Questi identificatori possono essere costituiti da un minimo di un carattere. Il primo carattere deve essere una lettera o un carattere di sottolineatura e i caratteri successivi possono essere lettere, cifre o caratteri di sottolineatura. Ecco alcuni esempi di nomi di identificatori corretti ed errati:

| Corretto | Errato |
|---|---|
| Numero | 1numero |
| test23 | ciao! |
| valori_bilancio | valori...bilancio |

In C gli identificatori possano essere di qualsiasi lunghezza. Tuttavia, non tutti i caratteri saranno necessariamente significativi. Se l'identificatore verrà impiegato in un processo di linking esterno, saranno significativi almeno i primi sei caratteri. Questi identificatori, chiamati *nomi esterni*, includono i nomi di funzioni e le variabili globali condivise fra i vari file. Se l'identificatore non è utilizzato in un processo di linking esterno, saranno significativi almeno i primi 31 caratteri. Questo tipo di identificatore è chiamato *nome interno;* ad esempio sono nomi interni i nomi delle variabili locali. In C++ invece non vi è alcun limite alla



lunghezza di un identificatore e sono significativi almeno 1024 caratteri. Questa differenza è importante se si deve convertire un programma dal C al C++.

In un identificatore le lettere maiuscole e minuscole sono considerate diverse. Pertanto, numero, Numero e NUMERO sono considerati tre identificatori distinti.

Un identificatore non può avere lo stesso nome di una parola chiave del C e non dovrebbe avere lo stesso nome delle funzioni contenute nella libreria del C.

## 2.4 Le variabili

Come probabilmente il lettore già sa, una *variabile* corrisponde a una determinata cella di memoria utilizzata per contenere un valore che può essere modificato dal programma. Tutte le variabili C prima di essere utilizzate devono essere dichiarate. La forma generale di una dichiarazione è:

*tipo elenco_variabili;*

dove *tipo* deve essere un tipo di dati valido in C comprendendo eventuali modificatori mentre *elenco_variabili* può essere formato da uno o più nomi di identificatori separati da virgole. Ecco alcune dichiarazioni:

```
int i,j,l;
short int si;
unsigned int ui;
double balance, profit, loss;
```

Occorre ricordare che in C il nome della variabile non ha nulla a che fare con il suo tipo.

### Dove vengono dichiarate le variabili?

Le variabili possono essere dichiarate in tre luoghi: all'interno delle funzioni, nella definizione dei parametri delle funzioni e all'esterno di tutte le funzioni. Queste tre posizioni corrispondono a tre diversi tipi di variabili: le variabili locali, i parametri formali e le variabili globali.

#### Le variabili locali

Le variabili dichiarate all'interno di una funzione sono chiamate *variabili locali*. In alcuni testi si parla di queste variabili come di *variabili automatiche*. Questo libro utilizza il termine più comune di "variabile locale". Le variabili locali possono essere utilizzate solo dalle istruzioni che si trovano all'interno del blocco in

cui sono dichiarate. In altre parole, le variabili locali non sono note all'esterno del proprio blocco di codice. Occorre ricordare che un blocco di codice inizia con una parentesi graffa aperta e termina con una parentesi graffa chiusa.

La vita delle variabili locali è legata all'esecuzione del blocco di codice in cui sono dichiarate: questo significa che una variabile locale viene creata nel momento in cui si entra nel blocco e viene distrutta all'uscita.

Il blocco di codice più comune in cui sono dichiarate le variabili locali è la funzione. Ad esempio, si considerino le due funzioni seguenti:

```
void func1(void)
{
  int x;

  x = 10;
}


void func2(void)
{
  int x;

  x = -199;
}
```

La variabile intera x viene dichiarata due volte, una in func1() e una in func2(). La x in func1() non vede e non ha alcuna relazione con la x dichiarata in func2(). Questo avviene perché ogni x è nota solo al codice che si trova all'interno dello stesso blocco in cui la variabile è stata dichiarata.

Il linguaggio C contiene anche la parola chiave auto utilizzabile per dichiarare variabili locali. Tuttavia, poiché tutte le variabili non globali sono, normalmente, di tipo automatico, questa parola chiave non viene praticamente mai utilizzata. Questo è il motivo per cui negli esempi di questo libro questa parola non verrà utilizzata (si dice che la parola chiave auto sia stata inclusa nel C per consentire una compatibilità a livello di sorgente con il suo predecessore B). Di conseguenza, la parola chiave auto è stata inclusa anche nel C++ per garantire la compatibilità con il C.

Per comodità e per abitudine la maggior parte dei programmatori dichiara tutte le variabili locali utilizzate da una funzione immediatamente dopo la parentesi graffa di apertura della funzione e prima di ogni altra istruzione. Tuttavia, è possibile dichiarare variabili locali in qualunque altro punto del blocco di codice. Il blocco definito da una funzione non è che un caso specifico. Ad esempio,



```
void f(void)
{
  int t;

  scanf("%d",&t);

  if(t==1) {
    char s[80];   /* questa variabile viene creata
            solo dopo l'ingresso in questo blocco */
    printf("immettere un nome:");
    gets(s);
    /* operazioni varie ... */
  }
}
```

Qui, la variabile locale s viene creata all'interno del blocco di codice dell'istruzione if e distrutta alla sua uscita. Inoltre, s è nota solo all'interno del blocco appartenente all'if e non può essere utilizzata in altri punti, anche in altre parti della funzione che la contiene.

Un vantaggio della dichiarazione di una variabile locale all'interno di un blocco condizionale consiste nel fatto che la variabile verrà allocata solo se necessario. Questo avviene perché le variabili locali non vengono create se non nel momento in cui si accede al blocco in cui sono dichiarate. Questo può essere un fattore importante quando ad esempio si produce codice per controller dedicati (ad esempio un sistema di apertura di una porta di un garage che risponde a un codice di sicurezza digitale) in cui la RAM disponibile è molto scarsa.

La dichiarazione di variabili all'interno del blocco di codice che ne fa uso evita anche l'insorgere di effetti collaterali indesiderati. Poiché la variabile non esiste all'esterno del blocco in cui viene dichiarata, non potrà essere modificata nemmeno accidentalmente.

Vi è un'importante differenza fra il C e il C++ che consiste nella posizione in cui è possibile dichiarare le variabili locali. In C si devono dichiarare tutte le variabili locali all'inizio del blocco in cui sono definite e prima di ogni istruzione che esegua un'azione. Ad esempio, la seguente funzione produrrà un errore se compilata con un compilatore C.

```
/* Questa funzione è errata in C
     ma è accettata da qualsiasi
     compilatore C++. */
void f(void)
{
  int i;
```

```
  i = 10;

  int j;  /* questa riga provoca un errore */
  j = 20;
}
```

In C++ questa funzione è perfettamente corretta poiché è possibile definire variabili locali in qualsiasi punto del programma (la dichiarazione di variabili in C++ viene discussa nella Parte seconda di questa Guida completa).

Poiché le variabili locali vengono create e distrutte ad ogni ingresso e uscita dal blocco in cui sono state dichiarate, il loro contenuto viene perso all'uscita dal blocco. È fondamentale ricordare ciò quando si richiama una funzione. Nel momento in cui la funzione viene chiamata, vengono create tutte le sue variabili locali e alla sua uscita, tutte queste variabili vengono distrutte. Questo significa che le variabili locali non possono conservare il proprio valore da una chiamata della funzione alla successiva (anche se è possibile chiedere al compilatore di conservarne il valore utilizzando il modificatore static).

Se non si specifica altrimenti, le variabili locali vengono memorizzate nello stack. Il fatto che lo stack sia una regione di memoria dinamica e continuamente alterata spiega il motivo per cui le variabili locali non possono, in generale, conservare il proprio valore fra due chiamate di funzioni.

È possibile inizializzare una variabile locale a un determinato valore noto. Questo valore verrà assegnato alla variabile ogni volta che si accede al blocco di codice in cui la variabile stessa è dichiarata. Ad esempio, il seguente programma visualizza per dieci volte il numero 10:

```
#include <stdio.h>

void f(void);

int main(void)
{
  int i;

  for(i=0; i<10; i++)  f();

  return 0;
}

void f(void)

{
  int j = 10;
  printf("%d ", j);

  j++;  /* questa riga non ha effetti duraturi */
}
```



## Parametri formali

Se una funzione deve utilizzare argomenti, deve anche dichiarare variabili che accoglieranno i valori passati come argomenti. Queste variabili sono chiamate *parametri formali* della funzione. Essi si comportano come una qualunque altra variabile locale all'interno della funzione. Come si può vedere nel seguente frammento di programma, la loro dichiarazione si trova dopo il nome della funzione e fra parentesi:

```
/* Restituisce 1 se c si trova nella stringa s; 0 in caso contrario */
int is_in(char *s, char c)
{
  while(*s)
    if(*s==c) return 1;
    else s++;
  return 0;
}
```

La funzione is_in() ha due parametri: s e c. Questa funzione restituisce il valore 1 se il carattere specificato in c si trova all'interno della stringa s; in caso contrario restituisce 0.

Si deve specificare il tipo dei parametri formali utilizzando la dichiarazione mostrata nell'esempio. I parametri formali potranno quindi essere utilizzati all'interno della funzione come se fossero comuni variabili locali. Occorre ricordare che, come le variabili locali, anche i parametri formali sono dinamici e vengono distrutti all'uscita dalla funzione.

Come nel caso delle variabili locali, è possibile utilizzare i parametri formali per qualunque operazione di assegnamento e in qualunque espressione. Anche se queste variabili ricevono il loro valore dagli argomenti passati alla funzione, è possibile utilizzarle come qualsiasi altra variabile locale.

## Le variabili globali

A differenza della variabili locali, le *variabili globali* sono note all'intero programma e possono essere utilizzate in ogni punto del codice. Inoltre esse conservano il proprio valore durante l'intera esecuzione del programma. Le variabili globali devono essere dichiarate all'esterno di ogni funzione. In seguito, ogni

espressione ne potrà fare uso, indipendentemente dal blocco di codice in cui si trova.

Nel seguente programma, la variabile count è stata dichiarata all'esterno di tutte le funzioni. Anche se la sua dichiarazione si trova prima della funzione main(), la si sarebbe potuta posizionare in qualsiasi altro punto precedente il suo primo uso ma non all'interno di una funzione. Tuttavia è sempre meglio dichiarare le variabili globali all'inizio del programma.

```
#include <stdio.h>
int count;  /* count è globale  */

void func1(void);
void func2(void);

int main(void)
{
  count = 100;
  func1();
}

void func1(void)
{
  int temp;

  temp = count;
  func2();
  printf("count è uguale a %d", count); /* visualizza 100 */
}

void func2(void)
{
  int count;


  for(count=1; count<10; count++)
    putchar('.');
}
```

Osservando attentamente questo programma, si può notare che sebbene la variabile count non sia dichiarata né in main() né in func1(), entrambe le funzioni ne fanno uso. Al contrario, in func2() viene dichiarata la variabile locale count. Quindi, quando func2() utilizza count, accede alla propria variabile locale e non alla variabile globale. Se una variabile globale e una variabile locale hanno lo



stesso nome, ogni riferimento alla variabile all'interno del blocco di codice in cui è dichiarata la variabile locale, farà riferimento alla variabile locale e non avrà alcun effetto sulla variabile globale. Questo comportamento può essere estremamente utile ma dimenticandolo un programma, anche se può sembrare corretto, potrebbe iniziare a comportarsi in modo incomprensibile.

Le variabili globali vengono conservate dal compilatore C in una regione fissa della memoria che ha proprio questo scopo. Le variabili globali sono utili quando molte funzioni del programma devono utilizzare gli stessi dati. In generale, si deve però evitare di utilizzare le variabili globali quando non sono necessarie. Infatti esse occupano memoria per l'intera esecuzione del programma e non solo quando ve ne è bisogno. Inoltre, utilizzando una variabile globale in punti in cui si sarebbe potuta utilizzare una variabile locale si rende una funzione meno generale, in quanto essa fa affidamento su qualcosa che deve essere definito al suo esterno. Infine, utilizzando un gran numero di variabili globali, il programma può essere soggetto a errori a causa di effetti collaterali sconosciuti e indesiderati. Il problema principale nello sviluppo di grossi programmi è il sorgere di modifiche accidentali al valore di una variabile utilizzata in altri punti di un programma. Questo può avvenire in C e in C++ se si fa uso di un numero eccessivo di variabili globali.

## 2.5 I modificatori di accesso

Il C definisce due modificatori che controllano il modo in cui le variabili possono essere lette e modificate. Questi qualificatori sono const e volatile. Essi devono precedere il modificatore e il nome del tipo che qualificano.

### const

Le variabili di tipo const non possono essere modificate dal programma ma è possibile assegnare loro un valore iniziale. Il compilatore è libero di posizionare queste variabili in qualsiasi punto della memoria. Ad esempio,

```
const int a=10;
```

crea una variabile intera chiamata a con un valore iniziale pari a 10 che il programma non può modificare. Tuttavia, è possibile usare la variabile a in altri tipi di espressioni. Una variabile const riceve il proprio valore da un'inizializzazione esplicita o da strumenti hardware.

Il qualificatore const può essere utilizzato per evitare che una funzione modifichi gli oggetti puntati dagli argomenti della funzione stessa. Ovvero, quando un

puntatore viene passato a una funzione, tale funzione può modificare il valore della variabile puntata dal puntatore. Tuttavia, se il puntatore è specificato come const nella dichiarazione dei parametri, il codice della funzione non sarà in grado di modificare l'oggetto puntato. Ad esempio, la funzione sp_to_dash() del programma seguente stampa un trattino al posto di ogni spazio di un argomento stringa. In questo modo, la stringa "questa è una prova" verrà visualizzata come "questa-è-una-prova". L'uso di const nella dichiarazione del parametro assicura che il codice presente all'interno della funzione non possa modificare l'oggetto puntato dal parametro.

```
#include <stdio.h>

void sp_to_dash(const char *str);

int main(void)
{
  sp_to_dash("questa è una prova");


  return 0;
}

void sp_to_dash(const char *str)
{
  while(*str) {
    if(*str== ' ') printf("%c", '-');
    else printf("%c", *str);
    str++;
  }
}
```

Se si scrive sp_to_dash() in modo che la stringa possa essere modificata, la funzione non verrà compilata. Ad esempio, la seguente funzione produrrà un errore di compilazione:

```
/* errata */
void sp_to_dash(const char *str)
{
  while(*str) {
    if(*str==' ' ) *str = '-'; /* operazione non consentita */
    printf("%c", *str);        /* str è const */
    str++;
  }
}
```



Molte funzioni nella libreria standard del C utilizzano const nelle proprie dichiarazioni di parametri. Ad esempio, il prototipo della funzione strlen() è il seguente:

```
size_t strlen(const char *str);
```

Specificando str come const ci si assicura che strlen() non modifichi la stringa puntata da str. In generale, quando una funzione della libreria standard non deve modificare un oggetto puntato da un argomento, il parametro viene dichiarato come const.

È possibile utilizzare const per verificare che il programma non modifichi una variabile. Occorre ricordare che una variabile di tipo const può essere modificata da qualcosa che si trova all'esterno del programma, ad esempio un dispositivo hardware. Tuttavia, dichiarando una variabile come const, si può stabilire che ogni modifica a tale variabile avvenga in seguito a eventi esterni.

### volatile

Il modificatore volatile dice al compilatore che il valore di una variabile può essere modificato in modi non specificati esplicitamente dal programma. Ad esempio, l'indirizzo di una variabile globale può essere passato alla routine dell'orologio di sistema e utilizzata per conservare l'ora. In questa situazione, il contenuto della variabile viene modificato senza alcun assegnamento esplicito da parte del programma.

Questo è un fatto importante poiché la maggior parte dei compilatori C/C++ ottimizza automaticamente determinate espressioni assumendo che il contenuto di una variabile non possa essere modificato se non sul lato sinistro di una istruzione di assegnamento; pertanto la variabile non verrà riesaminata ad ogni accesso. Inoltre, alcuni compilatori alterano l'ordine di valutazione di un'espressione durante il processo di compilazione. Il modificatore volatile evita queste modifiche.

È anche possibile utilizzare i modificatori const e volatile insieme. Ad esempio, se si assume che 0x30 sia il valore di una porta che può essere modificata solo da eventi esterni, la seguente dichiarazione eviterà ogni possibilità di effetti collaterali accidentali.

```
const volatile char *port = (const volatile char *)0x30;
```

## 2.6 Specificatori di classe di memorizzazione

Il C consente l'uso di quattro specificatori di classe di memorizzazione:

extern
static
register
auto

Questi specificatori dicono al compilatore il modo in cui memorizzare le variabili. Si noti che lo specificatore precede la parte rimanente della dichiarazione della variabile. La sua forma generica è:

specificatore_memoria tipo *nome_var*

**NOTA** *Il C++ introduce un nuovo specificatore di classe d'accesso chiamato mutable che verrà descritto nella Parte seconda.*

## extern

Poiché il C/C++ consente la compilazione separata dei vari moduli che formano un grosso programma e il loro successivo collegamento con il linker, vi deve essere un modo per comunicare a tutti i file informazioni sulle variabili globali richieste dal programma. Anche se il C consente, tecnicamente, di definire più volte una variabile globale, non si tratta di una pratica "elegante" (oltre a causare potenziali problemi di linking). Ma soprattutto, in C++ è possibile definire una variabile globale una sola volta. Ma allora, come è possibile fornire a tutti i file che compongono il programma informazioni relative a tutte le variabili globali utilizzate? La soluzione di questo problema è rappresentata dalla distinzione fra la dichiarazione e la definizione di una variabile. Una dichiarazione dichiara semplicemente il nome e il tipo di una variabile. La definizione provoca invece l'allocazione della memoria per la variabile. Nella maggior parte dei casi, le dichiarazioni delle variabili sono anche definizioni. Tuttavia, se si fa precedere al nome della variabile lo specificatore extern, si può dichiarare una variabile senza definirla. Pertanto, in un programma composto da più file, è possibile dichiarare tutte le variabili globali in un file e utilizzare delle dichiarazioni extern negli altri file, come indicato nella Figura 2.1.

Nel File 2, l'elenco di variabili globali è stato copiato dal File 1 e alle dichiarazioni è stato aggiunto lo specificatore extern. Lo specificatore extern dice al compilatore che i nomi e i tipi di variabili che lo seguono sono stati definiti altrove. In altre parole, extern fa conoscere al compilatore il tipo e il nome di queste variabili globali senza in effetti occupare alcuno spazio di memoria. Quando il linker collegherà i due moduli, verranno risolti tutti i riferimenti alle variabili esterne.



**File 1**

```
int x,y;
char ch;
int main(void)
{
/* ... */
}

void func1()
{
x=123
}
```

**File 2**

```
extern int x,y;
extern char ch;
void func22(void)
{
x=y/10;
}

void func23()
{
y=10;
{
```

**Figura 2.1** Uso di variabili globali in moduli compilati separatamente.

La parola chiave extern ha la seguente forma generale:

extern *elenco-variabili*;

Vi è anche un altro uso, opzionale, della parola extern.

Quando si usa una variabile globale all'interno di una funzione, è possibile dichiararla come extern:

```
int first, last;  /* definizione globale di first
                     e last */

main(void)
{
  extern int first;  /* uso opzionale della
                        dichiarazione extern */
  .
  .
  .
}
```

Anche se le dichiarazioni di variabili extern mostrate in questo esempio sono consentite dal linguaggio, si tratta di dichiarazioni non necessarie. Se il compilatore trova una variabile che non è stata dichiarata all'interno del blocco corrente,

controlla se essa corrisponde a una delle variabili dichiarate in un blocco più esterno e in caso di risposta negativa, controlla le variabili globali. In caso la variabile venga trovata, il compilatore assume che si debba far riferimento a tale variabile globale.

In C++ lo specificatore extern ha anche un altro uso che verrà descritto nella Parte seconda.

## static

Le variabili static sono variabili permanenti all'interno della propria funzione o del proprio file. A differenza delle variabili globali, esse non sono note all'esterno della propria funzione o del proprio file, ma mantengono il proprio valore fra una chiamata e la successiva. Questa caratteristica le rende utilissime quando si scrivono funzioni o librerie generalizzate che possono essere utilizzate da altri programmatori. La parola chiave static ha effetti diversi sulle variabili locali e le variabili globali.

### Le variabili locali static

Quando si applica il modificatore static a una variabile locale, il compilatore crea per la variabile una cella di memoria permanente, così come avviene per le variabili globali. La differenza principale fra una variabile locale static e una variabile globale consiste nel fatto che la prima rimane nota solo all'interno del blocco in cui è dichiarata. In altri termini, una variabile locale static è una variabile locale che conserva il proprio valore fra due chiamate di funzione.

Le variabili locali static sono molto importanti per la creazione di funzioni isolate in quanto molti tipi di routine devono conservare un valore da una chiamata alla successiva. Se non fosse consentito l'uso di variabili locali static, si sarebbe costretti a usare variabili globali, che possono provocare effetti collaterali. Un esempio di funzione che beneficia dell'uso di variabili locali static è un generatore di serie di numeri che produce un nuovo valore sulla base del valore fornito in precedenza. Per conservare questo valore si potrebbe usare una variabile globale. Tuttavia, ogni volta che la funzione viene utilizzata in un programma, sarebbe necessario dichiarare tale variabile globale e assicurarsi che essa non entri in conflitto con altre variabili globali. La soluzione migliore consiste nel dichiarare la variabile che deve conservare il numero generato come static, come nel seguente frammento di programma:

```
int series(void)
{
  static int series_num;

  series_num = series_num+23;
  return series_num;
}
```



In questo esempio, la variabile series_num conserva il proprio valore da una chiamata alla funzione alla successiva, a differenza delle comuni variabili locali che vengono create e distrutte in continuazione. Questo significa che ogni chiamata alla funzione series() produce un nuovo valore basandosi sul numero precedentemente fornito e senza dichiarare alcuna variabile globale.

Inoltre, è possibile assegnare a una variabile locale static anche un valore di inizializzazione. Questo valore viene assegnato una sola volta all'avvio del programma (e non ogni volta che si entra nel blocco di codice, come nel caso delle comuni variabili locali). Ad esempio, questa versione di series() inizializza la variabile series_num a 100:

```
int series(void)
{
  static int series_num = 100;

  series_num = series_num+23;
  return series_num;
}
```

Con questa funzione, la serie inizia sempre dal valore 123. Anche se questo è accettabile per alcune applicazioni, la maggior parte dei generatori di serie richiede che sia l'utente a specificare il valore iniziale. Un modo per assegnare a series_num un valore specificato dall'utente consiste nel rendere series_num una variabile globale e di assegnarle il valore specificato. Tuttavia, il modificatore static serve proprio a non dichiarare series_num globale. Questo introduce il secondo uso di static.

### Le variabili globali static

Applicando lo specificatore static a una variabile globale, si chiede al compilatore di creare una variabile globale che sia nota solo all'interno del file in cui è stata dichiarata. Questo significa che anche se la variabile è globale, le routine di altri file non saranno in grado di vederla né di modificarne direttamente il contenuto, evitando così il sorgere di effetti collaterali. Per i pochi casi in cui una variabile locale static non è adatta, è possibile creare un piccolo file che contenga solo le funzioni che devono utilizzare la variabile globale static, compilare separatamente tale file e utilizzarlo senza temere effetti collaterali.

Per illustrare l'uso delle variabili globali static, si può immaginare che il generatore di serie dell'esempio precedente debba essere ricodificato in modo che la serie sia inizializzata da un valore *series* fornito da una chiamata a una seconda

funzione chiamata series_start(). L'intero file contenente series(), series_start() e series_num è illustrato di seguito:

```
/* Queste funzioni devono trovarsi nello stesso file, possibilmente da sole.
*/

static int series_num;
void series_start(int seed);
int series(void);

int series(void)
{
  series_num = series_num+23;
  return series_num;
}

/* inizializza series_num */
void series_start(int seed)
{
  series_num = seed;
}
```

Richiamando series_start() con un valore intero noto, si inizializza il generatore di serie. Dopo questo, le chiamate a series() generano l'elemento successivo della serie.

Per riassumere: i nomi delle variabili locali static sono noti solo all'interno del blocco di codice in cui sono dichiarate; i nomi delle variabili globali static sono note solo all'interno del file in cui si trovano. Se si inseriscono le funzioni series() e series_start() in una libreria, sarà possibile utilizzare le funzioni mentre sarà impossibile fare accesso alla variabile series_num che lo specificatore static nasconde al resto del codice del programma. Questo significa che è anche possibile dichiarare e utilizzare un'altra variabile anch'essa chiamata series_num (naturalmente in un altro file). In sostanza, il modificatore static consente l'uso di variabili note solo all'interno delle funzioni che ne richiedono l'uso, senza effetti collaterali indesiderati.

Le variabili static consentono di nascondere parti del programma rispetto ad altre parti. Questo può essere un notevole vantaggio quando si deve gestire un programma molto esteso e complesso.

**NOTA** *In C++ questo uso di static, pur essendo ancora supportato, è sconsigliato. Questo significa che è opportuno non utilizzarlo quando si realizza nuovo codice. In questo caso si dovrà invece far ricorso ai namespace, argomento discusso nella Parte seconda.*



## Variabili register

Lo specificatore di memorizzazione register si applica tradizionalmente solo a variabili di tipo int, char e puntatori. Lo Standard C ha esteso questa definizione per consentire l'applicazione dello specificatore register a ogni tipo di variabile.

Originariamente, lo specificatore register chiedeva che il compilatore conservasse il valore di una variabile in un registro della CPU e non in memoria insieme alle altre variabili. Questo significava che le operazioni su una variabile register potevano avvenire molto più velocemente rispetto a una comune variabile in quanto il valore della variabile register veniva sempre conservato nella CPU e non richiedeva alcun accesso alla memoria per determinarne o modificarne il valore.

Oggi, la definizione di register è stata notevolmente espansa e può essere applicata a qualsiasi tipo di variabile. Il C standard stabilisce semplicemente che "l'accesso all'oggetto sia il più rapido possibile" (lo standard C++ stabilisce che la parola register sia un "suggerimento per il compilatore che indica che l'oggetto dichiarato come tale verrà impiegato in modo massiccio"). In pratica, i caratteri e gli interi verranno ancora conservati nei registri della CPU. Gli oggetti di maggiori dimensioni come ad esempio gli array non potranno però esser conservati in un registro, ma riceveranno dal compilatore un trattamento preferenziale. A seconda dell'implementazione del compilatore C/C++ e del suo ambiente operativo, le variabili register possono essere gestite in molti modi sulla base delle decisioni dell'implementatore del compilatore. Ad esempio, è tecnicamente corretto anche che un compilatore ignori lo specificatore register e gestisca questo tipo di variabile come qualunque altra, ma in realtà questa è una pratica utilizzata raramente.

È possibile applicare lo specificatore register solo a variabili locali e ai parametri formali di una funzione. Pertanto, non è consentito l'uso di variabili globali register. Ecco un esempio che utilizza variabili register. Questa funzione calcola il risultato di $m^e$ per due numeri interi:

```
int int_pwr(register int m,  register int e)
{
  register int temp;

  temp = 1;

  for(; e; e--) temp = temp * m;
  return temp;
}
```

In questo esempio, e, m e temp sono dichiarate come variabili register in quanto vengono utilizzate all'interno di un ciclo. Il fatto che le variabili register siano ottimizzate per quanto riguarda la velocità, le rende ideali per il controllo o

l'uso all'interno di cicli. Generalmente, le variabili register vengono utilizzate nei punti in cui si dimostrano più efficaci, che sono spesso i luoghi in cui si eseguono più accessi alla stessa variabile. Questo è importante poiché è possibile dichiarare un qualsiasi numero di variabili di tipo register ma non tutte riceveranno la stessa ottimizzazione nella velocità di accesso.

Il numero di variabili register che è possibile ottimizzare in un determinato blocco di codice dipende sia dall'ambiente operativo che dalla specifica implementazione di C/C++. In ogni caso non ci si deve preoccupare di dichiarare troppe variabili register in quanto il compilatore C trasforma automaticamente le variabili register in variabili non register non appena si supera il limite consentito (questo assicura la trasportabilità del codice in un'ampia gamma di microprocessori).

Normalmente, nei registri della CPU possono essere conservate almeno due variabili register di tipo char o int. Poiché gli ambienti operativi possono essere molto diversi, per determinare se è possibile applicare le opzioni di ottimizzazione ad altri tipi di variabili, occorre consultare la documentazione del compilatore.

In C non è possibile conoscere l'indirizzo di una variabile register utilizzando l'operatore & (introdotto più avanti in questo stesso capitolo). Ciò è dovuto al fatto che una variabile register deve essere memorizzata in un registro della CPU del quale in genere non è possibile conoscere l'indirizzo. Questa restrizione non si applica al linguaggio C++. Tuttavia, il fatto di richiedere l'indirizzo di una variabile register in C++ può evitare che essa venga ottimizzata al meglio.

Anche se lo standard C/C++ ha espanso la descrizione di register oltre il suo significato tradizionale, in pratica ha un effetto significativo solo con variabili di tipo intero o carattere. Pertanto, non si dovrà prevedere un sostanziale aumento di prestazioni con altri tipi di variabili.

## 2.7 Inizializzazione delle variabili

Alla maggior parte delle variabili è possibile assegnare un valore direttamente nella dichiarazione, inserendo un segno di uguaglianza e un valore dopo il nome della variabile. La forma generica di inizializzazione è:

*tipo nome_variabile = valore;*

Ecco alcuni esempi di inizializzazione:

```
char ch = 'a';

int first = 0;

float balance = 123.23;
```



Le variabili globali e static locali sono inizializzate solo all'inizio del programma. Le variabili locali (ad esclusione delle variabili locali static) sono inizializzate ogni volta che si accede al blocco in cui sono dichiarate. Le variabili locali che non sono inizializzate, prima del primo assegnamento hanno un valore indefinito. Le variabili globali e le variabili static locali non esplicitamente inizializzate, vengono automaticamente inizializzate a 0.

## 2.8 Le costanti

Le *costanti* fanno riferimento a valori fissi che il programma non può alterare. Le costanti possono essere di uno qualsiasi dei tipi principali. Il modo in cui ogni costante è rappresentata dipende dal suo tipo. Le costanti vengono chiamate anche letterali. Le costanti di tipo carattere devono essere racchiuse fra apici (ad esempio 'a' e '%'). In C/C++ sono definiti anche i caratteri estesi (utilizzati principalmente per lingue non comuni) le quali occupano 16 bit. Per specificare una costante carattere estesa, si deve far precedere al carattere la lettera "L". Ad esempio:

```
wchar_t wc;
wc = L'A';
```

Qui a wc viene assegnata la costante a caratteri estesi equivalente alla lettera "A". Il tipo dei caratteri estesi si chiama wchar_t. In C questo tipo è definito in un file header, ovvero non è uno dei tipi standard del linguaggio. In C++, wchar_t è un tipo standard.

Le costanti intere sono specificate come numeri senza componente decimale. Ad esempio 10 e -100 sono costanti intere. Le costanti in virgola mobile contengono il punto decimale seguito da una componente decimale. Ad esempio, 11.123 è una costante in virgola mobile. In C/C++ è consentito anche l'uso della notazione scientifica per i numeri in virgola mobile.

Vi sono due tipi di numeri in virgola mobile: float e double. Vi sono anche molte varianti dei tipi principali che è possibile generare utilizzando i modificatori di tipo. Normalmente, il compilatore inserisce una costante numerica nel più piccolo tipo di dati compatibile in grado di contenerla. Pertanto, supponendo di utilizzare interi a 16 bit, 10 sarà normalmente un int mentre 103000 sarà un long. Anche se il valore 10 potrebbe rientrare nel tipo carattere, il compilatore farà normalmente uso di un int. L'unica eccezione alla regola del "tipo più piccolo" sono le costanti in virgola mobile che vengono sempre memorizzate come valori double.

Per la maggior parte dei programmi, le assunzioni fatte dal compilatore sono corrette. Tuttavia, è possibile anche specificare con precisione il tipo delle costan-

ti numeriche utilizzando un suffisso. Per i tipi in virgola mobile, se il numero è seguito dalla lettera F, verrà trattato come un float. Se il numero è seguito da una L, verrà memorizzato come un long double. Per i tipi interi, il suffisso U sta per unsigned mentre L sta per long. Ecco alcuni esempi:

| Tipo | Esempi di costanti |
|---|---|
| int | 1 123 21000 -234 |
| long int | 35000L -34L |
| short int | 10 -12 90 |
| unsigned int | 10000U 987U 40000 |
| float | 123.23F 4.34e-3F |
| double | 123.23 12312333 -0.9876324 |
| long double | 1001.2L |

## Costanti esadecimali e ottali

Talvolta è più comodo utilizzare un sistema numerico in base 8 o 16. Il sistema numerico che si basa sull'otto è chiamato *ottale* e utilizza le cifre da 0 a 7. Il numero 10 ottale corrisponde quindi al numero 8 decimale. Il sistema numerico in base 16 è chiamato *esadecimale* e utilizza le cifre da 0 a 9 e le lettere da A a F che sostituiscono rispettivamente i numeri 10, 11, 12, 13, 14 e 15. Ad esempio, il numero esadecimale 10 corrisponde al numero 16 decimale. Poiché questi sistemi numerici sono utilizzati molto frequentemente, il C/C++ consente di specificare costanti intere anche in formato esadecimale e ottale. Una costante esadecimale è formata dal prefisso 0x seguito dalla costante in formato esadecimale. Una costante ottale inizia con uno 0. Ecco alcuni esempi:

```
int hex = 0x80;    /* 128 in decimale */
int oct = 012;     /* 10 in decimale */
```

## Costanti stringa

Il C/C++ consente l'uso di un altro tipo di costanti: la stringa. Una *stringa* è formata da gruppi di caratteri racchiusi fra doppi apici. Ad esempio, "questo è un esempio" è una stringa. Altri esempi di stringhe sono quelli presenti in alcune istruzioni printf() dei programmi di esempio. Anche se il C consente di definire costanti stringa, non prevede formalmente il tipo stringa. Al contrario, il C++ prevede una classe apposita per le stringhe.



Occorre fare attenzione a non confondere le stringhe e i caratteri. Una costante carattere è racchiusa tra singoli apici, come ad esempio 'a'. Al contrario "a" è una stringa formata da un'unica lettera.

## Costanti carattere speciali

L'inclusione di costanti di tipo carattere in singoli apici è consentita anche per la maggior parte dei caratteri stampabili. Alcuni, però, come ad esempio il carattere di Carriage Return, non possono essere immessi in una stringa utilizzando la tastiera. Per questo motivo, il C/C++ prevede l'uso di *costanti carattere speciali* (elencate nella Tabella 2.2) che consentono di utilizzare questi caratteri come costanti. Tali elementi sono chiamati anche sequenze di escare. L'uso di questi codici al posto degli equivalenti codici ASCII aiuta a garantire la trasportabilità del codice.

**Tabella 2.2** Codici speciali.

| CODICE | SIGNIFICATO |
|---|---|
| \b | Backspace |
| \f | Form feed |
| \n | New line |
| \r | Carriage return |
| \t | Tabulazione orizzontale |
| \" | Doppi apici |
| \' | Apici |
| \0 | Carattere nullo |
| \\ | Backslash |
| \v | Tabulazione verticale |
| \a | Bip |
| \? | Punto interrogativo |
| \N | Costante ottale (dove N è una costante ottale) |
| \xN | Costante esadecimale (dove N è una costante esadecimale) |

Ad esempio, il programma seguente salta su una nuova riga, visualizza un carattere di tabulazione e poi stampa la stringa Questo è un esempio.

```
#include <stdio.h>
int main(void)
{
  printf("\n\tQuesto è un esempio");

  return 0;
}
```

## 2.9 Gli operatori

Il C/C++ è un linguaggio molto ricco di operatori. Infatti, fa molto più affidamento sugli operatori della maggior parte degli altri linguaggi di programmazione. Vi sono quattro classi principali di operatori: *aritmetici*, *relazionali*, *logici* e *bit-a-bit*. Inoltre prevede alcuni operatori adibiti a compiti specifici.

### L'operatore di assegnamento

In C/C++ è possibile utilizzare l'operatore di assegnamento all'interno di ogni espressione valida. In questo senso il C/C++ differisce dalla maggior parte dei linguaggi di programmazione (inclusi il Pascal, il BASIC e il FORTRAN), che considerano l'operatore di assegnamento come un'istruzione. Nel linguaggio C la forma generale dell'operatore di assegnamento è:

*nome_variabile=espressione*;

dove *espressione* può essere una semplice costante o complessa a piacere. Il C/C++ usa per l'assegnamento un singolo segno di uguaglianza (il Pascal e il Modula-2 utilizzano il costrutto :=). Sul lato sinistro dell'assegnamento deve trovarsi una variabile o un puntatore, non può invece trovarsi una funzione o una costante.

Frequentemente, nei volumi che trattano la programmazione C/C++ e nei messaggi di errore del compilatore si trovano i due termini *lvalue* e *rvalue*. Questi oggetti corrispondono a ciò che si trova, rispettivamente, sul lato sinistro (left value – *lvalue*) e sul lato destro (right value – *rvalue*) dell'operatore di assegnamento. In termini pratici, con *lvalue* si intende la variabile mentre con *rvalue* si intende il valore dell'espressione che si trova sul lato destro dell'operatore di assegnamento.



### Conversione di tipo negli assegnamenti

Quando si utilizzano insieme variabili di tipo diverso, avviene una *conversione di tipo* automatica. In un'istruzione di assegnamento, la regola per la conversione di tipo è semplice: il valore che si trova al lato destro dell'assegnamento (l'espressione) viene convertito in ciò che si trova al lato sinistro (la variabile) come illustrato di seguito:

```
int x;
char ch;
float  f;

void func(void)
{
  ch = x;    /* riga 1 */
  x = f;     /* riga 2 */
  f = ch;    /* riga 3 */
  f = x;     /* riga 4 */
}
```

Nella riga 1, i primi bit (superiori) della variabile intera x vengono eliminati, lasciando in ch solo gli 8 bit meno significativi. Quindi, se x è compreso fra 0 e 256, ch e x avranno valori identici. In tutti gli altri casi, il valore di ch rifletterà solo gli 8 bit inferiori di x. Nella riga 2, x riceve la parte intera di f. Nella riga 3, f converte il valore intero a 8 bit memorizzato in ch nello stesso valore in virgola mobile. Questo avviene anche nella riga 4, tranne per il fatto che f converte un valore intero in un valore in virgola mobile.

Quando si eseguono conversioni da interi a caratteri e da interi long a interi normali, viene eliminato un numero appropriato di bit superiori. In molti ambienti operativi a 16 bit, questo significa che passando da un intero a un carattere, si perdono 8 bit e passando da un intero long a un intero normale si perdono 16 bit. In ambienti a 32 bit, la conversione di un intero in un carattere provoca la perdita di 24 bit mentre la conversione da un intero a un intero short provoca la perdita di 16 bit.

La Tabella 2.3 riassume le conversioni di tipo eseguite dagli assegnamenti. È bene sapere che una conversione di un int in un float o di un float in un double e così via non aggiunge precisione al numero. Questo tipo di conversione cambia solo il formato in cui il valore viene rappresentato. Inoltre alcuni compilatori, nella conversione di una variabili char in un int o in un float, considerano la variabile sempre positiva indipendentemente dal valore che essa contiene. Altri compilatori considerano negativi i valori char maggiori di 127. In generale, è bene usare variabili char solo per i caratteri e usare per i numeri solo i tipi int, short int e signed char; questo eviterà problemi di trasportabilità.

Per utilizzare la Tabella 2.3 per eseguire una conversione non indicata, basta convertire un tipo alla volta fino a giungere al tipo di destinazione. Ad esempio, per convertire un double in un int, si deve prima convertire il double in un float e poi il float in un int.

## Assegnamenti multipli

Il C/C++ consente di specificare più variabili in un'unica istruzione di assegnamento; tutte le variabili assumeranno quindi lo stesso valore. Ad esempio, questo frammento di programma assegna alle variabili x, y e z il valore 0:

```
x = y = z = 0;
```

Nei programmi professionali, si fa normalmente un grande uso di questo di questo metodo di assegnamento.

**Tabella 2.3** Conversioni di tipo (assumendo word di 16 bit).

| TIPO DI DESTINAZIONE | TIPO DELL'ESPRESSIONE | POSSIBILE PERDITA DI INFORMAZIONI |
|---|---|---|
| signed char | char | Se il valore è > 127, la destinazione è negativa |
| char | short int | 8 bit più significativi |
| char | int (16 bit) | 8 bit più significativi |
| char | int (32 bit) | 24 bit più significativi |
| char | long int | 24 bit più significativi |
| short int | int (16 bit) | Nulla |
| short int | int (32 bit) | 16 bit più significativi |
| int (16 bit) | long int | 16 bit più significativi |
| int (32 bit) | long int | Nulla |
| int | float | La parte decimale e, in alcuni casi, parte dell'intero |
| float | double | Precisione, il risultato viene arrotondato |
| double | long double | Precisione, il risultato viene arrotondato |



## Operatori aritmetici

La Tabella 2.4 elenca gli operatori aritmetici presenti in C/C++. Gli operatori +, -, * e / si comportano come i corrispondenti operatori presenti in altri linguaggi di programmazione. Questi operatori possono essere applicati a quasi tutti i tipi di dati consentiti. Se si applica l'operatore / a un intero o a un carattere, il resto verrà troncato. Ad esempio, in una divisione fra interi 5 / 2 sarà uguale a 2.

L'operatore di modulo, opera in C/C++ come in molti altri linguaggi, fornendo il resto di una divisione intera. Tuttavia, non è possibile utilizzare l'operatore per i tipi in virgola mobile. Il seguente frammento di codice illustra l'uso dell'operatore %:

```
int x, y;

x = 5;
y = 2;

printf("%d", x/y);      /* visualizza 2 */
printf("%d", x%y);      /* visualizza 1, il resto della
                           divisione intera */

x = 1;
y = 2;

printf("%d %d", x/y, x%y);     /* visualizza 0 1 */
```

L'ultima riga stampa uno 0 e un 1 poiché 1 / 2 nella divisione intera è uguale a 0 e il resto è 1.

**Tabella 2.4** Operatori aritmetici.

| OPERATORE | AZIONE |
|---|---|
| - | Sottrazione, anche meno unario |
| + | Addizione |
| * | Moltiplicazione |
| / | Divisione |
| % | Modulo |
| - - | Decremento |
| + + | Incremento |

Il meno unario moltiplica il suo operando per -1. Questo significa che un numero preceduto dal segno meno cambia il proprio segno.

## Incremento e decremento

Il C/C++ include due utili operatori che generalmente non sono presenti in altri linguaggi di programmazione: gli operatori di incremento (+ +) e decremento (- -). Questi operatori, rispettivamente, aggiungono e sottraggono una unità al proprio operando. In altre parole:

```
x = x+1;
```

equivale a:

```
++x;
```

e

```
x = x-1;
```

equivale a:

```
x- -;
```

Gli operatori di incremento e decremento possono precedere (forma prefissa) o seguire (forma postfissa) l'operando. Ad esempio

```
x = x+1;
```

può essere scritto come

```
++x;
```

o

```
x++;
```

Tuttavia, vi è una differenza fra le forme prefissa e postfissa quando si utilizzano questi operatori in un'espressione. Quando l'operatore di incremento o decremento precede il proprio operando, il C/C++ esegue l'incremento o il decremento prima di fornire il valore dell'operando all'espressione. Se l'operatore segue il proprio operando, il C/C++ prima fornisce il valore dell'operando e poi lo incrementa o decrementa. Ad esempio,



```
x = 10;
y = ++ x;
```

assegna a y il valore 11. Se lo stesso codice fosse stato scritto come

```
x = 10;
y = x++;
```

a y sarebbe stato assegnato il valore 10. In entrambi i casi, a x sarà stato assegnato il valore 11; la differenza sta nel momento in cui cambia il valore di x.

La maggior parte dei compilatori C/C++ produce codice oggetto molto efficiente per le operazioni di incremento e decremento, molto migliore rispetto a quello generato utilizzando l'equivalente operatore di assegnamento. Per questo motivo, è sempre bene preferire, quando possibile, gli operatori di incremento e decremento.

La precedenza degli operatori aritmetici è la seguente:

| | |
|---|---|
| **alta** | + + - - |
| | - (meno unario) |
| | * / % |
| **bassa** | + - |

Gli operatori con lo stesso livello di precedenza vengono valutati dal compilatore da sinistra verso destra. Naturalmente, è possibile utilizzare le parentesi per modificare l'ordine di valutazione. Il C/C++ considera le parentesi nello stesso modo di ogni altro linguaggio di programmazione. Le parentesi forzano il calcolo di un'operazione o di un gruppo di operazioni in modo che assumano un livello di precedenza superiore.

## Operatori relazionali e logici

Nel termine *operatore relazionale*, con relazionale si intende la relazione che un valore ha rispetto a un altro. Nel termine *operatore logico*, l'aggettivo "logico" fa riferimento ai modi in cui queste relazioni possono essere connesse. Questi due tipi di operatori vengono discussi insieme in quanto spesso vengono utilizzati congiuntamente. Il concetto di operatore relazionale e logico si basa sull'idea di verità e falsità. È vero (true) qualsiasi valore diverso da zero mentre è falso (false) lo zero. Le espressioni che utilizzano operatori relazionali o logici restituiscono 0 per false e 1 per true.

Il C++ supporta completamente il concetto di zero associato a false e "non-zero" associato a true. Tuttavia viene definito anche un nuovo tipo di dati chiamato bool e le costanti booleane true e false. In C++ il valore 0 viene automaticamente convertito in false e un valore diverso da zero viene automaticamente convertito in true. Vale anche il contrario: true viene convertito in 1 e false viene convertito in 0. In C++ il risultato di un'operazione relazionale o logica è true o false. Ma poiché questi valori vengono automaticamente convertiti in 1 e 0, la distinzione fra C e C++ in questo campo è perlopiù accademica.

La Tabella 2.5 elenca gli operatori relazionali e logici. La tabella di verità per gli operatori logici è mostrata in termini di 1 e 0.

| p | q | p&&q | p\|\|q | !p |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 1 |
| 0 | 1 | 0 | 1 | 1 |
| 1 | 1 | 1 | 1 | 0 |
| 1 | 0 | 0 | 1 | 0 |

Gli operatori relazionali e logici hanno una precedenza inferiore rispetto agli operatori aritmetici. Perciò un'espressione come 10 > 1 + 12 viene valutata come 10 > (1 + 12) e il risultato è ovviamente falso.

**Tabella 2.5** Operatori relazionali e logici.

**OPERATORI RELAZIONALI**

| OPERATORE | AZIONE |
|---|---|
| > | Maggiore di |
| >= | Maggiore o uguale |
| < | Minore di |
| <= | Minore o uguale |
| == | Uguale |
| != | Diverso |

**OPERATORI LOGICI**

| OPERATORE | AZIONE |
|---|---|
| && | AND |
| \|\| | OR |
| ! | NOT |



In un'espressione è possibile riunire più operazioni come nell'esempio seguente:

```
10 > 5 && !(10 < 9) || 3 <= 4
```

In questo caso, il risultato è vero.

Anche se né il C né il C++ contengono l'operatore logico di OR esclusivo (XOR), è facile creare una funzione che esegua questa operazione utilizzando gli altri operatori logici. Il risultato di uno XOR è true se e solo se un solo operando è vero ma non entrambi.

Il seguente programma contiene la funzione xor() che restituisce il risultato di un'operazione di OR esclusivo eseguita su due argomenti:

```
#include <stdio.h>

int xor(int a, int b);

int main(void)
{
  printf("%d", xor(1, 0));
  printf("%d", xor(1, 1));
  printf("%d", xor(0, 1));
  printf("%d", xor(0, 0));

  return 0;
}

/* Esegue uno XOR logico utilizzando
   i due argomenti. */
int xor(int a, int b)
{
  return (a || b) && !(a && b);
}
```

La tabella seguente mostra i livelli di precedenza relativa esistenti fra gli operatori relazionali e logici:

| | |
|---|---|
| **Alta** | ! |
| | > >= < <= |
| | == != |
| | && |
| **Bassa** | \|\| |

Come nel caso delle espressioni aritmetiche, è possibile utilizzare le parentesi per modificare l'ordine di valutazione naturale. Ad esempio, l'espressione:

```
!0 && 0 || 0
```

è falsa. Se invece si aggiungono le parentesi alla stessa espressione, come mostrato di seguito, il risultato sarà vero:

```
!(0 && 0) || 0
```

Occorre ricordare che tutte le espressioni relazionali e logiche producono un risultato pari a 0 o a 1. Pertanto, il seguente frammento di programma non solo è corretto, ma visualizza il numero 1.

```
int x;

x = 100;
printf("%d", x>10);
```

## Operatori bit-a-bit

A differenza di molti altri linguaggi, il C/C++ è dotato di una dotazione completa di operatori bit-a-bit. Poiché il C è stato progettato per prendere il posto del linguaggio Assembler nella maggior parte dei campi di programmazione, doveva essere in grado di eseguire molte operazioni che normalmente potevano essere svolte solo in Assembler, incluse le operazioni sui bit. Le *operazioni bit-a-bit* si occupano di controllare, impostare o spostare i bit che compongono un byte o una word, che corrispondono ai tipi char e int e relative varianti. Quindi non è possibile utilizzare le operazioni bit-a-bit sui tipi float, double, long double, void, bool e sui tipi più complessi. La Tabella 2.6 elenca tutti gli operatori bit-a-bit. Queste operazioni si applicano ai singoli bit degli operandi.

**Tabella 2.6** Operatori bit-a-bit.

| OPERATORE | AZIONE |
|---|---|
| & | AND |
| \| | OR |
| ^ | OR esclusivo (XOR) |
| ~ | Complemento a uno |
| >> | Scorrimento a destra |
| << | Scorrimento a sinistra |

Gli operatori bit-a-bit AND, OR e NOT (complemento a uno) sono governati dalla stessa tabella di verità dei corrispondenti operatori logici, tranne per il fatto che operano sui bit. La tabella dell'operatore di OR esclusivo (XOR) è la seguente:

| p | q | p ^q |
|---|---|---|
| 0 | 0 | 0 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |
| 0 | 1 | 1 |

Come indica la tabella, il risultato di uno XOR è vero se uno solo degli operandi è vero; in caso contrario il risultato è falso.

Le operazioni bit-a-bit trovano applicazione molto spesso nel caso della realizzazione di driver per dispositivi (programmi per modem, routine per la gestione dei file e routine di stampa) in quanto consentono di mascherare determinati bit, ad esempio quello di parità (il bit di parità conferma che la parte rimanente dei bit del byte non sia modificata; normalmente si tratta del bit più alto di ogni byte).

Si può pensare all'operatore di AND bit-a-bit come a un modo per cancellare un bit. Infatti, ogni bit per il quale almeno uno degli operandi è uguale a 0 assumerà il valore 0. Ad esempio, la funzione seguente legge un carattere dalla porta del modem e riporta a 0 il bit di parità:

```
char get_char_from_modem(void)
{
      char ch;

      ch = read_modem(); /* preleva un carattere
                            dalla porta del modem */
      return(ch & 127);
}
```

La parità è costituita normalmente dall'ottavo bit che viene posto uguale a 0 eseguendo un'operazione di AND bit-a-bit con un byte in cui i sette bit più bassi sono uguali a 1 e il bit più alto è uguale a 0. L'espressione ch & 127 esegue un AND dei bit contenuti in ch con i bit che compongono il numero 127. Il risultato è che l'ottavo bit di ch diviene uguale a 0. Nell'esempio seguente, si assume che in ch sia stato ricevuto il carattere "A" e che il bit di parità sia uguale a 1:

```
   Bit di parità
        ↓
   11000001     ch contenente una "A" e bit di parità impostato a 1
   01111111     127 in binario
 & --------     AND bit-a-bit
   01000001     "A" senza parità
```

L'operatore di OR bit-a-bit è il contrario dell'operatore AND e può essere utilizzato per impostare a 1 un bit. Infatti, il bit risultante sarà uguale a 1 se almeno uno dei bit che fungono da operandi è uguale a 1. Ad esempio, la seguente operazione corrisponde a 128 | 3:

```
   10000000     128 in binario
   00000011     3 in binario
 | --------     OR bit-a-bit
   10000011     risultato
```

Un'operazione di OR esclusivo, normalmente abbreviato con XOR, imposta un bit a 1 se e solo se i bit confrontati sono diversi. Ad esempio, il risultato di 127^120 è:

```
   01111111     127 in binario
   01111000     120 in binario
 ^ --------     XOR bit-a-bit
   00000111     risultato
```

Occorre ricordare che gli operatori relazionali e logici producono sempre un risultato uguale a 0 o 1, mentre le analoghe operazioni bit-a-bit possono produrre un valore diverso, sulla base dell'operazione eseguita. In altre parole, le operazioni bit-a-bit possono produrre valori diversi da 0 o 1, mentre gli operatori logici restituiscono sempre un valore pari a 0 o a 1.

Gli operatori di scorrimento di bit, >> e <<, spostano tutti i bit di una variabile rispettivamente verso destra o verso sinistra. La forma generale di un'istruzione di scorrimento a destra è:

*variabile* >> *numero di posizioni*

La forma generale di un'istruzione di scorrimento a sinistra è:

*variabile* << *numero di posizioni*



In queste operazioni, si perde un bit a un'estremità mentre il nuovo bit che si crea all'altra estremità è sempre impostato a 0 (nel caso di un numero intero signed negativo, lo scorrimento a destra fa in modo che all'inizio del numero venga conservato il bit a 1 che corrisponde al segno negativo). Occorre ricordare che uno scorrimento non è una rotazione. Questo significa che i bit che fuoriescono da un'estremità non riappaiono all'altra estremità e vengono pertanto persi.

Le operazioni di scorrimento dei bit possono essere molto utili per decodificare l'input di un dispositivo esterno, come ad esempio un convertitore digitale/analogico e per leggere informazioni di stato. Gli operatori di scorrimento bit-a-bit consentono inoltre eseguire velocissime moltiplicazioni e divisioni di interi. Uno scorrimento a destra infatti divide un numero per due e uno scorrimento a sinistra lo moltiplica per due, come si può vedere dalla Tabella 2.7. Il programma seguente illustra l'uso degli operatori di scorrimento:

```
/* Un esempio di scorrimento di bit. */
#include <stdio.h>

int main(void)
{
  unsigned int i;
  int j;

  i = 1;

  /* scorrimenti a sinistra */
  for(j=0; j<4; j++) {
    i = i << 1;    /* scorrimento a sinistra di i di 1 posizione,
            che corrisponde a una moltiplicazione per 2 */
    printf("Scorrimento a sinistra di %d: %d\n", j, i);
  }

  /* scorrimenti a destra */
  for(j=0; j<4; j++) {
    i = i >> 1;    /* scorrimento a destra di i di 1 posizione,
            che corrisponde a una divisione per 2 */
    printf("scorrimento a destra di %d: %d\n", j, i);
  }

  return 0;
}
```

L'operatore di complemento a uno, ~, inverte lo stato di ogni bit del suo operando. In altre parole ogni 1 sarà tramutato in uno 0 e viceversa.

**Tabella 2.7** Moltiplicazione e divisione con gli operatori di scorrimento.

| UNSIGNED CHAR X; | VALORE DI X DOPO L'ISTRUZIONE | VALORE DI X |
|---|---|---|
| x=7; | 0 0 0 0 0 1 1 1 | 7 |
| x=x<<1; | 0 0 0 0 1 1 1 0 | 14 |
| x=x<<3; | 0 1 1 1 0 0 0 0 | 112 |
| x=x<<2; | 1 1 0 0 0 0 0 0 | 192 |
| x=x>>1; | 0 1 1 0 0 0 0 0 | 96 |
| x=x>>2; | 0 0 0 1 1 0 0 0 | 24 |

Ogni scorrimento a sinistra moltiplica il numero per due. Come si può notare dopo l'operazione x<<2, si perde un bit a un'estremità. Ogni scorrimento a destra divide il numero per due. Come si può notare le successive divisioni non fanno riapparire i bit persi.

Gli operatori bit-a-bit sono molto utilizzati nelle routine di cifratura. Se si vuole fare in modo che un file appaia illeggibile, basta eseguire su di esso qualche operazione bit-a-bit. Uno dei metodi più semplici consiste nell'eseguire il complemento di ogni byte utilizzando il complemento a 1 per invertire tutti i bit del byte, come indicato dal seguente esempio:

| | | |
|---|---|---|
| Byte originale | 00101100 | |
| Dopo il primo complemento | 11010011 | Uguali |
| Dopo il secondo complemento | 00101100 | |

Una sequenza di due complementi su un byte produce sempre il numero originale. Pertanto, il primo complemento rappresenta la versione codificata del byte e il secondo complemento decodifica il byte ritornando al valore originale.

Per codificare un carattere si può utilizzare la funzione encode() mostrata di seguito:

```
/* Una semplice funzione di cifratura. */
char encode(char ch)
{
  return(~ch); /* complemento */
}
```

## L'operatore ?

Il C/C++ contiene un operatore molto potente e comodo che sostituisce alcune istruzioni nel formato if-then-else.



La forma generale dell'operatore ternario ? è la seguente:

*Esp1* ? *Esp2* : *Esp3*;

dove *Esp1*, *Esp2* e *Esp3* sono espressioni. Si noti l'uso e la posizione dei due punti.

L'operatore ? esegue la seguente operazione: viene valutata l'espressione *Esp1*; se è vera, viene valutata *Esp2* che diverrà il valore dell'espressione. Se *Esp1* è falsa, viene valutata *Esp3* e il suo valore diviene il valore dell'espressione. Ad esempio, in:

```
x = 10;

y = x>9 ? 100 : 200;
```

a y viene assegnato il valore 100. Se x fosse stata minore di 9, ad y sarebbe stato assegnato il valore 200. Lo stesso codice scritto utilizzando un'espressione if-else avrebbe il seguente aspetto:

```
x = 10;

if(x>9) y = 100;
else y = 200;
```

L'operatore ? verrà discusso dettagliatamente nel Capitolo 3 insieme alle altre istruzioni condizionali.

## Gli operatori per i puntatori: & e *

Un *puntatore* è l'indirizzo in memoria di un oggetto. Una *variabile puntatore* è una variabile dichiarata in modo specifico per contenere un puntatore a un oggetto di un determinato tipo. La conoscenza dell'indirizzo di una variabile può essere molto importante in alcuni tipi di routine. Tuttavia, in C/C++ i puntatori hanno principalmente tre funzioni. Possono essere un metodo rapido per far riferimento agli elementi di un array. Essi consentono alle funzioni C di modificare i parametri di chiamata. Infine, consentono la creazione di liste concatenate e di altre strutture di dati dinamiche. I puntatori vengono discussi approfonditamente nel Capitolo 5. In questo Capitolo si introducono semplicemente i due operatori utilizzati per manipolare i puntatori.

Il primo operatore è &, un operatore unario che restituisce l'indirizzo in memoria del proprio operando (un operatore unario richiede un solo operando). Ad esempio,

```
m = &count;
```

inserisce nella variabile m l'indirizzo di memoria in cui si trova la variabile count.

Questo indirizzo corrisponde alla posizione fisica della variabile nella memoria del computer. Quindi l'indirizzo non ha nulla a che vedere con il valore di count. Si può quindi pensare a & leggendolo come "indirizzo di". Pertanto, l'istruzione di assegnamento precedente significa "m riceve l'indirizzo di count".

Per meglio comprendere questo assegnamento, si può assumere che la variabile count si trovi nell'indirizzo di memoria 2000. Inoltre si può assumere che count contenga il valore 100. Quindi, dopo l'assegnamento precedente, in m si troverà il valore 2000.

Il secondo operatore per i puntatori è *, che è l'inverso di &. L'operatore * è un operatore unario che restituisce il valore della variabile che si trova all'indirizzo che segue l'asterisco. Ad esempio, se m contiene l'indirizzo di memoria della variabile count,

```
q = *m;
```

inserisce il valore di count in q. Ora in q si troverà il valore 100, poiché all'indirizzo 2000 (l'indirizzo di memoria che è memorizzato in m) si trova il valore 100. Si può pensare a * leggendolo come "all'indirizzo". In questo caso, l'istruzione precedente si potrebbe leggere come "q riceve il valore che si trova all'indirizzo m".

Sfortunatamente, il simbolo di moltiplicazione e il simbolo "all'indirizzo" sono identici così come i simboli per l'operatore AND bit-a-bit e il simbolo "indirizzo di". Nonostante ciò, questi operatori non hanno alcuna relazione gli uni con gli altri. Gli operatori & e * hanno entrambi una precedenza più elevata rispetto a tutti gli operatori aritmetici tranne il meno unario il quale ha la loro stessa precedenza.

Le variabili che devono contenere indirizzi di memoria (ad esempio i puntatori) devono essere dichiarate inserendo un * davanti al nome della variabile. Questo comunica al compilatore che la variabile deve contenere un puntatore. Ad esempio, per dichiarare ch come puntatore a un carattere, si deve utilizzare la forma:

```
char *ch;
```

In questo caso, ch non è un carattere ma un puntatore a un carattere: la differenza è notevole. Il tipo di dati puntato dal puntatore, in questo caso char, è detto *tipo base* del puntatore. Tuttavia, anche il puntatore è una variabile che contiene l'indirizzo di un oggetto del proprio tipo base. Pertanto, un puntatore a carattere (o un qualsiasi puntatore) ha dimensioni sufficienti per contenere un indirizzo; tali dimensioni sono definite dall'architettura del computer utilizzato. Tuttavia, occorre ricordare che un puntatore può puntare solo a dati corrispondenti al proprio tipo base.

In una stessa istruzione di dichiarazione è possibile utilizzare variabili normali e variabili puntatore. Ad esempio,

```
int x, *y, count;
```

dichiara x e count come tipi interi e y come un puntatore a un tipo intero.

Il programma seguente utilizza gli operatori * e & per inserire il valore 10 nella variabile target. Come ci si può attendere, questo programma visualizza il valore 10.

```
#include <stdio.h>

int main(void)
{
  int target, source;
  int *m;

  source = 10;
  m = &source;
  target = *m;

  printf("%d", target);

  return 0;
}
```

## L'operatore sizeof

sizeof è un operatore unario che interviene al momento della compilazione restituendo la lunghezza, in byte, di una variabile o di uno specificatore di tipo racchiuso fra parentesi. Ad esempio, assumendo che gli interi siano formati da 4 byte e i double da 8 byte, il frammento di codice

```
double f;

printf("%f ", sizeof f);
printf("%d", sizeof(int));
```

visualizzerà i numeri 8 e 4.

Per calcolare le dimensioni di un tipo, quest'ultimo deve essere racchiuso fra parentesi. Questo non è invece necessario nel caso di nomi di variabili.

Il C/C++ definisce (con typedef) un tipo particolare chiamato size_t, che corrisponde a grandi linee a un intero unsigned. Tecnicamente, il valore restituito da

sizeof è di tipo size_t. Per ogni utilizzo pratico, tuttavia, si può utilizzare un valore intero unsigned.

sizeof aiuta principalmente a generare codice trasportabile che dipende dalle dimensioni dei tipi standard del C. Ad esempio, si immagini un programma di database che debba memorizzare sei valori interi in ogni record. Se si vuole rendere il programma trasportabile su più piattaforme, non si deve fare alcuna assunzione sulle dimensioni degli interi ma determinare la loro lunghezza effettiva con sizeof. In questo modo, per scrivere un record sul file si potrà usare una routine simile alla seguente:

```
/* Scrive 6 interi su un file. /
void put_rec(int rec[6], FILE fp)
{
  int len;

  len = fwrite(rec, sizeof(int)*6, 1, fp);
  if(len != 1) printf("Errore di scrittura");
}
```

In questo modo, put_rec() viene compilata ed eseguita correttamente su qualsiasi computer, indipendentemente dalle dimensioni in byte di un intero. Un'ultima annotazione: sizeof viene valutato al momento della compilazione e il valore prodotto viene trattato all'interno del programma come se fosse una costante.

## L'operatore virgola

L'operatore virgola consente di concatenare più espressioni. Il lato sinistro dell'operatore virgola viene sempre valutato come void. Questo significa che l'espressione che si trova al lato destro diviene il valore dell'intera espressione complessa separata da virgole. Ad esempio, l'espressione

```
x=(y=3, y+1);
```

assegna a y il valore 3 e quindi a x il valore 4. Le parentesi sono necessarie poiché l'operatore virgola ha una precedenza più bassa rispetto all'operatore di assegnamento.

Essenzialmente, la virgola crea una sequenza di operazioni. Se utilizzata sul lato destro di un'istruzione di assegnamento, il valore assegnato è quello dell'ultima espressione dell'elenco separato da virgole.

L'operatore virgola può essere considerato come la congiunzione "e" del linguaggio naturale così come viene utilizzata nella frase "fai questo-e-questo".



## Gli operatori punto (.) e freccia (->)

In C gli operatori . (punto) e -> (freccia) consentono di accedere ai singoli elementi delle strutture e delle unioni. Le *strutture* le *unioni* sono tipi aggregati ai quali è possibile fare riferimento con un solo nome (vedere il Capitolo 7). In C++ gli operatori punto e freccia vengono utilizzati anche per accedere ai membri di una classe.

L'operatore punto è utilizzato quando si opera direttamente sulla struttura o sull'unione. L'operatore freccia è utilizzato quando si opera con un puntatore a una struttura o a un'unione. Ad esempio, dato il frammento di codice,

```
struct employee
{
  char name[80];
  int age;
  float wage;
} emp;

struct employee *p = &emp; /* a p viene assegnato l'indirizzo di emp */
```

per assegnare il valore 123.23 al membro wage della variabile strutturata emp si deve utilizzare la seguente riga di codice:

```
emp.wage = 123.23;
```

Se invece si utilizza il puntatore alla variabile emp, si dovrà utilizzare la riga:

```
p->wage = 123.23;
```

## Gli operatori ( ) e [ ]

Le parentesi tonde sono operatori che aumentano la precedenza delle operazioni che racchiudono.

Le parentesi quadre consentono di accedere tramite indici agli elementi di un array (gli array verranno discussi approfonditamente nel Capitolo 4). Dato un array, l'espressione fra parentesi quadre deve fornire un indice per accedere all'array. Ad esempio,

```
#include <stdio.h>
char s[80];

int main(void)
```

```
{
  s[3] = 'X';
  printf("%c", s[3]);

  return 0;
}
```

prima assegna il valore "X" al quarto elemento (in C/C++ gli array iniziano dall'indice 0) dell'array s e quindi stampa tale elemento.

### Riepilogo dell'ordine di precedenza

La Tabella 2.8 elenca la precedenza di tutti gli operatori C. Occorre notare che tutti gli operatori, ad eccezione degli operatori unari e del ?, sono associativi da sinistra a destra. Gli operatori unari (*, &, -) e ? sono associativi da destra a sinistra.

**NOTA** *Il C++ definisce alcuni operatori aggiuntivi che verranno però discussi dettagliatamente nella Parte seconda di questa guida*

## 2.10 Le espressioni

Gli operatori, le costanti e le variabili sono gli elementi costitutivi delle espressioni. Un'*espressione* C/C++ è una combinazione valida di questi elementi. Poiché la maggior parte delle espressioni tende a seguire le regole generali dell'algebra, si dà per nota la loro conoscenza. Tuttavia, vi sono alcuni aspetti delle espressioni che sono specifici del C e del C++.

**Tabella 2.8** La precedenza degli operatori C.

| | |
|---|---|
| **Alta** | () [] -> . |
| | ! ~ ++ -- - (tipo) * & sizeof |
| | * / % |
| | + - |
| | << >> |
| | < <= > >= |
| | == != |
| | & |
| | ^ |
| | \| |
| | && |
| | \|\| |
| | ?: |
| | = += -= *= /= ecc. |
| **Bassa** | , |



### Ordine di valutazione

Né il C né il C++ specificano l'ordine in cui devono essere valutate le sottoespressioni di un'espressione. Questo lascia il compilatore libero di disporre un'espressione in modo da produrre codice il più possibile ottimizzato. Tuttavia, questo significa anche che il codice prodotto non deve mai fare affidamento sull'ordine di valutazione delle sottoespressioni. Ad esempio nell'espressione

```
x = f1() + f2();
```

non si può essere certi che f1() venga richiamata prima di f2().

### Conversioni di tipo nelle espressioni

Quando in un'espressione si utilizzano costanti e variabili di tipi diversi, tutti i valori vengono convertiti nello stesso tipo. Il compilatore converte tutti gli operandi in modo che assumano lo stesso tipo dell'operando più esteso; questa operazione è chiamata *promozione di tipo*. Questo significa che tutti i char e gli short int vengono convertiti automaticamente in int. Questo processo è chiamato *promozione intera*. Terminata questa fase, tutte le altre conversioni vengono eseguite operazione per operazione secondo quanto descritto nel seguente algoritmo per la conversione dei tipi:

SE un operando è un long double
ALLORA il secondo è convertito in long double
ALTRIMENTI SE un operando è un double
ALLORA il secondo è convertito in double
ALTRIMENTI SE un operando è un float
ALLORA il secondo è convertito in float
ALTRIMENTI SE un operando è un unsigned long
ALLORA il secondo è convertito in unsigned long
ALTRIMENTI SE un operando è un long
ALLORA il secondo è convertito in long
ALTRIMENTI SE un operando è un unsigned int
ALLORA il secondo è convertito in unsigned int

Vi è anche un caso speciale: se un operando è long e l'altro è unsigned int e se il valore del unsigned int non può essere rappresentato da un long, entrambi gli operandi vengono convertiti in un unsigned long.

Al termine di queste conversioni, tutte le coppie di operandi saranno dello stesso tipo e il risultato di ogni operazione sarà dello stesso tipo di entrambi gli operandi.

Ad esempio, si considerino le conversioni di tipo che avvengono nella Figura 2.2. Innanzi tutto, il carattere ch viene convertito in un intero. Quindi, il risultato di ch / i viene convertito in un double poiché f*d è un double. Il risultato di f+i è un float poiché f è un float. Il risultato finale è double

## Conversioni cast

Questo tipo di conversione costringe un'espressione ad assumere un determinato tipo. La forma generale di conversione cast è:

*(tipo) espressione*

dove *tipo* è un tipo valido per il C. Ad esempio, per essere sicuri che l'espressione x / 2 fornisca un risultato di tipo float, si deve usare l'espressione

```
(float)x / 2
```

Le conversioni cast corrispondono tecnicamente ad operatori. Come operatore il cast è unario ed ha la stessa precedenza di ogni altro operatore unario.

Anche se le conversioni cast non sono fra le operazioni più utilizzate in programmazione, possono, in alcuni casi rivelarsi molto utili. Ad esempio, si sup-

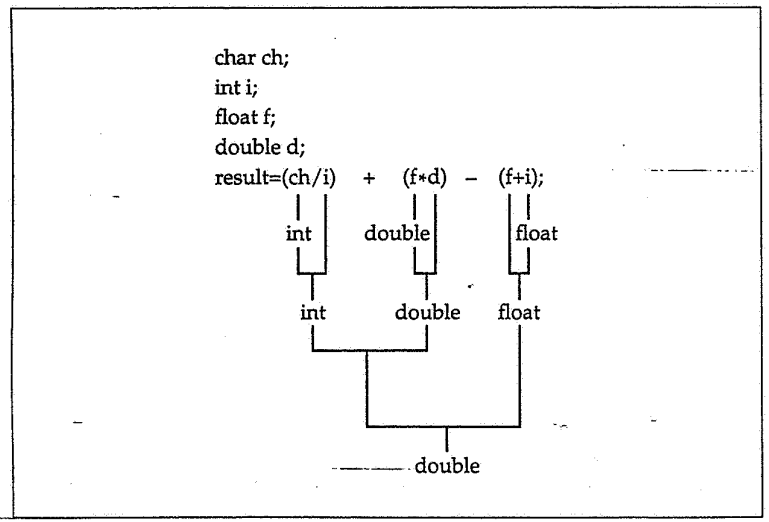



Figura 2.2 Un esempio di conversione di tipo.



ponga di voler usare un intero per controllare un ciclo e di dover eseguire un'operazione che richiede una parte frazionaria, come nel seguente programma:

```
#include <stdio.h>

int main(void) /* stampa i e i/2 con frazioni */
{
  int i;

  for(i=1; i<=100; ++i)
    printf("%d / 2 è uguale a: %f\n", i, (float) i /2);

  return 0;
}
```

Senza la conversione (float), verrebbe eseguita una divisione intera. La conversione cast assicura che venga sempre visualizzata anche la parte frazionale della risposta.

NOTA *Il C++ definisce alcuni operatori di conversione cast aggiuntivi (come ad esempio const_cast e static_cast) che verranno però discussi dettagliatamente nella seconda parte di questa guida*

## Spaziatura e parentesi

Per semplificare la lettura delle espressioni si possono utilizzare caratteri di tabulazione e spazi. Ad esempio, le due espressioni seguenti sono identiche:

```
x=10/y - (127/x);

x = 10 / y - (127/x);
```

Eventuali parentesi in eccesso non provocano alcun errore né rallentano l'esecuzione dell'espressione. Quindi si possono usare le parentesi per rendere più chiaro l'ordine di valutazione esatto di un'espressione, sia per se stessi che per gli altri. Ad esempio, quale delle due espressioni seguenti è più facile da leggere?

```
x=y/2-34*temp&127;
x = (y/3) - ((34*temp) & 127);
```

### Abbreviazioni C

Una variante dell'istruzione di assegnamento semplifica la codifica di determinate operazioni di assegnamento. Ad esempio,

```
x = x+10;
```

può essere scritta come

```
x += 10;
```

L'operatore += chiede al compilatore di assegnare a x il valore di x più 10.

Questa abbreviazione funziona con tutti gli operatori binari (gli operatori che richiedono due operandi). In generale, le istruzioni come:

*var = var operatore espressione*

può essere riscritta come

*var operatore= espressione*

Ad esempio,

```
x = x-100;
```

può essere scritta come

```
x -= 100;
```

La notazione abbreviata è ampiamente utilizzata nei programmi C/C++ scritti dai professionisti; quindi è molto importante conoscerne bene l'uso.

Capitolo 3

# Le istruzioni


3.1 **La verità e la falsità in C e C++**
3.2 **Le istruzioni di selezione**
3.3 **Le istruzioni di iterazione**
3.4 **La dichiarazione di variabili nelle istruzioni di selezione e iterazione**
3.5 **Le istruzioni di salto**
3.6 **Le espressioni**
3.7 **I blocchi**


Questo capitolo si occupa delle istruzioni. In senso generale, un'istruzione è un elemento eseguibile del programma; quindi un'istruzione specifica un'azione. Il C/C++ raggruppa le istruzioni nei seguenti gruppi:

- istruzioni di selezione
- istruzioni di iterazione
- istruzioni di salto
- etichette
- espressioni
- blocchi

Fra le istruzioni di selezione vi sono if e switch (al posto di "istruzione di selezione" viene spesso utilizzato il termine "istruzione condizionale"). Le istruzioni di iterazione sono while, for e do-while. Queste vengono normalmente chiamate *istruzioni di ciclo*. Le istruzioni di salto sono break, continue, goto e return. Le istruzioni a etichette includono case e default (discusse insieme all'istruzione switch) e l'istruzione di etichetta (discussa con goto). Le istruzioni di espressione sono istruzioni formate di una espressione valida in C. Le istruzioni a blocco sono formate semplicemente da blocchi di codice (come si ricorderà un blocco inizia con una { e termina con una }). Si fa riferimento ai blocchi come a *istruzioni composte*.

**NOTA** *Il C++ aggiunge anche due nuovi tipi di istruzioni: il blocco try (per la gestione delle eccezioni) e l'istruzione di dichiarazione che verranno discussi nella Parte seconda.*

Poiché molte espressioni C usano il risultato di alcuni test condizionali, si può iniziare parlando del concetto di verità e falsità.

## 3.1 La verità e la falsità in C e C++

Molte istruzioni C si basano su un'espressione condizionale che determina l'azione che deve essere intrapresa. Un'espressione condizionale può fornire un risultato vero o falso. In C il valore vero corrisponde a qualsiasi valore diverso da zero, inclusi i numeri negativi. Il valore falso corrisponde allo 0. Questo approccio alla verità e alla falsità consente di codificare un'ampia gamma di routine in modo estremamente efficiente. Il linguaggio C++ supporta la definizione di true e false (zero / non-zero) appena descritta. Ma il C++ definisce anche un tipo di dati booleano chiamato bool che può assumere i soli valori true e false. Come si è detto nel Capitolo 2, in C++ il valore 0 viene automaticamente convertito in false e un valore diverso da zero viene automaticamente convertito in true. Vale anche il contrario: true viene convertito in 1 e false viene convertito in 0. In C++, l'espressione che controlla un'istruzione condizionale è tecnicamente di tipo bool. Ma poiché ogni valore diverso da 0 viene convertito in true e il valore 0 viene convertito in false, sotto questo punto di vista non esistono differenze pratiche fra C e C++.

## 3.2 Le istruzioni di selezione

Il C/C++ consente di utilizzare due istruzioni di selezione: if e switch. In alcune circostanze, in alternativa a if si può utilizzare l'operatore ?.

### L'istruzione if

La forma generale dell'istruzione if è la seguente:

if(*espressione*) *istruzione*;
else *istruzione*;

dove *istruzione* può essere una singola istruzione, un blocco di istruzioni o nulla (in questo caso si parla di istruzioni vuote). La clausola else è opzionale.

Se *espressione* fornisce un risultato vero (diverso da zero), viene eseguita l'istruzione o il blocco relativo all'if; in caso contrario, verrà eseguita, se esiste, l'istruzione (o il blocco) relativa all'else. Occorre ricordare che viene eseguito solo il codice associato all'if oppure il codice associato all'else, mai entrambi.

In C, l'istruzione condizionale che controlla l'if deve produrre un risultato scalare. Uno *scalare* può essere un intero, un carattere, un puntatore o un numero in virgola mobile. In C++ può anche essere di tipo bool. È raro l'uso di un numero in virgola mobile per controllare un'istruzione condizionale in quanto questo rallenta considerevolmente il tempo di esecuzione (per eseguire un'operazione in virgola mobile occorrono molte più istruzioni rispetto alla corrispondente operazione con un intero o un carattere).

Il seguente programma contiene un esempio di if. Il programma presenta una versione molto semplice del gioco "indovina il numero magico". Quando il giocatore indovina il numero, visualizza il messaggio ** Corretto **. Per generare il numero magico, il programma utilizza il generatore di numeri casuali rand(), che restituisce un numero arbitrario compreso fra 0 e RAND_MAX (che definisce un valore intero maggiore o uguale a 32767). La funzione rand() richiede l'uso del file header stdlib.h. Un programma C++ può anche usare il nuovo file header <cstdlib>.

```
/* Programma del numero magico - Versione 1. */
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
  int magic; /* numero magico */
  int guess; /* valore immesso dall'utente */

  magic = rand(); /* genera il numero magico */

  printf("Indovina il numero magico: ");
  scanf("%d", &guess);

  if(guess == magic) printf("** Corretto **");

  return 0;
}
```

Sviluppando ulteriormente il programma del numero magico, la versione successiva illustra l'uso dell'istruzione else per stampare un messaggio nel caso in cui il numero sia errato.

```
/* Programma del numero magico - Versione 2. */
#include <stdio.h>
#include <stdlib.h>

int main(void)
```

```
{
  int magic; /* numero magico */
  int guess; /* valore immesso dall'utente */

  magic = rand(); /* genera il numero magico */

  printf("Indovina il numero magico: ");
  scanf("%d", &guess);

  if(guess == magic) printf("** Corretto **");
  else printf("Errato");

  return 0;
}
```

## If nidificati

Un if nidificato è un if che si trova all'interno di un altro if o di un else. Gli if nidificati sono molto comuni in programmazione. In un if nidificato, un istruzione else fa sempre riferimento all'istruzione if più vicina che si trova all'interno dello stesso blocco e alla quale non sia associato nessun altro else. Ad esempio,

```
if(i)
{
  if(j) istruzione 1;
  if(k) istruzione 2; /* questo if */
  else  istruzione 3; /* è associato a questo else */
}
else istruzione 4; /* associato a if(i) */
```

Come è indicato nel listato, l'else finale non è associato a if(j) poiché non si trova nello stesso blocco. Infatti, l'else finale è associato all'istruzione if(i). Inoltre, l'else interno è associato a if(k), che è l'if più vicino.

Lo Standard C specifica che debbano essere consentiti almeno 15 livelli di nidificazione. In pratica, la maggior parte dei compilatori consente un numero di livelli molto maggiore. Ma soprattutto, lo Standard C++ prevede l'uso in un programma C++ di un massimo di 256 livelli di if nidificati. In ogni caso, è consigliabile contenere il più possibile il livello di nidificazione per evitare di confondere il significato di un algoritmo.

L'uso di if nidificati consente di migliorare ulteriormente il programma del numero magico, fornendo al giocatore ulteriori informazioni in caso di errore.



```
/* Programma del numero magico - Versione 3. */
#include <stdio.h>
#include <stdlib.h>


int main(void)
{
  int magic; /* numero magico */
  int guess; /* valore immesso dall'utente */

  magic = rand(); /* genera il numero magico */

  printf("Indovina il numero magico: ");
  scanf("%d", &guess);

  if (guess == magic) {
    printf("** Corretto **");
    printf("Il numero magico è: %d\n", magic);
  }
  else {
    printf("Errato, ");
    if(guess > magic) printf("troppo alto\n");
    else printf("troppo basso\n");
  }

  return 0;
}
```

## Il costrutto if-else-if

Si tratta di un costrutto molto comune in programmazione; la sua forma generale è:

```
if (espressione)istruzione;
else
 if (espressione)istruzione;
 else
  if (espressione)istruzione;
  .
  .
  .
  else istruzione;
```

Le condizioni vengono valutate dall'alto verso il basso. Quando viene trovata una condizione vera, viene eseguita l'*istruzione* associata e la parte rimanente del costrutto viene ignorata. Se nessuna delle condizioni è vera viene eseguita l'*istruzione* associata all'else finale. Se l'else finale non è presente, quando tutte le condizioni sono false non viene eseguita alcuna istruzione.

Anche se lo schema di indentazione utilizzato precedentemente per il costrutto if-else-if è tecnicamente corretto, può portare a rientri troppo ingenti. Per questo motivo, il costrutto if-else-if viene normalmente indentato nel seguente modo:

```
if (espressione)
  istruzione;
else if (espressione)
  istruzione;
else if (espressione)
  istruzione;
.
.
.
else
  istruzione;
```

Utilizzando un costrutto if-else-if, il programma del numero magico diviene:

```
/* Programma del numero magico - Versione 4. */
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
  int magic; /* numero magico */
  int guess; /* valore immesso dall'utente */

  magic = rand(); /* genera il numero magico */

  printf("Indovina il numero magico: ");
  scanf("%d", &guess);

  if(guess == magic) {
    printf("** Corretto ** ");
    printf("Il numero magico è: %d", magic);
  }
  else if(guess > magic)
    printf("Errato, troppo alto");
  else printf("Errato, troppo basso");

  return 0;
}
```



## L'alternativa: l'operatore ?

L'operatore ? può sostituire le istruzioni if-else nella loro forma generale:

```
if (condizione) espressione
else espressione
```

Tuttavia, per poter utilizzare il punto interrogativo è necessario che le espressioni sia dell'if che dell'else siano una singola espressione e non un'altra istruzione.

Il ? è chiamato operatore ternario in quanto richiede tre operandi. La sua forma generale è:

```
Esp1 ? Esp2: Esp3
```

dove *Esp1*, *Esp2* e *Esp3* sono espressioni. Si noti l'uso e il posizionamento del segno di due punti.

Il valore di un'espressione ? è determinato nel modo seguente: innanzi tutto viene valutata *Esp1*. Se è vera, viene valutata *Esp2* che diverrà il valore dell'intera espressione ?. Se *Esp1* è falsa, allora viene valutata *Esp3* e il suo valore diviene il valore dell'intera espressione. Ad esempio, si considerino le seguenti istruzioni:

```
x = 10;
y = x>9 ? 100 : 200;
```

In questo esempio, a y viene assegnato il valore 100. Se x fosse stata minore di 9, ad y sarebbe stato assegnato il valore 200. Lo stesso codice scritto con un'istruzione if-else avrebbe avuto il seguente aspetto:

```
x = 10;
if(x>9) y = 100;
else y = 200;
```

Il seguente programma utilizza l'operatore ? per calcolare il quadrato di un numero intero immesso dall'utente. Tuttavia, questo programma conserva il segno (10 al quadrato darà 100 e -10 al quadrato darà -100).

```
#include <stdio.h>

int main(void)
{
  int isqrd, i;

  printf("Immettere un numero: ");

  scanf("%d", &i);

  isqrd = i>0 ? i*i : -(i*i);

  printf("%d al quadrato è uguale a %d", i, isqrd);

  return 0;
}
```

L'uso dell'operatore ? per sostituire le istruzioni if-else non si limita solo agli assegnamenti. Tutte le funzioni (tranne quelle dichiarate come void) possono restituire un valore. Pertanto, in un'espressione è possibile utilizzare una o più chiamate di funzioni. Quando si incontra il nome della funzione, essa viene eseguita in modo che possa restituire un valore. Pertanto, è possibile eseguire con un operatore ? una o più funzioni, inserendo le chiamate nelle espressioni che formano gli operandi di ?, come in:

```
#include <stdio.h>

int f1(int n);
int f2(void);

int main(void)
{
  int t;

  printf("Inserire un numero: ");
  scanf("%d", &t);

  /* stampa un messaggio appropriato */
  t ? f1(t) + f2() : printf("è stato immesso lo zero");

  return 0;
}

f1(int n)
{
```



```
  printf("%d ", n);
  return 0;
}

f2(void)
{
  printf("è il valore immesso");
  return 0;
}
```

Se in questo esempio si immette uno 0, verrà richiamata la funzione printf() che visualizza il messaggio è stato immesso lo zero. Se invece si immette un altro numero, verranno eseguite le funzioni f1() e f2(). In questo esempio il valore dell'espressione ? viene ignorato.

Alcuni compilatori C++ cercano di ridisporre l'ordine di valutazione delle espressioni per cercare di ottimizzare il codice oggetto. In questi casi, le funzioni che formano gli operandi dell'operatore ? possono essere eseguite in una sequenza errata.

Utilizzando l'operatore ? si può riscrivere il programma del numero magico nel seguente modo:

```
/* Programma del numero magico - Versione 5. */
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
  int magic;
  int guess;
  magic = rand(); /* genera il numero magico */
  printf("Indovina il numero magico: ");
  scanf("%d", &guess);

  if(Guess == magic) {
    printf("** Corretto ** ");
    printf("Il numero magico è: %d", magic);
  }
  else
    guess > magic ? printf("Alto") : printf("Basso");

  return 0;
}
```

Qui, l'operatore ? visualizza il messaggio corretto sulla base del risultato del test guess>magic.

## L'espressione condizionale

Talvolta, coloro che incontrano per la prima volta il C/C++ rimangono confusi dal fatto che è possibile controllare gli operatori if o ? utilizzando una qualsiasi espressione valida. Questo significa che non si è costretti a usare le sole espressioni riguardanti operatori relazionali o logici (come nel caso del BASIC o del Pascal). L'espressione deve semplicemente restituire un valore false o true (uguale o diverso da zero). Ad esempio, il seguente programma legge due interi dalla tastiera e visualizza il quoziente. Il programma utilizza un'istruzione if, controllata dal secondo numero per evitare l'errore di divisione per zero.

```
/* Divide il primo numero per il secondo. */

#include <stdio.h>

int main(void)
{
  int a, b;

  printf("Immettere due numeri: ");
  scanf("%d%d", &a, &b);

  if(b) printf("%d\n", a/b);
  else printf("Non è possibile dividere per zero.\n");

  return 0;
}
```

Questo approccio funziona poiché se b è uguale a 0, la condizione che controlla l'if è falsa e viene eseguita l'istruzione dell'else. In caso contrario, la condizione sarà vera (diversa da zero) e avrà quindi luogo la divisione. L'uso di un'istruzione if come la seguente:

```
if(b != 0) printf("%d\n", a/b);
```

è ridondante e potenzialmente inefficiente e inoltre è considerata un esempio di cattivo stile di programmazione. Poiché il valore di b è sufficiente per controllare l'if, non è necessario confrontarlo con lo zero.

## L'istruzione switch

Il C/C++ è dotato di un'istruzione di selezione a più opzioni, chiamata switch, che controlla in successione il valore di un'espressione confrontandolo con un elenco



di costanti intere o caratteri. Quando viene trovata una corrispondenza, vengono eseguite le istruzioni associate alla costante. La forma generale dell'istruzione switch è la seguente:

```
switch (espressione) {
  case costante1:
   sequenza istruzioni
   break;
  case costante2:
   sequenza istruzioni
   break;
  case costante3:
   sequenza istruzioni
   break;
  .
  .
  .
  default
   sequenza istruzioni
}
```

L'espressione deve fornire un valore costituito da un carattere o da un intero. Ad esempio non è consentito l'uso di espressioni in virgola mobile. Il valore di *espressione* viene confrontato, nell'ordine, con i valori delle costanti specificate nelle istruzioni case. Quando viene trovata una corrispondenza, viene eseguita la *sequenza istruzioni* associata al case e ciò fino alla successiva istruzione break o alla fine dello switch. L'istruzione default viene eseguita solo se non viene trovata alcuna corrispondenza. Il default è opzionale e, se assente, fa in modo che, nel caso in cui non venga trovata alcuna corrispondenza, non venga eseguita alcuna operazione.

Il C standard specifica che uno switch possa contenere almeno 257 istruzioni case. Il C++ standard suggerisce che il compilatore accetti *almeno* 16.384 istruzioni case. In pratica, per motivi di efficienza, è preferibile limitare il più possibile il numero di istruzioni case. Anche se case è un'istruzione di etichetta, non può esistere da sola, all'esterno di uno switch.

L'istruzione break è una delle istruzioni di salto del C/C++. È possibile utilizzarla in cicli e in istruzioni switch (vedere la sezione "Le istruzioni di iterazione"). Quando viene raggiunto un break in uno switch, l'esecuzione del programma "salta" alla riga di codice che segue l'istruzione switch.

Vi sono tre cose importanti da sapere sull'istruzione switch:

- Lo switch è diverso dal if poiché il primo esegue solo verifiche di uguaglianza mentre il secondo può valutare ogni tipo di espressione relazionale o logica.

- Non è possibile specificare due costanti case con valori identici nello stesso switch. Naturalmente, un'istruzione switch racchiusa in un'altra istruzione switch esterna può avere costanti case uguali allo switch esterno.
- Se in un'istruzione switch vengono utilizzate costanti di tipo carattere, esse verranno automaticamente convertite in interi.

L'istruzione switch è normalmente utilizzata per gestire i comandi alla tastiera, come ad esempio la scelta di opzioni da un menu. Come si può vedere nell'esempio seguente, la funzione menu() visualizza un menu per un programma di verifica ortografica e richiama le procedure appropriate:

```
void menu(void)
{
  char ch;

  printf("1. Verifica ortografica\n");
  printf("2. Correzione errori ortografici\n");
  printf("3. Visualizza errori ortografici\n");
  printf("Premere un altro tasto per uscire\n");
  printf("      Immettere la scelta: ");

  ch = getchar(); /* legge il tasto premuto */

  switch(ch) {
    case '1':
      check_spelling();
      break;
    case '2':
      correct_errors();
      break;
    case '3':
      display_errors();
      break;
    default :
      printf("Non è stata selezionata alcuna opzione");
  }
}
```

Tecnicamente, le istruzioni break che si trovano all'interno di un'istruzione switch sono opzionali. Esse concludono la sequenza di istruzioni associata ad ogni costante. Se si omettesse l'istruzione break, l'esecuzione continuerebbe nella successiva istruzione case e ciò fino a raggiungere il primo break o fino alla fine dello switch. Ad esempio, la seguente funzione utilizza questa caratteristica di case per semplificare il codice di un gestore di input:



```
/* Elabora un valore */
void inp_handler(int i)
{
  int flag;

  flag = -1;

  switch(i) {
    case 1:  /* questi case hanno sequenze di istruzioni */
    case 2:  /* comuni */
    case 3:
      flag = 0;
      break;
    case 4:
      flag = 1;
    case 5:
      error(flag);
      break;
    default:
      process(i);
  }
}
```

Questo esempio illustra due aspetti dello switch. Innanzi tutto, è possibile utilizzare istruzioni case senza alcuna istruzione associata. In questo caso, l'esecuzione salta semplicemente al successivo case. In questo esempio, i primi tre case eseguono le stesse istruzioni che sono

```
flag = 0;
break;
```

In secondo luogo, l'esecuzione di una sequenza di istruzioni continua nel case successivo finché non viene trovata un'istruzione break. Se i è uguale a 4, flag è impostato a 1 e, poiché non vi è alcuna istruzione break al termine di tale case l'esecuzione continua e richiama la funzione error(flag). Se i è uguale a 5, viene richiamata la funzione error(flag) con un valore di flag uguale a -1.

Il fatto che i case possano essere eseguiti insieme quando non si specifica un'istruzione break evita inutili duplicazioni di istruzioni, consentendo di produrre un codice più efficiente.

## Istruzioni switch nidificate

In C è possibile inserire uno switch come parte di una sequenza di istruzioni di uno switch più esterno. Anche se le costanti case degli switch contengono valori

comuni, questo non provocherà alcun conflitto. Ad esempio, il seguente frammento di codice è perfettamente corretto:

```
switch(x) {
  case 1:
    switch(y) {
      case 0: printf("Errore di divisione per zero. \n");
              break
      case 1: process(x,y);
    }
    break;
  case 2:
       .
       .
       .
```

## 3.3 Le istruzioni di iterazione

In C/C++ e in tutti gli altri linguaggi di programmazione moderni, le istruzioni di iterazione (chiamate anche istruzioni di ciclo) consentono la ripetizione di un gruppo di istruzioni fino al verificarsi di una determinata condizione. Questa condizione può essere predefinita (come ad esempio nel ciclo for) o aperta (come nel caso dei cicli while e do-while).

### Il ciclo for

La forma generale del ciclo for è presente, in una forma o nell'altra, in tutti i linguaggi di programmazione procedurali. Tuttavia, in C/C++, presenta una flessibilità, e quindi una potenza, molto maggiore.

La forma generale dell'istruzione for è la seguente:

for (*inizializzazione* ; *condizione* ; *incremento*) *istruzione*;

Il ciclo for consente molte varianti ma la sua forma più comune funziona nel modo seguente. L'*inizializzazione* è normalmente un'istruzione di assegnamento utilizzata per impostare la variabile di controllo del ciclo. La *condizione* è un'espressione relazionale che determina l'uscita dal ciclo. L'*incremento* definisce il valore di cui la variabile di controllo deve variare ad ogni ripetizione del ciclo. Queste tre sezioni devono essere separate da un punto e virgola. Il ciclo for continua a essere ripetuto finché la *condizione* si mantiene vera. Quando la *condizione* diviene falsa, l'esecuzione del programma riprende dall'istruzione che segue il for.



Nel seguente programma, viene utilizzato un ciclo for per visualizzare i numeri da 1 a 100:

```
#include <stdio.h>

int main(void)
{
  int x;

  for(x=1; x <= 100; x++) printf("%d ", x);

  return 0;
}
```

Nel ciclo, alla variabile x viene inizialmente assegnato il valore 1 che viene poi confrontato con il 100. Poiché x è minore di 100, viene richiamata la funzione printf() e si continua nel ciclo. La variabile x viene aumentata di una unità e nuovamente verificata per determinare se è ancora minore o uguale a 100. In caso affermativo, viene nuovamente richiamata printf(). Questo processo si ripete finché x non diviene maggiore di 100, condizione di uscita dal ciclo. In questo esempio, x è la variabile di controllo del ciclo, che viene modificata e verificata ogni volta che il ciclo viene ripetuto.

Il seguente esempio è un ciclo for che ripete l'esecuzione di più istruzioni:

```
for(x=100; x != 65; x-=5 ) {
  z = x*x;
  printf("Il quadrato di %d è %f", x, z);
}
```

Sia il quadrato di x che la chiamata a printf() vengono eseguiti fintantoché x non diviene uguale a 65. Come si può notare, il ciclo procede in senso negativo: x è inizializzata a 100 e ad ogni ripetizione del ciclo viene sottratto il valore 5.

Nei cicli for, il test condizionale viene sempre eseguito all'inizio del ciclo. Questo significa che il codice all'interno del ciclo potrebbe anche non venir mai eseguito se la condizione dovesse essere falsa fin dall'inizio. Ad esempio, nel frammento di codice:

```
for(y=10; y!=x; ++y) printf("%d", y);
printf("%d", y);  /* questa è l'unica istruzione printf() che verrà eseguita
*/
```

il ciclo non verrà mai eseguito poiché all'ingresso del ciclo x e y sono uguali. Per questo motivo, l'espressione condizionale fornirà il valore falso e non verranno

eseguiti né il corpo del ciclo né la porzione di incremento. Pertanto, y avrà ancora il valore 10 e l'unico output prodotto da questo frammento di programma sarà una sola visualizzazione del numero 10.

## Le varianti del ciclo for

La discussione precedente, si occupava della forma più comune del ciclo for. Tuttavia vi sono molte varianti che aumentano la potenza, la flessibilità e l'applicabilità del ciclo for in alcune situazioni.

Una delle varianti più comuni prevede l'uso dell'operatore virgola che consente di controllare il ciclo con due o più variabili (come si ricorderà, l'operatore virgola consente di concatenare una serie di espressioni assumendo il significato di "fai questo e questo"; vedere il Capitolo 2). Ad esempio, il seguente ciclo è controllato dalle variabili x e y ed entrambe vengono inizializzate all'interno dell'istruzione for:

```
for(x=0, y=0; x+y<10; ++x) {
  y = getchar();
  y = y-'0'; /* sottrae a y
              il codice ASCII di 0 */
    .
    .
    .
}
```

Le due istruzioni di inizializzazione sono separate da una virgola. Ad ogni ripetizione del ciclo, viene incrementata x e il valore di y è determinato da ciò che viene immesso alla tastiera. Il ciclo ha termine quando sia x che y hanno raggiunto il valore corretto. Anche se il valore di y viene impostato tramite una lettura della tastiera, tale variabile deve essere inizializzata a 0 in modo che il suo valore sia definito prima della prima valutazione dell'espressione condizionale (se y non fosse definita, potrebbe anche contenere il valore 10, rendendo il test condizionale falso fin dall'inizio e saltando così l'esecuzione del ciclo).

La funzione converge() mostrata di seguito mostra l'uso di più variabili di controllo in un solo ciclo. La funzione converge() copia una stringa in un'altra iniziando a copiare i caratteri dalle estremità e convergendo verso il centro.

```
/* Uso di più variabili di controllo nei cicli. */
#include <stdio.h>
#include <string.h>

void converge(char *targ, char *src);
```



```
int main(void)
{
  char target[80] = "XXXXXXXXXXXXXXXXXXXXXXXXXXXXXX";

  converge(target, "Prova dell'uso di converge().");
  printf("Stringa finale: %s", target);

  return 0;
}

/* Questa funzione copia una stringa in un'altra
   copiando I caratteri da entrambe le estremità
   e convergendo al centro. */

void converge(char *targ, char *src)
{
  int i, j;

  printf("%s", targ);
  for(i=0, j=strlen(src); i<j; i++, j--) <R    targ[i] = src[i];
    targ[j] = src[j];
    printf("%s", targ);
  }
}
```

Ecco l'output prodotto dal programma.

```
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
PXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
PrXXXXXXXXXXXXXXXXXXXXXXXXXXX.
ProXXXXXXXXXXXXXXXXXXXXXXXXX).
ProvXXXXXXXXXXXXXXXXXXXXXX().
ProvaXXXXXXXXXXXXXXXXXXXXe().
Prova XXXXXXXXXXXXXXXXXge().
Prova dXXXXXXXXXXXXXXXrge().
Prova deXXXXXXXXXXXXXerge().
Prova delXXXXXXXXXXXverge().
Prova dellXXXXXXXXXnverge().
Prova dell'XXXXXXXonverge().
Prova dell'uXXXXXconverge().
Prova dell'usXXXX converge().
Prova dell'usoXXi converge().
Prova dell'uso di converge().
Stringa finale: Prova dell'uso di converge().
```

In converge(), il ciclo for utilizza due variabili di controllo, i e j, che fungono da indici della stringa alle sue estremità opposte. Durante l'iterazione del ciclo, i aumenta e j decrementa. Il ciclo termina quando i è maggiore di j, ovvero quando sono stati copiati tutti i caratteri.

L'espressione condizionale non deve quindi confrontare il valore della variabile di controllo del ciclo con un altro valore. Infatti, la condizione può essere un'istruzione relazionale o logica.

Questo significa che è possibile utilizzare vari tipi di condizioni di uscita. Ad esempio, per collegare un utente a un sistema remoto si potrebbe utilizzare la seguente funzione. L'utente ha tre tentativi per immettere la password. Il ciclo termina quando si esauriscono i tre tentativi oppure quando l'utente immette la password corretta.

```
void sign_on(void)
{
  char str[20];

  int x;

  for(x=0; x<3 && strcmp(str, "password"); ++x) {
    printf("Immettere la password:");
    gets(str);
  }

  if(x==3) return;
  /* altrimenti collega l'utente ... */
}
```

Questa funzione utilizza strcmp(), la funzione della libreria standard che confronta due stringhe e restituisce zero se le stringhe sono uguali.

Occorre ricordare che ognuna delle tre sezioni del ciclo for può essere formata da una qualsiasi espressione valida. Non necessariamente le espressioni devono avere a che fare con l'uso che generalmente si fa delle relative sezioni. Ad esempio, si consideri il seguente listato:

```
#include <stdio.h>

int sqrnum(int num);
int readnum(void);
int prompt(void);

int main(void)
{
  int t;

  for(prompt(); t=readnum(); prompt())
    sqrnum(t);

  return 0;
}

int prompt(void)
{
  printf("Immettere un numero ");
  return 0;
}

int readnum(void)
{
  int t;


  scanf("%d", &t);
  return t;
}

int sqrnum(int num)
{
  printf("%d\n", num*num);
  return num*num;
}
```



Osservando attentamente il ciclo for in main(), si noterà che ogni parte del ciclo è formata da chiamate a funzioni che visualizzano un messaggio per l'utente e leggono un numero immesso alla tastiera. Se il numero immesso è 0, il ciclo termina in quanto l'espressione condizionale sarà falsa. In caso contrario, il numero viene elevato al quadrato. Pertanto, questo ciclo for utilizza le porzioni di inizializzazione e di incremento in senso non tradizionale ma assolutamente corretto.

Un altra caratteristica interessante del ciclo for è il fatto che non richiede la presenza di tutte le sue componenti. Addirittura, potrebbe non esservi alcuna espressione in alcuna delle sue componenti, ovvero le espressioni sono opzionali. Ad esempio, questo ciclo continua finché l'utente non immette il valore 123:

```
for(x=0; x!=123; ) scanf("%d", &x);
```

Come si può notare, la porzione di incremento nella definizione del for è vuota. Questo significa che ogni volta che il ciclo viene ripetuto, viene controllato il valore di x per vedere se è uguale a 123, ma non viene eseguita nessun'altra ope-

razione. Se si immette 123 alla tastiera, la condizione del ciclo diviene falsa e il ciclo ha quindi termine.

L'inizializzazione della variabile di controllo del ciclo può verificarsi all'esterno del ciclo for. Questo avviene frequentemente quando la condizione iniziale della variabile di controllo del ciclo deve essere calcolata in modo piuttosto complesso, come nell'esempio seguente:

```
if(*s) x = strlen(s); /* calcola la lunghezza della stringa */
else x = 10;

for( ; x<10; ) {
  printf("%d", x);
  ++x;
}
```

La sezione di inizializzazione è stata lasciata vuota e x viene inizializzata prima dell'ingresso nel ciclo.

## I cicli infiniti

Anche se è possibile creare un ciclo infinito con qualsiasi istruzione di ciclo, normalmente si utilizza un ciclo for. Poiché nessuna delle tre espressioni che formano il ciclo for è assolutamente necessaria, è possibile creare un ciclo infinito lasciando vuota l'espressione condizionale, come nell'esempio seguente:

```
for( ; ; ) printf(" Ciclo infinito.\n");
```

Quando l'istruzione condizionale è assente, si assume che questa sia vera. Si può specificare un'espressione di inizializzazione e una di incremento, ma frequentemente i programmatori C creano cicli infiniti utilizzando il costrutto for(;;).

In effetti, il costrutto for(;;) non garantisce che il ciclo sia veramente infinito, a causa dell'istruzione break che, se si trova all'interno di un corpo di un ciclo, ne causa immediatamente l'uscita (l'istruzione break verrà discussa in seguito in questo capitolo). Il controllo del programma riprenderà dal codice che segue il ciclo, come nell'esempio seguente:

```
ch = ' \0';

for( ; ; ) {
  ch = getchar(); /* legge un carattere */
  if(ch=='A') break; /* esce dal ciclo */
}
```



```
printf("è stata immessa una A");
```

Questo ciclo continuerà ad essere eseguito finché l'utente non immetterà una A alla tastiera.

## I cicli for senza corpo

Le istruzioni possono anche essere vuote. Questo significa che anche il corpo di un ciclo for (o di un qualunque altro ciclo) può essere vuoto. È possibile utilizzare questo fatto per migliorare l'efficienza di alcuni algoritmi e per creare cicli di ritardo.

La rimozione di spazi da un canale di input è un'operazione piuttosto comune. Ad esempio, un programma di database potrebbe consentire la generazione di interrogazioni come: "mostra tutti i valori minori di 400". Il database richiede che ogni parola venga inviata separatamente, senza spazi. Questo significa che il gestore dell'input del database riconosce "mostra" ma non " mostra". Il ciclo seguente mostra come ottenere questo risultato oltrepassando gli spazi contenuti nella stringa puntata da str.

```
for( ; *str == ' '; str++) ;
```

Come si può vedere, in questo ciclo il corpo è assente, semplicemente perché è inutile.

I cicli per *l'introduzione di ritardi* sono molto utilizzati nei programmi. Il seguente codice mostra la creazione di un ciclo di ritardo utilizzando un for:

```
for(t=0; t<VALORE; t++) ;
```

## Il ciclo while

Il secondo tipo di ciclo disponibile in C è il ciclo while. La sua forma generale è:

while (*condizione*) *istruzione*;

dove *istruzione* può essere un'istruzione vuota, una singola istruzione oppure un blocco di istruzioni. La *condizione* può essere una qualsiasi espressione e, come si ricorderà, la verità dell'espressione è data da un valore diverso da zero. Il ciclo itera mentre la condizione permane vera. Quando la condizione diviene falsa, il controllo del programma passa alla riga di codice che segue il ciclo.

Il seguente esempio mostra una routine di input da tastiera che continua il ciclo finché l'utente non immette una A:

```
wait_for_char(void)
{
  char ch;

  ch = ' \0';  /* inizializza ch */
  while(ch != 'A') ch = getchar();
  return ch;
}
```

Innanzi tutto, a ch viene assegnata la stringa nulla. Come variabile locale, il suo valore non è noto quando inizia l'esecuzione di wait_for_char(). Il ciclo while controlla che ch sia diversa da A. Poiché ch è stata inizializzata con la stringa nulla, il test è vero e il ciclo ha inizio. La condizione viene nuovamente verificata ogni volta che si preme un tasto. Quando si immette una A, la condizione diviene falsa poiché ch è uguale ad A e il ciclo ha termine.

Come nel caso dei cicli for, anche il ciclo while verifica la condizione di test all'inizio; questo significa che il corpo del ciclo non viene eseguito se la condizione iniziale è falsa. Questa caratteristica elimina la necessità di eseguire un test condizionale distinto prima del ciclo. La funzione pad() è un ottimo esempio di ciò. Essa aggiunge spazi alla fine di una stringa in modo da completare una stringa di lunghezza predefinita. Se la stringa è già della lunghezza desiderata, non viene aggiunto alcuno spazio.

```
#include <stdio.h>
#include <string.h>

void pad(char *s, int length);

int main(void)
{
  char str[80];

  strcpy(str, "stringa di prova");
  pad(str, 40);
  printf("%d", strlen(str));

  return 0;
}

/* Aggiunge spazi alla fine della stringa. */
void pad(char *s, int length)
{
  int l;

  l = strlen(s);  /* determina la lunghezza della stringa */

  while(l<length) {
    s[l] = ' ';   /* inserisce uno spazio */
    l++;
  }
  s[l]= '\0';     /* le stringhe sono terminate
                     da un carattere nullo */
}
```



I due argomenti di pad() sono s, un puntatore alla stringa da allungare e length, il numero di caratteri che s dovrà avere. Se la lunghezza della stringa s è già uguale o maggiore di length, il codice del ciclo while non verrà mai eseguito. Se s è più breve di length, pad() aggiunge il numero necessario di spazi. Per conoscere la lunghezza della stringa si utilizza la funzione strlen() che si trova nella libreria standard.

Se un ciclo while può essere terminato da più condizioni distinte, l'espressione condizionale sarà formata da una singola variabile. Il valore di tale variabile verrà impostato in vari punti del ciclo. In questo esempio:

```
void func1(void)
{
  int working;

  working = 1; /* Vero */

  while(working) {
    working = process1();
    if(working)
      working = process2();
    if(working)
      working = process3();
  }
}
```

l'uscita dal ciclo può essere causata da una qualsiasi delle tre routine, se dovesse restituire il valore falso.

Non vi è nemmeno nessuna necessità di inserire istruzioni nel corpo di un ciclo while. Ad esempio,

```
while((ch=getchar())!='A') ;
```

continua il ciclo finché l'utente non immette la lettera A. Se l'utente trova scomodo l'inserimento dell'assegnamento all'interno dell'espressione condizionale del while, si ricordi che il segno di uguaglianza non è che un operatore che valuta il valore dell'operando posto alla sua destra.

### Il ciclo do-while

A differenza dei cicli for e while, che controllano la condizione del ciclo all'inizio, il costrutto do-while verifica tale condizione al termine del ciclo. Questo significa che un ciclo do-while viene sempre eseguito almeno una volta. La forma generale di un ciclo do-while è la seguente:

```
do{
  istruzione;
} while(condizione);
```

Anche se le parentesi graffe non sono necessarie quando si utilizza una sola istruzione, sono normalmente utilizzate per evitare confusione (per il programmatore, non per il compilatore). Il ciclo do-while continua a ripetersi finché la *condizione* non diviene falsa.

Il seguente ciclo do-while legge numeri dalla tastiera fino a trovare un numero minore o uguale a 100.

```
do {
   scanf("%d", &num);
} while(num > 100);
```

Probabilmente, l'uso più comune dei cicli do-while è nella selezione delle opzioni di un menu. Quando l'utente immette una risposta valida, essa viene restituita come valore della funzione. Le risposte errate ripresentano il messaggio di richiesta. Il seguente codice mostra una versione migliorata del menu per il correttore ortografico sviluppato precedentemente in questo Capitolo:

```
void menu(void)
{
  char ch;

  printf("1. Verifica ortografica\n");
  printf("2. Correzione errori ortografici\n");
  printf("3. Visualizza errori ortografici\n");
  printf("      Immettere la scelta: ");
  do {
    ch = getchar(); /* legge la scelta
                       dalla tastiera */
    switch(ch) {
      case '1':
        check_spelling();
        break;
      case '2':
        correct_errors();
        break;
      case '3':
        display_errors();
        break;
    }
  } while(ch!='1' && ch!='2' && ch!='3');
}
```



Qui, il ciclo do-while è un'ottima scelta, in quanto si desidera che la funzione del menu venga sempre eseguita almeno una volta. Dopo la visualizzazione delle opzioni, il programma continua il ciclo finché non viene selezionata un'opzione valida.

## 3.4 La dichiarazione di variabili nelle istruzioni di selezione e iterazione

In C++ (ma non in C) è possibile dichiarare una variabile nell'espressione condizionale di un if o di uno switch, nell'espressione condizionale di un ciclo while o nella parte di inizializzazione di un ciclo for. Una variabile dichiarata in uno di questi luoghi ha un campo di visibilità limitato al blocco di codice controllato da tale istruzione. Ad esempio, una variabile dichiarata in un ciclo for sarà una variabile locale di tale ciclo.

Ecco un esempio che dichiara una variabile nella porzione di inizializzazione di un ciclo for:

```
/* i è visibile all'interno del ciclo; j non è visibile all'esterno del
ciclo. */
int j;
for(int i = 0; i<10; i++)
  j = i * i;

/* i = 10; // *** Errore *** -- i non è visibile! */
```

Qui, i viene dichiarata nella porzione di inizializzazione del for e viene utilizzata per controllare il ciclo. All'esterno del ciclo, i è sconosciuta.

Poiché spesso la variabile di controllo di un ciclo for è utilizzata solo all'interno di tale ciclo, sta diventando pratica comune dichiarare la variabile nella porzione di inizializzazione del for. Si deve solo ricordare che tale funzionalità non è supportata dal linguaggio C.

**SUGGERIMENTO** *Il fatto che una variabile dichiarata nella porzione di inizializzazione di un ciclo for sia locale di tale ciclo è un concetto che si è modificato nel tempo. Originariamente la variabile risultava disponibile dal ciclo for in avanti. Tuttavia lo standard C++ ha ristretto il campo di visibilità di tale variabile al solo ciclo for.*

Se il compilatore aderisce completamente allo standard C++, si può dichiarare una variabile all'interno di qualsiasi espressione condizionale, come quelle di un if o di un while. Ad esempio, il seguente frammento di codice:

```
if(int x = 20) {
  x = x - y;
  if(x>10) y = 0;
}
```

dichiara x e le assegna il valore 20. Poiché questo è un valore che viene valutato true, la condizione dell'if dà esito positivo. Le variabili dichiarate in un'istruzione condizionale hanno il loro campo visibilità limitato al blocco di codice controllato da tale istruzione. Pertanto, in questo caso la variabile x non risulta nota all'esterno dell'if. Onestamente, non tutti programmatori credono sia opportuno dichiarare le variabili all'interno delle istruzioni condizionali e dunque questa tecnica non verrà impiegata in questo volume.

## 3.5 Le istruzioni di salto

Il C/C++ è dotato di quattro istruzioni che eseguono salti incondizionati: return, goto, break e continue. Di queste, return e goto possono essere utilizzate in qualsiasi punto del programma. Le istruzioni break e continue possono essere utilizzate insieme a una qualsiasi istruzione di ciclo. Come si è visto precedentemente in questo stesso Capitolo, si può utilizzare break anche in uno switch.



### L'istruzione return

L'istruzione return consente di uscire da una funzione. Si trova raggruppata insieme alle istruzioni di salto poiché provoca un salto dell'esecuzione al punto in cui era stata eseguita la chiamata alla funzione. A un return può essere associato o meno un valore. Se a un return viene associato un valore, questo diverrà il valore restituito dalla funzione. In C, non è tecnicamente necessario che una funzione non void restituisca un valore. Al contrario, in C++ una funzione non void *deve* restituire un valore. Ovvero, in C++, se una funzione è dichiarata in modo da restituire un valore, deve contenere un'istruzione return alla quale sia associato un valore (anche in C, se una funzione è dichiarata in modo da restituire un valore, è bene che restituisca effettivamente un valore).

La forma generale dell'istruzione return è la seguente:

return *espressione*;

La parte *espressione* deve essere presente solo se la funzione è dichiarata in modo da restituire un valore. Se presente, *espressione* diverrà il valore restituito dalla funzione.

In una funzione possono esservi tutte le istruzioni return desiderate ma la funzione terminerà l'esecuzione non appena incontra il primo return. La funzione ha termine anche quando incontra la parentesi graffa che ne chiude la definizione. In questo senso, la parentesi graffa chiusa corrisponde a un return senza associato alcun valore. Se questo si verifica in una funzione non void, il valore restituito dalla funzione sarà indefinito.

Una funzione dichiarata come void non può contenere un'istruzione return che specifica un valore (poiché una funzione void non restituisce alcun valore, non ha alcun senso che un return al suo interno restituisca un valore).

Per ulteriori informazioni sull'uso di return consultare il Capitolo 6.

### L'istruzione goto

Poiché il C/C++ è dotato di un'ampia gamma di strutture di controllo e consente un ulteriore controllo tramite break e continue, non vi è in genere alcuna necessità di utilizzare goto. La maggior parte dei programmatori non gradisce utilizzare i goto poiché tendono a rendere illeggibili i programmi. Ma, anche se l'istruzione goto ha goduto di scarsa popolarità fino a pochi anni fa, è stata recentemente rispolverata. Non vi è alcuna situazione di programmazione che richieda un goto. Si tratta piuttosto di un'istruzione comoda ma solo in ambiti molto ristretti, come ad esempio l'uscita da una serie di cicli molto nidificati.

L'istruzione goto richiede l'uso di un'etichetta (un'*etichetta* è un identificatore valido seguito da un due punti). Inoltre, l'etichetta deve trovarsi nella stessa funzione in cui si trova il goto (ovvero non è possibile saltare da una funzione a un'altra). La forma generale dell'istruzione goto è la seguente:

```
goto etichetta;
.
.
.
etichetta:
```

dove *etichetta* è un'etichetta valida che può trovarsi prima o dopo il goto. Ad esempio, è possibile creare un ciclo che conta i numeri da 1 a 100 usando un goto e un'etichetta:

```
x = 1;
loop1:
  x++;
  if(x<100) goto loop1;
```

## L'istruzione break

L'istruzione break può essere utilizzata in due casi. Può concludere un case in un'istruzione switch (di cui si parla nella sezione dedicata all'istruzione switch precedentemente in questo stesso capitolo). Si può utilizzare il break anche per causare l'immediata terminazione di un ciclo, saltando il test condizionale.

Quando si incontra un'istruzione break all'interno di un ciclo, il ciclo ha immediatamente termine e il controllo del programma riprende dall'istruzione che segue il ciclo. Ad esempio,

```
#include <stdio.h>

int main(void)
{
  int t;

  for(t=0; t<100; t++) {
    printf("%d ", t);
    if(t==10) break;
  }

  return 0;
}
```



stampa i numeri da 0 a 10. Poi il ciclo termina poiché un break causa l'immediata uscita dal ciclo, saltando il test condizionale t<100.

I programmatori utilizzano spesso l'istruzione break nei cicli in cui una determinata condizione può causare una terminazione immediata. Ad esempio, nel programma seguente la pressione di un tasto può concludere l'esecuzione della funzione look_up():

```
Void look_up(char *name)
{
  do {
     /* ricerca dei nomi ... */
     if(kbhit()) break;
  } while(!found);
  /* process match */
}
```

Se non si preme alcun tasto, la funzione kbhit() restituisce 0. Altrimenti, la funzione restituisce un valore diverso da 0. A causa delle grandi differenze fra diversi ambienti di calcolo, né il C standard né il C++ standard definiscono la funzione kbhit(), ma certamente il proprio compilatore ne prevede una (o una funzione con un nome leggermente diverso).

Un'istruzione break provoca l'uscita solo dal ciclo più interno. Ad esempio,

```
for(t=0; t<100; ++t) {
  count = 1;
  for(;;) {
    printf("%d ", count);
    count++;
    if(count==10) break;
  }
}
```

stampa i numeri da 1 a 10 per 100 volte. Ogni volta che l'esecuzione raggiunge un break, il controllo torna al ciclo for esterno.

Un break utilizzato in un'istruzione switch riguarda solo tale switch e non ha alcun effetto sul ciclo in cui lo switch può trovarsi.

## La funzione exit()

Anche se exit() non è un'istruzione per il controllo del programma, a questo punto è necessario introdurla. Come è possibile uscire con break da un ciclo, è anche possibile uscire da un programma utilizzando la funzione standard di libreria exit().

Questa funzione provoca l'immediata uscita dall'intero programma ed il ritorno al sistema operativo. In effetti, la funzione exit() opera come un break relativo all'intero programma.

La forma generale della funzione exit() è la seguente:

void exit(int *codice_di_uscita*);

Il valore di *codice_di_uscita* è restituito al processo chiamante, che normalmente è il sistema operativo. Una terminazione normale del programma viene normalmente segnalata dal codice di uscita 0. Altri argomenti vengono generalmente utilizzati per indicare una situazione di errore. Per il codice di uscita è possibile utilizzare anche le macro EXIT_SUCCESS ed EXIT_FAILURE. La funzione exit() richiede l'impiego del file header stdlib.h. Un programma C++ può utilizzare anche il nuovo file header <cstdlib>.

I programmatori utilizzano frequentemente la funzione exit() quando una delle condizioni di funzionamento del programma non è soddisfatta. Ad esempio, si provi a immaginare un gioco per computer di realtà virtuale che richieda la presenza di un determinato adattatore grafico. La funzione main() di tale gioco potrebbe avere il seguente aspetto:

```
#include <stdlib.h>

int main(void)
{
   if(!virtual_graphics()) exit(1);
   play();
   /* ... */
}
/* ... */
```

dove virtual_graphics() è una funzione definita dall'utente che restituisce il valore logico vero se sul computer è presente l'adattatore grafico di realtà virtuale. Se tale adattatore non è installato, virtual_graphics() restituirà il valore falso e il programma avrà termine.

Nell'esempio seguente menu() utilizza exit() per uscire dal programma e tornare al sistema operativo:

```
void menu(void)
{
  char ch;

  printf("1. Verifica ortografica\n");
  printf("2. Correzione errori ortografici\n");
  printf("3. Visualizza errori ortografici\n");
  printf("4. Esce\n");
  printf("      Immettere la scelta: ");

  do {
    ch=getchar(); /* legge la scelta dalla tastiera */
      switch(ch) {
        case '1':
          check_spelling();
          break;
        case '2':
          correct_errors();
          break;
        case '3':
          display_errors();
          break;
        case '4':
          exit(0); /* torna al sistema operativo */
      }
    } while(ch!='1' && ch!='2' && ch!='3');
}
```



## L'istruzione continue

L'istruzione continue funziona come un'istruzione break. Ma invece di causare la terminazione del ciclo, continue causa l'esecuzione della successiva iterazione del ciclo, saltando tutto il codice seguente. Nel caso di cicli for, continue provoca l'esecuzione del test condizionale e della porzione di incremento del ciclo. Nel caso di cicli while e do-while, il controllo del programma passa ai test condizionali. Ad esempio, il programma seguente conta il numero di spazi contenuti in una stringa immessa dall'utente:

```
/* Conteggio degli spazi */
#include <stdio.h>

int main(void)
{
  char s[80], *str;
  int space;

  printf("Immettere una stringa: ");
  gets(s);
  str = s;
```

```
  for(space=0; *str; str++) {
    if(*str != ' ') continue;
    space++;
  }
  printf("%d spazi\n", space);
}
```

In questo programma viene controllato ogni carattere per determinare se è uno spazio. In caso negativo, l'istruzione continue provoca la successiva iterazione del ciclo for. Se invece il carattere è uno spazio, viene incrementata la variabile space.

Il seguente esempio mostra come è possibile utilizzare continue per accelerare l'uscita da un ciclo causando la precoce esecuzione del test condizionale:

```
void code(void)
{
  char done, ch;

  done = 0;
  while(!done) {
    ch = getchar();
    if(ch=='$') {
      done = 1;
      continue;
    }
    putchar(ch+1); /* passa alla posizione
                      alfabetica successiva */
  }
}
```

Questa funzione codifica un messaggio spostando tutti i caratteri immessi alla lettera successiva. Ad esempio una A diverrà una B. La funzione ha termine quando si immette il carattere $. Dopo l'immissione del $, non viene visualizzato alcunché poiché il test condizionale, richiamato da continue troverà la variabile done vera e questo provocherà l'uscita dal ciclo.

## 3.6 Le espressioni

Il Capitolo 2 ha parlato approfonditamente delle espressioni. Tuttavia, in questo Capitolo occorre ricordare alcuni aspetti particolari. Occorre ricordare che un'istruzione di espressione non è che un espressione valida in C seguita da un punto e virgola come in:



```
func();     /* chiamata di una funzione */
a = b+c;    /* istruzione di assegnamento */
b+f();       /* istruzione corretta, anche se curiosa */
;            /* istruzione vuota */
```

La prima istruzione di espressione esegue una chiamata a una funzione. La seconda è un assegnamento. La terza espressione, anche se può sembrare curiosa, è comunque valutata dal compilatore C in quanto la funzione f() può eseguire alcune operazioni necessarie. L'ultimo esempio dimostra la possibilità di usare istruzioni vuote (chiamate anche *istruzioni nulle*).

## 3.7 I blocchi

Le istruzioni di blocco non sono altro che gruppi di istruzioni correlate trattate come una unità. Le istruzioni che compongono un blocco sono connesse fra di loro. I blocchi sono chiamati anche *istruzioni* composte. Un blocco inizia con una { e termina con la corrispondente }. I programmatori utilizzano istruzioni di blocco soprattutto per creare gruppi di istruzioni per altre istruzioni, come ad esempio un if. Tuttavia, è possibile porre un'istruzione di blocco in qualunque punto in cui possa essere accettata una qualsiasi altra istruzione. Ad esempio, il seguente programma è perfettamente corretto (anche se insolito):

```
#include <stdio.h>

int main(void)
{
  int i;

  {  /* inizio del blocco */
     i = 120;
     printf("%d", i);
  }

  return 0;
}
```

Capitolo 4

# Gli array e le stringhe


4.1 **Gli array monodimensionali**

4.2 **La generazione di un puntatore a un array**

4.3 **Come passare un array monodimensionale a una funzione**

4.4 **Le stringhe**

4.5 **Gli array bidimensionali**

4.6 **Gli array multidimensionali**

4.7 **L'indicizzazione dei puntatori**

4.8 **L'inizializzazione degli array**

4.9 **L'esempio del tris (tic-tac-toe)**


Un *array* è formato da una serie di variabili dello stesso tipo cui si fa riferimento utilizzando un nome comune. Per accedere a un determinato elemento di un array si utilizza un indice. In C, tutti gli array sono memorizzati in locazioni di memoria contigue. L'indirizzo più basso corrisponde al primo elemento e l'indirizzo più alto all'ultimo elemento. Gli array non sono necessariamente monodimensionali. Il tipo più comune di array è la *stringa chiusa dal carattere nullo*, che è semplicemente un array di caratteri al termine del quale vi è un carattere nullo.

Gli array e i puntatori sono strettamente correlati; quando si parla degli uni si fa normalmente riferimento agli altri. Questo capitolo si concentra sugli array, mentre il Capitolo 5 si occupa più approfonditamente dei puntatori. Per comprendere appieno questi importanti costrutti del C è necessario leggere entrambi i capitoli.

## 4.1 Gli array monodimensionali

La forma generale di una dichiarazione di un array monodimensionale è la seguente:

tipo *nome_var[dim];*

Come ogni altra variabile, anche un array deve essere dichiarato esplicitamente in modo che il compilatore possa allocare lo spazio in memoria richiesto da tale array. Qui, *tipo* dichiara il tipo base dell'array, ovvero il tipo di ogni elemento dell'array. *dim* definisce il numero di elementi che l'array deve contenere. Ad esempio, per dichiarare un array di 100 elementi chiamato bilancio di tipo double si deve utilizzare la seguente istruzione:

```
double balance[100];
```

Per accedere a un elemento si associa un indice al nome dell'array. Si deve porre l'indice dell'elemento fra parentesi quadre dopo il nome dell'array. Ad esempio:

```
bilance[3]=12.23;
```

assegna all'elemento numero 3 dell'array balance il valore 12.23.

In C/C++, tutti gli array iniziano dall'indice 0. Pertanto, quando si scrive

```
char p[10];
```

si sta dichiarando un array di caratteri formato da 10 elementi, da p[0] a p[9]. Ad esempio, il seguente programma inserisce in un array di interi i numeri da 0 a 99.

```
#include <stdio.h>

int main(void)
{
  int x[100]; /* dichiarazione di un array di 100 interi /
  int t;

  /* Inserisce in x i valori da 0 a 99 /*
  for(t=0; t<100; ++t) x[t] = t;

  /* Visualizza il contenuto di x /*
  for(t=0; t<100; ++t) printf("%d ", x[t]);

  return 0;
}
```

La quantità di memoria richiesta per contenere un array è strettamente legata al suo tipo e alle sue dimensioni. Per un array monodimensionale, le dimensioni totali in byte vengono calcolate nel modo seguente:

byte totali = sizeof(tipo) * dimensioni array



In C/C++ non è prevista alcuna verifica di superamento dei limiti degli array. Questo significa che è consentito scrivere oltre i limiti di un array, in un'altra variabile e perfino nel codice del programma. È quindi compito del programmatore verificare sempre di non superare i limiti degli array. Ad esempio, questo codice verrà compilato senza errori, ma è errato poiché il ciclo for consente di superare le dimensioni dell'array count.

```
int count[10], i;

/* i limiti dell'array vengono superati */
for(i=0; i<100; i++) count[i] = i;
```

Gli array monodimensionali sono essenzialmente elenchi di informazioni dello stesso tipo conservate in locazioni di memoria contigue secondo un ordine ben preciso. Ad esempio, la Figura 4.1 mostra l'aspetto in memoria dell'array A che inizia dalla locazione di memoria 1000 ed è dichiarato nel modo seguente:

```
char a[7];
```

## 4.2 La generazione di un puntatore a un array

Per generare un puntatore al primo elemento di un array basta specificare il nome dell'array, senza alcun indice. Ad esempio, dato

```
int sample[10];
```

è possibile generare un puntatore al primo elemento utilizzando il nome sample. Ad esempio, il seguente frammento di programma assegna a p l'indirizzo del primo elemento di sample:

```
int *p;
int sample[10];
```

| Elemento | a[0] | a[1] | a[2] | a[3] | a[4] | a[5] | a[6] |
|---|---|---|---|---|---|---|---|
| Indirizzo | 1000 | 1001 | 1002 | 1003 | 1004 | 1005 | 1006 |

**Figura 4.1** Un array di caratteri di sette elementi che inizia dalla locazione 1000.

```
p = sample;
```

Inoltre è possibile specificare l'indirizzo del primo elemento di un array utilizzando l'operatore &. Ad esempio, sia sample che &sample[0] producono lo stesso risultato. Tuttavia, nel software professionale, sarà difficile trovare la forma &sample[0].

## 4.3 Come passare un array monodimensionale a una funzione

In C non è possibile passare un intero array come argomento a una funzione. È possibile, tuttavia, passare alla funzione un puntatore all'array, specificando il nome dell'array senza alcun indice. Ad esempio, il seguente frammento di programma passa a func1() l'indirizzo di i:

```
int main(void)
{
  int i[10];

  func1(i);
  .
  .
  .
}
```

Se una funzione riceve un array monodimensionale, si può dichiarare il parametro formale in tre modi: come puntatore, come array dimensionato o come array non dimensionato. Ad esempio, una funzione chiamata func1() che riceve l'array i può essere dichiarata come:

```
void func1(int *x) /* puntatore */
{
  .
  .
  .
}
```

oppure come

```
void func1(int x[10]) /* array dimensionato */
{
  .
  .
  .
}
```

o infine come

```
void func1(int x[]) /* array non dimensionato */
{
  .
  .
  .
}
```



Queste tre dichiarazioni producono risultati simili in quanto ognuna di esse dice al compilatore che si riceverà un puntatore a un intero. La prima dichiarazione utilizza a tutti gli effetti un puntatore. La seconda impiega la dichiarazione standard di array. Nell'ultima versione, una versione modificata della dichiarazione di un array specifica semplicemente che si riceverà un array di tipo int di lunghezza non specificata. Come si può vedere, la lunghezza dell'array non è importante per l'esecuzione della funzione in quanto il C/C++ non esegue alcuna verifica di superamento dei limiti. Per quanto riguarda il compilatore, sarà accettabile anche la forma:

```
void func1(int x[32])
{
  .
  .
  .
}
```

in quanto il compilatore C genera il codice che prepara func1() a ricevere un puntatore e non crea un array di 32 elementi.

## 4.4 Le stringhe

L'uso di gran lunga più comune degli array monodimensionali li vede nelle vesti di stringhe di caratteri. Il linguaggio C++ supporta due tipi di stringhe. Il primo tipo è rappresentato dalle stringhe chiuse dal carattere nullo. Si tratta di array di caratteri che terminano con il carattere nullo (il carattere numero 0). Pertanto una stringa chiusa dal carattere nullo contiene tutti i caratteri che formano la stringa

più un carattere nullo. Questo è l'unico tipo di stringa definito dal C ed è tuttora molto utilizzato. Per questo motivo, le stringhe chiuse dal carattere nullo sono chiamate anche stringhe C. Il linguaggio C++ definisce anche una classe per le stringhe, chiamata string, che fornisce un approccio a oggetti alla gestione delle stringhe. Tale classe verrà descritta più avanti in questo volume. Qui si parlerà unicamente delle stringhe chiuse dal carattere nullo.

Quando si dichiara un array che contiene una stringa chiusa dal carattere nullo, occorre dunque indicare dimensioni pari al massimo contenuto che verrà memorizzato nella stringa aumentato di una unità per il carattere nullo. Ad esempio, per dichiarare un array str che può contenere stringhe di 10 caratteri, si dovrà scrivere:

```
char str[11];
```

Questa istruzione lascia uno spazio per il carattere nullo al termine della stringa.

Anche quando si crea una costante stringa quotata in realtà viene creata una stringa chiusa dal carattere nullo. Una *costante stringa* è un elenco di caratteri racchiuso tra doppi apici. Ad esempio,

"ciao a tutti"

Non è necessario aggiungere il carattere nullo per terminare la stringa: questa operazione viene svolta automaticamente dal compilatore C/C++.

Il C/C++ prevede un'ampia gamma di funzioni per la manipolazione delle stringhe chiuse dal carattere nullo. Le più comuni sono:

| NOME | FUNZIONE |
|---|---|
| `strcpy(s1, s2)` | Copia s2 in s1. |
| `strcat(s1, s2)` | Concatena s2 alla fine di s1. |
| `strlen(s1)` | Restituisce la lunghezza di s1. |
| `strcmp(s1, s2)` | Restituisce 0 se s1 e s2 sono uguali; un valore minore di 0 se s1<s2; maggiore di 0 se s1>s2. |
| `strchr(s1, ch)` | Restituisce un puntatore alla prima occorrenza di ch in s1. |
| `strstr(s1, s2)` | Restituisce un puntatore alla prima occorrenza di s2 in s1. |

Queste funzioni utilizzano il file header standard string.h (i programmi C++ possono anche usare il file header C++). Il seguente programma illustra l'uso di queste funzioni:



```
#include <stdio.h>
#include <string.h>

int main(void)
{
  char s1[80], s2[80];

  gets(s1);
  gets(s2);

  printf("lunghezze: %d %d\n", strlen(s1), strlen(s2));

  if(!strcmp(s1, s2)) printf("Le stringhe sono uguali\n");
  strcat(s1, s2);
  printf("%s\n", s1);

  strcpy(s1, "Questa è una prova.\n");
  printf(s1);
  if(strchr("salve", 'e')) printf("e si trova in salve\n");
  if(strstr("ciao a tutti", "tu")) printf("trovato tu");

  return 0;
}
```

Se si esegue questo programma e si immettono le stringhe "salve" e "salve", l'output sarà:

```
lunghezze: 5 5
Le stringhe sono uguali
salvesalve
Questa è una prova.
e si trova in salve
trovato tu
```

È da notare che quando le stringhe sono uguali, strcmp() restituisce il valore false. Se si sta controllando l'uguaglianza delle stringhe, occorre quindi ricordarsi di utilizzare l'operatore logico ! per invertire la condizione.

Anche se ora il linguaggio C++ definisce una classe per le stringhe, le stringhe chiuse dal carattere nullo sono ancora ampiamente utilizzate nei programmi. Probabilmente questo tipo di stringhe rimarrà in uso ancora a lungo in quanto offre un elevato livello di efficienza e garantisce al programmatore il massimo controllo sulle operazioni sulle stringhe. Ma per le normali operazioni di manipolazione delle stringhe, è molto più comodo utilizzare la classe C++ string.

## 4.5 Gli array bidimensionali

Il C/C++ prevede l'uso di array multidimensionali. La forma più semplice di array multidimensionale è l'array bidimensionale. Un array bidimensionale è, essenzialmente, un array di array monodimensionali. Per dichiarare un array bidimensionale di interi d di dimensioni 10,20, si dovrà scrivere

```
int d[10][20];
```

Occorre fare particolare attenzione a questa dichiarazione. Alcuni linguaggi di programmazione utilizzano una virgola per separare le dimensioni dell'array; in C/C++, ogni dimensione deve invece essere racchiusa fra parentesi quadre.

Analogamente, per accedere al punto 1,2 dell'array d si dovrà usare la forma

```
d[1][2]
```

Il seguente esempio inserisce in un array bidimensionale i numeri da 1 a 12 e poi li stampa riga per riga.

```
#include <stdio.h>

int main(void)
{
  int t, i, num[3][4];

  for(t=0; t<3; ++t)
    for(i=0; i<4; ++i)
      num[t][i] = (t*4)+i+1;

  /* stampa */
  for(t=0; t<3; ++t) {
    for(i=0; i<4; ++i)
      printf("%3d ", num[t][i]);
    printf("\n");
  }

  return 0;
}
```

In questo esempio, num[0][0] ha il valore 1, num[0][1] ha il valore 2, num[0][2] ha il valore 3 e così via. Il valore di num[2][3] è 12.



Si può visualizzare l'array num nel modo seguente:

num [t] [i]

| | 0 | 1 | 2 | 3 |
|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 |
| 1 | 5 | 6 | 7 | 8 |
| 2 | 9 | 10 | 11 | 12 |

Gli array bidimensionali sono memorizzati in una matrice di righe e colonne, dove il primo indice indica la riga e il secondo indica la colonna. Questo significa che l'indice più a destra cambia più velocemente rispetto a quello più a sinistra quando si accede agli elementi dell'array nell'ordine in cui essi sono effettivamente conservati in memoria. Per una rappresentazione grafica del modo in cui un array bidimensionale viene conservato in memoria, consultare la Figura 4.2.

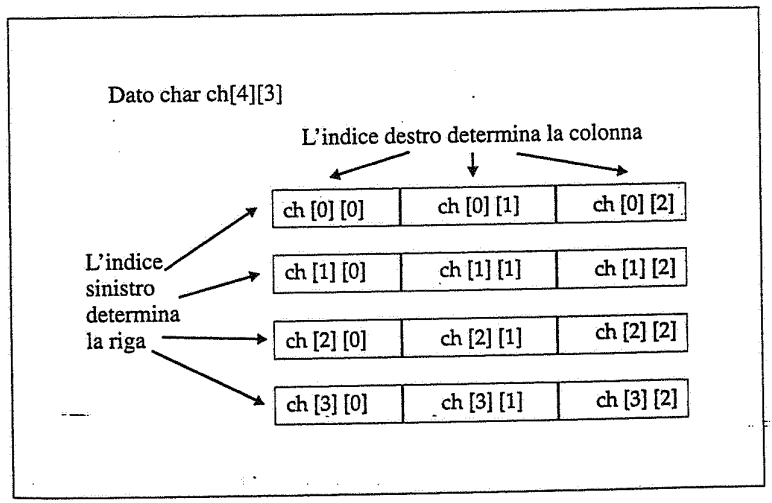



Figura 4.2 Memorizzazione di un array bidimensionale.

Nel caso di un array bidimensionale, il numero di byte di memoria richiesti per contenere l'array è dato dalla seguente formula:

*byte = dimensioni 1° indice * dimensioni 2° indice * sizeof(tipo base)*

Pertanto, se si considera che gli interi occupano 4 byte, un array bidimensionale di interi di dimensioni 10,5 richiederà:

10 x 5 x 4

ovvero 200 byte.

Quando si utilizza un array bidimensionale come argomento di una funzione, viene in effetti passato solo il primo elemento dell'array. Tuttavia, il parametro che riceve un array bidimensionale deve definire almeno le dimensioni che si trovano più a destra (si può anche specificare la dimensione più a sinistra ma questo non è strettamente necessario). La dimensione più a destra è necessaria in quanto il compilatore C/C++ deve conoscere la lunghezza di ogni riga per poter accedere correttamente ai vari elementi dell'array. Ad esempio, una funzione che riceve un array bidimensionale di interi di dimensioni 10,10 dovrà essere dichiarata nel seguente modo:

```
void func1(int x[][10])
{
  .
  .
  .
}
```

Il compilatore deve conoscere le dimensioni più a destra per poter eseguire correttamente espressioni come la seguente:

```
x[2][4]
```

Se la lunghezza delle righe non fosse nota, il compilatore non potrebbe determinare l'inizio della terza riga.

Il seguente breve programma utilizza un array bidimensionale per memorizzare i voti numerici di ogni studente delle classi di un professore. Il programma assume che il professore abbia tre classi e un massimo di 30 studenti per classe. Si noti il modo in cui si accede all'array grade da parte di ognuna delle funzioni.

```
#include <stdio.h>
#include <ctype.h>
```



```
#include <stdlib.h>

/* Un semplice database dei voti degli studenti. */

#define CLASSES  3
#define GRADES  30

int grade[CLASSES][GRADES];

void enter_grades(void);
int get_grade(int num);
void disp_grades(int g[][GRADES]);

int main(void)
{
  char ch, str[80];

  for(;;) {
    do {
      printf("(I)mmissione voti\n");
      printf("(S)tampa voti\n");
      printf("(U)scita\n");
      gets(str);
      ch = toupper(*str);
    } while(ch!='I' && ch!='S' && ch!='U');

    switch(ch) {
      case 'I':
        enter_grades();
        break;
      case 'S':
        disp_grades(grade);
        break;
      case 'U':
        exit(0);
    }
  }

  return 0;
}

/* Immissione dei voti. */
void enter_grades(void)
{
  int t, i;
```

```
  for(t=0; t<CLASSES; t++) {
    printf("Classe  %d:\n", t+1);
    for(i=0; i<GRADES; ++i)
      grade[t][i] = get_grade(i);
  }
}

/* Lettura dei voti. */
int get_grade(int num)
{
  char s[80];

  printf("Immettere il voto dello studente %d:\n", num+1);
  gets(s);
  return(atoi(s));
}
/* Visualizzazione dei voti. */
void disp_grades(int g[][GRADES])
{
  int t, i;

  for(t=0; t<CLASSES; ++t) {
    printf("Classe  %d:\n", t+1);
    for(i=0; i<GRADES; ++i)
      printf("Studente %d è %d\n", i+1, g[t][i]);
  }
}
```

## Gli array di stringhe

In programmazione non è difficile incontrare array di stringhe. Ad esempio, il processore di input di un database potrebbe verificare la corrispondenza dei comandi immessi dall'utente con un array di comandi validi. Per creare un array di stringhe chiuse dal carattere nullo, si deve usare un array di caratteri bidimensionale. Le dimensioni dell'indice di sinistra determinano il numero di stringhe e le dimensioni dell'indice di destra specifica la lunghezza massima di ogni stringa. Il codice seguente dichiara un array di 30 stringhe ognuna delle quali può essere lunga al massimo 79 caratteri.

```
char str_array[30][80];
```

In questo modo è molto facile accedere alle singole stringhe: basta specificare solo l'indice di sinistra. Ad esempio, l'istruzione che segue richiama gets() con la terza stringa contenuta in str_array.

```
gets(str_array[2]);
```

L'istruzione precedente è funzionalmente equivalente a:

```
gets(&str_array[2][0]);
```

ma la prima delle due forme è molto più comune nel codice professionale. Per meglio comprendere il funzionamento degli array di stringhe, si studi il seguente breve programma che utilizza un array di stringhe come base per un semplicissimo editor di testi:

```
/* Un semplicissimo editor di testi. */
#include <stdio.h>

#define MAX 100
#define LEN 80

char text[MAX][LEN];

int main(void)
{
  register int t, i, j;

  printf("Per uscire, immettere una riga vuota.\n");

  for(t=0; t<MAX; t++) {
    printf("%d: ", t);
    gets(text[t]);
    if(!*text[t]) break; /* riga vuota: uscita */
  }

  for(i=0; i<t; i++) {
    for(j=0; text[i][j]; j++) putchar(text[i][j]);
    putchar('\n');
  }

  return 0;
}
```

Questo programma legge righe di testo fino a incontrare una riga vuota. Quindi visualizza ogni riga carattere per carattere.

## 4.6 Gli array multidimensionali

Il C/C++ consente di creare array a più di due dimensioni. Il limite esatto, se esiste, è determinato dal compilatore utilizzato. La forma generica della dichiarazione di un array multidimensionale è la seguente:

*tipo nome*[*Dim1*][*Dim2*][*Dim3*]...[*DimN*];

Gli array di tre o più dimensioni non vengono utilizzati molto spesso a causa della quantità di memoria che richiedono. Ad esempio, un array di caratteri quadridimensionale di dimensioni 10,6,9,4 richiede:

10*6*9*4

ovvero 2160 byte. Se l'array contenesse interi di 2 byte, occuperebbe 4320 byte. Se contenesse valori double (assumendo che un double occupi 8 byte), occuperebbe 17280 byte. La quantità di memoria richiesta aumenta esponenzialmente con l'aumentare delle dimensioni. Ad esempio, se all'array precedente viene aggiunta una quinta dimensione di 10 elementi, allora si raggiungerebbero i 172.800 byte.

Negli array multidimensionali, il calcolo dell'indice richiede una grande quantità di tempo di elaborazione. Questo significa che l'accesso a un elemento di un array multidimensionale può essere più lento rispetto all'accesso a un elemento di un array monodimensionale. Quando si passa a una funzione un array multidimensionale, si devono dichiarare tutte le dimensioni tranne quella più a sinistra. Ad esempio, se si dichiara l'array m come

```
int m[4][3][6][5];
```

una funzione func1() che riceva m dovrà avere il seguente aspetto:

```
void func1(int d[][3][6][5])
{
  .
  .
  .
}
```

Naturalmente, è comunque possibile includere anche la prima dimensione.



## 4.7 L'indicizzazione dei puntatori

In C/C++, i puntatori e gli array sono oggetti strettamente correlati. Come si è già visto, il nome di un array senza indice è il puntatore al primo elemento dell'array. Ad esempio, considerando il seguente array:

```
char p[10];
```

Le seguenti due istruzioni hanno lo stesso significato:

```
p
&p[0]
```

Detto in altri termini,

```
p == &p[0]
```

fornisce un risultato vero poiché l'indirizzo del primo elemento di un array corrisponde all'indirizzo dell'array.

Come si è detto, il nome dell'array senza un indice genera un puntatore. Analogamente, un puntatore può essere indicizzato come se fosse dichiarato tramite un array. Ad esempio, si consideri questo frammento di programma:

```
int *p, i[10];
p = i;
p[5] = 100;    /* assegnamento tramite indice */
*(p+5) = 100;  /* uso dell'aritmetica dei puntatori */
```

Entrambe le istruzioni di assegnamento inseriscono il valore 100 nel sesto elemento di i. La prima istruzione fa riferimento a p; la seconda utilizza l'aritmetica dei puntatori. In entrambi i casi, il risultato è lo stesso (i puntatori e l'aritmetica dei puntatori sono l'argomento del Capitolo 5).

Questo stesso concetto si applica anche agli array di due o più dimensioni. Ad esempio, assumendo che a sia un array di 10 per 10 interi, queste due istruzioni sono equivalenti:

```
a
&a[0][0]
```

Questo significa che è possibile far riferimento all'elemento 0,4 di a in due modi: utilizzando l'indice dell'array, a[0][4] o utilizzando il puntatore *((int *)a+4).

Analogamente, l'elemento 1,2 può essere visto come a[1][2] o come *((int *)a+12). In generale, per ogni array bidimensionale,

```
a[j][k]
```

è equivalente a

```
*((tipo-base*)a + (j * lunghezza-riga) + k)
```

La conversione cast del puntatore all'array in un puntatore al suo tipo base è necessaria perché l'aritmetica dei puntatori possa funzionare correttamente. Spesso vengono utilizzati i puntatori per accedere agli array in quanto l'aritmetica dei puntatori è normalmente più veloce rispetto all'indicizzazione degli array.

Un array bidimensionale può essere ridotto a un puntatore a un array di array monodimensionali. Pertanto, l'uso di una variabile puntatore distinta è un modo semplice per utilizzare i puntatori per accedere agli elementi di una riga di un array bidimensionale. La seguente funzione illustra questa tecnica. La funzione stampa il contenuto della riga specificata per l'array di interi globale num:

```
int num[10][10];
  .
  .
  .
void  pr_row(int j)
{
  int *p, t;

  p = (int *) &num[j][0]; /* determina l'indirizzo
                     del primo elemento della riga j */

  for(t=0; t<10; ++t) printf("%d ", *(p+t));
}
```

È possibile generalizzare questa routine facendo in modo che gli argomenti di chiamata siano la riga, la lunghezza della riga e un puntatore al primo elemento dell'array:

```
void pr_row(int j, int row_dimension, int *p)
{
  int t;

  p = p + (j * row_dimension);

  for(t=0; t<row_dimension; ++t)
    printf("%d ", (p+t));
}
.
.
.
void f(void)
{
  int num[10][10];

  pr_row (0, 10, (int ) num); /* stampa la prima riga */
}
```



Gli array di più di due dimensioni possono essere ridotti in modo analogo. Ad esempio, un array tridimensionale può essere ridotto a un puntatore a un array bidimensionale il quale può essere ridotto a un puntatore a un array monodimensionale. Generalmente, un array *n*-dimensionale può essere ridotto a un puntatore a un array (*n*-1)-dimensionale. Questo nuovo array può essere ulteriormente ridotto utilizzando lo stesso metodo. Il processo termina quando viene prodotto un array monodimensionale.

## 4.8 L'inizializzazione degli array

Il C(C++ consente di inizializzare un array nel momento della dichiarazione. La forma generica dell'inizializzazione di un array è simile a quella delle altre variabili, come si può vedere nella riga seguente:

*specificatore_tipo nome_array*[*dim1*]...[*dimN*] = { *elenco_valori*};

L'*elenco_valori* è un elenco di valori separati da virgole il cui tipo deve essere compatibile con *specificatore_tipo*. Il primo valore verrà posizionato nella prima posizione dell'array, il secondo nella seconda posizione e così via. Si faccia particolare attenzione al punto e virgola che segue la parentesi graffa di chiusura.

Nell'esempio seguente, viene inizializzato un array di dieci elementi interi con i numeri da 1 a 10:

```
int i[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
```

Questo significa che i[0] conterrà il valore 1 e che i[9] conterrà il valore 10.

Gli array di caratteri che contengono stringhe consentono di utilizzare un'inizializzazione semplificata che ha la seguente forma:

char *nome_array*[*dim*] = "*stringa*";

Ad esempio, questo frammento di codice inizializza la stringa str con la frase "Il C++ è bello".

```
char str[15] = "Il C++ è bello";
```

Questo equivale a scrivere:

```
char str[15] = {'I', 'l', ' ', 'C', '+', '+', ' ', 'è',' ', 'b', 'e', 'l',
'l', 'o', '\0'};
```

Poiché in C tutte le stringhe terminano con il carattere nullo, è necessario assicurarsi che l'array dichiarato sia lungo quanto basta per contenere il carattere nullo. Questo è il motivo per cui la stringa str è lunga quindici caratteri, anche se la frase "Il C ++ è bello" è lunga solo quattordici caratteri. Quando si utilizza una costante stringa, è il compilatore a inserire automaticamente il carattere finale nullo.

Gli array multidimensionali vengono inizializzati nello stesso modo degli array monodimensionali. Ad esempio, il seguente array inizializza sqrs con i numeri da 1 a 10 e con i rispettivi quadrati.

```
int sqrs[10][2] = {
  1,1,
  2,4,
  3,9,
  4,16,
  5,25,
  6,36,
  7,49,
  8,64,
  9,81,
  10,100
};
```

Quando si inizializza un array multidimensionale, si possono aggiungere le parentesi graffe attorno agli inizializzatori di ciascuna dimensione. Questo è detto raggruppamento dei sotto-aggregati. Ad esempio, ecco un altro modo con cui si può scrivere la dichiarazione precedente.

```
int sqrs[10][2] = {
  {1,1},
  {2,4},
  {3,9},
  {4,16},
  {5,25},
  {6,36},
  {7,49},
  {8,64},
  {9,81},
  {10,100}
};
```



Quando si usa il raggruppamento dei sotto-aggregati, se non si fornisce un numero sufficiente di inizializzatori per il gruppo, a tutti i membri rimanenti verrà assegnato il valore 0.

## L'inizializzazione di array non dimensionati

Si immagini di utilizzare l'inizializzazione di un array per costruire una tabella di messaggi di errore:

```
char e1[18] = "Errore di lettura\n";
char e2[20] = "Errore di scrittura\n";
char e3[27] = "Impossibile aprire il file\n";
```

Come si può immaginare, non è comodo contare manualmente i caratteri contenuti in ogni messaggio per determinare le dimensioni corrette dell'array. Fortunatamente si può lasciare che il compilatore calcoli automaticamente le dimensioni degli array, ovvero si possono usare gli array non dimensionati. Se, nell'istruzione di inizializzazione di un array, non si specificano le dimensioni dell'array, il compilatore C/C++ crea automaticamente un array grande a sufficienza per contenere tutti gli inizializzatori presenti. In questo caso si parla di *array non dimensionato*. Utilizzando questo approccio, la tabella dei messaggi diviene

```
char e1[] = "Errore di lettura\n";
char e2[] = "Errore di scrittura\n";
char e3[] = "Impossibile aprire il file\n";
```

Date queste inizializzazione, l'istruzione

```
printf("La lunghezza di %s è %d\n",  e2,  sizeof e2);
```

visualizzerà:

```
La lunghezza di Errore di scrittura è 20
```

Oltre a essere più comoda, l'inizializzazione di array non dimensionati consente di cambiare i messaggi senza temere di aver dimensionato in modo errato l'array.

Le inizializzazioni di array non dimensionati non si limitano agli array monodimensionali. Per gli array multidimensionali si deve specificare tutto tranne la dimensione più a sinistra (le altre dimensioni sono necessarie per consentire al compilatore di indicizzare correttamente l'array). In questo modo, è possibile costruire tabelle di varie lunghezze e il compilatore allocherà automaticamente lo spazio sufficiente. Ad esempio, ecco la dichiarazione di sqrs come array non dimensionato:

```
int sqrs[][2] = {
  1,1,
  2,4,
  3,9,
  4,16,
  5,25,
  6,36,
  7,49,
  8,64,
  9,81,
  10,100
};
```

Il vantaggio di questa dichiarazione rispetto alla versione dimensionata consiste nella possibilità di allungare o accorciare la tabella senza cambiare le dimensioni dell'array.

## 4.9 L'esempio del tris (tic-tac-toe)

Il corposo esempio che segue illustra molti dei modi in cui è possibile manipolare gli array in C/C++. Molto spesso, per simulare un gioco da scacchiera si utilizza un array bidimensionale. Questa sezione sviluppa un semplice programma di tris (tic-tac-toe).

Il computer svolge una strategia molto semplice. Quando è il turno del computer, utilizza la funzione get_computer_move() per eseguire la scansione della matrice, alla ricerca di una cella non occupata. Quando ne trova una, vi posiziona una O. Se non riesce a trovare una casella libera, esce dal gioco. La funzione



get_player_move() chiede di specificare la posizione in cui inserire una X. L'angolo superiore sinistro si trova alla posizione 1,1 e l'angolo inferiore destro alla posizione 3,3.

L'array a matrice viene inizializzato in modo da contenere spazi. Ogni mossa eseguita dal giocatore o dal computer trasforma uno spazio in una X o in una O. Questo semplifica anche la visualizzazione della matrice sullo schermo.

Ogni volta che viene eseguita una mossa, il programma richiama la funzione check(). Questa funzione restituisce uno spazio se non vi è ancora un vincitore, una X se ha vinto il giocatore o una O se ha vinto il computer.

Per fare ciò, scandisce le righe, le colonne e le diagonali, alla ricerca di file di X o di O.

La funzione disp_matrix() visualizza lo stato attuale del gioco. Si noti che il fatto di aver inizializzato la matrice con spazi semplifica questa funzione.

Le routine di questo esempio accedono all'array matrix ognuna in un modo diverso. Lo studio di queste funzioni aiuterà a comprendere meglio il funzionamento di ogni operazione che è possibile svolgere su un array.

```
/* Un semplice gioco del tris (Tic Tac Toe). */
#include <stdio.h>
#include <stdlib.h>

char matrix[3][3];  /* matrice del gioco */

char check(void);
void init_matrix(void);
void get_player_move(void);
void get_computer_move(void);
void disp_matrix(void);

int main(void)
{
  char done;

  printf("Questo è il gioco del Tris.\n");
  printf("Giocherai contro il computer.\n");

  done = ' ';
  init_matrix();
  do{
    disp_matrix();
    get_player_move();
    done = check(); /* ha vinto? */
    if(done!= ' ') break; /* vincitore!*/
    get_computer_move();
    done = check(); /* ha vinto? */
```

```
  } while(done== ' ');
  if(done=='X') printf("Hai vinto tu!\n");
  else printf("Ho vinto io!!!!\n");
  disp_matrix(); /* mostra le posizioni finali */

  return 0;
}

/* Inizializza la matrice. */
void init_matrix(void)
{
  int i, j;

  for(i=0; i<3; i++)
    for(j=0; j<3; j++) matrix[i][j] =  ' ';
}


/* Legge la mossa del giocatore. /
void get_player_move(void)
{
  int x, y;

  printf("Immettere le coordinate X,Y: ");
  scanf("%d%*c%d", &x, &y);

  x--; y--;

  if(matrix[x][y]!= ' '){
    printf("Errore, riprovare.\n");
    get_player_move();
  }
  else matrix[x][y] = 'X';
}

/* Mossa del computer. */
void get_computer_move(void)
{
  int i, j;
  for(i=0; i<3; i++){
    for(j=0; j<3; j++)
      if(matrix[i][j]==' ') break;
    if(matrix[i][j]==' ') break;
  }

  if(i*j==9)  {
```



```
    printf("patta\n");
    exit(0);
  }
  else
    matrix[i][j] = 'O';
}

/* Visualizza la matrice. */
void disp_matrix(void)
{
  int t;

  for(t=0; t<3; t++) {


    printf(" %c | %c | %c ",matrix[t][0],
           matrix[t][1], matrix [t][2]);
    if(t!=2) printf("\n--|--|--\n");
  }
  printf("\n");
}

/* Controlla se c'è un vincitore. */
char check(void)
{
  int i;

  for(i=0; i<3; i++)  /* righe */
    if(matrix[i][0]==matrix[i][1] &&
       matrix[i][0]==matrix[i][2]) return matrix[i][0];

  for(i=0; i<3; i++)  /* colonne */
    if(matrix[0][i]==matrix[1][i] &&
       matrix[0][i]==matrix[2][i]) return matrix[0][i];

  /* diagonali */
  if(matrix[0][0]==matrix[1][1] &&
     matrix[1][1]==matrix[2][2])
       return matrix[0][0];

  if(matrix[0][2]==matrix[1][1] &&
     matrix[1][1]==matrix[2][0])
       return matrix[0][2];

  return ' ';
}
```

Capitolo 5

# I puntatori


5.1 **Che cosa sono i puntatori?**
5.2 **Variabili puntatore**
5.3 **Gli operatori per i puntatori**
5.4 **Espressioni con puntatori**
5.5 **Puntatori e array**
5.6 **Indirizzamento multilivello**
5.7 **Inizializzazione di puntatori**
5.8 **Puntatori a funzioni**
5.9 **Le funzioni di allocazione dinamica del C**
5.10 **Problemi con i puntatori**


Per poter scrivere programmi C/C++ è fondamentale comprendere appieno l'uso dei puntatori. Questo per tre motivi: innanzi tutto, i puntatori costituiscono un metodo con il quale le funzioni possono modificare i propri argomenti. In secondo luogo, i puntatori consentono di utilizzare le routine di allocazione dinamica. In terzo luogo, i puntatori possono aumentare l'efficienza di determinate routine. Inoltre, come si vedrà nella seconda parte della guida, i puntatori giocano nuovi e importanti ruoli anche in C++.

I puntatori sono una delle funzionalità più potenti e più pericolose del C/C++. Ad esempio, un puntatore non inizializzato o un puntatore che contiene valori non corretti possono provocare il blocco del sistema. Ma c'è di peggio: è facile utilizzare i puntatori in modo errato, inserendo nel codice bug difficilissimi da scovare.

Per l'importanza e i potenziali abusi dei puntatori, questo capitolo si occupa dei puntatori in modo dettagliato.

## 5.1 Che cosa sono i puntatori?

Un *puntatore* è una variabile che contiene un indirizzo di memoria. Questo indirizzo corrisponde alla posizione di un altro oggetto (normalmente un'altra variabile) in memoria. Ad esempio, se una variabile contiene l'indirizzo di un'altra variabile, si dice che la prima variabile *punta* alla seconda. La Figura 5.1 illustra questa situazione.

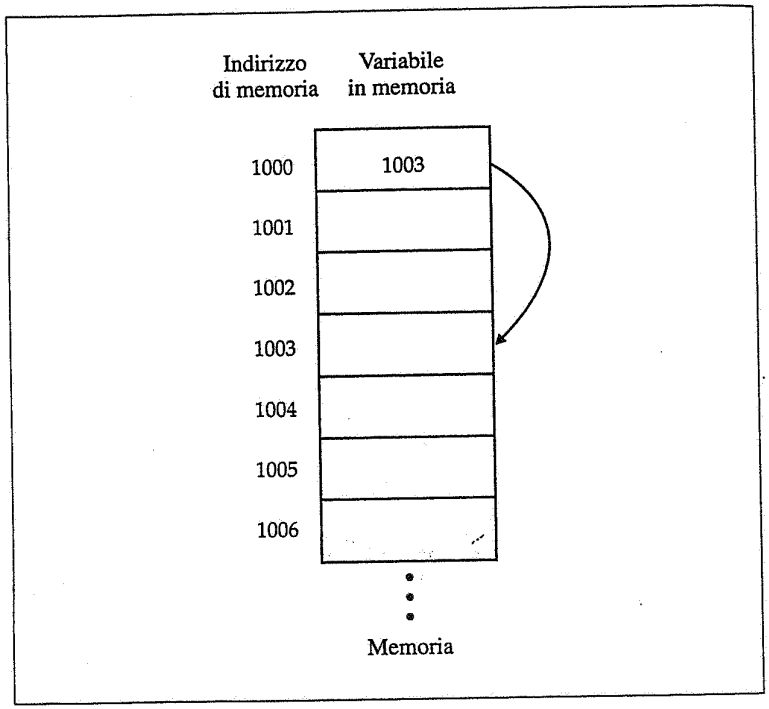



**Figura 5.1** Una variabile punta a un'altra.

## 5.2 Variabili puntatore

Se una variabile deve contenere un puntatore, deve essere dichiarata come tale. La dichiarazione di un puntatore è formata da un tipo base, un asterisco e un nome di variabile. La forma generale di dichiarazione di una variabile puntatore è la seguente:

*tipo* **nome*;*

dove *tipo* è il tipo base del puntatore e può essere un qualsiasi tipo valido. *nome* definisce il nome della variabile puntatore.



Il tipo base del puntatore definisce il tipo delle variabili a cui può puntare il puntatore. Tecnicamente, un qualsiasi tipo di puntatore può puntare a un qualunque indirizzo della memoria. Tuttavia, tutta l'aritmetica dei puntatori si basa sul tipo della variabile puntata e quindi è importante dichiarare correttamente il puntatore (l'aritmetica dei puntatori verrà discussa più avanti in questo stesso capitolo).

## 5.3 Gli operatori per i puntatori

Gli operatori per i puntatori sono stati descritti nel Capitolo 2. Qui l'argomento verrà approfondito, a partire dalle loro funzionalità di base. Vi sono due speciali operatori per i puntatori: * e &. Il secondo è un operatore unario che restituisce l'indirizzo di memoria del proprio operando (un operatore unario richiede un solo operando). Ad esempio:

```
m = &count;
```

assegna a m l'indirizzo di memoria della variabile count. Questo indirizzo corrisponde all'indirizzo della variabile nella memoria fisica del computer. Quindi l'indirizzo non ha nulla a che fare con il valore di count. Si può pensare all'operatore & traducendolo in italiano come "indirizzo di". Pertanto, l'istruzione di assegnamento precedente si può leggere come "m riceve l'indirizzo di count".

Per meglio comprendere l'assegnamento precedente, si immagini che la variabile count conservi il proprio valore nell'indirizzo di memoria 2000. Inoltre si immagini che count abbia il valore 100. Quindi, dopo l'assegnamento precedente, m conterrà il valore 2000.

Il secondo operatore sui puntatori, *, è complementare rispetto a &. Si tratta di un operatore unario che restituisce il valore che si trova nell'indirizzo di memoria che lo segue.

Ad esempio, se m contiene l'indirizzo di memoria della variabile count,

```
q = *m;
```

inserisce il valore di count in q. Pertanto, q conterrà il valore 100 in quanto nell'indirizzo 2000, che è l'indirizzo di memoria memorizzato in m, è contenuto il valore 100. Si può pensare a * come all'operatore "all'indirizzo"; per questo, l'istruzione precedente significa "q riceve il valore che si trova all'indirizzo di m".

Sia & che * hanno un livello di precedenza superiore rispetto a quello di tutti gli operatori aritmetici tranne il meno unario che condivide lo stesso livello di precedenza.

Occorre fare attenzione che le variabili puntatore puntino sempre al tipo di dati corretto. Ad esempio, quando si dichiara un puntatore a una variabile di tipo int, il compilatore assume che ogni indirizzo che il puntatore si troverà a contenere farà riferimento a una variabile intera. Poiché il C consente di assegnare qualsiasi indirizzo a una variabile puntatore, il seguente frammento di codice verrà compilato senza alcun messaggio di errore (o solo qualche avvertimento, a seconda del compilatore usato) ma non produrrà il risultato atteso:

```
#include <stdio.h>

int main(void)
{
  double x=100.1, y;
  int  *p;

  /* La prossima istruzione fa in modo che p (un puntatore
     a un intero) punti a un valore double. */
  p = &x;

  /* La prossima istruzione non funziona
     nel modo atteso. */
  y = *p;

  printf("%f", y); /* non visualizza 100.1 */
  return 0;
}
```

Questo frammento di codice non assegna il valore di x a y. Poiché p è dichiarato come un puntatore a un intero, solo 2 o 4 byte di informazioni verranno trasferiti da x a y e non gli 8 byte che normalmente compongono un numero double.

NOTA *In C++ non è consentito convertire un tipo di puntatore in un altro senza utilizzare uno specifico operatore di cast. Per questo motivo, il programma precedente non potrà essere compilato come programma C++ ma solo come programma C. Il tipo di errore descritto può in ogni caso verificarsi anche in C++ ma in un modo più sottile.*

## 5.4 Espressioni con puntatori

In generale, le espressioni contenenti puntatori seguono le stesse regole delle altre espressioni C. Questa sezione esamina alcuni aspetti peculiari delle espressioni con puntatori.



### Assegnamento di puntatori

Un puntatore, come ogni altra variabile, può essere utilizzato sul lato destro di un'istruzione di assegnamento in modo da assegnare il suo valore a un altro puntatore come ad esempio in:

```
#include <stdio.h>

int main(void)
{
  int x;
  int *p1, *p2;

  p1 = &x;
  p2 = p1;

  printf(" %p", p2); /* stampa l'indirizzo di x,
                        non il suo valore! */

  return 0;
}
```

Ora, sia p1 che p2 puntano a x. L'indirizzo di x viene visualizzato utilizzando lo specificatore di formato %p di printf(), che fa in modo che printf() visualizzi un indirizzo nel formato utilizzato dal computer.

### Aritmetica dei puntatori

Sui puntatori si possono utilizzare solo due operazioni aritmetiche: addizione e sottrazione. Per comprendere cosa avviene nell'aritmetica dei puntatori, si immagini che p1 sia un puntatore a un intero il cui valore attuale è 2000. Inoltre, si assumi che gli interi siano lunghi 2 byte. Dopo l'espressione

```
p1++;
```

p1 conterrà il valore 2002 e non 2001. Il motivo di ciò risiede nel fatto che ogni volta che si incrementa p1, esso deve puntare all'intero successivo. Lo stesso avviene per i decrementi. Ad esempio, se si immagina che p1 contenga il valore 2000, l'espressione

```
p1--;
```

fa in modo che p1 contenga il valore 1998.

Generalizzando l'esempio precedente, l'aritmetica dei puntatori è governata dalle seguenti regole. Ogni volta che si incrementa un puntatore, esso punterà alla locazione di memoria dell'elemento successivo considerando il tipo base. Ogni volta che il puntatore viene decrementato, punterà all'indirizzo dell'elemento precedente. Se l'aritmetica dei puntatori viene applicata a puntatori a caratteri, si ottiene la comune aritmetica in quanto i caratteri sono sempre lunghi un byte. Tutti gli altri puntatori verranno invece incrementati o decrementati della lunghezza del tipo di dati a cui essi puntano. Questo approccio assicura che un puntatore punti sempre a un elemento appropriato nel proprio tipo base. Questo concetto è illustrato dalla Figura 5.2.

La manipolazione dei puntatori non è limitata agli operatori di incremento e decremento. Ad esempio è possibile sommare o sottrarre interi a un puntatore. L'espressione:

```
p1 = p1 + 12;
```

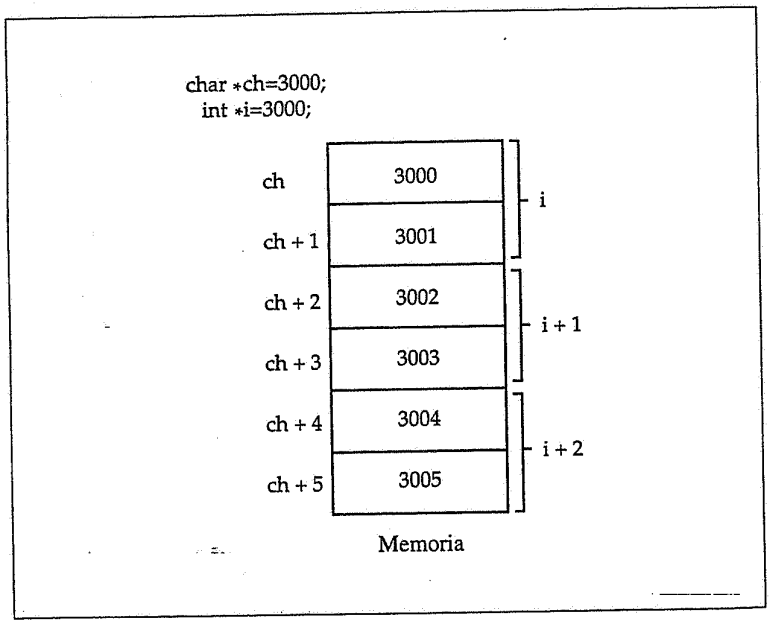



**Figura 5.2** Tutta l'aritmetica dei puntatori fa riferimento al tipo base.



fa in modo che p1 punti al dodicesimo elemento dello stesso tipo di p1 oltre a quello a cui attualmente punta.

Oltre all'addizione e alla sottrazione di un puntatore e di un intero, è consentita solo un'altra operazione aritmetica: è possibile sottrarre un puntatore a un altro puntatore per conoscere il numero degli oggetti del tipo base che separano i due puntatori. Tutte le altre operazioni aritmetiche sono proibite. In particolare, non si può moltiplicare o dividere puntatori; non è possibile sommare due puntatori; non è possibile applicare loro operatori bit-a-bit e non è possibile sommare o sottrarre valori di tipo float o double a o da puntatori.

## Confronti fra puntatori

I puntatori possono essere confrontati in un'espressione relazionale. Ad esempio, dati i due puntatori p e q, la seguente istruzione è perfettamente corretta:

```
if(p<q) printf("p punta a una cella di memoria inferiore rispetto a q\n");
```

Generalmente, il confronto dei puntatori è utilizzato quando due o più puntatori puntano a un oggetto comune, come ad esempio un array. Come esempio, sono state presentate due routine per la manipolazione dello stack che memorizzano e leggono valori interi. Uno stack è una struttura di elementi che utilizza l'accesso "first in - last out". Si può pensare a uno stack come a una pila di piatti su un tavolo: il primo piatto poggiato sul tavolo sarà l'ultimo a essere tolto. Gli stack sono molto impiegati nei compilatori, negli interpreti, nei fogli elettronici e in molto altro software di sistema. Per creare uno stack, occorre utilizzare due funzioni: push() e pop(). La funzione push() inserisce un valore nello stack e la funzione pop() estrae un valore. Queste routine sono illustrate nel listato seguente con una semplice funzione main(). Il programma inserisce in uno stack i valori immessi dall'utente. Se si immette un 0, verrà estratto un valore dallo stack. Il programma termina quando si immette il numero -1.

```
#include <stdio.h>
#include <stdlib.h>

#define SIZE 50

void push(int i);
int pop(void);

int  *tos, *p1, stack[SIZE];

int main(void)
```

```
{
  int value;
  tos = stack; /* tos punta alla cima dello stack */
  p1 = stack; /* inizializza p1 */

  do {
    printf("Immettere un valore: ");
    scanf("%d", &value);
    if(value!=0) push(value);
    else printf("Il valore in cima allo stack è %d\n", pop());
  } while(value!=-1);
}

void push(int i)
{
  p1++;
  if(p1==(tos+SIZE)) {
    printf("Superato il limite superiore dello stack");
    exit(1);
  }
  *p1 = i;
}

pop(void)
{
  if(p1==tos) {
    printf("Superato il limite inferiore dello stack");
    exit(1);
  }
  p1--;
  return *(p1+1);
}
```

Come si può vedere, la memoria che costituisce lo stack è formata dall'array stack. Il puntatore p1 punta al primo byte contenuto in stack. La variabile p1 consente di accedere allo stack. La variabile tos contiene l'indirizzo di memoria della cima dello stack. Il valore di tos consente di evitare di oltrepassare i limiti superiore e inferiore dello stack. Dopo l'inizializzazione dello stack, sarà possibile usare push() e pop(). Entrambe queste funzioni eseguono un test relazionale sul puntatore p1 per rilevare eventuali errori di superamento dei limiti. In push() si confronta il valore di p1 con l'indirizzo della fine dello stack, aggiungendo a tos il valore SIZE (le dimensioni dello stack). Questo evita di superare il limite massimo dello stack. In pop() si confronta con tos il valore di p1 per assicurarsi di non superare i limiti inferiori dello stack.



In pop() nell'istruzione return sono necessarie le parentesi. Senza di esse, l'istruzione avrebbe il seguente aspetto:

```
return *p1 +1;
```

In questa forma, l'istruzione restituisce il valore contenuto all'indirizzo p1 più 1 e non il valore contenuto nell'indirizzo p1+1.

## 5.5 Puntatori e array

Vi è una stretta relazione fra puntatori e array. Si consideri il seguente frammento di programma:

```
char str[80], *p1;
p1 = str;
```

Qui, p1 conterrà l'indirizzo del primo elemento dell'array str. Per accedere al quinto elemento di str si può scrivere:

```
str[4]
```

oppure

```
*(p1+4)
```

Entrambe queste istruzioni restituiranno il quinto elemento (occorre ricordare che gli array iniziano sempre da 0). Per accedere al quinto elemento si deve utilizzare 4 per indicizzare str. Ma si può anche aggiungere 4 al puntatore p1 in quanto p1 punta attualmente al primo elemento di str (occorre ricordare che il nome di un array senza alcun indice restituisce l'indirizzo iniziale dell'array, che corrisponde all'indirizzo del primo elemento).

L'esempio precedente può essere generalizzato. In pratica il C/C++ consente di usare due metodi per accedere agli elementi di un array: l'aritmetica dei puntatori e l'indicizzazione dell'array. Anche se la notazione a indici è più facile da comprendere, l'aritmetica dei puntatori può essere più veloce. Poiché la velocità è spesso molto importante in programmazione, per accedere agli elementi di un array i programmatori professionali utilizzano comunemente i puntatori.

Queste due versioni di putstr(), una delle quali utilizza l'indicizzazione dell'array e l'altra i puntatori, illustrano l'uso dei puntatori al posto dell'indicizzazione.

La funzione putstr() scrive una stringa sul dispositivo di output standard un carattere per volta.

```
/* Indicizza s come un array. */
void putstr(char *s)
{
  register int t;
  for(t=0; s[t]; ++t) putchar(s[t]);
}

/* Accede a s come un puntatore. */
void putstr(char *s)
{
  while(*s) putchar(*s++);
}
```

La maggior parte dei programmatori professionisti troverà la seconda versione più facile da leggere e da comprendere. Infatti, la versione a puntatori è il modo in cui questo tipo di routine viene comunemente scritta in C/C++.

### Array di puntatori

Anche i puntatori possono essere disposti in un array come qualsiasi altro tipo di dati. La dichiarazione di un array di puntatori a int di dimensione 10 è la seguente:

```
int *x[10];
```

Per assegnare l'indirizzo di una variabile intera chiamata var al terzo elemento dell'array di puntatori, si deve utilizzare l'istruzione

```
x[2] = &var;
```

Per conoscere il valore di var si deve utilizzare:

```
*x[2]
```

Se si deve passare un array di puntatori a una funzione, si può utilizzare lo stesso metodo già visto per il passaggio di altri tipi di array: semplicemente richiamare la funzione con il nome dell'array senza alcun indice. Ad esempio, una funzione che riceve l'array x avrà il seguente aspetto:

```
void display_array(int *q[])
{
  int t;

  for(t=0; t<10; t++)
    printf("%d ", *q[t]);
}
```



Si ricordi che q non è un puntatore a interi ma un puntatore a un array di puntatori a interi. Pertanto si deve dichiarare il parametro q come un array di puntatori a interi, come si è visto nell'esempio. Non è possibile dichiarare q semplicemente come un puntatore a interi poiché ciò è falso.

I puntatori ad array sono utilizzati molto spesso per contenere puntatori a stringhe. È possibile creare una funzione che visualizzi un messaggio di errore sulla base di un codice numerico:

```
void syntax_error(int num)
{
  static char *err[] = {
    "Impossibile aprire il file\n",
    "Errore di lettura\n",
    "Errore di scrittura\n",
    "Guasto al dispositivo\n"
  };

  printf("%s", err[num]);
}
```

L'array err contiene i puntatori ad ogni stringa. Come si può vedere, l'istruzione printf() che si trova all'interno di syntax_error() viene richiamata con un puntatore a caratteri che punta a uno dei vari messaggi di errore indicizzati sulla base del numero di errore passato alla funzione. Ad esempio, se num contiene il valore 2, verrà visualizzato il messaggio Errore di scrittura.

Può essere interessante sapere che l'argomento della riga di comando argv è un array di puntatori a carattere (vedere il Capitolo 6).

## 5.6 Indirizzamento multilivello

Esistono anche puntatori che puntano a un altro puntatore che punta al valore di destinazione. Una situazione di questo tipo è chiamata *indirizzamento multilivello* e si parla quindi di *puntatori a puntatori*. L'uso di puntatori a puntatori può essere fonte di confusione. La Figura 5.3 aiuta a chiarire il concetto di indirizzamento multilivello. Come si può vedere, il valore di un comune puntatore è l'indirizzo

dell'oggetto che contiene il valore desiderato. Nel caso di un puntatore a un puntatore, il primo puntatore contiene l'indirizzo del secondo puntatore che punta all'oggetto che contiene il valore desiderato.

L'indirizzamento multilivello può essere replicato fino al livello desiderato, ma raramente è necessario utilizzare più di un puntatore a un puntatore. Infatti, livelli di indirizzamento eccessivi risultano difficili da seguire e possono portare a errori concettuali.

**NOTA** *Non si deve confondere l'indirizzamento multilivello con le strutture di dati ad alto livello, come ad esempio le liste concatenate che utilizzano i puntatori. Si tratta di due concetti completamente diversi.*

Una variabile di tipo puntatore a puntatore deve essere dichiarata come tale. Si può fare ciò inserendo un ulteriore asterisco di fronte al nome della variabile. Ad esempio, la seguente dichiarazione dice al compilatore che newbalance è un puntatore a un puntatore a un oggetto di tipo float:

```
float **newbalance;
```

È fondamentale comprendere che newbalance non è un puntatore a un numero in virgola mobile ma un puntatore a un puntatore a un numero float.

Per accedere al valore di destinazione puntato in modo indiretto da un puntatore a un puntatore, si deve applicare per due volte l'operatore asterisco, come nell'esempio seguente:

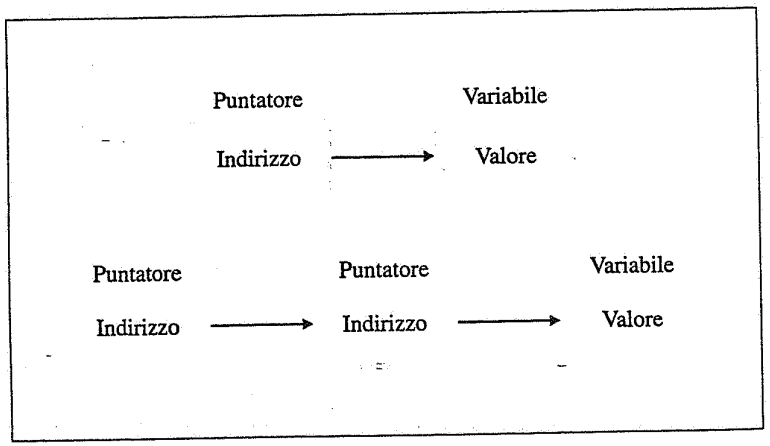



Figura 5.3- Indirizzamento semplice e multilivello.

```
#include <stdio.h>

int main(void)
{
  int x, *p, **q;

  x = 10;
  p = &x;
  q = &p;

  printf("%d", **q); /* stampa il valore di x */
}
```

Qui, p è dichiarato come puntatore a un intero e q come puntatore a un puntatore a un intero. La chiamata a printf() stampa sullo schermo il numero 10.

## 5.7 Inizializzazione di puntatori

Dopo la dichiarazione di un puntatore locale e prima che gli sia stato assegnato un valore, esso contiene un valore non noto (al contrario, i puntatori globali vengono automaticamente inizializzati a null).

**NOTA** *Se si tenta di utilizzare il puntatore prima di avergli assegnato un valore valido, si corre il rischio di bloccare il programma e, talvolta, persino il sistema operativo del computer: un tipo di errore veramente grave!*

Nell'utilizzo dei puntatori, la maggior parte dei programmatori C/C++ utilizza un'importante convenzione: un puntatore che attualmente non punta a un indirizzo di memoria valido deve avere valore nullo (zero). Per convenzione, ogni puntatore nullo si intende che non punti a nulla e non dovrebbe essere utilizzato. Tuttavia, il fatto che un puntatore abbia un valore nullo non lo rende "sicuro". Il nome di "puntatore nullo" non è che una convenzione fra programmatori. Non si tratta di una regola stabilita dal linguaggio C/C++. Ad esempio, se si utilizza un puntatore nullo sul lato sinistro di un'istruzione di assegnamento, si corre ancora il rischio di bloccare il programma o il sistema operativo.

Poiché si presume che un puntatore nullo sia inutilizzato, lo si può utilizzare per semplificare la codifica e aumentare l'efficienza delle routine che utilizzano puntatori. Ad esempio, si può utilizzare un puntatore nullo per indicare la fine di un array di puntatori. Una routine che accede a tale array saprà di averne incontrato la fine quando incontrerà il valore nullo. Questo tipo di approccio è illustrato dalla funzione search().

```
/* ricerca un nome */
int search(char *p[], char *name)
{
  register int t;

  for(t=0; p[t]; ++t)
    if(!strcmp(p[t], name)) return t;

    return -1; /* non trovato */
}
```

Il ciclo for all'interno di search() continua a essere ripetuto fino a che non viene trovata una corrispondenza o fino al raggiungimento del puntatore nullo. Se si assume che la fine dell'array sia indicata da un puntatore nullo, la condizione che controlla il ciclo diverrà falsa non appena verrà raggiunto il puntatore nullo.

I programmatori C/C++ inizializzano normalmente tutte le stringhe. Si è visto un esempio di ciò nella funzione syntax_error() nella sezione "Array di puntatori". Un'altra variante del tema dell'inizializzazione è il seguente tipo di dichiarazione di una stringa:

```
char *p = "ciao a tutti";
```

Come si può vedere, il puntatore p non è un array. Il motivo di questo tipo di inizializzazione risiede nel modo in cui opera il compilatore. Tutti i compilatori C/C++ creano una *tabella di stringhe*, utilizzata internamente dal compilatore per conservare le costanti di tipo stringa utilizzate nel programma. Pertanto, l'istruzione di dichiarazione precedente inserisce nel puntatore p l'indirizzo della stringa ciao a tutti che è memorizzata nella tabella delle stringhe. All'interno di un programma, p potrà essere utilizzata come qualsiasi altra stringa. Ad esempio, il seguente programma è perfettamente corretto:

```
#include <stdio.h>
#include <string.h>

char *p = "ciao a tutti";

int main(void)
{
  register int t;

  /* stampa la stringa in avanti e indietro */
  printf(p);
  for(t=strlen(p)-1; t>-1; t--) printf("%c", p[t]);
  return 0;
}
```

Nel C++ standard, il tipo di una stringa letterale è tecnicamente const char *. Ma il linguaggio C++ fornisce una conversione automatica in char *. Pertanto il programma precedente rimane valido. Tuttavia, questa conversione automatica è una funzionalità su cui è opportuno non fare affidamento quando si realizza nuovo codice. Nei nuovi programmi si deve sempre presumere che le stringhe letterali siano costanti e che la dichiarazione di p nel programma precedente debba essere scritta nel seguente modo:

```
const char *p = "ciao a tutti";
```

## 5.8 Puntatori a funzioni

Una funzionalità molto potente (ma anche fonte di confusione) del C++ è il concetto di *puntatore a funzione*. Anche se una funzione non è una variabile, essa ha un indirizzo fisico in memoria che può pertanto essere assegnato a un puntatore. L'indirizzo della funzione è il punto di accesso a tale funzione e dunque può essere utilizzato anche per richiamarla. Un puntatore che punta a una funzione può essere utilizzato per richiamarla. I puntatori a funzioni possono anche essere utilizzati come parametri di altre funzioni.

Per comprendere il funzionamento dei puntatori a funzione, è necessario conoscere di più sul modo in cui le funzioni vengono compilate e richiamate Innanzitutto, durante la compilazione di una funzione, il codice sorgente viene trasformato in codice oggetto e viene definito un punto di ingresso nella funzione. Quando viene eseguita una chiamata alla funzione durante l'esecuzione del programma, viene eseguita una chiamata in linguaggio macchina a tale punto di ingresso. Pertanto, se un puntatore contiene l'indirizzo del punto di ingresso di una funzione, potrà anche essere utilizzato per richiamare tale funzione.

È possibile conoscere l'indirizzo di una funzione utilizzando il nome della funzione senza parentesi o argomenti (corrisponde al modo in cui si ottiene l'indirizzo degli array: si usa solo il nome dell'array senza indici). Per vedere il funzionamento di ciò, si studi il seguente programma, ponendo particolare attenzione alle dichiarazioni:

```
#include <stdio.h>
#include <string.h>

void check(char *a, char *b,
           int (*cmp)(const char *, const char *));
```

```
int main(void)
{
  char s1[80], s2[80];
  int (*p)(const char *, const char *);

  p = strcmp;

  gets(s1);
  gets(s2);

  check(s1, s2, p);

  return 0;
}

void check(char *a, char *b,
           int (*cmp)(const char *, const char *))
{
  printf("controllo di uguaglianza\n");
  if(!(*cmp)(a, b)) printf("uguali");
  else printf("diversi");
}
```

Quando viene richiamata la funzione check(), come parametri vengono passati due puntatori a caratteri e un puntatore a funzione. All'interno della funzione check(), gli argomenti sono dichiarati come puntatori a carattere e puntatore a funzione. Si noti la dichiarazione del puntatore a funzione. Si deve utilizzare una forma simile quando si deve dichiarare ogni altro puntatore a funzione, anche se il tipo restituito dalla funzione può essere diverso. Le parentesi attorno a *cmp sono necessarie perché il compilatore interpreti correttamente questa istruzione.

All'interno di check(), l'espressione:

```
(*cmp)(a, b)
```

richiama strcmp(), puntata da cmp, con gli argomenti a e b. Anche in questo caso sono necessarie le parentesi attorno a *cmp. Questo esempio illustra anche il metodo generale per utilizzare un puntatore a funzione per richiamare la funzione puntata. Si può usare anche la seguente forma semplificata:

```
cmp(a, b);
```

Il motivo per cui si troverà con maggiore frequenza la prima forma è il fatto che rende palese per chiunque il fatto che la funzione viene richiamata tramite un



puntatore (ovvero che cmp è un puntatore a funzione e non il nome di una funzione). A parte questo le due espressioni sono equivalenti.

Si noti che si può richiamare check() utilizzando direttamente strcmp(), come si vede di seguito:

```
check (s1, s2, strcmp);
```

Questo elimina la necessità di utilizzare un'ulteriore variabile puntatore.

Ci si potrebbe chiedere perché qualcuno dovrebbe scrivere un programma in questo modo. Ovviamente non vi è alcun vantaggio e nell'esempio precedente si è introdotta un bel po' di confusione. Tuttavia, talvolta può essere vantaggioso passare le funzioni come parametri o creare un array di funzioni. Ad esempio, quando si scrive un compilatore o un interprete, il parser (la parte del compilatore che valuta le espressioni) richiama spesso varie funzioni di supporto, come ad esempio quelle che calcolano operazioni matematiche (seno, coseno, tangente e così via), quelle che eseguono operazioni di I/O o quelle che accedono a risorse del sistema. Invece di avere una grande istruzione switch contenente un elenco di tutte queste funzioni, si può creare un array di puntatori a funzione. In questo modo, si seleziona la funzione corretta in base a un indice. Si può vedere l'uso di questa tecnica studiando una versione espansa dell'esempio precedente. In questo programma, check() può controllare l'uguaglianza alfabetica o numerica richiamando semplicemente due funzioni di confronto diverse.

```
#include <stdio.h>
#include <ctype.h>
#include <stdlib.h>
#include <string.h>

void check(char *a, char *b,
           int (*cmp)(const char *, const char *));
int numcmp(const char *a, const char *b);

int main(void)
{
  char s1[80], s2[80];

  gets(s1);
  gets(s2);

  if(isalpha(*s1))
        check(s1, s2, strcmp);
  else
        check(s1, s2, numcmp);
```

```
  return 0;
}

void check(char *a, char *b,
           int (*cmp)(const char *, const char *))
{
  printf("controllo di uguaglianza\n");
  if(!(*cmp)(a, b)) printf("uguali");
  else printf("diversi");
}

int numcmp(const char *a, const char *b)
{
  if(atoi(a)==atoi(b)) return 0;
  else return 1;
}
```

In questo programma, se si immette una lettera, a check() viene passata strcmp(), altrimenti viene usata numcmp(). Poiché check() richiama la funzione che le viene passata, in questo modo si possono usare funzioni di confronto differenti a seconda dei casi.

## 5.9 Le funzioni di allocazione dinamica del C

I puntatori forniscono il supporto necessario per il potente sistema di allocazione dinamica del C. L'*allocazione dinamica* è il modo in cui un programma può ottenere memoria durante l'esecuzione. Come si è detto in precedenza, lo spazio di memoria delle variabili globali viene allocato al momento della compilazione. Le variabili locali utilizzano invece lo stack. Tuttavia, durante l'esecuzione del programma non è possibile aggiungere né variabili globali né variabili locali. Vi sono casi in cui le esigenze di memoria di un programma non possono essere determinate prima della sua esecuzione. Ad esempio, un word processor o un database dovranno poter utilizzare tutta la RAM disponibile nel sistema. Tuttavia, poiché la quantità di memoria RAM disponibile varia da computer a computer, non è possibile utilizzare per questi scopi le comuni variabili. Questi ed altri programmi dovranno quindi allocare memoria su richiesta. Il linguaggio C++ supporta due sistemi di allocazione dinamica: quello definito dal C e quello specifico del C++. Il sistema specifico del C++ contiene varie estensioni rispetto a quello utilizzato dal C ma questo approccio verrà descritto nella Parte seconda. In questa prima parte verranno invece descritte le funzioni di allocazione dinamica del C.

La memoria allocata dalle funzioni di allocazione dinamica del C è ottenuta dallo *heap*, la regione di memoria libera che si trova fra il programma (e la sua area di memoria permanente) e lo stack. Anche se le dimensioni dello heap non sono note, si può presumere che esso contenga una grande quantità di memoria libera.



Il nucleo del sistema di allocazione dinamica del C è formato dalle funzioni malloc() e free(). La maggior parte dei compilatori forniscono molte altre funzioni di allocazione dinamica, ma queste due sono le più importanti. Queste funzioni operano insieme utilizzando la regione di memoria libera per definire e gestire un elenco delle celle di memoria disponibili. La funzione malloc() alloca memoria mentre free() la rende nuovamente disponibile. Questo significa che ogni volta che viene eseguita una richiesta di memoria con malloc(), viene allocata una porzione della memoria che precedentemente era libera. Ogni volta che viene chiamata la funzione free(), la memoria viene restituita al sistema. Ogni programma che utilizza queste funzioni deve includere il file header stdlib.h (un programma C++ può utilizzare il nuovo file header <cstdlib>).

Il prototipo della funzione malloc() è il seguente:

void *malloc(size_t *numero_di_byte*);

Qui, *numero_di_byte* è il numero di byte di memoria che si intende allocare. Il tipo size_t è definito in stdlib.h come (più o meno) un intero unsigned. La funzione malloc() restituisce un puntatore di tipo void; questo significa che è possibile assegnarlo a ogni tipo di puntatore. Dopo una chiamata avvenuta con successo, malloc() restituisce un puntatore al primo byte della regione di memoria allocata nello heap. Se la memoria disponibile non è sufficiente per soddisfare le richieste di malloc(), si verifica un errore di allocazione e la funzione malloc() restituisce un puntatore nullo.

Il frammento di codice mostrato di seguito alloca 1000 byte contigui di memoria:

```
char *p;
p = malloc(1000); /* chiede 1000 byte */
```

Dopo l'assegnamento, p punta all'inizio dei 1000 byte di memoria libera.

Si noti che, per assegnare a p il valore restituito da malloc() non è richiesta alcuna conversione di tipo (cast). In C un puntatore void * viene automaticamente convertito nel tipo di puntatore che si trova sul lato sinistro dell'assegnamento. Questa conversione automatica *non avviene* invece in C++. In particolare, in C++, per assegnare un puntatore void * a un altro tipo di puntatore è necessario specificare una conversione di tipo esplicita. Pertanto, in C++, l'assegnamento precedente si sarebbe dovuto scrivere come:

```
p = (char *) malloc(1000);
```

Come regola generale, quando in C++ si deve assegnare (o convertire) un tipo di puntatore in un altro si deve sempre impiegare una conversione cast. Questa è una delle differenze più importanti fra il C e il C++.

Il prossimo esempio alloca uno spazio per 50 interi. Si noti l'uso di sizeof per garantire la trasportabilità.

```
int *p;
p = (int *)malloc(50*sizeof(int));
```

Poiché lo heap non è infinito, quando si alloca memoria, si deve controllare il valore restituito da malloc() per assicurarsi che non sia un puntatore nullo, prima di impiegarlo. L'uso di un puntatore nullo provoca quasi certamente un blocco del programma. Il modo corretto per allocare memoria e verificare la validità del puntatore restituito è illustrato dal seguente frammento di codice.

```
p = (int *) malloc(100);
if(!p) {
  printf("Memoria esaurita. \n");
  exit(1);
}
```

Naturalmente, al posto della chiamata a exit() si può utilizzare un qualsiasi altro gestore di errori. Occorre semplicemente assicurarsi di non utilizzare il puntatore p nel caso in cui questo sia nullo.

La funzione free() è l'esatto opposto di malloc() in quanto restituisce al sistema la memoria precedentemente allocata. Dopo aver liberato un'area di memoria, essa potrà essere riutilizzata con una successiva chiamata a malloc(). Il prototipo della funzione free() è il seguente:

void free(void *p);

Qui, *p* è un puntatore a memoria che era stata precedentemente allocata utilizzando la funzione malloc(). È fondamentale richiamare *sempre* free() con un argomento valido, in caso contrario, si distruggerà l'elenco di gestione della memoria libera.

## 5.10 Problemi con i puntatori

Nulla può dare più problemi di un puntatore a zonzo! I puntatori sono un'arma a doppio taglio. Forniscono immense potenzialità e sono necessari per molti programmi. Allo stesso tempo, quando un puntatore contiene un valore errato, può dare origine a uno dei bug più difficili da scovare.



Un puntatore errato è difficile da scovare poiché il problema non risiede, in effetti, nel puntatore. Il problema è che quando si esegue un'operazione con un puntatore errato, si legge o scrive su una locazione di memoria. Se l'operazione è di lettura, il peggio che può capitare è di leggere oggetti senza senso. Se invece l'operazione è di scrittura, si potrebbe iniziare a scrivere su parti del codice o su altri dati. Questo genere di problema potrebbe quindi palesarsi solo molto più avanti nell'esecuzione del programma e potrebbe quindi condurre il programmatore a ricercare il bug nel luogo errato. Potrebbe non esservi alcun indizio a suggerire che un puntatore sia la causa del problema. Questo tipo di bug è il problema più temuto dai programmatori.

Poiché gli errori che vedono protagonisti i puntatori sono così problematici, occorre fare del proprio meglio per evitarli. Questa parte del capitolo si occupa di alcuni dei più comuni errori che vedono come protagonisti i puntatori. L'esempio classico di errore dovuto a un puntatore riguarda i *puntatori non inizializzati*.

Si consideri il seguente programma:

```
/* Questo programma è errato. */
int main(void)
{
  int x, *p;

  x = 10;
  *p = x;

  return 0;
}
```

Questo programma assegna il valore 10 a una locazione di memoria sconosciuta. Ecco il motivo: poiché il puntatore p non ha mai ricevuto alcun valore, nel momento in cui viene eseguito l'assegnamento *p=x, conterrà un valore ignoto. In questo modo, il valore di x verrà scritto in una locazione di memoria sconosciuta. Questo tipo di problema passa spesso inosservato quando il programma è piccolo poiché le casualità sono a favore di p che potrebbe in ogni caso contenere un indirizzo "sicuro" (ovvero un indirizzo che non si trovi nel codice, nell'area dati o nel sistema operativo). Tuttavia, quando il programma comincia ad essere piuttosto lungo, aumentano le probabilità che p punti a qualche informazione vitale. Nel caso peggiore, il programma si bloccherà. La soluzione consiste nell'assicurarsi sempre che un puntatore punti a un indirizzo valido, prima di utilizzarlo.

Un secondo errore molto comune è dovuto a una semplice incomprensione sull'uso dei puntatori. Si consideri il seguente programma:

```
/* Questo programma è errato. */
#include <stdio.h>

int main(void)
{
  int x, *p;

  x = 10;
  p = x;

  printf("%d", *p);

  return 0;
}
```

La chiamata a printf() non visualizza il valore di x, che è 10, ma un valore sconosciuto, in quanto l'assegnamento:

```
p = x;
```

è errato. Questa istruzione assegna il valore 10 al puntatore p. Ma p deve contenere un indirizzo, non un valore. Per correggere il programma, basta scrivere:

```
p = &x;
```

Un altro errore che può verificarsi è dovuto a un'assunzione errata a proposito della posizione delle variabili in memoria. Non si può mai sapere dove i dati si troveranno in memoria o se verranno nuovamente posizionati nello stesso luogo o se ogni compilatore li tratterà nello stesso modo. Per questo motivo, un confronto fra puntatori che non puntino a un oggetto comune può portare a risultati imprevedibili. Ad esempio,

```
char s[80], y[80];
char *p1, *p2;

p1 = s;
p2 = y;
if(p1 < p2) . . .
```

è normalmente un concetto errato. Si tratta di una situazione piuttosto insolita, si potrebbe utilizzare qualcosa del genere per determinare la posizione relativa delle variabili, ma questo è un caso raro.



Un errore correlato sorge quando si assume che due array adiacenti possano essere indicizzati come se formassero un solo array, incrementando semplicemente un puntatore anche quando giunge ai limiti dell'array. Ad esempio,

```
int first[10], second[10];
int *p, t;

p = first;
for(t=0; t<20; ++t)  *p++ = t;
```

Questo non è un buon modo per inizializzare gli array first e second con i numeri da 0 a 19. Anche se potrebbe funzionare con alcuni compilatori e in alcuni casi. Questo frammento di programma assume che gli array siano posizionati fianco a fianco in memoria a partire da first. In qualche caso questo potrebbe non essere vero.

Il programma successivo illustra un tipo di bug molto pericoloso. Prima di procedere nella lettura, si provi a ricercarlo da soli.

```
/* Questo programma contiene un bug. */
#include <string.h>
#include <stdio.h>

int main(void)
{
  char *p1;
  char s[80];

  p1 = s;
  do {
    gets(s);  /* legge una stringa */

    /* stampa l'equivalente decimale
       di ogni carattere */
    while(*p1) printf(" %d", *p1++);

  } while(strcmp(s, "fine"));

  return 0;
}
```

Questo programma utilizza p1 per stampare i valori ASCII associati ai caratteri contenuti in s. Il problema consiste nel fatto che a p1 viene assegnato l'indirizzo di s una sola volta. La prima volta che si entra nel ciclo, p1 punta al primo

carattere contenuto in s. La seconda volta, continua da dove era rimasto, poiché non viene riposizionato all'inizio di s. Questo successivo carattere può appartenere alla seconda stringa, a un'altra variabile o anche a una porzione del programma!

Il modo corretto per scrivere questo programma è il seguente:

```
/* Questo è il programma corretto. */
#include <string.h>
#include <stdio.h>

int main(void)
{
  char *p1;
  char s[80];

  do {
    p1 = s;
    gets(s);  /* legge una stringa */

    /* stampa l'equivalente decimale
       di ogni carattere */
    while(*p1) printf(" %d", *p1++);

  } while(strcmp(s, "fine"));

  return 0;
}
```

Qui, ad ogni iterazione del ciclo, a p1 viene assegnato l'inizio della stringa. In generale, si deve ricordare di reinizializzare un puntatore ogni volta che lo si riutilizza.

Il fatto che la gestione errata dei puntatori possa provocare bug difficili da scovare non è un buon motivo per evitare l'uso dei puntatori. Occorre semplicemente fare attenzione e assicurarsi di sapere dove punta un puntatore prima di utilizzarlo.

Capitolo 6

# Le funzioni


6.1 La forma generale di una funzione
6.2 Regole di visibilità delle funzioni
6.3 Gli argomenti delle funzioni
6.4 Gli argomenti di main(): argc e argv
6.5 L'istruzione return
6.6 Ricorsione
6.7 Prototipi di funzioni
6.8 Dichiarazione di elenchi di parametri di lunghezza variabile
6.9 Dichiarazione di parametri con metodi vecchi e nuovi
6.10 Elementi implementativi


Le funzioni sono i blocchi fondamentali che costituiscono il C/C++ e sono il luogo in cui si svolgono tutte le attività del programma. Questo capitolo esamina le funzionalità delle funzioni C, incluso il passaggio di parametri, la restituzione di un valore e la ricorsione. Nella Parte seconda verranno discusse tutte le funzionalità tipiche delle funzioni C++, come ad esempio l'overloading e i parametri indirizzo.

## 6.1 La forma generale di una funzione

La forma generale di funzione è la seguente:

```
tipo_restituito nome_funzione(elenco_parametri)
{
  corpo_della_funzione
}
```

il *tipo_restituito* specifica il tipo di dati restituito dalla funzione. Una funzione può restituire un qualsiasi tipo di dati tranne un array. L'*elenco_parametri* è un elenco di nomi di variabili separati da virgole e seguiti dai rispettivi tipi che ricevono i valori degli argomenti nel momento in cui la funzione viene richiamata.

Una funzione può anche essere senza parametri, in tal caso l'elenco dei parametri sarà vuoto. Tuttavia, anche se non vi è alcun parametro, è richiesta la presenza delle parentesi.

Nella dichiarazione di variabili, è possibile dichiarare più variabili di un tipo comune, utilizzando un elenco di nomi di variabili separati da virgole. Al contrario, tutti i parametri della funzione devono essere dichiarati singolarmente e per ognuno si deve specificare sia il tipo che il nome. Questo significa che l'elenco di dichiarazioni di parametri per una funzione ha la seguente forma generale:

*f(tipo nomevar1, tipo nomevar2, . . . , tipo nomevarN)*

Ad esempio, ecco un modo corretto ed uno errato per dichiarare i parametri delle funzioni:

```
f(int i, int k, int j)        /* corretta */
f(int i, k, float j)          /* errata */
```

## 6.2 Regole di visibilità delle funzioni

Le *regole di visibilità* di un linguaggio sono le regole che governano ciò che una parte del codice conosce e le possibilità di accesso che ha su un'altra parte di codice o dati.

Ogni funzione forma un blocco unitario di codice. Il codice della funzione è privato e appartiene a tale funzione; non è possibile quindi accedere a un'istruzione in nessun'altra funzione se non tramite una chiamata a tale funzione (ad esempio, non è possibile usare un goto per saltare all'interno di un'altra funzione). Il codice che forma il corpo di una funzione è nascosto al resto del programma e, se non si usano variabili o dati globali, non può influire né subire influssi da altri parti del programma. In altre parole, il codice e i dati che sono definiti all'interno di una funzione non possono interagire con il codice o i dati definiti in un'altra funzione in quanto le due funzioni hanno diverse aree di visibilità.

Le variabili che sono definite all'interno di una funzione sono chiamate *variabili locali*. Una variabile locale nasce nel momento in cui si entra nella funzione e viene distrutta alla sua uscita. Questo significa che le variabili locali non possono conservare il proprio valore fra due chiamate di funzione. L'unica eccezione a questa regola si verifica quando la variabile è dichiarata con lo specificatore di classe di memorizzazione static. In questo caso, il compilatore tratterà la variabile come se fosse una variabile globale per quanto riguarda la memorizzazione, ma continuerà a limitare la sua visibilità solo all'interno della funzione (questo argomento è stato descritto approfonditamente nel Capitolo 2).



In C (e in C++) non è possibile definire una funzione all'interno di un'altra funzione. Questo è il motivo per cui né il C né il C++ sono tecnicamente linguaggi strutturati a blocchi.

## 6.3 Gli argomenti delle funzioni

Se una funzione deve utilizzare degli argomenti, deve dichiarare delle variabili che accettino i valori degli argomenti. Queste variabili sono chiamate i *parametri formali* della funzione. Essi si comportano come ogni altra variabile locale della funzione e vengono creati all'ingresso nella funzione e distrutti all'uscita. Come si può vedere nella seguente funzione, la dichiarazione dei parametri si trova subito dopo il nome della funzione:

```
/* Restituisce 1 se c si trova nella stringa s; altrimenti restituisce 0. */
int is_in(char *s,  char c)
{
  while(*s)
    if(*s==c) return 1;
    else s++;
  return 0;
}
```

La funzione is_in() ha due parametri: s e c. Questa funzione restituisce 1 se il carattere c si trova nella stringa s; in caso contrario, la funzione restituisce 0.

Come nel caso delle variabili locali, è possibile eseguire assegnamenti ai parametri formali di una funzione oppure utilizzarli in qualsiasi espressione. Anche se queste variabili eseguono il compito speciale di ricevere il valore degli argomenti passati alle funzioni, è comunque possibile utilizzarle come ogni altra variabile locale.

### Chiamata per valore e chiamata per indirizzo

In un linguaggio di programmazione è possibile passare gli argomenti alle subroutine in due modi. Il primo è detto *chiamata per valore*. Questo metodo copia il *valore* di un argomento nel parametro formale della subroutine. In questo caso, ogni modifica eseguita sul parametro non avrà alcun effetto sull'argomento passato.

Il secondo metodo per passare argomenti a una subroutine è la *chiamata per indirizzo*. Con questo metodo, nel parametro viene copiato l'*indirizzo* dell'argomento. All'interno della subroutine, è possibile utilizzare l'indirizzo per accedere

all'effettivo argomento utilizzato nella chiamata. Questo significa che le modifiche eseguite sul parametro avranno effetto anche sull'argomento.

Normalmente il C/C++ utilizza la chiamata per valore per il passaggio degli argomenti. In generale, questo significa che il codice all'interno di una funzione non può modificare gli argomenti utilizzati per richiamare la funzione. Si consideri il seguente programma:

```
#include <stdio.h>

int sqr(int x);

int main(void)
{
  int t=10;

  printf("%d %d", sqr(t), t);

  return 0;
}

int sqr(int x)
{
  x = x*x;
  return(x);
}
```

In questo esempio, il valore dell'argomento di sqrt(), 10, viene copiato nel parametro x. Quando viene eseguito l'assegnamento x=x*x, solo la variabile locale x viene modificata. La variabile t utilizzata per richiamare sqr() continuerà a contenere il valore 10. Pertanto l'output sarà 100 10.

**NOTA** *Alla funzione viene passata solo una copia del valore dell'argomento. Ciò che avviene all'interno della funzione non ha alcun effetto sulla variabile utilizzata nella chiamata.*

## Creazione di una chiamata per indirizzo

Anche se la convenzione di passaggio di parametri del C/C++ prevede la chiamata per valore, è possibile creare una chiamata per indirizzo passando alla funzione un puntatore a un argomento al posto dell'argomento stesso. Poiché alla funzione viene passato l'indirizzo dell'argomento, il codice che si trova all'interno della funzione può modificare il valore dell'argomento che si trova all'esterno della funzione stessa.



I puntatori possono essere passati a una funzione come ogni altro valore. Naturalmente, è necessario dichiarare i parametri di tipo puntatore. Ad esempio, la funzione swap(), che scambia i valori delle due variabili intere puntate dai suoi argomenti potrebbe avere il seguente aspetto:

```
void swap(int *x, int *y)
{
  int temp;

  temp = *x;  /* salva il valore all'indirizzo x */
  *x = *y;    /* copia y in x */
  *y = temp;  /* copia x in y */
}
```

swap() è in grado di scambiare i valori delle due variabili puntate da x e y perché le vengono passati gli indirizzi delle variabili e non i valori. Pertanto, all'interno della funzione, sarà possibile accedere al contenuto delle variabili utilizzando le comuni operazioni sui puntatori. Questo è il motivo per cui è possibile scambiare il contenuto delle variabili utilizzate per chiamare la funzione.

Si ricordi che swap() (e ogni altra funzione che utilizza parametri puntatore) deve essere richiamata utilizzando gli *indirizzi degli argomenti*. Il seguente programma mostra il modo corretto per richiamare swap():

```
void swap(int *x, int *y);

int main(void)
{
  int i, j;

  i = 10;
  j = 20;

  swap(&i, &j); /* passa gli indirizzi di i e j */
}
```

In questo esempio, alla variabile i viene assegnato il valore 10 e a j viene assegnato il valore 20. Quindi viene chiamata swap() con gli indirizzi di i e j. Per produrre gli indirizzi delle variabili viene utilizzato l'operatore unario &. Pertanto, alla funzione swap() vengono passati gli indirizzi di i e j e non i rispettivi valori.

**NOTA** *Il linguaggio C++ consente di automatizzare completamente una chiamata per indirizzo tramite parametri indirizzo. Questa funzionalità verrà descritta nella Parte seconda.*

## Chiamate a funzione e array

Gli array sono stati trattati in dettaglio nel Capitolo 4. Questa sezione si occupa del passaggio degli array come argomenti a funzioni in quanto questa è una eccezione alla regola di passaggio dei parametri.

Quando si utilizza un array come argomento di una funzione, alla funzione ne viene passato solo l'indirizzo. Questa è un'eccezione alla convenzione di chiamata per valore degli argomenti. In questo caso, il codice che si trova all'interno della funzione opera e può modificare il contenuto effettivo dell'array utilizzato per richiamare la funzione. Ad esempio, si consideri la funzione print_upper() che stampa in lettere maiuscole la stringa passata come argomento:

```
#include <stdio.h>
#include <ctype.h>

void print_upper(char *string);

int main(void)
{
  char s[80];

  gets(s);
  print_upper(s);
  printf("in maiuscolo: %s", s);

  return 0;
}

/* Stampa una stringa in lettere maiuscole. */
void print_upper(char *string)
{
  register int t;

  for(t=0; string[t]; ++t)  {
    string[t] = toupper(string[t]);
    putchar(string[t]);
  }
}
```

Dopo la chiamata a print_upper(), il contenuto dell'array s in main() sarà la stessa stringa ma in lettere maiuscole. Se questo non è il comportamento desiderato, si potrebbe riscrivere il programma nel modo seguente:

```
#include <stdio.h>
#include <ctype.h>

void print_upper(char *string);

int main(void)
{
  char s[80];

  gets(s);
  print_upper(s);
  printf("non viene modificata: %s", s);

  return 0;
}

void print_upper(char *string)
{
  register int t;

  for(t=0; string[t]; ++t)
    putchar(toupper(string[t]));
}
```



In questa versione, il contenuto dell'array s non viene toccato in quanto print_upper() non modifica i valori che esso contiene.

La funzione gets() contenuta nella libreria standard è un classico esempio di passaggio di array a una funzione. Anche se la funzione gets() della libreria standard è molto più sofisticata, la seguente versione semplificata, chiamata xgets() può dare un'idea del suo funzionamento.

```
/* Una versione semplificata
   della funzione gets() della libreria standard */
char *xgets(char *s)
{
  char ch, *p;
  int t;

  p = s;  /* gets() restituisce un puntatore a s */

  for(t=0; t<80; ++t){
    ch = getchar();

    switch(ch) {
      case '\n':
        s[t] = '\0'; /* conclude
                        la stringa */
        return p;
```

```
      case '\b':
        if(t>0) t--;
        break;
      default:
        s[t] = ch;
    }
  }
  s[79] = ' \0';
  return p;
}
```

La funzione xgets() può essere richiamata con un puntatore a carattere, che naturalmente può essere anche il nome di un array di caratteri che, per definizione, è un puntatore a carattere. Subito dopo l'ingresso in xgets() si trova un ciclo for da 0 a 80. Questo impedisce l'immissione di stringhe troppo lunghe. Se si cerca di immettere più di 80 caratteri, la funzione termina con un return. La vera funzione gets() non ha questo limite. Poiché il C/C++ non prevede verifiche dei limiti di un array, è necessario assicurarsi che qualsiasi array utilizzato per richiamare xgets() possa accettare almeno 80 caratteri. Mano a mano che si immetteranno caratteri alla tastiera, essi verranno sistemati nella stringa. Se si preme il tasto BACKSLASH, il contatore t verrà ridotto di 1, cancellando in effetti dall'array il carattere precedente. Quando si preme INVIO, al termine della stringa viene inserito un carattere nullo che ne indica la fine. Poiché viene modificato l'array utilizzato per richiamare xgets(), all'uscita dalla funzione l'array conterrà i caratteri immessi.

## 6.4 Gli argomenti di main(): argc e argv

Talvolta può essere utile passare informazioni a un programma al momento dell'esecuzione. Generalmente, si passano informazioni alla funzione main() attraverso gli argomenti della riga di comando. Un *argomento della riga di comando* è formato dalle informazioni che seguono il nome del programma sulla riga di comando del sistema operativo. Ad esempio, quando si compilano programmi C, si potrebbe utilizzare un comando simile al seguente:

cc *nome_programma*

dove *nome_programma* è un argomento della riga di comando che specifica il nome del programma che si intende compilare. Il C prevede l'uso di due speciali argomenti: argv e argc, utilizzati per ricevere gli argomenti dalla riga di comando. Il parametro argc è un intero che contiene il numero di argomenti che si trovano nella riga di comando. Il suo valore è sempre almeno pari a 1 in quanto il nome



del programma viene considerato come primo argomento. Il parametro argv è un puntatore a un array di puntatori a caratteri. Ogni elemento di questo array punta a un argomento della riga di comando. Tutti gli argomenti della riga di comando sono stringhe ed i numeri dovranno pertanto essere convertiti nel formato interno corretto. Ad esempio, questo semplice programma stampa sullo schermo Salve e il nome immesso come argomento.

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
  if(argc!=2) {
    printf("Hai dimenticato di scrivere il nome.\n");
    exit(1);
  }
  printf("Salve %s", argv[1]);

  return 0;
}
```

Se si chiama questo programma nome e si immette il nome Alice, si potrà richiamare il programma con il comando nome Alice. L'output del programma sarà quindi Salve Alice.

In molti ambienti operativi, i vari argomenti della riga di comando devono essere separati da uno spazio o da un carattere di tabulazione. Le virgole, i punti e virgola e gli altri segni di punteggiatura non sono considerati separatori.

Ad esempio,

```
run Spot, run
```

è formato da tre stringhe mentre:

```
Mario,Luigi,Pietro
```

è una singola stringa in quanto le virgole non vengono normalmente considerate come separatori.

Alcuni ambienti consentono di racchiudere una stringa contenente spazi fra doppi apici. In questo modo l'intera stringa verrà trattata come un unico argomento. Per informazioni sull'uso dei parametri della riga di comando, si consulti la documentazione del sistema operativo.

Il parametro argv deve essere dichiarato correttamente. Il metodo più comune è il seguente:

```
char *argv[];
```

Le parentesi quadre vuote indicano che l'array è di lunghezza non determinata. In questo modo sarà possibile accedere ai singoli argomenti indicizzando argv. Ad esempio, argv[0] punta alla prima stringa che è sempre il nome del programma; argv[1] punta al primo argomento e così via.

Un altro breve esempio che utilizza gli argomenti della riga di comando è il programma chiamato countdown mostrato di seguito. Il programma esegue il conto alla rovescia partendo da un valore iniziale (specificato nella riga di comando) ed emette un segnale acustico quando raggiunge lo zero. Si noti che il primo argomento contenente il numero viene convertito in un intero dalla funzione standard atoi(). Se come secondo argomento si utilizza la stringa "display", il conto alla rovescia verrà anche visualizzato sullo schermo.

```
/* Programma Countdown. */
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include <string.h>

int main(int argc, char *argv[])
{
  int disp, count;

  if(argc<2) {
    printf("Immetti il valore di partenza\n");
    printf("nella riga di comando. Riprova.\n");
    exit(1);
  }

  if(argc==3 && !strcmp(argv[2], "display")) disp = 1;
  else disp = 0;

  for(count=atoi(argv[1]); count; --count)
    if(disp) printf("%d\n", count);

  putchar('\a');  /* questa istruzione
                     emette un segnale acustico */
  printf("Fine");

  return 0;
}
```

Si noti che se non si specificano argomenti della riga di comando, viene visualizzato il messaggio appropriato. Questo è un comportamento classico: un programma che utilizza gli argomenti della riga di comando, spesso visualizza qualche riga di istruzioni se l'utente cerca di eseguire il programma senza aver immesso le informazioni corrette.

Per accedere a un singolo carattere di uno degli argomenti del comando, basta aggiungere un secondo indice ad argv. Ad esempio, il programma seguente visualizza tutti gli argomenti con cui è stato chiamato, un carattere per volta:

```
#include <stdio.h>

int main(int argc, char *argv[])
{
  int t, i;

  for(t=0; t<argc; ++t) {
    i = 0;

    while(argv[t][i]) {
      putchar(argv[t][i]);
      ++i;
    }
    printf("\n");
  }

  return 0;
}
```

Si ricordi che il primo indice accede alla stringa e il secondo accede ai singoli caratteri della stringa.

Normalmente, si utilizzano argc e argv per fornire dei comandi iniziali al programma. In teoria, è possibile specificare fino a 32767 argomenti, ma la maggior parte dei sistemi operativi ne prevede molti meno. Normalmente si utilizzano questi argomenti per indicare il nome di un file o un'opzione. L'uso degli argomenti nella riga di comando può dare a un programma un aspetto professionale e ne facilita l'uso all'interno di file batch.

Quando un programma non richiede parametri sulla riga di comando, è pratica comune dichiarare esplicitamente main() senza parametri. Nel caso dei programmi C si deve specificare la parola chiave void nell'elenco dei parametri (questo è l'approccio utilizzato nei programmi della Parte prima di questo volume). In C++ basta semplicemente specificare un elenco di parametri vuoto (in C++ l'utilizzo di void per indicare un elenco di parametri vuoto è ridondante).

I nomi argc e argv sono tradizionali ma arbitrari. In pratica, si possono utilizzare altri nomi a scelta. Inoltre, alcuni compilatori sono in grado di gestire ulteriori argomenti di main(), quindi è sempre bene controllare sul manuale dell'utente.

## 6.5 L'istruzione return

L'istruzione return è già stata descritta nel Capitolo 3. Essa ha due importanti utilizzi. Innanzi tutto, provoca l'immediata uscita dalla funzione in cui si trova. In pratica, fa in modo che l'esecuzione del programma ritorni al codice chiamante. In secondo luogo, return può essere utilizzata per restituire il valore della funzione.

### Uscita da una funzione

Vi sono due modi in cui una funzione può terminare la propria esecuzione e tornare al chiamante. La prima si verifica quando viene raggiunta l'ultima istruzione della funzione e viene incontrata la parentesi graffa di chiusura della funzione (naturalmente, la parentesi graffa non è in effetti presente all'interno del codice oggetto ma può essere conveniente ragionare in questo modo). Ad esempio, la funzione pr_reverse() contenuta in questo programma visualizza sullo schermo invertendo le lettere la stringa "Mi piace il C" e poi semplicemente termina.

```
#include <string.h>
#include <stdio.h>

void pr_reverse(char *s);

int main(void)
{
  pr_reverse("Mi piace il C++");

  return 0;
}

void pr_reverse(char *s)
{
  register int t;

  for(t=strlen(s)-1; t>=0; t--) putchar(s[t]);
}
```

Dopo la visualizzazione della stringa, il codice di pr_reverse() termina e quindi l'esecuzione ritorna al punto in cui la funzione era stata chiamata.

Nella pratica, sono poche le funzioni che utilizzano questo metodo per terminare al propria esecuzione. La maggior parte delle funzioni utilizza l'istruzione return che consente di restituire un valore e di semplificare e di rendere più efficiente il codice di uscita dalla funzione.



Si ricordi, una funzione può contenere più istruzioni return. Ad esempio, la funzione find_substr() contenuta nel seguente programma restituisce la posizione iniziale della sottostringa contenuta in una stringa principale o restituisce -1 se non viene trovata alcuna corrispondenza.

```
#include <stdio.h>

int find_substr(char *s1, char *s2);

int main(void)
{
  if(find_substr("Mi piace il C", "ac") != -1)
    printf("sottostringa presente nella stringa");

  return 0;
}

/* Restituisce l'indice della prima corrispondenza di s2 in s1. */
find_substr(char *s1, char *s2)
{
  register int t;
  char *p, *p2;

  for(t=0; s1[t]; t++) {
    p = &s1[t];
    p2 = s2;

    while(*p2 && *p2 ==*p) {
      p++;
      p2++;
    }
    if(!*p2) return t; /* Primo return */
  }
  return -1; /* secondo return */
}
```

### Restituzione di valori

Tutte le funzioni, tranne quelle di tipo void, restituiscono un valore. Questo valore è specificato tramite un'istruzione return. In C, se una funzione non-void non restituisce esplicitamente un valore tramite un'istruzione return, verrà restituito un valore senza senso. In C++ una funzione non-void deve contenere un'istruzione return la quale deve restituire un valore. Questo significa che in C++ se una funzione specifica la restituzione di un valore, tutte le istruzioni return in essa

contenute devono restituire un valore. Se l'esecuzione raggiunge la fine di una funzione non-void, verrà restituito un valore senza senso. Anche se questa non è una condizione di errore di sintassi, si tratta di una situazione che dovrebbe essere evitata. Se una funzione non è dichiarata come void, sarà possibile utilizzarla come un qualsiasi altro operando in ogni espressione valida. Pertanto, ognuna delle seguenti espressioni è perfettamente lecita:

```
power(y);
if(max(x,y) > 100) printf("maggiore");
for(ch=getchar(); isdigit(ch); ) ... ;
```

Tuttavia, come regola generale, una funzione non può essere la destinazione di un assegnamento. Un'istruzione come:

```
swap(x,y) = 100; /* istruzione errata */
```

è errata. Il compilatore C/C++ la evidenzierà come un errore e non compilerà un programma che la contenga (come si vedrà nella seconda parte di questa guida, il C++ presenta alcune interessanti eccezioni a questa regola generale che consentono di posizionare alcuni tipi di funzioni sul lato sinistro di un'istruzione di assegnamento).

Quando si scrivono programmi, le funzioni generalmente sono di tre tipi. Il primo tipo è di semplice calcolo. Queste funzioni sono progettate con lo scopo di eseguire operazioni sui propri argomenti e restituire un valore sulla base di tale operazione. Una funzione di calcolo è una funzione "pura". Esempi di questo tipo sono le funzioni sqrt() e sin() della libreria standard, che calcolano rispettivamente la radice quadrata e il seno dei propri argomenti.

Il secondo tipo di funzioni manipola le informazioni e restituisce un valore che indica semplicemente il successo o il fallimento di tale manipolazione. Un esempio è la funzione di libreria fclose() utilizzata per chiudere un file. Se l'operazione di chiusura ha successo, la funzione restituisce zero; se l'operazione non ha successo, la funzione restituisce EOF.

L'ultimo tipo di funzione non ha alcun valore esplicito da restituire. In pratica, la funzione è strettamente procedurale e non produce alcun valore. Un esempio è la funzione exit() che termina il programma. Tutte le funzioni che non restituiscono alcun valore devono essere dichiarate di tipo void. Dichiarando una funzione come void si evita che possa essere utilizzata in un'espressione, evitando così errori accidentali di utilizzo.

Talvolta, le funzioni che non producono alcun risultato interessante restituiscono comunque un valore. Ad esempio, la funzione printf() restituisce il numero dei caratteri scritti. In realtà è difficile che un programma controlli questo valore. In altre parole, anche se tutte le funzioni, tranne quello di tipo void, restituiscono



un valore, non necessariamente occorre utilizzare il valore da esse restituito. Una domanda molto comune riguardante i valori restituiti da una funzione è la seguente: "Il valore restituito dalla funzione deve essere necessariamente assegnato a qualche variabile?". La risposta è no. Se non si specifica alcun assegnamento, il valore restituito viene semplicemente ignorato. Si consideri il seguente programma che utilizza la funzione mul():

```
#include <stdio.h>

int mul(int a, int b);

int main(void)
{
  int x, y, z;

  x = 10;   y = 20;
  z = mul(x, y);          /* 1 */
  printf("%d", mul(x,y)); /* 2 */
  mul(x, y);       /* 3 */

  return 0;
}

mul(int a, int b)
{
  return a*b;
}
```

Nella riga 1, il valore restituito da mul() viene assegnato a z. Nella riga 2, il valore restituito non viene assegnato ma viene utilizzato dalla funzione printf(). Infine, nella riga 3, il valore restituito viene perso poiché non viene né assegnato a un'altra variabile né utilizzato in un'espressione.

## Restituzione di puntatori

Anche se le funzioni che restituiscono puntatori vengono trattate come ogni altro tipo di funzione, è necessario discutere alcuni importanti concetti.

I puntatori a variabili non sono né interi né interi unsigned. Sono indirizzi di memoria che puntano a un determinato tipo di dati. Il motivo di questa distinzione è dovuto alle caratteristiche specifiche dell'aritmetica dei puntatori che si basa sulle dimensioni del tipo di dati puntato. Ad esempio, se viene incrementato un puntatore a interi, esso conterrà un valore maggiore di 2 unità rispetto al valore precedente (assumendo interi di 2 byte). In generale, ogni volta che si incrementa

(o decrementa) un puntatore, esso punta all'oggetto successivo (o precedente) considerando il tipo puntato. Poiché ogni tipo di dati può avere lunghezza diversa, il compilatore deve necessariamente sapere il tipo di dati cui il puntatore fa riferimento. Per questo motivo, una funzione che restituisca un puntatore deve dichiarare esplicitamente il tipo di tale puntatore. Ad esempio, per restituire un puntatore char * non si deve utilizzare il tipo int *!

Per restituire un puntatore, una funzione deve essere esplicitamente dichiarata in modo da restituire un oggetto di tipo puntatore. Ad esempio, la seguente funzione restituisce un puntatore alla prima occorrenza del carattere c nella stringa s:

```
/* Restituisce un puntatore alla prima occorrenza di c in s. */
char *match(char c, char *s)
{
  while(c!=*s && *s) s++;
  return(s);
}
```

Se non viene trovata alcuna corrispondenza, viene restituito un puntatore al carattere nullo di fine stringa. Ecco un breve programma che utilizza la funzione match():

```
#include <stdio.h>

char *match(char c, char *s);  /* prototipo */

int main(void)
{
  char s[80], *p, ch;

  gets(s);
  ch = getchar();
  p = match(ch, s);

  if(*p)  /* il carattere è stato trovato */
    printf("%s ", p);
  else
    printf("Carattere non presente.");

  return 0;
}
```

Questo programma legge una stringa e un carattere. Se il carattere è presente nella stringa, il programma stampa la stringa a partire dal carattere ricercato. In caso contrario, stampa il messaggio Carattere non presente.



## Funzioni di tipo void

Uno degli usi di void consiste nella dichiarazione esplicita di funzioni che non restituiscono valori. Questo evita che vengano utilizzate espressioni e aiuta a evidenziare errori di utilizzo accidentali. Ad esempio, la funzione print_vertical() stampa verticalmente la stringa fornita come argomento. Poiché la funzione non restituisce alcun valore, viene dichiarata come void.

```
void print_vertical(char *str)
{
  while(*str)
    printf("%c\n", *str++);
}
```

Il seguente programma mostra un esempio che stampa verticalmente un argomento della riga di comando:

```
#include <stdio.h>

void print_vertical(char *str);  /* prototipo */

int main(int argc, char *argv[])
{
  if(argc > 1) print_vertical(argv[1]);

  return 0;
}

void print_vertical(char *str)
{
  while(*str)
    printf("%c\n", *str++);
}
```

Un'ultima annotazione: le prime versioni di C non definivano la parola riservata void. Pertanto, nei vecchi programmi C le funzioni che non restituivano alcun valore venivano considerate di tipo int; pertanto non ci si deve sorprendere di vedere molti esempi di ciò nei programmi meno recenti.

## Che cosa restituisce main()?

La funzione main() restituisce al processo chiamante (che è normalmente il sistema operativo) un valore intero. La restituzione di un valore da main() equivale a chiamare exit() con lo stesso valore. Se main() non restituisce esplicitamente un

valore, il valore passato al processo chiamante sarà tecnicamente non definito. In pratica, la maggior parte dei compilatori C/C++ restituirà automaticamente uno 0, ma per motivi di trasportabilità del codice, è bene non fare affidamento su questa abitudine.

## 6.6 Ricorsione

In C/C++, una funzione può richiamare se stessa. Quando un'istruzione all'interno del corpo della funzione richiama la funzione stessa, tale funzione si dice *ricorsiva*. La ricorsione è un processo che definisce qualcosa in termini di sé stesso e in alcuni casi viene chiamata *definizione circolare*.

Un semplice esempio di funzione ricorsiva è factr() che calcola il fattoriale di un intero. Il fattoriale di un numero n è il prodotto di tutti i numeri interi da 1 a n. Ad esempio, il fattoriale di 3 è 1 x 2 x 3 ovvero 6. Di seguito sono mostrate sia factr() che la sua equivalente iterativa:

```
int factr(int n)  /* ricorsiva */
{
  int answer;

  if(n==1) return(1);
  answer = factr(n-1)*n; /* chiamata ricorsiva */
  return(answer);
}

int fact(int n)    /* non ricorsiva */
{
  int t, answer;

  answer = 1;

  for(t=1; t<=n; t++)
    answer=answer*(t);

  return(answer);
}
```

La versione non ricorsiva di fact() dovrebbe risultare chiara. Utilizza un ciclo che va da 1 a n e moltiplica progressivamente ogni numero per il prodotto accumulato.

L'operazione svolta dalla versione ricorsiva factr() è un po' più complessa. Quando factr() viene richiamata con un argomento uguale a 1, la funzione restituisce 1. In ogni altro caso, restituisce il prodotto di factr(n-1)*n. Per valutare questa espressione, viene nuovamente richiamata factr() con il parametro n-1. Questo avviene finché n non diviene uguale a 1 e la funzione inizia a restituire un valore.



Ad esempio, per calcolare il fattoriale di 2, la prima chiamata a factr() provoca una seconda chiamata ricorsiva con argomento uguale a 1. Questa chiamata restituisce il valore 1 che viene poi moltiplicato per 2 (il valore originale di n). Il risultato sarà quindi 2. Ora si provi a seguire il procedere del calcolo per il fattoriale di 3 (ad esempio si potrebbero inserire istruzioni printf() in factr() per vedere il livello raggiunto da ogni chiamata ed i vari risultati intermedi).

Quando una funzione richiama se stessa, sullo stack viene allocata una nuova serie di variabili locali e parametri e il codice della funzione viene eseguito nuovamente dall'inizio utilizzando tali variabili. Una chiamata ricorsiva non crea una nuova copia della funzione. Solo i valori su cui opera sono nuovi. Quando ogni chiamata ricorsiva esegue un return, le vecchie variabili locali e i parametri vengono rimossi dallo stack e l'esecuzione riprende nel punto in cui la funzione aveva richiamato se stessa. Questo è il motivo per cui le funzioni ricorsive sono chiamate anche funzione telescopiche.

La maggior parte delle routine ricorsive non riducono in modo significativo le dimensioni del codice né migliorano l'utilizzo della memoria. Inoltre, le versioni ricorsive della maggior parte delle routine viene eseguita in modo leggermente più lento rispetto alle loro equivalenti iterative a causa del sovraccarico dovuto alla continua ripetizione delle chiamate alle funzioni. Addirittura, un eccesso di chiamate ricorsive può provocare la fuoriuscita dallo stack. Poiché la memorizzazione dei parametri di funzione e delle variabili locali avviene sullo stack e ogni nuova chiamata crea una nuova copia di queste variabili, lo stack potrebbe andare a scrivere su altri dati o persino sulla memoria destinata al programma. Tuttavia, in genere non ci si deve preoccupare di ciò a meno che non si perda il controllo di una funzione ricorsiva.

Il vantaggio principale delle funzioni ricorsive consiste nella possibilità di creare versioni più chiare e più semplici di molti algoritmi. Ad esempio, l'algoritmo QuickSort è piuttosto difficile da implementare in modo iterativo. Inoltre, alcuni problemi, specialmente quelli di intelligenza artificiale, sono più adatti a soluzioni ricorsive. Infine, l'esecuzione ricorsiva viene considerata più lineare.

Quando si scrivono funzioni ricorsive, da qualche parte si deve prevedere un'istruzione condizionale (ad esempio un if) che faccia in modo che la funzione termini senza eseguire la chiamata ricorsiva. Se si omette l'istruzione condizionale, una volta entrati nel corpo della funzione non sarà più possibile uscirne. Durante lo sviluppo del programma si consiglia di utilizzare abbondantemente istruzioni printf() in modo da sapere sempre cosa sta avvenendo e annullare l'esecuzione nel momento in cui si rileva un errore.

## 6.7 Prototipi di funzioni

In C++ tutte le funzioni devono essere dichiarate prima dell'uso. A tale scopo si utilizza un *prototipo di funzione*. I prototipi delle funzioni non erano presenti nel linguaggio C originario. Si tratta, però, di una delle aggiunte eseguite nel momento in cui è stato prodotto lo standard C. Anche se lo Standard non richiede tecnicamente di utilizzare prototipi, si consiglia vivamente di utilizzarli. Il C++ ha invece sempre richiesto l'uso di prototipi. In questa guida pertanto, tutti gli esempi includono prototipi di funzione completi. I prototipi consentono al C e al C++ di eseguire verifiche di tipo più strette, analoghe a quelle svolte da linguaggi come il Pascal. Quando si utilizzano i prototipi, il compilatore può rilevare ogni conversione illecita fra il tipo degli argomenti utilizzati per chiamare una funzione e le definizioni di tipo dei suoi parametri. Il compilatore individuerà inoltre le differenze fra il numero degli argomenti utilizzati per richiamare una funzione e il numero di parametri della funzione.

La forma generale di un prototipo di funzione è la seguente:

*tipo nome_funz(tipo nome_par1, tipo nome_par2,...,tipo nome_parN);*

L'uso dei nomi dei parametri è opzionale. Tuttavia, essi consentono al compilatore di identificare ogni differenza di tipo utilizzando un nome e quindi è sempre consigliabile includerli.

Il seguente programma illustra l'importanza dei prototipi di funzione. Il programma produce un messaggio di errore in quanto contiene un tentativo di chiamare sqr_it() con un argomento intero al posto di un puntatore a interi (la conversione di un intero in un puntatore non è consentita).

```
/* Questo programma utilizza un prototipo di funzione
   per ottenere una più forte verifica dei tipi. */

void sqr_it(int *i); /* prototipo */

int main(void)

{
  int x;

  x = 10;
  sqr_it(x);  /* errore di tipo */

  return 0 ;
}

void sqr_it(int *i)
{
  *i = *i * *i;
}
```



Anche la definizione della funzione, se si trova prima del primo uso della funzione nel programma, può fungere da prototipo. Ad esempio, il seguente programma è perfettamente valido.

```
#include <stdio.h>

/* Questa definizione funge inoltre
   da prototipo al programma. */
void f(int a, int b)
{
  printf("%d ", a % b);
}

int main(void)
{
  f(10,3);

  return 0;
}
```

In questo esempio, poiché f() viene definita prima del suo uso in main(), non è necessario creare un prototipo distinto. Anche se è possibile che la definizione di una funzione funga anche da prototipo (specialmente nei programmi più piccoli) capita raramente di sfruttare questa possibilità nei programmi più estesi, specialmente quando il programma è costituito da più file. I programmi contenuti in questo volume includono un prototipo per ciascuna funzione poiché questo è il modo in cui viene normalmente realizzato il codice C/C++.

L'unica funzione che non richiede un prototipo è main(), dato che questa è la prima funzione che viene richiamata all'avvio del programma.

Per motivi di compatibilità con la versione originale del C, vi è una piccola ma importante differenza nel modo in cui i linguaggi C e C++ gestiscono i prototipi di una funzione che non accetta parametri. In C++, un elenco vuoto di parametri viene indicato nel prototipo come una semplice mancanza di parametri. Ad esempio,

```
int f(); /* Prototipo C++ per una funzione senza parametri */
```

In C questo prototipo significa qualcosa di leggermente differente. Per motivi storici, una lista di parametri vuota dice che non vengono fornite informazioni sui

parametri. Per quanto riguarda il compilatore, la funzione potrebbe avere svariati parametri o nessun parametro. In C, quando una funzione non ha parametri, all'interno dell'elenco dei parametri del prototipo si deve specificare void. Ad esempio, ecco l'aspetto del prototipo di f() un programma C.

```
float f(void);
```

Questa riga dice al compilatore che la funzione non ha parametri e che quando la funzione viene richiamata con parametri, ci si trova in una condizione di errore. In C++, è ancora possibile inserire void per specificare un elenco di parametri vuoto ma tale precauzione è ridondante.

NOTA *In C++, f() e f(void) sono equivalenti.*

I prototipi di funzioni aiutano a individuare i bug. Inoltre aiutano a verificare che il programma funzioni correttamente in quanto impediscono che le funzioni vengano richiamate con argomenti errati.

Un'ultima cosa: poiché le prime versioni di C non supportavano completamente la sintassi dei prototipi, in senso strettamente tecnico i prototipi sono opzionali in C. Questo è stato necessario per supportare il codice C sviluppato prima della creazione dei prototipi. Se si deve convertire in C++ un programma C, può essere necessario aggiungere i prototipi di tutte le funzioni perché il programma possa essere compilato. Occorre ricordare che i prototipi sono opzionali in C mentre sono obbligatori in C++. Questo significa che ogni funzione contenuta in un programma C++ deve avere un prototipo.

### I prototipi di funzione della libreria standard

A ogni funzione della libreria standard utilizzata dal programma deve essere associato un prototipo. A tale scopo si deve includere il file header appropriato per ciascuna funzione di libreria. Il compilatore C/C++ fornisce tutti i file header necessari. In C i file header hanno l'estensione .h. In C++ i file header possono essere file distinti o possono essere contenuti nel compilatore stesso. In entrambi i casi, un file header contiene due elementi: le definizioni utilizzate dalle funzioni della libreria e i prototipi delle funzioni della libreria. Ad esempio, stdio.h viene incluso in quasi tutti i programmi contenuti in questa parte del volume in quanto contiene il prototipo della funzione printf(). I file header per la libreria standard sono descritti nella Parte terza.



## 6.8 Dichiarazione di elenchi di parametri di lunghezza variabile

È possibile specificare funzioni con un numero variabile di parametri. L'esempio più comune è printf(). Per dire al compilatore che a una funzione può essere passato un numero indefinito di argomenti, si deve terminare la dichiarazione dei suoi parametri utilizzando tre punti. Ad esempio, questo prototipo specifica che func() riceve sicuramente almeno due parametri interi seguiti da un numero ignoto di parametri o anche da nessun parametro.

```
int func(int a, int b, ...);
```

Questa forma di dichiarazione è utilizzata anche per la definizione della funzione.

Ogni funzione che utilizza un numero variabile di parametri deve però avere almeno un parametro dichiarato. Il seguente esempio è quindi errato:

```
int func(...);  /* errato */
```

## 6.9 Dichiarazione di parametri con metodi vecchi e nuovi

Le prime versioni di C utilizzava un metodo di dichiarazione dei parametri differente da quanto stabilito dagli standard C e C++. Questa guida utilizza un approccio più *moderno*. Lo Standard C consente di utilizzare entrambe le forme ma consiglia vivamente di utilizzare la forma moderna. Lo Standard C++ prevede invece solo il metodo di dichiarazione dei parametri moderno. Tuttavia, è importante conoscere la forma classica in quanto molti programmi C meno recenti ne fanno ancora uso.

La dichiarazione classica dei parametri delle funzioni è formata da due parti: un elenco di parametri, che si trova all'interno delle parentesi che seguono il nome della funzione, e l'effettiva dichiarazione dei parametri, che sta fra la parentesi chiusa e la parentesi graffa aperta della funzione. La forma generale della definizione classica dei parametri è la seguente:

```
tipo nome_funz(par1,par2,...parN)
tipo par1;
tipo par2;
.
.
.
```

*tipo parN*;
{
*codice della funzione*
}

Ad esempio, questa dichiarazione moderna:

```
float f(int a, int b, char ch)
{
  /* ... */
}
```

corrisponde alla forma classica:

```
float f(a, b, ch)
int a, b;
char ch;
{
  /* ... */
}
```

Si noti che la forma classica consente la dichiarazione di più di un parametro dopo il nome del tipo.

**NOTA** *La forma classica della dichiarazione dei parametri è considerata oggi obsoleta in C e non è consentita dal C++.*

## 6.10 Elementi implementativi

Vi sono alcune cose molto importanti da ricordare che riguardano l'efficienza e l'usabilità delle funzioni.

### Parametri e funzioni di utilizzo generale

Una funzione di utilizzo generale è una funzione che può essere utilizzata in più situazioni, anche da programmatori diversi. In generale, si dovrà fare in modo che le funzioni di utilizzo generale non si basino su dati globali. Tutte le informazioni richieste dalla funzione dovranno essere passate attraverso i suoi parametri. Se questo non fosse possibile si devono usare variabili statiche.

Oltre a consentire l'utilizzo generale delle funzioni, l'uso dei parametri renderà il codice più leggibile e meno soggetto ai bug dovuti a effetti collaterali.



### L'efficienza

Le funzioni sono i mattoni costitutivi del C/C++ e sono fondamentali per qualsiasi programma di complessità non elementare. Tuttavia, in alcune applicazioni specializzate, può essere conveniente eliminare una funzione e sostituirla con codice *in linea*. Il codice in linea esegue le stesse azioni di una funzione ma senza il sovraccarico associato alla chiamata di funzione. Per questo motivo, quando si è molto interessati ai tempi di esecuzione di un programma, si utilizza preferibilmente codice in linea al posto delle funzioni.

Il codice in linea è più veloce rispetto a una chiamata a una funzione per due motivi. Innanzi tutto, l'esecuzione dell'istruzione CALL richiede tempo. In secondo luogo, gli argomenti passati alla funzione devono essere posizionati sullo stack e anche questo richiede tempo. Per la maggior parte delle applicazioni, si tratta di una quantità di tempo molto limitata e di poco conto. Ma quando si è interessati alla velocità, si deve ricordare che ogni chiamata a funzione utilizza tempo che si potrebbe risparmiare posizionando il codice della funzione in linea. I due listati seguenti presentano un programma che stampa il quadrato dei numeri da 1 a 10. La versione in linea viene eseguita più velocemente rispetto all'altra.

codice in linea

```
#include <stdio.h>

int main(void)
{
  int x;

  for(x=1; x<11; ++x)
    printf("%d", x*x);
}
```

chiamata a funzione

```
#include <stdio.h>
int sqr(int a);
      int main(void)
{
        int x;

        for(x=1; x<11; ++x)
          printf("%d", sqr(x));
}

int sqr(int a)
{
return a*a;
}
```

**NOTA** *In C++ il concetto di funzione in linea viene espanso e formalizzato. Infatti, le funzioni in linea sono una componente importante del linguaggio C++.*

Capitolo 7

# Strutture, unioni, enumerazioni e tipi definiti dall'utente


7.1 **Le strutture**
7.2 **Gli array di strutture**
7.3 **Il passaggio di strutture alle funzioni**
7.4 **I puntatori a strutture**
7.5 **Gli array e strutture all'interno di altre strutture**
7.6 **I campi bit**
7.7 **Le unioni**
7.8 **Le enumerazioni**
7.9 **Uso di sizeof per assicurare la trasportabilità del codice**
7.10 **La parola riservata typedef**


Il linguaggio C consente di creare tipi di dati personalizzati in cinque modi diversi.

1. La *struttura*, un raggruppamento di variabili sotto un unico nome che è anche chiamato *tipo di dati aggregato* (o talvolta *conglomerato*).
2. Il *campo bit* che è una variazione della struttura e consente l'accesso ai singoli bit.
3. L'*unione* che consente di utilizzare la stessa area di memoria per definire due o più tipi di variabili diverse.
4. L'*enumerazione*: un elenco di costanti intere cui viene associato un nome.
5. L'ultimo tipo definibile dall'utente viene creato mediante typedef che definisce un nuovo nome per un tipo preesistente.

Il linguaggio C++ supporta tutte le forme precedenti cui aggiunge le classi che verranno descritte nella Parte seconda. Qui verranno descritti gli altri metodi di creazione di dati personalizzati. In C++ le strutture e le unioni possono avere attributi tipici della programmazione a oggetti. Questo capitolo si occuperà unicamente delle funzionalità C, senza occuparsi degli elementi tipici della programmazione a oggetti, di cui ci si occuperà più avanti.

## 7.1 Le strutture

Una struttura è formata da una serie di variabili a cui si fa riferimento con un unico nome; grazie a questo, la struttura rappresenta un modo molto comodo per riunire insieme informazioni correlate. La *dichiarazione della struttura* opera da modello per creare le istanze della struttura. Le variabili che compongono la struttura sono chiamate i *membri* di tale struttura (i membri delle strutture sono talvolta chiamati *elementi* o *campi*).

In genere, tutti i membri di una struttura sono logicamente correlati fra loro. Ad esempio, le informazioni relative al nome e all'indirizzo in una mailing list vengono normalmente rappresentati con una struttura. Il seguente frammento di codice mostra la dichiarazione di una struttura che definisce i campi del nome e dell'indirizzo. La parola chiave struct comunica al compilatore che si sta dichiarando una struttura.

```
struct addr
{
  char name[30];
  char street[40];
  char city[20];
  char state[3];
  unsigned long int zip;
};
```

Si noti che la dichiarazione si chiude con un punto e virgola. Il punto e virgola è necessario in quanto la dichiarazione della struttura è essa stessa un'istruzione. Inoltre, il nome della struttura addr identifica questa determinata struttura di dati e diviene il suo specificatore di tipo.

A questo punto, *non si è ancora creata alcuna variabile*. È stata semplicemente definita la struttura dei dati. Quando si definisce una struttura, si definisce essenzialmente un tipo di variabile complesso e non la variabile stessa. Finché non si dichiara una variabile di tale tipo, non esisterà quindi alcuna variabile. In C, per dichiarare una variabile (ovvero un oggetto fisico) di tipo addr, occorre utilizzare la riga:

```
struct addr addr_info;
```

Questa riga dichiara una variabile di tipo struct addr e chiamata addr_info. In C++ si può utilizzare una forma abbreviata.

```
addr addr_info;
```



Come si può vedere, la parola riservata struct in C++ non è necessaria. In C++, quando si dichiara una struttura, si possono dichiarare variabili di tale tipo utilizzando semplicemente il nome, non preceduto dalla parola struct. Il motivo di questa differenza è che in C il solo nome della struttura non identifica un tipo. In pratica, per il C Standard, il nome di una struttura è un tag. In C, nella dichiarazione di variabili il tag della struttura deve essere preceduto dalla parola riservata struct. Al contrario, in C++ il nome della struttura è già un nome di tipo completo e dunque può essere utilizzato per definire variabili. Si deve notare che la dichiarazione C può essere utilizzata anche in un programma C++. Poiché i programmi della Parte prima di questo volume sono validi sia in C che in C++, verrà utilizzato il metodo di dichiarazione C. Si deve solo ricordare che il linguaggio C++ prevede anche una forma abbreviata.

Quando si dichiara una variabile del tipo della struttura (come ad esempio addr_info), il compilatore alloca automaticamente una quantità di memoria sufficiente per contenere tutti i membri della struttura. La Figura 7.1 mostra l'aspetto in memoria della variabile addr_info assumendo che i caratteri occupino 1 byte e che gli interi long occupino 4 byte.

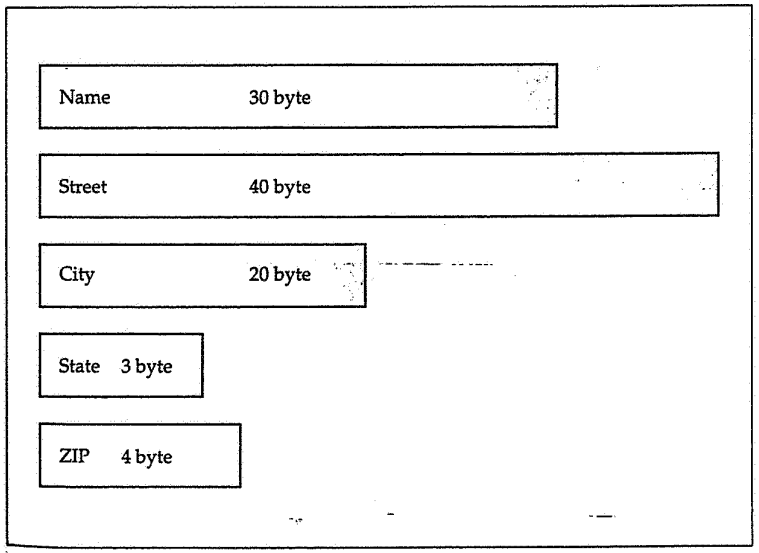



**Figura 7.1** Aspetto in memoria della struttura addr.

Quando si dichiara una struttura è anche possibile dichiarare una o più variabili. Ad esempio,

```
struct addr {
  char name[30];
  char street[40];
  char city[20];
  char state[3];
  unsigned long int zip;
} addr_info, binfo, cinfo;
```

definisce un tipo di struttura chiamato addr e dichiara le variabili addr_info, binfo e cinfo che appartengono a tale tipo.

Se si deve utilizzare una sola variabile del tipo definito dalla struttura, il tag della struttura non è necessario.

Questo significa che:

```
struct {
  char name[30];
  char street[40];
  char city[20];
  char state[3];
  unsigned long int zip;
} addr_info;
```

dichiara una variabile chiamata addr_info definita dalla struttura che la precede.

La forma generale di una dichiarazione di una struttura è:

struct *nome_struttura* {
  *tipo nome_membro* ;
  *tipo nome_membro* ;
  *tipo nome_membro* ;
  .
  .
  .
} *variabili_struttura*;

dove le parti *nome_struttura* o *variabili_struttura* (ma non entrambe) possono essere omesse.



## Accesso ai membri delle strutture

Per accedere ai singoli membri di una struttura si utilizza l'operatore "." (chiamato spesso *operatore punto*). Ad esempio, il frammento di codice seguente assegna il codice 12345 al campo zip della variabile addr_info dichiarata precedentemente:

```
addr_info.zip = 12345;
```

Per accedere ai singoli membri di una struttura si utilizza quindi il nome della variabile definita del tipo della struttura seguito da un punto e dal nome del membro. La forma generale dell'accesso a un membro di una struttura è la seguente:

*nome_struttura.nome_membro*

Pertanto, per stampare sullo schermo il codice ZIP, si deve utilizzare la riga:

```
printf("%d", addr_info.zip);
```

Questa riga stampa il codice ZIP contenuto nel membro zip della variabile di struttura addr_info.

Analogamente, l'array di caratteri addr_info.name può essere utilizzato nel seguente modo:

```
gets(addr_info.name);
```

Questa riga passa un puntatore al carattere iniziale di name.

Poiché name è un array di caratteri, è possibile accedere ai singoli caratteri di addr_info.name indicizzando name. Ad esempio, è possibile stampare il contenuto di addr_info.name un carattere per volta utilizzando il codice seguente:

```
register int t;

for(t=0; addr_info.name[t]; ++t)

  putchar(addr_info.name[t]);
```

## Assegnamenti a una struttura

Le informazioni contenute in una struttura possono essere assegnate a un'altra struttura dello stesso tipo utilizzando un'unica istruzione di assegnamento. Que-

sto significa che non è necessario assegnare separatamente i valori di ogni membro della struttura. Il seguente programma illustra questo concetto:

```
#include <stdio.h>

int main(void)
{
  struct {
    int a;
    int b;
  } x, y;

  x.a = 10;

x;  /* assegna una struttura a un'altra struttura */

  printf("%d", y.a);

  return 0;
}
```

Dopo l'assegnamento, y.a conterrà il valore 10.

## 7.2 Gli array di strutture

Probabilmente, l'uso più comune delle strutture è negli array di strutture. Per dichiarare un array di strutture, occorre innanzi tutto definire una struttura e quindi dichiarare una variabile come un array di tale tipo. Ad esempio, per dichiarare un array di 100 elementi di strutture di tipo addr si può scrivere:

```
struct addr addr_info[100];
```

Questa riga crea 100 gruppi di variabili organizzate nel modo definito nella struttura addr.

Per accedere a una determinata struttura basta indicizzare il nome della struttura. Ad esempio, per stampare il codice ZIP della struttura 3, basta scrivere:

```
printf("%d", addr_info[2].zip);
```

Come ogni array, anche gli array di strutture partono dall'indice 0.



## 7.3 Il passaggio di strutture alle funzioni

Questa sezione si occupa del passaggio delle strutture e dei relativi membri a una funzione.

### Passaggio dei membri di una struttura a una funzione

Quando si passa a una funzione un membro di una struttura, la funzione riceve solo il valore del membro. Pertanto, si passa semplicemente una variabile (a meno che, ovviamente, tale elemento sia complesso, come ad esempio un array). Ad esempio, si consideri questa struttura:

```
struct fred
{
  char x;
  int y;
  float z;
  char s[10];
} mike;
```

Ecco alcuni esempi di passaggio a una funzione dei membri di una struttura:

```
func(mike.x);      /* passa il valore carattere di x */
func2(mike.y);     /* passa il valore intero di y */
func3(mike.z);     /* passa il valore float di z */
func4(mike.s);     /* passa l'indirizzo della stringa s */
func(mike.s[2]);   /* passa il valore carattere di s[2] */
```

Se si desidera passare l'*indirizzo* di un determinato membro di una struttura, basterà inserire l'operatore & prima del nome della struttura. Ad esempio, per passare l'indirizzo dei membri della struttura mike, basterà scrivere:

```
func(&mike.x);     /* passa l'indirizzo del carattere x */
func2(&mike.y);    /* passa l'indirizzo dell'intero y */
func3(&mike.z);    /* passa l'indirizzo del float z */
func4(mike.s);     /* passa l'indirizzo della stringa s */
func(&mike.s[2]);  /* passa l'indirizzo del carattere s[2] */
```

Si ricordi che l'operatore & precede il nome della struttura e non il nome del membro. Si noti inoltre che s è già un indirizzo e quindi non si deve utilizzare l'operatore &.

### Passaggio a una funzione di un'intera struttura

Quando una struttura viene utilizzata come argomento di una funzione, l'intera struttura viene passata utilizzando il metodo standard della chiamata per valore. Naturalmente, questo significa che ogni modifica che la funzione apporta al contenuto della struttura non modifica la struttura utilizzata come argomento.

Quando si utilizza una struttura come parametro, occorre ricordare che il tipo dell'argomento deve corrispondere al tipo del parametro. Ad esempio, nel seguente programma sia l'argomento arg che il parametro parm sono dichiarati dello stesso tipo della struttura.

```
#include <stdio.h>

/* Definisce un tipo di struttura. */
struct struct_type {
  int a, b;
  char ch;
} ;
void f1(struct struct_type parm);

int main(void)
{
  struct struct_type arg;

  arg.a = 1000;

  f1(arg);

  return 0;
}

void f1(struct struct_type parm)
{
  printf("%d", parm.a);
}
```

Come si vede nel programma, se si dichiarano parametri che corrispondono a strutture, occorre rendere globale la dichiarazione del tipo della struttura in modo che possa essere utilizzata in ogni parte del programma. Ad esempio, se la struttura struct_type viene dichiarata (ad esempio) all'interno di main(), non potrà essere visibile a f1().

Come si è appena detto, quando si passano strutture alle funzioni, il tipo dell'argomento deve corrispondere al tipo del parametro. Non è sufficiente che siano fisicamente simili; devono corrispondere i nomi. Ad esempio, questa versione del



programma precedente è errata e non verrà compilata in quanto il nome del tipo dell'argomento utilizzato per richiamare f1() è diverso dal nome del tipo del suo parametro.

```
/* Questo programma è errato e non verrà compilato. */
#include <stdio.h>

/* Definisce un tipo di struttura. */
struct struct_type {
  int a, b;
  char ch;
} ;

/* Definisce una struttura simile a struct_type,
   ma con un nome diverso. */
struct struct_type2 {
  int a, b;
  char ch;
} ;
void f1(struct struct_type2 parm);

int main(void)
{
  struct struct_type arg;

  arg.a = 1000;

  f1(arg); /* differenza di tipo */

  return 0;
}

void f1(struct struct_type2 parm)
{
  printf("%d", parm.a);
}
```

## 7.4 I puntatori a strutture

Il C/C++ consente di utilizzare un puntatori a struttura come un puntatore a qualsiasi altro tipo di variabile. Tuttavia, vi sono alcuni aspetti specifici dei puntatori a strutture.

## Dichiarazione di un puntatore a struttura

Come ogni altro puntatore, un puntatore a struttura deve essere dichiarato inserendo un asterisco (*) davanti al nome di una variabile della struttura. Ad esempio, assumendo l'uso della struttura addr definita precedentemente, la riga seguente dichiara addr_pointer come un puntatore a dati di tipo addr:

```
struct addr *addr_pointer;
```

In C++ non è necessario far precedere a questa dichiarazione la parola chiave struct.

## Uso dei puntatori a struttura

Vi sono due utilizzi principali dei puntatori a struttura: il passaggio di una struttura a una funzione tramite una chiamata per indirizzo e la creazione di liste concatenate e di altre strutture di dati dinamiche utilizzando il sistema di allocazione dinamica. Questo capitolo si occupa del primo utilizzo.

Un aspetto da non sottovalutare nel passaggio di strutture complesse alle funzioni è il sovraccarico richiesto per inserire la struttura nello stack nel momento in cui viene richiamata la funzione (si ricordi che gli argomenti vengono passati alle funzioni sullo stack). Per semplici strutture costituite da pochi membri, questo sovraccarico non è così significativo. Se la struttura contiene molti membri o se alcuni dei suoi membri sono array, le prestazioni potrebbero degradare a livelli inaccettabili. La soluzione di questo problema consiste nel passaggio alla funzione del solo puntatore.

Quando alla funzione viene passato un puntatore alla struttura, nello stack viene copiato solo l'indirizzo della struttura. Questo consente di realizzare chiamate a funzioni notevolmente veloci. Un secondo vantaggio, in alcuni casi, consiste nella possibilità, da parte della funzione, di accedere all'effettiva struttura utilizzata come argomento e non a una sua copia. Passando un puntatore, la funzione può modificare il contenuto della struttura utilizzata nella chiamata.

Per conoscere l'indirizzo di una variabile di tipo struttura, basta inserire l'operatore & prima del nome della struttura. Ad esempio, dato il seguente frammento di codice:

```
struct bal {
  float balance;
  char name[80];
} person;

struct bal *p;  /* dichiara un puntatore a struttura */
```



la riga

```
p = &person;
```

inserisce nel puntatore p l'indirizzo della struttura person.

Per accedere ai membri di una struttura utilizzando un puntatore a tale struttura, si deve utilizzare l'operatore ->. Ad esempio, per accedere al campo balance si utilizza la riga:

```
p->balance
```

L'operatore -> viene normalmente chiamato *operatore freccia* ed è formato dal segno meno seguito dal segno maggiore. L'operatore freccia viene utilizzato al posto dell'operatore punto quando si accede a un membro di una struttura tramite un puntatore alla struttura stessa.

Per vedere l'uso dei puntatori a struttura, si esamini il seguente programma che visualizza sullo schermo le ore, i minuti e i secondi utilizzando un timer software.

```
/* Visualizza un timer software. */
#include <stdio.h>

#define DELAY 128000

struct my_time {
  int hours;
  int minutes;
  int seconds;
};

void display(struct my_time *t);
void update(struct my_time *t);
void delay(void);

int main(void)
{
  struct my_time systime;

  systime.hours = 0;
  systime.minutes = 0;
  systime.seconds = 0;

  for(;;) {
    update(&systime);
```

```
    display(&systime);
  }

  return 0;
}

void update(struct my_time *t)
{
  t->seconds++;
  if(t->seconds==60) {
    t->seconds = 0;
    t->minutes++;
  }

  if(t->minutes==60) {
    t->minutes = 0;
    t->hours++;
  }

  if(t->hours==24) t->hours = 0;
  delay();
}

void display(struct my_time *t)
{
  printf("%02d:", t->hours);
  printf("%02d:", t->minutes);
  printf("%02d\n", t->seconds);
}

void delay(void)
{
  long int t;

  /* modificare a piacere DELAY */
  for(t=1; t<DELAY; ++t) ;
}
```

La sincronizzazione di questo programma può essere regolata modificando la definizione di DELAY.

Come si può vedere, è stata definita una struttura globale chiamata my_time ma non è stata dichiarata alcuna variabile. All'interno di main() è stata dichiarata la struttura systime inizializzata a 00:00:00. Questo significa che systime è utilizzabile direttamente solo dalla funzione main().



Alle funzioni update() (che cambia l'ora) e display() (che visualizza l'ora) viene passato l'indirizzo di systime. In entrambe le funzioni, gli argomenti sono dichiarati come puntatori a una struttura di tipo my_time.

All'interno di update() e display() l'accesso ai membri di systime avviene tramite un puntatore. Poiché update() riceve un puntatore alla struttura systime, ne può aggiornare anche il valore. Ad esempio, per riportare l'ora a zero quando si raggiunge l'indicazione 24:00:00, update() contiene la riga di codice seguente:

```
if(t->hours==24) t->hours = 0;
```

che chiede al compilatore di prendere l'indirizzo di t (che punta a systime in main()) e imposta hours a zero.

Si ricordi l'uso dell'operatore punto per accedere agli elementi della struttura quando si opera sulla struttura stessa. Quando si utilizza un puntatore a una struttura si deve utilizzare l'operatore freccia.

## 7.5 Gli array e strutture all'interno di altre strutture

Un membro di una struttura può essere di tipo semplice o complesso. Un membro di tipo semplice appartiene a uno dei tipi base, come ad esempio int o char. All'inizio di questo Capitolo si è già visto un tipo di elemento complesso: l'array di caratteri utilizzato in addr. Altri tipi di dati complessi sono gli array mono e multi dimensionali di altri tipi di dati e strutture.

Quando un membro di una struttura è un array, esso viene trattato nel modo in cui ci si può attendere visti gli esempi precedenti. Ad esempio, si consideri la struttura:

```
struct x {
int a[10][10]; /* array di 10 x 10 int */
  float b;
} y;
```

Per accedere all'intero 3,7 nel membro a della struttura y si deve scrivere:

```
y.a[3][7]
```

Quando una struttura è un membro di un'altra struttura, si parla di *struttura nidificata*. Ad esempio, nell'esempio seguente, la struttura address è nidificata all'interno della struttura emp:

```
struct emp {
  struct addr address; /* struttura nidificata */
  float wage;
} worker;
```

Qui, la struttura emp è stata definita in modo da contenere due membri. Il primo è una struttura di tipo addr che contiene l'indirizzo e l'altro è il valore float wage. Il seguente frammento di codice assegna il valore 93456 all'elemento zip di address.

```
worker.address.zip = 93456;
```

Come si può vedere, l'accesso ai membri di ogni struttura avviene da quello più esterno a quello più interno. Lo Standard C specifica che le strutture possono essere nidificate fino a 15 livelli ma la maggior parte dei compilatore consente di utilizzare un maggior numero di livelli. Lo Standard C++ suggerisce di consentire almeno 256 livelli di nidificazione.

## 7.6 I campi bit

A differenza di altri linguaggi, il C/C++ prevede una funzionalità chiamata *campo bit* che consente di accedere ai singoli bit dei dati. I campi bit sono utili in molte occasioni:

- ad esempio quando lo spazio di memoria è limitato e si vogliano inserire più variabili booleane (con valori vero o falso) in un singolo byte;
- alcuni dispositivi trasmettono informazioni sul loro stato codificate in bit;
- alcune routine di crittografia devono accedere ai bit di un byte.

Anche se queste operazioni possono essere eseguite utilizzando operatori bit-a-bit, un campo bit può aggiungere un livello di strutturazione ed efficienza maggiore al codice.

Per accedere ai singoli bit, il C utilizza un metodo che si basa sulla struttura. Infatti, un campo bit non è che un tipo speciale di membro di struttura che definisce la lunghezza, in bit, del campo. La forma generale della definizione di un campo bit è la seguente:

```
struct nome {
  tipo nome1 : lunghezza;
  tipo nome2 : lunghezza;
  tipo nomeN : lunghezza;
} elenco_variabili;
```



Qui, *tipo* specifica il tipo del campo bit e *lunghezza* è il numero di bit che compongono il campo. Un campo bit può essere di un tipo intero o enumerativo. I campi bit di lunghezza 1 devono essere dichiarati come unsigned poiché un singolo bit non può avere un segno.

I campi bit sono molto utilizzati per analizzare l'input proveniente da un dispositivo hardware. Ad esempio, un adattatore di comunicazione seriale può restituire un byte di stato organizzato nel seguente modo:

| BIT | SIGNIFICATO |
|---|---|
| 0 | Cambia in clear-to-send |
| 1 | Cambia in data-set-ready |
| 2 | Trailing-edge |
| 3 | Cambia in ricezione |
| 4 | Clear-to-send |
| 5 | Data-set-ready |
| 6 | Squillo del telefono |
| 7 | Segnale ricevuto |

È possibile rappresentare queste informazioni in un byte di stato utilizzando i seguenti campi bit:

```
struct status_type {
  unsigned delta_cts: 1;
  unsigned delta_dsr: 1;
  unsigned tr_edge:   1;
  unsigned delta_rec: 1;
  unsigned cts:       1;
  unsigned dsr:       1;
  unsigned ring:      1;
  unsigned rec_line:  1;
} status;
```

Per consentire al programma di determinare quando è possibile inviare o ricevere dati, si deve utilizzare una routine simile alla seguente:

```
status = get_port_status();
if(status.cts) printf("clear-to-send");
if(status.dsr) printf("data-ready");
```

Per assegnare un valore a un campo bit, si può usare la forma già vista per altri tipi di elementi. Ad esempio, il seguente frammento di codice cancella il contenuto del campo ring:

```
status.ring = 0;
```

Come si può vedere da questo esempio, per accedere ai campi bit si utilizza l'operatore punto. Tuttavia, se l'accesso alla struttura avviene tramite un puntatore, si deve utilizzare l'operatore ->.

Non è necessario assegnare un nome a ogni campo bit. Il nome semplifica però l'accesso al bit desiderato, saltando quelli non utilizzati. Ad esempio, se interessa solo il contenuto dei bit cts e dsr, si può dichiarare una struttura status_type nel modo seguente:

```
struct status_type {
  unsigned :    4;
  unsigned cts: 1;
  unsigned dsr: 1;
} status;
```

Inoltre, si noti che i bit che seguono dsr non devono essere specificati se non vengono utilizzati.

Inoltre è consentito usare insieme nella stessa struttura membri standard e campi bit. Ad esempio,

```
struct emp {
  struct addr address;
  float pay;
  unsigned lay_off:    1; /* a riposo o attivo */
  unsigned hourly:     1; /* paga oraria o mensile */
  unsigned deductions: 3; /* deduzioni */
};
```

definisce il record di un dipendente che utilizza solo un byte per contenere tre tipi di informazioni: lo stato del dipendente, il fatto che sia pagato a ore e il numero di deduzioni. Senza l'uso di campi bit, queste informazioni avrebbero occupato tre byte.



I campi bit hanno però alcune restrizioni. Non è possibile conoscere l'indirizzo di un campo bit. Non è possibile creare array di campi bit. Non possono essere dichiarati static. Non è possibile conoscere, su sistemi operativi diversi, se i campi vanno da sinistra a destra o viceversa. In altre parole, il codice che utilizza campi bit è soggetto ad alcune dipendenze relative alla macchina su cui viene impiegato.

Possono esservi anche altre restrizioni imposte dall'implementazione. A tale proposito si rimanda alla documentazione del compilatore.

## 7.7 Le unioni

In C, un'*unione* è un indirizzo di memoria condiviso, in momenti diversi, da due o più variabili generalmente di tipo diverso. La dichiarazione di un'*unione* è simile a quella di una struttura. La sua forma generale è:

union *nome_unione* {
    *tipo nome_membro*;
    *tipo nome_ membro*;
    *tipo nome_ membro*;
    .
    .
    .
} *variabili_unione*;

Ad esempio,

```
union u_type {
  int i;
  char ch;
};
```

Questa dichiarazione non crea alcuna variabile. In C si può dichiarare una variabile specificandone il nome alla fine della dichiarazione o utilizzando una dichiarazione distinta. Per dichiarare una variabile union di nome cnvt e di tipo u_type utilizzando la definizione precedente, si deve scrivere:

```
union u_type cnvt;
```

Quando si dichiarano variabili union in C++, basta utilizzare il nome del tipo; dunque non è necessario specificare la parola riservata union. Ad esempio, ecco come cnvt viene dichiarata in C++:

```
u_type cnvt;
```

Anche in C++, è comunque possibile specificare la parola riservata union che, tuttavia, è ridondante. In C++ infatti il nome dell'unione definisce da solo il nome completo del tipo. In C il nome dell'unione è un tag e deve essere necessariamente preceduto dalla parola riservata union (è una situazione analoga a quella delle strutture, descritte in precedenza). Poiché i programmi di questo capitolo sono validi sia in C che in C++, verrà utilizzata la dichiarazione in stile C.

In cnvt l'intero i e il carattere ch condividono lo stesso indirizzo di memoria. Naturalmente i occupa 2 byte (nel caso di interi di 2 byte) e ch ne occupa uno solo. La Figura 7.2 mostra il modo in cui i e ch condividono lo stesso indirizzo. In un determinato punto del programma è possibile accedere ai dati memorizzati in cnvt come a un intero o a un carattere.

Quando si dichiara una variabile union, il compilatore alloca automaticamente uno spazio di memoria sufficiente a contenere il membro più esteso dell'unione. Ad esempio, assumendo l'uso di interi di 2 byte, cnvt occuperà 2 byte in modo da poter contenere i, anche se ch richiede solo un byte.

Per accedere a un membro di un'unione, si utilizza la stessa sintassi già vista per le strutture: gli operatori punto e freccia. Se si opera direttamente su un'unione si utilizza l'operatore punto. Se l'accesso all'unione avviene tramite un puntatore, si utilizza l'operatore freccia. Ad esempio, per assegnare l'intero 10 all'elemento i di cnvt, si deve scrivere

```
cnvt.i = 10;
```

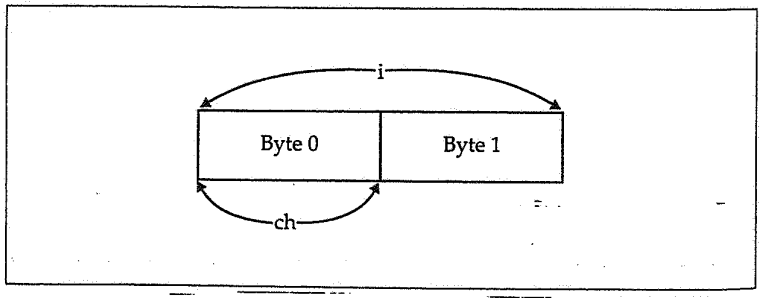



**Figura 7.2** Il modo in cui i e ch utilizzano l'unione cnvt (assumendo l'uso di interi di 2 byte).



Nell'esempio seguente, alla funzione viene passato un puntatore a cnvt:

```
void funcl(union u_type *un)
{
  un->i = 10; /* assegna 10 a cnvt utilizzando
                 una funzione */
}
```

L'uso di un'unione può essere di grande aiuto nella produzione di codice indipendente dalla macchina su cui viene utilizzato. Poiché il compilatore registra le effettive dimensioni dei membri dell'unione, non viene prodotta alcuna dipendenza dalla macchina. Questo significa che non ci si deve preoccupare delle dimensioni di un int, di un long, di un float o di quant'altro.

Le unioni sono molto utilizzate quando si richiedono conversioni di tipo specializzate in quanto è possibile far riferimento ai dati contenuti in un'unione in modi completamente diversi. Ad esempio, si può utilizzare un'unione per manipolare i byte che formano un double in modo da modificarne la precisione o da eseguire un qualche tipo insolito di arrotondamento.

Per avere un'idea dell'utilità di un'unione quando occorre eseguire conversioni di tipo non standard, si consideri il problema della scrittura di un intero short su un file. La libreria standard del C/C++ non contiene alcuna funzione scritta in modo specifico. Anche se è possibile scrivere dati di qualsiasi tipo su un file utilizzando fwrite(), il suo impiego sembra un po' eccessivo per un'operazione così semplice. Utilizzando un'unione si può facilmente creare una funzione chiamata putw() che scrive su file la rappresentazione binaria di uno short int, un byte per volta (in questo esempio si suppone che un intero short occupi 2 byte). Per vedere come ciò sia possibile si inizi a creare un'unione formata da uno short int e un array di caratteri di 2 byte:

```
union pw {
  short int i;
  char ch[2];
};
```

Ora si potrebbe utilizzare pw per creare versione di putw() mostrata nel seguente programma:

```
#include <stdio.h>
union pw {
  short int i;
  char ch[2];
};
```

```
putw(short int num, FILE *fp);

int main(void)
{
  FILE *fp;

  fp = fopen("test.tmp","w+");

  putw(1000, fp);              /* scrive il valore 1000 come un intero */
  fclose(fp);

  return 0;
}

int putw(short int num, FILE *fp)
{
  union pw word;

  word.i = num;

  putc(word.ch[0], fp);        /* scrive la prima metà */
  return putc(word.ch[1], fp); /* scrive la seconda metà */
}
```

Anche se putw() viene richiamata con uno short int, essa può utilizzare la funzione standard putc() per scrivere su disco l'intero un byte per volta.

NOTA *Il C++ consente l'uso di un tipo particolare di unione chiamata unione anonima che verrà discussa nella seconda parte di questa guida.*

## 7.8 Le enumerazioni

Un'*enumerazione* è formata da un gruppo di costanti intere dotate di un nome che specificano tutti i valori consentiti per una variabile di tale tipo. Le enumerazioni sono molto comuni nella vita di tutti i giorni. Ad esempio, un'enumerazione delle monete utilizzate negli Stati Uniti potrebbe essere:

penny, nickel, dime, quarter, half-dollar, dollar

Le enumerazioni sono definite come le strutture; la parola chiave enum segnala l'inizio di un tipo enumerativo. La forma generale delle enumerazioni è la seguente:



enum *nome_tipo_enumerativo* { *elenco enumerazioni* } *elenco_variabili*;

Qui, sia il nome del tipo enumerativo che l'elenco delle variabili sono elementi opzionali (ma deve essere presente almeno uno di questi due elementi). Il seguente frammento di codice definisce un'enumerazione chiamata coin:

```
enum coin { penny, nickel, dime, quarter,
            half_dollar, dollar};
```

Il nome del tipo enumerativo può essere utilizzato per dichiarare variabili di tale tipo. In C, la seguente riga dichiara money come una variabile di tipo coin.

```
enum coin money;
```

In C++, la variabile money può essere dichiarata con la seguente forma abbreviata:

```
coin money;
```

In C++, il nome di un'enumerazione specifica l'intero tipo. In C, il nome dell'enumerazione è un tag che richiede l'impiego della parola riservata enum. Questa situazione è simile a quella descritta nel caso delle strutture e delle unioni di cui si è parlato in precedenza.

Date tali dichiarazioni, le seguenti istruzioni sono perfettamente corrette:

```
money = dime;
if(money==quarter) printf("La moneta è un quarter.\n");
```

Il fattore chiave per comprendere il funzionamento di un'enumerazione è che a ognuno dei simboli corrisponde un valore intero. Questi valori possono quindi essere utilizzati in qualunque punto in cui si potrebbe utilizzare un intero. A ogni simbolo viene assegnato un valore maggiore di una unità rispetto al simbolo precedente. Il valore del primo simbolo dell'enumerazione è zero. Pertanto,

```
printf("%d %d", penny, dime);
```

visualizza i valori 0 2.

Il valori di uno o più simboli possono essere specificati esplicitamente tramite un inizializzatore. A questo scopo si deve far seguire al simbolo il segno di uguaglianza e un valore intero. Ai simboli che appaiono dopo gli inizializzatori vengono assegnati valori maggiori rispetto all'inizializzazione precedente. Ad esempio, il seguente codice assegna a quarter il valore 100:

```
enum coin { penny, nickel, dime, quarter=100,
            half_dollar, dollar};
```

I valori di questi simboli saranno quindi:

| | |
|---|---|
| penny | 0 |
| nickel | 1 |
| dime | 2 |
| quarter | 100 |
| half_dollar | 101 |
| dollar | 102 |

Un'assunzione piuttosto comune ma erronea riguardante le enumerazioni è il fatto che i simboli possano essere utilizzati direttamente in operazioni di input e output. Ad esempio il seguente frammento di codice non funzionerà nel modo desiderato:

```
/* non funzionante */
money = dollar;
printf("%s", money);
```

Si ricordi che dollar è il nome di un intero, non è una stringa. Per lo stesso motivo non è possibile utilizzare questo codice per ottenere i risultati desiderati:

```
/* codice errato */
strcpy(money, "dime");
```

Quindi, una stringa che contiene il nome di un simbolo non viene convertita automaticamente in tale simbolo.

La creazione di codice per l'input e l'output dei simboli delle enumerazioni è piuttosto noiosa (a meno che ci si voglia basare solo sui valori interi). Ad esempio, per visualizzare, in lettere, i valori contenuti in money, si deve utilizzare uno switch di questo tipo:

```
switch(money) {
  case penny: printf("penny");
    break;
  case nickel: printf("nickel");
    break;
  case dime: printf("dime");
    break;
  case quarter: printf("quarter");
    break;
  case half_dollar: printf("half_dollar");
    break;
  case dollar: printf("dollar");
}
```



Talvolta, è possibile dichiarare un array di stringhe e utilizzare i valori dell'enumerazione come indici per tradurre tali valori nella stringa corrispondente. Ad esempio, anche questo codice produce la stringa corretta:

```
char name[][12]={
  "penny",
  "nickel",
  "dime",
  "quarter",
  "half_dollar",
  "dollar"
};
printf("%s", name[money]);
```

Naturalmente, questo funziona solo se non viene inizializzato alcun simbolo in quanto l'array di stringhe deve essere indicizzato a partire da zero.

Poiché i valori delle enumerazioni devono essere convertiti manualmente nei corrispondenti valori stringa per le operazioni di I/O, la loro utilità risulta più evidente all'interno di routine che non eseguono tali conversioni. Un'enumerazione viene spesso utilizzata per definire tabelle di simboli, ad esempio per un compilatore. Le enumerazioni sono spesso utilizzate anche per dimostrare la validità di un programma fornendo una verifica di ridondanza al momento della compilazione per confermare che a una variabile vengano assegnati solo valori validi.

## 7.9 Uso di sizeof per assicurare la trasportabilità del codice

Si è detto che le strutture e le unioni possono essere utilizzate per creare variabili di dimensioni diverse e che le effettive dimensioni di tali variabili possono variare da macchina a macchina. L'operatore sizeof calcola le dimensioni di una variabile o di un tipo e aiuta quindi a eliminare dal programma il codice dipendente dalla macchina. Questo operatore è particolarmente utile quando si utilizzano strutture o unioni.

Per la discussione seguente, si assume un implementazione, molto comune nei compilatori C/C++, in cui i dati abbiano le seguenti dimensioni:

| Tipo | Dimensioni in byte |
|---|---|
| char | 1 |
| int | 4 |
| double | 8 |

Pertanto, il seguente frammento di codice visualizzerà sullo schermo i numeri 1, 4 e 8:

```
char ch;
int i;
double f;

printf("%d", sizeof(ch));

printf("%d", sizeof(i));

printf("%d", sizeof(f));
```

Le dimensioni di una struttura sono uguali o maggiori della somma delle dimensioni dei suoi membri. Ad esempio,

```
struct s {
  char ch;
  int i;
  double f;
} s_var;
```

Qui, sizeof(s_var) è uguale almeno a 13 (8 + 4 +1). Tuttavia, le dimensioni di s_var potrebbero essere maggiori in quanto il compilatore potrebbe sistemare diversamente una struttura per consentirne l'allineamento all'interno di una word (2 byte) o di un paragrafo (16 byte). Poiché le dimensioni di una struttura potrebbero essere maggiori rispetto alla somma delle dimensioni dei suoi membri, per conoscere la dimensione della struttura si deve sempre utilizzare sizeof.

Poiché sizeof è un operatore che viene eseguito al momento della compilazione, tutte le informazioni necessarie per calcolare le dimensioni di una variabile saranno note al momento della compilazione. Questo è particolarmente importante nel caso delle union in quanto le dimensioni di una union sono sempre uguali alle dimensioni del suo membro più grande. Ad esempio si consideri la seguente unione

```
union u {
  char ch;
```



```
  int i;
  double f;
} u_var;
```

Qui, sizeof(u_var) vale 8. Al momento dell'esecuzione, non importa cosa in effetti u_var contenga. Tutto ciò che interessa sapere sono le dimensioni del suo membro più grande, in quanto ogni unione ha le dimensioni dell'elemento più grande dichiarato al suo interno.

## 7.10 La parola riservata typedef

Il C/C++ consente di definire esplicitamente nuovi tipi di dati utilizzando la parola chiave typedef. In questo modo non si *crea* un nuovo tipo di dati ma si definisce un nuovo nome per un tipo preesistente. Questo processo può aiutare a rendere più trasportabili i programmi dipendenti dalla macchina. Se si definisce un proprio nome per ogni tipo di dati dipendente dalla macchina utilizzato dal programma, quando il programma verrà compilato in un nuovo ambiente basterà cambiare solo le istruzioni typedef. typedef può anche essere utile per l'auto documentazione del codice, consentendo l'uso di nomi descrittivi per i tipi di dati standard. La forma generale dell'istruzione typedef è la seguente:

typedef *tipo nuovonome*;

dove *tipo* è un qualunque tipo di dati valido e *nuovonome* è il nuovo nome che si intende assegnare a questo tipo. Questo nuovo nome si aggiunge al tipo preesistente ma non lo sostituisce.

Ad esempio, è possibile creare un nuovo nome per il tipo float utilizzando:

```
typedef float balance;
```

Questa istruzione dice al compilatore di riconoscere balance come un altro nome di float. In seguito sarà possibile creare una variabile float utilizzando il tipo balance:

```
balance over_due;
```

Qui, over_due è una variabile in virgola mobile di tipo balance che non è che un modo diverso di chiamare il tipo float.

Ora che si è definito il tipo balance, esso potrà anche essere utilizzato in un altro typedef. Ad esempio,

```
typedef balance overdraft;
```

chiede al compilatore di riconoscere overdraft come un altro nome di balance che non è altro che un float.

L'uso di typedef può rendere il codice più facile da leggere e da trasportare su una nuova macchina. Si ricordi però sempre che con typedef non si sta creando un nuovo tipo di dati.

Capitolo 8

# Operazioni di I/O da console


8.1 **Un'importante nota applicativa**

8.2 **La lettura e la scrittura di caratteri**

8.3 **La lettura e la scrittura di stringhe**

8.4 **Le operazioni di I/O formattato da console**

8.5 **La funzione printf()**

8.6 **La funzione scanf()**


Il linguaggio C++ supporta due sistemi di I/O. Il primo deriva dal C e il secondo è il sistema di I/O a oggetti definito dal C++. Questo capitolo, insieme al prossimo, descrive il sistema di I/O del C (il sistema del C++ verrà descritto nella Parte seconda). Anche se probabilmente si preferirà utilizzare il sistema C++ per tutti i nuovi progetti, può capitare frequentemente di trovare codice che impiega il sistema C. In ogni caso la conoscenza del sistema di I/O C è fondamentale per comprendere appieno le funzionalità del sistema C++.

In C, tutte le operazioni di input e output vengono eseguite utilizzando le funzioni della libreria. Le operazioni di I/O si possono svolgere da console e da file. Tecnicamente, non esiste una grande distinzione fra I/O su console e I/O su file benché, si tratti concettualmente di situazioni molto diverse. Questo capitolo esamina in dettaglio le funzioni di I/O da console. Il prossimo capitolo si occupa del sistema di I/O da file e descrive le relazioni fra questi due sistemi.

Con un'unica eccezione, questo capitolo si occupa solo delle funzioni di I/O da console definite dallo Standard C++. Lo Standard C++ non definisce nessuna funzione per il controllo dello schermo (ad esempio per il posizionamento del cursore) o per la visualizzazione di oggetti grafici, poiché queste operazioni possono essere molto diverse da una macchina a un'altra. Inoltre non definisce alcuna funzione per la scrittura in una finestra di Windows. Quindi, le funzioni standard di I/O da console eseguono solo operazioni di output di puro testo in formato "teletype". Tuttavia, la maggior parte dei compilatori include nelle proprie librerie numerose funzioni per il controllo dello schermo e funzioni grafiche che si applicano solo all'ambiente per il quale il compilatore è stato progettato. Naturalmente si può scrivere un programma Windows in C++ ma occorre tenere presente che il linguaggio non fornisce funzioni in grado di eseguire queste operazioni.

Le funzioni di I/O Standard C usano tutte il file header stdio.h. I programmi C++ possono anche utilizzare il nuovo file header <cstdio>.

Il capitolo si occupa delle funzioni di I/O da console che accettano input dalla tastiera e producono output sullo schermo. Tuttavia, queste funzioni hanno come origine e/o destinazione delle proprie operazioni i canali di input e output standard del sistema. Inoltre, i canali di input e output standard possono essere diretti verso altri dispositivi. Questi concetti saranno chiariti nel Capitolo 9.

## 8.1 Un'importante nota applicativa

La prima parte di questa guida utilizza il sistema di I/O C perché questo è l'unico metodo di I/O definito per il sottoinsieme C del C++. Come si è detto, il C++ definisce anche un proprio sistema di I/O orientato agli oggetti. Dunque, per la maggior parte dei programmi a oggetti, sarà preferibile impiegare il sistema di I/O specifico del C++ e non il sistema di I/O C descritto in questo capitolo. Tuttavia, una conoscenza approfondita del sistema di I/O C è importante per i seguenti motivi:

- Potrebbe capitare di dover scrivere codice che deve limitarsi al sottoinsieme C. In questo caso si dovrà necessariamente impiegare le funzioni di I/O C.
- Nel prossimo futuro, vi sarà una coesistenza fra C e C++. Inoltre vi saranno molti programmi ibridi contenenti codice C e C++. Inoltre sarà molto comune l'aggiornamento dei programmi C e la loro trasformazione in programmi C++. Pertanto sarà necessario conoscere sia il sistema di I/O del C che quello del C++. Ad esempio, per trasformare le funzioni di I/O del C in funzioni di I/O a oggetti del C++ sarà necessario conoscere il funzionamento delle operazioni di I/O sia in C che in C++.
- Una conoscenza dei principi di base tipici del sistema di I/O del C è fondamentale per comprendere il sistema di I/O a oggetti del C++ (entrambi condividono la stessa filosofia).
- In alcune situazioni (ad esempio nei programmi più brevi), è più facile utilizzare il sistema di I/O del C rispetto a quello orientato agli oggetti del C++.

Infine, vi è una tacita regola che stabilisce che ogni programmatore C++ debba essere anche un programmatore C. Non conoscendo il sistema di I/O del C il lettore corre il rischio di limitare i propri orizzonti professionali.

## 8.2 La lettura e la scrittura di caratteri

Le più semplici funzioni di I/O da console sono getchar() che legge un carattere dalla tastiera e putchar() che stampa un carattere sullo schermo. La funzione



getchar() attende la pressione di un tasto e restituisce il valore corrispondente. Il tasto premuto viene inoltre automaticamente visualizzato sullo schermo. La funzione putchar() scrive un carattere sullo schermo alla posizione corrente del cursore. I prototipi delle funzioni getchar() e putchar() sono i seguenti:

```
int getchar(void);
int putchar(int c);
```

Come si può vedere dal prototipo, la funzione getchar() restituisce un intero. Tuttavia, si può assegnare questo valore a una variabile char, operazione in genere molto comune, in quanto il carattere immesso si troverà nel byte di ordine inferiore (il byte di ordine superiore sarà normalmente uguale a zero). In caso di errore, getchar() restituisce EOF.

Nel caso di putchar(), anche se nel prototipo si indica che accetta un parametro intero, sarà possibile richiamare la funzione utilizzando un argomento di tipo carattere. In effetti, sullo schermo viene visualizzato solo il byte di ordine inferiore del parametro. La funzione putchar() restituisce il carattere che essa stessa scrive oppure EOF in caso di errore (la macro EOF è definita nel file stdio.h e generalmente è uguale a -1).

Il seguente programma illustra l'uso delle funzioni getchar() e putchar(). Questo breve programma legge un carattere dalla tastiera e lo trasforma in maiuscolo se è minuscolo e viceversa. Per fermare il programma basta immettere un punto.

```
#include <stdio.h>
#include <ctype.h>

int main(void)
{
  char ch;

  printf("Immettere del testo (per uscire, premere il punto).\n");
  do {
    ch = getchar();

    if(islower(ch)) ch = toupper(ch);
    else ch = tolower(ch);

    putchar(ch);
  } while (ch != '.');

  return 0;
}
```

### I problemi di getchar()

Nell'uso di getchar() possono sorgere alcuni problemi. Normalmente getchar() è implementata in modo da bufferizzare l'input fino alla pressione del tasto INVIO. Questa tecnica di input è chiamata *bufferizzazione della riga*: per inviare qualunque cosa al programma è necessario premere il tasto INVIO. Inoltre, poiché getchar() legge un solo carattere per volta, la bufferizzazione della riga poteva lasciare uno o più caratteri in attesa nella coda di input e questo può rappresentare un problema in ambienti interattivi. Anche se lo Standard C/C++ specifica che getchar() possa essere implementata come funzione interattiva, ciò avviene raramente. Questo è il motivo per cui il programma precedente potrebbe non comportarsi nel modo atteso.

### Le alternative a getchar()

La funzione getchar() potrebbe essere implementata in modo non utile in ambienti interattivi. In questo caso, si vorrà probabilmente eseguire la lettura di caratteri dalla tastiera utilizzando una funzione diversa. Lo Standard C++ non definisce nessuna funzione in grado di eseguire input interattivo, ma praticamente tutti i compilatori C++ ne offrono una. Anche se queste funzioni non sono definite dallo Standard C++, si tratta di funzioni ampiamente utilizzate in quanto getchar() non risponde alle esigenze di molti programmatori.

Due delle più comuni funzioni alternative, getch() e getche() hanno i seguenti prototipi:

```
int getch(void);
int getche(void);
```

Nella maggior parte dei compilatori, i prototipi di queste funzioni si trovano nel file conio.h. Per alcuni compilatori, queste funzioni sono precedute dal carattere di sottolineatura. Ad esempio, in Microsoft Visual C++ queste funzioni si chiamano _getch() e _getche(). La funzione getch() attende la pressione di un tasto e ne restituisce immediatamente il valore ma non visualizza il carattere sullo schermo. La funzione getche() è uguale a getch() ma visualizza sullo schermo il carattere corrispondente al tasto premuto. In questa guida si utilizzano molto spesso sia getche() che getch() al posto di getchar() ogni volta che un programma interattivo richiede la lettura di un carattere dalla tastiera. Se il compilatore non dovesse prevedere l'uso di queste funzioni alternative o se getchar() fosse implementata come funzione interattiva, si potrà sostituire, se necessario, getchar().

Ad esempio, si può riscrivere il programma precedente in modo che usi getch() al posto di getchar():



```
#include <stdio.h>
#include <conio.h>
#include <ctype.h>

int main(void)
{
  char ch;

  printf("Immettere del testo (per uscire, premere il punto).\n");
  do {
    ch = getch();

    if(islower(ch)) ch = toupper(ch);
    else ch = tolower(ch);

    putchar(ch);
  } while (ch != '.');

  return 0;
}
```

Quanto si esegue questa versione del programma, il tasto viene immediatamente trasmesso al programma e visualizzato in maiuscolo se era minuscolo e viceversa. L'input non risulta più bufferizzato. Anche se nel codice di questo volume non utilizzerà più getch() o getche(), si tratta di funzioni molto utili nei programmi.

**NOTA** *Al momento attuale, le funzioni _getche() e _getch() del compilatore Microsoft Visual C++ non sono compatibili con le funzioni di input standard C/C++ come scanf() e gets(). Al loro posto si devono usare speciali versioni di queste funzioni standard chiamate cscanf() e cgets(). Per informazioni dettagliate, consultare la documentazione di Visual C++.*

## 8.3 La lettura e la scrittura di stringhe

Il passo successivo nelle operazioni di I/O da console, in termini di complessità e di potenza, è formato dalle funzioni gets() e puts(). Queste funzioni consentono di leggere e scrivere stringhe di caratteri.

La funzione gets() legge una stringa di caratteri immessa alla tastiera e la inserisce all'indirizzo puntato dal proprio argomento. L'immissione dei caratteri ha termine quando si preme il tasto INVIO. A questo punto, nella stringa verrà inserito il carattere nullo di fine stringa e gets() terminerà con un return. Quindi

non si può usare gets() per restituire il codice del tasto di INVIO (per questo scopo si può usare getchar()). Gli errori di digitazione possono essere corretti prima della pressione del tasto INVIO utilizzando il tasto BACKASPACE. Il prototipo della funzione gets() è il seguente:

```
char *gets(char) *str);
```

dove *str* è un array di caratteri che riceve l'input dall'utente. La funzione gets() restituisce la stringa letta in str. Il seguente programma legge una stringa nell'array str e ne visualizza la lunghezza:

```
#include <stdio.h>
#include <string.h>

int main(void)
{
  char str[80];

  gets(str);
  printf("Lunghezza stringa = %d", strlen(str));

  return 0;
}
```

Occorre fare attenzione a utilizzare gets() poiché essa non esegue alcuna verifica di fuoriuscita dall'array che riceve l'input. Pertanto è possibile che l'utente immetta un numero di caratteri eccessivo rispetto alle dimensioni dell'array. Anche se gets() è adatta per semplici programmi e semplici utility di utilizzo privato dello sviluppatore, è opportuno evitarne l'uso nei programmi commerciali. Un'alternativa è rappresentata dalla funzione fgets() che verrà descritta nel prossimo capitolo. Tale funzione non consente di fuoriuscire dall'array.

La funzione puts() scrive sullo schermo il contenuto del proprio argomento stringa seguito dal codice di fine riga. Il suo prototipo è:

```
int puts(const char *str);
```

La funzione puts() riconosce gli stessi codici backslash di printf(), come ad esempio "\t" per la tabulazione orizzontale. Una chiamata a puts() richiede un sovraccarico inferiore (in termini di tempo e memoria) rispetto alla funzione printf() in quanto la prima può solo visualizzare stringhe di carattere e non può visualizzare né numeri né eseguire conversioni di formato. Pertanto, puts() richiede meno spazio e viene eseguita più velocemente rispetto a printf(). Per questo motivo, la funzione puts() viene spesso utilizzata quando è importante produrre codice perfettamente ottimizzato. In caso di errore, la funzione puts() restituisce EOF. In tutti gli altri casi, restituisce un valore non negativo. Ma in genere, quando si scrive sulla console, si può ragionevolmente supporre che non si verifichi alcun errore e quindi difficilmente si controlla il valore restituito da puts(). La seguente istruzione visualizza sullo schermo la parola ciao:



```
puts("ciao");
```

La Tabella 8.1 riepiloga le principali funzioni di I/O da console.

Il seguente programma, un semplice dizionario computerizzato, mostra l'uso di molte delle funzioni di I/O da console. Il programma chiede all'utente di immettere una parola e quindi la confronta con un piccolo database interno. Se viene trovata una corrispondenza, il programma visualizza il significato della parola. Occorre fare particolare attenzione all'uso dei puntatori in questo programma. Se si ha qualche difficoltà a comprenderne il funzionamento, si ricordi che l'array dic è un array di puntatori a stringhe. Si noti inoltre che l'elenco deve essere concluso da due stringhe nulle.

```
/* Un semplice dizionario. */
#include <stdio.h>
#include <string.h>
#include <ctype.h>

/* elenco delle parole e significato */
char  *dic[][40] = {
  "atlante", "Un libro di mappe",
  "auto", "Un veicolo motorizzato",
  "telefono", "Un dispositivo di comunicazione",
  "aereo", "Una macchina volante",
  "", ""  /* stringa nulla di terminazione dell'elenco */
};

int main(void)
{
  char word[80], ch;
  char **p;

  do {
    puts("\nImmettere una parola: ");
    scanf("%s", word);

    p = (char **)dic;
```

```
    /* trova la parola corrispondente e ne visualizza il significato */
    do {
      if(!strcmp(*p, word)) {
        puts("Significato:");
        puts(*(p+1));
        break;
      }
      if(!strcmp(*p, word)) break;
      p = p + 2;  /* scorre l'elenco */
    } while(*p);
    if(!*p) puts("La parola non si trova nel dizionario");
    printf("Altre parole? (s/n): ");
    scanf(" %c%*c", &ch);
  } while(toupper(ch) != 'N');

  return 0;
}
```

## 8.4 Le operazioni di I/O formattato da console

Le funzioni printf() e scanf() eseguono rispettivamente operazioni di output e input formattato; questo significa che sono in grado di scrivere e leggere i dati in vari formati sotto il controllo del programmatore. La funzione printf() scrive i dati sullo schermo. La funzione complementare scanf(), legge i dati dalla tastiera. Entrambe le funzioni accettano qualsiasi tipo di dati interno del C, inclusi i caratteri, le stringhe e i numeri.

**Tabella 8.1** Le principali funzioni di I/O da console.

| FUNZIONE | OPERAZIONE |
|---|---|
| `getchar()` | Legge un carattere dalla tastiera; attende la pressione di INVIO. |
| `getche()` | Legge un carattere dalla tastiera e lo visualizza; non attende la pressione di INVIO: non è definita dallo standard ANSI ma è un'estensione molto comune. |
| `getch()` | Legge un carattere dalla tastiera senza visualizzarlo; non attende la pressione di INVIO: non è definita dallo standard ANSI ma è un'estensione molto comune. |
| `putchar()` | Scrive un carattere sullo schermo. |
| `gets()` | Legge una stringa dalla tastiera. |
| `puts()` | Scrive una stringa sullo schermo. |



## 8.5 La funzione printf()

Il prototipo di printf() è:

int printf(const char **stringa_controllo*, ...);

La funzione printf() restituisce il numero di caratteri scritto o un valore negativo nel caso in cui si verifichi un errore.

La *stringa_controllo* è formata da due tipi di oggetti. Il primo tipo è costituito dai caratteri che verranno visualizzati sullo schermo. Il secondo tipo contiene specificatore di formato che definiscono il modo in cui dovranno essere visualizzati gli argomenti successivi. Uno specificatore di formato inizia con il segno di percentuale ed è seguito dal codice del formato. Il numero degli specificatori di formato deve corrispondere esattamente al numero di argomenti e inoltre vi deve essere una corrispondenza esatta da sinistra a destra. Ad esempio, questa chiamata a printf()

```
printf("Il %c mi %s", 'C', "piace molto!");
```

visualizza la frase

```
Il C mi piace molto!
```

La funzione printf() accetta un'ampia gamma di specificatori di formato, indicati nella Tabella 8.2.

### La visualizzazione dei caratteri

Per visualizzare un singolo carattere, si usa lo specificatore %c. In questo modo, l'argomento corrispondente verrà visualizzato senza alcuna modifica. Per visualizzare una stringa, si deve utilizzare lo specificatore di formato %s.

### La visualizzazione dei numeri

Per visualizzare un numero decimale con segno si può usare sia lo specificatore %d sia %i. Questi specificatori di formato sono equivalenti e vengono conservati entrambi per motivi storici.

Per visualizzare un valore senza segno si utilizza lo specificatore %u, e lo specificatore di formato %f visualizza numeri in virgola mobile.

Gli specificatori %e e %E chiedono di visualizzare un argomento double in notazione scientifica. I numeri in notazione scientifica hanno la seguente forma generale:

x.dddddE+/-yy

Per visualizzare la lettera "E" in maiuscolo, si dovrà utilizzare lo specificatore %E mentre per visualizzare la "e" minuscola si dovrà utilizzare lo specificatore %e.

Si può chiedere al compilatore di decidere se usare %f o %e utilizzando gli specificatori di formato %g e %G. In questo modo, printf() sceglierà lo specificatore di formato che produce l'output più breve. Lo specificatore di formato %G visualizza, se necessario, la lettera esponenziale "E" in lettere maiuscole; %g la visualizza in minuscolo.

**Tabella 8.2** Gli specificatori di formato di printf().

| CODICE | FORMATO |
|---|---|
| %c | Carattere |
| %d | Interi decimali con segno |
| %i | Interi decimali con segno |
| %e | Notazione scientifica (e minuscola) |
| %E | Notazione scientifica (e maiuscola) |
| %f | Numero decimale in virgola mobile |
| %g | Usa il più breve fra %e e %f |
| %G | Usa il più breve fra %E e %F |
| %o | Ottale senza segno |
| %s | Stringa di caratteri |
| %u | Interi decimali senza segno |
| %x | Esadecimale senza segno (lettere minuscole) |
| %X | Esadecimale senza segno (lettere maiuscole) |
| %p | Visualizza un puntatore |
| %n | L'argomento associato è un puntatore a interi in cui viene inserito il numero di caratteri scritti |
| %% | Stampa il carattere "%" |



Il seguente programma mostra gli effetti dello specificatore di formato %g:

```
#include <stdio.h>

int main(void)
{
  double f;

  for(f=1.0; f<1.0e+10; f=f*10)
    printf("%g ", f);

  return 0;
}
```

Il programma produce il seguente output:

```
1 10 100 1000 10000 100000 1e+006 1e+007 1e+008 1e+009
```

Gli interi senza segno possono essere visualizzati in formato ottale o esadecimale utilizzando rispettivamente gli specificatori %o e %x. Poiché il formato numerico esadecimale utilizza le lettere da A a F per rappresentare i numeri da 10 a 15, si può decidere di visualizzare queste lettere in maiuscolo o in minuscolo. Lo specificatore %X visualizza le lettere esadecimali in maiuscolo mentre lo specificatore %x le visualizza in lettere minuscole:

```
#include <stdio.h>

int main(void)
{
  unsigned num;

  for(num=0; num<255; num++) {
    printf("%o ", num);
    printf("%x ", num);
    printf("%X\n", num);
  }

  return 0;
}
```

## La visualizzazione di un indirizzo di memoria

Per visualizzare un indirizzo si utilizza lo specificatore di formato %p. In questo modo la funzione printf() visualizzerà un indirizzo di memoria del computer in un

formato compatibile con il tipo di indirizzamento utilizzato. Il seguente programma visualizza l'indirizzo della variabile sample:

```
#include <stdio.h>

int sample;

int main(void)
{
  printf("%p", &sample);

  return 0;
}
```

## Lo specificatore %n

Lo specificatore di formato %n differisce da ogni altro specificatore. Invece di chiedere a printf() di visualizzare qualcosa, chiede di inserire nella variabile puntata dal suo argomento un valore uguale al numero di caratteri visualizzati. In altre parole, il valore che corrisponde allo specificatore di formato %n deve essere un puntatore a una variabile. Al termine dell'esecuzione di printf(), questa variabile conterrà il numero di caratteri visualizzati, fino al punto in cui è stato inserito lo specificatore %n. Per meglio comprendere questo particolare specificatore di formato, si esamini il seguente programma:

```
#include <stdio.h>

int main(void)
{
  int count;

  printf("questa%n è una prova\n", &count);
  printf("%d", count);

  return 0;
}
```

Questo programma visualizza una stringa seguita dal numero 4. Lo specificatore di formato %n viene normalmente utilizzato per consentire al programma di eseguire una formattazione dinamica del proprio output.



## I modificatori di formato

Molti specificatori di formato prevedono l'uso di modificatori che ne alterano leggermente il significato. Ad esempio, si può specificare un'ampiezza minima di un campo, il numero di cifre esadecimali e l'allineamento a sinistra. Il modificatore di formato deve trovarsi fra il segno di percentuale e il codice di formattazione.

## Lo specificatore di minima ampiezza di campo

Un intero posto fra il segno di percentuale e il codice di formattazione agisce come *specificatore di minima ampiezza del campo*. Questo modificatore inserisce una serie di spazi nell'output in modo da raggiungere sempre almeno la larghezza minima desiderata. Stringhe e numeri più lunghi di questa dimensione minima verranno comunque interamente visualizzati. Normalmente il raggiungimento della larghezza minima specificata viene eseguito tramite spazi. Se invece si desidera utilizzare un carattere diverso, ad esempio lo zero, si dovrà inserire tale carattere prima dello specificatore dell'ampiezza. Ad esempio, lo specificatore %05d aggiunge a un numero composto da meno di cinque cifre una serie di zeri in modo che la sua lunghezza totale sia uguale a cinque. Il seguente programma mostra l'uso dello specificatore di ampiezza minima del campo:

```
#include <stdio.h>

int main(void)
{
  double item;

  item = 10.12304;

  printf("%f\n", item);
  printf("%10f\n", item);
  printf("%012f\n", item);

  return 0;
}
```

Questo programma produce il seguente output:

```
10.123040
 10.123040
00010.123040
```

Lo specificatore di ampiezza minima del campo è utilizzato principalmente per produrre tabelle con colonne allineate. Ad esempio, il programma successivo produce una tabella di quadrati e cubi per i numeri compresi fra 1 e 19:

```
#include <stdio.h>

int main(void)
{
  int i;

  /* visualizza una tabella di quadrati e cubi */
  for(i=1; i<20; i++)
    printf("%8d %8d %8d\n", i, i*i, i*i*i);

  return 0;
}
```

Il programma produce il seguente output:

```
 1        1        1
 2        4        8
 3        9       27
 4       16       64
 5       25      125
 6       36      216
 7       49      343
 8       64      512
 9       81      729
10      100     1000
11      121     1331
12      144     1728
13      169     2197
14      196     2744
15      225     3375
16      256     4096
17      289     4913
18      324     5832
19      361     6859
```

## Lo specificatore di precisione

Lo *specificatore di precisione* segue lo specificatore di ampiezza minima del campo (se presente) ed è formato da un punto seguito da un intero. Il suo esatto significato dipende dal tipo di dati a cui viene applicato.



Quando si applica lo specificatore di precisione a un numero in virgola mobile visualizzato con gli specificatori %f, %e o %E, determina il numero di cifre decimali visualizzate. Ad esempio, %10.4f visualizza un numero assegnandogli almeno dieci caratteri e con quattro cifre decimali. Se non si specifica la precisione, verrà utilizzato il valore standard di sei cifre.

Quando lo specificatore di precisione si applica ai formati %g e %G, indica il numero di cifre significative.

Applicato alle stringhe, lo specificatore di precisione indica la lunghezza massima del campo. Ad esempio, %5.7s visualizza una stringa di almeno cinque caratteri e senza superare la lunghezza di sette caratteri. Se la stringa è più lunga rispetto all'ampiezza massima del campo, verranno troncati i caratteri finali.

Se applicato ai tipi interi, lo specificatore di precisione determina il numero minimo di cifre che devono apparire per ciascun valore. Per ottenere il numero richiesto di cifre, verrà inserita una serie di zeri iniziali.

Il seguente programma illustra l'uso dello specificatore di precisione:

```
#include <stdio.h>

int main(void)
{
  printf("%.4f\n", 123.1234567);
  printf("%3.8d\n", 1000);
  printf("%10.15s\n", "Questa è solo una prova.");

  return 0;
}
```

Il programma produce il seguente output:

```
123.1235
00001000
Questa è solo u
```

## L'allineamento dell'output

Normalmente, tutto l'output di printf() viene allineato a destra. Questo significa che se l'ampiezza del campo è maggiore rispetto ai dati da visualizzare, i dati verranno posizionati sul margine destro del campo. Si può chiedere l'allineamento a sinistra dell'output inserendo il segno meno subito dopo il segno di percentuale. Ad esempio, lo specificatore %-10.2f allinea a sinistra un numero in virgola mobile con due cifre decimali e posizionato in un campo largo dieci caratteri.

Il seguente programma illustra l'uso dell'allineamento a sinistra:

```
#include <stdio.h>

int main(void)
{
  printf("allineato a destra:%8d\n", 100);
  printf("allineato a sinistra:%-8d\n", 100);

  return 0;
}
```

### La gestione di altri tipi di dati

Vi sono due modificatori di formato che consentono a printf() di visualizzare valori interi short e long. Questi modificatori possono essere applicati agli specificatori di tipo d, i, o, u e x. Il modificatore l dice a printf() che i dati relativi sono di tipo long. Ad esempio, %ld chiede la visualizzazione di un numero long int. Il modificatore h chiede a printf() di visualizzare un intero short. Ad esempio, %hu indica che i dati sono di tipo short unsigned int.

Il modificatore L può precedere gli specificatori in virgola mobile e, f e g e chiede la visualizzazione di un valore long double.

### I modificatori * e #

La funzione printf() prevede altri due modificatori di alcuni dei suoi specificatori di formato: * e #.

Se si fa precedere il carattere # agli specificatori g, G, f, E oppure e, si richiede la visualizzazione del punto decimale anche se non vi è alcuna cifra decimale. Se invece si fa precedere allo specificatore di formato x o X il carattere #, il numero esadecimale verrà visualizzato con il prefisso 0x. Se si fa precedere allo specificatore o il carattere #, il numero verrà visualizzato con uno zero iniziale. Questi sono gli unici specificatori di formato che possono impiegare il modificatore #.

Oltre che tramite costanti, gli specificatori di ampiezza minima del campo e di precisione possono essere forniti anche dagli argomenti di printf(). Per sfruttare questa possibilità, si deve utilizzare l'asterisco *. Quando viene letta la stringa di formattazione di printf(), all'asterisco verrà sostituito l'argomento che si trova nella posizione corrispondente. Ad esempio, nella Figura 8.1, l'ampiezza minima del campo è pari a 10, la precisione è 4 e il valore che verrà visualizzato sarà 123.3.



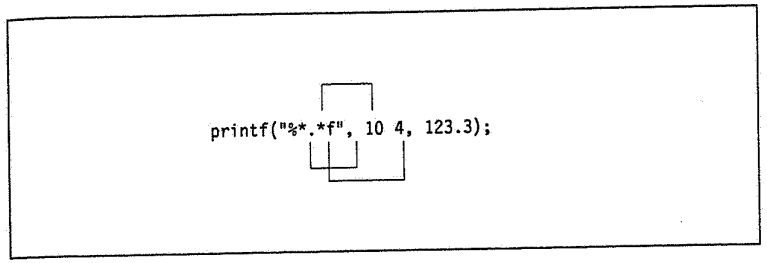



**Figura 8.1** Corrispondenza di valori nell'utilizzo di *.

Il seguente programma illustra l'uso dei modificatori # e *:

```
#include <stdio.h>

int main(void)
{
  printf("%x %#x\n", 10, 10);
  printf("%*.*f", 10, 4, 1234.34);

  return 0;
}
```

## 8.6 La funzione scanf()

La funzione scanf() è la routine di input da console di utilizzo più generale. Questa funzione può leggere valori in tutti i dati interni e converte automaticamente i numeri nel formato interno corretto. Quindi non è solamente l'inverso di printf(). Il prototipo di scanf() è il seguente:

int scanf(const char *_stringa_controllo_, ...);

La funzione scanf() restituisce il numero di dati a cui è stato assegnato un valore. Se si verifica un errore, scanf() restituisce EOF. La *stringa_controllo* determina il modo in cui i valori vengono inseriti nelle variabili puntate dall'elenco di argomenti.

La stringa di controllo è formata da tre tipi di caratteri:

- specificatori di formato
- spazi
- caratteri diversi da spazi

## Gli specificatori di formato

Gli specificatori del formato di input sono preceduti dal segno % e dicono a scanf() il tipo dei dati che devono essere letti. Questi codici sono elencati nella Tabella 8.3. Gli specificatori di formato corrispondono da sinistra a destra agli argomenti presenti nell'elenco argomenti.

## La lettura di numeri

Per leggere un numero intero si possono utilizzare gli specificatori %d o %i. Per leggere un numero in virgola mobile rappresentato in notazione standard o scientifica si possono utilizzare gli specificatori %e, %f o %g.

È possibile utilizzare scanf() per leggere valori interi in formato ottale o esadecimale utilizzando i comandi di formato %o e %x. Lo specificatore %x può essere indicato a piacere in lettere maiuscole o minuscole e consentirà in ogni caso l'immissione di numeri esadecimali sia in lettere maiuscole che minuscole.

Il seguente programma legge un numero ottale e un numero esadecimale:

```
#include <stdio.h>

int main(void)
{
  int i, j;

  scanf("%o%x", &i, &j);
  printf("%o %x", i, j);

  return 0;
}
```

La funzione scanf() termina la lettura di un numero nel momento in cui incontra il primo carattere non numerico.



**Tabella 8.3** Gli specificatori di formato di scanf()

| CODICE | SIGNIFICATO |
|---|---|
| %c | Legge un singolo carattere |
| %d | Legge un intero decimale |
| %i | Legge un intero decimale |
| %e | Legge un numero in virgola mobile |
| %f | Legge un numero in virgola mobile |
| %g | Legge un numero in virgola mobile |
| %o | Legge un numero ottale |
| %s | Legge una stringa |
| %x | Legge un numero esadecimale |
| %p | Legge un puntatore |
| %n | Riceve un valore intero uguale al numero di caratteri letti |
| %u | Legge un intero senza segno |
| %[ ] | Attende l'immissione di un determinato gruppo di caratteri |
| %% | Legge un segno percentuale |

## La lettura di interi senza segno

Per leggere un intero senza segno, si utilizza lo specificatore di formato %u. Ad esempio,

```
unsigned num;
scanf("%u", &num);
```

legge un numero senza segno e inserisce il suo valore in num.

## La lettura di singoli caratteri con scanf()

Come si è detto in precedenza in questo capitolo, è possibile leggere singoli caratteri utilizzando getchar() o una delle sue funzioni derivate. Per eseguire la stessa operazione utilizzando scanf() si utilizza lo specificatore di formato %c. Tuttavia, come molte implementazioni di getchar(), anche scanf() con lo specificatore %c

legge un input bufferizzato a riga. Questa caratteristica rende la funzione scanf() inadatta all'utilizzo in ambienti interattivi.

Anche se gli spazi, i caratteri di tabulazione e i caratteri di fine riga sono utilizzati come separatori di campo nella lettura di altri tipi di dati, quando si deve leggere un singolo carattere gli spazi vengono letti come un qualsiasi altro carattere. Ad esempio, se si immettono i caratteri "x y", questo frammento di codice:

```
scanf("%c%c%c", &a, &b, &c);
```

restituisce il carattere x in a, uno spazio in b e il carattere y in c.

## La lettura di stringhe

La funzione scanf() può essere utilizzata per leggere una stringa dal canale di input utilizzando lo specificatore di formato %s. Con %s, scanf() legge una serie di caratteri fino a incontrare uno spazio vuoto. I caratteri letti vengono inseriti nell'array di caratteri puntato dall'argomento corrispondente e il risultato viene completato dal carattere nullo finale. Per quanto riguarda scanf(), con spazio vuoto si intende il carattere di spazio, un carattere di fine riga, una tabulazione orizzontale, una tabulazione verticale o un carattere di fine pagina. A differenza di gets(), che legge una stringa fino alla pressione del tasto INVIO (codice di line-feed/carriage-return), scanf() legge una stringa fino all'immissione del successivo spazio. Questo significa che non si può utilizzare scanf() per leggere una stringa come "questa è una prova" in quanto il primo spazio conclude il processo di lettura. Per osservare l'effetto dello specificatore %s, si provi a utilizzare questo programma immettendo una stringa composta da più di una parola.

```
#include <stdio.h>

int main(void)
{
  char str[80];

  printf("Immettere una stringa: ");
  scanf("%s", str);
  printf("Ecco la stringa: %s", str);

  return 0;
}
```

Il programma visualizzerà solo la prima parola immessa.



## La lettura di un indirizzo

Per immettere un indirizzo di memoria, si utilizza lo specificatore di formato %p. Questo specificatore fa in modo che scanf() legga un indirizzo nel formato definito dall'architettura della CPU. Ad esempio, questo programma legge un indirizzo e visualizza il contenuto della corrispondente cella di memoria:

```
#include <stdio.h>

int main(void)
{
  char *p;

  printf("Specificare un indirizzo: ");
  scanf("%p", &p);
  printf("Nell'indirizzo %p è contenuto il valore %c\n", p, *p);

  return 0;
}
```

## Lo specificatore %n

Lo specificatore %n chiede a scanf() di assegnare il numero di caratteri letti dal canale di input nel punto in cui si è incontrato lo specificatore, alla variabile puntata dall'argomento corrispondente.

## L'utilizzo di un gruppo di scansione

La funzione scanf() consente l'uso di uno specificatore di formato generico chiamato gruppo di scansione. Un *gruppo di scansione* definisce un gruppo di caratteri. Quando scanf() elabora il gruppo di scansione, immette solo i caratteri definiti dal gruppo di scansione. I caratteri letti verranno assegnati all'array di caratteri puntato dall'argomento che corrisponde al gruppo di scansione. Si definisce un gruppo di scansione immettendo i caratteri desiderati fra parentesi quadre. La parentesi quadra aperta deve però essere preceduta dal segno di percentuale. Ad esempio, il gruppo di scansione seguente chiede a scanf() di leggere solo i caratteri X, Y e Z.

```
%[XYZ]
```

Quando si utilizza un gruppo di scansione, scanf() continua a leggere caratteri e li inserisce nel corrispondente array di caratteri fino a che non incontra un carat-

tere che non appartiene al gruppo di scansione. All'uscita da scanf(), questo array conterrà una stringa chiusa dal carattere nullo e formata da tutti i caratteri letti. Per vedere il funzionamento di questo specificatore di formato, si provi ad eseguire il seguente programma:

```
#include <stdio.h>

int main(void)
{
  int i;
  char str[80], str2[80];

  scanf("%d%[abcdefg]%s", &i, str, str2);
  printf("%d %s %s", i, str, str2);

  return 0;
}
```

Si provi a immettere 123abcdtye e al termine si prema INVIO. Il programma visualizzerà 123 abd tye. Poiché "t" non appartiene al gruppo di scansione, scanf() termina la lettura dei caratteri in str nel momento in cui incontra la lettera "t". I caratteri rimanenti verranno quindi inseriti nella stringa str2.

È anche possibile specificare un gruppo invertito se il primo carattere del gruppo è l'accento circonflesso ^. Il carattere "^" chiede a scanf() di accettare tutti i caratteri *non* definiti nel gruppo di scansione.

In molte implementazioni è possibile definire un intervallo utilizzando un trattino. Ad esempio, lo specificatore seguente chiede a scanf() di accettare tutti i caratteri compresi fra A e Z:

```
%[A-Z]
```

Un fatto molto importante da ricordare è che il gruppo di scansione fa distinzione fra lettere maiuscole e minuscole. Per eseguire la scansione di lettere sia maiuscole che minuscole, sarà necessario specificarle singolarmente.

### Eliminazione degli spazi non desiderati

Uno spazio vuoto nella stringa di controllo fa in modo che scanf() salti uno o più spazi presenti nel canale di input. Lo spazio vuoto può essere uno spazio, una tabulazione orizzontale, una tabulazione verticale, un codice di fine riga o un codice di fine pagina. In pratica, un carattere di spazio nella stringa di controllo fa in modo che scanf() legga ma non memorizzi un qualsiasi numero (anche uguale a zero) di spazi fino al successivo carattere diverso da uno spazio.



### Caratteri diversi da spazi nella stringa di controllo

Un carattere diverso da uno spazio nella stringa di controllo fa in modo che scanf() legga e elimini tutti i corrispondenti caratteri nel canale di input. Ad esempio, "%d,%d" fa in modo che scanf() legga un intero, legga e ignori una virgola e poi legga un altro intero. Se il carattere specificato non viene trovato, scanf() ha termine. Per leggere e ignorare il carattere di %, si deve utilizzare la stringa di controllo %%.

### I parametri di scanf() devono essere indirizzi

Tutte le variabili utilizzate per ricevere valori tramite scanf() devono essere passate utilizzando i relativi indirizzi. Questo significa che tutti gli argomenti devono essere puntatori alle variabili utilizzate come argomenti. Si ricordi che questo è uno dei modi utilizzabili per creare una chiamata per indirizzo e che consente a una funzione di modificare il contenuto di un argomento. Ad esempio, per leggere un intero nella variabile count si deve utilizzare la seguente chiamata a scanf():

```
scanf("%d", &count);
```

Le stringhe verranno lette in array di caratteri e il nome dell'array, senza alcun indice, corrisponde all'indirizzo del primo elemento dell'array. Quindi, per leggere una stringa nell'array di caratteri str si dovrà utilizzare l'istruzione:

```
scanf("%s", str);
```

In questo caso str è già una variabile puntatore e non dovrà pertanto essere preceduta dall'operatore &.

### I modificatori di formato

Come printf(), anche scanf() consente di modificare alcuni dei suoi specificatori di formato.

Gli specificatori di formato possono includere un modificatore di lunghezza massima del campo. Si tratta di un intero che deve trovarsi fra il segno % e lo specificatore di formato il quale limita il numero di caratteri leggibili per un de-

terminato campo. Ad esempio, per leggere non più di venti caratteri ed inserirli nella stringa str, si deve scrivere:

```
scanf("%20s", str);
```

Se nel canale di input sono presenti più di venti caratteri, la chiamata successiva alla funzione inizierà dal punto in cui era arrivata questa chiamata. Ad esempio, se si immette la stringa

ABCDEFGHIJKLMNOPQRSTUVWXYZ

come risposta alla chiamata scanf() di questo esempio, nella stringa str verranno immessi solo i primi venti caratteri, fino alla lettera "T", a causa dello specificatore di ampiezza massima del campo. Questo significa che i caratteri rimanenti, UVWXYZ, rimarranno nel canale di input. Se viene eseguita un'altra chiamata a scanf(), come ad esempio:

```
scanf("%s", str);
```

nella stringa str verranno immesse le lettere UVWXYZ. L'immissione di dati in un campo può avere termine prima della lunghezza massima del campo nel caso in cui si incontri uno spazio vuoto. In questo caso, scanf() si posiziona sul campo successivo. Per leggere un intero long, si deve inserire una l davanti allo specificatore di formato. Per leggere un intero short, si deve inserire una h davanti allo specificatore di formato. Questi modificatori possono essere utilizzati con i codici di formato d, i, o, u e x.

Normalmente, gli specificatori f, r e g chiedono a scanf() di assegnare dati ad un float. Se si inserisce una l davanti a uno di questi specificatori, scanf() assegnerà i dati a un double. Se si utilizza una L si informa scanf() che la variabile che riceve i dati è un long double.

### Soppressione dell'input

Si può chiedere a scanf() di leggere un campo e di non assegnarlo ad alcuna variabile facendo precedere al codice di formato del campo il carattere *. Ad esempio, data la chiamata:

```
scanf("%d%*c%d", &x, &y);
```

si potrà immettere la coppia di coordinate 10,10. La virgola verrà letta correttamente ma non verrà assegnata a nulla. La soppressione dell'assegnamento è particolarmente utile quando si deve elaborare solo una parte dei dati immessi.

Capitolo 9

# Operazioni di I/O da file


9.1 **Operazioni di I/O C e C++**
9.2 **Stream e file**
9.3 **Gli stream**
9.4 **I file**
9.5 **Principi di funzionamento del file system**
9.6 **fread() e fwrite()**
9.7 **fseek() e operazioni di I/O ad accesso diretto**
9.8 **fprint() e fscanf()**
9.9 **Gli stream standard**


Questo capitolo descrive il sistema di I/O su file di tipo C. Come si è detto nel Capitolo 8, il linguaggio C++ supporta due diversi sistemi di I/O: quello ereditato dal C e quello a oggetti definito dal C++. Questo capitolo si occupa del file system C (il file system C++ verrà discusso nella Parte seconda). Anche se il codice sviluppato di recente utilizza il file system C++, la conoscenza del file system C è importante per i motivi elencati all'inizio del capitolo precedente.

## 9.1 Operazioni di I/O C e C++

Talvolta si fa un po' di confusione sulle relazioni esistenti fra I/O C e C++. Innanzitutto occorre dire che il C++ supporta l'intero sistema di I/O su file del C. Pertanto, la trasformazione di codice C in C++ non richiede alcuna modifica alle routine di I/O del programma. In secondo luogo, il C++ definisce un proprio sistema di I/O a oggetti che include funzioni e operatori di I/O. Il sistema di I/O del C++ duplica completamente le funzionalità del sistema di I/O del C che risulta pertanto ridondante. In generale, anche se probabilmente si preferirà utilizzare il sistema C++, si sarà comunque liberi di scegliere il file system C. naturalmente la maggior parte dei programmatori C++ preferisce impiegare il sistema di I/O C++ per motivi che diverranno chiari leggendo la Parte seconda di questa guida.

## 9.2 Stream e file

Prima di iniziare la discussione sul file system del C, è importante comprendere la differenza esistente fra i termini *stream* e *file*. Il sistema di I/O del C presenta un interfaccia consistente per il programmatore, indipendente dal dispositivo effettivamente utilizzato. Questo significa che il sistema di I/O del C fornisce un livello di astrazione che si interpone fra il programmatore e il dispositivo. Questa astrazione è chiamata *stream* e il dispositivo effettivo è chiamato *file*. È importante comprendere le interazioni che si svolgono tra stream e file.

NOTA *I concetti di stream e file sono importanti anche per il sistema di I/O del C++ descritto nella Parte seconda.*

## 9.3 Gli stream

Il file system del C è progettato per funzionare con un'ampia varietà di dispositivi, come terminali, unità disco e unità nastro. Anche se ogni dispositivo è molto diverso da un altro, il file system bufferizzato trasforma ogni dispositivo fisico in un dispositivo logico chiamato stream. Tutti gli stream si comportano in modo analogo. Poiché gli stream sono in gran parte indipendenti dai dispositivi, le stesse funzioni saranno in grado di scrivere su un file su disco ma potranno anche essere utilizzate per scrivere su un altro tipo di dispositivo, come ad esempio lo schermo. Vi sono due tipi di stream: stream di testo e binari.

### Stream di testo

Uno *stream di testo* è formato da una sequenza di caratteri. Lo standard C consente (ma non richiede) che uno stream di testo sia organizzato in righe concluse da un carattere di fine riga. Tuttavia, il carattere di fine riga è opzionale sull'ultima riga (in effetti, molti compilatori C/C++ non concludono gli stream di testo con un carattere di fine riga). In uno stream di testo, possono svolgersi determinate traduzioni di caratteri richieste dall'ambiente ospite. Ad esempio, un carattere di fine riga può essere convertito in una coppia carriage-return/line-feed. Pertanto, potrebbe non esservi una relazione uno-a-uno fra i caratteri scritti (o letti) e quelli presenti sul dispositivo fisico. Inoltre, a causa della possibilità di traduzioni, il numero di caratteri effettivamente scritti (o letti) potrebbe essere diverso dal numero di caratteri presenti nel dispositivo fisico.



### Stream binari

Uno stream *binario* è formato da una sequenza di byte con una corrispondenza uno a uno con i byte presenti sul dispositivo fisico (questo significa che non viene eseguita alcuna traduzione dei caratteri). Inoltre, il numero di byte scritti (o letti) corrisponde al numero di byte presenti nel dispositivo fisico. A uno stream binario può però essere aggiunto un numero di byte nulli definito dall'implementazione. Questi byte nulli possono essere utilizzati per allineare le informazioni in modo, ad esempio, da riempire un settore di un disco.

## 9.4 I file

In C/C++, un *file* può corrispondere a qualsiasi cosa, da un disco a un terminale a una stampante. Si associa uno stream a un determinato file eseguendo un'operazione di apertura. Una volta che il file è aperto, sarà possibile scambiare informazioni fra il file e il programma.

Non tutti i file hanno le stesse funzionalità. Ad esempio, un file su disco può consentire operazioni di accesso diretto mentre alcune stampanti no. Questo introduce un fatto importante relativo al sistema di I/O del C: tutti gli stream sono uguali mentre i file no.

Se il file consente operazioni di *posizionamento*, l'apertura di tale file inizializza anche l'*indicatore di posizione nel file* assegnandogli la posizione iniziale del file. Mano a mano che si leggono o scrivono caratteri sul file, l'indicatore di posizione viene incrementato, seguendo le operazioni svolte dal programma.

Per eliminare l'associazione fra un file e un determinato stream si utilizza l'operazione di chiusura. Se si chiude un file aperto in output, l'eventuale contenuto dello stream ad esso associato viene scritto sul dispositivo interno. Questo processo viene chiamato di *svuotamento* dello stream e garantisce che nessuna informazione venga accidentalmente lasciata nel buffer del disco. Tutti i file vengono chiusi automaticamente, se il programma termina normalmente, da main() che ritorna al sistema operativo o da una chiamata a exit(). I file non vengono invece chiusi se un programma termina in modo anormale, ad esempio quando blocca il sistema o quando viene eseguita una chiamata ad abort().

Ogni stream a cui è associato un file ha una propria struttura di controllo di tipo FILE. Si faccia attenzione a non modificare mai questo blocco di controllo del file.

Se si è alle prime armi nella programmazione, la distinzione fra gli stream e i file può sembrare inutile e superflua. Ma si ricordi che lo scopo principale consi-

ste nel fornire un interfaccia uniforme. Basterà quindi pensare in termini di stream e utilizzare un solo file system per eseguire tutte le operazioni di I/O. Il sistema di I/O convertirà quindi automaticamente le semplici operazioni di input o output tipiche di ogni dispositivo nelle operazioni di alto livello dello stream.

## 9.5 Principi di funzionamento del file system

Il file system C è formato da numerose funzioni correlate. Le funzioni più comunemente utilizzate sono elencate nella Tabella 9.1. Queste funzioni richiedono l'inclusione nel programma del file header stdio.h. I programmi C++ possono utilizzare il file header <cstdio>. I file header stdio.h e <cstdio> contengono i prototipi delle funzioni di I/O e definiscono tre tipi: size_t, fpos_t e FILE. Il tipo size_t è in genere un intero unsigned, come fpos_t. Il tipo FILE verrà discusso nella prossima sezione del capitolo.

**Tabella 9.1** Le funzioni più utilizzate per il file system di tipo ANSI.

| NOME | FUNZIONE |
|---|---|
| `fopen()` | Apre un file |
| `fclose()` | Chiude un file |
| `putc()` | Scrive un carattere su un file |
| `fputc()` | Come `putc()` |
| `getc()` | Legge un carattere da un file |
| `fgetc()` | Come `getc()` |
| `fgets()` | Legge una stringa da un file |
| `fputs()` | Scrive una stringa su un file |
| `fseek()` | Si posiziona su un determinato byte di un file |
| `ftell()` | Restituisce la posizione del file |
| `fprintf()` | Esegue su un file ciò che `printf()` esegue sulla console |
| `fscanf()` | Esegue su un file ciò che `scanf()` esegue sulla console |
| `feof()` | Restituisce il valore vero quando viene raggiunta la fine del file |
| `ferror()` | Restituisce il valore vero se si verifica un errore |
| `rewind()` | Riporta l'indicatore di posizione del file all'inizio del file |
| `remove()` | Cancella un file |
| `fflush()` | Scarica sul file il contenuto del buffer in memoria |



I file header definiscono anche numerose macro. Quelle più importanti per gli scopi di questo capitolo sono NULL, EOF, FOPEN_MAX, SEEK_SET, SEEK_CUR e SEEK_END. La macro NULL definisce un puntatore nullo. La macro EOF è generalmente definita uguale a -1 ed è il valore restituito quando una funzione di input cerca di leggere oltre la fine del file. FOPEN_MAX definisce un valore intero che determina il numero di file che possono essere contemporaneamente aperti. Le altre macro sono utilizzate con fseek(), la funzione che consente di eseguire accessi casuali a un file.

### Il puntatore del file

Il puntatore del file è il filo conduttore che unifica il sistema di I/O C. Un *puntatore a file* è un puntatore a una struttura di tipo FILE. Esso punta a informazioni che definiscono vari fattori relativi al file, incluso il nome, lo stato e la posizione corrente del file. In pratica, il puntatore a file identifica un determinato file del disco e viene utilizzato dallo stream ad esso associato per dirigere il funzionamento delle funzioni di I/O. Per leggere o scrivere i file, il programma deve utilizzare il puntatore a file. Per ottenere una variabile di tipo puntatore a file, si utilizza un'istruzione simile alla seguente:

```
FILE *fp;
```

### Apertura di un file

La funzione fopen() apre uno stream e vi collega un file; quindi restituisce il puntatore associato a tale file. La maggior parte delle volte (e per la parte rimanente di questa discussione) il file si trova su disco. La funzione fopen() ha il seguente prototipo:

FILE *fopen(const char **nomefile*, const char **modalità*);

dove *nomefile* è un puntatore a una stringa di caratteri che contiene un nome valido per un file e può includere l'indicazione di un percorso di directory. La stringa puntata da *modalità* determina il modo in cui il file verrà aperto. La Tabella 9.2 mostra i valori consentiti per *modalità*. Le stringhe come "r+b" possono essere anche rappresentati come "rb+".

Come si è detto, la funzione fopen() restituisce un puntatore a file. Il programma non dovrà mai modificare il valore di questo puntatore. Se si verifica un errore quando si cerca di aprire il file, fopen() restituirà un puntatore nullo.

**Tabella 9.2** Valori consentiti per modalità.

| MODALITÀ | SIGNIFICATO |
|---|---|
| r | Apre un file di testo in lettura |
| w | Crea un file di testo in scrittura |
| a | Apre un file di testo in modalità append (aggiunta di dati) |
| rb | Apre un file binario in lettura |
| wb | Crea un file binario in scrittura |
| ab | Apre un file binario in modalità append (aggiunta di dati) |
| r+ | Apre un file di testo in lettura/scrittura |
| w+ | Crea un file di testo in lettura/scrittura |
| a+ | Crea o apre in modalità append (aggiunta di dati) un file di testo per operazioni di lettura/scrittura |
| r+b | Apre un file binario in lettura/scrittura |
| w+b | Crea un file binario in lettura/scrittura |
| a+b | Crea o apre in modalità append (aggiunta) un file binario per operazioni di lettura/scrittura |

Il codice seguente utilizza fopen() per aprire in output un file chiamato TEST.

```
FILE *fp;
.
.
fp = fopen("test", "w");
```

Anche se è tecnicamente corretto, il codice precedente viene normalmente scritto in questo modo:

```
FILE *fp;

if ((fp = fopen("test","w"))==NULL) {
  printf("Impossibile aprire il file.\n");
  exit(1);
}
```

Questo metodo ha il vantaggio di rilevare eventuali errori di apertura di un file, ad esempio in caso di disco protetto in scrittura o di disco pieno, prima che il programma tenti di scrivervi. In generale, prima di cercare di utilizzare il file occorre assicurarsi che la chiamata a fopen() sia stata eseguita con successo.



Anche se la maggior parte delle modalità di apertura dei file è autoesplicativa, è opportuno commentare un attimo l'argomento. Se, quando si apre un file per operazioni di sola lettura, il file non esiste, la funzione fopen() non ha successo. Quando si apre un file in modalità append, se il file non esiste, verrà creato. Inoltre, quando un file viene aperto in modalità append, tutti i nuovi dati avverranno scritti alla fine del file. Il contenuto originale del file non verrà modificato. Se, quando un file viene aperto in scrittura, il file non esiste, verrà creato. Se esiste, il contenuto del file verrà distrutto e sostituito dai nuovi dati che vi verranno scritti. La differenza fra le modalità r+ e w+ è il fatto che r+ non crea un file che non esiste. Inoltre, se il file esiste, con w+ ne viene distrutto il contenuto, al contrario di quanto avviene con r+.

Come si può vedere nella Tabella 9.2, un file può essere aperto in modalità testo o in modalità binaria. Nella maggior parte delle implementazioni, in modalità testo le sequenze Carriage return / Line feed vengono tradotte in un carattere Newline. In output accade il contrario: i caratteri Newline vengono tradotti in Carriage return / Line feed. Sui file binari tale traduzione non avviene.

Il numero di file che possono essere aperti contemporaneamente è specificato dalla macro FOPEN_MAX. Questo valore è normalmente uguale almeno a 8 ma a tale proposito è sempre bene consultare il manuale del compilatore.

## Chiusura di un file

La funzione fclose() chiude uno stream che era stato precedentemente aperto con una chiamata a fopen(). La funzione scrive sul file i dati eventualmente rimasti nel buffer del disco e quindi esegue una richiesta al sistema operativo di chiusura del file. La mancata chiusura di uno stream può provocare ogni genere di problemi, inclusi la perdita di dati, la distruzione di file ed eventuali errori intermittenti nel programma. La chiamata a fclose() libera anche il blocco di controllo del file associato allo stream, rendendolo nuovamente disponibile. Nella maggior parte dei casi, vi è un limite determinato dal sistema operativo al numero di file apribili contemporaneamente, e quindi si potrebbe essere costretti a chiudere un file prima di aprirne un altro. Il prototipo della funzione fclose() è il seguente:

```
int fclose(FILE *pf);
```

dove *pf* è il puntatore a file restituito dalla chiamata a fopen(). Se il valore restituito è uguale a zero, significa che la chiusura del file è avvenuta con successo. In caso di errore, la funzione restituisce il valore EOF. Per determinare eventuali problemi, si può utilizzare la funzione standard ferror() discussa fra breve. Generalmente, fclose() entrerà in errore solo quando un disco viene prematuramente estratto dal drive o quando non vi è più spazio sul disco.

## Scrittura di un carattere

Il sistema di I/O C definisce due funzioni equivalenti che scrivono un carattere: putc() e fputc(). In realtà, putc() è normalmente implementata come macro. Vi sono due funzioni identiche solamente per conservare la compatibilità con le versioni precedenti del C. In questa guida si utilizza putc() ma, se si preferisce, si può usare fputc().

La funzione putc() scrive caratteri su un file precedentemente aperto in scrittura utilizzando la funzione fopen(). Il prototipo di questa funzione è:

int putc(int *car*, FILE **pf*);

dove *pf* è il puntatore a file restituito da fopen() e *car* è il carattere che deve essere scritto sul file. Il puntatore a file dice a putc() su quale file si deve scrivere il carattere. Per motivi storici, la variabile *car* è definita come int ma di essa verrà scritto solo il byte di ordine inferiore.

Se un'operazione putc() ha successo, restituirà il carattere scritto. In caso contrario, restituirà EOF.

## Lettura di un carattere

Vi sono due funzioni equivalenti anche per la lettura di un carattere: getc() e fgetc(). Sono definite entrambe per garantire la compatibilità con le versioni meno recenti di C. In questa guida si utilizza getc() (che in effetti è implementata come macro) ma, se si preferisce, si può usare fgetc().

La funzione getc() legge caratteri da un file aperto in modalità di lettura tramite fopen(). Il prototipo di getc() è il seguente:

int putc(int *car*, FILE **pf*);

dove *pf* è un puntatore a file di tipo FILE restituito da fopen(). Per motivi storici, getc() restituisce un intero, ma il carattere è contenuto nel byte di ordine inferiore. Se non si verifica un errore,il byte di ordine superiore sarà sempre uguale a zero.

Quando viene raggiunta la fine del file, la funzione getc() restituisce EOF. Pertanto, per leggere dalla fine di un file di testo si può utilizzare il codice seguente:

```
do {
ch = getc(fp);
} while(ch!=EOF);
```



Tuttavia, getc() restituisce EOF anche quando si verifica un errore. Per determinare con precisione ciò che è avvenuto, si può utilizzare ferror().

## Uso di fopen(), getc(), putc() e fclose()

Le funzioni fopen(), getc(), putc() e fclose() formano il gruppo minimo di routine per le operazioni di I/O su file. Il programma seguente, KTOD, è un semplice esempio d'uso delle funzioni putc(), fopen() e fclose(). Il programma legge semplicemente caratteri dalla tastiera e li scrive su un file finché l'utente non immette il segno di dollaro. Il nome del file deve essere specificato nella riga di comando. Ad esempio, se si chiama questo programma KTOD, scrivendo KTOD TEST sarà possibile immettere righe di testo in un file chiamato TEST.

```
/* KTOD: Un programma di scrittura su file */
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
  FILE *fp;
  char ch;

  if(argc!=2) {
    printf("Immettere il nome del file.\n");
    exit(1);
  }

  if((fp=fopen(argv[1], "w"))==NULL) {
    printf("Impossibile aprire il file.\n");
    exit(1);
  }

  do {
    ch = getchar();
    putc(ch, fp);
  } while (ch!='$');

  fclose(fp);

  return 0;
}
```

Il programma complementare DTOS legge un file di testo e ne visualizza il contenuto sullo schermo.

```
/* DTOS: Legge il contenuto di un file e lo visualizza sullo schermo. */
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{  FILE *fp;
  char ch;

  if(argc!=2) {
    printf("Immettere il nome del file.\n");
    exit(1);
  }

  if((fp=fopen(argv[1], "r"))==NULL) {
    printf("Impossibile aprire il file.\n");
    exit(1);
  }

  ch = getc(fp);   /* legge un carattere */

  while (ch!=EOF) {
    putchar(ch);  /* lo visualizza */
    ch = getc(fp);
  }

  fclose(fp);

  return 0;
}
```

Si provi a usare questi due programmi. Prima si può usare KTOD per creare un file di testo e quindi si può leggere il contenuto del file utilizzando DTOS.

## Uso di feof()

Come si è detto, quando si raggiunge la fine del file, getc() restituisce EOF. Tuttavia, la verifica del valore restituito da getc() può non essere il modo migliore per determinare se si è arrivati alla fine del file. Innanzitutto il sistema di I/O su file opera su file binari e di testo. Quando il file viene aperto per operazioni di input binario, può essere letto un valore intero uguale al codice EOF. Questo può far



pensare di aver raggiunto la condizione di fine file anche quando non viene raggiunta la fine fisica del file. In secondo luogo, getc() restituisce EOF anche quando fallisce l'operazione di lettura (oltre che alla fine del file). Utilizzando il solo valore restituito da getc() è impossibile capire cosa è capitato. Per risolvere questo problema, il C include la funzione feof(), che determina il raggiungimento della fine del file. Il prototipo della funzione feof() è il seguente:

```
int feof(FILE *pf);
```

La funzione feof() restituisce il valore logico vero quando viene raggiunta la fine del file e zero in tutti gli altri casi. Pertanto, la seguente routine legge dati da un file binario finché non viene raggiunta la fine del file:

```
while(!feof(fp)) ch = getc(fp);
```

Naturalmente, si può applicare questo metodo anche ai file di testo oltre che ai file binari.

Il seguente programma, che copia file di testo binari, contiene un esempio d'uso di feof(). I file vengono aperti in modalità binaria e feof() controlla quando viene raggiunta la fine del file.

```
/* Copia un file. */
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
  FILE *in, *out;
  char ch;

  if(argc!=3) {
    printf("Immettere il nome del file.\n");
    exit(1);
  }

  if((in=fopen(argv[1], "rb"))==NULL) {
    printf("Impossibile aprire il file di origine.\n");
    exit(1);
  }
  if((out=fopen(argv[2], "wb")) == NULL) {
    printf("Impossibile aprire il file di destinazione.\n");
    exit(1);
  }
```

```
  /* Questa parte del codice copia il file. */
  while(!feof(in)) {
    ch = getc(in);
    if(!feof(in)) putc(ch, out);
  }

 fclose(in);
  fclose(out);

  return 0;
}
```

## Le stringhe: fputs() e fgets()

Oltre a getc() e putc(), il sistema di I/O C è dotato di due funzioni correlate, fgets() e fputs(), che leggono e scrivono stringhe di caratteri da un file su disco. Queste funzioni operano in modo analogo a putc() e getc() ma invece di leggere o scrivere un singolo carattere, operano su intere stringhe. I prototipi di queste funzioni sono:

int fputs(const char **str*, FILE ***pf*);
char *fgets(char ***str*, int *lunghezza*, FILE ***pf*);

La funzione fputs() scrive sullo stream specificato la stringa puntata da *str* e in caso di errore restituisce il valore EOF.

La funzione fgets() legge una stringa dallo stream specificato fino al raggiungimento di un carattere di fine riga o fino alla lettura di *lunghezza*-1 caratteri. Se viene letto un codice di fine riga, questo entrerà a far parte della stringa (a differenza di quanto avviene con la funzione gets()). La stringa risultante verrà conclusa con un carattere nullo. La funzione restituisce *str* se ha successo o un puntatore nullo in caso di errore.

Il programma seguente dimostra l'uso di fputs(). Il programma legge stringhe dalla tastiera e le scrive sul file TEST. Per uscire dal programma basta immettere una riga vuota. Poiché gets() non memorizza il carattere di fine riga, il programma ne aggiunge manualmente uno prima della scrittura della stringa sul file, in modo che il file stesso possa essere letto con più facilità.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(void)
```



```
{
  char str[80];
  FILE *fp;

  if((fp = fopen("TEST", "w"))==NULL) {
    printf("Impossibile aprire il file.\n");
    exit(1);
  }

  do {
    printf("Immettere una stringa (INVIO per uscire):\n");
    gets(str);
    strcat(str, "\n");  /* aggiunge un codice di fine riga */
    fputs(str, fp);
  } while(*str!='\n');

  return 0;
}
```

## rewind()

La funzione rewind() riporta l'indicatore di posizione del file all'inizio del file specificato come argomento. In pratica "riavvolge" il file. Il suo prototipo è:

void rewind(FILE ***pf*);

dove *pf* è un puntatore a file valido.

Per vedere un esempio di rewind(), si può modificare il programma della sezione precedente in modo che visualizzi il contenuto del file appena creato. Per ottenere ciò, il programma deve riavvolgere il file al termine delle operazioni di input e quindi utilizza fgets() per rileggere il file. Si noti che ora il file deve essere aperto in modalità di lettura e scrittura utilizzando "W+" come parametro di modalità.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(void)
{
  char str[80];
  FILE *fp;
```

```
  if((fp = fopen("TEST", "w+"))==NULL) {
    printf("Impossibile aprire il file.\n");
    exit(1);
  }

  do {
    printf("Immettere una stringa (INVIO per uscire):\n");
    gets(str);
    strcat(str, "\n");  /* aggiunge un codice di fine riga */
    fputs(str, fp);
  } while(*str!='\n');

  /* ora, legge e visualizza il file */
  rewind(fp);  /* riporta l'indicatore di posizione all'inizio del file. */
  while(!feof(fp)) {
    fgets(str, 79, fp);
    printf(str);
  }

  return 0;
}
```

## ferror()

La funzione ferror() determina se un'operazione svolta su un file ha prodotto un errore. Il prototipo della funzione ferror() è il seguente:

int ferror(FILE *pf);

dove *pf* è un puntatore a file valido. Nel caso in cui si sia verificato un errore durante l'ultima operazione sul file, la funzione restituisce il valore vero; in caso contrario, restituisce il valore logico falso. Poiché lo stato di errore viene impostato da ogni operazione su file, è necessario richiamare ferror() subito dopo ogni operazione svolta su file; in caso contrario, la condizione d'errore verrà persa.

Il seguente programma illustra l'uso di ferror() eliminando le tabulazioni da un file di testo e sostituendole con un numero appropriato di spazi. Le dimensioni delle tabulazioni sono definite da TAB_SIZE. Si noti come ferror() venga richiamata dopo ogni operazione svolta sui file. Per utilizzare il programma, si devono specificare i nomi dei file di input e di output sulla riga di comando.

```
/* Il programma sostituisce alle tabulazioni una serie di spazi
   in un file di testo e controlla il verificarsi di errori. */

#include <stdio.h>
#include <stdlib.h>

#define TAB_SIZE 8
#define IN 0
#define OUT 1

void err(int e);

int main(int argc, char *argv[])
{
  FILE *in, *out;
  int tab, i;
  char ch;

  if(argc!=3) {
    printf("uso: detab <in> <out>\n");
    exit(1);
  }

  if((in = fopen(argv[1], "rb"))==NULL) {
    printf("Impossibile aprire %s.\n", argv[1]);
    exit(1);
  }
 if((out = fopen(argv[2], "wb"))==NULL) {
    printf("Impossibile aprire %s.\n", argv[1]);
    exit(1);
  }

  tab = 0;
  do {
    ch = getc(in);
    if(ferror(in)) err(IN);

    /* Se viene trovata una tabulazione, scrive un numero di spazi appropria-
 to */
    if(ch=='\t') {
      for(i=tab; i<8; i++) {
        putc(' ', out);
        if(ferror(out)) err(OUT);
      }
      tab = 0;
    }
    else {
      putc(ch, out);
      if(ferror(out)) err(OUT);
```

```
      tab++;
      if(tab==TAB_SIZE) tab = 0;
      if(ch=='\n' || ch=='\r') tab = 0;
    }
  } while(!feof(in));
  fclose(in);
  fclose(out);

  return 0;
}

void err(int e)
{
  if(e==IN) printf("Errore di input.\n");
  else printf("Errore di output.\n");
  exit(1);
}
```

### Cancellazione di file

La funzione remove() cancella il file specificato. Il suo prototipo è:

int remove(const char **nomefile*);

Quando viene eseguita con successo, la funzione restituisce zero. In caso contrario restituisce un valore diverso da zero.

Il programma seguente cancella il file specificato nella riga di comando dando la possibilità di annullare l'operazione prima di eseguirla. Un programma di questo tipo può essere utile per gli utenti alle prime armi.

```
/* Doppia verifica prima della cancellazione. */
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>

int main(int argc, char *argv[])
{
  char str[80];
  if(argc!=2) {
    printf("uso: xerase <nomefile>\n");
    exit(1);
  }
```



```
  printf("Devo cancellare %s? (S/N): ", argv[1]);
  gets(str);

  if(toupper(*str)=='S')
    if(remove(argv[1])) {
      printf("Impossibile cancellare il file.\n");
      exit(1);
    }
  return 0;  /* ritorna con successo al sistema operativo */
}
```

### Svuotamento di uno stream

Per vuotare il contenuto di uno stream di output si utilizza la funzione fflush() il cui prototipo è il seguente:

int fflush(FILE **pf*);

Questa funzione scrive il contenuto di un buffer nel file associato a *pf*. Se si richiama fflush() con *pf* nullo, verranno svuotati i buffer di tutti i file.

Se viene eseguita con successo, la funzione fflush() restituisce 0; in caso contrario restituisce EOF.

## 9.6 fread() e fwrite()

Per leggere e scrivere tipi di dati più lunghi di un byte, il file system del C ANSI fornisce le due funzioni fread() e fwrite(). Queste funzioni consentono di leggere e scrivere blocchi di dati di qualsiasi dimensione. I loro prototipi sono i seguenti:

size_t fread(void ***buffer*, size_t *num_byte*, size_t *numero*, FILE *fp);
size_t fwrite(const void ***buffer*, size_t *num_byte*, size_t *numero*, FILE *fp);

Per fread(), *buffer* è un puntatore a una regione di memoria che riceverà i dati letti dal file. Per fwrite(), *buffer* è un puntatore a una regione di memoria che conserva i dati da scrivere sul file. Il valore di *num* determina il numero di oggetti letti o scritti, ognuno dei quali ha una lunghezza pari a *num_byte* byte (si ricordi che il tipo size_t è definito come un intero unsigned). Infine, *pf* è un puntatore a file corrispondente a uno stream precedentemente aperto.

La funzione fread() restituisce il numero di oggetti letti. Questo valore può essere minore di *numero* quando viene raggiunta la fine del file o quando si veri-

fica un errore. La funzione fwrite() restituisce il numero di oggetti scritti. Questo valore è sempre uguale a *numero* sempre che non si verifichi un errore.

### Uso di fread() e fwrite()

Se un file è stato aperto per operazioni su dati binari, fread() e fwrite() possono leggere e scrivere ogni tipo di informazioni. Ad esempio, il seguente programma scrive su un file e poi rilegge un double, un int e un long. Si noti l'uso di sizeof per determinare la lunghezza di ogni tipo di dati.

```
/* Scrive e poi rilegge una serie di valori diversi da caratteri.  */
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
  FILE *fp;
  double d = 12.23;
  int i = 101;
  long l = 123023L;

  if((fp=fopen("test", "wb+"))==NULL) {
    printf("Impossibile aprire il file.\n");
    exit(1);
  }

  fwrite(&d, sizeof(double), 1, fp);
  fwrite(&i, sizeof(int), 1, fp);
  fwrite(&l, sizeof(long), 1, fp);

  rewind(fp);

  fread(&d, sizeof(double), 1, fp);
  fread(&i, sizeof(int), 1, fp);
  fread(&l, sizeof(long), 1, fp);

  printf("%f %d %ld", d, i, l);

  fclose(fp);

  return 0;
}
```



Come si può vedere in questo programma, il buffer può essere (e spesso è) semplicemente la memoria utilizzata per contenere una variabile. In questo semplice programma, i valori restituiti da fread() e fwrite() vengono ignorati. Nell'utilizzo pratico invece questi valori devono essere sempre controllati per evitare errori.

Una delle applicazioni più utili di fread() e fwrite() riguarda la lettura e la scrittura di tipi di dati definiti dall'utente, specialmente strutture. Ad esempio, data la seguente struttura:

```
struct struct_type {
  float balance;
  char name[80];
} cust;
```

la seguente istruzione scrive il contenuto di cust sul file puntato da fp.

```
fwrite(&cust, sizeof(struct struc_type), 1, fp;
```

## 9.7 fseek() e operazioni di I/O ad accesso diretto

Per eseguire operazioni di lettura e scrittura diretta con il sistema di I/O C, si utilizza la funzione fseek() che imposta la posizione dell'indicatore di file. Il suo prototipo è il seguente:

int fseek(FILE **pf*, long *num_byte*, int *origine*);

Qui, *pf* è un puntatore a file restituito da una chiamata a fopen(). *num_byte* è il numero di byte a partire da *origine* in cui si intende portare l'indicatore di posizione del file mentre *origine* può essere una delle seguenti macro:

| ORIGINE | NOME MACRO |
|---|---|
| Inizio del file | SEEK_SET |
| Posizione corrente | SEEK_CUR |
| Fine del file | SEEK_END |

Pertanto, per posizionare l'indicatore di posizione del file a *num_byte* rispetto all'inizio del file, si dovrà utilizzare come *origine* SEEK_SET. Per eseguire il posizionamento sulla base della posizione attuale nel file, si dovrà utilizzare

SEEK_CUR e per eseguire il posizionamento sulla base della fine del file, si dovrà usare SEEK_END. La funzione fseek() restituisce 0 quando viene eseguita con successo e un valore diverso da zero in caso di errore.

Il seguente programma illustra l'uso di fseek(). Il frammento si posiziona su un determinato byte di un file e ne visualizza il contenuto. Il nome del file e il byte su cui posizionarsi devono essere indicati nella riga di comando.

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
  FILE *fp;

  if(argc!=3) {
    printf("Uso: SEEK nomefile byte\n");
    exit(1);
  }

  if((fp = fopen(argv[1], "r"))==NULL) {
    printf("Impossibile aprire il file.\n");
    exit(1);
  }

  if(fseek(fp, atol(argv[2]), SEEK_SET)) {
    printf("Errore di posizionamento.\n");
    exit(1);
  }

  printf("Il byte all'indirizzo %ld contiene %c.\n", atol(argv[2]),
getc(fp));
  fclose(fp);

  return 0;
}
```

Si può usare fseek() per posizionarsi a un multiplo di qualsiasi tipo di dati moltiplicando semplicemente le dimensioni dei dati per il numero di oggetti da saltare. Ad esempio, si immagini di avere un file contenente un elenco di indirizzi costituito da strutture di tipo list_type. Il seguente frammento di codice si posizionerà sul decimo indirizzo.

```
fseek(fp, 9*sizeof(struct list_type), SEEK_SET);
```



Per determinare la posizione corrente all'interno di un file si usa la funzione ftell(). Il suo prototipo è:

```
long ftell(FILE *fp)
```

Questa funzione restituisce la posizione corrente all'interno del file associato a fp. In caso di errore viene restituito il valore -1.

In generale, l'accesso diretto dovrebbe essere utilizzato solo su file binari. Il motivo di ciò è semplice. Poiché ai file di testo può essere applicata una traduzione dei caratteri, potrebbe non esservi una corrispondenza diretta fra il contenuto del file e il byte in cui viene eseguito il posizionamento. L'unico caso in cui si dovrebbe usare fseek() con un file di testo si verifica quando ci si deve portare su una porzione precedentemente determinata da ftell(), utilizzando come origine SEEK_SET.

Un ultimo elemento importante: anche un file che contiene testo può essere aperto come un file binario. Non esistono divieti all'uso di operazioni accesso diretto su file contenenti testo. Questa restrizione si applica solo ai file di testo aperti come file di testo.

## 9.8 fprint() e fscanf()

Oltre alle funzioni di I/O di base discusse precedentemente, il sistema di I/O ANSI include le funzioni fprint() e fscanf(). Queste funzioni si comportano esattamente come printf() e scanf() tranne per il fatto che operano su file. I prototipi di fprint() e fscanf() sono i seguenti:

```
int fprintf(FILE *pf, const char *stringa_controllo,. . .);
int fscanf(FILE *pf, const char *stringa_controllo,. . .);
```

dove *pf* è un puntatore a file restituito da una chiamata a fopen(). Le funzioni fprint() e fscanf() operano sul file puntato da *pf*.

Come esempio viene presentato il seguente programma che legge dalla tastiera una stringa e un intero e li scrive sul file TEST. Il programma quindi legge il file e ne visualizza il contenuto sullo schermo. Dopo aver eseguito questo programma, si provi ad esaminare il contenuto del file TEST. Come si può vedere, contiene testo leggibile.

```
/* esempio d'uso di fscanf() e fprintf() */
#include <stdio.h>
#include <io.h>
```

```
#include <stdlib.h>

int main(void)
{
  FILE *fp;
  char s[80];
  int t;

  if((fp=fopen("test", "w")) == NULL) {
    printf("Impossibile aprire il file.\n");
    exit(1);
  }

  printf("Immettere una stringa e un numero: ");
  fscanf(stdin, "%s%d", s, &t); /* lettura dalla tastiera */

  fprintf(fp, "%s %d", s, t); /* scrittura sul file */
  fclose(fp);

  if((fp=fopen("test","r")) == NULL) {
    printf("Impossibile aprire il file.\n");
    exit(1);
  }

  fscanf(fp, "%s%d", s, &t); /* lettura dal file */
  fprintf(stdout, "%s %d", s, t); /* visualizzazione */

  return 0;
}
```

**NOTA** *Anche se fprint() e fscanf() sono normalmente il modo più facile per scrivere e leggere dati di vario genere, normalmente non sono il modo più efficiente di operare. Poiché i dati vengono scritti così come appaiono sullo schermo (e non in binario), ogni chiamata alle funzioni richiede un sovraccarico di tempo. Quindi, se si è interessati alla velocità e alle dimensioni dei file prodotti, si dovrà in genere preferire l'uso di fread() e fwrite().*

## 9.9 Gli stream standard

Quando inizia l'esecuzione di un programma C, vengono automaticamente aperti tre stream. I loro nomi sono stdin (stream di input standard), stdout (stream di output standard) e stderr (stream di errore standard). Normalmente, questi stream



fanno riferimento alla console, ma possono essere rediretti dal sistema operativo in modo da connettersi ad altri dispositivi. La redirezione delle operazioni di I/O è gestita, per fare qualche esempio, dai sistemi operativi Windows, DOS, UNIX e OS/2.

Poiché gli stream standard sono puntatori a file, essi possono essere utilizzati dal sistema di I/O C che esegue operazioni di I/O su console. Ad esempio, putchar() può essere definita nel seguente modo:

```
int putchar(char c)
{
  return putc(c, stdout);
}
```

In generale, lo stream stdin viene utilizzato per leggere dalla console, mentre stdout e stderr sono utilizzati per scrivere sulla console.

È possibile utilizzare stdin, stdout e stderr come puntatori a file in qualsiasi funzione che utilizzi una variabile di tipo FILE *. Ad esempio, è possibile usare fgets() per leggere una stringa dalla console utilizzando una chiamata come la seguente:

```
char str[255];
fgets(str, 89, stdin);
```

Infatti può essere molto utile usare fgets() in questo modo. Come si è detto in precedenza in questo capitolo, con gets() è possibile fuoriuscire dall'array utilizzato per ricevere i caratteri immessi dall'utente in quanto la funzione gets() non offre alcuna verifica del superamento dei limiti dell'array. Quando viene usata con stdin, la funzione fgets() rappresenta un'utile alternativa in quanto consente di specificare il numero di caratteri da leggere e pertanto evita il problema appena descritto. L'unico problema è il fatto che fgets() (a differenza di gets()) non rimuove il carattere di fine riga; pertanto sarà necessario rimuovere manualmente tale carattere, come illustrato nel seguente programma.

```
#include <stdio.h>
#include <string.h>

int main(void)
{
  char str[80];
  int i;

  printf("Immettere una stringa: ");
  fgets(str, 10, stdin);
```

```
  /* eliminare, se presente, il carattere newline */
  i = strlen(str)-1;
  if(str[i]=='') str[i] = '\0';

  printf("Questa è la stringa immessa: %s", str);

  return 0;
}
```

stdin, stdout e sterr non sono variabili nel senso comune del termine e non è possibile assegnare loro un valore utilizzando fopen(). Inoltre, poiché questi puntatori a file vengono creati automaticamente all'inizio del programma, vengono chiusi automaticamente al suo termine e non si dovrà quindi cercare di chiuderli.

## Collegamenti per operazioni di I/O da console

Nel Capitolo 8 si è detto che il C/C++ fa poche distinzioni fra I/O da console e I/O da file. Le funzioni di I/O da console descritte nel Capitolo 8 dirigono le proprie operazioni di I/O sugli stream stdin o stdout. In pratica, le funzioni di I/O da console sono solamente versioni speciali delle corrispondenti funzioni che operano sui file. Si tratta di funzioni diverse solo per comodità del programmatore.

Come si è detto nella sezione precedente, si possono eseguire operazioni di I/O da console utilizzando una qualsiasi delle funzioni del file system. Tuttavia, potrà sorprendere che è possibile anche eseguire operazioni di I/O su disco utilizzando le funzioni per la console come ad esempio printf()! Infatti tutte le funzioni di I/O da console operano sugli stream stdin e stdout. In ambienti che consentono la redirezione delle operazioni di I/O, questo significa che stdin e stdout possono far riferimento a un dispositivo diverso dalla tastiera e dallo schermo. Ad esempio, si consideri il seguente programma:

```
#include <stdio.h>

int main(void)
{
  char str[80];

  printf("Immettere una stringa: ");
  gets(str);
  printf(str);

  return 0;
}
```



Si supponga che questo programma si chiami TEST. Se viene eseguito normalmente, il programma visualizzerà il messaggio sullo schermo, leggerà la stringa dalla tastiera e visualizzerà tale stringa sullo schermo. Tuttavia, in un ambiente che consente la redirezione delle operazioni di I/O, è possibile redirigere sia stdin che stdout che entrambi su un file. Ad esempio, in ambiente DOS o Windows, si può eseguire TEST nel modo seguente:

```
TEST > OUTPUT
```

in questo modo, l'output di TEST verrà scritto sul file OUTPUT. Se invece si esegue TEST in questo modo:

```
TEST < INPUT > OUTPUT
```

si redirige stdin da un file chiamato INPUT e si invia l'output su un file chiamato OUTPUT.

**NOTA** *Al termine del programma C, gli stream rediretti vengono riportati al loro stato originario.*

## Uso di freopen() per redirigere gli stream standard

Per redirigere gli stream standard si può utilizzare la funzione freopen(). Questa funzione associa uno stream esistente a un nuovo file. Pertanto la si può utilizzare anche per associare uno stream standard a un altro file. Il suo prototipo è:

```
FILE *freopen(const char *nomefile,
              const char *modalità, FILE *stream);
```

dove *nomefile* è un puntatore a un file da associare allo stream puntato da *stream*. Il file viene aperto utilizzando il valore di *modalità* che corrisponde ai valori utilizzati con fopen(). In caso di successo, freopen() restituisce *stream* e in caso di insuccesso restituisce NULL.

Il seguente programma utilizza freopen() per redirigere lo stream stdout sul file OUTPUT:

```
#include <stdio.h>

int main(void)
{
  char str[80];
```

```
    freopen("OUTPUT", "w", stdout);

    printf("Immettere una stringa: ");
    gets(str);
    printf(str);

    return 0;
}
```

In generale, la redirezione degli stream standard utilizzando freopen() è utile in casi particolari, come ad esempio per il debugging. L'uso di operazioni di I/O su disco con gli stream stdin e stdout rediretti non è così efficiente come l'uso delle funzioni fread() e fwrite().

Capitolo 10

# Il preprocessore e i commenti


10.1 **Il preprocessore**
10.2 **La direttiva #define**
10.3 **La direttiva #error**
10.4 **La direttiva #include**
10.5 **Le direttive per compilazioni condizionali**
10.6 **La direttiva #undef**
10.7 **Uso di defined**
10.8 **La direttiva #line**
10.9 **La direttiva #pragma**
10.10 **Gli operatori del preprocessore # e ##**
10.11 **Le macro predefinite**
10.12 **I commenti**


el codice sorgente di un programma C/C++ è possibile includere una serie di istruzioni per il compilatore. Queste istruzioni sono chiamate *direttive per il preprocessore* e, anche se non fanno parte del linguaggio C o C++, ne espandono notevolmente le possibilità. Oltre a trattare tali direttive, questo capitolo si occuperà anche dei commenti.

## 10.1 Il preprocessore

Prima di iniziare è importante riportare il preprocessore alla sua prospettiva storica. Per quanto riguarda il linguaggio C++, il preprocessore si può considerare in larga misura un retaggio derivante dal C. Inoltre il preprocessore C++ è praticamente identico a quello definito dal C. La differenza principale fra C e C++ è l'affidamento che il linguaggio fa sul preprocessore. In C ogni direttiva del preprocessore è necessaria. In C++ alcune funzionalità sono state rese ridondanti grazie a nuovi elementi introdotti nel linguaggio. In realtà, uno degli obiettivi a lungo termine del linguaggio C++ è la completa eliminazione del preprocessore.

Ma per il momento e anche nel prossimo futuro il preprocessore continuerà ad essere ampiamente utilizzato.

Il preprocessore C accetta le seguenti direttive:

| | | | |
|---|---|---|---|
| #define | #elif | #else | #endif |
| #error | #if | #ifdef | #ifndef |
| #include | #line | #pragma | #undef |

Come si può vedere, tutte le direttive iniziano con il segno #. Inoltre, ogni direttiva deve trovarsi su una propria riga. Ad esempio, la riga seguente:

```
#include <stdio.h>  #include <stdlib.h>
```

è errata.

## 10.2 La direttiva #define

La direttiva #define definisce un identificatore e una sequenza di caratteri che verranno sostituiti all'identificatore ogni volta che questo si presenta all'interno del file sorgente. Questo identificatore è chiamato *nome della macro* e il processo di sostituzione viene chiamato *sostituzione della macro*. La forma generica della direttiva è la seguente:

#define *nome_macro sequenza_car*

Si noti l'assenza del punto e virgola al termine di questa istruzione. Fra l'identificatore e la sequenza di caratteri può essere inserito un numero arbitrario di spazi ma una volta che la sequenza di caratteri ha inizio, viene conclusa solo dal codice di fine riga.

Ad esempio, se si vuole usare la parola SINISTRA per il valore 1 e la parola DESTRA per il valore 0, si possono creare le due macro seguenti:

```
#define SINISTRA 1
#define DESTRA 0
```

In questo modo, ogni volta che il compilatore troverà nel file sorgente le parole SINISTRA o DESTRA, sostituirà i valori 1 e 0. Ad esempio, la riga seguente visualizza sullo schermo i numeri 0 1 2:

```
printf("%d %d %d", DESTRA, SINISTRA, SINISTRA+1);
```



Dopo la definizione del nome della macro, essa può essere utilizzata anche all'interno delle definizioni di altre macro. Ad esempio, le tre righe seguenti definiscono i valori di UNO, DUE e TRE:

```
#define UNO 1
#define DUE UNO+UNO
#define TRE UNO+DUE
```

La sostituzione delle macro è semplicemente la sostituzione di un identificatore con la sequenza di caratteri ad esso associata. Pertanto, per definire un messaggio di errore standard, si può procedere nel seguente modo:

```
#define E_MS "errore di input\n"

printf(E_MS);
```

Il compilatore, ogni volta che incontra l'identificatore E_MS sostituirà la stringa "errore di input\n". Quindi, per il compilatore, l'istruzione printf() avrà il seguente aspetto:

```
printf("errore di input\n");
```

Se all'interno del listato appare l'identificatore racchiuso fra virgolette, non viene eseguita alcuna sostituzione. Ad esempio,

```
#define XYZ questa è una prova

printf("XYZ");
```

non visualizza questa è una prova ma XYZ.

Se la sequenza si estende su più di una riga la si può continuare sulla riga seguente inserendo il carattere \ al termine della riga nel modo seguente:

```
#define LONG_STRING "questa è una stringa molto \
lunga utilizzata a titolo di esempio"
```

Normalmente, i programmatori C/C++ definiscono gli identificatori utilizzando lettere maiuscole. Questa convenzione aiuta nella lettura del programma in quanto si troveranno a colpo d'occhio i punti in cui avverrà la sostituzione di macro. Inoltre, è sempre bene inserire tutti i #define all'inizio del file o in un file header distinto evitando quindi di disperderli all'interno del programma.

Molto spesso le macro sono utilizzate per definire numeri chiave che appaiono in più punti di un programma. Ad esempio, se un programma definisce un

array e ha numerose routine che accedono a tale array, invece di inserire nel programma le dimensioni dell'array utilizzando una costante, si può definire la dimensione utilizzando un'istruzione #define e quindi utilizzare il nome della macro ogni volta che si deve specificare al dimensione dell'array. In questo modo, se è necessario cambiare le dimensioni dell'array, basterà modificare l'istruzione #define e ricompilare il programma. Ad esempio,

```
#define MAX_SIZE 100
/* ... */
float balance[MAX_SIZE];
/* ... */
for(i=0; i<MAX_SIZE; i++) printf("%f", balance[i]);
/* ... */
for(i=0; i<MAX_SIZE; i++) x =+ balance[i];
```

Poiché MAX_SIXE definisce le dimensioni dell'array balance se è necessario cambiare le dimensioni di balance basterà modificare la definizione di MAX_SIZE. Tutti i successivi riferimenti alla macro verranno quindi aggiornati automaticamente alla successiva ricompilazione del programma.

**NOTA** *Il linguaggio C++ fornisce un modo migliore per definire le costanti, ovvero utilizzando la parola riservata* const *che verrà descritta nella Parte seconda.*

### Macro che operano come funzioni

La direttiva #define ha però altre possibilità: il nome della macro può avere associati degli argomenti. Ogni volta che nel listato il compilatore incontra il nome della macro, gli argomenti utilizzati nella definizione della macro vengono sostituiti dagli effettivi argomenti trovati nel programma. Questa forma di macro è chiamata *macro funzione*. Ad esempio,

```
#include <stdio.h>

#define ABS(a)  (a)<0 ? -(a) : (a)

int main(void)
{
  printf("Valore assoluto di -1 e 1: %d %d", ABS(-1), ABS(1));

  return 0;
}
```



Al momento della compilazione del programma, al parametro a della definizione della macro verranno sostituiti prima il valore -1 e poi il valore 1. Le parentesi che racchiudono la a garantiscono la corretta sostituzione in ogni caso. Ad esempio, se le parentesi attorno alla a venissero rimosse, dopo la sostituzione della macro questa espressione:

```
ABS(10-20)
```

verrebbe convertita in:

```
10-20<0 ? -10-20 : 10-20
```

e ciò porterebbe a risultati errati.

L'uso di macro funzioni al posto di funzioni vere ha un vantaggio principale: aumenta la velocità di esecuzione del codice in quanto elimina il sovraccarico di tempo e memoria dovuto alla chiamata alla funzione. Tuttavia, se le dimensioni della macro funzione sono molto estese, l'aumento di velocità si paga in termini di aumento delle dimensioni del programma a causa della duplicazione del codice.

**NOTA** *Anche se le macro parametrizzate sono una funzionalità molto importante, il C++ ha un modo migliore per creare codice in linea, ovvero tramite la parola riservata* inline.

## 10.3 La direttiva #error

La direttiva #error chiede al compilatore di concludere la compilazione. Questa direttiva è utilizzata principalmente per il debugging. La forma generale della direttiva #error è la seguente:

#error *messaggio_errore*

Il *messaggio_errore* non deve essere posto fra doppi apici. Quando il compilatore incontra la direttiva #error, visualizza il messaggio di errore ad essa associato, insieme ad altre informazioni eventualmente definite dal compilatore.

## 10.4 La direttiva #include

La direttiva #include chiede al compilatore di leggere un altro file sorgente oltre a quello che contiene la direttiva #include. Il nome del file sorgente deve essere racchiuso fra doppi apici o fra parentesi angolari. Ad esempio,

```
#include "stdio.h"
#include <stdio.h>
```

chiedono al compilatore di leggere e compilare il file header delle funzioni di libreria dedicate ai file.

I file inclusi possono contenere altre direttive #include. In questo caso si parla di *include nidificati*. Il numero di livelli di nidificazione varia da compilatore a compilatore. Il C standard stabilisce che debbano essere consentiti almeno otto livelli di inclusione. Lo standard C++ raccomanda di concedere almeno 256 livelli di nidificazione.

L'inclusione del nome del file fra doppi apici o fra parentesi angolari determina il modo in cui deve essere condotta la ricerca del file specificato. Se il nome del file è racchiuso fra parentesi angolari, il file viene ricercato in un modo definito dal creatore del compilatore. Spesso, la ricerca viene eseguita in alcune speciali directory dedicate ai file di inclusione. Se il nome del file è racchiuso fra doppi apici, il file viene ricercato utilizzando un altro metodo definito dall'implementazione. Per molti compilatori, i doppi apici consentono di eseguire la ricerca all'interno della directory corrente. Se il file non viene trovato. la ricerca viene ripetuta come se il file fosse racchiuso fra parentesi angolari.

Normalmente, la maggior parte dei programmatori utilizza le parentesi angolari per includere i file header standard. L'uso dei doppi apici è normalmente riservato all'inclusione di file che hanno una relazione stretta con il programma. In ogni caso, non vi è nessuna regola che governi questo tipo di comportamenti.

Un programma C++ può utilizzare la direttiva #include anche per includere un header C++. Il linguaggio C++ definisce una serie di header standard che forniscono tutte le informazioni necessarie alle varie librerie C++. Un header è un identificatore standard che non fa riferimento necessariamente al nome di un file. Pertanto un header è semplicemente un'astrazione che garantisce che nel programma vengano incluse tutte le informazioni richieste. L'uso degli header verrà descritto nella Parte seconda.

## 10.5 Le direttive per compilazioni condizionali

Vi sono alcune direttive che consentono di compilare in modo selettivo alcune porzioni del codice sorgente di un programma. Questo processo è chiamato *compilazione condizionale* ed è ampiamente utilizzato da tutte le software house che forniscono programmi ed eseguono la manutenzione di più versioni personalizzate di un programma.



### Le direttive #if, #else, #elif e #endif

#if, #else, #elif e #endif sono le direttive di compilazione condizionale probabilmente più utilizzate. Esse consentono di includere in modo condizionale alcune porzioni di codice sulla base del risultato di un'espressione costante.

La forma generale di #if è la seguente:

```
#if espressione_costante
  sequenza istruzioni
#endif
```

Se l'espressione costante che segue la direttiva #if è vera, verrà compilato il codice che si trova fra #if e #endif. In caso contrario, tale codice viene saltato. La direttiva #endif segnala la fine di un blocco #if. Ad esempio,

```
/* Esempio d'uso di #if. */
#include <stdio.h>

#define MAX 100

int main(void)
{
#if MAX>99
  printf("compilazione per array con più di 99 elementi\n");
#endif

  return 0;
}
```

Questo programma visualizza il messaggio sullo schermo poiché MAX è maggiore di 99. L'esempio illustra un fatto molto importante. L'espressione che segue la direttiva #if viene valutata al momento della compilazione. Pertanto deve contenere solo identificatori e costanti precedentemente definiti e non è consentito l'uso di variabili.

La direttiva #else è analoga all'istruzione #else del linguaggio C++: in pratica stabilisce un'alternativa nel caso in cui non sia verificata l'espressione costante associata alla direttiva #if. L'esempio precedente può essere espanso nel seguente modo:

```
/* Esempio d'uso di #if/#else. */
#include <stdio.h>

#define MAX 10

int main(void)
{
#if MAX>99
  printf("compilazione per array con più di 99 elementi\n");
#else
  printf("compilato con un array breve\n");
#endif

  return 0;
}
```

In questo caso, MAX è minore di 99 e quindi la porzione #if del codice non viene compilata. Viene invece compilata l'alternativa indicata da #else e pertanto verrà visualizzato il messaggio compilato con un array breve.

Si noti che la direttiva #else indica sia la fine del blocco #if che l'inizio del blocco #else. Questa precisazione è necessaria in quanto vi può essere una sola direttiva #endif associata a una determinata #if.

La direttiva #elif significa "else if" e definisce una catena if-else-if che presenta più opzioni di compilazione. La direttiva #elif deve essere seguita da un'espressione costante. Se l'espressione è vera, viene compilato il blocco di codice ad essa associato e verranno saltate tutte le altre eventuali espressioni #elif. In caso contrario, viene controllata l'espressione del blocco successivo. La forma generale di #elif è la seguente:

```
#if espressione
      sequenza istruzioni
#elif espressione 1
      sequenza istruzioni
#elif espressione 2
      sequenza istruzioni
#elif espressione 3
      sequenza istruzioni
#elif espressione 4
.
.
.
#elif espressione N
      sequenza istruzioni
#endif
```



Ad esempio, il seguente frammento di codice utilizza il valore di ACTIVE_COUNTRY per definire il simbolo monetario:

```
#define FRANCIA 0
#define INGHILTERRA 1
#define ITALIA 2

#define ACTIVE_COUNTRY FRANCIA

#if ACTIVE_COUNTRY == FRANCIA
  char currency[] = "franco";
#elif ACTIVE_COUNTRY == INGHILTERRA
  char currency[] = "sterlina";
#else
  char currency[] = "lira";
#endif
```

Il C standard stabilisce che le direttive #if e #elif possano essere nidificate fino a otto livelli. Lo standard C++ suggerisce di consentire almeno 256 livelli di nidificazione. In caso di nidificazione, ogni #endif, #else e #elif si associa al più vicino #if o #elif. Ad esempio, il listato seguente è perfettamente corretto:

```
#if MAX>100
  #if SERIAL_VERSION
    int port=198;
  #elif
    int port=200;
  #endif
#else
  char out_buffer[100];
#endif
```

## Le direttive #ifdef e #ifndef

Un altro metodo per la compilazione condizionale utilizza le direttive #ifdef e #ifndef che possono essere tradotte come "se è definito" e "se non è definito". La forma generale di #ifdef è la seguente:

```
#ifdef nome_macro
  sequenza istruzioni
#endif
```

Se *nome_macro* è stata definita precedentemente in un'istruzione #define, il blocco di codice corrispondente verrà compilato.

La forma generale di #ifndef è la seguente:

#ifndef *nome_macro*
  *sequenza istruzioni*
#endif

Se nome_macro si trova attualmente non definito da un'istruzione #define, il blocco di codice corrispondente verrà compilato.

Sia #ifdef che #ifndef possono utilizzare un'istruzione #else o #elif.

Ad esempio,

```
#include <stdio.h>

#define TED 10

int main(void)
{
#ifdef TED
  printf("Ciao Ted\n");
#else
  printf("Ciao a tutti\n");
#endif
#ifndef RALPH
  printf("RALPH non definito\n");
#endif

  return 0;
}
```

visualizzerà i messaggi Ciao Ted e RALPH non definito. Se anche TED non fosse definito, il programma visualizzerebbe Ciao a tutti seguito da RALPH non definito.

Le direttive #ifdef e #ifndef possono essere nidificate in C fino a otto livelli. Lo standard C++ suggerisce di consentire almeno 256 livelli di nidificazione.

## 10.6 La direttiva #undef

La direttiva #undef elimina una definizione precedente relativa al nome della macro specificata. In pratica cancella la definizione di una macro. La forma generale di #undef è la seguente:



#undef *nome_macro*

Ad esempio,

```
#define LEN 100
#define WIDTH 100

char array[LEN][WIDTH];

#undef LEN
#undef WIDTH
/* a questo punto sia LEN che WIDTH non sono più definite */
```

Sia LEN che WIDTH rimangono definite finché non vengono incontrate le istruzioni #undef. La direttiva #undef è utilizzata principalmente per fare in modo che i nomi delle macro siano locali rispetto alla sezione di codice in cui sono richieste.

## 10.7 Uso di defined

Oltre a #ifdef, per determinare se il nome di una macro è definito, si può utilizzare la direttiva #if insieme all'operatore di compilazione defined. La forma generale dell'operatore defined è la seguente:

defined *nome-macro*

Se *nome-macro* è attualmente definita, l'espressione è vera. In caso contrario l'espressione è falsa. Ad esempio, per determinare se la macro MYFILE è definita, si possono usare le due direttive seguenti:

```
#if defined MYFILE
```

o

```
#ifdef MYFILE
```

Per invertire la condizione, basta far precedere alla parola defined il punto esclamativo (!). Ad esempio, il seguente frammento di codice viene compilato solo se DEBUG non è definita.

```
#if !defined DEBUG
  printf("Versione finale!\n");
#endif
```

Un motivo che consiglia di usare defined rispetto a #ifdef è la possibilità di determinare l'esistenza di un nome di macro all'interno di un'istruzione #elif.

## 10.8 La direttiva #line

La direttiva #line consente di alterare il contenuto di __LINE__ e __FILE__ che sono identificatori predefiniti del compilatore. L'identificatore __LINE__ contiene il numero di riga corrente nel codice compilato. L'identificatore __FILE__ è una stringa che contiene il nome del file sorgente compilato. La forma generale di #line è la seguente:

#line *numero "nomefile"*

dove *numero* è un numero positivo intero e diverrà il nuovo valore di __LINE__ e il parametro opzionale *nomefile* è un qualunque identificatore valido di file che diverrà il nuovo valore di __FILE__. La direttiva #line è utilizzata principalmente per scopi di debugging e per particolari applicazioni.

Ad esempio, il seguente codice specifica che il conteggio delle righe deve ripartire dal numero 100 e quindi l'istruzione printf() visualizzerà il numero 102 in quanto è la terza riga nel programma sorgente dopo l'istruzione #line 100.

```
#include <stdio.h>

#line 100                       /* reinizializza il contatore di riga */

int main(void)                  /* riga 100 */
{                               /* riga 101 */
  printf("%d\n",_ _LINE_ _);    /* riga 102 */

  return 0;
}
```

## 10.9 La direttiva #pragma

#pragma è una direttiva specifica dell'implementazione che consente l'invio di vari tipi di informazioni al compilatore. Ad esempio, un compilatore può prevedere un'opzione che consenta di attivare l'opzione di Trace sull'esecuzione del programma. Molto probabilmente questa opzione potrà essere specificata con un'istruzione #pragma. Per informazioni sulle opzioni disponibili è necessario consultare la documentazione del compilatore.



## 10.10 Gli operatori del preprocessore # e ##

Il preprocessore prevede due operatori: # e ##. Questi operatori possono essere utilizzati con l'istruzione #define.

L'operatore #, chiamato anche operatore di *conversione in stringa*, tramuta l'argomento seguente in una stringa fra doppi apici. Ad esempio, si consideri il programma seguente:

```
#include <stdio.h>

#define mkstr(s)  # s

int main(void)
{
  printf(mkstr(Il C++ è bello));

  return 0;
}
```

Il preprocessore C trasforma la riga

```
printf(mkstr(Il C++ è bello));
```

in

```
printf("Il C++ è bello");
```

L'operatore ## è chiamato anche operatore di *concatenamento*. Ad esempio:

```
#include <stdio.h>

#define concat(a, b)  a ## b

int main(void)
{
  int xy = 10;
```

```
  printf("%d", concat(x, y));

  return 0;
}
```

Il preprocessore trasforma

```
printf("%d", concat(x, y));
```

in

```
printf("%d", xy);
```

Se il funzionamento di questi operatori può sembrare un po' curioso, si tenga a mente che non sono necessari e che in genere non vengono impiegati. La loro esistenza consente al preprocessore di gestire casi particolari.

## 10.11 Le macro predefinite

Il C++ specifica sei nomi di macro predefinite:

__LINE__
__FILE__
__DATE__
__TIME__
__STDC__
__cplusplus

Il C ne definisce solo cinque. Le macro __LINE__ e __FILE__ sono state discusse precedentemente nella sezione che riguardava la direttiva #line. In breve esse contengono rispettivamente il numero di riga e il nome del file sottoposto a compilazione.

La macro __DATE__ contiene una stringa nel formato *mese/giorno/anno*. Questa stringa rappresenta la data della traduzione del codice sorgente in codice oggetto.

La macro __TIME__ contiene una stringa che riporta l'ora della traduzione del codice sorgente in codice oggetto. La forma di questa stringa è: *ore:minuti:secondi*.

Il significato della macro __STDC__ è definito dall'implementazione. In genere se __STDC__ è definita, il compilatore accetterà unicamente codice C/C++ standard, rifiutando le estensioni non standard.



Lo standard C++ aggiunge alle macro precedenti una nuova macro chiamata __cplusplus, che contiene almeno sei cifre. I compilatori non standard usano cinque o meno cifre.

## 10.12 I commenti

In C, tutti i commenti iniziano con la coppia di caratteri /* e terminano con */. Fra l'asterisco e la barra non devono essere inseriti spazi. Tutto ciò che si trova fra questi simboli di apertura e di chiusura verrà ignorato dal compilatore. Ad esempio, questo programma visualizza sullo schermo solo la parola ciao:

```
#include <stdio.h>

int main(void)
{
  printf("ciao");
  /* printf("a tutti"); */

  return 0;
}
```

I commenti C sono chiamati anche commenti multiriga in quanto possono anche estendersi su più righe, come nel seguente esempio:

```
/* Questo è
un commento
su più righe */
```

I commenti possono essere inseriti in qualunque punto di un programma, sempre che non appaiano all'interno di una parola chiave o di un identificatore. Quindi, questo commento è valido:

```
x = 10+ /* somma dei numeri */5;
```

mentre

```
swi/*non funziona*/tch(c) { ...
```

è errato poiché una parola chiave non può contenere un commento. Tuttavia, è sconsigliabile inserire commenti all'interno di espressioni in quanto ne confondono il significato. Non è possibile nidificare i commenti C: quindi un commento

non può contenere un altro commento. Ad esempio, questo frammento di codice provoca un errore di compilazione:

```
/* questo è un commento esterno
  x = y/a;
/* questo è un commento interno e provoca un errore */
*/
```

Al momento attuale, lo Standard C definisce solo lo stile di commenti appena descritto. Al contrario il linguaggio C++ supporta due tipi di commenti. Il primo è il commento multiriga C. Il secondo è il commento su una sola riga. I commenti su una sola riga iniziano con la sequenza // e terminano alla fine della riga. Ad esempio:

```
// Questo è un commento su una sola riga
```

Anche se lo Standard C attualmente non definisce questo stile di commenti, in realtà tale stile è accettato dalla maggior parte dei compilatori e probabilmente in futuro verrà incorporato ufficialmente nello Standard C. L'argomento dei commenti su una sola riga verrà ulteriormente sviluppato nella Parte seconda.

È buona norma utilizzare sempre i commenti per descrivere il funzionamento del codice. Tutte le funzioni di complessità non elementare, dovranno prevedere un commento all'inizio che indichi lo scopo della funzione, il modo in cui deve essere chiamata e il valore da essa restituito.

Parte seconda

# IL LINGUAGGIO C++

La Parte prima ha esaminato il sottoinsieme C del linguaggio C++. La Parte seconda si occupa delle funzionalità specifiche del C++, ovvero di quelle funzionalità del C++ che non sono presenti nel linguaggio C. Poiché la maggior parte delle estensioni che il C++ apporta al C sono dedicate al supporto della programmazione a oggetti (OOP), la seconda parte fornisce anche una discussione sulla teoria e i vantaggi di questa tecnica di programmazione.

Capitolo 11

# Panoramica del linguaggio C++


11.1 **Le origini del C++**
11.2 **Che cos'è la programmazione a oggetti**
11.3 **Elementi di base del linguaggio C++**
11.4 **C++ vecchio stile e C++ moderno**
11.5 **Introduzione alle classi C++**
11.6 **L'overloading delle funzioni**
11.7 **L'overloading degli operatori**
11.8 **L'ereditarietà**
11.9 **I costruttori e i distruttori**
11.10 **Le parole riservate del C++**
11.11 **La forma generale di un programma C++**


Questo capitolo presenta una panoramica dei concetti di base che hanno condotto allo sviluppo del C++. Il C++ è un linguaggio di programmazione a oggetti le cui funzionalità sono strettamente correlate fra loro. In molti casi, questa correlazione rende difficile descrivere una funzionalità del C++ senza menzionarne nel contempo anche altre. In molte situazioni, le funzionalità a oggetti del C++ sono così correlate fra loro che per trattare una funzionalità è necessario che il lettore sia a conoscenza di una o più funzionalità correlate. Per risolvere questo problema, questo capitolo presenta una rapida panoramica degli aspetti più importanti del C++, la sua storia, le sue funzionalità principali e le differenze esistenti fra il C++ tradizionale e quello definito dallo standard. I capitoli successivi di questa parte della guida esaminano più in dettaglio il C++.

## 11.1 Le origini del C++

Il linguaggio C++ nacque come estensione del C. Le estensioni del C++ sono state inizialmente sviluppate da Bjarne Stroustrup nel 1979 presso i laboratori Bell di Murray Hill nel New Jersey. Inizialmente il nuovo linguaggio fu chiamato semplicemente "C con classi". Nel 1983 questo nome venne cambiato in C++.

Anché se il linguaggio C è stato uno dei linguaggi di programmazione professionali più apprezzati e ampiamente utilizzati al mondo, l'invenzione del C++ fu dettata dalla necessità di raggiungere maggiori livelli di complessità. Nel trascorrere degli anni, i programmi per computer sono diventati sempre più estesi e complessi. Anche se il C è un linguaggio di programmazione eccellente, anch'esso ha i propri limiti. In C, quando un programma supera le 25.000 o le 100.000 righe di codice, diviene così complesso che risulta difficile considerarlo nella sua totalità. Il C++ consente di superare questa barriera. L'essenza del C++ è stata quindi concepita con lo scopo di permettere ai programmatori di comprendere e gestire programmi più estesi e complessi.

La maggior parte delle funzionalità aggiunte da Stroustrup al C consente il supporto della programmazione a oggetti, chiamata anche OOP (per una breve descrizione della programmazione a oggetti, si consulti la prossima sezione di questo capitolo). Stroustrup asserisce che alcune delle funzionalità a oggetti del C++ sono state ispirate da un altro linguaggio di programmazione a oggetti il Simula67. Pertanto, il C++ rappresenta il punto di unione fra due dei metodi di programmazione più potenti.

Da quando è stato inventato, il linguaggio C++ è stato sottoposto a tre grandi revisioni, ognuna delle quali ha apportato aggiunte e modifiche al linguaggio. La prima revisione si è svolta nel 1985 e la seconda nel 1998. La terza si è verificata durante la fase di standardizzazione del C++. Il lavoro per la standardizzazione del linguaggio C++ è iniziato molti anni fa. A quell'epoca è stato creato un comitato congiunto fra ANSI (American National Standards Institute) e ISO (International Standards Organization). La prima bozza di standard nacque il 25 gennaio 1994. In tale bozza, il comitato di standardizzazione C++ ANSI/ISO (di cui l'autore è membro) ha mantenuto le funzionalità inizialmente definite da Stroustrup e ve ne ha aggiunte di nuove. In generale la bozza iniziale rifletteva lo stato del linguaggio C++ a quel tempo.

Poco dopo il completamento della prima bozza dello standard, si è verificato un evento che ha provocato una grande espansione del linguaggio: la creazione della libreria STL (Standard Template Library) da parte di Alexander Stepanov. La libreria STL è costituita da una serie di routine generiche per la manipolazione dei dati. Si tratta di un oggetto potente ed elegante ma anche piuttosto esteso. Successivamente alla prima bozza, il comitato ha deciso di includere la libreria STL nelle specifiche del linguaggio C++. L'aggiunta della libreria STL ha esteso notevolmente le capacità del linguaggio, molto oltre la definizione originale e l'inclusione della libreria STL ha rallentato la standardizzazione del linguaggio.

Dunque si può dire che la standardizzazione del linguaggio C++ ha richiesto molto più tempo di quanto chiunque potesse attendersi. Nel frattempo sono state apportate alcune aggiunte e molte piccole modifiche al linguaggio. In pratica la versione di C++ definita dal comitato di standardizzazione è molto più estesa e complessa del progetto originale di Stroustrup, ma ora finalmente è pronto lo standard. La bozza finale è stata prodotta il 14 novembre 1997 e finalmente lo



standard per il C++ è una realtà. Il materiale contenuto in questo volume descrive lo Standard per il linguaggio C++ , comprendendo tutte le funzionalità più recenti. Questa è la versione del linguaggio C++ creata dal comitato di standardizzazione ANSI/ISO e accettata da tutti i più importanti compilatori.

## 11.2 Che cos'è la programmazione a oggetti

Poiché la *programmazione a oggetti* (OOP) ha dato origine al C++, è necessario comprendere i suoi principi fondamentali. La programmazione a oggetti rappresenta un nuovo e potente metodo di programmazione. Le metodologie di programmazione sono cambiate notevolmente dall'invenzione del computer, soprattutto per consentire di aumentare la complessità dei programmi. Ad esempio, quando furono inventati i computer, la programmazione veniva eseguita impostando istruzioni binarie tramite il pannello frontale del computer. Finché i programmi erano composti da poche centinaia di istruzioni, questo approccio ha funzionato. Con la crescita dei programmi è stato sviluppato il linguaggio Assembler con il quale un programmatore poteva realizzare programmi più estesi e complessi, utilizzando rappresentazioni simboliche delle istruzioni in linguaggio macchina. Ma i programmi continuavano a crescere e furono perciò introdotti linguaggi di alto livello che davano al programmatore più strumenti per gestire questa nuova richiesta di complessità. Il primo linguaggio di questo genere fu naturalmente il FORTRAN. Anche se il FORTRAN è stato un notevole passo in avanti rispetto al passato, si trattava di un linguaggio che non incoraggiava la realizzazione di programmi chiari e facili da comprendere.

Il 1960 ha dato i natali alla programmazione strutturata. Questo è il metodo seguito da linguaggi come il C e il Pascal. L'impiego di linguaggi strutturati ha reso possibile la realizzazione di programmi piuttosto complessi con una discreta facilità. I linguaggi strutturati sono caratterizzati dal supporto di subroutine indipendenti, variabili locali, costrutti di controllo avanzati e dal fatto di non impiegare GOTO. Tuttavia, anche utilizzando metodi di programmazione strutturata, un progetto può diventare incontrollabile una volta che raggiunga determinate dimensioni.

Si consideri questo fatto: ad ogni punto di svolta nel campo della programmazione, sono stati creati strumenti e tecniche che consentivano al programmatore di realizzare programmi più complessi. Ogni passo in questo percorso consisteva nell'utilizzo dei migliori elementi dei metodi precedenti e nel loro sviluppo. Prima dell'invenzione della programmazione a oggetti, molti progetti raggiungevano o superavano il punto in cui l'approccio strutturato non può più essere adottato. La programmazione a oggetti è nata con lo scopo di superare questa barriera.

La programmazione a oggetti ha preso le migliori idee della programmazione strutturata e le ha combinate con nuovi concetti. Il risultato è un'organizzazione

completamente nuova dei programmi. In generale un programma può essere realizzato in due modi: ponendo al centro il codice ("ciò che accade") o ponendo al centro i dati ("gli attori interessati"). Utilizzando le tecniche della programmazione strutturata, i programmi vengono tipicamente organizzati attorno al codice. Questo approccio prevede che il codice operi sui dati. Ad esempio, un programma scritto con un linguaggio di programmazione strutturato come il C è definito dalle sue funzioni, le quali operano sui dati usati dal programma. I programmi a oggetti seguono l'altro approccio. Infatti sono organizzati attorno ai dati e si basano sul fatto che sono i dati a controllare l'accesso al codice. In un linguaggio a oggetti si definiscono i dati e le routine che sono autorizzate ad agire su tali dati. Pertanto sono i dati a stabilire quali sono le operazioni che possono essere eseguite. I linguaggi che consentono di attuare i principi della programmazione a oggetti hanno tre fattori in comune: l'incapsulamento, il polimorfismo e l'ereditarietà.

## L'incapsulamento

L'*incapsulamento* è il meccanismo che riunisce insieme il codice e i dati da esso manipolati e che mette entrambi al sicuro da interferenze o errati utilizzi. In un linguaggio a oggetti, il codice e i dati possono essere raggruppati in modo da creare una sorta di "scatola nera". Quando il codice e i dati vengono raggruppati in questo modo, si crea un oggetto. In altre parole, un oggetto è un "dispositivo" che supporta l'incapsulamento.

All'interno di un oggetto, il codice, i dati o entrambi possono essere privati di tale oggetto oppure pubblici. Il codice o i dati privati sono noti e accessibili solo da parte degli elementi dell'oggetto stesso. Questo significa che il codice e i dati privati non risultano accessibili da parte di elementi del programma che si trovano all'esterno dell'oggetto. Se il codice o i dati sono pubblici, risulteranno accessibili anche da altre parti del programma che non sono definite all'interno dell'oggetto. Generalmente le parti pubbliche di un oggetto sono utilizzate per fornire un'interfaccia controllata agli elementi privati dell'oggetto stesso.

Un oggetto è in tutto e per tutto una variabile di un tipo definito dall'utente. Può sembrare strano pensare a un oggetto, che contiene codice e dati, come a una variabile. Tuttavia nella programmazione a oggetti, avviene proprio questo. Ogni volta che si definisce un nuovo tipo di oggetto, si crea implicitamente un nuovo tipo di dati. Ogni specifica istanza di questo tipo è una variabile composta.

## Il polimorfismo

I linguaggi di programmazione a oggetti supportano il *polimorfismo* che è caratterizzato dalla frase "un'interfaccia, più metodi". In altri termini, il polimorfismo consente a un'interfaccia di controllare l'accesso a una classe generale di azioni.



La specifica azione selezionata è determinata dalla natura della situazione. Un esempio di polimorfismo tratto dal mondo reale è il termostato. Non importa il tipo di combustibile utilizzato (gas, petrolio, elettricità e così via): il termostato funziona sempre nello stesso modo. In questo caso, il termostato (che è l'interfaccia) è sempre lo stesso qualsiasi sia il tipo di combustibile (metodo) utilizzato. Ad esempio, se si desidera raggiungere una temperatura di 20 gradi, si imposta il termostato a 20 gradi. Non importa quale sia il tipo di combustibile che fornisce il calore.

Questo stesso principio si può applicare anche in programmazione. Ad esempio, un programma potrebbe definire tre diversi tipi di stack. Uno stack per i valori interi, uno per i caratteri e uno per valori in virgola mobile. Grazie al polimorfismo, sarà possibile creare un solo insieme di nomi (push() e pop()) utilizzabile per i tre tipi di stack. Nel programma verranno create tre diverse versioni di queste funzioni, una per ogni tipo di stack, ma il nome delle funzioni rimarrà lo stesso. Il compilatore selezionerà automaticamente la funzione corretta sulla base del tipo dei dati memorizzati. Pertanto, l'interfaccia dello stack (ovvero le funzioni push() e pop()) non cambierà indipendentemente dal tipo di stack utilizzato. Naturalmente le singole versioni di queste funzioni definiscono implementazioni (metodi) specifiche per ciascun tipo di dati.

Il polimorfismo aiuta a ridurre la complessità del programma consentendo di utilizzare la stessa interfaccia per accedere a una classe generale di azioni. Sarà compito del compilatore selezionare l'*azione specifica* (ovvero il metodo) da applicare in una determinata situazione. Il programmatore non dovrà più fare questa selezione manualmente, ma dovrà semplicemente ricordare e utilizzare l'interfaccia generale.

I primi linguaggi di programmazione a oggetti erano interpretati e quindi il polimorfismo era, per forza di cose, supportato al momento dell'esecuzione (runtime). Ma il C++ è un linguaggio compilato pertanto il polimorfismo è supportato al momento dell'esecuzione e al momento della compilazione (compile-time).

## L'ereditarietà

L'*ereditarietà* è il processo grazie al quale un oggetto acquisisce le proprietà di un altro oggetto. Questo è un concetto fondamentale poiché chiama in causa il concetto di *classificazione*. Se si prova a riflettere, la maggior parte della conoscenza è resa più gestibile da classificazioni gerarchiche. Ad esempio, una mela rossa Delicious appartiene alla classificazione *mela* che a sua volta appartiene alla classe *frutta* che a sua volta si trova nella classe più estesa *cibo*. Senza l'uso della classificazione, ogni oggetto dovrebbe essere definito esplicitamente con tutte le proprie caratteristiche. L'uso della classificazione consente di definire un oggetto sulla base delle qualità che lo rendono unico all'interno della propria classe. Sarà il meccanismo di ereditarietà a rendere possibile per un oggetto di essere una

specifica istanza di un caso più generale. Come si vedrà, l'ereditarietà è un importante aspetto della programmazione a oggetti.

## 11.3 Elementi di base del linguaggio C++

Nella Parte prima, è stato descritto il sottoinsieme C del linguaggio C++ e sono stati presentati alcuni programmi C che avevano lo scopo di illustrare queste funzionalità. Da qui in avanti, tutti gli esempi saranno programmi C++. Questo significa che tali programmi faranno uso delle funzionalità specifiche del linguaggio C++. Per semplificare la discussione, d'ora in poi si farà riferimento alle funzionalità specifiche del linguaggio C++ parlando di "funzionalità C++". Chi avesse esperienza di programmazione in C o chi abbia studiato i programmi del sottoinsieme C contenuti nella Parte prima, noterà che i programmi C++ differiscono dai programmi C per alcuni aspetti importanti. La maggior parte delle differenze riguarda l'utilizzo delle funzionalità a oggetti tipiche del linguaggio C++. Ma i programmi C++ differiscono dai programmi C in molti altri sensi, ad esempio nel modo in cui vengono eseguite le operazioni di I/O e nelle operazioni di inclusione dei file header. Inoltre la maggior parte dei programmi C++ ha una serie di tratti il comune che li identificano chiaramente come tali. Prima di affrontare l'argomento dei costrutti a oggetti del linguaggio C++, è opportuno conoscere gli elementi fondamentali di un programma C++.
Questa sezione descrive vari elementi riferiti a quasi tutti i programmi C++. Nel frattempo verranno evidenziate alcune delle differenze più importanti fra il C e le prime versioni di C++.

### Un programma C++ di esempio

Si può partire con il semplice programma C++ presentato di seguito.

```
#include <iostream>
using namespace std;

int main()
{
  int i;

  cout << "Stringa di output.\n";  // commento su una sola riga
  /* si può utilizzare anche lo stile di commenti C */

  // input di un numero con l'operatore >>
  cout << "Immettere un numero: ";
  cin >> i;

  // output di un numero con l'operatore <<
  cout << i << " al quadrato è uguale a " << i*i << "\n";

  return 0;
}
```



Come si può vedere, questo programma ha un aspetto molto diverso dai programmi C presentati nella Parte prima. Può essere utile commentarlo riga per riga. Per iniziare, viene incluso l'header <iostream>. Questo file è utilizzato per consentire l'esecuzione di operazioni di I/O in stile C++ (<iostream> è per il C++ ciò che stdio.h è per il C). Si può anche notare che il nome iostream non ha l'estensione .h. Questo è dovuto al fatto che iostream è un header definito dallo Standard C++. I nuovi header non usano l'estensione .h.

La riga successiva del programma è:

```
using namespace std;
```

Questa riga chiede al compilatore di utilizzare il namespace std. I namespace sono una funzionalità che è stata aggiunta solo recentemente al C++. Un namespace crea una regione di dichiarazione in cui possono essere inseriti vari elementi del programma. Il namespace aiuta a organizzare meglio i programmi più estesi. L'istruzione using informa il compilatore che si vuole utilizzare il namespace std. Questo è il namespace in cui è dichiarata l'intera libreria standard C++. Dunque utilizzando il namespace std si semplifica l'accesso alla libreria standard. I programmi della Parte prima che utilizzavano solo il sottoinsieme C non avevano bisogno dell'istruzione namespace poiché le funzioni della libreria C sono disponibili anche nel namespace globale.

**NOTA** *Poiché gli header di questo tipo e i namespace sono funzionalità aggiunte recentemente al linguaggio C++, capiterà con facilità di trovare vecchio codice che non le impiega. Inoltre i compilatori non recenti non prevedono il supporto di queste funzionalità. Più avanti in questo stesso capitolo si potranno trovare informazioni utili per impiegare vecchi compilatori.*

Ora si esamini la seguente riga.

```
int main()
```

Si noti che l'elenco dei parametri di main() è vuoto. In C++, questo significa che main() non ha parametri. In C, una funzione che non ha parametri deve specificare come elenco di parametri la dichiarazione void.

```
int main(void)
```

Questo era il modo in cui main() veniva dichiarata nei programmi della Parte prima. In C++, l'uso di void è ridondante e inutile. Come regola generale, in C++ quando una funzione non ha parametri, basterà che il suo elenco di parametri sia vuoto senza la necessità di utilizzare la parola chiave void.

La riga successiva contiene due nuove funzionalità del C++:

```
cout << "Stringa di output.\n";  // commento su una sola riga
```

Questa riga introduce due nuove funzionalità del C++. Innanzi tutto, l'istruzione:

```
cout << "Stringa di output.\n";
```

provoca la visualizzazione sullo schermo della frase Stringa di output seguita dalla combinazione Carriage Return - Line Feed. In C++, l'operatore << assume nuovi significati. Continua a fungere da operatore di scorrimento a sinistra ma quando viene utilizzato nel modo illustrato dall'esempio, assume il significato di *operatore di output*. La parola cout è un identificatore che fa riferimento allo schermo (in realtà anche il C++ come il C supporta la redirezione delle operazioni di I/O, ma per quanto riguarda questa discussione, si suppone che cout faccia riferimento solo allo schermo). Si può utilizzare cout e l'operatore << per visualizzare ogni genere di dati predefiniti come pure stringhe di caratteri.

In C++ è comunque possibile utilizzare printf() o una qualsiasi delle altre funzioni di I/O del C. Tuttavia la maggior parte dei programmatori trova che l'utilizzo di << sia più nello spirito del C++. Inoltre, anche se la visualizzazione di una stringa con printf() è praticamente equivalente all'utilizzo di <<, il sistema di I/O del C++ può essere espanso in modo da eseguire operazioni sugli oggetti definiti dall'utente (un'operazione non eseguibile utilizzando printf()).

Quello che segue l'espressione da visualizzare in output è un commento C++ su una sola riga. Come si è detto nel Capitolo 10, in C++ i commenti possono essere definiti in due modi. Si può utilizzare un commento C, che funziona nello stesso modo anche in C++. Ma in C++ è anche possibile definire un *commento su una singola riga* utilizzando la coppia di caratteri //; ciò che segue viene ignorato dal compilatore fino alla fine della riga. In generale, i programmatori C++ utilizzano commenti C per creare commenti multi riga e commenti C++ quando devono inserire un commento formato da un'unica riga.

Quindi, il programma chiede all'utente un numero. Il numero viene letto dalla tastiera dalla seguente istruzione:

```
cin >> i;
```



In C++, l'operatore >> continua a eseguire l'operazione di scorrimento a destra ma quando viene utilizzato in questo modo, assume il significato di *operatore di input*. Questa istruzione assegna a i il valore letto dalla tastiera. L'identificatore cin fa riferimento al dispositivo di input standard che è normalmente la tastiera. In generale, si utilizza cin >> per assegnare un valore a una variabile di uno qualsiasi dei tipi di base più le stringhe.

**NOTA** *La riga di codice appena descritta è stampata correttamente. In particolare, non si deve inserire il carattere & davanti alla i. Quando si esegue l'input di informazioni utilizzando una funzione C come scanf(), è necessario passare alla funzione un puntatore alla variabile che riceverà le informazioni. Questo significa che la variabile deve essere preceduta dall'operatore "indirizzo di" ovvero &. Ma, grazie al modo in cui l'operatore >> è implementato in C++, questo non è necessario. Il motivo verrà descritto nel Capitolo 13.*

Anche se questo non viene illustrato dall'esempio, rimane comunque possibile utilizzare funzioni di input C, come ad esempio scanf() al posto di cin >>. Tuttavia, come si è detto nel caso di cout, la maggior parte dei programmatori trova che cin >> sia più nello spirito del C++.

Di seguito viene presentata un'altra riga molto interessante del programma:

```
cout << i << " al quadrato è uguale a " << i*i << "\n";
```

Se si suppone che il valore di i sia 10, questa istruzione visualizza la frase 10 al quadrato è uguale a 100, seguito dalla combinazione Carriage Return-Line Feed. Questa riga dimostra che è possibile utilizzare di seguito più operazioni di output <<.

Il programma termina con l'istruzione:

```
return 0;
```

Questa riga fa in modo che al processo chiamante (normalmente il sistema operativo) venga restituito il valore zero. Questa riga ha lo stesso significato già visto per il C. La restituzione del valore zero indica che il programma è terminato normalmente. Una terminazione anormale del programma dovrà essere segnalata restituendo un valore diverso da zero. In alternativa si possono usare i valori EXIT_SUCCESS e EXIT_FAILURE.

## Il funzionamento degli operatori di I/O

Come si è detto, quando vengono utilizzati per operazioni di I/O, gli operatori << e >> sono in grado di gestire qualsiasi tipo di dati predefinito del C++. Ad esem-

pio, questo programma legge in input un valore float, un double e una stringa e poi li visualizza.

```
#include <iostream>
using namespace std;

int main()
{
  float f;
  char str[80];
  double d;

  cout << "Immettere due numeri in virgola mobile: ";
  cin >> f >> d;

  cout << "Immettere una stringa: ";
  cin >> str;

  cout << f << " " << d << " " << str;

  return 0;
}
```

Quando si esegue questo programma, si provi a immettere come stringa la frase Questa è una prova. Quando il programma visualizzerà le informazioni immesse, presenterà la sola parola "Questa". La parte rimanente della stringa non verrà visualizzata poiché l'operatore >> termina la lettura della stringa nel momento in cui incontra il primo spazio. Pertanto, il resto della frase, ovvero "è una prova", non verrà mai letto dal programma. Questo programma illustra inoltre la possibilità di inserire più operazioni di input in un'unica istruzione.

Gli operatori di I/O del C++ riconoscono tutte le costanti descritte nel Capitolo 2. Ad esempio è perfettamente lecito scrivere:

```
cout << "A\tB\tC";
```

Questa istruzione produce in output le lettere A, B e C separate da uno spazio di tabulazione.

## La dichiarazione di variabili locali

Chi proviene da un'esperienza di programmazione in C++ deve conoscere un'altra importante differenza fra il codice C e C++, ovvero la posizione in cui è possibile dichiarare le variabili locali. In C, tutte le variabili locali utilizzate all'interno



di un blocco devono essere dichiarate all'inizio di tale blocco. Quindi non è possibile dichiarare una variabile in un blocco dopo un'istruzione di "azione". Ad esempio, in C, il seguente frammento di codice è errato:

```
/* Errato in C. Accettato in C++. */
int f()
{
  int i;
  i = 10;

  int j;  /* istruzione non compilabile in C */
  j = i*2;

  return j;
}
```

Poiché la dichiarazione di j è preceduta da un'istruzione che esegue un'azione, il compilatore C individuerà un errore e si rifiuterà di compilare la funzione. In C++ invece questo frammento di codice è perfettamente corretto e verrà compilato senza alcun errore. In C++ le variabili locali possono essere dichiarate in qualsiasi punto di un blocco e non solo all'inizio.

Ecco un'altra versione del programma contenuto nella sezione precedente, in cui ogni variabile viene dichiarata nel momento in cui vi è l'effettiva necessità.

```
#include <iostream>
using namespace std;

int main()
{
  float f;
  double d;
  cout << "Immettere due numeri in virgola mobile: ";
  cin >> f >> d;

  cout << "Immettere una stringa: ";
  char str[80];  // str viene dichiarata appena prima del suo uso
  cin >> str;

  cout << f << " " << d << " " << str;

  return 0;
}
```

La posizione in cui si dichiarano le variabili dipende quindi dal programmatore. Poiché molta della teoria del C++ è legata all'incapsulamento di codice e dati,

ha senso dichiarare le variabili il più possibile vicino al luogo in cui vengono impiegate piuttosto che all'inizio del blocco. Nell'esempio precedente, le dichiarazioni sono state separate solo per semplificarne l'individuazione. È facile però immaginare esempi più calzanti in cui questa caratteristica del C++ risulta molto più importante.

La dichiarazione di variabili vicino al luogo in cui verranno utilizzate aiuta a evitare effetti collaterali indesiderati. In ogni caso, i maggiori benefici derivanti dalla dichiarazione delle variabili nel luogo in cui vengono usate si ottengono nelle funzioni più estese. Francamente, nelle funzioni più brevi (come molti degli esempi presenti in questa guida) non vi è alcun motivo per dichiarare le variabili in un luogo diverso dall'inizio della funzione. Per questo motivo, in questa guida le variabili verranno dichiarate nel luogo in cui vengono utilizzate per la prima volta solo quando ciò è giustificato dalle dimensioni o dalla complessità di una funzione.

Vi è un acceso dibattito sul luogo in cui sia più saggio localizzare la dichiarazione delle variabili. Alcuni sostengono che spargendo le dichiarazioni all'interno di un blocco, si complica e non si semplifica la lettura del codice poiché è più difficile trovare rapidamente le dichiarazioni di tutte le variabili utilizzate in tale blocco, complicando inutilmente la manutenzione del programma. Per questo motivo, alcuni programmatori C++ non sfruttano questa caratteristica. Questa guida non intende schierarsi in questo dibattito. Tuttavia, è da considerare che quando viene applicata correttamente, specialmente nelle funzioni più estese, la dichiarazione delle variabili nel punto in cui vengono utilizzate per la prima volta può aiutare nella realizzazione di programmi esenti da bug.

### Trasformazione automatica in int

Il linguaggio C++ è stato recentemente sottoposto a una modifica che può influenzare il vecchio codice C++ e anche la conversione del codice C in C++. Il linguaggio C e le specifiche originali del linguaggio C++ stabilivano che quando in una dichiarazione non era indicato esplicitamente un tipo, doveva essere impiegato il tipo int. Questa regola è stata eliminata dal C++ un paio di anni fa nella fase di standardizzazione. Probabilmente anche il prossimo standard del linguaggio C eliminerà questa regola che tuttavia è attualmente in uso e viene impiegata da una grande quantità di programmi. Questa regola è stata anche impiegata nel software C++ meno recente.

L'uso più comune della regola di trasformazione in int riguarda il tipo restituito dalle funzioni. Era infatti pratica comune non specificare esplicitamente il tipo int quando la funzione restituiva un risultato intero. Ad esempio, in C e nelle vecchie versioni di C++, la seguente funzione sarebbe stata valida:



```
func(int i)
{
  return i*i;
}
```

In C++ standard in questa funzione si deve specificare esplicitamente il tipo int.

```
int func(int i)
{
  return i*i;
}
```

In pratica, quasi tutti i compilatori C++ supportano ancora la regola della trasformazione in int per motivi di compatibilità con il codice meno recente. Tuttavia si deve evitare di utilizzare questo automatismo nel nuovo codice poiché in futuro tale trasformazione automatica non sarà più consentita.

### Il tipo di dati bool

Il linguaggio C++ definisce un tipo booleano chiamato bool. Al momento attuale, lo standard per il C non prevede questo tipo. Gli oggetti di tipo bool possono contenere i soli valori true e false che divengono parole riservate del linguaggio C++. Come si è detto nella Parte prima, il linguaggio esegue delle conversioni automatiche che consentono di convertire i valori bool in valori interi e viceversa. In particolare, ogni valore diverso da 0 viene convertito in true e 0 viene convertito in false. Si verifica anche la situazione opposta: true viene convertito in 1 e false viene convertito in 0. Pertanto, permane il concetto generale che prevede che 0 equivalga a false e che un valore diverso da 0 equivalga a true.

## 11.4 C++ vecchio stile e C++ moderno

Come si è detto, il linguaggio C++ è stato sottoposto a un processo evolutivo piuttosto intenso durante la fase di sviluppo e standardizzazione. Questo ha portato all'esistenza di due versioni di C++. La prima è la versione tradizionale che si basa sul progetto originale di Bjarne Stroustrup. Questa è la versione di C++ che veniva utilizzata dai programmatori nel decennio scorso. La versione più recente, il C++ standard, è stata creata da Stroustrup e dal comitato di standardizzazione ANSI/ISO. Anche se queste due versioni di C++ sono molto simili, il C++ standard contiene numerose estensioni che sono assenti nella versione precedente. Pertanto il C++ standard rappresenta un "sovrainsieme" del C++ tradizionale.

Questo volume si occupa del linguaggio C++ standard, ovvero la versione di C++ definita dal comitato di standardizzazione ANSI/ISO e implementata da tutti i compilatori C++ recenti. Il codice contenuto in questo volume descrive lo stile di codifica e le pratiche di programmazione incoraggiate dallo standard. Tuttavia se si usa un vecchio compilatore, potrebbe accadere che i programmi di questo volume non vengano accettati. Ecco il motivo. Durante il processo di standardizzazione, il comitato ANSI/ISO ha aggiunto al linguaggio molte nuove funzionalità. A mano a mano che queste funzionalità venivano definite, sono state implementate dagli sviluppatori di compilatori. Naturalmente esiste sempre un intervallo di tempo fra l'aggiunta di una nuova funzionalità e la sua disponibilità nei compilatori commerciali. Poiché tali funzionalità sono state aggiunte al C++ lungo un arco di alcuni anni, un compilatore non recente potrebbe non supportate una o più di esse. Questo è particolarmente importante nel caso di due recenti aggiunte del linguaggio C++ che influenzano tutti i programmi, anche i più semplici. Se si usa un vecchio compilatore che non accetta queste nuove funzionalità, nessun problema. Vi è una soluzione rapida e agevole.

Le differenze principali fra codice "vecchio stile" e codice moderno riguardano due funzionalità: i nuovi header e l'istruzione namespace. Per comprendere queste differenze, si partirà da due versioni di un semplicissimo programma C++ che non fa assolutamente nulla. La prima versione riflette il modo in cui venivano scritti i programmi C++ vecchio stile.

```
/*
   Un programma C++ vecchio stile.
*/

#include <iostream.h>

int main()
{
  return 0;
}
```

Si faccia attenzione all'istruzione #include. Tale istruzione include il file iostream.h e non l'header <iostream>. Inoltre si noti che non vi è alcuna istruzione namespace. Ecco la seconda versione di questo programma, riscritta in C++ standard.

```
/*
   Un moderno programma C++ che utilizza
   il nuovo stile header e un namespace.
*/
#include <iostream>

using namespace std;

int main()
{
  return 0;
}
```



Questa versione utilizza il nuovo stile di header e specifica un namespace. Entrambe queste funzionalità sono state accennate in precedenza e ora verranno descritte in modo più approfondito.

## I nuovi header C++

Come si sa, quando si usa una funzione di libreria in un programma C, si deve includere il relativo file header. Tale operazione viene eseguito tramite l'istruzione #include. Ad esempio, in C, per includere il file header per le funzioni di I/O si deve utilizzare la seguente istruzione:

```
#include <stdio.h>
```

Qui, stdio.h è il nome del file utilizzato dalle funzioni di I/O e l'istruzione precedente provoca l'inclusione fisica di tale file nel programma.

Quando venne inventato il linguaggio C++ e per molti anni, si è usato lo stesso stile di inclusione dei file header ereditato dal C. Pertanto venivano utilizzati veri e propri file. Lo standard C++ supporta ancora l'inclusione di file header in stile C, ad esempio per tutti i file header creati dal programmatore oltre che per motivi di compatibilità all'indietro.

Ma lo standard per il C++ ha creato un nuovo tipo di header utilizzato dalla libreria standard C++. I nuovi header C++ non specificano nomi di file ma identificatori standard che non necessariamente sono file. I nuovi header C++ sono un'astrazione che garantisce semplicemente la dichiarazione dei prototipi e delle definizioni richiesti dalla libreria C++.

Poiché i nuovi header non sono file, non hanno l'estensione .h. Essi sono costituiti unicamente dal nome dell'header racchiuso fra parentesi angolari. Ad esempio ecco alcuni dei nuovi header supportati dal C++ standard.

```
<iostream>      <fstream>      <vector>      <string>
```

I nuovi header sono inclusi utilizzando l'istruzione #include. L'unica differenza è il fatto che i nuovi header non rappresentano necessariamente file. Poiché il linguaggio C++ include l'intera libreria di funzioni C, supporta anche i file header C associati a tale libreria. Pertanto sono ancora disponibili i file header

stdio.h e ctype.h. Tuttavia lo standard del C++ definisce anche nuovi header da utilizzare in luogo di questi file. Le versioni C++ degli header aggiungono semplicemente un prefisso "c" al nome del file e non usano il suffisso .h. Ad esempio, il nuovo header C++ di math.h è <cmath>. Quello per string.h è <string>. Anche se attualmente è ancora possibile includere file header C quando si devono utilizzare delle funzioni della libreria C, tale approccio è sconsigliato dal C++ standard. Per questo motivo, da questo punto in avanti si utilizzerà l'istruzione #include unicamente con i nuovi header. Se il compilatore non dovesse supportare questi nuovi header, basterà sostituirli con le vecchie versioni in stile C.

Dato che i nuovi header sono un'aggiunta recente al linguaggio C++, si troveranno moltissimi vecchi programmi che non li impiegano. Questi programmi utilizzano i file header C. Ecco il metodo tradizionale di inclusione del file header per le funzioni di I/O.

```
#include <iostream.h>
```

Questa istruzione provoca l'inclusione nel programma del file iostream.h. In generale, un vecchio file header utilizza lo stesso nome del corrispondente nuovo header ma gli aggiunge il suffisso .h.

Al momento attuale, tutti i compilatori C++ supportano anche i vecchi file header. Tuttavia questa soluzione è stata dichiarata obsoleta e dunque se ne sconsiglia l'uso nei programmi di nuovo sviluppo. Questo è il motivo per cui tale approccio non verrà impiegato in questo volume.

**SUGGERIMENTO** *Anche se i vecchi file header sono tuttora molto comuni, se ne sconsiglia l'uso in quanto obsoleti.*

### I namespace

Quando si include un nuovo header in un programma, il contenuto di tale header si trova nel namespace std. Un namespace è semplicemente una regione di dichiarazioni. Lo scopo di un namespace è quello di localizzare i nomi degli identificatori per evitare conflitti. Gli elementi dichiarati in un namespace sono distinti rispetto agli elementi dichiarati in un altro namespace. Originariamente, i nomi delle funzioni di libreria C++ venivano semplicemente inseriti nel namespace globale (si pensi ad esempio al C). Con la nascita dei nuovi header, il contenuto di questi header è stato inserito nel namespace std. Si parlerà più dettagliatamente dei namespace più avanti in questo volume. Per il momento non occorre preoccuparsene troppo in quanto l'istruzione:

```
using namespace std;
```



non fa altro che rendere visibile il namespace std (ovvero inserisce std nel namespace globale). Dopo la compilazione di tale istruzione, non vi è alcuna differenza fra lavorare con un file header vecchio stile o con un nuovo header.

Un'ultima indicazione: per motivi di compatibilità, quando un programma C++ include un file header C come ad esempio stdio.h, il suo contenuto viene inserito nel namespace globale. Questo consente ai compilatori C++ di compilare i programmi C.

### Utilizzo di un vecchio compilatore

Come si è detto, sia i namespace che i nuovi header sono stati aggiunti piuttosto recentemente al linguaggio C++, durante la fase di standardizzazione. Dunque non tutti i nuovi compilatori C++ potrebbero supportare queste funzionalità. In questo caso il compilatore rileverà uno o più errori quando tenterà di compilare le prime due righe dei programmi presentati in questo volume. In questo caso vi è una semplice soluzione: basta utilizzare i file header vecchio stile e cancellare l'istruzione namespace. In pratica basta sostituire:

```
#include <iostream>
using namespace std;
```

con:

```
#include <iostream.h>
```

Questa modifica trasforma un programma moderno in un programma vecchio stile. Poiché in questo caso il contenuto dei file header vecchio stile viene inserito nel namespace globale, non vi è alcuna necessità di impiegare l'istruzione namespace.

Un'ultima annotazione: per il momento e per i prossimi anni, si troveranno molti programmi C++ che utilizzano i file header vecchio stile e non impiegano l'istruzione namespace. Il compilatore non avrà problemi a compilarli ma è opportuno realizzare i nuovi programmi in stile moderno per adeguarsi allo standard del linguaggio C++. Comunque le vecchie funzionalità continueranno ad essere supportate per anni.

## 11.5 Introduzione alle classi C++

Questa sezione introduce la funzionalità più importante del C++: la classe. In C++, per creare un oggetto si deve innanzi tutto definire la sua forma generale

utilizzando la parola chiave class. Una classe ha una sintassi simile a una struttura. Ecco un esempio. La seguente classe definisce un tipo chiamato stack utilizzato appunto per creare uno stack:

```
#define SIZE 100

// Creazione della classe stack.
class stack {
  int stck[SIZE];
  int tos;
public:
  void init();
  void push(int i);
  int pop();
};
```

Una classe può contenere parti private e pubbliche. In generale, tutti gli oggetti definiti all'interno di una classe sono privati. Ad esempio, le variabili stck e tos sono private. Questo significa che non sono visibili da nessun'altra funzione che non sia un membro della classe. Questo è uno dei modi in cui si ottiene l'incapsulamento: l'accesso a determinati oggetti e dati può essere controllato in modo rigido mantenendoli privati. Anche se questo non viene illustrato dall'esempio presentato, è possibile definire funzioni private che possono essere richiamate solo dai membri della classe.

Per rendere pubblica (ovvero accessibile da altre parti del programma) una parte della classe, è necessario dichiararla esplicitamente come pubblica utilizzando la parola chiave public. Tutte le variabili e le funzioni definite dopo public possono essere utilizzate da tutte le altre funzioni del programma. Essenzialmente, la parte rimanente del programma accede a un oggetto utilizzando le sue funzioni pubbliche. Anche se è possibile avere variabili pubbliche, si deve in generale cercare di limitarne l'uso. Quindi si dovrà cercare di rendere tutti i dati privati e controllare l'accesso ai dati utilizzando funzioni pubbliche. Un'ultima annotazione: si noti che la parola chiave public è seguita dal carattere di due punti.

Le funzioni init(), push() e pop() sono chiamate *funzioni membro* poiché sono membri della classe stack. Le variabili stck e tos sono chiamate *variabili membro*. Si ricordi che un oggetto racchiude codice e dati. Solo le funzioni membro hanno accesso ai membri privati della classe in cui sono dichiarate. Pertanto solo init(), push() e pop() potranno accedere a stck e tos.

Dopo aver definito una classe, è possibile creare un oggetto di tale tipo semplicemente impiegando il nome della classe. In pratica, il nome della classe diviene un nuovo specificatore di tipo. Ad esempio, la seguente istruzione crea l'oggetto mystack di tipo stack.



```
stack mystack;
```

Quando si dichiara un oggetto di una classe, si crea un'*istanza* di tale classe. In questo caso, mystack è un'istanza di stack. È anche possibile creare oggetti nel luogo stesso in cui viene definita la classe, specificandone il nome dopo la parentesi graffa di chiusura, esattamente come avviene nel caso delle strutture. Per ricapitolare: in C++ class crea un nuovo tipo di dati che può essere utilizzato per creare oggetti di tale tipo. Pertanto, un oggetto è un'istanza di una classe esattamente come altre variabili sono istanze, ad esempio del tipo int. In altre parole, una classe è un'astrazione logica, mentre un oggetto è reale (ovvero esiste all'interno della memoria del computer).

La forma generale di una dichiarazione di una semplice classe è la seguente:

class *nome-classe* {
  *dati e funzioni private*
public:
  *dati e funzioni pubbliche*
} *elenco oggetti*;

Naturalmente, l'*elenco oggetti* può essere vuoto.

All'interno della dichiarazione di stack, le funzioni membro sono identificate dall'uso dei prototipi. In C++, tutte le funzioni devono avere un prototipo. In altre parole i prototipi non sono opzionali. Il prototipo di una funzione membro all'interno della definizione di una classe funge in generale da prototipo della funzione.

Quando si dovrà realizzare una funzione membro di una classe, si dovrà dire al compilatore la classe a cui appartiene la funzione, qualificandone il nome con il nome della classe di cui la funzione è membro. Ad esempio, la funzione push() può essere codificata nel seguente modo:

```
void stack::push(int i)
{
  if(tos==SIZE) {
    cout << "Stack esaurito.";
    return;
  }
  stck[tos] = i;
  tos++;
}
```

L'operatore :: è chiamato *operatore di risoluzione del campo d'azione*. Essenzialmente, l'operatore dice al compilatore che questa versione di push() appartiene alla classe stack o in altre parole, che push() è nel campo d'azione di stack. In

C++ lo stesso nome di funzione può essere utilizzato da più classi. Il compilatore può determinare la classe a cui appartiene una funzione grazie all'operatore di risoluzione del campo d'azione.

Quando si fa riferimento a un membro di una classe da una parte di codice che non si trova all'interno della classe stessa, l'operazione deve essere eseguita sempre in congiunzione con un oggetto di tale classe. A tale scopo si deve utilizzare il nome dell'oggetto seguito dall'operatore punto seguito a sua volta dal membro. Questa regola si applica sia che si debba accedere a dati membro che a funzioni membro. Ad esempio, il frammento di codice seguente richiama la funzione init() per l'oggetto stack1.

```
stack stack1, stack2;

stack1.init();
```

Questo frammento di codice crea i due oggetti stack1 e stack2 e poi inizializza stack1. È molto importante comprendere che stack1 e stack2 sono due oggetti distinti. Questo significa, ad esempio, che l'inizializzazione di stack1 non provoca l'inizializzazione anche di stack2. L'unica relazione che intercorre fra stack1 e stack2 consiste nel fatto che sono oggetti dello stesso tipo.

All'interno di una classe, una funzione membro può richiamare un'altra funzione membro oppure far riferimento direttamente ai dati membro senza utilizzare l'operatore punto. Il nome dell'oggetto e l'operatore punto devono essere utilizzati solo quando l'accesso a un membro avviene da parte di codice che non appartiene alla classe.

Il programma seguente raccoglie le parti illustrate finora e le parti mancanti e mostra l'utilizzo della classe stack:

```
#include <iostream>
using namespace std;

#define SIZE 100

// Creazione della classe stack.
class stack {
  int stck[SIZE];
  int tos;
public:
  void init();
  void push(int i);
  int pop();
};
```



```
void stack::init()
{
  tos = 0;
}

void stack::push(int i)
{
  if(tos==SIZE) {
    cout << "Stack esaurito.";
    return;
  }
  stck[tos] = i;
  tos++;
}

int stack::pop()
{
  if(tos==0) {
    cout << "Stack vuoto.";
    return 0;
  }
  tos--;
  return stck[tos];
}

int main()
{
  stack stack1, stack2;  // crea due oggetti della classe stack

  stack1.init();
  stack2.init();

  stack1.push(1);
  stack2.push(2);

  stack1.push(3);
  stack2.push(4);

  cout << stack1.pop() << " ";
  cout << stack1.pop() << " ";
  cout << stack2.pop() << " ";
  cout << stack2.pop() << "\n";

  return 0;
}
```

Ecco l'output del programma:

```
3 1 4 2
```

Un'annotazione: le parti private di un oggetto sono accessibili solo da parte delle funzioni membro di tale oggetto. Ad esempio, l'istruzione:

```
stack1.tos = 0; // errore
```

non può trovarsi nella funzione main() del programma precedente poiché tos è privata.

## 11.6 L'overloading delle funzioni

Un modo utilizzato dal C++ per ottenere il polimorfismo consiste nell'uso di tecniche di overloading delle funzioni. In C++, due o più funzioni possono condividere lo stesso nome sempre che i loro parametri siano differenti. In questo caso, si dice che le funzioni che condividono lo stesso nome sono state *sovraccaricate* (o che hanno subito *overloading*) e il processo è chiamato *overloading delle funzioni*.

Per vedere il motivo per cui l'overloading delle funzioni è un fattore importante, si considerino tre funzioni definite nel sottoinsieme C: abs(), labs() e fabs(). La funzione abs() restituisce il valore assoluto di un intero, labs() restituisce il valore assoluto in un long e fabs() restituisce il valore assoluto di un double. Anche se queste funzioni eseguono operazioni praticamente identiche, in C devono essere rappresentate da tre nomi leggermente diversi. Questo complica la situazione sia concettualmente che in pratica. Anche se il funzionamento di ogni funzione è identico, il programmatore dovrà ricordare tre nomi al posto di uno. In C++ è invece possibile utilizzare lo stesso nome per le tre funzioni come si può vedere dall'esempio seguente:

```
#include <iostream>
using namespace std;

// abs assume tre significati grazie all'overloading
int abs(int i);
double abs(double d);
long abs(long l);

int main()
{
  cout << abs(-10) << "\n";

  cout << abs(-11.0) << "\n";

  cout << abs(-9L) << "\n";

  return 0;
}
int abs(int i)
{
  cout << "abs() per interi\n";

  return i<0 ? -i : i;
}

double abs(double d)
{
  cout << "abs() per double\n";

  return d<0.0 ? -d : d;
}

long abs(long l)
{
  cout << "abs() per long\n";

  return l<0 ? -l : l;
}
```

Ecco l'output prodotto dal programma:

```
abs() per interi
10
abs() per double
11
abs() per long
9
```

Questo programma crea tre funzioni simili ma differenti chiamate abs() ognuna delle quali restituisce il valore assoluto del proprio argomento. Il compilatore può determinare la funzione da richiamare in un determinato momento sulla base del tipo dell'argomento. L'importanza delle funzioni modificate tramite overloading è dovuta al fatto che consentono di accedere a un gruppo di funzioni correlate con un unico nome. Pertanto, il nome abs() rappresenterà l'*azione generale* che deve

essere eseguita. Sarà compito del compilatore scegliere il *metodo specifico* corretto per una determinata circostanza. Il programmatore dovrà semplicemente ricordare l'azione generale da eseguire. Grazie al polimorfismo, al posto di tre oggetti basterà ricordarsi di uno. Questo esempio è molto semplice ma se si espande il concetto, si può immaginare come il polimorfismo possa aiutare a gestire programmi molto complessi.

In generale, per eseguire l'overloading di una funzione, basta dichiararne versioni diverse. Il resto rimarrà a carico del compilatore. Quando si esegue l'overloading di una funzione, si deve tenere conto di un'importante restrizione: il tipo e/o il numero dei parametri di ogni funzione modificata tramite overloading deve essere diverso. Non è sufficiente che le funzioni restituiscano semplicemente valori di tipo diverso. Devono essere diversi anche i tipi o il numero dei parametri (non sempre il tipo restituito fornisce informazioni sufficienti che consentano al compilatore di decidere la funzione da utilizzare). Naturalmente, le funzioni modificate tramite overloading *possono* restituire valori di tipo diverso.

- Ecco un altro esempio che impiega funzioni modificate tramite overloading:

```
#include <iostream>
#include <cstdio>
#include <cstring>
using namespace std;

void stradd(char *s1, char *s2);
void stradd(char *s1, int i);

int main()
{
  char str[80];

  strcpy(str, "Salve ");
  stradd(str, "a tutti");
  cout << str << "\n";

  stradd(str, 100);
  cout << str << "\n";

  return 0;
}

// concatena due stringhe
void stradd(char *s1, char *s2)
{
  strcat(s1, s2);
}
```



```
// concatena una stringa con un intero convertito in stringa
void stradd(char *s1, int i)
{
  char temp[80];

  sprintf(temp, "%d", i);
  strcat(s1, temp);
}
```

In questo programma, la funzione stradd() è stata modificata tramite overloading. Una versione concatena due stringhe (proprio come strcat()). L'altra versione converte in stringa un intero e poi lo aggiunge a una stringa. In questo caso l'overloading è utilizzato per creare un'interfaccia che consenta di aggiungere a una stringa un'altra stringa oppure un valore intero.

È possibile utilizzare lo stesso nome anche per eseguire l'overloading di funzioni non correlate ma questo utilizzo è sconsigliato. Ad esempio, si potrebbe utilizzare il nome sqr() per creare funzioni che restituiscano il *quadrato* di un valore int e la *radice quadrata* di un valore double. Ma queste operazioni sono per principio diverse. Questa applicazione dell'overloading delle funzioni è contraria ai suoi obiettivi (ed è considerata un pessimo stile di programmazione). In pratica si dovrà eseguire l'overloading solo di funzioni strettamente correlate.

## 11.7 L'overloading degli operatori

In C++ il polimorfismo può essere ottenuto anche eseguendo l'overloading degli operatori. Come si sa, in C++ è possibile utilizzare gli operatori << e >> per eseguire operazioni di I/O dalla console. Questi operatori possono svolgere queste operazioni aggiuntive poiché nell'header <iostream> questi operatori vengono modificati tramite overloading. Quando si esegue l'overloading di un operatore, questo assume un significato aggiuntivo rispetto a una determinata classe. Nel contempo continuerà a conservare i suoi significati precedenti.

In generale, è possibile eseguire l'overloading della maggior parte degli operatori C++ definendone il significato rispetto a una determinata classe. Ad esempio, per tornare alla classe stack sviluppata precedentemente in questo capitolo, è possibile eseguire l'overloading dell'operatore + rispetto a oggetti del tipo stack in modo che l'operatore consenta di aggiungere il contenuto di uno stack al contenuto di un altro. In ogni caso, l'operatore + conserverà il suo significato originario rispetto ad altri tipi di dati.

Poiché l'overloading degli operatori è, nella pratica, un po' più complesso rispetto all'overloading delle funzioni, ne verranno presentati esempi solo a partire dal Capitolo 14.

## 11.8 L'ereditarietà

Come si è detto in precedenza in questo capitolo, l'ereditarietà è uno dei fattori più importanti di un linguaggio di programmazione a oggetti. In C++, l'ereditarietà è ottenuta consentendo a una classe di incorporare un'altra classe nella propria dichiarazione. L'ereditarietà consente la realizzazione di una gerarchia di classi, partendo dalla classe più generale per giungere a quella più specifica. Il processo richiede quindi la definizione di una *classe base* che definisce le qualità comuni a tutti gli oggetti che derivano da tale classe. La classe base rappresenta la descrizione più generale. Le classi che discendono dalla classe base sono chiamate *classi derivate*. Una classe derivata include tutte le funzionalità della classe base e vi aggiunge qualità specifiche. Per dimostrare il funzionamento di questa tecnica, il prossimo esempio crea una serie di classi che consente di catalogare diversi tipi di edifici.

Per iniziare, viene dichiarata la classe building. Questa classe fungerà da base per due classi derivate.

```
class building {
  int rooms;
  int floors;
  int area;
public:
  void set_rooms(int num);
  int get_rooms();
  void set_floors(int num);
  int get_floors();
  void set_area(int num);
  int get_area();
};
```

Poiché (semplificando) tutti gli edifici hanno tre caratteristiche in comune (una o più stanze, uno o più piani e un'area totale) la classe building incorpora queste componenti nella dichiarazione. Le funzioni membro che iniziano con set impostano i valori dei dati privati. Le funzioni che iniziano con get restituiscono tali valori.

Ora è possibile utilizzare questa definizione generica di edificio per creare classi derivate che descrivono tipi specifici di edifici. Ad esempio, ecco una classe derivata chiamata house:

```
// house deriva da building
class house : public building {
  int bedrooms;
  int baths;
public:
  void set_bedrooms(int num);
  int get_bedrooms();
  void set_baths(int num);
  int get_baths();
};
```



Si noti il modo in cui vengono ereditate le caratteristiche della classe building. La forma generale di ereditarietà è:

```
class classe-derivata : accesso classe-ereditata {
  // corpo della nuova classe
}
```

Qui, *accesso* è opzionale ma quando è presente deve essere public, private o protected (queste opzioni verranno esaminate in dettaglio nel Capitolo 12). Per ora, tutte le classi ereditate utilizzeranno lo specificatore di accesso public. L'utilizzo di public significa che tutti gli elementi pubblici della classe base diverranno elementi pubblici anche delle classi derivate. Pertanto i membri pubblici della classe building diverranno membri pubblici della classe derivata house e saranno disponibili alle funzioni membro di house come se fossero state dichiarate all'interno di house. Le funzioni membro di house non avranno però accesso agli elementi privati di building. Questo è un fatto molto importante: anche se house eredita building, avrà accesso solo ai membri pubblici di building. In questo modo, l'ereditarietà non si scontra con i principi di incapsulamento necessari nella programmazione a oggetti.

**SUGGERIMENTO** *Una classe derivata può accedere direttamente sia ai propri membri che ai membri pubblici della classe base da cui deriva.*

Il seguente programma illustra l'uso dell'ereditarietà. Il programma crea due classi derivate di building facendo uso dell'ereditarietà: una si chiama house e l'altra school.

```
#include <iostream>
using namespace std;

class building {
  int rooms;
  int floors;
  int area;
public:
  void set_rooms(int num);
```

```
  int get_rooms();
  void set_floors(int num);
  int get_floors();
  void set_area(int num);
  int get_area();
};

// house deriva da building
class house : public building {
  int bedrooms;
  int baths;
public:
  void set_bedrooms(int num);
  int get_bedrooms();
  void set_baths(int num);
  int get_baths();
};

// anche school deriva da building
class school : public building {
  int classrooms;
  int offices;
public:
  void set_classrooms(int num);
  int get_classrooms();
  void set_offices(int num);
  int get_offices();
};

void building::set_rooms(int num)
{
  rooms = num;
}

void building::set_floors(int num)
{
  floors = num;
}

void building::set_area(int num)
{
  area = num;
}

int building::get_rooms()
{
```



```
  return rooms;
}

int building::get_floors()
{
  return floors;
}

int building::get_area()
{
  return area;
}

void house::set_bedrooms(int num)
{
  bedrooms = num;
}

void house::set_baths(int num)
{
  baths = num;
}

int house::get_bedrooms()
{
  return bedrooms;
}

int house::get_baths()
{
  return baths;
}

void school::set_classrooms(int num)
{
  classrooms = num;
}

void school::set_offices(int num)
{
  offices = num;
}

int school::get_classrooms()
{
```

```
  return classrooms;
}

int school::get_offices()
{
  return offices;
}

int main()
{
  house h;
  school s;

  h.set_rooms(12);
  h.set_floors(3);
  h.set_area(4500);
  h.set_bedrooms(5);
  h.set_baths(3);

  cout << "La casa ha " << h.get_bedrooms();
  cout << " camere da letto\n";

  s.set_rooms(200);
  s.set_classrooms(180);
  s.set_offices(5);
  s.set_area(25000);

  cout << "La scuola ha " << s.get_classrooms();
  cout << " classi\n";
  cout << "La sua area è di " << s.get_area();

  return 0;
}
```

Ecco l'output prodotto dal programma:

```
La casa ha 5 camere da letto
La scuola ha 180 classi
La sua area è di 25000
```

Come si vede da questo programma, il vantaggio principale dell'ereditarietà consiste nel fatto che è possibile creare una classificazione generale che può essere incorporata in oggetti più specifici. In questo modo, ogni oggetto può rappresentare con precisione la propria sottoclasse.



Parlando del C++, per descrivere la relazione di ereditarietà in genere si usano i termini *classe base* e *classe derivata*, ma talvolta si dice anche *classe genitore* e *classe figlia* oppure *superclasse* e *sottoclasse*.

Oltre a fornire i vantaggi della classificazione gerarchica, l'ereditarietà funge anche da supporto per il polimorfismo al momento dell'esecuzione (run-time), grazie al meccanismo delle funzioni virtuali (per informazioni consultare il Capitolo 16).

## 11.9 I costruttori e i distruttori

È molto comune che una parte di un oggetto debba essere inizializzata prima dell'uso. Ad esempio, per tornare alla classe stack sviluppata precedentemente in questo capitolo, prima di iniziare a utilizzare lo stack, tos deve essere inizializzata a zero. Questo si ottiene richiamando la funzione init(). Poiché capita molto spesso di dover inizializzare un oggetto, il C++ consente di inizializzare gli oggetti al momento della creazione. Questa inizializzazione automatica è ottenuta grazie all'impiego di una funzione costruttore.

Una *funzione costruttore* è una particolare funzione membro di una classe che porta lo stesso nome della classe. Ad esempio, ecco l'aspetto della classe stack convertita in modo da utilizzare una funzione costruttore per l'inizializzazione:

```
// Creazione della classe stack.
class stack {
  int stck[SIZE];
  int tos;
public:
  stack();  // costruttore
  void push(int i);
  int pop();
};
```

Si noti che per il costruttore stack() non viene specificato il tipo restituito. In C++ le funzioni costruttore non possono restituire valori e pertanto non si deve specificare il tipo restituito.

La funzione stack() può essere codificata nel seguente modo:

```
// funzione costruttore dello stack
stack::stack()
{
  tos = 0;
  cout << "Stack inizializzato\n";
}
```

Si deve ricordare che il messaggio Stack inizializzato viene prodotto solo per illustrare l'uso del costruttore. Nella pratica, la maggior parte delle funzioni costruttore non ha bisogno ne di input ne di output. Semplicemente la funzione si occupa di eseguire varie inizializzazioni.

Il costruttore di un oggetto viene richiamato automaticamente nel momento in cui deve essere creato l'oggetto. Questo significa che viene richiamata al momento della dichiarazione dell'oggetto. Se si è abituati a pensare che una dichiarazione sia un'istruzione passiva, occorre prepararsi a cambiare idea. In C++ una dichiarazione è un'istruzione che viene eseguita come qualunque altra. La distinzione non è puramente accademica. Il codice eseguito per costruire un oggetto può essere anche molto ingente. Il costruttore di un oggetto viene richiamato una sola volta per ogni oggetto globale o locale static. Nel caso di oggetti locali, il costruttore viene richiamato ogni volta che si incontra la dichiarazione di un nuovo oggetto.

L'operazione complementare del costruttore è svolta dal *distruttore*. In molte circostanze, un oggetto deve eseguire una o più azioni nel momento in cui ne finisce l'esistenza. Gli oggetti locali vengono costruiti nel momento in cui si entra nel blocco in cui si trovano e vengono distrutti all'uscita dal blocco. Gli oggetti globali vengono distrutti nel momento in cui termina il programma. Quando viene distrutto un oggetto, viene automaticamente richiamato il relativo distruttore (se presente). Vi sono molti casi in cui è necessario utilizzare una funzione distruttore. Ad esempio, potrebbe essere necessario deallocare la memoria precedentemente allocata dall'oggetto oppure potrebbe essere necessario chiudere un file aperto. In C++ è la funzione distruttore a gestire gli eventi di disattivazione. Il distruttore ha lo stesso nome del costruttore ma è preceduto dal carattere ~. Ad esempio, ecco una classe stack con le relative funzioni costruttore e distruttore (in realtà la classe stack non richiede l'uso di un distruttore che viene presentato a puro scopo illustrativo).

```
// Creazione della classe stack.
class stack {
  int stck[SIZE];
  int tos;
public:
  stack();  // costruttore
  ~stack(); // distruttore
  void push(int i);
  int pop();
};

// funzione costruttore dello stack
stack::stack()
{
```



```
  tos = 0;
  cout << "Stack inizializzato\n";
}

// funzione distruttore dello stack
stack::~stack()
{
  cout << "Stack distrutto\n";
}
```

Si noti che, come le funzioni costruttore, le funzioni distruttore non restituiscono valori.

Per vedere il funzionamento dei costruttori e dei distruttori, ecco una nuova versione del programma stack esaminato precedentemente in questo capitolo. Si noti che non è più necessario utilizzare la funzione init().

```
#include <iostream>
using namespace std;

#define SIZE 100

// Creazione della classe stack.
class stack {
  int stck[SIZE];
  int tos;
public:
  stack();  // costruttore
  ~stack(); // distruttore
  void push(int i);
  int pop();
};

// funzione costruttore dello stack
stack::stack()
{
  tos = 0;
  cout << "Stack inizializzato\n";
}

// funzione distruttore dello stack
stack::~stack()
{
  cout << "Stack distrutto\n";
}
```

```
void stack::push(int i)
{
  if(tos==SIZE) {
    cout <<"Stack esaurito.";
    return;
  }
  stck[tos] = i;
  tos++;
}

int stack::pop()
{
  if(tos==0) {
    cout << "Stack vuoto.";
    return 0;
  }
  tos--;
  return stck[tos];
}

int main()
{
  stack a, b;  // crea due oggetti della classe stack

  a.push(1);
  b.push(2);

  a.push(3);
  b.push(4);

  cout << a.pop() << " ";
  cout << a.pop() << " ";
  cout << b.pop() << " ";
  cout << b.pop() << "\n";

  return 0;
}
```

Il programma produce il seguente output:

```
Stack inizializzato
Stack inizializzato
3 1 4 2
Stack distrutto
Stack distrutto
```



## 11.10 Le parole riservate del C++

Attualmente lo Standard per il linguaggio C++ definisce 63 parole riservate, elencate nella Tabella 11.1. Insieme alla sintassi formale del linguaggio esse costituiscono il nucleo del linguaggio C++. Le versioni meno recenti di C++ definivano anche la parola overload che oggi è obsoleta. Si deve tenere in considerazione il fatto che il C++ distingue fra lettere maiuscole e minuscole e che pertanto le parole riservate devono essere scritte in lettere minuscole.

**Tabella 11.1** Le parole chiave del C++.

| | | | |
|---|---|---|---|
| asm | auto | bool | break |
| case | catch | char | class |
| const | const_cast | continue | default |
| delete | do | double | dynamic_cast |
| else | enum | explicit | export |
| extern | false | float | for |
| frieng | goto | if | inline |
| int | long | mutable | namespace |
| new | operator | private | protected |
| public | register | reinterpret_cast | return |
| short | signed | sizeof | static |
| static_cast | struct | switch | template |
| this | throw | true | try |
| typedef | typeid | typename | union |
| unsigned | using | virtual | void |
| volatile | wchar_t | while | |

## 11.11 La forma generale di un programma C++

Anche se esistono vari stili di programmazione, la maggior parte dei programmi C++ ha la seguente forma generale:

```
#include
dichiarazioni delle classi base
```

```
dichiarazioni delle classi derivate
prototipi delle funzioni non-membro
int main( )
{
   .
   .
   .
}
definizioni delle funzioni non-membro
```

Nella maggior parte dei progetti più estesi, tutte le dichiarazioni delle classi verranno inserite in un file header e incluse in ogni modulo ma l'aspetto generale del programma rimarrà lo stesso.

I prossimi capitoli esaminano in dettaglio le funzionalità introdotte in questo capitolo insieme ad altri aspetti del linguaggio C++.

Capitolo 12

# Le classi e gli oggetti


12.1 **Le classi**
12.2 **Le strutture e le classi**
12.3 **Le unioni e le classi**
12.4 **Le funzioni friend**
12.5 **Le classi friend**
12.6 **Le funzioni inline**
12.7 **Definizione di funzioni inline all'interno di una classe**
12.8 **I costruttori parametrizzati**
12.9 **I membri static di una classe**
12.10 **Quando vengono eseguiti i costruttori e i distruttori?**
12.11 **L'operatore di risoluzione del campo d'azione**
12.12 **La nidificazione delle classi**
12.13 **Le classi locali**
12.14 **Il passaggio di oggetti a funzioni**
12.15 **La restituzione di oggetti**
12.16 **L'assegnamento di oggetti**


In C++, la classe costituisce la base della programmazione a oggetti. In particolare, la classe definisce la natura di un oggetto ed è l'unità principale di incapsulamento del C++. In questo capitolo vengono esaminati in dettaglio le classi e gli oggetti.

## 12.1 Le classi

Le classi vengono create mediante la parola chiave class. La dichiarazione di una classe definisce un nuovo tipo che racchiude sia il codice che i dati. Questo nuovo tipo verrà utilizzato per dichiarare oggetti di tale classe. Pertanto, una classe è un'astrazione logica mentre un oggetto ha esistenza fisica. In altre parole, un oggetto è un'*istanza* di una classe.

La dichiarazione di una classe è sintatticamente simile a quella di una struttura. Nel Capitolo 11 si è mostrata una forma generale e semplificata della dichiarazione di una classe. Di seguito viene presentata la forma generale completa della dichiarazione di una classe che non erediti proprietà da altre classi.

class *nome-classe*{
  *dati e funzioni privati*
*specificatori di accesso*:
  *dati e funzioni*
*specificatori di accesso*:
  *dati e funzioni*
.
.
.
*specificatori di accesso*:
  *dati e funzioni*
} *elenco oggetti*;

L'*elenco oggetti* è opzionale. Se è presente dichiara gli oggetti di tale classe. Qui la parte *specificatori di accesso* può essere rappresentata da una di queste tre parole chiave del C++:

public
private
protected

Le funzioni e i dati dichiarati all'interno di una classe sono normalmente privati di tale classe e possono essere utilizzati solo dagli altri membri della classe. Utilizzando lo specificatore di accesso public si consente però anche ad altre parti del programma di accedere alle funzioni o ai dati della classe. Lo specificatore di accesso protected è richiesto solo in caso di ereditarietà (vedere il Capitolo 15). Una volta utilizzato, uno specificatore d'accesso rimane attivo finché non viene indicato un altro specificatore di accesso o finché non viene raggiunta la fine della dichiarazione della classe. All'interno della dichiarazione di una classe, è possibile cambiare specificatore di accesso il numero di volte desiderato. Ad esempio è possibile utilizzare lo specificatore public per un gruppo di dichiarazioni e poi tornare allo specificatore private. Questa possibilità è esemplificata dalla dichiarazione della seguente classe:

```
#include <iostream>
#include <cstring>
using namespace std;
```



```
class employee {
  char name[80];            // dichiarazione privata
public:
  void putname(char *n);    // pubbliche
  void getname(char *n);
private:
  double wage;              // ancora privata
public:
  void putwage(double w);   // di nuovo pubbliche
  double getwage();
} ;

void employee::putname(char *n)
{
  strcpy(name, n);
}

void employee::getname(char *n)
{
  strcpy(n, name);
}

void employee::putwage(double w)
{
  wage = w;
}

double employee::getwage()
{
  return wage;
}

int main()
{
  employee ted;
  char name[80];

  ted.putname("Mario Rossi");
  ted.putwage(75000);

  ted.getname(name);
  cout << name << " guadagna ";
  cout << ted.getwage() << " KLire l'anno.";

  return 0;
}
```

Qui, employee è una semplice classe che può essere utilizzata per memorizzare il nome e lo stipendio di un dipendente. Si noti che lo specificatore di accesso public viene utilizzato due volte.

Anche se all'interno della dichiarazione di una classe è possibile utilizzare gli specificatori di accesso il numero di volte desiderato, l'unico vantaggio che si trarrà consiste nella maggior facilità di lettura e di comprensione del programma. Dal punto di vista del compilatore invece l'uso di più specificatori di accesso non fa alcuna differenza. I programmatori invece trovano in genere più comodo avere una sezione private, una sezione protected e una sezione public in ogni classe. Ad esempio, la maggior parte dei programmatori C++ utilizzerà una classe employee simile alla seguente, con tutti gli elementi privati e pubblici raggruppati.

```
class employee {
  char name[80];
  double wage;
public:
  void putname(char *n);
  void getname(char *n);
  void putwage(double w);
  double getwage();
} ;
```

Le funzioni dichiarate all'interno di una classe sono chiamate *funzioni membro*. Le funzioni membro possono accedere a tutti gli elementi della classe di cui fanno parte e quindi anche agli elementi private. Le variabili che sono elementi di una classe sono chiamate *variabili membro* o *dati membri*. In senso generale, tutti gli elementi di una classe sono detti *membri* di tale classe.

Sono poche le restrizioni applicabili ai membri di una classe. Una variabile membro non static non può avere un inizializzatore. Nessun membro può essere un oggetto della classe dichiarata (anche se un membro può essere un puntatore alla classe dichiarata). Nessun membro può essere dichiarato come auto, extern o register. In generale, si dovranno rendere tutti i dati membri di una classe privati di tale classe. Questo consente di mantenere l'incapsulamento dei dati. Tuttavia vi possono essere situazioni in cui si devono rendere pubbliche una o più variabili (ad esempio per una variabile molto utilizzata potrebbe essere necessario consentire un accesso globale in modo da ottenere tempi di esecuzione più rapidi). Quando una variabile è pubblica, è possibile accedere ad essa direttamente da qualsiasi punto del programma. La sintassi di accesso a dati membri pubblici è la stessa di una chiamata a una funzione membro: si deve specificare il nome dell'oggetto, il punto e il nome della variabile. Il semplice programma seguente illustra l'uso di una variabile pubblica.



```
#include <iostream>
using namespace std;

class myclass {
public:
  int i, j, k; // accessibile all'intero programma
};

int main()
{
  myclass a, b;
  a.i = 100;  // accesso diretto a i, j e k
  a.j = 4;
  a.k = a.i * a.j;

  b.k = 12;  // attenzione, a.k e b.k sono diverse
  cout << a.k << " " << b.k;

  return 0;
}
```

## 12.2 Le strutture e le classi

Le strutture fanno parte del sottoinsieme che il C++ ha ereditato dal C. Come si è visto, una classe è molto simile a una struttura. Ma le relazioni che legano le classi e le strutture sono anche maggiori di quanto possa sembrare. Il C++ ha elevato il ruolo della classica struttura C a quello di metodo alternativo per la creazione di una classe. Infatti l'unica differenza fra una classe e una struttura è il fatto che normalmente tutti i membri di una struttura sono pubblici e tutti i membri di una classe sono privati. In tutti gli altri sensi, le strutture e le classi sono equivalenti. Questo significa che in C++ una struttura definisce un tipo di classe. Ad esempio, si consideri il breve programma seguente che utilizza una struttura per dichiarare una classe che controlla l'accesso a una stringa.

```
#include <iostream>
#include <cstring>
using namespace std;

struct mystr {
  void buildstr(char *s); // pubblica
  void showstr();
private: // ora passa al privato
```

```
  char str[255];
} ;

void mystr::buildstr(char *s)
{
  if(!*s) *str = '\0'; // inizializzazione della stringa
  else strcat(str, s);
}

void mystr::showstr()
{
  cout << str << "\n";
}

int main()
{
  mystr s;

  s.buildstr(""); // init
  s.buildstr("Salve ");
  s.buildstr("a tutti!");

  s.showstr();

  return 0;
}
```

Questo programma visualizza la stringa Salve a tutti!

La classe mystr può essere riscritta utilizzando una classe nel modo seguente:

```
class mystr {
  char str[255];
public:
  void buildstr(char *s); // pubblica
  void showstr();
} ;
```

Ci si potrebbe chiedere il motivo per cui il C++ contenga due parole chiave praticamente equivalenti come struct e class. Questa che sembra una ridondanza è giustificata da vari motivi. Innanzi tutto non vi è alcun motivo per non espandere le funzionalità di una struttura. In C le strutture forniscono già un mezzo per raggruppare i dati, pertanto, basta poco per consentire che includano funzioni membro. In secondo luogo, poiché le strutture e le classi sono correlate fra loro, può essere più facile trasportare i programmi C in C++. Infine, anche se struct e class sono oggi praticamente equivalenti, la presenza di due diverse parole chiave



consente di far evolvere la definizione di classe. Per fare in modo che il C++ conservi la compatibilità con il C, struct deve invece mantenere il significato originale che ha in C.

Anche se è possibile utilizzare una struttura al posto di una classe, questo è generalmente sconsigliabile. In generale si dovrà utilizzare una classe quando nel programma si avrà bisogno di una classe e una struttura quando si deve realizzare una classica struttura C. Questo è anche lo stile seguito in questa guida.

**SUGGERIMENTO** *In C++ la dichiarazione di una struttura definisce un tipo di classe.*

## 12.3 Le unioni e le classi

Per definire una classe, oltre a una struttura si può utilizzare una union. In C++ le unioni possono contenere funzioni membro e variabili membro. Inoltre le unioni possono includere funzioni costruttore e distruttore. In C++ un'unione conserva tutte le sue funzionalità C, la più importante delle quali è il fatto che i dati possono condividere la stessa posizione in memoria. Come nel caso delle strutture, i membri delle unioni sono normalmente pubblici e sono completamente compatibili con il C. Nel prossimo esempio verrà utilizzata un'unione per scambiare i due byte che compongono un intero unsigned short (in questo esempio si presuppone che un intero short occupi 2 byte).

```
#include <iostream>
using namespace std;

union swap_byte {
  void swap();
  void set_byte(unsigned i);
  void show_word();

  unsigned u;
  unsigned char c[2];
};

void swap_byte::swap()
{
  unsigned char t;

  t = c[0];
  c[0] = c[1];
  c[1] = t;
}
```

```
void swap_byte::show_word()
{
  cout << u;
}

void swap_byte::set_byte(unsigned i)
{
  u = i;
}

int main()
{
  swap_byte b;

  b.set_byte(49034);
  b.swap();
  b.show_word();

  return 0;
}
```

Come per le strutture, anche la dichiarazione di un'unione definisce in C++ un tipo particolare di classe. Questo significa che il principio di incapsulamento viene sempre conservato.

Vi sono alcune restrizioni che è necessario osservare quando si utilizzano unioni C++. Innanzi tutto un'unione non può ereditare proprietà da altre classi. Inoltre un'unione non può essere una classe base. Un'unione non può contenere funzioni membro virtuali (le funzioni virtuali verranno discusse nel Capitolo 17). Nessuna variabile static può essere membro di un'unione, né si può usare un membro rappresentato da un indirizzo. Un'unione non può avere come membro oggetti che eseguono l'overloading dell'operatore =. Infine nessun oggetto che abbia associata un'esplicita funzione costruttore o distruttore può essere membro di un'unione.

### Unioni anonime

In C++ vi è un tipo particolare di unione chiamata *unione anonima.* Un'unione anonima non include il nome del tipo e pertanto non consente la dichiarazione di variabili di tale tipo. Infatti un'unione anonima dice al compilatore che le variabili membro dell'unione devono condividere la stessa locazione di memoria. I riferimenti alle variabili avvengono però direttamente senza utilizzare l'operatore punto. Ad esempio, si consideri questo programma:



```
#include <iostream>
#include <cstring>
using namespace std;

int main()
{
  // definisce un'unione anonima
  union {
    long l;
    double d;
    char s[4];
  } ;

  // riferimento diretto agli elementi di un'unione
  l = 100000;
  cout << l << " ";
  d = 123.2342;
  cout << d << " ";
  strcpy(s, "hi");
  cout << s;

  return 0;
}
```

Come si può vedere, i riferimenti agli elementi dell'unione, avvengono come se si trattasse di variabili dichiarate come comuni variabili locali. Infatti, dal punto di vista del programma, questo è esattamente ciò che avviene. Inoltre, anche se sono definite all'interno della dichiarazione di un'unione, queste variabili hanno lo stesso livello di visibilità di ogni altra variabile presente nello stesso blocco. Questo significa che i nomi dei membri di un'unione anonima non devono entrare in conflitto con altri identificatori noti all'interno del campo di visibilità di un'unione.

Alle unioni anonime si applicano tutte le restrizioni viste nel caso delle comuni unioni con le seguenti aggiunte. Innanzitutto un'unione anonima può contenere solo dati; non è consentito quindi l'uso di funzioni membro. Le unioni anonime non possono contenere elementi private o protected. Infine, le unioni anonime globali devono essere specificate come static.

## 12.4 Le funzioni friend

Una funzione friend (letteralmente "amica") può accedere a tutti i membri private e protected della classe per la quale è dichiarata come friend. Per dichiarare una

funzione friend, se ne deve includere il prototipo nella classe, facendole precedere la parola chiave friend. Si consideri il seguente programma:

```
#include <iostream>
using namespace std;

class myclass {
  int a, b;
public:
  friend int sum(myclass x);
  void set_ab(int i, int j);
};

void myclass::set_ab(int i, int j)
{
  a = i;
  b = j;
}

// Nota: sum() non è una funzione membro di alcuna classe.
int sum(myclass x)
{
  /* Poiché sum() è friend di myclass,
     può accedere direttamente ad a e b. */

  return x.a + x.b;
}

int main()
{
  myclass n;

  n.set_ab(3, 4);

  cout << sum(n);

  return 0;
}
```

In questo esempio, la funzione sum() non è un membro di myclass. Nonostante questo, la funzione ha pieno accesso ai membri privati della classe. Inoltre, si noti che sum() viene chiamata senza usare l'operatore punto. Poiché non si tratta di una funzione membro, la funzione non deve essere qualificata dal nome dell'oggetto.



Anche se non si trae alcun vantaggio dal fatto che sum() sia friend piuttosto che membro di myclass, vi sono alcuni casi in cui le funzioni friend sono insostituibili. Innanzi tutto le funzioni friend possono essere utili quando si deve eseguire l'overloading di alcuni tipi di operatori (vedere il Capitolo 14). In secondo luogo le funzioni friend semplificano la creazione di alcuni tipi di funzioni di I/O (vedere il Capitolo 17). La terza situazione in cui può essere utile l'uso di funzioni friend si verifica quando due o più classi contengono membri correlati con altre parti del programma. Per iniziare si esaminerà questo terzo uso.

Si immagini che esistano due diverse classi ognuna delle quali visualizza un messaggio sullo schermo nel caso si verifichi una condizione di errore. In altre parti del programma potrebbe essere necessario conoscere se sullo schermo è attualmente visualizzato un messaggio d'errore prima di iniziare a scrivere sullo schermo (in modo da evitare che il messaggio d'errore possa essere accidentalmente cancellato dal nuovo messaggio). Si potrebbe creare una funzione membro in ciascuna classe che restituisca un valore il quale indichi se sullo schermo è attivo un messaggio ma questo richiede l'aggiunta di ulteriore codice per verificare la condizione (ovvero due chiamate di funzione al posto di una). Se la condizione deve essere verificata frequentemente, la continua ripetizione di verifiche potrebbe risultare inaccettabile. Se invece si impiega una funzione che sia friend di entrambe le classi, sarà possibile verificare lo stato di ogni oggetto richiamando una sola funzione. Pertanto, in questo genere di situazioni, una funzione friend consente di generare codice più efficiente. Il concetto è illustrato dal seguente programma.

```
#include <iostream>
using namespace std;

const int IDLE = 0;
const int INUSE = 1;

class C2;  // dichiarazione forward

class C1 {
  int status;  // IDLE = off, INUSE = sullo schermo
  // ...
public:
  void set_status(int state);
  friend int idle(C1 a, C2 b);
};

class C2 {
  int status; // IDLE = off, INUSE = sullo schermo
  // ...
```

```
public:
  void set_status(int state);
  friend int idle(C1 a, C2 b);
};

void C1::set_status(int state)
{
  status = state;
}

void C2::set_status(int state)
{
  status = state;
}

int idle(C1 a, C2 b)
{
  if(a.status || b.status) return 0;
  else return 1;
}

int main()
{
  C1 x;
  C2 y;

  x.set_status(IDLE);
  y.set_status(IDLE);

  if(idle(x, y)) cout << "Si può usare lo schermo.\n";
  else cout << "In uso.\n";

  x.set_status(INUSE);

  if(idle(x, y)) cout << "Si può usare lo schermo.\n";
  else cout << "In uso.\n";

  return 0;
}
```

Si noti che questo programma utilizza una *dichiarazione forward* (*anticipata*) per la classe C2. Questa dichiarazione anticipata è necessaria poiché la dichiarazione di idle() all'interno di C1 fa riferimento a C2 prima che questa venga dichiarata. Per creare una dichiarazione anticipata di una classe, basta utilizzare la forma mostrata in questo programma.



Una funzione friend di una classe può anche essere membro di un'altra. Ad esempio, nel seguente programma la funzione idle() è un membro di C1:

```
#include <iostream>
using namespace std;

const int IDLE = 0;
const int INUSE = 1;

class C2;  // dichiarazione anticipata

class C1 {
  int status;  // IDLE = off, INUSE = sullo schermo
  // ...
public:
  void set_status(int state);
  int idle(C2 b);  // ora è un membro di C1
};

class C2 {
  int status;  // IDLE = off, INUSE = sullo schermo
  // ...
public:
  void set_status(int state);
  friend int C1::idle(C2 b);
};

void C1::set_status(int state)
{
  status = state;
}

void C2::set_status(int state)
{
  status = state;
}

// idle() è membro di C1 e friend di C2
int C1::idle(C2 b)
{
  if(status || b.status) return 0;
  else return 1;
}

int main()
```

```
{
  C1 x;
  C2 y;

  x.set_status(IDLE);
  y.set_status(IDLE);

  if(x.idle(y)) cout << "Si può usare lo schermo.\n";
  else cout << "In uso.\n";

  x.set_status(INUSE);

  if(x.idle(y)) cout << "Si può usare lo schermo.\n";
  else cout << "In uso.\n";

  return 0;
}
```

Poiché idle() è un membro di C1, può accedere direttamente alla variabile status di oggetti di tipo C1. Pertanto, alla funzione idle() basterà passare oggetti di tipo C2.

Vi sono due importanti restrizioni che si applicano alle funzioni friend. Innanzi tutto, una classe derivata non eredita funzioni friend. In secondo luogo, una funzione friend non può essere dotata di uno specificatore di classe di memorizzazione ovvero non può essere dichiarata come static o extern.

## 12.5 Le classi friend

È anche possibile che un'intera classe sia friend di un'altra classe. In questo caso, la classe friend e tutte le sue funzioni membro avranno accesso ai membri privati definiti all'interno dell'altra classe. Si consideri il seguente esempio.

```
//Uso di una classe friend
#include <iostream>
using namespace std;

class TwoValues {
  int a;
  int b;
public:
  TwoValues(int i, int j) { a = i; b = j; }
  friend class Min;
};

class Min {
public:
  int min(TwoValues x);
};

int Min::min(TwoValues x)
{
  return x.a < x.b ? x.a : x.b;<
}

int main()
{
  TwoValues ob(10, 20);
  Min m;

  cout << m.min(ob);

  return 0;
}
```



Qui, la classe Min ha accesso alle variabili membro a e b nella classe TwoValues.

È fondamentale comprendere che quando una classe è friend di un'altra, ha solamente accesso ai nomi definiti nell'altra classe ma non eredita le caratteristiche dell'altra classe. In particolare i membri della prima classe non divengono membri della classe friend.

Nella pratica, le classi friend vengono raramente impiegate. La loro presenza consente semplicemente di gestire alcune situazioni molto particolari.

## 12.6 Le funzioni inline

In C++ vi è una funzionalità molto importante chiamata *funzione inline*, comunemente impiegata all'interno delle classi. La parte rimanente di questo capitolo (e dell'intera guida) farà largo impiego di questa funzionalità.

In C++ è possibile creare brevi funzioni che non vengono mai effettivamente richiamate; il loro codice viene infatti espanso nel punto in cui dovrebbero essere richiamate e questo le rende simili alle macro-funzioni del C. Per fare in modo che una funzione venga espansa in linea invece che richiamata, si deve far precedere alla sua definizione la parola chiave inline. Ad esempio, nel seguente programma la funzione max() non viene richiamata ma espansa in linea.

```
#include <iostream>
using namespace std;

inline int max(int a, int b)
{
  return a>b ? a : b;
}

int main()
{
  cout << max(10, 20);
  cout << " " << max(99, 88);

  return 0;
}
```

Quindi, per quanto riguarda il compilatore, questo programma sarà equivalente al seguente:

```
#include <iostream>
using namespace std;

int main()
{

  cout << (10>20 ? 10 : 20);
  cout << " " << (99>88 ? 99 : 88);

  return 0;
}
```

Il motivo per cui le funzioni inline sono così importanti è dovuto al fatto che consentono di creare codice molto efficiente. Poiché le classi richiedono normalmente di eseguire con grande frequenza alcune funzioni di interfacciamento (che consentono l'accesso ai dati privati), l'efficienza di queste funzioni è un fattore fondamentale in C++. Come il lettore probabilmente già sa, ogni volta che viene richiamata una funzione, il meccanismo di chiamata e di uscita richiede una certa quantità di tempo. Normalmente, gli argomenti vengono inseriti nello stack e al momento della chiamata vengono salvati vari registri che vengono poi ripristinati all'uscita della funzione. Tutte queste operazioni richiedono tempo. Quando invece una funzione viene espansa in linea, non si verifica nessuna di queste operazioni. D'altra parte, anche se l'espansione delle chiamate a funzioni può produrre codice più veloce, può anche avere influenze negative sulle dimensioni del codice a causa delle duplicazioni richieste. Per questo motivo è consigliabile espandere in linea



solo funzioni molto brevi. Inoltre, è preferibile espandere in linea solo quelle funzioni che hanno un impatto significativo sulle prestazioni del programma.

Come nel caso dello specificatore register, anche inline è una semplice *richiesta* per il compilatore e non un comando. Il compilatore può quindi decidere di ignorare tale richiesta. Inoltre, alcuni compilatori potrebbero rifiutare di espandere in linea alcuni tipi di funzioni. Ad esempio difficilmente un compilatore espanderà in linea funzioni ricorsive. Per conoscere le restrizioni legate all'uso di funzioni inline si deve pertanto consultare la documentazione del compilatore. Se una funzione non può essere espansa in linea, verrà semplicemente richiamata come una comune funzione.

Le funzioni inline possono anche essere funzioni membro di una classe. Ad esempio, questo è un programma C++ perfettamente corretto:

```
#include <iostream>
using namespace std;

class myclass {
  int a, b;
public:
  void init(int i, int j);
  void show();
};

// Crea una funzione inline
inline void myclass::init(int i, int j)
{
  a = i;
  b = j;
}

// Crea un'altra funzione inline
 inline void myclass::show()
{
  cout << a << " " << b << "\n";
}

int main()
{
  myclass x;

  x.init(10, 20);
  x.show();

  return 0;
}
```

## 12.7 Definizione di funzioni inline all'interno di una classe

È possibile definire brevi funzioni anche all'interno della dichiarazione di una classe. Quando una funzione è definita all'interno della dichiarazione di una classe viene automaticamente resa una funzione inline (se possibile). Non è necessario (ma non costituisce un errore) far precedere alla dichiarazione la parola inline. Ad esempio, il programma precedente può essere riscritto inserendo le definizioni di init() e show() all'interno della dichiarazione di myclass.

```
#include <iostream>
using namespace std;

class myclass {
  int a, b;
public:
  // inline automatico
  void init(int i, int j) {a=i; b=j;}
  void show() {cout << a << " " << b << "\n";}
};

int main()
{
  myclass x;

  x.init(10, 20);
  x.show();

  return 0;
}
```

Si noti il formato del codice della funzione all'interno di myclass. Poiché le funzioni inline sono normalmente molto brevi, è piuttosto comune la loro codifica all'interno di una classe. In ogni caso il programmatore è libero di utilizzare il formato desiderato. Ad esempio, la seguente dichiarazione di classe è perfettamente corretta:

```
#include <iostream>
using namespace std;

class myclass {
  int a, b;
public:
  // inline automatico
  void init(int i, int j)
  {
    a = i;
    b = j;
  }
  void show()
  {
    cout << a << " " << b << "\n";
  }
};
```



Tecnicamente, il fatto che la funzione show() sia stata resa inline è ininfluente poiché in generale il tempo richiesto da un'operazione di I/O supera notevolmente l'aggravio di tempo dovuto alla chiamata della funzione. Tuttavia è molto comune vedere tutte le funzioni membro più brevi definite all'interno della rispettiva classe (o meglio è difficile trovare all'interno di programmi C++ professionali funzioni membro brevi definite all'esterno delle rispettive dichiarazioni di classe).

È possibile definire inline anche le funzioni costruttore e distruttore, sfruttando le caratteristiche del linguaggio (se sono definite all'interno delle rispettive classi) oppure tramite definizione esplicita.

## 12.8 I costruttori parametrizzati

Le funzioni costruttore possono ricevere argomenti. Normalmente questi argomenti aiutano a inizializzare un oggetto al momento della creazione. Per creare un costruttore parametrizzato, basta aggiungervi parametri così come si fa con qualsiasi altra funzione. Quando si definisce il corpo del costruttore si possono utilizzare i parametri per inizializzare l'oggetto. Ad esempio, ecco una semplice classe che include un costruttore parametrizzato.

```
#include <iostream>
using namespace std;

class myclass {
  int a, b;
public:
  myclass(int i, int j) {a=i; b=j;}
  void show() {cout << a << " " << b;}
};

int main()
{
```

```
  myclass ob(3, 5);

  ob.show();

  return 0;
}
```

Si noti che nella definizione di myclass() per assegnare i valori iniziali ad a e b vengono utilizzati i parametri i e j.

Il programma illustra il modo più comune per specificare gli argomenti quando si dichiara un oggetto che utilizza una funzione costruttore parametrizzata. In particolare, l'istruzione

```
myclass ob(3, 4);
```

provoca la creazione di un oggetto chiamato ob e passa gli argomenti 3 e 4 ai parametri i e j di myclass(). È possibile passare gli argomenti anche utilizzando questo tipo di istruzione di dichiarazione:

```
myclass ob = myclass(3, 4);
```

Il primo dei due metodi è quello più ampiamente utilizzato ed è anche l'approccio seguito dalla maggior parte degli esempi di questa guida. Vi è una piccola differenza tecnica fra questi due tipi di dichiarazioni che fa riferimento ai costruttori di copie (argomento del Capitolo 14).

Ecco un altro esempio che utilizza una funzione costruttore parametrizzata. Il programma crea una classe che conserva informazioni relative ai libri di una biblioteca.

```
#include <iostream>
#include <cstring>
using namespace std;

const int IN = 1;
const int CHECKED_OUT = 0;

class book {
  char author[40];
  char title[40];
  int status;
public:
  book(char *n, char *t, int s);
  int get_status() {return status;}
```



```
  void set_status(int s) {status = s;}
  void show();
};

book::book(char *n, char *t, int s)
{
  strcpy(author, n);
  strcpy(title, t);
  status = s;
}

void book::show()
{
  cout << title << " di " << author;
  cout << " è ";
  if(status==IN) cout << "presente.\n";
  else cout << "in prestito.\n";
}

int main()
{
  book b1("Dante", "Divina commedia", IN);
  book b2("Manzoni", "I promessi sposi", CHECKED_OUT);

  b1.show();
  b2.show();

  return 0;
}
```

Le funzioni costruttore parametrizzate sono molto utili poiché evitano di dover eseguire una nuova chiamata di funzione semplicemente per inizializzare una o più variabili in un oggetto. Ogni chiamata di funzione evitata renderà il programma più efficiente. Inoltre si noti che le funzioni get_status() e set_status() sono definite all'interno della classe book. Questa è una pratica molto comune in C++.

## Un caso particolare: costruttori con un solo parametro

Se un costruttore ha un solo parametro, vi è un terzo modo per passare un valore iniziale a tale costruttore. Ad esempio, si consideri il seguente programma:

```
#include <iostream>
using namespace std;
```

```
class X {
  int a;
public:
  X(int j) { a = j; }
  int geta() { return a; }
};

int main()
{
  X ob = 99; // passa 99 a j

  cout << ob.geta(); // stampa 99

  return 0;
}
```

Qui il costruttore di X prende un parametro. Si faccia attenzione al modo in cui ob viene dichiarato in main(). In questo tipo di inizializzazione, 99 viene automaticamente passato al parametro j nel costruttore X(). Pertanto, l'istruzione di dichiarazione viene gestita dal compilatore come se fosse scritta nel seguente modo:

```
X ob = X(99);
```

In generale, ogni volta che un costruttore richiede un solo argomento, si può inizializzare un oggetto con ob(i) oppure ob = i. Il motivo è che quando si crea un costruttore che accetta un argomento, si crea implicitamente una funzione di conversione dal tipo dell'argomento al tipo della classe. Si ricordi che l'alternativa appena illustrata si applica solo ai costruttori che hanno un solo parametro.

## 12.9 I membri static di una classe

Le funzioni e i dati che sono membri di una classe possono essere resi static. Questa sezione spiega cosa ciò significhi per ogni tipo di membro.

### Dati membri static

Quando la dichiarazione di una variabile membro è preceduta dalla parola chiave static si chiede al compilatore di creare una sola copia di tale variabile e di utilizzare tale copia per tutti gli oggetti della classe. A differenza dei comuni dati membri, non verrà creata una singola copia di una variabile membro static per ogni



oggetto. Indipendentemente dal numero di oggetti creati di una classe, esisterà una sola copia dei dati membri static. Pertanto, tutti gli oggetti di tale classe utilizzeranno la stessa variabile. Tutte le variabili static vengono inizializzate a zero nel momento in cui viene creato il primo oggetto.

Quando si dichiarano i dati membri static all'interno di una classe, non si definiscono tali dati. Questo significa che non si sta allocando spazio di memoria per tali dati (in C++, una dichiarazione *descrive* qualcosa e una definizione *crea* qualcosa), Si dovrà pertanto fornire una definizione globale per i dati membri static in un altro punto, all'esterno della classe. Questo può essere ottenuto dichiarando la variabile come static e utilizzando l'operatore di risoluzione del campo d'azione per identificare la classe di appartenenza. Questo provoca l'allocazione di memoria per la variabile (si ricordi che la dichiarazione di una classe non è che un costrutto logico che non ha realtà fisica).

Per comprendere l'uso e gli effetti dei dati membri static, si consideri questo programma:

```
#include <iostream>
using namespace std;

class shared {
  static int a;
  int b;
public:
  void set(int i, int j) {a=i; b=j;}
  void show();
} ;

int shared::a; // definisce a

void shared::show()
{
  cout << "Variabile statica a: " << a;
  cout << "\nVariabile non statica b: " << b;
  cout << "\n";
}

int main()
{
  shared x, y;

  x.set(1, 1); // assegna 1 alla variabile a
  x.show();

  y.set(2, 2); // ora le assegna 2
```

```
  y.show();

  x.show(); /* Qui, a è stata modificata sia per x che per y
               poiché a è condivisa da entrambi gli oggetti. */

  return 0;
}
```

Questo programma produce il seguente output:

```
Variabile statica a: 1
Variabile non statica b: 1
Variabile statica a: 2
Variabile non statica b: 2
Variabile statica a: 2
Variabile non statica b: 1
```

Si noti che l'intero a è dichiarato sia all'interno di shared che al suo esterno. Come si è detto precedentemente questo è necessario poiché la dichiarazione di a all'interno di shared non alloca memoria per la variabile.

**NOTA** *Per comodità, le prime versioni di C++ non richiedevano la seconda dichiarazione di una variabile membro static. Tuttavia questa comodità dava origine a gravi incongruenze e fu eliminata molti anni fa. In ogni caso si potrebbe trovare codice C++ non molto recente che non esegue la ridichiarazione delle variabili membro static. In questi casi sarà necessario aggiungere le definizioni richieste.*

Una variabile membro static esiste *prima* che venga creato qualsiasi oggetto della sua classe. Ad esempio, nel seguente breve programma, a è sia public che static. In questo modo main() può accedervi direttamente. Inoltre, poiché a esiste prima della creazione di qualsiasi oggetto della classe shared, sarà possibile assegnare un valore ad a in qualsiasi momento. Come si può vedere nel seguente programma, il valore di a non viene modificato dalla creazione dell'oggetto x. Per questo motivo, entrambe le istruzioni di output visualizzano lo stesso valore: 99.

```
#include <iostream>
using namespace std;

class shared {
public:
  static int a;
} ;
```



```
int shared::a; // definisce a

int main()
{
  // inizializza a prima di creare qualsiasi oggetto
  shared::a = 99;

  cout << "Questo è il valore iniziale di a: " << shared::a;
  cout << "\n";

  shared x;

  cout << "Questo è x.a: " << x.a;

  return 0;
}
```

Si noti come il riferimento ad a avvenga tramite l'uso del nome della classe e dell'operatore di risoluzione del campo d'azione. In generale, quando il programma fa riferimento a un membro static indipendentemente da un oggetto, si deve qualificare il membro static utilizzando il nome della classe di cui è membro.

Uno degli utilizzi delle variabili membro static consiste nel fornire un controllo per l'accesso ad alcune risorse condivise utilizzate da tutti gli oggetti della classe. Ad esempio, si potrebbero creare più oggetti, ognuno dei quali deve eseguire operazioni di scrittura su un determinato file su disco. È chiaro però che un solo oggetto potrà scrivere sul file in un determinato momento. In questo caso, si potrebbe voler dichiarare una variabile static che indichi quando il file è in uso e quando è disponibile. Prima di iniziare a scrivere sul file ogni oggetto potrà quindi interrogare questa variabile. Il programma seguente mostra questo uso di una variabile static per controllare l'accesso a una risorsa condivisa.

```
#include <iostream>
using namespace std;

class cl {
  static int resource;
public:
  int get_resource();
  void free_resource() {resource = 0;}
};

int cl::resource; // definisce la risorsa

int cl::get_resource()
```

```
{
  if(resource) return 0; // la risorsa è già in uso
  else {
    resource = 1;
    return 1;  // la risorsa è allocata a questo oggetto
  }
}

int main()
{
  cl ob1, ob2;

  if(ob1.get_resource()) cout << "la risorsa è di ob1\n";

  if(!ob2.get_resource()) cout << "ob2 non può utilizzare la risorsa\n";

  ob1.free_resource();  // la risorsa viene liberata

  if(ob2.get_resource())
    cout << "ora ob2 può usare la risorsa\n";

  return 0;
}
```

Un altro interessante uso di una variabile membro static consiste nel registrare il numero di oggetti esistenti di una determinata classe. Ad esempio:

```
#include <iostream>
using namespace std;

class Counter {
public:
  static int count;
  Counter() { count++; }
  ~Counter() { count--; }
};
int Counter::count;

void f();

int main(void)
{
  Counter o1;
  cout << "Oggetti esistenti: ";
  cout << Counter::count << "\n";
```



```
  Counter o2;
  cout << "Oggetti esistenti: ";
  cout << Counter::count << "\n";

  f();
  cout << "Oggetti esistenti: ";
  cout << Counter::count << "\n";

  return 0;
}
void f()
{
  Counter temp;
  cout << "Oggetti esistenti: ";
  cout << Counter::count << "\n";
  // temp viene distrutta all'uscita da f()
}
```

Questo programma produce il seguente output:

```
Oggetti esistenti: 1
Oggetti esistenti: 2
Oggetti esistenti: 3
Oggetti esistenti: 2
```

Come si può vedere, la variabile membro static count viene incrementata ogni volta che viene creato un oggetto e decrementata quando viene distrutto un oggetto. In questo modo registra sempre il numero di oggetti Counter esistenti.

L'impiego di variabili membro static dovrebbe consentire di eliminare la necessità di utilizzare variabili globali. Il problema derivante dall'uso di variabili globali in tecniche di programmazione a oggetti consiste nel fatto che quasi sempre esse violano il principio di incapsulamento.

## Funzioni membro static

Anche le funzioni membro possono essere dichiarate static. Vi sono però molte restrizioni relative all'impiego di funzioni membro static. Innanzi tutto, tali funzioni possono accedere solo a membri static della classe (naturalmente le funzioni membro static possono accedere a tutte le funzioni e ai dati globali). In secondo luogo, le funzioni membro static non possono avere un puntatore this (per informazioni consultare il Capitolo 13). Infine non possono esistere versioni static e non static della stessa funzione. Una funzione membro static non può essere virtuale e non può essere dichiarata come const o volatile.

Di seguito viene presentata una versione leggermente modificata del programma a risorse condivise della sezione precedente. Si noti che ora get_resource() è dichiarata come static. Come viene illustrato nel programma, l'accesso a get_resource() può avvenire da se stessa (indipendentemente dagli oggetti che utilizzano il nome della classe e l'operatore di risoluzione del campo d'azione) oppure in connessione con un oggetto.

```
#include <iostream>
using namespace std;

class cl {
  static int resource;
public:
  static int get_resource();
  void free_resource() {resource = 0;}
};

int cl::resource; // definisce la risorsa

int cl::get_resource()
{
  if(resource) return 0; // la risorsa è già in uso
  else {
    resource = 1;
    return 1;  // la risorsa è allocata a questo oggetto
  }
}

int main()
{
  cl ob1, ob2;

  /* get_resource() è static per poter essere richiamata in modo indipendente
     da qualsiasi oggetto. */
  if(cl::get_resource()) cout << "la risorsa è di ob1\n";

  if(!cl::get_resource()) cout << "ob2 non può utilizzare la risorsa\n";

  ob1.free_resource();

  if(ob2.get_resource()) // può essere richiamata utilizzando
la sintassi degli oggetti
    cout << "ora ob2 può usare la risorsa\n";
```



```
  return 0;
}
```

In realtà, le funzioni membro static hanno applicazioni piuttosto limitate ma sono ad esempio utili per "preinizializzare" i dati privati static prima della creazione di qualsiasi oggetto. Ad esempio, questo è un programma C++ perfettamente corretto:

```
#include <iostream>
using namespace std;

class static_type {
  static int i;
public:
  static void init(int x) {i = x;}
  void show() {cout << i;}
};

int static_type::i; // definisce i

int main()
{
  // inizializza i dati static prima della creazione dell'oggetto
  static_type::init(100);

  static_type x;
  x.show(); // visualizza 100

  return 0;
}
```

## 12.10 Quando vengono eseguiti i costruttori e i distruttori?

Come regola generale, il costruttore di un oggetto viene richiamato nel momento in cui l'oggetto inizia ad esistere mentre il distruttore dell'oggetto viene richiamato nel momento in cui l'oggetto deve essere distrutto. Ma quando hanno luogo esattamente queste operazioni?

La funzione costruttore di un oggetto locale viene eseguita nel momento in cui viene incontrata l'istruzione di dichiarazione dell'oggetto. Le funzioni distruttore per gli oggetti locali vengono eseguite in ordine inverso rispetto alle funzioni costruttore.

Le funzioni costruttore degli oggetti globali vengono eseguite *prima* che inizi l'esecuzione di main(). I costruttori globali vengono eseguiti nell'ordine di dichiarazione nel file. Non è possibile conoscere l'ordine di esecuzione dei costruttori globali specificati all'interno di vari file. I distruttori globali vengono eseguiti in ordine inverso *dopo* il termine dell'esecuzione di main().

Il seguente programma illustra l'esecuzione dei costruttori e dei distruttori.

```
#include <iostream>
using namespace std;

class myclass {
public:
  int who;
  myclass(int id);
  ~myclass();
} glob_ob1(1), glob_ob2(2);

myclass::myclass(int id)
{
  cout << "Inizializzazione di " << id << "\n";
  who = id;
}

myclass::~myclass()
{
  cout << "Distruzione di " << who << "\n";
}

int main()
{
  myclass local_ob1(3);

  cout << "Questa non sarà la prima riga visualizzata.\n";

  myclass local_ob2(4);

  return 0;
}
```

Il programma produce il seguente output:

```
Inizializzazione di 1
Inizializzazione di 2
Inizializzazione di 3
```



```
Questa non sarà la prima riga visualizzata.
Inizializzazione di 4
Distruzione di 4
Distruzione di 3
Distruzione di 2
Distruzione di 1
```

A causa delle differenze esistenti fra compilatori ed ambienti operativi, le ultime due righe potrebbero non venire visualizzate.

## 12.11 L'operatore di risoluzione del campo d'azione

L'operatore :: consente di collegare il nome di una classe con il nome di un membro per comunicare al compilatore la classe a cui appartiene il membro. L'operatore di risoluzione del campo d'azione ha però un altro utilizzo: consente infatti di accedere a un nome che si trova all'interno del campo di visibilità e che è nascosto da una dichiarazione locale avente lo stesso nome. Ad esempio, si consideri il seguente frammento di codice:

```
.
.
.
int i;  // i globale

void f()
{
  int i; // i locale

  i = 10; // usa la i locale
  .
  .
  .
}

.
.
.
```

Come dice il commento, l'assegnamento i = 10 fa riferimento alla variabile i locale. Ma cosa accade se la funzione f() deve accedere alla versione globale di i? Basterà far precedere alla variabile i l'operatore ::

```
  .
  .
  .

int i;  // i globale

void f()
{
  int i; // i locale

  ::i = 10; // ora fa riferimento alla i globale
  .
  .
  .
}

.
.
.
```

## 12.12 La nidificazione delle classi

È possibile definire una classe all'interno di un'altra definendo una *classe nidificata*. Poiché la dichiarazione di una classe definisce a tutti gli effetti le regole di visibilità, una classe nidificata è valida solo all'interno del campo d'azione della classe che la racchiude. In realtà, l'uso di classi nidificate è molto limitato. Grazie alla flessibilità e alla potenza del meccanismo di ereditarietà del C++, in pratica non vi è alcun bisogno di utilizzare classi nidificate.

## 12.13 Le classi locali

Una classe può essere definita all'interno di una funzione. Ad esempio, questo è un programma C++ perfettamente corretto:

```
#include <iostream>
using namespace std;

void f();

int main()
{
```



```
  f();
  // myclass non è nota in questo punto
  return 0;
}

void f()
{
  class myclass {
    int i;
  public:
    void put_i(int n) {i=n;}
    int get_i() {return i;}
  } ob;

  ob.put_i(10);
  cout << ob.get_i();
}
```

Quando si dichiara una classe all'interno di una funzione, la classe sarà nota solo all'interno di tale funzione e non sarà utilizzabile all'esterno.

Le classi locali sono soggette a notevoli restrizioni. Innanzi tutto, tutte le funzioni membro devono essere definite all'interno della dichiarazione della classe. La classe locale non dovrebbe utilizzare né effettuare accessi alle variabili locali della funzione in cui è dichiarata (ma una classe locale può avere accesso alle variabili locali static dichiarate all'interno della funzione e a quelle dichiarate con extern). Tuttavia può accedere ai nomi di tipi e agli enumeratori definiti nella funzione in cui è contenuta. Nessuna variabile static può essere dichiarata all'interno di una classe locale. A causa di queste restrizioni, l'impiego delle classi locali non è molto comune in C++.

## 12.14 Il passaggio di oggetti a funzioni

Gli oggetti possono essere passati alle funzioni come qualsiasi altro tipo di variabile. In particolare gli oggetti vengono passati alle funzioni utilizzando il classico meccanismo di chiamata per valore. Questo significa che in realtà alla funzione viene passata una copia dell'oggetto. Ma questo a sua volta significa che viene in effetti creato un nuovo oggetto. Sorgono spontanee due domande: quando viene creata la copia dell'oggetto viene richiamata la funzione costruttore dell'oggetto? E quando la copia viene distrutta viene richiamata la funzione distruttore? La risposta a queste due domande può essere per certi versi sorprendente. Per iniziare, ecco un breve esempio:

```
// Passaggio di un oggetto a una funzione

#include <iostream>
using namespace std;

class myclass {
  int i;
public:
  myclass(int n);
  ~myclass();
  void set_i(int n) {i=n;}
  int get_i() {return i;}
};

myclass::myclass(int n)
{
  i = n;
  cout << "Costruzione di " << i << "\n";
}

myclass::~myclass()
{
  cout << "Distruzione di " << i << "\n";
}

void f(myclass ob);

int main()
{
  myclass o(1);

  f(o);
  cout << "Questa è la i di main: ";
  cout << o.get_i() << "\n";

  return 0;
}

void f(myclass ob)
{
  ob.set_i(2);

  cout << "Questa è la i locale: " << ob.get_i();
  cout << "\n";
}
```



Questo programma produce il seguente output:

```
Costruzione di 1
Questa è la i locale: 2
Distruzione di 2
Questa è la i di main: 1
Distruzione di 1
```

Si noti che vengono eseguite due chiamate alla funzione distruttore e una sola chiamata alla funzione costruttore. Come si può vedere dall'output, quando a ob (all'interno di f()) viene passata la copia di o (in main()), non viene richiamata la funzione costruttore. Il motivo per cui non viene chiamata una funzione costruttore quando deve essere prodotta una copia dell'oggetto è molto semplice: quando si passa un oggetto a una funzione, si intende lo stato corrente di tale oggetto. Se venisse richiamato il costruttore per creare la copia, avverrebbe una completa inizializzazione dell'oggetto che nel frattempo potrebbe essere stato modificato. Pertanto, quando viene generata la copia di un oggetto per la chiamata di una funzione non può essere eseguita la funzione costruttore.

Anche se quando un oggetto viene passato a una funzione non viene richiamata la funzione costruttore, è necessario invece richiamare il distruttore nel momento in cui la copia deve essere distrutta (la copia viene distrutta come qualsiasi altra variabile locale nel momento in cui termina la funzione). Si ricordi che la copia dell'oggetto esiste fintantoché la funzione sarà in esecuzione. Questo significa che la copia potrebbe eseguire operazioni che rendono necessaria la distruzione da parte della funzione distruttore. Ad esempio, la copia dell'oggetto potrebbe allocare memoria che dovrà essere liberata al momento della distruzione. Per questo motivo, per distruggere la copia è necessario richiamare la funzione distruttore.

Per riassumere: quando viene generata una copia di un oggetto per consentire il passaggio di tale oggetto a una funzione, non viene richiamata la funzione costruttore dell'oggetto. Quando invece deve essere distrutta la copia dell'oggetto all'interno della funzione, viene richiamata la funzione distruttore.

Normalmente, la copia di un oggetto è una copia bit per bit. Questo significa che ogni nuovo oggetto è una copia identica dell'originale. Ma ciò può in alcuni casi dare origine a problemi. Anche se gli oggetti vengono passati alle funzioni tramite il normale meccanismo del passaggio per valore, che in teoria protegge e isola l'argomento chiamante, è comunque possibile che si verifichi un effetto collaterale che modifichi e persino distrugga l'oggetto utilizzato come argomento. Ad esempio, se l'oggetto utilizzato come argomento alloca memoria che viene poi liberata al momento della distruzione della copia locale dell'oggetto che si trova all'interno della funzione, verrà liberata la stessa area di memoria. Questo provocherà un danneggiamento dell'oggetto originario che diverrà inutilizzabile.

Come si vedrà nel Capitolo 14, è possibile evitare questo genere di problemi definendo un'operazione di copia relativa a una determinata classe creando un tipo particolare di costruttore chiamato *costruttore di copie*.

## 12.15 La restituzione di oggetti

Una funzione può restituire al chiamante un oggetto. Ad esempio, questo è un programma C++ perfettamente corretto:

```
// Restituzione di oggetti da parte di una funzione

#include <iostream>
using namespace std;

class myclass {
  int i;
public:
  void set_i(int n) {i=n;}
  int get_i() {return i;}
};

myclass f(); // restituisce un oggetto di tipo myclass

int main()
{
  myclass o;

  o = f();

  cout << o.get_i() << "\n";

  return 0;
}

myclass f()
{
  myclass x;

  x.set_i(1);
  return x;
}
```



Quando una funzione restituisce un oggetto, viene automaticamente creato un oggetto temporaneo che contiene il valore restituito. La funzione quindi restituisce in effetti questo secondo oggetto (temporaneo). Dopo la restituzione del valore, l'oggetto viene distrutto. La distruzione di questo oggetto temporaneo può in alcuni casi provocare effetti collaterali indesiderati. Ad esempio, se l'oggetto restituito dalla funzione ha un distruttore che libera la memoria allocata dinamicamente, tale memoria verrà liberata anche se l'oggetto che riceve il valore restituito continuerà a utilizzarla. È possibile risolvere questo problema utilizzando tecniche di overloading dell'operatore di assegnamento (vedere il Capitolo 15) e definendo un costruttore di copie (vedere il Capitolo 14).

## 12.16 L'assegnamento di oggetti

Assumendo che entrambi gli oggetti siano dello stesso tipo, è possibile assegnare un oggetto a un altro oggetto. In questo modo i dati dell'oggetto che si trova sul lato destro del segno di uguaglianza verranno copiati nei dati dell'oggetto che si trova a sinistra. Ad esempio, il seguente programma visualizza il valore 99.

```
// Assegnamento di oggetti

#include <iostream>
using namespace std;

class myclass {
  int i;
public:
  void set_i(int n) {i=n;}
  int get_i() {return i;}
};

int main()
{
  myclass ob1, ob2;

  ob1.set_i(99);
  ob2 = ob1; // assegna i dati da ob1 a ob2

  cout << "questa è la i di ob2: " << ob2.get_i();

  return 0;
}
```

Normalmente tutti i dati dell'oggetto di destra vengono assegnati all'oggetto di sinistra utilizzando una copia bit-a-bit. È però possibile eseguire l'overloading dell'operatore di assegnamento e definire altre procedure di assegnamento (vedere il Capitolo 15).

Capitolo 13

# Gli array, i puntatori, gli indirizzi e gli operatori di allocazione dinamica


13.1 **Gli array di oggetti**
13.2 **I puntatori a oggetti**
13.3 **Verifiche di tipo sui puntatori C++**
13.4 **Il puntatore this**
13.5 **I puntatori a tipi derivati**
13.6 **I puntatori ai membri di una classe**
13.7 **Gli indirizzi**
13.8 **Questione di stile**
13.9 **Gli operatori di allocazione dinamica del C++**


ella Parte prima è stato discusso l'argomento dei puntatori relativi ai tipi standard C++. Questo capitolo si occupa dei puntatori e degli indirizzi di memoria. Al termine del capitolo si troverà una discussione riguardante gli operatori di allocazione dinamica.

## 13.1 Gli array di oggetti

In C++ è possibile creare array di oggetti. La sintassi per la dichiarazione e l'uso di array di oggetti è esattamente la stessa già vista per altri tipi di array. Ad esempio, il seguente programma utilizza un array di tre oggetti:

```
#include <iostream>
using namespace std;

class cl {
  int i;
public:
  void set_i(int j) {i=j;}
  int get_i() {return i;}
};
```

```
int main()
{
  c1 ob[3];
  int i;

  for(i=0; i<3; i++) ob[i].set_i(i+1);

  for(i=0; i<3; i++)
    cout << ob[i].get_i() << "\n";

  return 0;
}
```

Questo programma visualizza sullo schermo i numeri 1, 2 e 3.

Se una classe definisce un costruttore parametrizzato, è possibile inizializzare ogni oggetto di un array specificando una lista di inizializzazione così come si fa per altri tipi di array. Tuttavia, la forma della lista di inizializzazione verrà decisa dal numero di parametri richiesti dalla funzione costruttore dell'oggetto. Nel caso di oggetti i cui costruttori richiedano un solo parametro, basta specificare un elenco dei valori iniziali utilizzando la normale sintassi di inizializzazione degli array. Ogni valore della lista verrà passato, nell'ordine, alla funzione costruttore per la creazione di ciascun elemento dell'array. Ad esempio, ecco una versione leggermente diversa del programma precedente che fa uso dell'inizializzazione:

```
#include <iostream>
using namespace std;

class c1 {
  int i;
public:
  c1(int j) {i=j;}  // costruttore
  int get_i() {return i;}
};

int main()
{
  c1 ob[3] = {1, 2, 3};  // inizializzatori
  int i;

  for(i=0; i<3; i++)
    cout << ob[i].get_i() << "\n";

  return 0;
}
```



Anche questo programma visualizza i numeri 1, 2 e 3.

In realtà, la sintassi di inizializzazione utilizzata nel programma precedente è una versione abbreviata della seguente forma:

```
c1 ob[3] =  c1(1), c1(2), c1(3) ;
```

Qui viene richiamato esplicitamente il costruttore di c1. Naturalmente è molto più comune trovare la forma abbreviata utilizzata nel programma. La forma abbreviata funziona grazie alla conversione automatica che si applica ai costruttore che accettano un solo argomento (vedere il Capitolo 12). Pertanto la forma abbreviata può essere utilizzata solo per inizializzare gli array di oggetti i cui costruttori richiedono un solo argomento.

Se il costruttore di un oggetto richiede due o più argomenti, sarà necessario ricorrere alla forma di inizializzazione più estesa. Ad esempio:

```
#include <iostream>
using namespace std;

class c1 {
  int h;
  int i;
public:
  c1(int j, int k) { h=j; i=k; } // costruttore con due parametri
  int get_i() {return i;}
  int get_h() {return h;}
};

int main()
{
  c1 ob[3] = {
    c1(1, 2),
    c1(3, 4),
    c1(5, 6)
  }; // inizializzatori

  int i;

  for(i=0; i<3; i++) {
    cout << ob[i].get_h();
    cout << ", ";
    cout << ob[i].get_i() << "\n";
  }

  return 0;
}
```

In questo esempio, il costruttore di cl ha due parametri e, pertanto, richiede due argomenti. Questo significa che non è possibile utilizzare la forma di inizializzazione "abbreviata" e sarà necessario utilizzare la forma estesa mostrata nell'esempio.

### Creazione di array inizializzati e non inizializzati

Una situazione particolare si verifica quando si cerca di creare array di oggetti in parte inizializzati e in parte non inizializzati. Si consideri la seguente classe:

```
class cl {
  int i;
public:
  cl(int j) {i=j;}
  int get_i() {return i;}
};
```

Qui, la funzione costruttore definita da cl richiama un parametro. Questo significa che qualsiasi array dichiarato di questo tipo dovrà essere inizializzato ma anche che non è possibile utilizzare la seguente dichiarazione di array:

```
cl a[9]; // errore, il costruttore richiede l'uso di inizializzatori
```

Questa istruzione non è corretta (così com'è attualmente definita cl) poiché implica che a cl sia associato un costruttore senza parametri in quanto non viene specificato alcun inizializzatore. Ma come si può vedere cl non ha un costruttore senza parametri. Poiché non vi è alcun costruttore valido che corrisponda a questa dichiarazione, il compilatore presenterà un messaggio d'errore.

Per risolvere questo problema si deve eseguire l'overloading della funzione costruttore aggiungendone una versione che non richieda parametri. In questo modo sarà possibile creare array inizializzati e non inizializzati. Ecco una nuova versione di cl:

```
class cl {
  int i;
public:
  cl() {i=0;}  // richiamata per array non inizializzati
  cl(int j) {i=j;}  // richiamata per array inizializzati
  int get_i() {return i;}
};
```



Data questa classe, sarà consentito l'uso di entrambe le istruzioni seguenti:

```
cl a1[3] = {3, 5, 6}; // inizializzato
cl a2[34]; // non inizializzato
```

## 13.2 I puntatori a oggetti

Così come è possibile definire puntatori ad altri tipi di variabili, è possibile anche definire puntatori a oggetti. Quando si deve accedere a un membro di una classe dato un puntatore a un oggetto, al posto dell'operatore punto si utilizza l'operatore freccia (->). Il programma seguente illustra l'accesso a un oggetto tramite un puntatore:

```
#include <iostream>
using namespace std;

class cl {
  int i;
public:
  cl(int j) {i=j;}
  int get_i() {return i;}
};

int main()
{
  cl ob(88), *p;

  p = &ob;  // legge l'indirizzo di ob

  cout << p->get_i(); // usa -> per richiamare get_i()

  return 0;
}
```

Quando si incrementa un puntatore, questo punterà all'elemento successivo dello stesso tipo. Ad esempio, un puntatore a interi punterà all'intero successivo. In generale, tutta l'aritmetica dei puntatori si basa sul tipo dell'elemento puntato dal puntatore (ovvero sul tipo dei dati specificato al momento della dichiarazione del puntatore). La stessa regola vale anche per i puntatori a oggetti. Ad esempio, il seguente programma utilizza un puntatore per accedere ai tre elementi dell'array ob dopo che a ob è stato assegnato un indirizzo iniziale.

```
#include <iostream>
using namespace std;

class cl {
  int i;
public:
  cl() {i=0;}
  cl(int j) {i=j;}
  int get_i() {return i;}
};

int main()
{
  cl ob[3] = {1, 2, 3};
  cl *p;
  int i;

  p = ob;  // punta all'inizio dell'array
  for(i=0; i<3; i++) {
    cout << p->get_i() << "\n";
    p++; // punta all'oggetto successivo
  }

  return 0;
}
```

È possibile assegnare a un puntatore l'indirizzo di un membro pubblico di un oggetto e poi utilizzare il puntatore per accedere a tale membro. Ad esempio, il seguente programma, perfettamente corretto in C++, visualizza sullo schermo il numero 1:

```
#include <iostream>
using namespace std;

class cl {
public:
  int i;
  cl(int j) {i=j;}
};

int main()
{
  cl ob(1);
  int *p;
```



```
  p = &ob.i;  // legge l'indirizzo di ob.i

  cout << *p; // accede a ob.i tramite p

  return 0;
}
```

Poiché p punta a un intero, viene dichiarato come puntatore a interi. In questa situazione, il fatto che i sia un membro dell'oggetto ob è irrilevante.

## 13.3 Verifiche di tipo sui puntatori C++

Vi è un fatto importantissimo da comprendere relativo all'uso dei puntatori in C++: è possibile eseguire un assegnamento da un puntatore a un altro solo se i tipi dei due puntatori sono compatibili. Dati i puntatori:

```
int *pi;
float *pf;
```

in C++ il seguente assegnamento non è consentito:

```
pi = pf; // errore per differenza di tipo
```

Naturalmente è possibile bypassare le incompatibilità di tipo utilizzando una conversione cast ma in questo modo verrà prodotta una violazione del meccanismo di verifica dei tipi del C++.

**NOTA** *Le forti verifiche di tipo che il C++ applica ai puntatori rappresentano una differenza fondamentale rispetto al C in cui è possibile assegnare a un puntatore un valore qualsiasi.*

## 13.4 Il puntatore this

Quando viene richiamata una funzione membro, le viene automaticamente passato un argomento implicito costituito da un puntatore all'oggetto chiamante (ovvero l'oggetto su cui viene richiamata la funzione). Questo puntatore è chiamato this. Per comprendere il significato del puntatore this si consideri innanzi tutto un puntatore che crea una classe chiamata pwr la quale calcoli il risultato di una base elevata a un esponente:

```
#include <iostream>
using namespace std;

class pwr {
  double b;
  int e;
  double val;
public:
  pwr(double base, int exp);
  double get_pwr() {return val;}
};

pwr::pwr(double base, int exp)
{
  b = base;
  e = exp;
  val = 1;
  if(exp==0) return;
  for( ; exp>0; exp--) val = val * b;
}

int main()
{
  pwr x(4.0, 2), y(2.5, 1), z(5.7, 0);

  cout << x.get_pwr() << " ";
  cout << y.get_pwr() << " ";
  cout << z.get_pwr() << "\n";
  return 0;}
```

All'interno di una funzione membro l'accesso ai membri della classe avviene direttamente senza che sia necessario qualificare l'oggetto o la classe. Pertanto, all'interno di pwr() l'istruzione:

```
b = base;
```

significa che alla copia di b associata all'oggetto chiamante viene assegnato il valore contenuto in base. Si sarebbe potuta scrivere la stessa istruzione nel seguente modo:

```
this->b = base;
```

Il puntatore this punta all'oggetto che ha richiamato pwr(). Pertanto, con this->b si fa riferimento alla copia di b contenuta in tale oggetto. Se pwr() fosse stata



richiamata da x (ad esempio con x(4.0, 2)), il puntatore this dell'istruzione precedente avrebbe puntato a x. È bene ricordare che tralasciando il puntatore this si utilizza in effetti una forma abbreviata dell'istruzione.

Ecco l'aspetto della funzione pwr() facendo uso del puntatore this:

```
pwr::pwr(double base, int exp)
{
  this->b = base;
  this->e = exp;
  this->val = 1;
  if(exp==0) return;
  for( ; exp>0; exp--)
    this->val = this->val * this->b;
}
```

Nessun programmatore C++ scriverebbe mai la funzione pwr() in questo secondo modo poiché in realtà non si guadagna nulla e la forma abbreviata è più semplice. Tuttavia, il puntatore this è molto importante nel caso di overloading degli operatori e in tutti i casi in cui una funzione membro debba utilizzare un puntatore all'oggetto che l'ha richiamata.

Il puntatore this viene passato automaticamente a tutte la funzioni membro. Pertanto, get_pwr() potrebbe essere riscritta anche nel seguente modo:

```
double get_pwr() {return this->val;}
```

In questo caso, se get_pwr() fosse richiamata nel seguente modo:

```
y.get_pwr();
```

this punterebbe all'oggetto y.

Due ultime annotazioni relative al puntatore this. Innanzi tutto, le funzioni friend non sono membri di una classe e pertanto ad esse non viene passato alcun puntatore this. In secondo luogo, le funzioni membro static non hanno alcun puntatore this.

## 13.5 I puntatori a tipi derivati

In generale, un puntatore di un determinato tipo non può puntare a un oggetto di un tipo differente. Vi è però un'importante eccezione a questa regola che riguarda solo le classi derivate. Si assuma di utilizzare due classi chiamate B e D e che D

derivi dalla classe base B. In questa situazione, un puntatore di tipo B* può anche puntare a un oggetto di tipo D. In senso più generale, un puntatore a una classe base può anche essere utilizzato come puntatore a un oggetto di una qualsiasi classe derivata da tale classe base.

Anche se un puntatore alla classe base può essere utilizzato per puntare a un oggetto derivato, non è possibile applicare la regola inversa. Un puntatore di tipo D* non può puntare a un oggetto di tipo B. Inoltre, anche se è possibile utilizzare un puntatore base per puntare a un oggetto derivato, sarà possibile accedere solo ai membri del tipo derivato che sono stati importati dalla classe base. Quindi non si sarà in grado di accedere ai membri aggiunti dalla classe derivata (è però possibile avere accesso all'intera classe derivata eseguendo una conversione cast del puntatore alla classe base, trasformandolo quindi in un puntatore alla classe derivata).

Ecco un breve programma che usa un puntatore alla classe base per accedere agli oggetti derivati.

```
#include <iostream>
using namespace std;

class base {
  int i;
public:
  void set_i(int num) {i=num;}
  int get_i() {return i;}
};

class derived: public base {
  int j;
public:
  void set_j(int num) {j=num;}
  int get_j() {return j;}
};

int main()
{
  base *bp;
  derived d;

  bp = &d; // il puntatore base punta all'oggetto derivato

  // accesso all'oggetto derivato utilizzando il puntatore base
  bp->set_i(10);
  cout << bp->get_i() << " ";

/* Questo non funziona. Non è possibile accedere a un elemento di
   una classe derivata utilizzando un puntatore alla classe base.
   bp->set_j(88);  // errore
   cout << bp->get_j(); // errore

*/
  return 0;
}
```



Come si può vedere, per accedere a un oggetto di una classe derivata si utilizza un puntatore base.

Anche se questa tecnica richiede attenzione, essa consente di eseguire una conversione cast del puntatore base in un puntatore alla classe derivata in modo da accedere ai membri della classe derivata utilizzando il puntatore base. Ad esempio, questo codice C++ è perfettamente corretto:

```
// accesso consentito grazie al cast
((derived *)bp)->set_j(88);
cout << ((derived *)bp)->get_j();
```

È importante ricordare che l'aritmetica dei puntatori fa riferimento al puntatore base. Per questo motivo, quando un puntatore base punta a un oggetto derivato, l'incremento del puntatore non fa in modo che questo punti all'oggetto successivo del tipo derivato. Al contrario il puntatore punterà a ciò che dovrebbe essere l'oggetto successivo nel tipo base. Questo in genere dà origine a problemi. Ad esempio, il seguente programma anche se è sintatticamente corretto contiene questo errore logico.

```
#include <iostream>
using namespace std;

class base {
  int i;
public:
  void set_i(int num) {i=num;}
  int get_i() {return i;}
};

class derived: public base {
  int j;
public:
  void set_j(int num) {j=num;}
  int get_j() {return j;}
};
```

```
int main()
{
  base *bp;
  derived d[2];

  bp = d;

  d[0].set_i(1);
  d[1].set_i(2);

  cout << bp->get_i() << " ";
  bp++;  // rispetto alla classe base e non alla classe derivata
  cout << bp->get_i();  // viene visualizzato un valore senza senso

  return 0;
}
```

L'uso di puntatori base a tipi derivati è utilissimo nella realizzazione del polimorfismo run-time mediante il meccanismo delle funzioni virtuali (vedere il Capitolo 17).

## 13.6 I puntatori ai membri di una classe

Il C++ consente di generare un tipo particolare di puntatore che punta genericamente a un membro di una classe e non a una specifica istanza di tale membro in un oggetto. Questo genere di puntatore è chiamato puntatore a un membro della classe o *puntatore a membro*. Un puntatore a membro non è la stessa cosa di un comune puntatore C++. Esso infatti fornisce solo un valore di scostamento all'interno di un oggetto appartenente alla classe del membro in cui è possibile trovare tale membro. Poiché i puntatori a membro non sono veri puntatori, non è possibile applicarvi gli operatori . e ->. Per accedere a un membro di una classe dato un puntatore ad esso, si deve utilizzare uno degli operatori specifici dei puntatori a membri ovvero .* e ->*. Lo scopo di questi operatori è di consentire l'accesso ai membri di una classe dato un puntatore a tale membro.

Ecco un esempio:

```
#include <iostream>
using namespace std;

class cl {
public:
  cl(int i) {val=i;}
```



```
  int val;
  int double_val() {return val+val;}
};

int main()
{
  int cl::*data; // puntatore a membro (dati)
  int (cl::*func)(); // puntatore a membro (funzione)
  cl ob1(1), ob2(2); // crea gli oggetti

  data = &cl::val; // calcola il valore di scostamento di val
  func = &cl::double_val;  // calcola il valore di scostamento
      di double_val()

  cout << "Ecco i valori: ";
  cout << ob1.*data << " " << ob2.*data << "\n";

  cout << "Ecco i valori raddoppiati: ";
  cout << (ob1.*func)() << " ";
  cout << (ob2.*func)() << "\n";

  return 0;
}
```

All'interno di main(), questo programma crea due puntatori a membri: data e func. Si osservi attentamente la sintassi delle due dichiarazioni. Quando si dichiara un puntatore a un membro, si deve specificare la classe e utilizzare l'operatore di risoluzione del campo d'azione. Il programma crea inoltre i due oggetti ob1 e ob2 di tipo cl. Come si può vedere, i puntatori a membri possono puntare a funzioni o dati. Inoltre, il programma ricava gli indirizzi di val e double_val(). Come si è detto precedentemente, questi "indirizzi" non sono altro che valori di scostamento all'interno di un oggetto di tipo cl in cui è possibile trovare val e double_val(). Quindi, per visualizzare i valori val degli oggetti viene eseguito un accesso tramite data. Infine, il programma utilizza func per richiamare la funzione double_val(). Le parentesi aggiuntive sono necessarie per associare correttamente l'operatore .*.

Quando si accede a un membro di un oggetto utilizzando un oggetto o un indirizzo (discusso successivamente in questo capitolo), si deve utilizzare l'operatore .*. Quando invece si utilizza un puntatore all'oggetto, si deve utilizzare l'operatore ->*, come illustrato dalla seguente versione del programma.

```
#include <iostream>
using namespace std;
```

```
class cl {
public:
  cl(int i) {val=i;}
  int val;
  int double_val() {return val+val;}
};

int main()
{
  int cl::*data; // puntatore a membro (dati)
  int (cl::*func)(); // puntatore a membro (funzione)
  cl ob1(1), ob2(2); // crea gli oggetti
  cl *p1, *p2;

  p1 = &ob1;   // Accesso agli oggetti tramite un puntatore
  p2 = &ob2;

  data = &cl::val; // calcola il valore di scostamento di val
  func = &cl::double_val;  // calcola il valore di scostamento di
                              double_val()

  cout << "Ecco i valori: ";
  cout << p1->*data << " " << p2->*data << "\n";

  cout << "Ecco i valori raddoppiati: ";
  cout << (p1->*func)() << " ";
  cout << (p2->*func)() << "\n";

  return 0;
}
```

In questa versione, p1 e p2 sono puntatori a oggetti di tipo cl. Pertanto, per accedere a val e a double_val() viene utilizzato l'operatore ->*.

Si ricordi che i puntatori a membri sono diversi rispetto ai puntatori a specifiche istanze degli elementi di un oggetto. Ad esempio, si consideri il seguente frammento di codice (immaginando che cl sia dichiarata come nei programmi precedenti).

```
int cl::*d;
int *p;
cl o;

p = &o.val // questo è l'indirizzo di uno specifico val

d = &cl::val //questo è lo scostamento di val generico
```



Qui, p è un puntatore a un intero all'interno di un oggetto ben determinato. Al contrario, d è semplicemente un valore di scostamento che indica l'indirizzo in cui è possibile trovare val all'interno di ogni oggetto di tipo cl.

In generale, gli operatori dei puntatori a membri sono applicati in alcuni casi particolari e non sono molto utilizzati nella comune programmazione.

## 13.7 Gli indirizzi

Il C++ contiene una funzionalità in stretta relazione con i puntatori: l'*indirizzo*. Un indirizzo è essenzialmente un puntatore implicito. Un indirizzo può essere utilizzato in tre modi: come parametro di una funzione, come valore restituito da una funzione e come indirizzo a sé stante.

### Gli indirizzi come parametri

Probabilmente l'uso più importante degli indirizzi è quello di consentire di creare funzioni che utilizzano automaticamente il passaggio di parametri per indirizzo.

Come si è detto nel Capitolo 6, gli argomenti possono essere passati alle funzioni in due diversi modi: per valore o per indirizzo. Quando si usa un passaggio per valore, alla funzione viene passata una copia dell'argomento. Con un passaggio per indirizzo si passa alla funzione l'indirizzo dell'argomento. Normalmente il linguaggio C++ utilizza la chiamata per valore ma fornisce due modi per ottenere il passaggio dei parametri per indirizzo. Innanzitutto è possibile passare esplicitamente un puntatore all'argomento. In alternativa si può utilizzare un parametro indirizzo. In molti casi quest'ultima rappresenta la soluzione migliore.

Per comprendere cos'è un parametro indirizzo e la sua importanza, si parlerà del modo in cui una chiamata per indirizzo può essere generata impiegando un puntatore. Il seguente programma crea manualmente un parametro puntatore per la funzione neg() la quale inverte il segno della variabile intera puntata dal suo argomento.

```
// Crea manualmente una chiamata per indirizzo con un puntatore

#include <iostream>
using namespace std;

void neg(int *i);

int main()
{
  int x;
```

```
  x = 10;
  cout << x << " al negativo è uguale a ";

  neg(&x);
  cout << x << "\n";

  return 0;
}

void neg(int *i)
{
  *i = -*i;
}
```

In questo programma, neg() prende come parametro un puntatore all'intero di cui si deve invertire il segno. Pertanto, neg() deve essere richiamata esplicitamente con l'indirizzo di i. Inoltre, all'interno di neg(), per accedere alla variabile puntata da i si deve utilizzare l'operatore *. In questo modo si genera una chiamata per indirizzo "manuale" in C++ ed è anche l'unico modo per ottenere una chiamata di questo tipo in C. Fortunatamente in C++ è possibile rendere automatica questa funzionalità utilizzando un parametro indirizzo.

Per creare un parametro indirizzo si deve far precedere al nome del parametro il carattere &. Ecco come è possibile dichiarare neg() utilizzando un indirizzo:

```
void neg(int &i);
```

Questa forma chiede al compilatore di rendere i un parametro indirizzo. Fatto ciò, i diviene a tutti gli effetti un altro nome per qualsiasi argomento utilizzato per richiamare neg(). Ogni operazione eseguita su i influenzerà l'argomento chiamante. In termini tecnici, i è un puntatore implicito che fa automaticamente riferimento all'argomento utilizzato per richiamare neg(). Dopo che i è stato tramutato in un puntatore indirizzo, non sarà più necessario (né consentito) applicare l'operatore *. Al contrario, ogni volta che si utilizzerà i, si intenderà implicitamente l'indirizzo dell'argomento e le modifiche apportate a i modificheranno in realtà l'argomento. Inoltre, quando si richiamerà neg() non sarà più necessario (né consentito) far precedere al nome dell'argomento l'operatore &. Tutta l'operazione verrà automaticamente eseguita dal compilatore. Ecco quindi una nuova versione del programma precedente che impiega un parametro indirizzo:

```
// Uso di un parametro indirizzo

#include <iostream>
using namespace std;
```



```
void neg(int &i);  // ora i è un indirizzo

int main()
{
  int x;

  x = 10;
  cout << x << " al negativo è uguale a ";

  neg(x);  // non è più necessario l'operatore &
  cout << x << "\n";
  return 0;
}

void neg(int &i)
{
  i = -i;  // ora i è un indirizzo e non è più necessario usare *
}
```

Per ricapitolare: quando si crea un parametro indirizzo, tale parametro fa automaticamente riferimento (o punta implicitamente) all'argomento utilizzato per richiamare la funzione. Pertanto, nel programma precedente, l'istruzione:

```
i = -i ;
```

opera direttamente su x e non su una sua copia. Non vi sarà più alcuna necessità di applicare l'operatore & in un argomento. Inoltre, all'interno della funzione, il parametro indirizzo viene utilizzato direttamente senza necessità di applicare l'operatore *.

In generale, quando si assegna un valore a un indirizzo, il valore viene assegnato alla variabile cui punta l'indirizzo. Nel caso dei parametri di funzione, si tratterà della variabile utilizzata per richiamare la funzione.

All'interno della funzione non è possibile cambiare ciò a cui punta il parametro indirizzo. Quindi, un'istruzione come:

```
i++;
```

all'interno di neg() incrementa il valore della variabile utilizzata nella chiamata e non fa in modo che i punti a un nuovo indirizzo.

Ecco un altro esempio. Questo programma utilizza parametri indirizzo per scambiare il valore delle variabili in cui la funzione viene richiamata (la funzione swap() è il classico esempio di passaggio di parametri per indirizzo):

```
#include <iostream>
using namespace std;

void swap(int &i, int &j);

int main()
{
  int a, b, c, d;

1;
  b = 2;
  c = 3;
  d = 4;

  cout << "a e b: " << a << " " << b << "\n";
  swap(a, b);  // non è necessario l'operatore &
  cout << "a e b: " << a << " " << b << "\n";

  cout << "c e d: " << c << " " << d << "\n";
  swap(c, d);
  cout << "c e d: " << c << " " << d << "\n";

  return 0;
}

void swap(int &i, int &j)
{
  int t;

  t = i;  // non è necessario l'operatore *
  i = j;
  j = t;
}
```

Questo programma produce il seguente output:

```
a e b: 1 2
a e b: 2 1
c e d: 3 4
c e d: 4 3
```



## Passaggio di indirizzi a oggetti

Nel Capitolo 12 si è detto che quando un oggetto viene passato come argomento a una funzione, viene in effetti eseguita una copia di tale oggetto. Al termine della funzione viene chiamata la funzione distruttore per eliminare la copia. Se, per qualche motivo, non si desidera che venga richiamata la funzione distruttore, basterà passare l'oggetto per indirizzo (più avanti in questa guida si vedranno alcuni esempi di questo tipo di chiamata). Con il passaggio per indirizzo, non viene eseguita alcuna copia dell'oggetto. Questo significa che nel momento in cui la funzione termina non verrà distrutto alcun oggetto utilizzato come parametro, ovvero non verrà chiamata la funzione distruttore sul parametro. Ad esempio, si provi il seguente programma:

```
#include <iostream>
using namespace std;

class cl {
  int id;
public:
  int i;
  cl(int i);
  ~cl();
  void neg(cl &o) {o.i = -o.i;} // non viene creato un oggetto temporaneo
};

cl::cl(int num)
{
  cout << "Costruzione di " << num << "\n";
  id = num;
}

cl::~cl()
{
  cout << "Distruzione di " << id << "\n";
}

int main()
{
  cl o(1);

  o.i = 10;
  o.neg(o);

  cout << o.i << "\n";
```

```
  return 0;
}
```

Questo è l'output del programma:

```
Costruzione di 1
-10
Distruzione di 1
```

Come si può vedere, viene eseguita una sola chiamata alla funzione distruttore di cl. Se o fosse stata passata per valore, all'interno di neg() sarebbe stato creato un secondo oggetto e sarebbe stata richiamata una seconda volta la funzione distruttore per distruggere l'oggetto all'uscita da neg().

Come si può capire dal codice di neg(), quando si accede a un membro di una classe tramite il suo indirizzo, si usa l'operatore punto. L'operatore freccia è utilizzato solo per i puntatori.

Quando si esegue il passaggio di parametri per indirizzo, si deve ricordare che le modifiche agli oggetti che si trovano all'interno della funzione alterano l'oggetto utilizzato per la chiamata.

Infine si deve ricordare che il passaggio per indirizzo di un oggetto di dimensioni non banali è molto veloce. Gli argomenti vengono normalmente passati sullo stack, pertanto il passaggio per valore di grandi oggetti richiede grandi quantità di cicli di CPU per le operazioni di push e pop dell'oggetto sullo stack.

## Restituzione di indirizzi

Una funzione può restituire un indirizzo. Questo significa che una funzione può essere utilizzata anche sul lato sinistro di un'istruzione di assegnamento! Ad esempio, si consideri questo semplice programma:

```
#include <iostream>
using namespace std;

char &replace(int i);  // restituisce un indirizzo

char s[80] = "Salve a tutti";

int main()
{

  replace(5) = 'X'; // assegna X allo spazio dopo Salve

  cout << s;

  return 0;
}

char &replace(int i)
{
  return s[i];
}
```



Questo programma sostituisce lo spazio fra "Salve" e "a tutti" con una "X". In pratica, il programma visualizza la stringa "SalveXa tutti". Ma come si ottiene questo risultato?

Innanzitutto, replace() restituisce l'indirizzo di un array di caratteri. Così come è realizzata, replace() restituisce l'indirizzo dell'elemento di s specificato dal suo argomento i. L'indirizzo restituito da replace() viene utilizzato in main() per assegnare a tale elemento il carattere X.

Una cosa cui fare attenzione quando si restituisce l'indirizzo è il fatto che l'oggetto cui si fa riferimento non esca dal campo di visibilità al termine della funzione.

## Indirizzi indipendenti

Gli utilizzi di gran lunga più comuni degli indirizzi sono il passaggio di un argomento tramite chiamate per indirizzo e l'impiego come valore restituito da una funzione. Ma è anche possibile dichiarare un indirizzo che sia semplicemente una variabile. Questo tipo di indirizzo è chiamato *indirizzo indipendente*.

Quando si crea un indirizzo indipendente, non si fa altro che creare un altro nome per una variabile. Tutte le variabili indirizzo indipendenti devono essere inizializzate al momento della creazione. Il motivo è ovvio: tranne che nell'inizializzazione, non è possibile modificare l'oggetto a cui punta la variabile indirizzo. Pertanto tale variabile deve essere inizializzata al momento della dichiarazione (in C++, l'inizializzazione è un'operazione completamente distinta dall'assegnamento).

Il seguente programma illustra l'uso degli indirizzi indipendenti.

```
#include <iostream>
using namespace std;

int main()
{
  int a;
```

```
  int &ref = a; // indirizzo indipendente

  a = 10;
  cout << a << " " << ref << "\n";

  ref = 100;
  cout << a << " " << ref << "\n";

  int b = 19;
  ref = b; // inserisce in a il valore di b
  cout << a << " " << ref << "\n";

  ref--;  // decrementa a
          // non modifica ciò a cui punta ref

  cout << a << " " << ref << "\n";

  return 0;
}
```

Il programma visualizza questo output:

```
10 10
100 100
19 19
18 18
```

In realtà, gli indirizzi indipendenti sono molto poco utilizzati in quanto si tratta semplicemente di nomi diversi per una determinata variabile. L'utilizzo di due nomi per identificare la stessa variabile complica inutilmente il programma.

### L'indirizzo di un tipo derivato

Come si è detto per i puntatori, l'indirizzo di una classe base può essere utilizzato anche per far riferimento a un oggetto appartenente a una classe derivata. Un'applicazione di ciò è nei parametri delle funzioni. Un parametro corrispondente a un indirizzo della classe base può ricevere oggetti della classe base o anche oggetti appartenenti a una classe da essa derivata.

### Restrizioni relative agli indirizzi

Gli indirizzi sono soggetti a un gran numero di restrizioni. Non è possibile conoscere l'indirizzo di un indirizzo. Non è possibile creare array di indirizzi. Non è possibile creare un puntatore a un indirizzo. Non è possibile conoscere l'indirizzo di un campo bit.



Una variabile indirizzo deve essere inizializzata al momento della dichiarazione a meno che non sia un membro di una classe, il parametro di una funzione o il valore restituito da una funzione. È proibito l'uso di indirizzi nulli.

## 13.8 Questione di stile

Quando si dichiarano variabili puntatore e indirizzi, alcuni programmatori C++ utilizzano un particolare stile di programmazione che associa agli operatori * e & il nome del tipo e non quello della variabile. Ad esempio queste due dichiarazioni sono equivalenti:

```
int& p; // & associato al tipo
int &p; // & associato alla variabile
```

L'associazione degli operatori * o & al nome del tipo riflette il desiderio di alcuni programmatori di utilizzare in C++ un tipo puntatore distinto. In questo senso, il problema che sorge associando tali operatori al nome del tipo piuttosto che al nome della variabile consiste nel fatto che secondo la sintassi formale del C++, né & né * sono distributivi in un elenco di variabili. Pertanto, questo potrebbe portare alla creazione di dichiarazioni fuorvianti. Ad esempio, la dichiarazione seguente crea *uno e non due* puntatori a interi.

```
int* a, b;
```

Qui, b viene dichiarato come intero (e non come puntatore a intero) poiché, in base alla sintassi del C++, quando l'operatore * (ma anche &) viene utilizzato in una dichiarazione, fa riferimento al nome della variabile seguente e non al nome del tipo precedente.

Il problema con questo tipo di dichiarazioni è che il messaggio visivo suggerisce che sia a che b siano puntatori mentre in effetti solo a è un puntatore. Questa confusione visiva non trae in inganno solo i programmatori alle prime armi ma anche i professionisti più esperti.

È importante comprendere che, per quanto riguarda il compilatore C++, non importa che si scriva int *p o int* p. Pertanto, si è liberi di specificare l'associazione alla variabile o al tipo. In ogni caso, per evitare confusioni, questa guida adotta l'associazione degli operatori * e & al nome delle variabili su cui operano piuttosto che al tipo.

## 13.9 Gli operatori di allocazione dinamica del C++

Il linguaggio C++ fornisce un sistema di allocazione dinamica che si basa sui due operatori new e delete. Come si vedrà, vi sono sostanziali vantaggi nell'approccio del C++ all'allocazione dinamica della memoria.

Gli operatori new e delete sono utilizzati per allocare e liberare la memoria run-time. L'allocazione dinamica della memoria è una parte importante di quasi ogni programma. Come si è detto nella Parte prima, il linguaggio C++ supporta anche le funzioni di allocazione dinamica della memoria malloc() e free() che sono state incluse per compatibilità con il linguaggio C. Tuttavia quando si lavora in C++, è opportuno utilizzare gli operatori new e delete che offrono numerosi vantaggi. L'operatore new alloca un'area di memoria e restituisce un puntatore all'inizio di tale area. L'operatore delete libera la memoria precedentemente allocata con new. Di seguito vengono presentate le forme generali di new e delete:

*var_p* = new *tipo*;

delete *var_p*;

Qui, *var_p* è una variabile puntatore che riceve un puntatore a un'area di memoria sufficientemente estesa da contenere un oggetto di tipo *tipo*.

Dato che l'heap ha dimensioni finite, può giungere ad esaurimento. Se la memoria disponibile è insufficiente per esaudire la richiesta di allocazione, allora la richiesta new non verrà esaudita e verrà generata l'eccezione bad_alloc. Questa eccezione è definita nell'header <new>. Il programma dovrebbe gestire questa eccezione e prendere le misure appropriate (la gestione delle eccezioni è descritta nel Capitolo 19). Se il programma non gestisce l'eccezione, verrà automaticamente chiuso.

Le azioni eseguite da new, così come sono state descritte, sono specificate dallo standard del linguaggio C++. Il problema è che non tutti i compilatori, specialmente quelli meno recenti, implementano new secondo lo standard. Quando venne inventato il C++, in caso di fallimento new restituiva il valore nullo. Successivamente venne deciso che in caso di fallimento new dovesse lanciare un'eccezione. Infine venne deciso che un fallimento di new generasse un'eccezione e che, opzionalmente, venisse restituito un puntatore nullo. Pertanto new è stato implementato in modo differente a seconda dei momenti e del produttore del compilatore. Anche se alla fine tutti i compilatori implementeranno new secondo lo standard, attualmente l'unico modo per sapere il modo in cui viene gestito il fallimento di new consiste nel consultare la documentazione del compilatore.

Dato che lo standard del C++ specifica che new generi un'eccezione in caso di fallimento, questo è il modo in cui verrà scritto il codice in questo volume. Se il proprio compilatore dovesse gestire un problema di allocazione in modo differente, sarà ovviamente necessario apportare al programma le modifiche appropriate.



Ad esempio, il seguente programma alloca la memoria necessaria per contenere un intero:

```
#include <iostream>
#include <new>
using namespace std;


int main()
{
  int *p;

  try {
    p = new int;  // alloca spazio per un int
  } catch (bad_alloc xa) {
    cout << "Errore di allocazione\n";
    return 1;
  }

  *p = 100;

  cout << "In " << p << " ";
  cout << "si trova il valore " << *p << "\n";

  delete p;

  return 0;
}
```

Questo programma assegna a p un indirizzo dello heap le cui dimensioni sono sufficienti per contenere un intero. Poi assegna a tale area di memoria il valore 100 e visualizza il contenuto della memoria sullo schermo. Infine libera la memoria allocata dinamicamente. Si ricordi che se il compilatore implementa new in modo da fargli restituire un valore nullo, sarà necessario adattare il programma precedente.

L'operatore delete deve essere utilizzato solo con un puntatore valido allocato precedentemente tramite new. Se si utilizza delete con un altro tipo di puntatore, il risultato sarà indefinito e provocherà quasi certamente il blocco del sistema.

Anche se new e delete eseguono funzioni simili a malloc() e free(), questi due operatori presentano numerosi vantaggi. Innanzi tutto, new alloca automaticamente la memoria necessaria per contenere un oggetto del tipo specificato. Quindi non sarà più necessario utilizzare l'operatore sizeof. Poiché le dimensioni del-

l'area allocata vengono calcolate automaticamente, si elimina ogni possibilità di errore. In secondo luogo, new restituisce automaticamente un puntatore del tipo specificato. Non è necessario utilizzare una conversione di tipo esplicita così come si fa quando si alloca la memoria utilizzando malloc(). Infine, sia new che delete possono essere modificati tramite overloading consentendo perciò di creare sistemi di allocazione personalizzati.

Anche se non vi è alcuna regola formale che stabilisce ciò, è meglio non utilizzare insieme new e delete con malloc() e free() nello stesso programma in quanto non vi è alcuna garanzia che essi siano compatibili.

## Inizializzazione della memoria allocata

È possibile inizializzare la memoria allocata con un determinato valore inserendo un inizializzatore dopo il nome del tipo nell'istruzione new. Ecco la forma generale di new quando viene inclusa anche l'inizializzazione:

*var_p* = new *tipo* (*inizializzatore*);

naturalmente il tipo dell'inizializzatore deve essere compatibile con il tipo dei dati per i quali è stata allocata la memoria. Ad esempio, il seguente programma dà all'intero allocato il valore iniziale 87.

```
#include <iostream>
#include <new>
using namespace std;

int main()
{
  int *p;

  try {
    p = new int (87);  // inizializzato a 87
  } catch(bad_alloc xa) {
    cout << "Errore di allocazione\n";
    return 1;
  }

  cout << "In " << p << " ";
  cout << "si trova il valore " << *p << "\n";
  delete p;

  return 0;
}
```



## Allocazione degli array

L'operatore new consente anche di allocare array utilizzando la forma generale:

*var_p* = new *tipo_array* [*dim*];

Dove *dim* specifica il numero di elementi dell'array. Per liberare la memoria occupata dall'array si deve utilizzare questa forma di delete:

delete [ ] *var_p*;

Qui, la coppia di parentesi quadre ([ ]) informa delete che si deve rilasciare la memoria occupata da un array.

Ad esempio, il programma seguente alloca un array formato da dieci elementi interi.

```
#include <iostream>
#include <new>
using namespace std;

int main()
{
  int *p, i;

  try {
    p = new int [10];  // alloca un array di 10 interi
  } catch(bad_alloc xa) {
    cout << "Errore di allocazione\n";
    return 1;
  }

  for(i=0; i<10; i++ )
    p[i] = i;

  for(i=0; i<10; i++)
    cout << p[i] << " ";

  delete [] p; // libera la memoria

  return 0;
}
```

Si noti l'istruzione delete. Come si è appena detto, quando si libera la memoria occupata da un array allocato da new, è necessario specificare che si sta libe-

rando la memoria di un array utilizzando delete insieme a [ ] (come si vedrà nella prossima sezione, questo accorgimento è particolarmente importante quando si devono allocare array di oggetti).

Vi è una restrizione all'allocazione di array: non è possibile assegnare valori iniziali ad array dinamici. Quindi, quando si alloca un array non è possibile specificare un inizializzatore.

## Allocazione di oggetti

L'operatore new consente anche di allocare dinamicamente oggetti. L'operazione crea un oggetto e restituisce un puntatore a tale oggetto. L'oggetto creato dinamicamente si comporta come qualsiasi altro oggetto e nel momento in cui viene creato, viene richiamata (se esiste) la sua funzione costruttore. Nel momento in cui l'oggetto viene eliminato con delete viene richiamata la sua funzione distruttore.

Ecco un breve programma che crea una classe chiamata balance che collega al nome di una persona il suo saldo di conto corrente. All'interno di main() viene creato dinamicamente un oggetto di tipo balance.

```
#include <iostream>
#include <new>
#include <cstring>
using namespace std;

class balance {
  double cur_bal;
  char name[80];
public:
  void set(double n, char *s) {
    cur_bal = n;
    strcpy(name, s);
  }

  void get_bal(double &n, char *s) {
    n = cur_bal;
    strcpy(s, name);
  }
};

int main()
{
  balance *p;
  char s[80];
  double n;
```



```
  try {
    p = new balance;
  } catch(bad_alloc xa) {
    cout << "Errore di allocazione\n";
    return 1;
  }

  p->set(12387.87, "Mario Rossi");

  p->get_bal(n, s);

  cout << s << " saldo: " << n;
  cout << "\n";

  delete p;

  return 0;
}
```

Poiché p contiene un puntatore a un oggetto, per accedere ai membri dell'oggetto si utilizza l'operatore freccia.

Come si è detto, gli oggetti allocati dinamicamente possono essere dotati di costruttori e distruttori. Inoltre, le funzioni costruttore possono essere parametrizzate. Si esamini la seguente versione del programma precedente.

```
#include <iostream>
#include <new>
#include <cstring>
using namespace std;

class balance {
  double cur_bal;
  char name[80];
public:
  balance(double n, char *s) {
    cur_bal = n;
    strcpy(name, s);
  }
  ~balance() {
    cout << "Distruzione di ";
    cout << name << "\n";
  }
  void get_bal(double &n, char *s) {
    n = cur_bal;
    strcpy(s, name);
```

```
  }
};

int main()
{
  balance *p;
  char s[80];
  double n;

  // questa versione usa un inizializzatore
  try {
    p = new balance (12387.87, "Mario Rossi");
  } catch(bad_alloc xa) {
    cout << "Errore di allocazione\n";
    return 1;
  }

  p->get_bal(n, s);

  cout << s << " saldo: " << n;
  cout << "\n";

  delete p;

  return 0;
}
```

I parametri della funzione costruttore dell'oggetto sono specificati dopo il nome del tipo come avviene in qualsiasi altra inizializzazione.

È possibile allocare anche array di oggetti ma vi è una limitazione. Poiché nessun array allocato da new può essere inizializzato, è necessario assicurarsi che se la classe contiene più funzioni costruttore, una non preveda parametri. In caso contrario, quando si cercherà di allocare l'array il compilatore C++ non troverà un costruttore adatto e non consentirà la compilazione del programma.

In questa versione del programma precedente viene allocato un array di oggetti balance e viene richiamato il costruttore senza parametri.

```
#include <iostream>
#include <new>
#include <cstring>
using namespace std;

class balance {
  double cur_bal;
```



```
  char name[80];
public:
  balance(double n, char *s) {
    cur_bal = n;
    strcpy(name, s);
  }
  balance() {} // costruttore senza parametri
  ~balance() {
    cout << "Distruzione di ";
    cout << name << "\n";
  }
  void set(double n, char *s) {
    cur_bal = n;
    strcpy(name, s);
  }
  void get_bal(double &n, char *s) {
    n = cur_bal;
    strcpy(s, name);
  }
};

int main()
{
  balance *p;
  char s[80];
  double n;
  int i;

  try {
    p = new balance [3]; // alloca l'intero array
  } catch(bad_alloc xa) {
    cout << "Errore di allocazione\n";
    return 1;
  }

  // si noti l'uso del punto, non della freccia
  p[0].set(12387.87, "Mario Rossi");
  p[1].set(144.00, "Bruno Bianchi");
  p[2].set(-11.23, "Paolo Verdi");

  for(i=0; i<3; i++) {
    p[i].get_bal(n, s);

    cout << s << " saldo: " << n;
    cout << "\n";
  }
```

```
  delete [] p;
  return 0;
}
```

Ecco l'output prodotto dal programma:

```
Mario Rossi saldo: 12387.9
Bruno Bianchi saldo: 144
Paolo Verdi saldo: -11.23
Distruzione di Paolo Verdi
Distruzione di Bruno Bianchi
Distruzione di Mario Rossi
```

Un motivo per cui si deve utilizzare la forma delete[] quando si deve cancellare un array di oggetti allocati dinamicamente è il fatto che la funzione distruttore deve essere richiamata per ogni oggetto dell'array.

## L'alternativa di new e delete

Lo standard del linguaggio C++ consente di fare in modo che, in caso di fallimento nell'allocazione della memoria, new restituisca null invece di lanciare un'eccezione. Questa forma di new è utile soprattutto quando si deve compilare codice non recente con un compilatore C++ standard. Inoltre è utile quando si devono sostituire con new le vecchie chiamate a malloc() (avviene quando si deve aggiornare al C++ del codice C).

Ecco l'utilizzo di questa forma di new:

*p_var* = new(nothrow) *tipo*;

Qui, *p_var* è una variabile puntatore di tipo *tipo*. La forma nothrow di new funziona esattamente come la forma originale di new, nata qualche tempo fa. Poiché in caso di fallimento restituisce null, può essere inserita nel vecchio codice evitando quindi di dover aggiungere la gestione delle eccezioni. Tuttavia, quando si deve realizzare nuovo codice, è opportuno impiegare le eccezioni. Per utilizzare l'opzione nothrow, si deve includere l'header new.

Il seguente programma mostra l'uso di nothrow.

```
// Dimostrazione della forma nothrow di new.
#include <iostream>
#include <new>
using namespace std;
```



```
int main()
{
  int *p, i;

  p = new(nothrow) int[32]; // uso dell'opzione nothrow
  if(!p) {
    cout << "Errore di allocazione.";
    return 1;
  }

  for(i=0; i<32; i++) p[i] = i;

  for(i=0; i<32; i++) cout << p[i] << " ";

  delete [] p; // memoria libera

  return 0;
}
```

Come si può vedere in questo programma, quando si impiega l'approccio nothrow occorre controllare il puntatore restituito da new dopo ogni richiesta di allocazione.

## Altre forme di new e delete

Vi è anche un'altra forma speciale di new che può essere utilizzata per specificare un metodo alternativo di allocazione della memoria. Tale forma è utile soprattutto quando si esegue l'overloading di new per casi particolari. L'implementazione standard di questa forma dell'operatore new ha il seguente aspetto:

*p_var* = new (*location*) *type*;

Qui, posizione specifica l'indirizzo restituito da new.

Per liberare la memoria allocata con questa forma di new occorre impiegare la corrispondente forma di delete.

Capitolo 14

# Overloading di funzioni, costruttori di copie e argomenti standard


14.1 **Overloading delle funzioni**

14.2 **Overloading delle funzioni costruttore**

14.3 **I costruttori di copie**

14.4 **Ricerca dell'indirizzo di una funzione modificata tramite overloading**

14.5 **L'anacronismo della parola riservata overload**

14.6 **Gli argomenti standard delle funzioni**

14.7 **Overloading di funzioni e ambiguità**


Questo capitolo esamina gli argomenti dell'overloading delle funzioni, dei costruttori di copie e degli argomenti standard. L'overloading delle funzioni è uno degli aspetti fondamentali del linguaggio di programmazione C++. Infatti l'overloading delle funzioni non solo fornisce il supporto per il polimorfismo in fase di compilazione ma aggiunge al linguaggio flessibilità e comodità. Tra le funzioni modificate tramite overloading, quelle più importanti sono i costruttori. La forma più importante è costituita dal costruttore di copie. Gli argomenti standard sono strettamente correlati al concetto di overloading delle funzioni. Gli argomenti standard possono talvolta costituire un'alternativa all'overloading delle funzioni.

## 14.1 Overloading delle funzioni

L'overloading consiste nell'impiegare lo stesso nome per due o più funzioni. Il segreto dell'overloading è il fatto che ogni ridefinizione della funzione deve utilizzare parametri di tipo differente oppure in numero differente. Grazie a queste differenze, il compilatore sa quale funzione richiamare in una determinata situazione. Ad esempio, questo programma esegue l'overloading della funzione myfunc() utilizzando parametri di tipo differente.

```
#include <iostream>
using namespace std;

int myfunc(int i); // differenze nel tipo dei parametri
double myfunc(double i);

int main()
{

  cout << myfunc(10) << " "; // richiama myfunc(int i)
  cout << myfunc(5.4); // richiama myfunc(double i)
  return 0;

}
double myfunc(double i)
{
  return i;
}

int myfunc(int i)
{
  return i;
}
```

Questo programma invece esegue l'overloading di myfunc() impiegando un numero differente di parametri:

```
#include <iostream>
using namespace std;

int myfunc(int i); // differiscono per il numero dei parametri
int myfunc(int i, int j);

int main()
{
  cout << myfunc(10) << " "; // richiama myfunc(int i)
  cout << myfunc(4, 5); // richiama myfunc(int i, int j)
  return 0;
}

int myfunc(int i)
{
  return i;
}
```



```
int myfunc(int i, int j)
{
  return i*j;
}
```

Come si è detto, la caratteristica principale dell'overloading delle funzioni è il fatto che queste devono differire per quanto riguarda il tipo e/o il numero dei parametri. Dunque due funzioni non possono differire solo per il tipo di dati restituito. Ad esempio, ecco un modo errato per eseguire l'overloading di myfunc():

```
int myfunc(int i);   // Errore: non è sufficiente la differenza
float myfunc(int i); // del solo valore restituito.
```

Talvolta due dichiarazioni di funzioni sembrano differenti mentre in realtà non è così. Ad esempio, si considerino le due dichiarazioni seguenti.

```
void f(int *p);
void f(int p[]); // errore, *p è la stessa cosa di p[]
```

Si deve sempre ricordare che per il compilatore * p è uguale a p [ ]. Pertanto anche se i due prototipi sembrano diversi quanto al tipo dei loro parametri, in realtà non lo sono.

## 14.2 Overloading delle funzioni costruttore

Le funzioni costruttore possono essere modificate tramite overloading; nella pratica comune, i costruttori modificati tramite overloading sono molto impiegati. Vi sono principalmente tre motivi che spingono a eseguire l'overloading di una funzione costruttore: la maggiore flessibilità, la possibilità di creare oggetti inizializzati e non inizializzati e la possibilità di definire costruttori di copie. In questa sezione verranno esaminati i primi due aspetti. L'argomento dei costruttori di copie verrà descritto nella prossima sezione.

### Overloading di un costruttore per acquisire maggiore flessibilità

Spesso si crea una classe per la quale esistono due o più modi per costruire un oggetto. In questi casi è opportuno fornire una funzione costruttore modificata tramite overloading per entrambi questi metodi. Questa è una regola fondamentale in quanto se si cerca di creare un oggetto per il quale non esiste un costruttore, viene prodotto un errore in fase di compilazione.

Offrendo un costruttore per ognuna delle modalità in cui un utilizzatore della classe può voler costruire un oggetto, si aumenta la flessibilità della classe. L'utente è libero di scegliere il modo migliore per costruire un oggetto in una determinata circostanza. Si consideri il seguente programma che crea una classe chiamata date che contiene una data. Si noti che esistono due versioni del costruttore:

```
#include <iostream>
#include <cstdio>
using namespace std;

class date {
  int day, month, year;
public:
  date(char *d);
  date(int m, int d, int y);
  void show_date();
};

// Inizializzazione con una stringa.
date::date(char *d)
{
  sscanf(d, "%d%*c%d%*c%d", &month, &day, &year);
}

// Initializzazione con interi.
date::date(int m, int d, int y)
{
day = d;
  month = m;
  year = y;
}

void date::show_date()
{
  cout << month << "/" << day;
  cout << "/" << year << "";
}
int main()
{
  date ob1(12, 4, 2001), ob2("10/22/2001");

  ob1.show_date();
  ob2.show_date();

  return 0;
}
```



In questo programma si inizializza un oggetto di tipo date; l'operazione può essere eseguita specificando la data tramite tre interi oppure utilizzando una stringa che contiene la data specificata in una forma generale:

*mm/dd/yyyy*

Poiché si tratta in entrambi casi di metodi comuni di rappresentazione di una data, è opportuno consentire all'utente di costruire l'oggetto in entrambi i modi.

Come illustrato dalla classe date, forse il motivo più comune che spinge a eseguire l'overloading di un costruttore è quello di consentire di creare un oggetto utilizzando il metodo più appropriato e naturale in una determinata circostanza. Ad esempio, nel seguente main(), all'utente viene richiesta la data che viene introdotta nell'array s. Questa stringa può anche essere utilizzata direttamente per creare d. Non vi è alcuna necessità di convertirla in un'altra forma. Se date() non fosse stata modificata tramite overloading per accertare la forma di stringa, sarebbe stato necessario convertire manualmente la data in tre interi.

```
int main()
{
  char s[80];
  cout << "Immettere la nuova data: ";
  cin >> s;

  date d(s);
  d.show_date();

  return 0;
}
```

Vi potrebbe essere anche una situazione in cui può essere più comodo inizializzare un oggetto di tipo date utilizzando tre interi. Ad esempio, se la data viene generata da qualche metodo di calcolo, allora è più naturale e appropriato creare un oggetto date utilizzando la forma date(int, int, int). Qui lo scopo dell'overloading del costruttore di date è quello di rendere più flessibile e semplice il suo uso. Questa maggiore flessibilità e facilità d'uso è particolarmente importante quando si creano librerie di classi che verranno utilizzate da altri programmatori.

## Creazione di oggetti inizializzati e non inizializzati

Un altro motivo che spesso spinge a eseguire l'overloading delle funzioni costruttore consiste nel consentire la creazione di oggetti inizializzati e non

inizializzati (o, più precisamente, oggetti a inizializzazione standard). Questo è particolarmente importante se si vuole essere in grado di creare array dinamici di oggetti di una determinata classe poiché non è possibile inizializzare un array allocato dinamicamente. Per consentire l'uso degli array di oggetti inizializzati e non inizializzati, occorre includere un costruttore che supporti l'inizializzazione e uno che non la supporti.

Ad esempio, il seguente programma dichiara due array di tipo powers; uno viene inizializzato l'altro no. Quindi il programma alloca dinamicamente un array.

```
#include <iostream>
#include <new>
using namespace std;

class powers {
  int x;
public:
  // overload del costruttore in due modi
  powers() { x = 0; } // senza inizializzatore
  powers(int n) { x = n; } // con inizializzatore

  int getx() { return x; }
  void setx(int i) { x = i; }
};

int main()
{
  powers ofTwo[] = { 1, 2, 4, 8, 16}; // inizializzato
  powers ofThree[5]; // non inizializzato
  powers *p;
  int i;

  // Visualizza le potenze di 2
  cout << "Potenze di 2: ";
  for(i=0; i<5; i++)  {
   cout  <<  ofTwo[i].getx() <<  " ";
}
  cout << "\n\n";

  // imposta le potenze di 3
  ofThree[0].setx(1);
  ofThree[1].setx(3);
  ofThree[2].setx(9);
  ofThree[3].setx(27);
  ofThree[4].setx(81);
```



```
  // Visualizza le potenze di 3
  cout << "Potenze di 3: ";
  for(i=0; i<5; i++) {
   cout << ofThree[i].getx() << " ";
}

  cout <<"\n\n";

  // allocazione dinamica di un array
  try {
    p = new powers[5]; // nessuna inizializzazione
}  catch (bad_alloc xa) {
     cout << "Errore di allocazione\n";
     return 1;


  // Inizializza l'array dinamico con le potenze di 2
  for(i=0; i<5; i++) {
   p[i].setx(ofTwo[i].getx());
}

// Visualizza le potenze di 2
  cout << "Potenze di 2: ";
  for(i=0; i<5; i++) {
   cout << p[i].getx() << " ";
}
  cout << "\n\n";

  delete [] p;
  return 0;
}
```

In questo esempio, sono necessari entrambi i costruttori. Il costruttore standard viene impiegato per costruire l'array non inizializzato ofThree e l'array allocato dinamicamente. Il costruttore parametrizzato viene richiamato per creare gli oggetti dell'array ofTwo.

## 14.3 I costruttori di copie

Uno dei costruttori più importanti da modificare tramite overloading è il costruttore di copie. La creazione di un costruttore di copie può aiutare a evitare i problemi che sorgono quando si utilizza un oggetto per inizializzarne un altro.

Si può iniziare ricordando il problema che il costruttore di copie deve risolvere. Normalmente, quando si usa un oggetto per inizializzarne un altro, il linguaggio C++ prevede l'esecuzione di una copia bit a bit. Questo significa che l'oggetto di destinazione sarà una copia identica dell'oggetto utilizzato per l'inizializzazione. Anche se questo comportamento è appropriato per la maggior parte dei casi (e in genere è esattamente ciò che si vuole ottenere) in alcuni casi non si deve usare una copia a bit. Ad esempio uno dei casi più comuni si presenta quando la creazione di un oggetto richiede l'allocazione di un'area di memoria. Ad esempio, si supponga che la creazione di una classe chiamata MyClass allochi un'area di memoria per ciascun oggetto e poi si immagini un oggetto A di tale classe. Questo significa che A ha già allocato la propria memoria. Inoltre si supponga di utilizzare A per inizializzare B come nel seguente esempio:

```
MyClass B = A;
```

Se viene eseguita una copia bit a bit, allora B sarà una copia esatta di A. Questo significa che B utilizzerà la stessa area di memoria allocata per A, non una propria area distinta. Chiaramente questo non è il risultato desiderato. Ad esempio, se MyClass include un distruttore che libera la memoria, allora la distruzione di A e B provocherà la doppia cancellazione della stessa area di memoria!

Lo stesso tipo di problema può verificarsi in altri due casi: innanzitutto quando viene eseguita una copia di un oggetto nel momento in cui questo viene passato come argomento a una funzione; in secondo luogo quando viene creato un oggetto temporaneo restituito da una funzione. Si ricordi che gli oggetti temporanei vengono creati automaticamente per contenere il valore restituito da una funzione ma possono anche essere creati in altre situazioni.

Per risolvere il problema appena descritto, il linguaggio C++ consente di creare un costruttore di copie che il compilatore impiega quando si usa un oggetto per inizializzarne un altro. Quando esiste un costruttore di copie, viene impiegato al posto del costruttore bit a bit. La forma più generale di costruttore di copie è:

```
classname (const nome-classe &o) {
// corpo del costruttore
}
```

Qui o è un riferimento all'oggetto che si trova sul lato destro dell'inizializzazione. Un costruttore di copie può essere dotato di parametri aggiuntivi sempre che siano stati definiti degli argomenti standard. Comunque, in ogni caso, il primo parametro deve essere un riferimento all'oggetto che esegue l'inizializzazione.

È importante comprendere che il linguaggio C++ definisce due diversi tipi di situazioni in cui a un oggetto viene assegnato il valore di un altro oggetto. Il primo caso è l'assegnamento. Il secondo è l'inizializzazione che può verificarsi in tre modi:



- quando un oggetto inizializza esplicitamente un altro oggetto, come nelle dichiarazioni;
- quando viene eseguita una copia di un oggetto che deve essere passato a una funzione;
- quando viene creato un oggetto temporaneo (normalmente come valore restituito da una funzione).

Il costruttore di copie viene applicato solo nel caso delle inizializzazioni. Ad esempio, supponendo che esista una classe chiamata myclass e che y sia un oggetto di tipo myclass, l'inizializzazione viene impiegata da ciascuna delle seguenti istruzioni:

```
myclass x = y; // y inizializza esplicitamente x
func(y);       // y passata come parametro
y = func();    // y riceve da func() un oggetto temporaneo
```

Di seguito viene presentato un esempio dove è necessario impiegare un costruttore di copie esplicito. Questo programma crea un array di interi "sicuro" che impedisce il superamento dei limiti. Nel Capitolo 15 si trova un esempio migliore che crea un array sicuro tramite operatori modificati tramite overloading. La memoria per gli array viene allocata tramite new e all'interno di ciascun oggetto array viene gestito un puntatore alla relativa area memoria.

```
/* Questo programma crea una classe per array "sicuri".
   Poiché lo spazio per l'array viene allocato con new, viene fornito
   un costruttore di copie per allocare la memoria quando si utilizza
   un oggetto dell'array per inizializzarne un altro.
*/
#include <iostream>
#include <new>
#include <cstdlib>
using namespace std;

class array {
  int *p;
  int size;
public:
  array(int sz) {
    try {
      p = new int[sz];
    } catch (bad_alloc xa){
    cout << "Errore di allocazione\n";
```

```
      exit(EXIT_FAILURE);
    }
    size = sz;
  }
  ~array() { delete [] p; }

  // costruttore di copie
  array(const array &a);

  void put(int i, int j) {
    if(i>=0 && i<size) p[i]  j;
  }
  int get(int i) {
    return p[i];
  }
};

// Costruttore di copie
array::array(const array &a) {
  int i;

  try {
    p = new int[a.size];
  } catch (bad_alloc xa) {
    cout << "Errore di allocazione\n";
    exit(EXIT_FAILURE);
  }
  for(i=0; i<a.size; i++) p[i]  a.p[i];
}

int main()
{
  array num(10);
  int i;

  for(i=0; i<10; i++) num.put(i, i);
  for(i=9; i>=0; i--) cout << num.get(i);
  cout << "\n";

  // crea un altro array e lo inizializza con num
  array x(num); // richiama il costruttore di copie
  for(i=0; i<10; i++) cout << x.get(i);

  return 0;
}
```



Ecco cosa accade quando si usa num per inizializzare x nell'istruzione:

```
array x(num); // richiama il costruttore di copie
```

Viene richiamato il costruttore di copie, viene allocata l'area di memoria per il nuovo array contenuta in x.p e nell'array di x viene copiato il contenuto di num. In questo modo, gli array x e num conterranno gli stessi valori ma ciascun array si troverà in un'area di memoria distinta. Questo significa che num.p e x.p non puntano alla stessa area di memoria. Se non fosse stato creato il costruttore di copie, l'inizializzazione bit a bit avrebbe fatto in modo che x e num condividessero la stessa area di memoria (pertanto num.p e x.p avrebbero puntato alla stessa area di memoria).

Si ricordi che il costruttore di copie viene richiamato solo per le inizializzazioni. Ad esempio, questa sequenza non richiama il costruttore di copie definito nel programma precedente:

```
array a(10);
// ...
array b(10);

b = a; // non richiama il costruttore di copie
```

In questo caso, b = a esegue l'operazione di assegnamento. Se = non è modificato tramite overloading (come in questo caso), verrà eseguita una copia bit a bit. Pertanto, in alcuni casi, per evitare problemi si deve eseguire l'overloading dell'operatore = oltre a creare un costruttore di copie (vedere il Capitolo 15).

## 14.4 Ricerca dell'indirizzo di una funzione modificata tramite overloading

Come si è detto nel Capitolo 5, è possibile ottenere l'indirizzo di una funzione. Ad esempio l'indirizzo può essere assegnato a un puntatore per poter richiamare la funzione tramite tale puntatore. Se la funzione non ha subito overloading, questa operazione è immediata. Al contrario, per le funzioni modificate tramite overloading, l'operazione è leggermente più complessa. Per capire il motivo di questa complessità, si consideri innanzitutto la seguente istruzione che assegna a un puntatore chiamato p l'indirizzo di una funzione chiamata myfunc():

```
p = myfunc;
```

Se myfunc() non è modificata tramite overloading, allora esisterà una e una sola funzione chiamata myfunc() e il compilatore non avrà alcuna difficoltà ad assegnare a p il suo indirizzo. Se invece myfunc() è stata modificata tramite overloading, come può il compilatore sapere qual è la versione di myfunc() che si intende assegnare a p? Ecco la risposta: tutto dipende dal modo in cui è stato dichiarato p. Ad esempio, si consideri il seguente programma:

```
#include <iostream>
using namespace std;

int myfunc(int a);
int myfunc(int a, int b);

int main()
{
  int (*fp)(int a); // puntatore a int f(int)

  fp = myfunc; // punta a myfunc(int)

  cout << fp(5);

  return 0;
}

int myfunc(int a)
{
  return a;
}

int myfunc(int a, int b)
{
  return a*b;
}
```

Qui esistono due versioni di myfunc(). Entrambe restituiscono un valore int ma una accetta un unico argomento intero mentre l'altra richiede due argomenti interi. Nel programma, fp è dichiarato come un puntatore a una funzione che restituisce un intero e che prende un argomento intero. Quando a fp viene assegnato l'indirizzo di myfunc(), il linguaggio C++ utilizza questa informazione per selezionare la versione myfunc(int a). Se fp fosse stato dichiarato nel seguente modo:

```
int (*fp)(int a, int b);
```

Allora a fp sarebbe stato assegnato l'indirizzo della versione myfunc(int a, int b).



In generale, quando si assegna a un puntatore a funzione l'indirizzo di una funzione modificata tramite overloading, è la dichiarazione del puntatore che determina la funzione il cui indirizzo verrà assegnato. Inoltre occorre notare che la dichiarazione del puntatore a funzione deve corrispondere esattamente a una e una sola delle dichiarazioni delle funzioni in overloading.

## 14.5 L'anacronismo della parola riservata overload

Quando venne creato il linguaggio C++, per creare una funzione modificata tramite overloading era necessario impiegare la parola riservata overload. Ora tale parola riservata è obsoleta e non viene più usata né supportata. Infatti non è neppure una parola riservata del linguaggio C++ standard. Tuttavia, poiché potrebbe capitare di incontrare qualche vecchio programma che la impiega, è utile conoscerne l'utilizzo. Ecco la forma generale della parola riservata overload:

overload *nome-funzione*;

Qui, nome-funzione è il nome della funzione di cui si sta eseguendo l'overloading. Questa istruzione deve precedere tutte le dichiarazioni della funzione modificata tramite overloading. Ad esempio, la seguente istruzione dice a un vecchio compilatore che si intende eseguire l'overloading di una funzione chiamata test():

```
overload test;
```

## 14.6 Gli argomenti standard delle funzioni

Il linguaggio C++ consente a una funzione di assegnare un valore standard a un parametro nel caso in cui nella chiamata a tale funzione non sia specificato alcun argomento corrispondente a tale parametro. Il valore standard è specificato in un modo sintatticamente simile all'inizializzazione di una variabile. Ad esempio, il seguente listato dichiara una funzione myfunc() che accetta un argomento double avente un valore standard uguale a 0.0:

```
void myfunc(double d = 0.0)
{
  // ...
}
```

Ora myfunc() può essere richiamata in due modi, come illustrato dai seguenti esempi:

```
myfunc(198.234); // passa un valore esplicito
myfunc();        // la funzione usa il default
```

La prima chiamata passa a d il valore 198.234. La seconda chiamata assegna automaticamente a d il valore 0.

Uno dei motivi che ha spinto a inserire gli argomenti standard nel linguaggio C++ è il fatto che essi costituiscono un metodo con il quale il programmatore può gestire elevati livelli di complessità. Per gestire la più ampia varietà di situazioni, capita frequentemente che una funzione contenga molti più parametri di quanti siano necessari nell'uso comune. Pertanto, quando vengono impiegati gli argomenti standard, è necessario specificare unicamente gli argomenti che hanno un significato nella situazione in cui ci si trova e non tutti gli argomenti previsti per il caso più generale. Ad esempio, molte delle funzioni di I/O del C++ utilizzano argomenti standard proprio per questo motivo.

Un semplice esempio dell'utilità degli argomenti standard delle funzioni è rappresentato dalla funzione clrscr() del seguente programma. Questa funzione cancella il contenuto dello schermo producendo in output una serie di codici di fine riga (line feed); non si tratta del metodo più efficiente ma è adatto per questo esempio. Poiché normalmente un monitor visualizza 25 righe di testo, viene impiegato un argomento standard pari a 25. Ma poiché alcuni terminali possono visualizzare un numero maggiore o inferiore di righe (spesso il valore dipende dalla modalità video impiegata), si può alterare l'argomento standard specificandone uno esplicitamente.

```
#include <iostream>
using namespace std;

void clrscr(int size=25);

int main()
{
  register int i;

  for(i=0; i<30; i++ ) cout << i << endl;
  cin.get();
  clrscr(); // cancella 25 righe

  for(i=0; i<30; i++ ) cout << i << endl;
  cin.get();
  clrscr(10); // cancella 10 righe
  return 0;
}

void clrscr(int size)
{
  for(; size; size--) cout << endl;
}
```



Come si può vedere in questo programma, quando l'argomento standard è appropriato alla situazione, non è necessario specificare alcun argomento quando viene richiamata clrscr(). Nel contempo resta possibile modificare il valore standard e assegnare a size un valore differente.

Un argomento standard può essere utilizzato anche per chiedere alla funzione di riutilizzare l'argomento precedente. Per illustrare questo uso, verrà sviluppata una funzione chiamata iputs() che fa rientrare automaticamente una stringa di un determinato numero di caratteri. Per iniziare ecco una versione di questa funzione che non utilizza alcun argomento standard:

```
void iputs(char *str, int indent)
{
  if(indent < 0) indent  0;

  for( ; indent; indent--) cout << " ";

  cout << str << "\n";
}
```

Questa versione di iputs() viene richiamata specificando come primo argomento la stringa da produrre in output e come secondo argomento l'entità del rientro. Anche se questo modo di scrivere iputs() non ha niente di sbagliato, se ne può migliorare l'utilizzabilità fornendo un parametro standard indent che chiede a iputs() di creare un rientro uguale al precedente. È infatti molto comune visualizzare un blocco di righe rientrate della stessa entità. In questa situazione, invece di dover fornire ripetutamente un argomento di indentazione, si può dare a indent un valore standard che chieda a iputs() di far rientrare le righe come nella chiamata precedente. Questo approccio è illustrato dal seguente programma:

```
#include <iostream>
using namespace std;

/* Rientro standard -1.  Questo valore dice alla funzione
   di riutilizzare il valore precedente. */
void iputs(char *str, int indent = -1);
```

```
int main()
{
  iputs("Salve a tutti", 10);
  iputs("Riutilizza il rientro di 10 spazi");
  iputs("Rientro di 5 spazi", 5);
  iputs("Non rientrato", 0);

  return 0;
}

void iputs(char *str, int indent)
{
  static i = 0; // memorizza il rientro precedente

  if(indent >= 0)
    i = indent;
  else  // riutilizza il valore di rientro precedente
    indent = i;

  for( ; indent; indent--) cout << " ";

  cout << str << "\n";
}
```

Questo programma produce il seguente output:

```
        Salve a tutti
        Riutilizza il rientro di 10 spazi
    Rientro di 5 spazi
Non rientrato
```

Quando si creano funzioni che hanno argomenti standard, è importante ricordare che i valori standard devono essere specificati una sola volta e questa deve essere la prima volta che la funzione viene dichiarata all'interno del file. Nell'esempio precedente, l'argomento standard è stato specificato nel prototipo di iputs(). Se si cerca di specificare un nuovo valore (o anche lo stesso valore) nella definizione di iputs(), il compilatore produrrà un messaggio d'errore e si rifiuterà di compilare il programma. Anche se non è possibile ridefinire gli argomenti standard della funzione, è comunque possibile specificare argomenti standard differenti per ciascuna versione di una funzione modificata tramite overloading.

Tutti i parametri che hanno valori standard devono comparire a destra di quelli che non prevedono valori standard. Ad esempio, è errato definire iputs() nel seguente modo:



```
// errato!
void iputs(int indent = -1, char *str);
```

Dopo aver iniziato a definire i parametri che accettano valori standard, non è più possibile specificare altri parametri che non prevedono valori standard. Pertanto anche la seguente dichiarazione è errata e non verrà compilata:

```
int myfunc(float f, char *str, int i=10, int j);
```

Poiché a i viene assegnato un valore standard, occorre assegnare un valore standard anche a j.

I parametri standard possono essere utilizzati anche nella funzione costruttore di un oggetto. Ad esempio, la classe cube illustrata di seguito gestisce le dimensioni di un cubo. Se non vengono specificati argomenti, la funzione costruttore assegna il valore 0 a tutti i valori, come illustrato dal seguente esempio:

```
#include <iostream>
using namespace std;

class cube {
  int x, y, z;
public:
  cube(int i=0, int j=0, int k=0) {
    x=i;
    y=j;
    z=k;
}

  int volume() {
    return x*y*z;
  }
};

int main()
{
  cube a(2,3,4), b;

  cout << a.volume() << endl;
  cout << b.volume();

  return 0;
}
```

L'uso degli argomenti standard (quando appropriato) offre due vantaggi in una funzione costruttore. Innanzitutto evita di dover fornire un costruttore modificato tramite overloading che non accetti alcun parametro. Ad esempio, se ai parametri di cube() non fosse stato assegnato un valore standard, il secondo costruttore avrebbe dovuto gestire la dichiarazione di b (che non contiene argomenti).

```
cube() {x=0; y=0; z=0}
```

In secondo luogo, il fatto di impiegare valori standard è decisamente più comodo rispetto al diversificare i valori ogni volta che l'oggetto viene dichiarato.

## Argomenti standard e overloading

In alcune situazioni, gli argomenti standard possono essere utilizzati come una forma semplificata di overloading. Il costruttore della classe cube è un esempio di questo tipo. Ora si proverà a vedere un altro esempio. Si immagini di voler creare due versioni personalizzate della funzione standard strcat(). La prima funzione si comporterà come strcat() concatenando l'intero contenuto di una stringa al termine dell'altra. La seconda versione accetta un terzo argomento che specifica il numero di caratteri da concatenare. Pertanto la seconda versione concatena alla fine di una stringa il numero di caratteri specificato tratto dall'altra stringa. Supponendo di chiamare le funzioni personalizzate con il nome mystrcat(), queste avranno i seguenti prototipi:

```
void mystrcat(char *s1, char *s2, int len);
void mystrcat(char *s1, char *s2);
```

La prima versione copierà len caratteri di s2 alla fine di s1. La seconda versione copierà l'intera stringa puntata da s2 alla fine della stringa puntata da s1 e dunque si comporterà come strcat().

Anche se non vi sarebbe nulla di errato a implementare due versioni di mystrcat() e creare le due versioni che si desiderano, esiste anche un modo più facile. Utilizzando un argomento standard è possibile implementare una sola versione di mystrcat() che svolge entrambe le operazioni. Si osservi ad esempio il seguente programma.

```
// Versione personalizzata di strcat().
#include <iostream>
#include <cstring>
using namespace std;

void mystrcat(char *s1, char *s2, int len = -1);

int main()
{
  char str1[80] = "Questa è una prova";
  char str2[80] = "0123456789";

  mystrcat(str1, str2, 5); // concatena 5 caratteri
  cout << str1 << '\n';

  strcpy(str1, "Questa è una prova"); // reinizializza di str1

  mystrcat(str1, str2); // concatena l'intera stringa
  cout << str1 << '\n';

  return 0;
}

// Versione personalizzata di strcat().
void mystrcat(char *s1, char *s2, int len)
{
  // Trova la fine di s1
  while(*s1) s1++;

  if(len == -1) len = strlen(s2);

  while(*s2 && len) {
    *s1 = *s2; // copia i caratteri
    s1++;
    s2++;
    len--;
}

  *s1 = '\0'; // Chiude la stringa s1
}
```

Qui, mystrcat() concatena alla fine della stringa puntata da s2, un numero di caratteri pari a len tratti dalla stringa puntata da s1. Se però len è uguale a -1, come nel caso previsto dall'argomento standard, mystrcat() concatena ad s1 l'intera stringa puntata da s2 (pertanto, quando len è uguale a -1 la funzione si comporta come la funzione strcat() standard). Utilizzando un argomento standard per len è possibile combinare entrambe le operazioni in una sola funzione. In questo modo, si nota che talvolta gli argomenti standard costituiscono un'alternativa all'overloading delle funzioni.

### Uso corretto degli argomenti standard

Anche se gli argomenti standard possono rappresentare uno strumento molto potente se utilizzati correttamente, possono talvolta essere impiegati in modo errato. Lo scopo degli argomenti standard è quello di consentire a una funzione di svolgere il proprio lavoro in modo efficiente e semplice, per aumentare considerevolmente la flessibilità. Pertanto tutti gli argomenti standard dovrebbero rappresentare l'uso più generale o ragionevole di una funzione. Quando non esiste un valore che venga normalmente associato a un parametro, non vi è alcun motivo per dichiarare un argomento standard. Infatti la dichiarazione di argomenti standard senza motivo riduce la strutturazione del codice in quanto spinge chiunque leggerà il programma a chiedersi i motivi di questa scelta.

Un'altra indicazione importante da seguire quando si impiegano gli argomenti standard è la seguente: nessun argomento standard dovrebbe provocare azioni pericolose o distruttive. In altre parole, un utilizzo accidentale di un argomento standard non deve provocare gravi danni.

## 14.7 Overloading di funzioni e ambiguità

Talvolta è possibile creare una situazione in cui il compilatore non riesce a scegliere fra due o più funzioni in overloading. In questi casi, si dice che la situazione è ambigua. Le istruzioni ambigue costituiscono errori e i programmi contenenti ambiguità non possono essere compilati. La causa principale di ambiguità riguarda le conversioni automatiche di tipo eseguite dal linguaggio C++. Come si sa, il linguaggio C++ tenta automaticamente di convertire gli argomenti utilizzati per richiamare una funzione nel tipo di argomenti attesi dalla funzione stessa. Ad esempio si consideri il seguente frammento di codice:

```
int myfunc(double d);
// ...
cout << myfunc('c'); // non è un errore, viene applicata la conversione
```

Come dice il commento, questo non è un errore poiché il linguaggio C++ converte automaticamente il carattere c nel suo equivalente double. In C++, sono molto poche le conversioni di questo tipo non consentite. Anche se le conversioni automatiche di tipo sono molto comode, rappresentano anche la causa principale di ambiguità. Ad esempio, si consideri il seguente programma:

```
#include <iostream>
using namespace std;
```



```
float myfunc(float i);
double myfunc(double i);

int main()
{
  cout << myfunc(10.1) << " "; // non ambigua, richiama myfunc(double)
  cout << myfunc(10); // ambigua

  return 0;
}

float myfunc(float i)
{
  return i;
}

double myfunc(double i)
{
  return -i;
}
```

Qui myfunc() viene modificata tramite overloading in modo da accettare argomenti di tipo float o double. Nella riga non ambigua, viene richiamata myfunc(double) poiché, tranne quando sono specificate esplicitamente come float, tutte le costanti in virgola mobile del C++ sono automaticamente di tipo double. Pertanto tale chiamata non è ambigua. Quando invece myfunc() viene richiamata con l'intero 10, si introduce un'ambiguità poiché il compilatore non può sapere se il valore dovrà essere convertito in float o double. Questo provoca la visualizzazione di un messaggio di errore e impedisce la compilazione del programma.

Come illustrato dall'esempio precedente, non è l'overloading di myfunc() per valori double o float a causare l'ambiguità. Piuttosto la responsabilità è della specifica chiamata a myfunc() che utilizza un argomento di tipo indeterminato. In altre parole, l'errore non è causato dall'overloading di myfunc() ma dalla chiamata alla funzione. Ecco un altro esempio di ambiguità provocata dalle conversioni automatiche di tipo del C++.

```
#include <iostream>
using namespace std;

char myfunc(unsigned char ch);
char myfunc(char ch);

int main()
{
```

```
  cout << myfunc('c'); // richiama myfunc(char)
  cout << myfunc(88) << " "; // ambigua

  return 0;
}

char myfunc(unsigned char ch)
{
  return ch-1;
}

char myfunc(char ch)
{
  return ch+1;
}
```

In C++, i tipi unsigned char e char non sono ambigui. Ma quando si richiama la funzione myfunc() utilizzando l'intero 88, il compilatore non sa quale funzione chiamare. In altre parole, il numero 88 deve essere convertito in char o in unsigned char? Un altro modo in cui può sorgere un'ambiguità si presenta quando vengono impiegati argomenti standard nelle funzioni modificate tramite overloading. Si provi a esaminare il seguente programma:

```
#include <iostream>
using namespace std;

int myfunc(int i);
int myfunc(int i, int j=1);

int main()
{
  cout << myfunc(4, 5) << " "; // non ambigua
  cout << myfunc(10); // ambigua

  return 0;
}

int myfunc(int i)
{
  return i;
}

int myfunc(int i, int j)
{
  return i*j;
}
```



Qui, nella prima chiamata a myfunc(), vengono specificati due argomenti; pertanto non vi è alcuna ambiguità e viene richiamata myfunc(int i, int j). Ma nella seconda chiamata a myfunc(), si verifica un'ambiguità poiché il compilatore non sa se richiamare la versione di myfunc() che accetta un argomento oppure applicare gli argomenti standard alla versione che accetta due argomenti.

Alcuni tipi di funzioni modificate tramite overloading sono invece inerentemente ambigue anche se, a prima vista, non sembrerebbe. Ad esempio, si consideri il seguente programma:

```
// Questo programma contiene un errore.
#include <iostream>
using namespace std;

void f(int x);
void f(int &x); // errore

int main()
{
  int a=10;

  f(a); // errore, quale f()?

  return 0;
}

void f(int x)
{
  cout << "In f(int)\n";
}

void f(int &x)
{
  cout << "In f(int &)\n";
}
```

Come si può dedurre dal commento, non è possibile eseguire l'overloading di due funzioni quando l'unica differenza consiste nel fatto che una accetta un parametro passato per indirizzo e l'altra accetta un normale parametro passato per valore. In questo caso, il compilatore non ha alcuna possibilità di sapere quale versione della funzione si intende richiamare. Si ricordi che non vi è alcuna differenza sintattica nel modo in cui un argomento viene specificato quando deve essere ricevuto per indirizzo o per valore.

Capitolo 15

# Overloading degli operatori


15.1 **Creazione di una funzione operator membro**

15.2 **Overloading di operatori tramite funzioni friend**

15.3 **Overloading di new e delete**

15.4 **Overloading di alcuni operatori particolari**

15.5 **Overloading dell'operatore virgola**


L'overloading degli operatori è strettamente correlato all'argomento dell'overloading delle funzioni. In C++ è possibile eseguire l'overloading della maggior parte degli operatori per consentire loro di svolgere operazioni particolari rispetto a determinate classi. Ad esempio, una classe che esegua la gestione di uno stack potrebbe eseguire l'overloading dell'operatore + per svolgere l'operazione push e dell'operatore - per svolgere l'operazione pop. Quando si esegue l'overloading di un operatore, non si perde nessuno dei suoi utilizzi originali. Al contrario, si estende lo spettro dei tipi ai quali può essere applicato l'operatore.

La possibilità di eseguire l'overloading degli operatori è una delle caratteristiche più potenti del linguaggio C++. Essa consente di integrare completamente nuovi tipi di classi nell'ambiente di programmazione. Se si esegue l'overloading degli operatori appropriati, sarà possibile utilizzare gli oggetti nelle espressioni esattamente come si utilizzano i tipi di dati standard del C++. L'overloading degli operatori sta anche alla base delle operazioni di I/O del C++.

L'overloading degli operatori viene eseguito creando funzioni operator. Una funzione operator definisce le specifiche operazioni che dovranno essere svolte dall'operatore modificato tramite overloading rispetto alla classe specificata. Le funzioni operator possono essere o meno funzioni membro della classe su cui operano. Le funzioni operator non membro sono quasi sempre funzioni friend della classe. Le funzioni operator membro e le funzioni operator friend devono essere realizzate in modo diverso. Pertanto, questi due argomenti verranno esaminati separatamente, iniziando dalle funzioni operator che sono membri della classe.

## 15.1 Creazione di una funzione operator membro

Le funzioni operator membro di una classe hanno la seguente forma generale:

*tipo-restituito nome-classe*::operator#(*elenco-argomenti*)
{
  *// operazioni*
}

Spesso, le funzioni operator restituiscono un oggetto della classe su cui operano ma *tipo-restituito* può essere un qualsiasi tipo valido. Il carattere # deve invece essere sostituito dall'operatore di cui si esegue l'overloading. Ad esempio, se si esegue l'overloading dell'operatore /, si dovrà utilizzare l'indicazione operator /. Quando si esegue l'overloading di un operatore unario, *elenco-argomenti* deve essere vuoto. Quando si esegue l'overloading di operatori binari, l'*elenco-argomenti* deve contenere un solo parametro (il motivo di ciò sarà chiaro tra breve).

Di seguito viene presentato un primo e semplice overloading degli operatori. Questo programma crea una classe chiamata loc che memorizza valori di longitudine e latitudine. Tale classe esegue l'overloading dell'operatore + rispetto a se stessa. Si esamini attentamente il programma, facendo particolare attenzione alla definizione di operator +().

```
#include <iostream>
using namespace std;

class loc {
  int longitude, latitude;
public:
  loc() {}
  loc(int lg, int lt) {
    longitude = lg;
    latitude = lt;
  }

  void show() {
    cout << longitude << " ";
    cout << latitude << "\n";
  }

  loc operator+(loc op2);
};

// Overloading di + per loc
loc loc::operator+(loc op2)
```



```
{
  loc temp;

  temp.longitude = op2.longitude + longitude;
  temp.latitude = op2.latitude + latitude;

  return temp;
}

int main()
{
  loc ob1(10, 20), ob2( 5, 30);

  ob1.show(); // visualizza 10 20
  ob2.show(); // visualizza 5 30

  ob1 = ob1 + ob2;
  ob1.show(); // visualizza 15 50

  return 0;
}
```

Come si può vedere, operator+() ha un solo parametro anche se esegue l'overloading dell'operatore binario + (invece dei due parametri corrispondenti ai due operandi dell'operatore binario). Ciò è dovuto al fatto che l'operando sul lato sinistro del + viene passato implicitamente alla funzione utilizzando il puntatore this. L'operando a destra viene passato nel parametro op2. Il fatto che l'operando di sinistra venga passato utilizzando this ha un'altra implicazione fondamentale: quando si esegue l'overloading di operatori binari, sarà l'oggetto a sinistra a generare la chiamata alla funzione operator.

Come si è detto, è molto facile che una funzione operator modificata tramite overloading restituisca un oggetto della classe su cui opera. In questo modo consente all'operatore di essere utilizzato in espressioni C++ più complesse. Ad esempio, se la funzione operator+() restituisse un valore di altro tipo, questa espressione non sarebbe valida:

```
ob1 = ob1 + ob2;
```

Per assegnare a ob1 la somma di ob1 e ob2, il risultato dell'operazione deve essere un oggetto di tipo loc.

Inoltre, poiché operator+() restituisce un oggetto di tipo loc è possibile utilizzare istruzioni simili alla seguente:

```
(ob1+ob2).show(); // visualizza il risultato di ob1+ob2
```

In questa situazione, ob1+ob2 genera un oggetto temporaneo che cessa di esistere al termine della chiamata a show().

È importante comprendere che una funzione operator può restituire un valore di qualsiasi tipo e che il tipo del valore restituito dipende unicamente dall'utilizzo che il programmatore intende farne. Semplicemente, molto spesso una funzione operator restituisce un oggetto della classe su cui opera.

Un'ultima annotazione relativa alla funzione operator+(): tale funzione non modifica gli operandi. Poiché l'uso tradizionale dell'operatore + non modifica gli operandi, ha senso creare versioni tramite overloading che conservino questa caratteristica (ad esempio 5+7 fornisce il valore 12 ma senza modificare né 5 né 7). Anche se all'interno di una funzione operator si può eseguire qualsiasi operazione, è in genere meglio considerare il contesto in cui viene utilizzato l'operatore "normale".

Il programma successivo aggiunge alla classe loc tre nuovi operatori: -, = e l'operatore unario ++. Si faccia particolare attenzione al modo in cui sono definite queste funzioni.

```
#include <iostream>
using namespace std;

class loc {
  int longitude, latitude;
public:
  loc() {} // necessario per costruire valori temporanei
  loc(int lg, int lt) {
    longitude = lg;
    latitude = lt;
  }

  void show() {
    cout << longitude << " ";
    cout << latitude << "\n";
  }

  loc operator+(loc op2);
  loc operator-(loc op2);
  loc operator=(loc op2);
  loc operator++();
};

// Overloading di + per loc
loc loc::operator+(loc op2)
{
  loc temp;

  temp.longitude = op2.longitude + longitude;
  temp.latitude = op2.latitude + latitude;

  return temp;
}

// Overloading di - per loc
loc loc::operator-(loc op2)
{
  loc temp;

  // si noti l'ordine degli operandi
  temp.longitude = longitude - op2.longitude;
  temp.latitude = latitude - op2.latitude;

  return temp;
}

// Overloading dell'assegnamento per loc
loc loc::operator=(loc op2)
{
  longitude = op2.longitude;
  latitude = op2.latitude;

  return *this; // ovvero restituisce l'oggetto che ha generato la chiamata
}

// Overloading  di ++ (prefisso) per loc
loc loc::operator++()
{
  longitude++;
  latitude++;

  return *this;
}

int main()
{
  loc ob1(10, 20), ob2( 5, 30), ob3(90, 90);

  ob1.show();
  ob2.show();

  ++ob1;
  ob1.show(); // visualizza 11 21
```

```
  ob2 = ++ob1;
  ob1.show(); // visualizza 12 22
  ob2.show(); // visualizza 12 22

  ob1 = ob2 = ob3; // assegnamento multiplo
  ob1.show(); // visualizza 90 90
  ob2.show(); // visualizza 90 90

  return 0;
}
```

Innanzitutto, si esamini la funzione operator-(). Si noti l'ordine degli operandi nella sottrazione. Per conservare il significato della sottrazione, l'operando sul lato destro del segno meno viene sottratto dall'operando che si trova a sinistra. Poiché è l'oggetto a sinistra che genera la chiamata alla funzione operator-(), i dati di op2 devono essere sottratti dai dati puntati da this. È importante ricordare l'operando che genera la chiamata alla funzione.

In C++, se il segno = non viene modificato tramite overloading, verrà automaticamente creato un operatore di assegnamento standard per ogni classe definita. L'assegnamento standard è costituito da una copia membro a membro e bit a bit. Eseguendo l'overloading dell'operatore = è possibile definire esplicitamente ciò che l'assegnamento dovrà eseguire per una determinata classe. In questo esempio, l'operatore = modificato tramite overloading esegue esattamente la stessa operazione dell'operatore standard ma in altre situazioni potrebbe eseguire operazioni diverse. Si noti che la funzione operator=() restituisce *this, ovvero l'oggetto che ha generato la chiamata. Questo accorgimento è necessario per consentire il concatenamento di più assegnamenti come nel seguente esempio:

```
ob1 = ob2 = ob3;  // assegnamento multiplo
```

Ora si osservi la definizione di operator++(). Come si può vedere, l'operatore non ha parametri. Poiché ++ è un operatore unario, il suo unico operando verrà passato implicitamente utilizzando il puntatore this.

Si noti che entrambe le funzioni operator=() e operator++() modificano il valore di un operando. Nel caso dell'assegnamento, viene assegnato un nuovo valore all'operando a sinistra (quello che genera la chiamata alla funzione operator=()). Nel caso di ++, l'operando viene incrementato. Come si è detto precedentemente, anche se si è liberi di far eseguire a queste funzioni le operazioni desiderate, è sempre bene che le nuove operazioni siano congruenti rispetto al significato originale dell'operatore.



## Creazione di forme prefisse e postfisse degli operatori di incremento e decremento

Nel programma precedente, è stato eseguito l'overloading solo della forma prefissa dell'operatore di incremento. Nella versione standard del C++ è possibile creare esplicitamente versioni prefisse e postfisse degli operatori di incremento e decremento. A tale scopo si devono definire due versioni della funzione operator++(). Una sarà definita come illustrato nel precedente programma mentre l'altra sarà dichiarata nel seguente modo:

```
loc operator++(int x);
```

Se il ++ precede il suo operando, verrà richiamata la funzione operator++(). Se il ++ segue il proprio operando, verrà richiamata la funzione operator++(int x) e x avrà il valore 0.

L'esempio precedente può essere generalizzato. Ecco l'aspetto generale delle forme prefissa e postfissa degli operatori ++ e --:

```
// Incremento prefisso
tipo operator++( ) {
   // corpo dell'operatore prefisso
}
// Incremento postfisso
tipo operator++(int x) {
  // corpo dell'operatore postfisso
}
// Decremento prefisso
tipo operator- - ( ) {
   // corpo dell'operatore prefisso
}
// Decremento postfisso
tipo operator- - (int x) {
  // corpo dell'operatore postfisso
}
```

**NOTA** *Si deve fare attenzione quando si interviene su vecchi programmi C++ che eseguono l'overloading degli operatori ++ e --, in quanto nei vecchi compilatori era impossibile definire versioni prefisse e postfisse differenti degli operatori. In entrambi i casi veniva utilizzata la forma prefissa.*

### Overloading delle forme abbreviate degli operatori

In C++ è possibile eseguire l'overloading anche delle forme abbreviate degli operatori, come ad esempio +=, -= e di tutte le forme analoghe. Ad esempio, questa funzione esegue l'overloading dell'operatore += rispetto alla classe loc:

```
loc loc::operator+=(loc op2)
{
  longitude = op2.longitude + longitude;
  latitude = op2.latitude + latitude;

  return *this;
}
```

Quando si esegue l'overloading di uno di questi operatori, si deve ricordare che si sta semplicemente unificando un assegnamento con un altro tipo di operatore.

### Restrizioni all'overloading degli operatori

L'overloading degli operatori deve sottostare ad alcune restrizioni. Non è possibile modificare la precedenza di un operatore. Non si può cambiare il numero degli operandi richiesti da un operatore (anche se si può decidere di ignorare un operando). Ad eccezione dell'operatore di chiamata a funzione (descritto più avanti), le funzioni operator non possono avere argomenti standard. Infine non è possibile eseguire l'overloading dei seguenti operatori:

. :: .* ?

Tecnicamente all'interno di una funzione operator vi è la possibilità di eseguire qualsiasi operazione. Ad esempio, si può eseguire l'overloading dell'operatore + in modo che scriva per dieci volte una frase sul disco. Tuttavia se si altera significativamente il normale significato di un operatore, si corre il rischio di distruggere la chiarezza di un programma. Ad esempio, quando qualcuno leggendo il programma troverà un'istruzione come Ob1+Ob2 si attenderà che debba avere luogo un qualche genere di somma e non un accesso a disco. Pertanto, prima di alterare il significato di un operatore, si consideri se vi sono motivi tali da rendere assolutamente necessaria questa operazione. Un ottimo esempio in cui era conveniente alterare il significato degli operatori è costituito dagli operatori di I/O << e >>. Anche se le operazioni di I/O non hanno alcuna relazione con lo scorrimento dei bit, tali operatori forniscono un'immagine visiva del loro significato rispetto sia all'operazione di I/O che allo scorrimento di bit e quindi l'alterazione del significato ha senso. In generale è però consigliabile conservare il contesto del normale significato dell'operatore.



Escludendo l'operatore =, le funzioni operator vengono ereditate da tutte le classi derivate. Tuttavia, una classe derivata è libera di eseguire l'overloading di qualsiasi operatore (inclusi quelli modificati tramite overloading in una classe base).

## 15.2 Overloading di operatori tramite funzioni friend

Si può eseguire l'overloading di un operatore per una classe anche utilizzando una funzione non membro, normalmente tramite una funzione friend. Poiché una funzione friend non è un membro della classe, non potrà impiegare il puntatore this. Pertanto, a una funzione di overloading friend operator gli operandi devono essere passati esplicitamente. Questo significa che una funzione friend che esegua l'overloading di un operatore binario richiede due parametri mentre una funzione friend che esegua l'overloading di un operatore unario avrà bisogno di un parametro. Quando si esegue l'overloading di un operatore binario con una funzione friend, l'operando di sinistra deve essere passato come primo parametro e l'operando di destra deve essere passato come secondo parametro.

Nel seguente programma, l'operatore + viene modificato tramite overloading utilizzando una funzione friend.

```
#include <iostream>
using namespace std;

class loc {
  int longitude, latitude;
public:
  loc() {} // necessario per costruire valori temporanei
  loc(int lg, int lt) {
    longitude = lg;
    latitude = lt;
  }

  void show() {
    cout << longitude << " ";
    cout << latitude << "\n";
  }

  friend loc operator+(loc op1, loc op2); // ora è una friend
  loc operator-(loc op2);
  loc operator=(loc op2);
  loc operator++();
};
```

```
// + viene modificato tramite una funzione di overloading friend
loc operator+(loc op1, loc op2)
{
  loc temp;

  temp.longitude = op1.longitude + op2.longitude;
  temp.latitude = op1.latitude + op2.latitude;

  return temp;
}

// Overloading di -- per loc
loc loc::operator-(loc op2)
{
  loc temp;

  // si noti l'ordine degli operandi
  temp.longitude = longitude - op2.longitude;
  temp.latitude = latitude - op2.latitude;

  return temp;
}

// Overloading dell'assegnamento per loc
loc loc::operator=(loc op2)
{
  longitude = op2.longitude;
  latitude = op2.latitude;

  return *this; // ovvero restituisce l'oggetto che ha generato la chiamata
}

// Overloading di ++ per loc
loc loc::operator++()
{
  longitude++;
  latitude++;

  return *this;
}

int main()
{
  loc ob1(10, 20), ob2( 5, 30);

  ob1 = ob1 + ob2;
```



```
  ob1.show();

  return 0;
}
```

L'impiego di funzioni friend operator impone alcune restrizioni. Innanzi tutto, utilizzando una funzione friend non è possibile eseguire l'overloading degli operatori =, (), [] o ->. In secondo luogo, come verrà descritto nella prossima sezione, se si esegue l'overloading degli operatori di incremento o decremento utilizzando una funzione friend, si dovrà utilizzare un parametro indirizzo.

## Uso di una funzione friend per eseguire l'overloading degli operatori ++ e --

Se si desidera utilizzare una funzione friend per eseguire l'overloading degli operatori di incremento o decremento, si deve passare l'operando come parametro indirizzo. Questo è dovuto al fatto che le funzioni friend non hanno il puntatore this. Per conservare il significato originale degli operatori ++ e - -, queste operazioni devono modificare il loro operando. Tuttavia, se si esegue l'overloading di questi operatori utilizzando una funzione friend, l'operando verrà passato per valore. Questo significa che una funzione friend operator non ha alcuna possibilità di modificare l'operando. Poiché alla funzione friend operator non viene passato un puntatore this all'operando ma una copia dell'operando, nessuna modifica apportata al parametro modificherà l'operando che ha generato la chiamata. È possibile risolvere questa situazione specificando il parametro della funzione friend operator come un parametro indirizzo. In questo modo ogni modifica apportata al parametro all'interno della funzione modificherà anche l'operando che ha generato la chiamata. Ad esempio, il seguente programma utilizza funzioni friend per eseguire l'overloading delle versioni prefisse di ++ e - - rispetto alla classe loc.

```
#include <iostream>
using namespace std;

class loc {
  int longitude, latitude;
public:
  loc() {}
  loc(int lg, int lt) {
    longitude = lg;
    latitude = lt;
  }

  void show() {
    cout << longitude << " ";
```

```
    cout << latitude << "\n";
  }

  loc operator=(loc op2);
  friend loc operator++(loc &op);
  friend loc operator--(loc &op);
};

// Overloading dell'assegnamento per loc
loc loc::operator=(loc op2)
{
  longitude = op2.longitude;
  latitude = op2.latitude;

  return *this; // ovvero restituisce l'oggetto che ha generato la chiamata
}

// ora è una friend - utilizza un parametro indirizzo
loc operator++(loc &op)
{
  op.longitude++;
  op.latitude++;

  return op;
}

// rende op-- un friend - usa un indirizzo
loc operator--(loc &op)
{
  op.longitude--;
  op.latitude--;

  return op;
}

int main()
{
  loc ob1(10, 20), ob2;

  ob1.show();
  ++ob1;
  ob1.show(); // visualizza 11 21

  ob2 = ++ob1;
  ob2.show(); // visualizza 12 22
```



```
  --ob2;
  ob2.show(); // visualizza 11 21
  return 0;
}
```

Per eseguire l'overloading delle versioni postfisse degli operatori di incremento e decremento utilizzando una funzione friend basta specificare un secondo parametro intero fittizio. Ad esempio, ecco il prototipo delle versioni postfisse friend dell'operatore di incremento rispetto alla classe loc:

```
// friend, versione postfissa di ++
friend loc operator++(loc &op, int x);
```

## Le funzioni friend operator offrono una maggiore flessibilità

In molti casi, se si esegue l'overloading di un operatore utilizzando una funzione friend o una funzione membro, non si riscontrano differenze funzionali. In tali casi è generalmente consigliabile eseguire l'overloading utilizzando funzioni membro. Vi è però una situazione in cui l'uso di una funzione friend aumenta la flessibilità di un operatore.

Come si è detto in precedenza, quando si esegue l'overloading di un operatore binario utilizzando una funzione membro, è l'oggetto sul lato sinistro dell'operatore a generare la chiamata della funzione operator che esegue l'overloading. Inoltre viene passato tramite il puntatore this un puntatore a tale oggetto. Ora, si supponga di utilizzare una classe CL nella quale sia stata definita la funzione membro operator+() che somma un oggetto della classe e un intero. Dato un oggetto Ob di tale classe, la seguente espressione sarà corretta.

```
Ob + 100 // corretta
```

In questo caso è Ob a generare la chiamata alla funzione che esegue l'overloading di + e viene eseguita la somma. Ma cosa sarebbe accaduto se l'espressione fosse stata scritta nel seguente modo?

```
100 + Ob // errata
```

In questo caso, sul lato sinistro dell'operatore appare l'intero. Poiché l'intero è un tipo predefinito, non si troverà alcuna operazione definita fra un intero e un oggetto del tipo di Ob. Pertanto, il compilatore non accetterà l'espressione. Come si può immaginare, in alcune applicazioni il fatto di dover sempre posizionare l'oggetto a sinistra pone limiti inutili oltre a essere fonte di errori.

La soluzione del problema precedente è costituita dall'overloading di una somma utilizzando una funzione friend al posto della funzione membro. In tal modo, alla funzione operator verranno esplicitamente passati entrambi gli argomenti. Pertanto, per consentire l'uso delle espressioni *oggetto+intero* e *intero+oggetto* basterà eseguire due volte l'overloading della funzione: una versione per ogni situazione. Quindi, quando si esegue l'overloading di un operatore utilizzando due funzioni friend l'oggetto potrà trovarsi sia sul lato sinistro che anche sul lato destro dell'operatore.

Questo programma illustra il modo in cui è possibile utilizzare funzioni friend per definire un'operazione che interessi un oggetto e un tipo predefinito.

```
#include <iostream>
using namespace std;

class loc {
  int longitude, latitude;
public:
  loc() {}
  loc(int lg, int lt) {
    longitude = lg;
    latitude = lt;
  }

  void show() {
    cout << longitude << " ";
    cout << latitude << "\n";
  }

  friend loc operator+(loc op1, int op2);
  friend loc operator+(int op1, loc op2);
};

// Overloading di + per loc + int
loc operator+(loc op1, int op2)
{
  loc temp;

  temp.longitude = op1.longitude + op2;
  temp.latitude = op1.latitude + op2;

  return temp;
}

// Overloading di + per int + loc
loc operator+(int op1, loc op2)
{
  loc temp;

  temp.longitude = op1 + op2.longitude;
  temp.latitude = op1 + op2.latitude;

  return temp;
}

int main()
{
  loc ob1(10, 20), ob2( 5, 30), ob3(7, 14);

  ob1.show();
  ob2.show();
  ob3.show();

  ob1 = ob2 + 10;  // entrambe queste forme
  ob3 = 10 + ob2;  // sono corrette

  ob1.show();
  ob3.show();

  return 0;
}
```



## 15.3 Overloading di new e delete

In C++ è possibile eseguire l'overloading anche di new e delete. Questo potrebbe consentire di utilizzare particolari metodi di allocazione della memoria. Ad esempio, si potrebbe aver bisogno di routine di allocazione che inizino automaticamente a utilizzare un file su disco come memoria virtuale nel caso la memoria dello heap fosse esaurita. Qualunque sia il motivo, è molto semplice eseguire l'overloading di questi operatori.

Di seguito viene presentata la struttura di base delle funzioni che eseguono l'overloading di new e di delete.

```
// Alloca un oggetto
void *operator new(size_t size)
{
  /* esegue l'allocazione e in caso di fallimento lancia l'eccezione
bad_alloc
     Viene automaticamente richiamato il costruttore */
```

```
  return puntatore_alla_memoria;
}

// Cancella un oggetto
void operator delete(void *p)
{
  /* libera la memoria puntata da p
     Viene automaticamente richiamato il distruttore */
}
```

Il tipo size_t è definito come un tipo in grado di contenere la più ampia area di memoria allocabile e corrisponde in pratica al tipo intero unsigned. Il parametro size contiene il numero di byte necessari per contenere l'oggetto da allocare. Questa è la quantità di memoria che deve essere allocata new. La funzione new, dopo l'overloading, deve restituire un puntatore alla memoria allocata oppure in caso di errore deve lanciare un'eccezione bad_alloc. Considerando questi unici vincoli, la nuova funzione new può eseguire qualsiasi altra operazione. Quando si alloca la memoria per un oggetto con new (la versione di base o una versione personalizzata) viene automaticamente richiamato il costruttore dell'oggetto.

La funzione delete riceve un puntatore alla regione di memoria da rendere nuovamente disponibile per il sistema.

Gli operatori new e delete possono essere modificati con un overloading globale in modo che vengano sempre utilizzate le versioni modificate. Alternativamente l'overloading può riferirsi solo a una o più classi. Per iniziare si vedrà un esempio delle funzioni new e delete modificate rispetto a una classe. Per semplicità non verrà utilizzato uno schema di allocazione completamente nuovo. Gli operatori modificati tramite overloading richiameranno semplicemente le funzioni standard malloc() e free() (in generale il programmatore deve invece implementare uno schema di allocazione alternativo). Per eseguire l'overloading degli operatori new e delete per una classe, basta rendere le funzioni operator di overloading membri della classe. Ad esempio, ecco il modo in cui è possibile eseguire l'overloading degli operatori new e delete per la classe loc:

```
#include <iostream>
#include <cstdlib>
#include <new>
using namespace std;

class loc {
  int longitude, latitude;
public:
  loc() {}
  loc(int lg, int lt) {
```



```
    longitude = lg;
    latitude = lt;
  }

  void show() {
    cout << longitude << " ";
    cout << latitude << "\n";
  }

  void *operator new(size_t size);
  void operator delete(void *p);
};

// new modificato rispetto a loc
void *loc::operator new(size_t size)
{
  void *p;

  cout << "Nuovo new\n";
  p = malloc(size);
  if (!p) {
    bad_alloc ba;
    throw ba;
  }
  return p;
}

// delete modificato rispetto a loc
void loc::operator delete(void *p)
{
  cout << "Nuovo delete\n";
  free(p);
}

int main()
{
  loc *p1, *p2;

  try {
    p1 = new loc (10, 20);
  } catch (bad_alloc xa) {
    cout << "Errore di allocazione per p1\n";
    return 1;
  }

  try {
```

```
    p2 = new loc (10, 20);
  } catch (bad_alloc xa) {
    cout << "Errore di allocazione per p2\n";
    return 1;
  }

  p1->show();
  p2->show();

  delete p1;
  delete p2;

  return 0;
}
```

Ecco l'output prodotto dal programma:

```
Nuovo new
Nuovo new


10 20
-10 -20
Nuovo delete
Nuovo delete
```

Quando new e delete vengono modificate tramite overloading per una determinata classe, l'uso di tali operatori su altri tipi provocherà la chiamata degli operatori originali. Gli operatori modificati tramite overloading verranno applicati solo ai tipi per i quali sono stati definiti. Questo significa che se si aggiunge a main() la seguente riga, verrà utilizzato l'operatore new standard.

```
int *f = new float;  // usa il new standard
```

Per eseguire un overloading globale degli operatori new e delete basta eseguire l'overloading all'esterno della dichiarazione di qualsiasi classe. Quando new e delete vengono modificati con un overloading generale, gli operatori new e delete standard del C++ verranno sempre ignorati e per tutte le richieste verranno impiegati i nuovi operatori. Naturalmente, se sono state definite nuove versioni di new e delete relative a una o più classi, per allocare gli oggetti della classe verranno sempre utilizzate le versioni specifiche. In altre parole, quando si incontrerà un new o un delete, il compilatore controllerà innanzi tutto se esiste una versione dell'operatore specifica per la classe su cui sta operando. In caso affermativo verranno utilizzate le versioni



specifiche. Se non esiste una versione specifica, il C++ utilizzerà le versioni new e delete definite globalmente. Se le versioni globali sono state modificate tramite overloading, verranno impiegate le versioni modificate.

Per vedere un esempio di overloading globale di new e delete, si esamini il seguente programma:

```
#include <iostream>
#include <cstdlib>
#include <new>
using namespace std;

class loc {
  int longitude, latitude;
public:
  loc() {}
  loc(int lg, int lt) {
    longitude = lg;
    latitude = lt;
  }

  void show() {
    cout << longitude << " ";
    cout << latitude << "\n";
  }
};

// new globale
void *operator new(size_t size)
{
  void *p;

  p = malloc(size);
  if (!p) {
    bad_alloc ba;
    throw ba;
  }
  return p;
}

// delete globale
void operator delete(void *p)
{
  free(p);
}

int main()
```

```
{
  loc *p1, *p2;
  float *f;

  try {
    p1 = new loc (10, 20);
  } catch (bad_alloc xa) {
    cout << "Errore di allocazione per p1\n";
    return 1;
  }

  try {
    p2 = new loc (10, 20);
  } catch (bad_alloc xa) {
    cout << "Errore di allocazione per p2\n";
    return 1;
  }

  try {
    f = new float; // usa la versione modificata di new
  } catch (bad_alloc xa) {
    cout << "Errore di allocazione per f\n";
    return 1;
  }

  *f = 10.10;
  cout << *f << "\n";

  p1->show();
  p2->show();

  delete p1;
  delete p2;
  delete f;  // usa la versione modificata di delete

  return 0;
}
```

Si provi a eseguire questo programma per dimostrare il funzionamento dell'overloading globale degli operatori new e delete.

## Overloading di new e delete per l'impiego con array

Se si vuole essere in grado di allocare array di oggetti utilizzando un proprio sistema di allocazione, sarà necessario eseguire un secondo overloading degli operatori new e delete. Per allocare e deallocare array, si dovranno utilizzare queste nuove versioni di new e delete:



```
//-Alloca un array di oggetti.
void *operator new[](size_t size)
{
  /* Esegue l'allocazione. In caso di fallimento lancia l'eccezione bad_alloc
     Viene automaticamente richiamato il costruttore di ciascun elemento */
  return puntatore_alla_memoria;
}

// Cancella un array di oggetti.
void operator delete[](void *p)
{
  /* Libera la memoria puntata da p.
     Viene automaticamente richiamato
     il distruttore di ciascun elemento.
  */
}
```

Quando viene allocato un array, viene automaticamente richiamata la funzione costruttore per ogni oggetto dell'array. Nel momento in cui l'array viene deallocato, viene automaticamente richiamato il distruttore su ogni oggetto dell'array. Quindi non sarà necessario utilizzare codice specifico per eseguire tali azioni.

Il programma seguente alloca e poi dealloca un oggetto e un array di oggetti di tipo loc.

```
#include <iostream>
#include <cstdlib>
#include <new>
using namespace std;

class loc {
  int longitude, latitude;
public:
  loc() {longitude = latitude = 0;}
  loc(int lg, int lt) {
    longitude = lg;
    latitude = lt;
  }

  void show() {
    cout-<< longitude-<< " ";
```

```
    cout << latitude << "\n";
  }

  void *operator new(size_t size);

  void operator delete(void *p);

  void *operator new[](size_t size);

  void operator delete[](void *p);
};

// new modificato rispetto a loc
void *loc::operator new(size_t size)
{
  void *p;

  cout << "Nuovo new\n";
  p = malloc(size);
  if (!p) {
    bad_alloc ba;
    throw ba;
  }
  return p;
}

// delete modificato rispetto a loc
void loc::operator delete(void *p)
{
  cout << "Nuovo delete\n";
  free(p);
}

// new modificato rispetto ad array di loc
void *loc::operator new[](size_t size)
{
  void *p;

  cout << "Nuovo new\n";
  p = malloc(size);
  if (!p) {
    bad_alloc ba;
    throw ba;
  }
  return p;
}
```



```
// delete modificato rispetto a loc per gli array
void loc::operator delete[](void *p)
{
  cout << "Cancellazione dell'array con il nuovo delete[]\n";
  free(p);
}

int main()
{
  loc *p1, *p2;
  int i;

  try {
    p1 = new loc (10, 20); // alloca un oggetto
  } catch (bad_alloc xa) {
    cout << "Errore di allocazione per p1\n";
    return 1;
  }

  try {
    p2 = new loc [10];     // alloca un array
  } catch (bad_alloc xa) {
    cout << "Errore di allocazione per p2\n";
    return 1;
  }

  p1->show();

  for(i=0; i<10; i++)
    p2[i].show();

  delete p1;    // libera un oggetto
  delete [] p2; // libera un array

  return 0;
}
```

## Overloading della versione nothrow di new e delete

È anche possibile eseguire l'overloading delle versioni nothrow di new e delete. Ecco come dovrà essere la struttura del programma

```
// versione nothrow di new.
void *operator new(size_t size, const nothrow_t &n)
```

```
{
  // Esegue l'allocazione.
  if(successo) return puntatore_alla_memoria;
  else return 0;
}

// versione nothrow di new per gli array.
void *operator new[](size_t size, const nothrow_t &n)
{
  // Esegue l'allocazione.
  if(successo) return puntatore_alla_memoria;
  else return 0;
}

void operator delete(void *p, const nothrow_t &n)
{
  // liberare la memoria
}

void operator delete[](void *p, const nothrow_t &n)
{
  // liberare la memoria
}
```

Il tipo nothrow_t è definito in <new>. Questo è il tipo dell'oggetto nothrow. Il parametro nothrow_t non viene utilizzato.

## 15.4 Overloading di alcuni operatori particolari

Il C++ definisce l'indicizzazione di array, la chiamata di funzioni e l'accesso ai membri di una classe come operazioni. Gli operatori che eseguono queste funzioni sono rispettivamente [], () e ->. Anche questi operatori possono essere modificati tramite overloading in C++, dando origine ad alcuni utilizzi molto interessanti.

Nell'overloading di questi operatori si applica un'importante restrizione: le funzioni di overloading devono essere funzioni membro non statiche. In particolare non possono essere funzioni friend.

### Overloading di []

In C++, quando si esegue l'overloading, la coppia [] è considerata un operatore binario. Pertanto, la forma generale di una funzione membro operator[]() è la seguente:



```
tipo nome-classe::operator[ ](int i)
{
  //...
}
```

Tecnicamente, il parametro non deve necessariamente essere di tipo int ma una funzione operator[]() viene utilizzata per fornire l'indice di un array e pertanto viene normalmente utilizzato un valore intero.

Dato un oggetto chiamato O l'espressione:

```
O[3]
```

si traduce nella seguente chiamata alla funzione operator[]():

```
O.operator[](3)
```

Ovvero, al parametro esplicito della funzione operator[]() viene passato il valore dell'espressione che si trova all'interno dell'operatore di indicizzazione. Il puntatore this punterà a O, l'oggetto che ha generato la chiamata.

Nel programma seguente, atype dichiara un array di tre interi. La sua funzione costruttore inizializza ogni membro dell'array. La funzione di overloading operator[]() restituisce il valore dell'array che si trova alla posizione indicata dal valore del suo parametro.

```
#include <iostream>
using namespace std;

class atype {
  int a[3];
public:
  atype(int i, int j, int k) {
    a[0] = i;
    a[1] = j;
    a[2] = k;
  }
  int operator[](int i) {return a[i];}
};

int main()
{
  atype ob(1, 2, 3);

  cout << ob[1]; // visualizza 2
```

```
  return 0;
}
```

È possibile realizzare la funzione operator[]() in modo tale che la coppia [] possa essere utilizzata sia sul lato sinistro che sul lato destro di un'istruzione di assegnamento. A tale scopo, basta specificare il valore restituito da operator[]() come un indirizzo. Il seguente programma apporta questa modifica e ne mostra l'utilizzo.

```
#include <iostream>
using namespace std;

class atype {
  int a[3];
public:
  atype(int i, int j, int k) {
    a[0] = i;
    a[1] = j;
    a[2] = k;
  }
  int &operator[](int i) {return a[i];}
};

int main()
{
  atype ob(1, 2, 3);

  cout << ob[1];  // visualizza 2
  cout << " ";

  ob[1] = 25;  // [] alla sinistra di =

  cout << ob[1];  // ora visualizza 25

  return 0;
}
```

Poiché ora operator[]() restituisce un indirizzo all'elemento dell'array indicato da i, la coppia [] può essere utilizzata anche sul lato sinistro di un'istruzione di assegnamento in modo da modificare un elemento dell'array (naturalmente continua a poter essere utilizzata sul lato destro di un'istruzione di assegnamento).

La possibilità di eseguire l'overloading di un operatore [] consente di implementare in C++ un metodo di indicizzazione "sicura" degli array. Come si sa, in C++ è possibile superare i limiti superiore o inferiore di un array senza che venga generato alcun messaggio di errore al momento dell'esecuzione.



Se invece si crea una classe che contiene l'array e che consente di accedere a tale array solo attraverso l'operatore di indicizzazione [] modificato tramite overloading, sarà possibile intercettare tutte le richieste che superano i limiti dell'array. Ad esempio, il seguente programma aggiunge la verifica dei limiti al programma precedente.

```
// Un esempio di array sicuro.
#include <iostream>
#include <cstdlib>
using namespace std;

class atype {
  int a[3];
public:
  atype(int i, int j, int k) {
    a[0] = i;
    a[1] = j;
    a[2] = k;
  }
  int &operator[](int i);
};

// Verifica i limiti per atype.
int &atype::operator[](int i)
{
  if(i<0 || i> 2) {
    cout << "Superamento dei limiti\n";
    exit(1);
  }
  return a[i];
}

int main()
{
  atype ob(1, 2, 3);

  cout << ob[1];  // visualizza 2
  cout << " ";

  ob[1] = 25;  // [] si trova a sinistra
  cout << ob[1];  // visualizza 25

  ob[3] = 44; // genera un errore run-time perché 3 è oltre i limiti
  return 0;
}
```

In questo programma, quando viene eseguita l'istruzione:

```
ob[3] = 44;
```

l'errore di superamento dei limiti viene intercettato da operator[]() e il programma verrà fermato prima che possa provocare danni (nella pratica, dovrà essere realizzata una funzione per la gestione dell'errore che si occupi della condizione di superamento dei limiti e che consenta al programma di continuare).

## Overloading di ()

Quando si esegue l'overloading dell'operatore di chiamata a funzione (), non si sta in realtà creando un nuovo modo per richiamare una funzione. Piuttosto si sta creando una funzione operator alla quale è possibile passare un numero arbitrario di parametri. Per iniziare ecco un esempio: data la dichiarazione della funzione operator:

```
double operator()(int a, float f, char *s);
```

e un oggetto O di tale classe, l'istruzione:

```
O(10, 23.34, "hi");
```

provocherà questa chiamata alla funzione operator():

```
O.operator()(10, 23.34, "hi");
```

In generale, quando si esegue l'overloading dell'operatore (), si definiscono i parametri che si intendono passare a tale funzione. Quando si utilizza l'operatore () in un programma, in tali parametri vengono copiati gli argomenti specificati. Come sempre, il puntatore this punta all'oggetto che ha generato la chiamata (in questo esempio O).

Ecco un esempio di overloading dell'operatore () rispetto alla classe loc. In questo esempio il valore dei due argomenti viene assegnato alla longitudine e alla latitudine dell'oggetto a cui viene applicato l'operatore.

```
#include <iostream>
using namespace std;

class loc {
  int longitude, latitude;
public:
  loc() {}
  loc(int lg, int lt) {
    longitude = lg;
    latitude = lt;
  }

  void show() {
    cout << longitude << " ";
    cout << latitude << "\n";
  }

  loc operator+(loc op2);

  loc operator()(int i, int j);
};

// Overloading di () per loc
loc loc::operator()(int i, int j)
{
  longitude = i;
  latitude = j;

  return *this;
}

// Overloading di + per loc
loc loc::operator+(loc op2)
{
  loc temp;

  temp.longitude = op2.longitude + longitude;
  temp.latitude = op2.latitude + latitude;

  return temp;
}

int main()
{
  loc ob1(10, 20), ob2(1, 1);

  ob1.show();
  ob1(7, 8);  // può essere utilizzata da sola ...
  ob1.show();

  ob1 = ob2 + ob1(10, 10);  // ... o all'interno di espressioni
  ob1.show();
```

```
  return 0;
}
```

Ecco l'output prodotto dal programma:

```
10 20
7 8
11 11
```

**NOTA** *Quando si esegue l'overloading di () è possibile utilizzare qualsiasi tipo di parametri e restituire un valore di qualsiasi tipo. In pratica tali tipi dipendono dalle richieste del programma. È anche possibile specificare degli argomenti standard.*

### Overloading dell'operatore ->

In caso di overloading l'operatore di accesso ai membri della classe -> è considerato un operatore unario. Di seguito viene presentato il suo utilizzo generale:

*oggetto->elemento*;

Qui, *oggetto* è l'oggetto che attiva la chiamata. La funzione operator->() deve restituire un puntatore a un oggetto della classe su cui opera operator->(). *elemento* deve essere un membro accessibile dall'interno dell'oggetto.

Il seguente programma illustra l'overloading dell'operatore -> mostrando l'equivalenza esistente fra ob.i e ob->i quando operator->() restituisce il puntatore this.

```
#include <iostream>
using namespace std;

class myclass {
public:
  int i;
  myclass *operator->() {return this;}
};

int main()
{
  myclass ob;

  ob->i = 10; // uguale a ob.i
```



```
  cout << ob.i << " " << ob->i;

  return 0;
}
```

Una funzione operator->() deve essere membro della classe su cui opera.

## 15.5 Overloading dell'operatore virgola

In C++ è possibile anche eseguire l'overloading dell'operatore virgola. La virgola è un operatore binario e, come per ogni altro operatore, l'overloading può farle eseguire qualsiasi operazione. Ma se si vuole che anche dopo l'overloading l'operatore virgola esegua un'operazione analoga a quella originaria, esso dovrà liberarsi del valore di tutti gli operandi ad eccezione di quello più a destra, il quale diverrà il risultato dell'operazione di virgola. Come si sa, questo è il modo in cui la virgola funziona nel C++ standard. Ecco un programma che illustra l'effetto dell'overloading dell'operatore virgola:

```
#include <iostream>
using namespace std;

class loc {
  int longitude, latitude;
public:
  loc() {}
  loc(int lg, int lt) {
    longitude = lg;
    latitude = lt;
  }

  void show() {
    cout << longitude << " ";
    cout << latitude << "\n";
  }

  loc operator+(loc op2);
  loc operator,(loc op2);
};

// Overloading dell'operatore virgola per loc
loc loc::operator,(loc op2)
{
  loc temp;
```

```
  temp.longitude = op2.longitude;
  temp.latitude = op2.latitude;
  cout << op2.longitude << " " << op2.latitude << "\n";

  return temp;
}

// Overloading di + per loc
loc loc::operator+(loc op2)
{
  loc temp;

  temp.longitude = op2.longitude + longitude;
  temp.latitude = op2.latitude + latitude;

  return temp;
}

int main()
{
  loc ob1(10, 20), ob2( 5, 30), ob3(1, 1);

  ob1.show();
  ob2.show();
  ob3.show();
  cout << "\n";

  ob1 = (ob1, ob2+ob2, ob3);

  ob1.show();  // visualizza 1 1, il valore di ob3
  return 0;
}
```

Questo programma visualizza il seguente output:

```
10 20
5 30
1 1

10 60
1 1
1 1
```



Si noti che anche se vengono eliminati tutti i valori degli operandi a sinistra, il compilatore valuterà comunque ogni espressione e quindi verrà eseguito qualsiasi effetto collaterale predisposto dal programmatore.

Si ricordi che l'operando di sinistra viene passato tramite il puntatore this e che il suo valore viene eliminato utilizzando la funzione operator(). La funzione restituisce il valore dell'operando più a destra. In questo modo anche dopo l'overloading la virgola si comporta in modo analogo alla sua operazione standard. Se si desidera che dopo l'overloading l'operatore virgola esegua una diversa operazione, si dovranno modificare queste due funzionalità.

Capitolo 16

# L'ereditarietà


16.1 **Controllo dell'accesso alla classe base**
16.2 **Ereditarietà dei membri protected**
16.3 **Ereditarietà da più classi base**
16.4 **Costruttori, distruttori ed ereditarietà**
16.5 **Accesso alle classi**
16.6 **Classi base virtuali**


L'ereditarietà costituisce una delle pietre angolari della programmazione orientata agli oggetti in quanto consente di creare classificazioni gerarchiche. Utilizzando l'ereditarietà, è possibile creare una classe generale che definisce le caratteristiche comuni a una serie di oggetti correlati. Questa classe può in seguito essere ereditata da una o più classi, ognuna delle quali aggiunge alla classe ereditata solo elementi specifici.

Per conservare la terminologia standard del C++, la classe ereditata viene chiamata *classe base*. La classe che "riceve" l'eredità è detta *classe derivata*. A sua volta, una classe derivata può fungere da classe base per un'altra classe derivata. In questo modo è possibile riprodurre a più livelli il meccanismo di ereditarietà.

Il supporto dell'ereditarietà del C++ è ricco e flessibile. Il meccanismo di ereditarietà, introdotto nel Capitolo 11 forma l'argomento di questo capitolo.

## 16.1 Controllo dell'accesso alla classe base

Quando una classe ne eredita un'altra, i membri della classe base divengono membri della classe derivata. L'ereditarietà delle classi ha la seguente forma generale:

```
class nome-classe-derivata : accesso nome-classe-base {
  // corpo della classe
};
```

Il tipo di accesso che la classe derivata può avere sui membri della classe base è determinato dallo specificatore *accesso*. Lo specificatore d'accesso della classe base può essere public, private o protected.

Se non si indica uno specificatore d'accesso, si possono verificare i seguenti casi: se la classe derivata è una class, lo specificatore d'accesso sarà private; se la classe derivate è una struct, lo specificatore d'accesso standard sarà public. Ora verranno esaminate le implicazioni dell'uso degli specificatori public e private (lo specificatore protected verrà esaminato nella prossima sezione).

Quando lo specificatore d'accesso alla classe base è public, tutti i membri pubblici della classe base diverranno membri pubblici della classe derivata e tutti i membri protetti della classe base diverranno membri protetti anche della classe derivata.

In tutti i casi, gli elementi privati della classe base rimarranno privati della classe base e non saranno pertanto accessibili da parte dei membri della classe derivata. Ad esempio, come si può vedere nel seguente programma, gli oggetti di tipo derived possono accedere direttamente ai membri pubblici di base.

```
#include <iostream>
using namespace std;

class base {
  int i, j;
public:
  void set(int a, int b) {i=a; j=b;}
  void show() { cout << i << " " << j << "\n";}
};

class derived : public base {
  int k;
public:
  derived(int x) {k=x;}
  void showk() {cout << k << "\n";}
};

int main()
{
  derived ob(3);

  ob.set(1, 2); // accesso a un membro di base
  ob.show();  // accesso a un membro di base

  ob.showk(); // usa un membro della classe derivata

  return 0;
}
```



Quando la classe base è ereditata tramite lo specificatore d'accesso private, tutti i membri pubblici e protetti della classe base diverranno membri privati della classe derivata. Ad esempio, il programma seguente non potrà nemmeno essere compilato poiché sia set() che show() sono ora elementi privati di derived.

```
// Questo programma non verrà compilato.

#include <iostream>
using namespace std;

class base {
  int i, j;
public:
  void set(int a, int b) {i=a; j=b;}
  void show() { cout << i << " " << j << "\n";}
};

// Gli elementi pubblici di base sono privati in derived.
class derived : private base {
  int k;
public:
  derived(int x) {k=x;}
  void showk() {cout << k << "\n";}
};

int main()
{
  derived ob(3);

  ob.set(1, 2); // errore: non si può accedere a set()
  ob.show(); // errore: non si può accedere a show()

  return 0;
}
```

**SUGGERIMENTO** *Quando lo specificatore d'accesso di una classe base è private, i membri pubblici e protetti della classe base divengono membri privati della classe derivata. Questo significa che rimarranno accessibili da parte dei membri della classe derivata e inaccessibili da parte di altri punti del programma che non siano membri della classe base o della classe derivata.*

## 16.2 Ereditarietà dei membri protected

La parola riservata protected è stata inclusa nel C++ per introdurre un maggior livello di flessibilità nel meccanismo di ereditarietà. Quando un membro di una classe è dichiarato protected, tale membro non sarà accessibile da parte di altri elementi non membri della classe. Con un'importante eccezione, l'accesso ai membri protetti equivale all'accesso ai membri privati ovvero l'accesso può avvenire solo da parte dei membri della classe. L'unica eccezione si verifica quando viene ereditato un membro protected. In questo caso, un membro protected è molto diverso da un membro private.

Come si è detto nella sezione precedente, un membro private di una classe base non è accessibile da altre parti del programma, incluse tutte le classi derivate. I membri protected si comportano in modo diverso: se la classe base è ereditata come public, i membri protected della classe base diverranno membri protected della classe derivata e saranno pertanto accessibili alla classe derivata stessa. In altre parole, utilizzando la parola riservata protected, un programmatore può creare membri della classe che siano privati di tale classe ma che possano essere ereditati e manipolati dalle classi derivate. Ecco un esempio:

```
#include <iostream>
using namespace std;

class base {
protected:
  int i, j; // privato di base, ma accessibile da derived
public:
  void set(int a, int b) {i=a; j=b;}
  void show() { cout << i << " " << j << "\n";}
};

class derived : public base {
  int k;
public:
  // derived può accedere alle variabili i e j di base
  void setk() {k=i*j;}

  void showk() {cout << k << "\n";}
};

int main()
{
  derived ob;

  ob.set(2, 3); // OK, noto in derived
  ob.show(); // OK, noto in derived

  ob.setk();
  ob.showk();
  return 0;
}
```



Qui, poiché base è ereditata da derived come public e poiché i e j sono dichiarati protected, la funzione setk() della classe derived può avere accesso a i e j. Se i e j fossero state dichiarate come private di base, la classe derived non avrebbe potuto eseguire accessi e non sarebbe stato neppure possibile compilare il programma.

Quando una classe derivata è utilizzata come classe base di un'altra classe derivata, tutti i membri protected della classe base iniziale ereditati (come public) dalla prima classe derivata potranno essere ereditati nuovamente come protected anche dalla seconda classe derivata. Ad esempio, il seguente programma è corretto e derived2 potrà avere accesso a i e a j.

```
#include <iostream>
using namespace std;

class base {
protected:
  int i, j;
public:
  void set(int a, int b) {i=a; j=b;}
  void show() { cout << i << " " << j << "\n";}
};

// i e j ereditati come protected.
class derived1 : public base {
  int k;
public:
  void setk() {k = i*j;} // consentito
  void showk() {cout << k << "\n";}
};

// i e j sono ereditate indirettamente tramite derived1.
class derived2 : public derived1 {
  int m;
public:
  void setm() {m = i-j;} // consentito
  void showm() {cout << m << "\n";}
};
```

```
int main()
{
  derived1 ob1;
  derived2 ob2;

  ob1.set(2, 3);
  ob1.show();
  ob1.setk();
  ob1.showk();

  ob2.set(3, 4);
  ob2.show();
  ob2.setk();
  ob2.setm();
  ob2.showk();
  ob2.showm();

  return 0;
}
```

Se invece base fosse stata ereditata come private, tutti i membri di base sarebbero divenuti membri privati di derived1 e sarebbero pertanto risultati inaccessibili da parte di derived2 (anche se i e j sarebbero rimasti accessibili da parte di derived1). Questa situazione è illustrata dal seguente programma che contiene un errore e non verrà compilato. Gli errori sono descritti dai commenti.

```
// Questo programma non verrà compilato.

#include <iostream>
using namespace std;

class base {
protected:
  int i, j;
public:
  void set(int a, int b) {i=a; j=b;}
  void show() { cout << i << " " << j << "\n";}
};

// Ora, tutti gli elementi di base sono privati di derived1.
class derived1 : private base {
  int k;
public:
  // consentito, poiché i e j sono private di derived1
  void setk() {k = i*j;} // OK
  void showk() {cout << k << "\n";}
};

// L'accesso a i, j, set() e show() non viene ereditato.
class derived2 : public derived1 {
  int m;
public:
  // non consentito poiché i e j sono private di derived1
  void setm() {m = i-j;} // errore
  void showm() {cout << m << "\n";}
};

int main()
{
  derived1 ob1;
  derived2 ob2;

  ob1.set(1, 2); // errore, non si può usare set()
  ob1.show(); // errore, non si può usare show()

  ob2.set(3, 4); // errore, non si può usare set()
  ob2.show(); // errore, non si può usare show()

  return 0;
}
```



**NOTA** *Anche se base è ereditata come private da derived1, quest'ultima potrà comunque avere accesso agli elementi public e protected di base anche se non potrà trasmettere questo privilegio.*

## Ereditarietà di una classe base come protected

In C++ è possibile ereditare una classe base come protected. In tal caso, tutti i membri public e protected della classe base diverranno membri protected della classe derivata. Ecco un esempio:

```
#include <iostream>
using namespace std;

class base {
protected:
  int i, j; // privato di base, ma accessibile da derived
public:
  void setij(int a, int b) {i=a; j=b;}
```

```
  void showij() { cout << i << " " << j << "\n";}
};

// Eredita base come protected.
class derived : protected base{
  int k;
public:
  // derived può accedere a i, j e setij() di base.
  void setk() {setij(10, 12); k = i*j;}

  // qui può accedere a showij()
  void showall() {cout << k << " "; showij();}
};

int main()
{
  derived ob;

//  ob.setij(2, 3);  // non consentito, setij() è
//                      membro protected di derived

  ob.setk(); // OK, membro pubblico di derived
  ob.showall(); // OK, membro pubblico di derived

// ob.showij(); // non consentito: showij() è un membro
//                 protected di derived

  return 0;
}
```

Come si può vedere leggendo i commenti, anche se setij() e showij() sono membri pubblici di base, divengono membri protected di derived poiché sono stati ereditati utilizzando lo specificatore d'accesso protected. Questo significa che main() non avrà accesso a tali funzioni.

## 16.3 Ereditarietà da più classi base

Una classe derivata può ricevere in eredità elementi di due o più classi base. In questo breve esempio, derived eredita sia da base1 che da base2:

```
// Un esempio di classi base multiple.

#include <iostream>
```



```
using namespace std;

class base1 {
protected:
  int x;
public:
  void showx() {cout << x << "\n";}
};
class base2 {
protected:
  int y;
public:
  void showy() {cout << y << "\n";}
};

// Eredita da più classi base.
class derived: public base1, public base2 {
public:
  void set(int i, int j) {x=i; y=j;}
};

int main()
{
  derived ob;

  ob.set(10, 20); // fornita da derived
  ob.showx();     // da base1
  ob.showy();     // da base2

  return 0;
}
```

Come si può vedere nell'esempio, per ereditare da più di una classe base, si deve utilizzare un elenco di classi separate da virgole. Inoltre, si deve utilizzare uno specificatore d'accesso per ogni classe ereditata.

## 16.4 Costruttori, distruttori ed ereditarietà

Vi sono due problemi principali derivanti dall'uso di costruttori e distruttori quando si impiega l'ereditarietà. Innanzi tutto, in quale momento vengono richiamate le funzioni costruttore e distruttore della classe base e della classe derivata? E poi, come è possibile passare parametri alle funzioni costruttore della classe base? Questa sezione esamina questi due importanti argomenti.

## Quando vengono eseguite le funzioni costruttore e distruttore

È possibile che una classe base, una classe derivata o entrambe contengano funzioni costruttore e/o distruttore. È importante comprendere l'ordine in cui vengono eseguite tali funzioni nel momento in cui si crea o si distrugge un oggetto della classe derivata. Per iniziare, si esamini questo breve programma:

```
#include <iostream>
using namespace std;

class base {
public:
  base() {cout << "Costruzione di base\n";}
  ~base() {cout << "Distruzione di base\n";}
};

class derived: public base {
public:
  derived() {cout << "Costruzione di derived\n";}
  ~derived() {cout << "Distruzione di derived\n";}
};

int main()
{
  derived ob;

  // non fa nulla tranne costruire e distruggere ob

  return 0;
}
```

Come si può vedere dal commento contenuto in main(), questo programma costruisce e quindi distrugge un oggetto chiamato ob appartenente alla classe derived. Il programma produce il seguente output:

```
Costruzione di base
Costruzione di derived
Distruzione di derived
Distruzione di base
```

Come si può vedere, innanzi tutto viene eseguito il costruttore della classe base seguito dal costruttore di derived. Quindi (poiché in questo programma ob viene distrutto immediatamente), viene richiamato il distruttore di derived seguito da quello di base.



Il risultato di questo esperimento può essere generalizzato. Quando viene creato un oggetto di una classe derivata, se la classe base contiene un costruttore, questo verrà richiamato per primo seguito dal costruttore della classe derivata. Quando viene distrutto un oggetto derivato, prima viene richiamato il suo distruttore seguito dal distruttore della classe base (sempre che esista). In altre parole, le funzioni costruttore sono eseguite nell'ordine di derivazione e le funzioni distruttore sono eseguite in ordine inverso rispetto a quello di derivazione.

Se si riflette sull'argomento ha senso che le funzioni costruttore vengano eseguite nell'ordine di derivazione. Poiché una classe base non conosce l'aspetto di una classe derivata, ogni operazione di inizializzazione che dovrà eseguire dovrà anche essere distinta e precedere qualsiasi inizializzazione eseguita dalla classe derivata. Pertanto dovrà essere eseguita per prima.

Analogamente è piuttosto ovvio che i distruttori vengano eseguiti in ordine inverso di derivazione. Poiché la classe base funge da fondamento per la classe derivata, la distruzione dell'oggetto base implica la distruzione dell'oggetto derivato. Pertanto, il distruttore derivato dovrà essere richiamato prima che l'oggetto venga completamente distrutto.

In caso di ereditarietà a più livelli (ovvero quando una classe derivata diviene la classe base di un'altra classe derivata) si applica la regola generale: i costruttori vengono richiamati in ordine di derivazione e i distruttori in ordine inverso. Ad esempio, questo programma:

```
#include <iostream>
using namespace std;

class base {
public:
  base() {cout << "Costruzione di base\n";}
  ~base() {cout << "Distruzione di base\n";}
};

class derived1 : public base {
public:
  derived1() {cout << "Costruzione di derived1\n";}
  ~derived1() {cout << "Distruzione di derived1\n";}
};

class derived2: public derived1 {
public:
  derived2() {cout << "Costruzione di derived2\n";}
  ~derived2() {cout << "Distruzione di derived2\n";}
};

int main()
```

```
{
  derived2 ob;

  // costruisce e distrugge ob

  return 0;
}
```

produce il seguente output:

```
Costruzione di base
Costruzione di derived1
Costruzione di derived2
Distruzione di derived2
Distruzione di derived1
Distruzione di base
```

La stessa regola generale si applica in tutte le situazioni che prevedono l'uso di più classi base. Ad esempio, questo programma:

```
#include <iostream>
using namespace std;

class base1 {
public:
  base1() {cout << "Costruzione di base1\n";}
  ~base1() {cout << "Distruzione di base1\n";}
};

class base2 {
public:
  base2() {cout << "Costruzione di base2\n";}
  ~base2() {cout << "Distruzione di base2\n";}
};

class derived: public base1, public base2 {
public:
  derived() {cout << "Costruzione di derived\n";}
  ~derived() {cout << "Distruzione di derived\n";}
};

int main()
{
  derived ob;

  // costruisce e distrugge ob

  return 0;
}
```



produce il seguente output:

```
Costruzione di base1
Costruzione di base2
Costruzione di derived
Distruzione di derived
Distruzione di base2
Distruzione di base1
```

Come si può vedere, i costruttori vengono richiamati in ordine di derivazione (da sinistra a destra) così come è specificato nell'elenco di ereditarietà di derived. I distruttori vengono richiamati in ordine inverso (da destra verso sinistra). Questo significa che se base2 fosse stata specificata prima di base1 nell'elenco di derived, come nell'esempio seguente:

```
class derived: public base2, public base1 {
```

l'output del programma avrebbe avuto il seguente aspetto:

```
Costruzione di base2
Costruzione di base1
Costruzione di derived
Distruzione di derived
Distruzione di base1
Distruzione di base2
```

## Passaggio di parametri ai costruttori della classe base

Fino ad ora, nessuno degli esempi ha introdotto funzioni costruttore che richiedessero argomenti. In tutti i casi in cui solo il costruttore della classe derivata richieda uno o più parametri, basterà utilizzare la sintassi standard dei costruttori parametrizzati (vedere il Capitolo 12). Ma come è possibile passare argomenti a una funzione costruttore situata nella classe base? La risposta prevede l'uso di una forma estesa di dichiarazione del costruttore della classe derivata che consente di passare argomenti a uno o più costruttori della classe base.

La forma generale della dichiarazione estesa di un costruttore di una classe derivata è la seguente:

*costruttore-derivata(elenco-argomenti) : base1(elenco-argomenti),*
*base2(elenco-argomenti),*
.
.
.
*baseN(elenco-argomenti)*
{
*// corpo del costruttore derivato*
}

Qui, i nomi da *base1* a *baseN* sono i nomi delle classi base ereditate dalla classe derivata. Si noti il carattere di due punti che separa la dichiarazione del costruttore della classe derivata dalle indicazioni delle classi base e si noti anche che, se vi sono più classi base, i loro nomi sono separati da virgole. Si consideri il seguente programma:

```
#include <iostream>
using namespace std;

class base {
protected:
  int i;
public:
  base(int x) {i=x; cout << "Costruzione di base\n";}
  ~base() {cout << "Distruzione di base\n";}
};

class derived: public base {
  int j;
public:
  // derived usa x; y viene passata a base.
  derived(int x, int y): base(y)
    {j=x; cout << "Costruzione di derived\n";}

  ~derived() {cout << "Distruzione di derived\n";}
  void show() {cout << i << " " << j << "\n";}
};

int main()
{
  derived ob(3, 4);

  ob.show();  // visualizza 4 3
  return 0;
}
```

Qui, il costruttore di derived viene dichiarato in modo da accettare due parametri, x e y. Tuttavia, derived utilizza solo x; y viene passata a base(). In generale, il costruttore della classe derivata deve dichiarare tutti i propri parametri e tutti quelli richiesti dalla classe base. Come si vede nell'esempio, tutti i parametri richiesti dalla classe base vengono passati ad essa nell'elenco degli argomenti della classe specificati dopo i due punti.

Ecco un esempio che utilizza più classi base:

```
#include <iostream>
using namespace std;

class base1 {
protected:
  int i;
public:
  base1(int x) {i=x; cout << "Costruzione di base1\n";}
  ~base1() {cout << "Distruzione di base1\n";}
};

class base2 {
protected:
  int k;
public:
  base2(int x) {k=x; cout << "Costruzione di base2\n";}
  ~base2() {cout << "Distruzione di base1\n";}
};

class derived: public base1, public base2 {
  int j;
public:
  derived(int x, int y, int z): base1(y), base2(z)
    {j=x; cout << "Costruzione di derived\n";}

  ~derived() {cout << "Distruzione di derived\n";}
  void show() {cout << i << " " << j << " " << k << "\n";}
};

int main()
{
  derived ob(3, 4, 5);
```

```
  ob.show();  // visualizza 4 3 5

  return 0;
}
```

È importante comprendere che gli argomenti per il costruttore della classe base vengono passati attraverso gli argomenti del costruttore della classe derivata. Pertanto, anche se un costruttore della classe derivata non utilizza argomenti, ne dovrà comunque dichiarare uno nel caso in cui la classe base lo richieda. In questa situazione, gli argomenti passati alla classe derivata vengono semplicemente trasportati verso la classe base senza essere modificati. Ad esempio, nel seguente programma il costruttore della classe derivata non richiede argomenti i quali vengono passati a base1() e base2().

```
#include <iostream>
using namespace std;

class base1 {
protected:
  int i;
public:
  base1(int x) {i=x; cout << "Costruzione di base1\n";}
  ~base1() {cout << "Distruzione di base1\n";}
};

class base2 {
protected:
  int k;
public:
  base2(int x) {k=x; cout << "Costruzione di base2\n";}
  ~base2() {cout << "Distruzione di base2\n";}
};

class derived: public base1, public base2 {
public:
  /* Il costruttore di derived non richiede parametri,
     ma deve essere dichiarato in questo modo
     per passare gli argomenti alle classi base.
  */

  derived(int x, int y): base1(x), base2(y)
    {cout << "Costruzione di derived\n";}

  ~derived() {cout << "Distruzione di derived\n";}
  void show() {cout << i << " " << k << "\n";}
};

int main()
{
  derived ob(3, 4);

  ob.show();  // visualizza 3 4

  return 0;
}
```



La funzione costruttore di una classe derivata è libera di utilizzare tutti i parametri dichiarati, anche nel caso in cui questi debbano essere passati a una classe base. In altre parole, il passaggio di un argomento a una classe base, non impedisce di utilizzarlo anche dalla classe derivata, Ad esempio, questo frammento di codice è perfettamente corretto:

```
class derived: public base {
  int j;
public:
  // derived usa sia x che y e poi li passa a base.
  derived(int x, int y): base(x, y)
    {j = x*y; cout << "Costruzione di derived\n";}
```

Un'ultima annotazione relativa al passaggio di argomenti ai costruttori di una classe base: l'argomento può essere costituito da qualsiasi espressione valida. Questo significa che gli argomenti possono essere chiamate a funzioni e variabili. Questo è in linea con il fatto che il C++ consente l'inizializzazione dinamica.

## 16.5 Accesso alle classi

Quando una classe base è ereditata come private, tutti i suoi membri pubblici e protected divengono membri private della classe derivata. In alcuni casi però si desidera ripristinare le specifiche di accesso originali di uno o più dei membri ereditati. Ad esempio, si potrebbe voler assicurare ad alcuni membri pubblici della classe base lo status di public nella classe derivata anche se la classe base è stata ereditata come private.

Nel C++ standard si può ottenere questo in due modi. Innanzitutto si può utilizzare un'istruzione using (che è la soluzione preferibile). L'istruzione using ha principalmente lo scopo di garantire il supporto dei namespace e verrà descritta nel Capitolo 23. Il secondo modo per ripristinare le specifiche di accesso di un

membro ereditato consiste nell'impiegare una dichiarazione di accesso nella classe derivata. Anche se le dichiarazioni di accesso sono ancora supportate dal C++ standard, se ne sconsiglia l'uso. Questo significa che non dovrebbero essere utilizzate nella realizzazione di nuovo codice. Ma poiché vi sono ancora molti programmi che impiegano queste dichiarazioni di accesso, è opportuno esaminare anche questa possibilità.

Una dichiarazione di accesso ha la seguente forma generale:

*classe-base::membro;*

La dichiarazione di accesso deve seguire l'intestazione di accesso appropriata nella dichiarazione della classe derivata. Si noti che nella dichiarazione di accesso non è richiesta (né consentita) alcuna dichiarazione di tipo.

Per vedere come funziona la dichiarazione di accesso, si inizierà con un breve frammento di codice:

```
class base {
public:
  int j;  // pubblica in base
};

// Eredita base come privata.
class derived: private base {
public:

  // ecco la dichiarazione di accesso
  base::j; // rende j nuovamente pubblica
  .
  .
  .
};
```

Poiché base è ereditata da derived come private, il membro pubblico j diviene in derived un membro privato. Ma se si utilizza:

```
base::j;
```

che dovrà essere inclusa come dichiarazione di accesso sotto l'intestazione public della classe derived, j tornerà al suo status di public.

Una dichiarazione di accesso consente anche di ripristinare i diritti di accesso dei membri public e protected. Non è invece possibile utilizzare una dichiarazione di accesso per innalzare o abbassare lo status di accesso di un membro. Ad esempio, un membro dichiarato come private di una classe base non può essere reso



public da una classe derivata (se il C++ consentisse ciò, verrebbe distrutto il meccanismo di incapsulamento).

Il seguente programma illustra l'uso della dichiarazione di accesso. Si noti come vengano utilizzate le dichiarazioni di accesso per ripristinare lo status public di j, seti() e geti().

```
#include <iostream>
using namespace std;

class base {
  int i;  // privata di base
public:
  int j, k;
  void seti(int x) {i = x;}
  int geti() {return i;}
};

// Eredita base come privata.
class derived: private base {
public:
  /* Le tre istruzioni seguenti ignorano
     il fatto che base venga ereditata come private,
     e ripristinano l'accesso pubblico di j, seti() e geti(). */
  base::j; // rende j (ma non k) nuovamente pubblica
  base::seti; // rende seti() pubblica
  base::geti; // rende geti() pubblica

// base::i;  // non consentito: non si può elevare l'accesso

  int a; // public
};

int main()
{
  derived ob;

//ob.i = 10;  // non consentito: i è privata in derived

  ob.j = 20;  // consentito poiché j è resa pubblica in derived
//ob.k = 30;  // non consentito poiché k è privata in derived

  ob.a = 40;  // consentito poiché a è pubblica in derived
  ob.seti(10);

  cout << ob.geti() << " " <<ob.j << " " << ob.a;
```

```
  return 0;
}
```

In C++ le dichiarazioni di accesso sono ammesse poiché consentono di risolvere situazioni in cui la maggior parte degli elementi di una classe ereditata deve rimanere privata mentre alcuni membri possono conservare in alcuni casi lo status public o protected.

SUGGERIMENTO *Il C++ standard consente ma sconsiglia l'uso delle dichiarazioni di accesso. Questo significa che anche se ora sono consentite, potrebbero non essere più presenti nelle versioni future dello standard. Lo standard suggerisce di ottenere lo stesso effetto applicando la parola riservata using.*

## 16.6 Classi base virtuali

Quando vengono ereditate più classi base in un programma viene introdotto un elemento di ambiguità. Ad esempio, si consideri il seguente programma (errato):

```
// Questo programma contiene un errore e non verrà compilato.
#include <iostream>
using namespace std;

class base {
public:
  int i;
};

// derived1 eredita base.
class derived1 : public base {
public:
  int j;
};

// derived2 eredita base.
class derived2 : public base {
public:
  int k;
};

/* derived3 eredita sia derived1 che derived2.
   Questo significa che in derived3 vi sono
   due copie di base! */
class derived3 : public derived1, public derived2 {
public:
  int sum;
};

int main()
{
  derived3 ob;

  ob.i = 10;  // ambiguità: quale i???
  ob.j = 20;
  ob.k = 30;

  // anche qui i è ambigua
  ob.sum = ob.i + ob.j + ob.k;

  // ancora ambigua, quale i?
  cout << ob.i << " ";

  cout << ob.j << " " << ob.k << " ";
  cout << ob.sum;

  return 0;
}
```



Come si vede dai commenti, la classe base è ereditata sia da derived1 che da derived2. In seguito derived3 eredita sia derived1 che derived2. Questo significa che in un oggetto di tipo derived3 sono presenti due copie di base. Pertanto, in un'espressione come:

```
ob.i = 20;
```

a quale i si fa riferimento? A quella di derived1 o a quella di derived2? Poiché in un oggetto di tipo ob sono presenti due copie di base, sono anche presenti due variabili ob.i! Come si può vedere, l'istruzione è per sua natura ambigua.

Vi sono due modi per risolvere la situazione generata dal precedente programma. La prima consiste nell'applicare alla variabile i l'operatore di risoluzione del campo d'azione e nella selezione manuale di una i. Ad esempio, la seguente versione del programma verrà compilata e funzionerà nel modo atteso:

```
// Questo programma seleziona una i con l'operatore di risoluzione del campo
d'azione.
#include <iostream>
using namespace std;
```

```
class base {
public:
  int i;
};

// derived1 eredita base.
class derived1 :  public base {
public:
  int j;
};

// derived2 eredita base.
class derived2 : public base {
public:
  int k;
};

/* derived3 eredita sia derived1 che derived2.
   Questo significa che in derived3 vi sono
   due copie di base! */
class derived3 : public derived1, public derived2 {
public:
  int sum;
};

int main()
{
  derived3 ob;

  ob.derived1::i = 10;  // viene usata la i di derived1
  ob.j = 20;
  ob.k = 30;

  // risolta l'ambiguità
  ob.sum = ob.derived1::i + ob.j + ob.k;

  // risolta anche qui
  cout << ob.derived1::i << " ";

  cout << ob.j << " " << ob.k << " ";
  cout << ob.sum;

  return 0;
}
```



Come si può vedere, grazie all'impiego dell'operatore ::, il programma ha selezionato manualmente la versione di base contenuta in derived2. Ma questa soluzione porta alla luce un problema più grave: come fare per utilizzare una sola copia di base? Vi è un modo per evitare che in derived3 vengano incluse due copie della classe base? La soluzione prevede l'uso di classi base virtuali.

Quando due o più oggetti sono derivati da una classe base comune, è possibile evitare che in un oggetto derivato da questi oggetti siano presenti più copie della classe base, dichiarando la classe base come virtual nel momento in cui viene ereditata. A tale scopo si deve far precedere al nome della classe base la parola riservata virtual. Ad esempio, ecco un'altra versione del programma di esempio in cui derived3 contiene una sola copia di base:

```
// Questo programma usa classi base virtuali.
#include <iostream>
using namespace std;

class base {
public:
  int i;
};

// derived1 eredita base come virtual.
class derived1 : virtual public base {
public:
  int j;
};

// derived2 eredita base come virtual.
class derived2 : virtual public base {
public:
  int k;
};

/* derived3 eredita sia derived1 che derived2.
   Questa volta in derived3 vi è una sola copia di base. */
class derived3 : public derived1, public derived2 {
public:
  int sum;
};

int main()
{
  derived3 ob;
```

```
  ob.i = 10;  // nessuna ambiguità
  ob.j = 20;
  ob.k = 30;

  // non ambigua
  ob.sum = ob.i + ob.j + ob.k;

  // non ambigua
  cout << ob.i << " ";

  cout << ob.j << " " << ob.k << " ";
  cout << ob.sum;

  return 0;
}
```

Come si può vedere, la parola riservata virtual precede tutte le specifiche di accesso della classe ereditata. Ora che sia derived1 che derived2 hanno ereditato base come virtual, ogni operazione ereditaria successiva provocherà l'uso di una sola copia della classe base. Pertanto in derived3 vi sarà una sola copia di base e l'istruzione ob.i=10 è perfettamente corretta e non presenta alcuna ambiguità.

Un'ultima annotazione: anche se derived1 e derived2 specificano base come virtual, la classe base continuerà ad essere presente in tutti gli oggetti di entrambi i tipi. Ad esempio, il seguente frammento di codice è perfettamente corretto.

```
// definisce una classe di tipo derived1
derived1 myclass;

myclass.i = 88;
```

L'unica differenza fra una comune classe base e una classe base virtuale è il fatto che quando un oggetto eredita più di una copia della classe base, se vengono utilizzate più classi base virtuali, nell'oggetto sarà presente una sola copia della classe base. In caso contrario l'oggetto conterrà più copie della classe base.

Capitolo 17

# Funzioni virtuali e polimorfismo

17.1 **Le funzioni virtuali**
17.2 **L'attributo virtual viene ereditato**
17.3 **Le funzioni virtuali sono gerarchiche**
17.4 **Le funzioni virtuali pure**
17.5 **Uso delle funzioni virtuali**
17.6 **Il binding anticipato e il binding ritardato**

Il polimorfismo è supportato dal C++ sia al momento della compilazione (compile-time) che al momento dell'esecuzione del programma (run-time). Come si è detto in precedenza, il polimorfismo in compilazione trova la sua espressione nell'uso di funzioni e di operatori modificati tramite overloading. Il polimorfismo run-time viene ottenuto utilizzando l'ereditarietà e le funzioni virtuali e verrà trattato nel presente capitolo.

## 17.1 Le funzioni virtuali

Una *funzione virtuale* è una funzione membro dichiarata come virtual in una classe base e ridefinita da una classe derivata. Per creare una funzione virtuale, la sua dichiarazione deve essere preceduta dalla parola chiave virtual. Quando si eredita una classe contenente una funzione virtuale, la classe derivata ridefinisce la funzione virtuale secondo le sue esigenze. In pratica le funzioni virtuali implementano la filosofia "un'interfaccia per più metodi" che sta alla base del polimorfismo. La funzione virtuale contenuta nella classe base definisce la forma dell'interfaccia della funzione. Ogni ridefinizione della funzione virtuale da parte di una classe derivata implementa le operazioni da eseguire con riferimento alla classe derivata stessa. In pratica la funzione virtuale ridefinita implementa un metodo specifico.

Quando si accede "normalmente" a una funzione virtuale, questa si comporta come qualsiasi altro tipo di funzione membro. Ciò che rende le funzioni virtuali così importanti e che le rende capaci di supportare il polimorfismo run-time è il modo in cui si comportano quando si accede ad esse tramite un puntatore. Come si è detto nel Capitolo 13, un puntatore alla classe base può essere utilizzato per

puntare a qualsiasi classe derivata dalla base. Quando un puntatore base punta a un oggetto derivato che contiene una funzione virtuale, il C++ determina quale versione di tale funzione richiamare sulla base del *tipo di oggetto puntato* dal puntatore. Questa determinazione viene eseguita run-time. Pertanto, quando cambia l'oggetto puntato dal puntatore cambierà anche la versione della funzione virtuale che verrà eseguita. Lo stesso effetto si applica ai riferimenti alla classe base.

Per iniziare, ecco un breve esempio:

```
#include <iostream>
using namespace std;

class base {
public:
  virtual void vfunc() {
    cout << "Questa è la funzione vfunc() della classe base.\n";
  }
};

class derived1 : public base {
public:
  void vfunc() {
    cout << "Questa è la funzione vfunc() della classe derived1.\n";
  }
};

class derived2 : public base {
public:
  void vfunc() {
    cout << "Questa è la funzione vfunc() della classe derived2.\n";
  }
};

int main()
{
  base *p, b;
  derived1 d1;
  derived2 d2;

  // punta a base
  p = &b;
  p->vfunc(); // accede alla vfunc() di base

  // punta a derived1
  p = &d1;
  p->vfunc(); // accede alla vfunc() di derived1
```



```
  // punta a derived2
  p = &d2;
  p->vfunc(); // accede alla vfunc() di derived2

  return 0;
}
```

Questo programma visualizza:

```
Questa è la funzione vfunc() della classe base.
Questa è la funzione vfunc() della classe derived1.
Questa è la funzione vfunc() della classe derived2.
```

In questo programma, all'interno di base viene dichiarata la funzione virtuale vfunc(). Si noti che la parola riservata virtual precede la parte rimanente della dichiarazione della funzione. Quando vfunc() viene ridefinita da derived1 e derived2, la parola riservata virtual non è necessaria (anche se non è un errore includerla quando si deve ridefinire una funzione virtuale all'interno di una classe derivata).

In questo programma, base è ereditata da derived1 e derived2. All'interno della definizione di queste due classi, vfunc() viene ridefinita rispetto a tali classi. In main() vengono dichiarate quattro variabili:

| NOME | TIPO |
|---|---|
| p | Puntatore alla classe base |
| b | Oggetto di base |
| d1 | Oggetto di derived1 |
| d2 | Oggetto di derived2 |

Quindi, a p viene assegnato l'indirizzo di b e tramite p viene richiamata vfunc(). Poiché p punta a un oggetto di tipo base, verrà eseguita tale versione di vfunc(). Quindi, a p viene assegnato l'indirizzo di d1 e viene nuovamente richiamata vfunc() utilizzando p. Questa volta p punta a un oggetto di tipo derived1. Quindi verrà eseguita la funzione derived1::vfunc(). Infine, a p viene assegnato l'indirizzo di d2 e p->vfunc() provoca l'esecuzione di vfunc() ridefinita all'interno di derived2. In sostanza, è il genere dell'oggetto a cui punta p a determinare la versione di vfunc() che verrà eseguita. Inoltre, questa verifica viene eseguita al momento dell'esecuzione e questo processo sta alla base del polimorfismo run-time.

Anche se è possibile richiamare una funzione virtuale in modo "normale" utilizzando il nome di un oggetto e l'operatore punto, è solo quando l'accesso

avviene attraverso un puntatore (o un indirizzo) alla classe base che è possibile ottenere il polimorfismo run-time. Ad esempio, prendendo in considerazione l'esempio precedente, l'istruzione seguente è sintatticamente corretta:

```
d2.vfunc(); // richiama la funzione vfunc() di derived2
```

Anche se questo modo di richiamare una funzione virtuale non è errato, semplicemente non sfrutta i vantaggi della natura virtuale di vfunc().

A prima vista, la ridefinizione di una funzione virtuale da parte di una classe derivata sembra simile alla tecnica di overloading delle funzioni. In realtà si tratta di meccanismi completamente diversi e non si è applicato il termine *overloading* alla ridefinizione delle funzioni virtuali poiché esistono molte differenze. La più importante è il fatto che il prototipo di una funzione virtuale ridefinita deve corrispondere esattamente al prototipo specificato nella classe base. Nel caso delle comuni funzioni modificate tramite overloading, il tipo restituito e il numero e il tipo dei parametri può (anzi deve) variare. È grazie a queste differenze che il C++ può selezionare la versione corretta della funzione. Quando invece viene ridefinita una funzione virtuale, non deve invece variare l'aspetto dei prototipi. Se si cambia il prototipo nel tentativo di ridefinire una funzione virtuale, il compilatore C++ considererà l'operazione come un overloading della funzione e si perderà pertanto la sua natura virtuale. Un'altra importante restrizione è il fatto che le funzioni virtuali devono essere membri non static delle classi di cui fanno parte. In particolare non possono essere friend. Infine, le funzioni costruttore, al contrario di quelle distruttore, non possono essere virtuali.

## La chiamata di una funzione virtuale tramite l'indirizzo della classe base

Nell'esempio precedente, veniva richiamata una funzione virtuale tramite un puntatore alla classe base ma la natura polimorfica di una funzione virtuale è disponibile anche quando essa viene richiamata attraverso l'indirizzo della classe base. Come si è detto nel Capitolo 13, un indirizzo è un puntatore implicito. Pertanto si può utilizzare l'indirizzo della classe base per far riferimento a un oggetto della classe base o a un oggetto derivato da tale classe. Quando si richiama una funzione virtuale tramite l'indirizzo della classe base, la funzione che verrà eseguita dipende dall'oggetto cui si fa riferimento al momento della chiamata.

La situazione più comune in cui una funzione virtuale viene richiamata tramite l'indirizzo della classe base si verifica quando l'indirizzo è il parametro di una funzione. Ad esempio si consideri la seguente variante del programma precedente.

```
/* Qui si usa l'indirizzo della classe base
   per accedere a una funzione virtuale. */
```



```
#include <iostream>
using namespace std;

class base {
public:
  virtual void vfunc() {
    cout << "Questa è la funzione vfunc() di base.\n";
  }
};

class derived1 : public base {
public:
  void vfunc() {
    cout << " Questa è la funzione vfunc() di derived1.\n";
  }
};

class derived2 : public base {
public:
  void vfunc() {
    cout << " Questa è la funzione vfunc() di derived2.\n";
  }
};

// Usa un parametro indirizzo della classe base.
void f(base &r) {
  r.vfunc();
}

int main()
{
  base b;
  derived1 d1;
  derived2 d2;

  f(b); // passa a f() un oggetto base
  f(d1); // passa a f() un oggetto derived1
  f(d2); // passa a f() un oggetto derived2

  return 0;
}
```

Questo programma produce lo stesso output della versione precedente. In questo esempio, la funzione f() definisce un parametro indirizzo di tipo base. All'interno di main(), la funzione viene richiamata utilizzando oggetti di tipo base,

derived1 e derived2. In f(), la versione di vfunc() che verrà richiamata dipende dal tipo di oggetto cui si fa riferimento nella chiamata a funzione.

Per semplicità, i rimanenti esempi di questo capitolo richiameranno le funzioni virtuali tramite puntatori alla classe base ma nel caso di indirizzi l'effetto è identico.

## 17.2 L'attributo virtual viene ereditato

Quando viene ereditata una funzione virtuale, viene ereditata anche la sua natura virtuale. Questo significa che quando una classe derivata che abbia ereditato una funzione virtuale viene, a sua volta, utilizzata come classe base per un'altra classe derivata, la funzione rimarrà virtuale. Ad esempio, si consideri questa variante del programma precedente:

```
#include <iostream>
using namespace std;

class base {
public:
  virtual void vfunc() {
    cout << "Questa è la funzione vfunc() della classe base.\n";
  }
};

class derived1 : public base {
public:
  void vfunc() {
    cout << "Questa è la funzione vfunc() della classe derived1.\n";
  }
};

/* derived2 eredita la funzione virtuale vfunc()
   da derived1. */
class derived2 : public derived1 {
public:
  // vfunc() è ancora virtuale
  void vfunc() {
    cout << "Questa è la funzione vfunc() della classe derived2.\n";
  }
};

int main()
{
```



```
  base *p, b;
  derived1 d1;
  derived2 d2;

  // punta a base
  p = &b;
  p->vfunc(); // accede alla vfunc() di base

  // punta a derived1
  p = &d1;
  p->vfunc(); // accede alla vfunc() di derived1

  // punta a derived2
  p = &d2;
  p->vfunc(); // accede alla vfunc() di derived2

  return 0;
}
```

Come ci si può attendere, il programma produce il seguente output:

```
Questa è la funzione vfunc() della classe base.
Questa è la funzione vfunc() della classe derived1.
Questa è la funzione vfunc() della classe derived2.
```

In questo caso, derived2 eredita da derived1 invece che da base ma la funzione vfunc() rimane virtuale.

## 17.3 Le funzioni virtuali sono gerarchiche

Come si è detto, quando una funzione viene dichiarata come virtual da una classe base, può essere ridefinita da una classe derivata. Questa non è però una necessità.

Se una classe derivata non ridefinisce una funzione virtuale, quando un oggetto della classe derivata tenterà di accedere a tale funzione, verrà utilizzata la funzione definita nella classe base. Ad esempio, si consideri il seguente programma in cui derived2 non modifica vfunc():

```
#include <iostream>
using namespace std;

class base {
public:
```

```
  virtual void vfunc() {
    cout << "Questa è la funzione vfunc() della classe base.\n";
  }
};

class derived1 : public base {
public:
  void vfunc() {
    cout << "Questa è la funzione vfunc() della classe derived1.\n";
  }
};

class derived2 : public base {
public:
// vfunc() non viene sostituita da quella di derived2 e viene usata quella
   di base
};

int main()
{
  base *p, b;
  derived1 d1;
  derived2 d2;

  // punta a base
  p = &b;
  p->vfunc(); // accede alla vfunc() di base

  // punta a derived1
  p = &d1;
  p->vfunc(); // accede alla vfunc() di derived1

  // punta a derived2
  p = &d2;
  p->vfunc(); // usa la vfunc() di base

  return 0;
}
```

Il programma produce il seguente output:

```
Questa è la funzione vfunc() della classe base.
Questa è la funzione vfunc() della classe derived1.
Questa è la funzione vfunc() della classe base.
```



Poiché derived2 non ridefinisce vfunc(), quando si richiama vfunc() con oggetti di tipo derived2 verrà utilizzata la funzione definita in base.

Il programma precedente illustra un caso speciale di una regola generale. Poiché in C++ l'ereditarietà è gerarchica, ha senso che anche le funzioni virtuali siano gerarchiche. Questo significa che quando una classe derivata non ridefinisce una funzione virtuale, verrà utilizzata la prima ridefinizione presente in ordine inverso di derivazione. Ad esempio, nel seguente programma, derived2 è derivata da derived1 che a sua volta è derivata da base. Ma derived2 non ridefinisce vfunc(). Questo significa che rispetto a derived2, la versione più vicina di vfunc() è quella presente in derived1. Pertanto, quando un oggetto di derived2 cerca di richiamare vfunc() verrà utilizzata la versione della funzione derived1::vfunc().

```
#include <iostream>
using namespace std;

class base {
public:
  virtual void vfunc() {
    cout << "Questa è la funzione vfunc() della classe base.\n";
  }
};

class derived1 : public base {
public:
  void vfunc() {
    cout << "Questa è la funzione vfunc() della classe derived1.\n";
  }
};

class derived2 : public derived1 {
public:
/* vfunc() non viene ridefinita da derived2.
   In questo caso, poichè derived2 deriva da
   derived1, viene usata la vfunc() di derived1.
*/
};

int main()
{
  base *p, b;
  derived1 d1;
  derived2 d2;

  // punta a base
  p = &b;
```

```
  p->vfunc(); // accede alla vfunc() di base

  // punta a derived1
  p = &d1;
  p->vfunc(); // accede alla vfunc() di derived1

  // punta a derived2
  p = &d2;
  p->vfunc(); // usa la vfunc() di derived1

  return 0;
}
```

Il programma produce il seguente output:

```
Questa è la funzione vfunc() della classe base.
Questa è la funzione vfunc() della classe derived1.
Questa è la funzione vfunc() della classe derived1.
```

## 17.4 Le funzioni virtuali pure

Come si è visto negli esempi della sezione precedente, quando una funzione virtuale non viene ridefinita da una classe derivata, viene impiegata la versione definita nella classe base. In molti casi però non vi è una definizione appropriata di una funzione virtuale all'interno della classe base. Ad esempio, una classe base potrebbe non essere in grado di definire un oggetto in modo sufficiente per consentire la creazione di una funzione virtuale nella classe base. Inoltre, in alcuni casi è necessario assicurarsi che tutte le classi derivate ridefiniscano una funzione virtuale. Per gestire questi due casi, il C++ prevede l'uso di funzioni virtuali pure.

Una *funzione virtuale pura* è una funzione virtuale che non viene definita all'interno della classe base. Per dichiarare una funzione virtuale pura, si utilizza questa forma generale:

virtual *tipo nome-funzione*(*elenco-parametri*) = 0;

Quando una funzione virtuale viene resa pura, ogni classe derivata deve fornire una propria definizione. Se la classe derivata non ridefinisce la funzione virtuale pura, questo provocherà un errore di compilazione.

Il seguente programma contiene un semplice esempio di funzione virtuale pura. Il tipo base number contiene un intero chiamato val, la funzione setvar() e la funzione virtuale pura show(). Le classi derivate hextype, dectype e octtype ereditano number e ridefiniscono show() in modo che visualizzi il valore di val nelle varie basi numeriche (esadecimale, decimale e ottale).



```
#include <iostream>
using namespace std;

class number {
protected:
  int val;
public:
  void setval(int i) {val = i;}

  // show() è una funzione virtuale pura
  virtual void show() = 0;
};

class hextype : public number {
public:
  void show() {
    cout << hex << val << "\n";
  }
};

class dectype : public number {
public:
  void show() {
    cout << val << "\n";
  }
};

class octtype : public number {
public:
  void show() {
    cout << oct << val << "\n";
  }
};

int main()
{
  dectype d;
  hextype h;
  octtype o;

  d.setval(20);
  d.show();  // visualizza 20 - decimale
```

```
  h.setval(20);
  h.show();  // visualizza 14 - esadecimale

  o.setval(20);
  o.show();  // visualizza 24 - ottale

  return 0;
}
```

Anche se questo esempio è piuttosto semplice, illustra il modo in cui una classe base potrebbe non essere in grado di definire una funzione virtuale che abbia un senso. In questo caso, number fornisce semplicemente l'interfaccia comune utilizzata dai tipi derivati. Non vi è alcun motivo di definire show() all'interno di number poiché la base del numero è indefinita. Naturalmente, si può sempre creare una definizione fittizia di una funzione virtuale. Tuttavia, l'utilizzo della versione pura di show() assicura che tutte le classi derivate eseguano la ridefinizione della funzione in base alle proprie esigenze.

Si deve ricordare che quando una funzione virtuale viene dichiarata pura, tutte le classi derivate sono obbligate a ridefinirla. Se una classe derivata non ridefinisce la funzione, verrà prodotto un errore di compilazione.

### Le classi astratte

Una classe che contenga almeno una funzione virtuale pura è chiamata *classe astratta*. Poiché una classe astratta contiene una o più funzioni per le quali non è presente alcuna definizione (ovvero funzioni virtuali pure), l'uso di una classe astratta non consentirà la creazione di alcun oggetto. Al contrario, una classe astratta costituisce un tipo incompleto utilizzato come base per le classi derivate.

Anche se non è possibile creare oggetti di una classe derivata, è possibile creare puntatori e richiederne l'indirizzo. Questo consente alle classi astratte di supportare il polimorfismo run-time, selezionando la funzione virtuale corretta sulla base dei puntatori e degli indirizzi della classe base.

## 17.5 Uso delle funzioni virtuali

Uno degli aspetti fondamentali della programmazione a oggetti è il principio "un'interfaccia più metodi". Questo significa che è possibile definire una classe generale di azioni in cui l'interfaccia rimane costante e ogni derivazione definisce operazioni specifiche. In termini pratici, una classe base può essere utilizzata per definire la natura dell'interfaccia di una classe generale. Ogni classe derivata



implementerà le specifiche operazioni in relazione al tipo di dati impiegato dal tipo derivato.

Uno dei modi più potenti e flessibili per implementare l'approccio "un'interfaccia, più metodi" prevede l'uso di funzioni virtuali, di classi astratte e del polimorfismo run-time. Utilizzando queste funzionalità è possibile creare una gerarchia di classi che passi dal caso generale a quello specifico (dalla classe base alle classi derivate). Secondo questa filosofia, si definiscono tutte le funzionalità e i metodi di interfacciamento comuni all'interno di una classe base. Nei casi in cui determinate azioni possono essere implementate solo dalla classe derivata si utilizza una funzione virtuale. In pratica, nella classe base si crea e definisce tutto ciò che fa riferimento al caso generale. I dettagli vengono completati dalle classi derivate.

Quello che segue è un semplice esempio che illustra il valore della filosofia "un'interfaccia, più metodi". Viene creata una gerarchia di classi che eseguono conversioni da un sistema di unità a un altro sistema (ad esempio da litri a galloni). La classe base convert dichiara due variabili, val1 e val2, le quali devono contenere rispettivamente i valori iniziale e convertito. Inoltre, definisce le funzioni getinit() e getconv() che restituiscono il valore iniziale e il valore convertito. Questi elementi di convert sono fissi e applicabili a tutte le classi derivate che ereditano convert. Ma la funzione che eseguirà effettivamente la conversione, compute() è una funzione virtuale pura che deve essere definita dalle classi derivate da convert. La specifica natura di compute() sarà determinata dal tipo di conversione eseguita.

```
// Esempio pratico di funzione virtuale.

#include <iostream>
using namespace std;

class convert {
protected:
  double val1;  // valore iniziale
  double val2;  // valore convertito
public:
  convert(double i) {
    val1 = i;
  }
  double getconv() {return val2;}
  double getinit() {return val1;}

  virtual void compute() = 0;
};
```

```
// Litri in galloni.
class l_to_g : public convert {
public:
  l_to_g(double i) : convert(i) { }
  void compute() {
    val2 = val1 / 3.7854;
  }
};

// Fahrenheit in Celsius
class f_to_c : public convert {
public:
  f_to_c(double i) : convert(i) { }
  void compute() {
    val2 = (val1-32) / 1.8;
  }
};

int main()
{
  convert *p;  // puntatore alla classe base

  l_to_g lgob(4);
  f_to_c fcob(70);

  // per la conversione impiega una funzione virtuale
  p = &lgob;
  cout << p->getinit() << " litri = ";
  p->compute();
  cout << p->getconv() << " galloni\n";  // l_to_g

  p = &fcob;
  cout << p->getinit() << " in gradi Fahrenheit è uguale a ";
  p->compute();
  cout << p->getconv() << " in gradi Celsius\n";  // f_to_c

  return 0;
}
```

Questo programma crea due classi derivate a partire da convert, chiamate l_to_g e f_to_c. Queste classi eseguono le conversioni da litri in galloni e da Fahrenheit in Celsius. Ogni classe derivata ridefinisce compute nel modo richiesto dalla conversione. Tuttavia, anche se la conversione eseguita (ovvero il metodo) è diverso da l_to_g e f_to_c, l'interfaccia rimane costante.



Uno dei pregi delle classi derivate e delle funzioni virtuali è il fatto che semplificano notevolmente la gestione di un nuovo caso. Ad esempio, sempre tenendo in considerazione il programma precedente, si potrebbe aggiungere una conversione da piedi in metri includendo questa classe:

```
// da piedi in metri
class f_to_m : public convert {
public:
  f_to_m(double i) : convert(i) { }
  void compute() {
    val2 = val1 / 3.28;
  }
};
```

Un utilizzo importante delle classi astratte e delle funzioni virtuali avviene nelle *librerie di classi*. È infatti possibile creare una libreria di classi generiche ed estendibili utilizzabili da altri programmatori. Un altro programmatore erediterà la vostra classe generale che definisce l'interfaccia e tutti gli elementi comuni a tutte le classi da essa derivate, aggiungendo semplicemente le funzioni specifiche della classe derivata. La creazione di librerie di classi consente di creare e controllare l'interfaccia di una classe generale lasciando agli altri programmatori l'adattamento per situazioni specifiche.

Un'ultima annotazione: la classe base convert è un esempio di classe astratta. La funzione virtuale compute non è definita in convert poiché non può essere fornita una definizione che abbia un qualche senso. Semplicemente la classe convert non contiene sufficienti informazioni per definire compute. Sarà solo quando convert sarà ereditata da una classe derivata che sarà possibile creare un tipo completo.

## 17.6 Il binding anticipato e il binding ritardato

Prima di concludere questo capitolo riguardante le funzioni virtuali e il polimorfismo run-time, vi sono due termini che devono essere definiti poiché molto utilizzati nelle discussioni che riguardano il C++ e la programmazione a oggetti: *binding anticipato* e *binding ritardato*.

Con *binding anticipato* si intendono gli eventi che si verificano al momento della compilazione. In pratica, si intende che tutte le informazioni richieste per richiamare una funzione sono note al momento della compilazione. In altre parole, binding anticipato significa che un oggetto e una chiamata a funzione possono essere associati durante la compilazione. Gli esempi di binding anticipato includono le comuni chiamate a funzioni (incluse le funzioni della libreria standard),

le chiamate alle funzioni modificate tramite overloading e l'uso degli operatori modificati tramite overloading. Il vantaggio principale del binding anticipato è l'efficienza. Poiché tutte le informazioni necessarie per richiamare una funzione vengono determinate al momento della compilazione, questo tipo di chiamate a funzione risulta molto veloce.

Il contrario di binding anticipato è il *binding ritardato*. Nel C++, il binding ritardato si riferisce a chiamate a funzioni che non possono essere determinate se non al momento dell'esecuzione del programma. Per ottenere il binding ritardato si utilizzano funzioni virtuali. Come si sa, quando l'accesso avviene tramite un puntatore (o un indirizzo) alla classe base, la funzione virtuale effettivamente richiamata viene determinata dal tipo di oggetto puntato dal puntatore. Poiché nella maggior parte dei casi questo non è noto al momento della compilazione, l'oggetto e la funzione non vengono collegati se non al momento dell'esecuzione. Il vantaggio principale del binding ritardato è la flessibilità. A differenza del binding anticipato, il binding ritardato consente di creare programmi che siano in grado di rispondere a eventi che si verificano durante l'esecuzione del programma senza dover creare codice di gestione dei vari casi. Questo significa anche che, poiché una chiamata a funzione non può essere risolta se non al momento dell'esecuzione, il binding ritardato può in una certa misura rallentare i tempi di esecuzione.

Capitolo_18

# I template

18.1 **Funzioni generiche**
18.2 **Uso delle funzioni generiche**
18.3 **Classi generiche**
18.4 **Le parole riservate typename ed export**
18.5 **La potenza dei template**

I *template* (modelli) sono una delle funzionalità più sofisticate e potenti del C++. Anche se i template non facevano parte delle specifiche originali del C++, sono stati aggiunti molto tempo fa e attualmente sono supportati da tutti i compilatori C++ più recenti. Con un template è possibile creare *funzioni* e *classi generiche*. In una funzione o classe generica, il tipo dei dati su cui la funzione o la classe operano viene specificato come parametro. Pertanto, si potrà utilizzare una funzione o una classe con vari tipi di dati senza dover ricodificare esplicitamente una diversa versione specifica per ogni tipo di dati.

## 18.1 Funzioni generiche

Una funzione generica definisce una serie di operazioni generali applicabili a vari tipi di dati. Il tipo dei dati su cui si troverà ad operare viene passato tramite un parametro. Grazie a una funzione generica, la stessa procedura generale può essere applicata a un'ampia gamma di dati. Molti algoritmi sono infatti logicamente identici indipendentemente dal tipo di dati su cui operano. Ad esempio, l'algoritmo di ordinamento Quicksort è uguale sia che operi su un array di interi, sia che debba ordinare un array di float. L'unica cosa che cambia è il tipo dei dati da ordinare. Creando una funzione generica è possibile definire, indipendentemente dai dati, la natura dell'algoritmo. In questo modo, il compilatore genererà automaticamente il codice corretto per il tipo di dati effettivamente utilizzato per richiamare la funzione. In pratica, quando si crea una funzione generica, si crea una funzione che è in grado di eseguire automaticamente l'overloading di sé stessa.

Per creare una funzione generica si utilizza la parola chiave template. Tale parola chiave crea un modello che descrive le operazioni svolte da una funzione, lasciando al compilatore il completamento dei dettagli. La forma generale della definizione di una funzione template è:

```
template <class tipo> tipo-restituito nome-funzione(elenco parametri)
{
  // corpo della funzione
}
```

Qui *tipo* indica il nome del tipo che verrà utilizzato dalla funzione. Questo nome può essere utilizzato anche nella definizione della funzione. Tuttavia si tratta solo di un segnaposto che il compilatore sostituirà automaticamente con il vero tipo di dati nel momento in cui si troverà a creare una versione specifica della funzione. Anche se normalmente per specificare un tipo di dati generico nella dichiarazione di un template si usa la parola riservata class, si può utilizzare anche la parola riservata typename.

Il breve esempio seguente crea una funzione generica che scambia il valore delle due variabili con le quali viene richiamata. Poiché il processo generale di scambio di due valori è indipendente dal tipo delle variabili, questa funzione è un candidato ideale per la trasformazione in funzione generica.

```
// Esempio di funzione template.

#include <iostream>
using namespace std;

// Questa è una funzione template.
template <class X> void swapargs(X &a, X &b)
{
  X temp;

  temp = a;
  a = b;
  b = temp;
}

int main()
{
  int i=10, j=20;
  float x=10.1, y=23.3;
  char a='x', b='z';

  cout << "i e j originali: " << i << ' ' << j << endl;
  cout << "x e y originali: " << x << ' ' << y << endl;
  cout << "a e b originali: " << a << ' ' << b << endl;

  swapargs(i, j); // scambia due interi
  swapargs (x, y); // scambia due float
  swapargs (a, b); // scambia due caratteri

  cout << "i e j scambiati: " << i << ' ' << j << endl;
  cout << "x e y scambiati: " << x << ' ' << y << endl;
  cout << "a e b scambiati: " << a << ' ' << b << endl;

  return 0;
}
```



Osservando attentamente il programma, si può notare che la riga:

```
template <class X> void swapargs(X &a, X &b)
```

dice al compilatore due cose: che si deve creare una funzione template e che sta per iniziare una definizione generica. Qui, X è un tipo generico utilizzato come segnaposto. Dopo la porzione template viene dichiarata la funzione swapargs() utilizzando X per definire il tipo dei valori che devono essere scambiati. In main(), la funzione swapargs() viene richiamata con tre diversi tipi di dati: int, double e char. Poiché swapargs() è una funzione generica, il compilatore crea automaticamente tre versioni di swapargs(): una di esse scambia valori interi, una valori in virgola mobile e la terza caratteri.

Ecco altri termini che vengono normalmente utilizzati quando si parla di template. Innanzitutto, una funzione generica (ovvero la definizione di una funzione preceduta dall'istruzione template) viene anche chiamata *funzione template*. In questo volume verranno utilizzati indifferentemente entrambi i termini. Quando il compilatore crea una versione specifica di tale funzione si dice che ha creato una specializzazione o una *funzione generata*. L'atto di generazione di una funzione viene chiamato *istanziazione*. In altre parole, una funzione generata è una specifica istanza di una funzione template.

Poiché il C++ non riconosce il codice di fine riga come chiusura dell'istruzione, la clausola *template* della definizione di una funzione generica non deve necessariamente trovarsi sulla stessa riga del nome della funzione. Ad esempio ecco un altro modo molto comune per scrivere la funzione swapargs().

```
template <class X>
void swapargs(X &a, X &b)
{
  X temp;

  temp = a;
  a = b;
  b = temp;
}
```

Utilizzando questa forma è importante comprendere che non vi possono essere altre istruzioni fra l'istruzione template e l'inizio della definizione della funzione generica. Ad esempio, questo frammento di codice non verrà compilato.

```
// questa forma non verrà compilata.
template <class X>
int i; // questo è un errore
void swapargs(X &a, X &b)
{
  X temp;

  temp = a;
  a = b;
  b = temp;
}
```

Come si può leggere nei commenti, la parte template deve precedere immediatamente la definizione della funzione.

## Una funzione con due tipi generici

Con l'istruzione template è possibile definire anche più di un tipo di dati generico, utilizzando un elenco separato da virgole. Ad esempio, il programma seguente crea una funzione template che utilizza due tipi generici.

```
#include <iostream>
using namespace std;

template <class type1, class type2>
void myfunc(type1 x, type2 y)
{
  cout << x << ' ' << y << endl;
}

int main()
{
  myfunc(10, "Programmare in C++");

  myfunc(0.23, 19L);

  return 0;
}
```



In questo esempio, i tipi type2 e type2 vengono sostituiti dal compilatore rispettivamente con i tipi int e char* e double e long nel momento in cui il compilatore genera le specifiche istanze di myfunc() all'interno di main().

**NOTA** *Quando si crea una funzione template, in realtà si chiede al compilatore di generare tutte le versioni di tale funzione necessarie per gestire tutte le situazioni nelle quali il programma deve impiegare tale funzione.*

## Overloading esplicito di una funzione generica

Anche se una funzione generica esegue l'overloading di sé stessa sulla base delle situazioni in cui si trova ad operare, rimane comunque possibile eseguire un *overloading esplicito* della funzione. Questa tecnica è chiamata specializzazione esplicita. Se si esegue l'overloading di una funzione generica, la versione modificata tramite overloading "nasconderà" la funzione generica relativa a questa specifica versione. Ad esempio, si consideri questa nuova versione dell'esempio che eseguiva lo scambio degli argomenti.

```
// Modifica di una funzione template.

#include <iostream>
using namespace std;

template <class X> void swapargs(X &a, X &b)
{
  X temp;

  temp = a;
  a = b;
  b = temp;
  cout << "Nel template di swapargs(int &, int &).\n";
}
// Questa versione nasconde la versione generica di swapargs().
void swapargs(int &a, int &b)
{
  int temp;

  temp = a;
  a = b;
  b = temp;
  cout << "Nella specializzazione int di swapargs(int &, int &).\n";
}
```

```
int main()
{
  int i=10, j=20;
  float x=10.1, y=23.3;
  char a='x', b='z';

  cout << "i e j originali: " << i << ' ' << j << endl;
  cout << "x e y originali: " << x << ' ' << y << endl;
  cout << "a e b originali: " << a << ' ' << b << endl;

  swapargs(i, j); // richiama la swapargs() modificata tramite overloading
  swapargs(x, y); // richiama la swapargs() generica
  swapargs(a, b); // richiama la swapargs() generica

  cout << "i e j scambiati: " << i << ' ' << j << endl;
  cout << "x e y scambiati: " << x << ' ' << y << endl;
  cout << "a e b scambiati: " << a << ' ' << b << endl;

  return 0;
}
```

Come si può dedurre dai commenti, quando viene richiamata swapargs(i, j), questa richiama la versione di swapargs() che modificata tramite un overloading esplicito. Pertanto, il compilatore non genererà questa versione della funzione generica swapargs() poiché la funzione generica viene nascosta dall'overloading esplicito.

L'esecuzione di un overloading manuale di un template, come si vede in questo esempio, consente di adattare con precisione una versione di una funzione generica in modo da gestire una particolare situazione. Tuttavia, in generale, se è necessario impiegare versioni diverse di una funzione per i diversi tipi di dati, si dovranno utilizzare funzioni modificate tramite overloading piuttosto che funzioni generiche.

Recentemente è stata introdotta una nuova forma sintattica che denota la specializzazione esplicita di una funzione. Questo nuovo metodo utilizza la parola riservata template. Ad esempio, utilizzando questa nuova sintassi di specializzazione, la funzione swapargs() modificata tramite overloading presentata nel precedente programma assumerà il seguente aspetto:

```
// Uso della nuova forma sintattica.
template<> void swapargs<int>(int &a, int &b)
{
  int temp;
  temp = a;
  a = b;
  b = temp;
  cout << "In swapargs con specializzazione int.\n";
}
```



Come si può vedere, la nuova sintassi indica la specializzazione tramite il costruttore template<>. Il tipo di dati per il quale viene creata la specializzazione deve essere specificato fra le parentesi angolari dopo il nome della funzione. Questa stessa sintassi è utilizzata per specializzare ogni tipo di funzione generica. Anche se attualmente non esistono vantaggi a utilizzare una versione della sintassi di specializzazione rispetto all'altra, a lungo termine è probabilmente meglio utilizzare la nuova sintassi.

La specializzazione esplicita di un template consente di personalizzare una versione di una funzione generica per rispondere a situazioni specifiche, ad esempio per sfruttare alcuni stratagemmi prestazionali che si applicano a un solo tipo di dati. Tuttavia, in generale, se si devono creare versioni differenti di una funzione per tipi di dati differenti, si dovrebbero usare le funzioni modificate tramite overloading piuttosto che i template.

## Overloading di un template di funzioni

Oltre a creare versioni esplicite e modificate tramite overloading di una funzione generica, si può eseguire l'overloading anche dello stesso template. A tale scopo basta creare una nuova versione del template che differisca dalle altre versioni per il tipo dei parametri. Ad esempio:

```
// Overloading della dichiarazione di un template di funzioni.
#include <iostream>
using namespace std;

// Prima versione del template di f().
template <class X> void f(X a)
{
   cout << "In f(X a)\n";
}

// Seconda versione del template di f().
template <class X, class Y> void f(X a, Y b)
{
   cout << "In f(X a, Y b)\n";
}
```

```
int main()
{
   f(10);         // richiama f(X)
   f(10, 20);     // richiama f(X, Y)

   return 0;
}
```

Qui il template di f() viene modificato tramite overloading per accettare uno o due parametri.

## Uso di parametri standard nelle funzioni template

In una funzione template è possibile usare insieme parametri standard e parametri di tipo generico. Questi parametri non generici funzionano esattamente come ogni altra funzione. Ad esempio:

```
// Uso di parametri standard in una funzione template.
#include <iostream>
using namespace std;

const int TABWIDTH = 8;

// Visualizza i dati alla posizione di rientro specificata.
template<class X> void tabOut(X data, int tab)
{
   for(; tab; tab--)
       for(int i=0; i<TABWIDTH; i++) cout << ' ';

       cout << data << "\n";
}

int main()
{
   tabOut("Questa è una prova", 0);
   tabOut(100, 1);
   tabOut('X', 2);
   tabOut(10/3, 3);

   return 0;
}
```

Ecco l'output prodotto dal programma.



```
Questa è una prova
        100-
                X
                        3
```

Nel programma, la funzione tabOut() visualizza il suo primo argomento nella posizione specificata dal secondo argomento. Dato che il primo argomento è di tipo generico, tabOut() può essere utilizzata per visualizzare qualsiasi tipo di dati. Il parametro tab è un parametro standard con chiamata per valore. L'utilizzo contemporaneo di parametri generici e non generici non provoca problemi ed è anzi molto comune e utile.

## Restrizioni delle funzioni generiche

Le funzioni generiche sono simili alle funzioni modificate tramite overloading ma hanno maggiori restrizioni. Quando viene eseguito l'overloading di una funzione, nel corpo di ogni funzione possono essere eseguite azioni diverse. Una funzione generica deve invece eseguire la stessa azione generale per tutte le versioni (l'unica differenza è costituita dal tipo). Si considerino le funzioni modificate tramite overloading contenute nel seguente programma. Non è possibile utilizzare una funzione generica al posto delle funzioni modificate tramite overloading poiché vengono eseguite operazioni diverse.

```
#include <iostream>
#include <cmath>
using namespace std;

void myfunc(int i)
{
  cout << "Il valore è: " << i << "\n";
}

void myfunc(double d)
{
  double intpart;
  double fracpart;

  fracpart = modf(d, &intpart);
  cout << "Parte decimale: " << fracpart;
  cout << "\n";
  cout << "Parte intera: " << intpart;
}
```

```
int main()
{
  myfunc(1);
  myfunc(12.2);

  return 0;
}
```

## 18.2 Uso delle funzioni generiche

Le funzioni generiche sono una delle funzionalità più utili del C++. Tali funzioni possono essere applicate a vari tipi di situazioni. Come si è detto in precedenza, quando si utilizza una funzione che definisce un algoritmo generalizzabile, la si può trasformare in una funzione template. In questo modo, la funzione potrà essere utilizzata con qualsiasi tipo di dati senza necessità di ricodifica. Prima di parlare delle classi generiche, questa sezione presenta due esempi di funzioni generiche. Questi esempi illustrano quanto sia facile sfruttare i vantaggi di questa importante funzionalità.

### Ordinamento generico

L'ordinamento è il genere di operazioni che ha suggerito l'introduzione di funzioni generiche. In generale, un algoritmo di ordinamento non cambia quando varia il tipo dei dati ordinati. Il programma seguente illustra questo concetto creando un algoritmo di Bubblesort generico. Anche se il Bubblesort è un algoritmo di basse prestazioni, il suo funzionamento è semplice e non presenta aspetti oscuri e questo lo rende un esempio di facile comprensione. La funzione bubble() è in grado di ordinare qualsiasi tipo di array. La funzione deve essere richiamata con un puntatore al primo elemento dell'array e con il numero di elementi contenuti nell'array.

```
// Bubblesort generico.

#include <iostream>
using namespace std;

template <class X> void bubble(
  X *items,  // puntatore all'array da ordinare
  int count) // numero di elementi contenuti nell'array
{
  register int a, b;
  X t;

  for(a=1; a<count; a++)
    for(b=count-1; b>=a; b--)
      if(items[b-1] > items[b]) {
        // scambia gli elementi
        t = items[b-1];
        items[b-1] = items[b];
        items[b] = t;
      }
}

int main()
{
  int iarray[7] = {7, 5, 4, 3, 9, 8, 6};
  double darray[5] = {4.3, 2.5, -0.9, 100.2, 3.0};

  int i;

  cout << "Ecco l'array di interi non ordinato: ";
  for(i=0;  i<7; i++)
    cout << iarray[i] << ' ';
  cout << endl;

  cout << "Ecco l'array di double non ordinato: ";
  for(i=0;  i<5; i++)
    cout << darray[i] << ' ';
  cout << endl;

  bubble(iarray, 7);
  bubble(darray, 5);
  cout << "Ecco l'array di interi ordinato: ";
  for(i=0;  i<7; i++)
    cout << iarray[i] << ' ';
  cout << endl;

  cout << "Ecco l'array di double ordinato: ";
  for(i=0;  i<5; i++)
    cout << darray[i] << ' ';
  cout << endl;

  return 0;
}
```

Questo programma genera il seguente output:

```
Ecco l'array di interi non ordinato: 7 5 4 3 9 8 6
Ecco l'array di double non ordinato: 4.3 2.5 -0.9 100.2 3
Ecco l'array di interi ordinato: 3 4 5 6 7 8 9
Ecco l'array di double ordinato: -0.9 2.5 3 4.3 100.2
```

Come si può vedere, il programma precedente crea due array: uno di interi e uno di double. Quindi procede a ordinarli entrambi; poiché bubble() è una funzione template, subirà un overloading automatico per accettare i due diversi tipi di dati.

## Compattamento di un array

Un'altra funzione che consente di sfruttare i vantaggi dati dall'uso dei template è compact(). Questa funzione compatta gli elementi di un array. Capita spesso di dover eliminare elementi da un array e quindi di dover spostare gli elementi rimanenti in modo da lasciare tutti gli elementi inutilizzati alla fine. Questo tipo di operazione è la stessa per tutti i tipi di array poiché non dipende dal tipo dei dati su cui si opera. La funzione generica compact() mostrata nell'esempio seguente deve essere richiamata con un puntatore al primo elemento dell'array, il numero di elementi contenuti nell'array e gli indici iniziale e finale degli elementi da eliminare. La funzione elimina dall'array tali elementi e quindi compatta l'array. Al termine inserisce il valore zero in tutti gli elementi inutilizzati che si trovano alla fine dell'array e che sono stati liberati dal compattamento.

```
// Una funzione generica per il compattamento di un array.

#include <iostream>
using namespace std;

template <class X> void compact(
  X *items,  // puntatore all'array da compattare
  int count, // numero di elementi contenuti nell'array
  int start, // indice iniziale della regione compattata
  int end)   // indice finale della regione compattata
{
  register int i;

  for(i=end+1; i<count; i++, start++)
    items[start] = items[i];

  /* Per semplicità, la parte rimanente
     dell'array viene azzerata. */
  for( ; start<count; start++) items[start] = (X) 0;
}
```



```
int main()
{
  int nums[7] = {0, 1, 2, 3, 4, 5, 6};
  char str[18] = "Funzioni generiche";

  int i;

  cout << "Ecco l'array di interi non compattato: ";
  for(i=0;  i<7; i++)
    cout << nums[i] << ' ';
  cout << endl;

  cout << "Ecco la stringa non compattata: ";
  for(i=0;  i<18; i++)
    cout << str[i] << ' ';
  cout << endl;

  compact(nums, 7, 2, 4);
  compact(str, 18, 6, 10);

  cout << "Ecco l'array di interi compattato: ";
  for(i=0;  i<7; i++)
    cout << nums[i] << ' ';
  cout << endl;

  cout << "Ecco la stringa compattata: ";
  for(i=0;  i<18; i++)
    cout << str[i] << ' ';
  cout << endl;

  return 0;
}
```

Questo programma compatta due tipi di array. Uno è un array di interi e l'altro è una stringa. La funzione compact() funzionerà con qualsiasi tipo di array. Di seguito viene presentato l'output del programma:

```
Ecco l'array di interi non compattato: 0 1 2 3 4 5 6
Ecco la stringa non compattata: F u n z i o n i   g e n e r i c h e
Ecco l'array di interi compattato: 0 1 5 6 0 0 0
Ecco la stringa compattata: F u n z i o e n e r i c h e
```

Come si può dedurre dagli esempi precedenti, quando si inizia a pensare in termini di template (modelli), è naturale che vengano in mente le situazioni in cui

è possibile utilizzare tale tecnica. Tutti i casi in cui la logica operativa di una funzione non cambia sono candidati ideali per l'utilizzo di una funzione generica.

## 18.3 Classi generiche

Oltre alle funzioni generiche è anche possibile definire una classe generica. In tal caso si crea una classe che definisce tutti gli algoritmi utilizzati da tale classe ma nella quale il tipo di dati da manipolare viene specificato come parametro nel momento in cui vengono creati gli oggetti della classe.

Le classi generiche sono utili nel caso in cui una classe contenga elementi logici generalizzabili. Ad esempio, lo stesso algoritmo che esegue la gestione di una coda di interi funzionerà anche per una coda di caratteri. Lo stesso meccanismo che gestisce una lista concatenata di indirizzi funzionerà anche per gestire una lista concatenata di informazioni relative alle parti di ricambio di un'automobile. Una classe generica può eseguire l'operazione definita, come ad esempio la gestione di una coda o di una lista concatenata per qualsiasi tipo di dati. Il compilatore genererà automaticamente il tipo di oggetti corretti sulla base del tipo specificato al momento della creazione dell'oggetto.

La forma generale della dichiarazione di una classe generica è:

template <class *tipo*> class *nome-classe* {
.
.
.
}

Qui *tipo* è un segnaposto a cui dovrà essere sostituito il nome del tipo al momento dell'istanziazione della classe. Se necessario, si può definire più di un tipo di dati generico utilizzando un elenco separato da virgole.

Dopo aver creato una classe generica, sarà possibile creare una specifica istanza di tale classe utilizzando la forma generale:

*nome-classe<tipo> oggetto;*

Qui *tipo* è il nome del tipo dei dati su cui la classe si troverà ad operare. Le funzioni membro di una classe generica sono anch'esse (automaticamente) generiche. Non è necessario usare template per specificarle esplicitamente come tali.

Nel programma seguente, la classe stack (introdotta nel Capitolo 11) viene rielaborata e trasformata in una classe generica. In questo modo potrà essere riutilizzata per memorizzare oggetti di qualsiasi tipo. In questo esempio, verranno creati uno stack di caratteri e uno stack di numeri in virgola mobile ma è possibile utilizzare dati di qualsiasi tipo.



```
// Creazione di una classe generica per stack.

#include <iostream>
using namespace std;

const int SIZE = 100;

// Creazione della classe generica stack.
template <class StackType> class stack {
  StackType stck[SIZE];
  int tos;  // indice della cima dello stack

public:
  stack() { tos = 0; };
  void push(StackType ob);
  StackType pop();
};

// Push di un oggetto nello stack.
template <class StackType> void stack<StackType>::push(StackType ob)
{
  if(tos==SIZE) {
    cout << "Lo stack è pieno.\n";
    return;
  }
  stck[tos] = ob;
  tos++;
}

// Pop di un oggetto dallo stack.
template <class StackType> StackType stack<StackType>::pop()
{
  if(tos==0) {
    cout << "Stack vuoto.\n";
    return 0;
  }
  tos--;
  return stck[tos];
}

int main()
{
  // dimostra l'uso degli stack di caratteri
```

```
  stack<char> s1, s2;   // crea due stack di caratteri
  int i;

  s1.push("a");
  s2.push("x");
  s1.push("b");
  s2.push("y");
  s1.push("c");
  s2.push("z");

  for(i=0; i<3; i++)  cout << "Pop s1: " << s1.pop() << \n;
  for(i=0; i<3; i++)  cout << "Pop s2: " << s2.pop() << \n;

  // dimostra l'uso degli stack di double
  stack<double> ds1, ds2;   // crea due stack di double

  ds1.push(1.1);
  ds2.push(2.2);
  ds1.push(3.3);
  ds2.push(4.4);
  ds1.push(5.5);
  ds2.push(6.6);

  for(i=0; i<3; i++)  cout << "Pop ds1: " << ds1.pop() << \n;
  for(i=0; i<3; i++)  cout << "Pop ds2: " << ds2.pop() << \n;

  return 0;
}
```

Come si può vedere, la dichiarazione di una classe generica è simile a quella di una funzione generica. Il tipo di dati generico viene utilizzato nella dichiarazione della classe e delle sue funzioni membro. Il tipo di dati da impiegare effettivamente è noto solo nel momento in cui viene dichiarato un oggetto dello stack. Quando viene dichiarata una specifica istanza di stack il compilatore genera automaticamente tutte le funzioni e le variabili necessarie per gestire i dati effettivamente impiegati. In questo esempio, vengono dichiarati due tipi di stack (due per i caratteri e due per i double). Si faccia particolare attenzione alle seguenti dichiarazioni:

```
stack<char> s1, s2;       // crea due stack di caratteri
stack<double> ds1, ds2;   // crea due stack di double
```

Si noti il modo in cui il tipo di dati viene passato fra parentesi angolari. Cambiando il tipo di dati specificato nel momento in cui vengono creati gli oggetti



stack, cambia il tipo di dati che verrà conservato nello stack. Ad esempio, si potrebbe creare un altro stack che conservi i puntatori a caratteri utilizzando la dichiarazione:

```
stack<char *> chrptrQ;
```

Ma è anche possibile creare stack per contenere tipi di dati definiti dal programmatore. Ad esempio, per conservare indirizzi, si può utilizzare la seguente struttura:

```
struct addr {
  char name[40];
  char street[40];
  char city[30];
  char state[3];
  char zip[12];
}
```

Quindi, si potrà utilizzare stack per generare uno stack che conterrà oggetti di tipo addr utilizzando una dichiarazione simile alla seguente:

```
stack<addr> obj;
```

Come si può dedurre dall'esempio sulla classe stack le funzioni e le classi generiche costituiscono strumenti importantissimi per sfruttare al massimo il proprio lavoro di programmazione poiché consentono di definire la forma generale di un oggetto che potrà essere utilizzata con qualsiasi tipo di dati. Si potrà così evitare di creare implementazioni distinte per ogni tipo di dati su cui un algoritmo dovrà operare. Le versioni specifiche della classe verranno create automaticamente dal compilatore.

## Un esempio con due tipi di dati generici

Una classe template può utilizzare anche più di un tipo di dati generico. A tale scopo basterà dichiarare tutti i tipi di dati richiesti dalla classe in un elenco separato da virgole all'interno delle specifiche di template. Il seguente breve esempio crea una classe che utilizza due tipi di dati generici.

```
/* Questo esempio utilizza due tipi di dati generici
   nella definizione di una classe. */

#include <iostream>
```

```
using namespace std;

template <class Type1, class Type2> class myclass
{
  Type1 i;
  Type2 j;
public:
  myclass(Type1 a, Type2 b) { i = a; j = b; }
  void show() { cout << i << ' ' << j << '\n'; }
};

int main()
{
  myclass<int, double> ob1(10, 0.23);
  myclass<char, char *> ob2('X', "Questa è una prova");

  ob1.show(); // int e double
  ob2.show(); // char e char *

  return 0;
}
```

Questo programma produce il seguente output:

```
10 0.23
X Questa è una prova
```

Il programma dichiara due tipi di oggetti: ob1 che utilizza dati interi e double e ob2 che utilizza caratteri e puntatori a caratteri. In entrambi i casi, il compilatore genererà automaticamente i dati e le funzioni necessari per gestire gli oggetti creati.

## Creazione di una classe template: una classe generica per array

Per illustrare i vantaggi pratici delle classi template si parlerà del loro utilizzo pratico. Come si è detto nel Capitolo 15 è possibile eseguire l'overloading anche dell'operatore []. In tal modo è possibile creare proprie implementazioni degli array "sicuri" che contengono la verifica del superamento dei limiti. Come si è detto, in C++ è possibile superare i limiti superiore o inferiore di un array nel momento dell'esecuzione senza che venga generato alcun messaggio di errore. Se però si crea una classe che contiene l'array e consente di accedere a tale array solo attraverso l'operatore [] modificato dall'overloading, sarà possibile intercettare ogni riferimento errato.



Combinando l'overloading degli operatori e le classi template, è possibile creare un tipo di array sicuro e generico che può essere utilizzato per creare array sicuri di qualsiasi tipo di dati. Il seguente programma illustra un esempio di ciò:

```
// Esempio di array generico sicuro.

#include <iostream>
#include <cstdlib>
using namespace std;

const int SIZE = 10;

template <class AType> class atype {
  AType a[SIZE];
public:
  atype() {
    register int i;
    for(i=0; i<SIZE; i++) a[i] = i;
  }
  AType &operator[](int i);
};

// Verifica dei limiti per atype.
template <class AType> AType &atype<AType>::operator[](int i)
{
  if(i<0 || i> SIZE-1) {
    cout << "\nL'indice ";
    cout << i << " è fuori dai limiti dell'array.\n";
    exit(1);
  }
  return a[i];
}

int main()
{
  atype<int> intob; // array di interi
  atype<double> doubleob; // array di double

  int i;

  cout << "Array di interi: ";
  for(i=0; i<SIZE; i++) intob[i] = i;
  for(i=0; i<SIZE; i++) cout << intob[i] << "  ";
  cout << '\n';
```

```
  cout << "Array di double: ";
  for(i=0; i<SIZE; i++) doubleob[i] = (double) i/3;
  for(i=0; i<SIZE; i++) cout << doubleob[i] << "  ";
  cout << '\n';

  intob[12] = 100; // genera un errore run-time

  return 0;
}
```

Questo programma implementa un tipo di array generico sicuro e ne dimostra l'uso creando un array di interi e un array di double (ma si possono creare anche altri tipi di array). Come si può vedere dall'esempio, una parte delle potenzialità delle classi generiche è dovuta al fatto che esse consentono di scrivere il codice una sola volta, di correggerlo e quindi di applicarlo a qualsiasi tipo di dati senza dover continuamente intervenire per ogni diversa applicazione.

## Uso di argomenti non-tipi nelle classi generiche

Nelle specifiche del template per una classe generica, è anche possibile indicare argomenti non-tipi. Questo significa che nelle specifiche di un template si può specificare ciò che si pensa sarà un argomento standard, ad esempio un intero o un puntatore. La forma sintattica per ottenere ciò è essenzialmente la stessa dei normali parametri di funzioni: basta includere il tipo e il nome dell'argomento. Ad esempio, ecco un modo migliore per implementare la classe per array sicuri presentata nella sezione precedente.

```
// Dimostra l'uso di argomenti non-tipi.
#include <iostream>
#include <cstdlib>
using namespace std;

// Qui, int size è un argomento non-tipo.
template <class AType, int size> class atype {
  AType a[size]; // in size viene passata la lunghezza dell'array
public:
  atype() {
    register int i;
    for(i=0; i<size; i++) a[i] = i;
  }
  AType &operator[](int i);
};
```



```
// Verifiche dei limiti per atype.
template <class AType, int size>
AType &atype<AType, size>::operator[](int i)
{
  if(i<0 || i> size-1) {
    cout << "\nIl valore dell'indice di ";
    cout << i << " è fuori dai limiti.\n";
    exit(1);
  }
  return a[i];
}

int main()
{
  atype<int, 10> intob;          // array di int di dimensioni 10
  atype<double, 15> doubleob;    // array di double di dimensioni 15

  int i;

  cout << "Array di interi: ";
  for(i=0; i<10; i++) intob[i] = i;
  for(i=0; i<10; i++) cout << intob[i] << "   ";
  cout << '\n';

  cout << "Array di double: ";
  for(i=0; i<15; i++) doubleob[i] = (double) i/3;
  for(i=0; i<15; i++) cout << doubleob[i] << "   ";
  cout << '\n';

  intob[12] = 100; // genera un errore runtime

  return 0;
}
```

Si osservi attentamente il template di atype. Si noterà che size è dichiarato come int. Questo parametro viene poi utilizzato in atype per dichiarare le dimensioni dell'array a. Anche se size è rappresentato come "variabile" nel codice sorgente, il suo valore è noto al momento della compilazione. Questo consente di utilizzarlo per definire le dimensioni dell'array. size viene utilizzato anche nella verifica dei limiti all'interno della funzione operator[](). In main() si noti come vengono creati gli array di interi e di numeri in virgola mobile. Il secondo parametro specifica le dimensioni di ciascun array. I parametri non-tipi possono essere interi, puntatori o indirizzi. Non è possibile utilizzare altri tipi, ad esempio valori float. Gli argomenti che si passano a un parametro non-tipo devono essere costituiti da una costante intera o un puntatore o indirizzo di una funzione o un oggetto globa-

le. Pertanto i parametri non-tipi dovrebbero essere pensati come costanti in quanto il loro valore non può essere cambiato. Ad esempio, all'interno di operator[](), non è possibile utilizzare la seguente istruzione.

```
size = 10; // Error
```

Poiché i parametri non-tipi sono trattati come costanti, possono essere utilizzati per impostare le dimensioni di un array, il che offre un notevole vantaggio pratico.

Come illustrato dall'esempio dell'array sicuro, l'uso parametri non-tipi estende notevolmente l'utilità delle classi template. Anche se le informazioni contenute nell'argomento non-tipo devono essere note al momento della compilazione, questa restrizione è accettabile in considerazione delle potenzialità offerte dai parametri non-tipi.

## Uso di argomenti standard nelle classi template

Una classe template può avere un argomento standard associato a un tipo generico. Ad esempio:

```
template <class X=int> class myclass { //...
```

Qui il tipo int verrà utilizzato solo nel caso in cui non venga specificato un altro tipo quando viene istanziato un oggetto di tipo myclass.

Gli argomenti non-tipi possono anche prendere valori standard. Il valore standard viene utilizzato quando non viene specificato alcun valore esplicito al momento dell'istanziazione della classe. Gli argomenti standard per parametri non-tipi vengono specificati utilizzando la stessa sintassi degli argomenti standard dei parametri delle funzioni.

Ecco un'altra versione della classe per array sicuri che utilizza argomenti standard sia per il tipo dei dati che per le dimensioni dell'array.

```
// Dimostra l'uso di argomenti standard.
#include <iostream>
#include <cstdlib>
using namespace std;

// Qui AType è normalmente int e size è normalmente 10.
template <class AType=int, int size=10> class atype {
  AType a[size]; // in size vengono passate le dimensioni dell'array
public:
  atype() {
    register int i;
    for(i=0; i<size; i++) a[i] = i;
  }
  AType &operator[](int i);
};

// Verifiche dei limiti per atype.
template <class AType, int size>
AType &atype<AType, size>::operator[](int i)
{
  if(i<0 || i> size-1) {
    cout << "\nIl valore dell'indice di ";
    cout << i << " è fuori dai limiti.\n";
    exit(1);
  }
  return a[i];
}

int main()
{
  atype<int, 100> intarray;   // array di int, size = 100
  atype<double> doublearray;  // array di double, valore standard di size
  atype<> defarray;           // usa lo standard int; size = 10

  int i;

  cout << "Array di int: ";
  for(i=0; i<100; i++) intarray[i] = i;
  for(i=0; i<100; i++) cout << intarray[i] << "   ";
  cout << '\n';

  cout << "Array di double: ";
  for(i=0; i<10; i++) doublearray[i] = (double) i/3;
  for(i=0; i<10; i++) cout << doublearray[i] << "   ";
  cout << '\n';

  cout << "Array standard: ";
  for(i=0; i<10; i++) defarray[i] = i;
  for(i=0; i<10; i++) cout << defarray[i] << "   ";
  cout << '\n';

  return 0;
}
```

Si faccia particolare attenzione alla seguente riga:

```
template <class AType=int, int size=10> class atype {
```

Qui l'impostazione standard di AType è il tipo int e size ha il valore standard 10. Come illustrato dal programma, gli oggetti atype possono essere creati in tre modi:

- specificando esplicitamente il tipo e le dimensioni dell'array;
- specificando esplicitamente il tipo ma utilizzando le dimensioni standard pari a 10;
- lasciando il tipo standard int e utilizzando le dimensioni standard 10.

L'uso degli argomenti standard (specialmente nel caso del tipo) aumenta la versatilità delle classi template. È possibile fornire un valore standard corrispondente al tipo più utilizzato ma consentendo nel contempo all'utente delle classi di specializzarle a piacere.

### Specializzazioni esplicite di una classe

Come nel caso delle funzioni template, è possibile creare una specializzazione esplicita di una classe generica. A tale scopo si deve utilizzare il costruttore template<> che funziona esattamente come per le specializzazioni esplicite delle funzioni. Ad esempio:

```
// Dimostra la specializzazione delle classi.
#include <iostream>
using namespace std;

template <class T> class myclass {
  T x;
public:
  myclass(T a) {
      cout << "In myclass generica\n";
      x = a;
  }
  T getx() { return x; }
};

// Specializzazione per int.
template <> class myclass<int> {
  int x;
public:
  myclass(int a) {
      cout << "Nella specializzazione myclass<int> \n";
      x = a * a;
  }
  int getx() { return x; }
};

int main()
{
  myclass<double> d(10.1);
  cout << "double: " << d.getx() << "\n\n";

  myclass<int> i(5);
  cout << "int: " << i.getx() << "\n";

  return 0;
}
```

Questo programma visualizza il seguente output:

```
In myclass generica
double: 10.1

Nella specializzazione myclass<int>
int: 25
```

In questo programma si deve fare particolare attenzione alla seguente riga:

```
template <> class myclass<int> {
```

Questa riga dice al compilatore che si sta per creare una specializzazione intera esplicita della classe myclass. Questa stessa forma sintattica generale viene utilizzata per ogni tipo di specializzazione di classi.

La specializzazione esplicita di classi estende l'utilità delle classi generiche in quanto consente di gestire con facilità uno o due casi specializzati lasciando che tutti gli altri casi vengano gestiti automaticamente dal compilatore. Naturalmente se si devono creare troppe specializzazioni è probabilmente meglio evitare di utilizzare una classe template.

## 18.4 Le parole riservate typename ed export

Recentemente al linguaggio C++ sono state aggiunte due parole riservate dedicate in modo specifico ai template: typename ed export.

Entrambe giocano ruoli specializzati nella programmazione C++.

La parola riservata typename ha due utilizzi. Innanzitutto, come si è detto in precedenza, può sostituire la parola riservata class nella dichiarazione di un template. Ad esempio, la funzione template swapargs() potrebbe essere specificata nel seguente modo:

```
template <typename X> void swapargs(X &a, X &b)
{
   X temp;

   temp = a;
   a = b;
   b = temp;
}
```

Qui typename specifica il tipo generico X. Non vi è differenza fra usare class e typename in questo contesto.

Il secondo uso di typename consente di informare il compilatore che un nome utilizzato nella dichiarazione di un template fa riferimento a un tipo e non al nome di un oggetto. Ad esempio:

```
typename X::Name someObject;
```

assicura che X::Name venga trattato come il nome di un tipo.

La parola riservata export può precedere una dichiarazione template. Essa consente ad altri file di utilizzare un template specificando solo la sua dichiarazione senza duplicare l'intera definizione.

## 18.5 La potenza dei template

I template aiutano a raggiungere uno degli obiettivi più difficili della programmazione: creare codice riutilizzabile. Grazie all'uso di classi template è possibile creare strutture di applicazioni che possono essere utilizzate più volte e applicate a varie situazioni di programmazione. Ad esempio si consideri la classe stack. La prima versione presentata nel Capitolo 11 poteva essere utilizzata solo per memorizzare valori interi. Anche se l'algoritmo poteva essere utilizzato per ogni tipo di dati, l'indicazione esplicita del tipo dei dati della classe stack limitava gravemente la sua applicazione. Se si trasforma stack in una classe generica, diventa possibile creare uno stack per qualsiasi tipo di dati.

Le funzioni e le classi generiche costituiscono strumenti molto potenti per amplificare i propri sforzi di programmazione. Dopo aver scritto e corretto una classe template, si sarà creato un solido componente software utilizzabile con sicurezza in varie situazioni. Dunque non sarà più necessario creare implementazioni distinte per ciascun tipo di dati cui dovrà essere applicata la classe. Anche se in effetti la sintassi template può inizialmente intimidire, vale la pena di approfondire l'argomento in modo da sfruttare al meglio le sue potenzialità. Le funzioni e le classi template stanno diventando sempre più comuni in programmazione e questa tendenza sembra affermarsi con il tempo. Ad esempio, la libreria STL (Standard Template Library) definita dal C++ è, come si può dedurre dal nome, costruita sui template. Un ultimo commento: anche se i template aggiungono un livello di astrazione al programma, producono comunque codice oggetto ad alte prestazioni.

Capitolo 19

# Gestione delle eccezioni


19.1 **Principi di gestione delle eccezioni**

19.2 **Gestione delle eccezioni per classi derivate**

19.3 **Opzioni della gestione delle eccezioni**

19.4 **Le funzioni terminate() e unexpected()**

19.5 **La funzione uncaught_exception()**

19.6 **Le classi exception e bad_exception**

19.7 **Applicazioni della gestione delle eccezioni**


Questo capitolo si occupa del sottosistema di *gestione delle eccezioni*. Con gestione delle eccezioni si intende la gestione degli errori che si verificano al momento dell'esecuzione, per evitare che provochino il blocco del programma. Utilizzando la gestione delle eccezioni del C++, nel momento in cui si verifica un errore il programma può richiamare automaticamente una routine per la gestione dell'errore. Il principale vantaggio derivante dalla gestione delle eccezioni è il fatto che automatizza una gran parte del codice che precedentemente doveva essere codificato "a mano" in qualsiasi programma di dimensioni non banali.

## 19.1 Principi di gestione delle eccezioni

Tutto il sistema di gestione delle eccezioni del C++ è racchiuso in tre parole chiave: try, catch e throw. In termini generali, le istruzioni del programma che si desidera controllare devono essere contenute in un blocco try. Se all'interno di un blocco try si verifica un'eccezione (ovvero un errore), questa verrà "lanciata" tramite throw. L'eccezione viene poi "raccolta" tramite catch ed elaborata. La seguente discussione descrive più dettagliatamente questa procedura.

Il codice che si ritiene di dover monitorare deve essere eseguito all'interno di un blocco try (ma anche le funzioni richiamate dall'interno di try possono lanciare un'eccezione). Le eccezioni che possono essere lanciate dal codice monitorato vengono raccolte dall'istruzione catch che segue immediatamente l'istruzione try che ha lanciato l'eccezione. Di seguito sono illustrate le forme generali di try e catch:

```
try {
  // blocco try
}
catch (tipo1 arg) {
  // blocco catch
}
catch (tipo2 arg) {
  // blocco catch
}
catch (tipo3 arg) {
  // blocco catch
}
.
.
.
catch (tipoN arg) {
  // blocco catch
}
```

Il blocco try può avere dimensioni estremamente variabili da poche righe di una funzione fino all'intera funzione main() (per tenere sotto controllo l'intero programma).

Quando viene lanciata un'eccezione, questa viene raccolta dalla corrispondente istruzione catch che gestisce l'eccezione. È anche possibile associare più istruzioni catch a un determinato try. L'istruzione catch utilizzata verrà determinata dal tipo dell'eccezione. Ovvero se il tipo di dati specificato da un'istruzione catch corrisponde con quello dell'eccezione, verrà eseguita tale istruzione catch (e tutte le altre verranno ignorate). Quando viene raccolta un'eccezione, *arg* riceverà il proprio valore. È possibile raccogliere qualsiasi tipo di dati, incluse le classi create dal programmatore. Se non viene lanciata alcuna eccezione (ovvero all'interno del blocco try non si verifica alcun errore) non verrà eseguita alcuna istruzione catch.

La forma generale dell'istruzione throw è:

throw *eccezione*;

throw genera l'eccezione specificata da eccezione. Se l'eccezione deve essere raccolta, throw deve essere eseguita dall'interno dello stesso blocco try o da qualsiasi altra funzione richiamata (direttamente o indirettamente) dall'interno del blocco try. Il valore di *eccezione* è il valore "lanciato".

Se si lancia un'eccezione per la quale non vi è alcuna istruzione catch verrà provocata la fine anormale del programma. Il lancio di un'eccezione non gestita provoca la chiamata della funzione standard terminate(). Normalmente, terminate() richiama a sua volta abort() che termina il programma ma, come si vedrà in questo capitolo, è possibile specificare altri tipi di terminatori.



Ecco un esempio molto semplice che illustra il funzionamento del sistema di gestione delle eccezioni del C++:

```
// Un semplice esempio di gestione delle eccezioni.

#include <iostream>
using namespace std;

int main()
{
  cout << "Inizio\n";

  try { // inizio del blocco try
    cout << "Siamo all'interno del blocco try\n";
    throw 100; // "lancia" un errore
    cout << "Questa istruzione non viene eseguita";
  }
  catch (int i) { // raccoglie l'errore
    cout << "È stata raccolta un'eccezione -- il suo valore è: ";
    cout << i << "\n";
  }

  cout << "Fine";

  return 0;
}
```

Questo programma visualizza il seguente output:

```
Inizio
Siamo all'interno del blocco try
È stata raccolta un'eccezione -- il suo valore è: 100
Fine
```

Si osservi attentamente il programma precedente. Come si può vedere, vi è un blocco try contenente tre istruzioni e un'istruzione catch(int i) che elabora un'eccezione intera. All'interno del blocco try vengono eseguite solo due delle tre istruzioni: la prima istruzione cout e throw. Quando viene lanciata un'eccezione, il controllo passa all'espressione catch e il blocco try ha termine. Quindi, la catch *non* viene richiamata, piuttosto l'esecuzione del programma viene trasferita ad essa (lo stack del programma viene automaticamente aggiornato per rendere pos-

sibile l'operazione). Pertanto, l'istruzione cout che segue la throw non verrà mai eseguita.

Normalmente, il codice presente all'interno di un'istruzione catch cerca di risolvere un errore eseguendo un'azione appropriata. Se l'errore può essere corretto, l'esecuzione continuerà con l'istruzione che segue la catch. Spesso però un errore non può essere risolto e il blocco catch conclude il programma con una chiamata a exit() o abort().

Come si è detto, il tipo dell'eccezione deve corrispondere con il tipo specificato in un'istruzione catch. Ad esempio, nell'esempio precedente, se si cambia il tipo specificato nell'istruzione catch sostituendogli double, l'eccezione non verrà raccolta e il programma terminerà in modo anormale. Di seguito viene illustrata questa modifica:

```
// Questo esempio non funziona.

#include <iostream>
using namespace std;

int main()
{
  cout << "Inizio\n";

  try { // inizio del blocco try
    cout << "Siamo all'interno del blocco try\n";
    throw 100; // "lancia" un errore
    cout << "Questa istruzione non viene eseguita";
  }
  catch (double i) { // Non funziona per eccezioni int
    cout << "È stata raccolta un'eccezione -- il suo valore è: ";
    cout << i << "\n";
  }

  cout << "Fine";

  return 0;
}
```

Questo programma produce il seguente output poiché l'eccezione intera non verrà raccolta dall'istruzione catch(double i).

```
Inizio
Siamo all'interno del blocco try
Fine anormale del programma
```



L'eccezione può essere lanciata anche dall'esterno del blocco try ma in una funzione richiamata da un'istruzione contenuta all'interno del blocco try. Ad esempio, questo è un programma perfettamente corretto:

```
/* Il lancio dell'eccezione avviene da una funzione
   che si trova all'esterno del blocco try. */

#include <iostream>
using namespace std;

void Xtest(int test)
{
  cout << "All'interno di Xtest, test è uguale a " << test << "\n";
  if(test) throw test;
}

int main()
{
  cout << "Inizio\n";

  try { // inizio del blocco try
    cout << "Siamo all'interno del blocco try\n";
    Xtest(0);
    Xtest(1);
    Xtest(2);
  }
  catch (int i) { // raccoglie l'errore
    cout << "È stata raccolta un'eccezione -- il suo valore è: ";
    cout << i << "\n";
  }

  cout << "Fine";

  return 0;
}
```

Questo programma produce il seguente output:

```
Inizio
Siamo all'interno del blocco try
All'interno di Xtest, test è uguale a 0
All'interno di Xtest, test è uguale a 1
È stata raccolta un'eccezione -- il suo valore è: 1
Fine
```

Un blocco try può essere inserito anche all'interno di una funzione. In questo caso, ogni volta che il programma accede alla funzione, verrà reinizializzato il sistema di gestione delle eccezioni rispetto a tale funzione. Ad esempio, si esamini il seguente programma:

```
#include <iostream>
using namespace std;

/* le istruzioni try/catch possono anche trovarsi
   all'interno di una funzione diversa da main(). */

void Xhandler(int test)
{
  try{
    if(test) throw test;
  }
  catch(int i) {
    cout << "Raccolta l'eccezione n. " << i << '\n';
  }
}

int main()
{
  cout << "Inizio\n";

  Xhandler(1);
  Xhandler(2);
  Xhandler(0);
  Xhandler(3);

  cout << "Fine";

  return 0;
}
```

Questo programma visualizza il seguente output:

```
Inizio
Raccolta l'eccezione n. 1
Raccolta l'eccezione n. 2
Raccolta l'eccezione n. 3
Fine
```

Come si può vedere, vengono lanciate tre eccezioni. Al termine di ogni eccezione, avviene l'uscita dalla funzione.



Quando la funzione viene nuovamente richiamata, il sistema di gestione delle eccezioni viene reinizializzato.

È importante comprendere che il codice associato a un'istruzione catch verrà eseguito solo se raccoglie un'eccezione. In caso contrario, l'eccezione salterà semplicemente l'istruzione catch (ovvero l'esecuzione non entrerà nell'istruzione catch). Ad esempio, nel programma seguente, non verrà lanciata alcuna eccezione e quindi non verrà eseguita alcuna istruzione catch.

```
#include <iostream>
using namespace std;

int main()
{
  cout << "Inizio\n";

  try { // inizio del blocco try
    cout << "Siamo all'interno del blocco try\n";
    cout << "Siamo ancora all'interno del blocco try\n";
  }
  catch (int i) { // raccoglie l'errore
    cout << "È stata raccolta un'eccezione -- il suo valore è: ";
    cout << i << "\n";
  }

  cout << "Fine";

  return 0;
}
```

Questo programma produce il seguente output:

```
Inizio
Siamo all'interno del blocco try
Siamo ancora all'interno del blocco try
Fine
```

Come si può vedere, l'istruzione catch viene saltata dal flusso dell'esecuzione.

## Eccezioni per classi

Un'eccezione può essere di qualsiasi tipo, incluse le classi create per il programma. In realtà, nei programmi la maggior parte delle eccezioni riguarda le classi e non i tipi standard del linguaggio. Il motivo più comune per il quale si definisce

una classe per eccezioni consiste nel creare un oggetto che descriverà l'errore verificatosi. Questa informazione può essere utilizzata dal gestore dell'eccezione per elaborare l'errore. Questa possibilità è illustrata dal seguente programma.

```
// Raccolta di eccezioni per classi.
#include <iostream>
#include <cstring>
using namespace std;

class MyException {
public:
  char str_what[80];
  int what;

  MyException() { *str_what = 0; what = 0; }

  MyException(char *s, int e) {
    strcpy(str_what, s);
    what = e;
  }
};

int main()
{
  int i;

  try {
    cout << "Immettere un numero positivo: ";
    cin >> i;
    if(i<0)
      throw MyException("Il numero è negativo", i);
  }
  catch (MyException e) { // Raccoglie l'errore
    cout << e.str_what << ": ";
    cout << e.what << "\n";
  }
  return 0;
}
```

Ecco una prova di esecuzione:

```
Immettere un numero positivo: -4
Il numero è negativo: -4
```



Il programma chiede all'utente un numero positivo. Se viene specificato un numero negativo, il programma crea un oggetto della classe MyException che descrive l'errore. Pertanto MyException incapsula le informazioni riguardanti l'errore. Queste informazioni vengono poi utilizzate dal gestore dell'eccezione. In generale è opportuno creare classi per eccezioni che incapsulino le informazioni relative a un errore per consentire al gestore delle eccezioni di rispondere in modo appropriato.

## Uso di più istruzioni catch

Come si è detto, è possibile associare a una try più di una catch. Nella pratica ciò avviene molto frequentemente. Tuttavia, ogni catch deve raccogliere un tipo diverso di eccezione. Ad esempio, questo programma raccoglie interi e stringhe.

```
#include <iostream>
using namespace std;

// Possono essere raccolti vari tipi di eccezioni.

void Xhandler(int test)
{
  try{
    if(test) throw test;
    else throw "Il valore è uguale a zero";
  }
  catch(int i) {
    cout << "Raccolta l'eccezione n. " << i << '\n';
  }
  catch(char *str) {
    cout << "Raccolta una stringa: ";
    cout << str << '\n';
  }
}

int main()
{
  cout << "Inizio\n";

  Xhandler(1);
  Xhandler(2);
  Xhandler(0);
  Xhandler(3);
```

```
  cout << "Fine";

  return 0;
}
```

Questo programma produce il seguente output:

```
Inizio
Raccolta l'eccezione n. 1
Raccolta l'eccezione n. 2
Raccolta una stringa: Il valore è uguale a zero
Raccolta l'eccezione n. 3
Fine
```

Come si può vedere, ogni istruzione catch risponde solo a eccezioni del proprio tipo.

In generale, le espressioni catch vengono controllate nell'ordine in cui appaiono nel programma. Verrà eseguita solo l'istruzione corrispondente e tutti gli altri blocchi catch verranno ignorati.

## 19.2 Gestione delle eccezioni per classi derivate

Occorre fare attenzione all'ordine delle istruzioni catch quando si cerca di raccogliere dei tipi di eccezioni che riguardano classi base e classi derivate in quanto una clausola catch per una classe base risponde anche ad ogni classe derivata da tale base. Pertanto se si vogliono raccogliere eccezioni sia della classe base che della classe derivata si deve porre la classe derivata per prima nella sequenza catch. In caso contrario, il catch della classe base raccoglierà anche le eccezioni per le classi derivate. Ad esempio, si consideri il seguente programma.

```
// Raccolta di eccezioni per classi derivate.
#include <iostream>
using namespace std;

class B {
};

class D: public B {
};

int main()
```



```
{
  D derived;

  try {
    throw derived;
  }
  catch(B b) {
    cout << "Raccolta la classe base.\n";
  }
  catch(D d) {
    cout << "Non viene eseguita.\n";
  }

  return 0;
}
```

Qui, poiché derived è un oggetto che deriva da B, verrà raccolto dalla prima clausola catch e la seconda clausola non verrà mai eseguita. Alcuni compilatori rilevano questa condizione con un messaggio warning. Altri producono un messaggio d'errore. In entrambi i casi, per correggere la situazione basta invertire l'ordine delle clausole catch.

## 19.3 Opzioni della gestione delle eccezioni

Il sistema di gestione delle eccezioni del C++ prevede numerose funzionalità e sfumature che ne semplificano e agevolano l'uso.

### Raccolta di tutte le eccezioni

In alcune circostanze si desidera che un gestore di eccezioni raccolga tutte le eccezioni e non solo quelle di un determinato tipo. A tale scopo basta utilizzare la seguente forma di catch:

```
catch(...) {
  // elabora tutte le eccezioni
}
```

Qui, i tre puntini di sospensione chiedono di accettare qualsiasi tipo di dati. Il seguente programma illustra l'uso di catch(...).

```
// Questo esempio raccoglie tutte le eccezioni.

#include <iostream>
using namespace std;

void Xhandler(int test)
{
  try{
    if(test==0) throw test; // lancia un int
    if(test==1) throw 'a'; // lancia un char
    if(test==2) throw 123.23; // lancia un double
  }
  catch(...) { // raccoglie tutte le eccezioni
    cout << "Raccolta un'eccezione!\n";
  }
}

int main()
{
  cout << "Inizio\n";

  Xhandler(0);
  Xhandler(1);
  Xhandler(2);

  cout << "Fine";

  return 0;
}
```

Questo programma visualizza il seguente output:

```
Inizio
Raccolta un'eccezione!
Raccolta un'eccezione!
Raccolta un'eccezione!
Fine
```

Come si può vedere, le tre eccezioni lanciate tramite throw vengono raccolte da un'unica istruzione catch.

Inoltre, catch(...) consente di raccogliere solo determinati sottoinsiemi di eccezioni. In questo modo, la forma catch(...) si dimostra utilissima per fungere da gestore di raccolta generalizzata. Ad esempio, questa versione del programma precedente raccoglie esplicitamente le eccezioni intere mentre lascia ogni altra eccezione a catch(...).



```
// Questo esempio impiega catch(...) come default.

#include <iostream>
using namespace std;

void Xhandler(int test)
{
  try{
    if(test==0) throw test; // lancia un int
    if(test==1) throw 'a'; // lancia un char
    if(test==2) throw 123.23; // lancia un double
  }
  catch(int i) { // raccoglie le eccezioni int
    cout << "Raccolta un'eccezione int\n";
  }
  catch(...) { // raccoglie tutte le altre eccezioni
    cout << "Raccolta un'eccezione!\n";
  }
}

int main()
{
  cout << "Inizio\n";

  Xhandler(0);
  Xhandler(1);
  Xhandler(2);

  cout << "Fine";

  return 0;
}
```

Questo programma visualizza il seguente output:

```
Inizio
Raccolta un'eccezione int
Raccolta un'eccezione!
Raccolta un'eccezione!
Fine
```

Come si può vedere in questo esempio, la forma catch(...) è ottima per raccogliere tutte le eccezioni che non si desidera gestire esplicitamente. Inoltre, in questo modo si raccoglieranno tutte le eccezioni prodotte evitando quindi la possibilità che si verifichi una chiusura anormale del programma.

## Restrizione delle eccezioni

È anche possibile restringere il tipo di eccezioni che la funzione può lanciare. Addirittura si può impedire che una funzione possa lanciare qualsiasi eccezione. Per attivare queste restrizioni, si deve aggiungere alla definizione della funzione una clausola throw:

*tipo-restituito nome-funzione*(*elenco-argomenti*) throw(*elenco-tipi*)
{
 // ...
}

Qui, la funzione potrà lanciare eccezioni relative ai soli tipi di dati contenuti in *elenco-tipi* e separati da virgole. La generazione di altri tipi di espressione provocherà la fine anormale del programma. Se si desidera che una funzione non possa essere in grado di lanciare eccezioni, si dovrà utilizzare un elenco vuoto.

Ogni tentativo di lanciare un'eccezione non consentita a una funzione provocherà la chiamata della funzione unexpecterd() contenuta nella libreria standard. Normalmente tale funzione richiama a sua volta la funzione abort() che provoca la fine anormale del programma. Se si preferisce è però possibile specificare altri gestori di terminazione, come descritto più avanti in questo stesso capitolo.

Il seguente programma mostra il modo in cui è possibile restringere i tipi delle eccezioni che possono essere lanciate da una funzione.

```
// Riduzione dei tipi di eccezioni lanciabili da una funzione.

#include <iostream>
using namespace std;

// Questa funzione può lanciare solo int, char e double.
void Xhandler(int test) throw(int, char, double)
{
  if(test==0) throw test; // lancia un int
  if(test==1) throw 'a'; // lancia un char
  if(test==2) throw 123.23; // lancia un double
}

int main()
{
  cout << "Inizio\n";

  try{
    Xhandler(0); // cerca anche di passare 1 e 2 a Xhandler()
  }
  catch(int i) {
    cout << "Raccolta un'eccezione int\n";
  }
  catch(char c) {
    cout << "Raccolta un eccezione char\n";
  }
  catch(double d) {
    cout << "Raccolta un eccezione double\n";
  }

  cout << "Fine";

  return 0;
}
```



In questo programma, la funzione Xhandler() può lanciare eccezioni intere, di caratteri e double. Se la funzione tenterà di lanciare un altro tipo di eccezioni, provocherà la fine anormale del programma (ovvero verrà richiamata la funzione unexpected()). Per vedere un esempio di ciò, si provi a eliminare il tipo int dall'elenco e a rieseguire il programma.

È importante comprendere che le limitazioni per la funzione riguardano solo i tipi delle eccezioni che vengono lanciate dal blocco try che l'ha richiamata. In altre parole, un blocco try che si trova *all'interno* di una funzione può lanciare qualsiasi tipo di eccezione sempre che questa venga raccolta all'interno di tale funzione. La restrizione si applica solo quando il lancio di un'eccezione viene eseguito all'esterno di una funzione. La seguente modifica a Xhandler() evita che questa possa lanciare eccezioni.

```
// Questa funzione NON può lanciare eccezioni!

void Xhandler(int test) throw()
{
  /* Le seguenti istruzioni non funzionano e provocano
     una fine anormale del programma. */
  if(test==0) throw test;
  if(test==1) throw 'a';
  if(test==2) throw 123.23;
}
```

**NOTA** *Al momento attuale, il compilatore Microsoft Visual C++ non supporta la clausola throw() per le funzioni.*

### Rilancio di un'eccezione

Se si desidera rilanciare un'eccezione dall'interno del gestore dell'eccezione stessa, basterà richiamare throw senza specificare alcuna eccezione. In questo modo l'eccezione corrente verrà passata a una sequenza try/catch più esterna. Questa tecnica consente di accedere alle eccezioni da parte di più gestori. Ad esempio un gestore potrebbe occuparsi si un aspetto di un'eccezione mentre un secondo gestore potrebbe concentrarsi solo su un altro. Un'eccezione può essere rilanciata solo dall'interno del blocco catch (o da una funzione richiamata all'interno del blocco). Quando si rilancia un'eccezione, questa non verrà nuovamente raccolta dalla stessa istruzione catch ma piuttosto si propagherà all'istruzione catch successiva. Il seguente programma illustra il rilancio di un'eccezione di tipo char*.

```
// Esempio di "rilancio" di un'eccezione.

#include <iostream>
using namespace std;

void Xhandler()
{
  try {
    throw "salve"; // lancia un char *
  }
  catch(char *) { // raccoglie un char *
    cout << "Raccolta un eccezione char * in Xhandler\n";
    throw ; // rilancia un char * fuori dalla funzione
  }
}

int main()
{
  cout << "Inizio\n";

  try{
    Xhandler();
  }
  catch(char *) {
    cout << "Raccolta un eccezione char * in main\n";
  }

  cout << "Fine";

  return 0;
}
```



Questo programma visualizza il seguente output:

```
Inizio
Raccolta un eccezione char * in Xhandler
Raccolta un eccezione char * in main
Fine
```

## 19.4 Le funzioni terminate() e unexpected()

Come si è detto in precedenza, quando qualcosa va storto durante il processo di gestione delle eccezioni, vengono richiamate le funzioni terminate() e unexpected(). Tali funzioni vengono fornite dalla libreria standard del C++. Questi sono i loro prototipi:

```
void terminate( );
void unexpected( );
```

Queste funzioni richiedono l'impiego dell'header <exception>.

La funzione terminate() viene richiamata quando il sottosistema di gestione delle eccezioni non riesce a trovare un'istruzione catch che corrisponda a un'eccezione. Viene richiamata anche se il programma tenta di rilanciare un'eccezione quando in precedenza non era stata lanciata alcuna eccezione. La funzione terminate() viene richiamata anche in altre circostanze. Ad esempio, quando nel processo di chiusura dello stack causato da un'eccezione, il distruttore di un oggetto lancia un'eccezione. In generale, terminate() è l'ultimo livello di gestione delle eccezioni, quando non sono disponibili altri gestori. Normalmente terminate() richiama abort().

La funzione unexpected() viene richiamata quando una funzione tenta di lanciare un'eccezione che non è consentita dall'elenco throw. Normalmente unexpected() richiama terminate().

### Impostazione dei gestori di terminate() e unexpected()

Le funzioni terminate() e unexpected() non fanno altro che richiamare altre funzioni che gestiscono un errore. Come si è appena detto, normalmente terminate() richiama abort() e unexpected() richiama terminate(). Pertanto in genere entrambe le funzioni bloccano l'esecuzione del programma ogni volta che si verifica un errore di gestione delle eccezioni. È tuttavia possibile cambiare le funzioni richiamate da terminate() e unexpected() per dare al programma il pieno controllo del sottosistema di gestione delle eccezioni.

Per cambiare il gestore di terminate() si usa la funzione set_terminate():

```
terminate_handler set_terminate(terminate_handler newhandler) throw( );
```

Qui *newhandler* è un puntatore al nuovo gestore di terminate(). La funzione restituisce un puntatore al vecchio gestore di terminate(). Il nuovo gestore di terminate() deve essere di tipo terminate_handler, definito nel seguente modo:

```
typedef void (*terminate_handler) ( );
```

L'unica cosa che il gestore di terminate() deve fare è fermare l'esecuzione del programma. Non deve tornare al programma o riprendere l'esecuzione in alcun modo. Per cambiare il gestore di unexpected() si usa set_unexpected():

```
unexpected_handler set_unexpected(unexpected_handler newhandler) throw( );
```

Qui newhandler è un puntatore al nuovo gestore di unexpected(). La funzione restituisce un puntatore al vecchio gestore di unexpected(). Il nuovo gestore di unexpected() deve essere di tipo unexpected_handler, definito nel seguente modo:

```
typedef void (*unexpected_handler) ( );
```

Anche questo gestore deve lanciare un'eccezione, fermare il programma o richiamare terminate() e non deve tornare al programma.

Sia set_terminate() che set_unexpected() richiedono l'uso dell'header <exception>.

Ecco un semplice esempio che definisce uno specifico gestore di terminate().

```
// Uso del gestore per terminate().
#include <iostream>
#include <cstdlib>
#include <exception>
using namespace std;

void my_Thandler() {
  cout << "Nel nuovo gestore di terminate()\n";
  abort();
}

int main()
{
```



```
// impostazione del nuovo gestore per terminate()
set_terminate(my_Thandler);

try {
  cout << "Nel blocco try";
  throw 100; // lancia un errore
}
catch (double i) { // non rileva un'eccezione int
  // ...
}

return 0;
}
```

Elenco l'output prodotto dal programma.

```
Nel blocco try
Nel nuovo gestore di terminate()
fine anormale del programma
```

## 19.5 La funzione uncaught_exception()

Il sistema di gestione delle eccezioni del C++ fornisce un'altra funzione che talvolta è molto utile: uncaught_exception(). Ecco il suo prototipo:

```
bool uncaught_exception( );
```

Questa funzione restituisce true quando un'eccezione lanciata non è ancora stata raccolta. Dopo aver raccolto un'eccezione la funzione restituisce false.

## 19.6 Le classi exception e bad_exception

Quando una funzione fornita dalla libreria standard C++ lancia un'eccezione, questo sarà un oggetto derivato dalla classe base exception. Il gestore di unexpected() può lanciare un oggetto della classe bad_exception. Queste classi richiedono l'impiego dell'header <exception>.

## 19.7 Applicazioni della gestione delle eccezioni

La gestione delle eccezioni fornisce al programma un metodo strutturato per gestire eventi anormali. Questo significa che il gestore dell'errore deve eseguire un'operazione che ponga rimedio all'errore. Ad esempio, si consideri il seguente breve programma. Il programma legge due numeri e divide il primo per il secondo utilizzando la gestione delle eccezioni per gestire errori per divisione per zero:

```
#include <iostream>
using namespace std;

void divide(double a, double b);

int main()
{
  double i, j;

  do {
    cout << "Immettere il numeratore (0 per uscire): ";
    cin >> i;
    cout << "Immettere il denominatore: ";
    cin >> j;
    divide(i, j);
  } while(i != 0);

  return 0;
}

void divide(double a, double b)
{
  try {
    if(!b) throw b; // verifica della divisione per zero
    cout << "Risultato: " << a/b << endl;
  }
  catch (double b) {
    cout << "Non si può dividere un numero per zero.\n";
  }
}
```

Anche se il programma precedente è un esempio molto semplice, illustra la natura essenziale della gestione delle eccezioni. Poiché la divisione per zero non è consentita, il programma non può continuare nel caso in cui venga immesso uno zero come secondo numero. In questo caso, l'eccezione semplicemente non esegue la divisione (che provocherebbe la conclusione anormale del programma) e notifica l'errore all'utente. Il programma quindi chiede di immettere altri due



numeri. Quindi, l'errore è stato gestito in modo ordinato e l'utente potrà continuare l'esecuzione del programma. Lo stesso concetto si applica anche a situazioni più complesse.

La gestione delle eccezioni è particolarmente utile per uscire da una serie di routine molto nidificate in cui si è verificato un grave errore. In questo senso, la gestione delle eccezioni del C++ può anche sostituire il rozzo approccio del C che si basa sulle funzioni setjmp() e longjmp().

Il motivo principale che consiglia l'uso dei gestori di eccezioni è la possibilità di utilizzare un metodo ordinato per gestire gli errori. Questo significa anche che, in alcuni casi, sarà possibile correggere la situazione.

Capitolo 20

# Il sistema di I/O C++: le basi


20.1 **Operazioni di I/O C++ vecchie e nuove**
20.2 **Gli stream del C++**
20.3 **Le classi per stream C++**
20.4 **Operazioni di I/O formattato**
20.5 **Overloading di << e >>**
20.6 **Creazione di funzioni di manipolazione**


Il C++ supporta due diversi sistemi di I/O: quello ereditato dal C e il proprio sistema di I/O a oggetti che in seguito verrà semplicemente chiamato sistema di I/O C++. Il sistema di I/O C è stato già descritto nella Parte prima e quindi qui si parlerà unicamente del sistema di I/O C++. Anche il sistema di I/O C++ (come quello C) è perfettamente integrato nel linguaggio. Questo significa che i vari aspetti del sistema di I/O C++, come le operazioni di I/O su console e su disco, sono in realtà aspetti diversi dello stesso meccanismo.

Questo capitolo discute le basi del sistema di I/O a oggetti del C++. Sebbene gli esempi di questo capitolo utilizzino operazioni di I/O su console, le informazioni si applicano anche ad altri dispositivi, inclusi i file (discussi nel Capitolo 21).

Poiché il sistema di I/O C è estremamente ricco, flessibile e potente, ci si potrebbe chiedere perché il C++ debba definire un nuovo sistema. Questo è dovuto al fatto che il sistema di I/O C non prevede l'uso di oggetti. Pertanto, per fare in modo che il C++ fornisse un completo supporto alla programmazione a oggetti, era necessario creare un sistema di I/O che potesse operare sugli oggetti definiti dall'utente. Oltre al supporto degli oggetti, vi sono altri vantaggi derivanti dall'uso del sistema di I/O C++ anche in programmi che non fanno grande uso (o non fanno affatto uso) di oggetti definiti dall'utente. In pratica, tutto il nuovo codice C++ impiega le nuove funzionalità a oggetti mentre il sistema di I/O C è stato mantenuto solo per motivi di compatibilità.

In questo capitolo viene descritta la formattazione dei dati. In particolare vengono esposti i metodi di overloading degli operatori << e >> del C++ per consentirne l'uso con tutte le classi. Inoltre si vedrà il modo in cui è possibile creare particolari funzioni di I/O, i manipolatori, in grado di rendere più efficienti i programmi.

## 20.1 Operazioni di I/O C++ vecchie e nuove

Attualmente sono in uso due versioni di librerie di I/O a oggetti: la più vecchia si basa sulle specifiche originali del C++ mentre la più recente è quella definita dal C++ standard. La vecchia libreria di I/O è supportata dal file header <iostream.h>. La nuova libreria di I/O è supportata dall'header <iostream>. Nella maggior parte dei casi, per il programmatore queste due librerie hanno lo stesso aspetto. Questo è dovuto al fatto che la nuova libreria di I/O è in pratica una versione migliorata ed estesa della vecchia. Infatti la maggior parte delle differenze fra le due riguardano ciò che si trova "sotto la superficie" ovvero il modo in cui le librerie sono implementate e non il loro uso. Dal punto di vista del programmatore, esistono principalmente due differenze fra queste due librerie di I/O. Innanzitutto la nuova libreria di I/O contiene alcune funzionalità e definisce alcuni nuovi tipi di dati. Pertanto la nuova libreria di I/O è in pratica un super-insieme della vecchia. Quasi tutti i programmi originariamente scritti per la vecchia libreria possono essere compilati senza modifiche sostanziali con la nuova. In secondo luogo, la libreria di I/O vecchio stile opera nel namespace globale. La nuova libreria opera nel namespace std (il namespace utilizzato da tutte le librerie del C++ standard). Dato che la vecchia libreria di I/O è obsoleta, questo volume descrive solo la nuova libreria ma la maggior parte delle informazioni è applicabile anche alla vecchia versione.

## 20.2 Gli stream del C++

Anche il sistema di I/O C++ opera mediante stream. Gli stream sono stati discussi in dettaglio nel Capitolo 9 e tale argomento non verrà trattato nuovamente in questo capitolo. Per riassumere, uno stream è un dispositivo logico che produce o consuma informazioni collegato a un dispositivo fisico tramite il sistema di I/O C++. Tutti gli stream si comportano nello stesso modo anche se il dispositivo fisico a cui sono connessi può essere molto diverso. Per questa ragione, le stesse funzioni di I/O C++ possono operare praticamente su qualsiasi tipo di dispositivo fisico. Ad esempio, si potrà utilizzare la stessa funzione che scrive su un file per scrivere sulla stampante o sullo schermo. Il vantaggio di questo approccio consiste nel fatto che si deve apprendere un solo sistema di I/O.

## 20.3 Le classi per stream C++

Come si è detto, il C++ standard fornisce il supporto al proprio sistema di I/O nell'header <iostream>. In questo file, viene definito un insieme piuttosto complesso di gerarchie di classi che supporta le operazioni di I/O.



Le classi di I/O partono da un sistema di classi template. Come si è detto nel Capitolo 18, una classe template definisce la forma di una classe senza specificare completamente i dati su cui essa si troverà ad operare. Dopo aver definito una classe template, è possibile crearne istanze specifiche. Per quanto riguarda la libreria di I/O, il C++ standard crea due specializzazioni delle classi template: una per caratteri a 8 bit e una per caratteri estesi. Questo volume utilizzerà solo le classi per caratteri a 8 bit in quanto sono di gran lunga quelle più utilizzate, ma le stesse tecniche si applicano a entrambe.

Il sistema di I/O del C++ si basa su due gerarchie di classi template differenti ma correlate. La prima deriva dalla classe di I/O a basso livello chiamata basic_streambuf. Questa classe fornisce le operazioni di input e output a basso livello e fornisce il supporto per l'intero sistema di I/O del C++. Se non si devono eseguire operazioni di I/O molto avanzate, è raro che si debba usare direttamente basic_streambuf. La gerarchia di classi su cui ci si troverà più frequentemente a operare è derivata da basic_ios. Questa è una classe di I/O ad alto livello che fornisce operazioni di formattazione, verifica degli errori e gestione delle informazioni di stato rispetto alle operazioni di I/O su stream. La classe base di basic_ios si chiama ios_base e definisce numerose versioni non template utilizzate da basic_ios. basic_ios viene utilizzata come base per varie classi derivate, fra le quali basic_istream, basic_ostream e basic_iostream. Queste classi sono utilizzate rispettivamente per creare stream di input, output e input/output. Come si è detto, la libreria di I/O crea due specializzazioni delle gerarchie di classi template: una per caratteri a 8 bit e una per caratteri estesi. Ecco un elenco che rappresenta il mappaggio dei nomi delle classi template nelle loro versioni a caratteri semplici e caratteri estesi.

| CLASSE TEMPLATE | CLASSE PER CARATTERI SEMPLICI | CLASSE PER CARATTERI ESTESI |
|---|---|---|
| basic_streambuf | streambuf | wstreambuf |
| basic_ios | ios | wios |
| basic_istream | istream | wistream |
| basic_ostream | ostream | wostream |
| basic_iostream | iostream | wiostream |
| basic_fstream | fstream | wfstream |
| basic_ifstream | ifstream | wifstream |
| basic_ofstream | ofstream | wofstream |

Nel volume verranno utilizzate le classi per caratteri semplici, così come avviene normalmente nei programmi. Esse hanno anche lo stesso nome utilizzato dalla

vecchia libreria di I/O. Questo è il motivo per cui la vecchia e la nuova libreria di I/O sono compatibili fra loro a livello di codice sorgente.
La classe ios contiene molte funzioni e variabili membro che controllano o monitorizzano le operazioni di base di uno stream. Tali elementi vengono utilizzati con grande frequenza. Basterà ricordare che se si include <iostream> nel programma, si fa accesso a questa importante classe.

### Gli stream predefiniti del C++

Quando inizia l'esecuzione di un programma C++ vengono automaticamente aperti quattro stream di I/O:

| STREAM | SIGNIFICATO | DISPOSITIVO |
|---|---|---|
| cin | Input standard | Tastiera |
| cout | Output standard | Schermo |
| cerr | Output di errori standard | Schermo |
| clog | Versione bufferizzata di cerr | Schermo |

Gli stream cin, cout e cerr corrispondono agli stream stdin, stdout e stderr del C.
Normalmente, gli stream standard sono utilizzati per comunicare con la console. In ambienti che consentono la redirezione delle operazioni di I/O (come ad esempio DOS, UNIX, OS/2 e Windows), gli stream standard possono essere rediretti ad altri dispositivi o file. Per semplicità, negli esempi di questo capitolo si assume che non sia consentita la redirezione delle operazioni di I/O.
Il C++ standard definisce anche quattro stream aggiuntivi: win, wout, werr e wlog. Si tratta delle versioni a caratteri estesi ("wide") degli stream standard. I caratteri wide sono del tipo wchar_t, che corrisponde generalmente a valori a 16 bit, e sono utilizzati per contenere interi alfabeti associati ad alcune lingue.

## 20.4 Operazioni di I/O formattato

Il sistema di I/O C++ consente di formattare le operazioni di I/O. Ad esempio, è possibile impostare l'ampiezza dei campi, specificare una base numerica o determinare il numero di cifre da visualizzare dopo il punto decimale.
I dati possono essere formattati in due modi correlati ma concettualmente differenti. Innanzi tutto, si può accedere direttamente ai membri della classe ios. In particolare, è possibile impostare vari indicatori (flag) di stato definiti all'interno della classe ios o richiamare varie funzioni membro di ios. In alternativa è possibile utilizzare particolari funzioni chiamate *manipolatori* che possono essere incluse all'interno di un'espressione di I/O.



La discussione inizierà dalle operazioni di I/O formattato utilizzando le funzioni membro e i flag della classe ios.

### Formattazione utilizzando i membri di ios

Ad ogni stream è associata una serie di flag che controllano la formattazione delle informazioni. La classe ios dichiara un'enumerazione a bit chiamata fmtflags nella quale sono definiti i seguenti valori (tecnicamente questi valori sono definiti in ios_base che, come si è detto in precedenza è la classe base di ios.

| | | | |
|---|---|---|---|
| adjustfield | basefield | boolalpha | dec |
| fixed | floatfield | hex | internal |
| left | oct | right | scientific |
| showbase | showpoint | showpos | skipws |
| unitbuf | uppercase | | |

Questi valori sono utilizzati per impostare o cancellare il valore dei flag di formattazione. I vecchi compilatori non definiscono il tipo enumerativo fmtflags. In questo caso i flag di formattazione verranno codificati in un intero long.
Quando si attiva il flag skipws, tutti gli spazi vuoti iniziali (spazi, tabulazioni e codici di fine riga) vengono eliminati dall'input. Quando skipws è uguale a zero, gli spazi vuoti non vengono eliminati.
Quando il flag left è attivo, l'output viene allineato a sinistra. Quando è attivo right, l'output viene allineato a destra. Quando è attivo il flag internal, i valori numerici vengono completati da una serie di spazi inseriti fra il segno o il carattere di base in modo da far rientrare il numero in un campo. Nel caso in cui nessuno di questi flag fosse impostato, l'output viene allineato a destra.
Come impostazione standard, i valori numerici vengono visualizzati in base decimale. È però possibile cambiare la base numerica: impostando il flag oct, l'output viene visualizzato in ottale. Impostando il flag hex l'output viene visualizzato in numeri esadecimali. Per tornare a visualizzare l'output in numeri decimali, si deve impostare il flag dec.
Impostando showbase viene visualizzata la base dei valori numerici. Ad esempio, se la base numerica utilizzata è esadecimale, il valore 1F viene visualizzato come 0x1F.
Quando si utilizza la notazione scientifica la "e" dell'esponente sarà in lettere minuscole come pure la "x" che indica l'utilizzo della base esadecimale. Quando si attiva il flag uppercase questi caratteri vengono visualizzati in lettere maiuscole.
Impostando showpos si visualizza il segno + davanti ai valori positivi.

Impostando showpoint si visualizzano il punto e i decimali per tutti i numeri in virgola mobile (anche quando il numero non presenta decimali).

Impostando il flag scientific i numeri in virgola mobile vengono visualizzati in notazione scientifica. Quando si imposta fixed, i numeri in virgola mobile vengono visualizzati in notazione normale. Quando non è impostato alcun flag, sarà il compilatore a scegliere un metodo appropriato.

Quando è impostato unitbuf, il sistema di I/O viene vuotato (flush) dopo ogni operazione di inserimento. Quando si imposta boolalpha diventa possibile eseguire l'input e l'output di valori booleani utilizzando le parole true e false.

Poiché si fa frequentemente uso dei campi oct, dec ed hex, si può far riferimento alla loro globalità con il nome basefield. Analogamente i campi left, right e internal possono essere impiegati con il nome generale adjustfield. Infine i campi scientific e fixed sono raggruppati in floatfield.

## Impostazione dei flag di formattazione

Per impostare un flag di formattazione si utilizza la funzione setf(). Questa funzione è un membro di ios. Di seguito viene illustrata la sua forma più comune:

```
fmtflags setf(fmtflags flag);
```

Questa funzione restituisce la precedente impostazione del flag di formattazione e attiva i flag specificati da *flag*. Tutti gli altri flag rimangono inalterati.

Ad esempio, per attivare il flag showpos, si deve utilizzare l'istruzione:

```
stream.setf(ios::showpos);
```

Qui, stream è lo stream su cui si deve operare. Si noti che showpos è qualificato con ios::. Poiché showpos è una costante enumerativa definita dalla classe ios, quando viene utilizzata deve essere qualificata da ios::.

Il seguente programma visualizza il valore 100 attivando i flag showpos e showpoint.

```
#include <iostream>
using namespace std;

int main()
{
  cout.setf(ios::showpoint);
  cout.setf(ios::showpos);

  cout << 100; // visualizza +100.0
  return 0;
}
```



È importante comprendere che setf() è una funzione membro della classe ios e che pertanto opera sugli stream creati da tale classe. Pertanto, ogni chiamata a setf() verrà eseguita rispetto a un determinato stream. Non ha senso chiamare setf() da sola. In altre parole, in C++ non esiste il concetto di stato di formattazione globale. Ogni stream conserva le proprie informazioni sullo stato della formattazione.

Anche se non vi è nulla di tecnicamente errato nel precedente programma, vi è un modo più efficiente per scriverlo. Invece di eseguire più chiamate a setf(), basterà utilizzare l'operatore OR su tutti i flag da impostare. Ad esempio, la seguente chiamata esegue le stesse operazioni viste nel programma precedente.

```
// Si può anche eseguire l'OR di due o più flag,
cout.setf(ios::showpoint | ios::showpos);
```

**SUGGERIMENTO** *Poiché i flag di formattazione sono definiti nella classe ios, per accedere ai loro valori si deve utilizzare ios e l'operatore di risoluzione del campo d'azione. Ad esempio, showbase non verrà di per se stessa riconosciuta. Si dovrà specificare ios::showbase.*

## Cancellazione dei flag di formattazione

La funzione complementare di setf() è unsetf(). Questa funzione membro di ios consente di azzerare uno o più flag di formattazione. La sua forma generale è:

void unsetf(fmtflags *flag*);

Questa funzione azzera i flag specificati da *flag* (tutti gli altri flag non vengono modificati). Viene invece restituita l'impostazione precedente dei flag.

Il seguente programma illustra l'uso di unsetf(). Innanzi tutto il programma attiva i flag uppercase e scientific. Poi visualizza 100.12 in notazione scientifica. In questo caso, la E utilizzata per la notazione scientifica apparirà in lettere maiuscole. Quindi il programma azzera il flag uppercase e visualizza nuovamente il numero 100.12 in notazione scientifica utilizzando una "e".

```
#include <iostream>
using namespace std;

int main()
```

```
{
  cout.setf(ios::uppercase | ios::scientific);

  cout << 100.12;  // visualizza 1.0012E+02

  cout.unsetf(ios::uppercase); // azzera uppercase

  cout << " \n" << 100.12; // visualizza 1.0012e+02

  return 0;
}
```

## Una forma di setf() modificata tramite overloading

Vi è una forma di setf() che ha il seguente aspetto:

fmtflags setf(fmtflags *flags1*, fmtflags *flags2*);

In questa versione, vengono modificati solo i flag specificati da *flag2*. Questi vengono prima azzerati e poi attivati sulla base dei flag specificati da *flag1*. Si noti che anche se *flag1* contiene anche altri flag, vengono alterati solo i flag specificati da *flag2*. La funzione restituisce l'impostazione precedente dei flag. Ad esempio:

```
#include <iostream>
using namespace std;

int main()
{
  cout.setf(ios::showpoint | ios::showpos, ios::showpoint);

  cout << 100.0 << " \n"; // visualizza 100.0 e non +100.0

  return 0;
}
```

Qui viene impostato showpoint ma non showpos che non è specificato dal secondo parametro.

L'uso più comune di questa forma a due parametri di setf() consente di impostare la base del numero, l'allineamento e i flag di formattazione. Come si è detto, i riferimenti ai campi oct, dec ed hex possono essere raccolti in basefield. Analogamente i campi left, right e internal possono essere raccolti in adjustfield. Infine i campi scientific e fixed possono essere raccolti in floatfield. Poiché i flag che formano questi raggruppamenti sono mutuamente esclusivi, occorre disattivare un flag quando se ne attiva un altro. Ad esempio, il seguente programma produce un output esadecimale. Per eseguire l'output in notazione esadecimale alcune implementazioni richiedono la disattivazione degli altri flag relativi alla base e l'attivazione del flag hex. Questa operazione può essere ottenuta con maggiore facilità se si usa la forma a due parametri di setf().



```
#include <iostream>
using namespace std;

int main()
{
  cout.setf(ios::hex, ios::basefield);

  cout << 100; // visualizza 64

  return 0;
}
```

Qui i flag di basefield (ovvero dec, oct ed hex) vengono innanzitutto cancellati e poi viene attivato il solo flag hex.

Si ricordi che solo i flag specificati da flags2 possono essere influenzati dai flag specificati da flags1. Ad esempio in questo programma il primo tentativo di impostare il flag showpos non ha successo.

```
// Questo programma non funziona.
#include <iostream>
using namespace std;

int main()
{
  cout.setf(ios::showpos, ios::hex);   // errore, showpos non è impostato

  cout << 100 << '\n';  // visualizza 100 e non +100

  cout.setf(ios::showpos, ios::showpos);  // corretto

  cout << 100;  // visualizza +100

  return 0;
}
```

La maggior parte delle volte si utilizzerà unsetf() per azzerare i flag e la versione a un unico parametro di setf(), descritta in precedenza per attivare i flag. La

versione setf(fmtflags *flag1*, fmtflags *flag2*) viene utilizzata in particolari situazioni, ad esempio per impostare la base del numero. Ad esempio si potrebbe utilizzare un modello di flag che specifichi lo stato di tutti i flag di formattazione ma si potrebbe voler modificare solo un paio di flag. In questo caso, si potrebbe specificare il modello in *flag1* e utilizzare *flag2* per specificare i flag da modificare.

## Il valore dei flag di formattazione

In alcuni casi si desidera semplicemente conoscere il valore attuale dei flag di formattazione senza modificarne alcuno. A tale scopo, ios include la funzione membro flags() che semplicemente restituisce l'impostazione corrente di ogni flag di formattazione codificata come un intero long. Il prototipo della funzione è il seguente:

```
fmtflags flags();
```

Il seguente programma utilizza flags() per visualizzare l'impostazione dei flag di formattazione di cout. Si faccia particolare attenzione alla funzione showflags() che potrebbe rivelarsi utile in molti programmi.

```
#include <iostream>
using namespace std;

void showflags() ;

int main()
{
  // mostra la condizione standard dei flag di formattazione
  showflags();

  cout.setf(ios::right | ios::showpoint | ios::fixed);

  showflags();

  return 0;
}

// Questa funzione visualizza lo stato dei flag di formattazione.

void showflags()
{
  ios::fmtflags f;
  long i;

  f = (long) cout.flags();  // legge le impostazioni dei flag

  // controlla ogni flag
  for(i=0X4000; i; i = i >> 1)
    if(i & f) cout << "1 ";
    else cout << "0 ";

  cout << " \n";
}
```

Di seguito viene presentato l'output del programma:

```
0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 1
0 1 0 0 0 1 0 1 0 0 1 0 0 0 1
```



## Attivazione di tutti i flag

La funzione flags() ha un secondo formato che consente di attivare tutti i flag associati a un determinato stream. Il prototipo di questa versione di flags() è:

```
fmtflags flags(fmtflags f);
```

Quando si utilizza questa versione della funzione, il modello di bit contenuto in *f* viene utilizzato per impostare i flag di formattazione associati allo stream. Pertanto, vengono modificati tutti i flag di formattazione. La funzione restituisce l'impostazione precedente.

Il seguente programma illustra questo utilizzo di flags(). Il programma innanzi tutto costruisce una maschera di flags che attiva showpos, showbase, oct e right. Tutti gli altri flag vengono disattivati. Il programma quindi utilizza flags() per impostare i flag di formattazione associati a cout. La funzione showflags() verifica che i flag siano stati impostati come indicato (è la stessa funzione utilizzata nel programma precedente).

```
#include <iostream>
using namespace std;

void showflags() ;

int main()
{
  // mostra la condizione standard dei flag di formattazione
  showflags();
```

```
  // showpos, showbase, oct, right sono on, gli altri sono off
  long f = ios::showpos | ios::showbase | ios::oct | ios:: right;
  cout.flags(f);  // attiva tutti i flag

  showflags();

  return 0;
}
```

## Uso di width(), precision() e fill()

Oltre ai flag di formattazione, all'interno della classe ios vi sono tre funzioni membro che impostano i seguenti parametri di formattazione: ampiezza del campo, precisione e carattere di riempimento.

Normalmente, quando viene prodotto in output un valore, esso occupa uno spazio pari al numero di caratteri visualizzato. È però possibile specificare un'ampiezza minima di campo tramite la funzione width(). Di seguito ne viene presentato il prototipo:

streamsize width(streamsize *w*);

Qui, *w* diviene l'ampiezza del campo e la funzione restituisce l'ampiezza precedente del campo. In alcune implementazioni, l'ampiezza del campo deve essere impostata prima di ogni operazione di output. In caso contrario, viene utilizzata un'ampiezza di campo standard. Il tipo streamsize corrisponde a una forma di intero.

Dopo aver impostato l'ampiezza minima del campo, se un valore non utilizza interamente l'ampiezza specificata, il campo viene riempito con il carattere di riempimento corrente (normalmente lo spazio) in modo da raggiungere l'ampiezza del campo. Se invece le dimensioni del valore superano l'ampiezza del campo, verranno superati i limiti. In nessun caso vengono troncati i valori.

Quando si produce in output un valore in virgola mobile, è possibile determinare il numero di cifre da visualizzare dopo il punto decimale utilizzando la funzione precision(). Il suo prototipo è:

streamsize precision(streamsize *p*);

Qui, la precisione viene impostata a *p* cifre decimali e viene restituito il valore precedente. La precisione standard è pari a sei cifre. In alcune implementazioni, la precisione deve essere impostata prima di ogni operazione di output di valori in virgola mobile. In caso contrario viene utilizzata una precisione standard.



Quando deve essere riempito un campo, il carattere utilizzato per riempire gli spazi vuoti è in genere lo spazio. È però possibile specificare un diverso carattere di riempimento utilizzando la funzione fill(). Il suo prototipo è:

char fill(char *ch*);

Dopo una chiamata a fill(), il nuovo carattere di riempimento sarà *ch* mentre la funzione restituirà il carattere precedentemente utilizzato. Il programma seguente illustra l'uso di queste tre funzioni:

```
#include <iostream>
using namespace std;

int main()
{
  cout.precision(4) ;
  cout.width(10);

  cout << 10.12345 << "\n";  // visualizza 10.12

  cout.fill('*');

  cout.width(10);
  cout << 10.12345 << "\n"; // visualizza *****10.12

  // l'ampiezza di campo si applica anche alle stringhe
  cout.width(10);
  cout << "Hi!" << "\n"; // visualizza *******Hi!
  cout.width(10);
  cout.setf(ios::left); // allineamento a sinistra
  cout << 10.12345; // visualizza 10.12*****

  return 0;
}
```

Di seguito viene presentato l'output del programma:

```
     10.12
*****10.12
*******Hi!
10.12*****
```

Esistono anche versioni di width(), precision() e fill() modificate tramite overloading che ottengono il valore corrente senza modificarlo:

```
char fill();
streamsize width();
streamsize precision();
```

## Uso dei manipolatori per la formattazione dell'I/O

Il secondo modo in cui è possibile modificare i parametri di formattazione di uno stream prevede l'uso di particolari funzioni, chiamate *manipolatori*, che possono essere incluse in un'espressione di I/O. Nella Tabella 20.1 vengono elencati tutti i manipolatori standard. Come si può vedere esaminando la tabella, molti manipolatori di I/O sono analoghi alle funzioni membro della classe ios.
Molti dei manipolatori sono stati aggiunti solo recentemente e non sono supportati dai compilatori meno recenti.

**Tabella 20.1** I manipolatori del C++.

| MANIPOLATORE | SCOPO | INPUT/OUTPUT |
|---|---|---|
| boolalpha | Attiva il flag boolalpha | Input e output |
| dec | Attiva il flag dec | Input e output |
| endl | Output del carattere di fine riga e svuotamento dello stream | Output |
| ends | Output di un carattere nullo | Output |
| fixed | Attiva il flag fixed | Output |
| flush | Svuotamento di uno stream | Output |
| hex | Attiva il flag hex | Input e output |
| internal | Attiva il flag internal | Output |
| left | Attiva il flag left | Output |
| noboolalpha | Disattiva il flag boolalpha | Input e output |
| noshowbase | Disattiva il flag showbase | Output |
| noshowpoint | Disattiva il flag showpoint | Output |
| noshowpos | Disattiva il flag showpos | Output |
| noskipws | Disattiva il flag skipws | Input |
| nounitbuf | Disattiva il flag unitbuf | Output |
| nouppercase | Disattiva il flag uppercase | Output |

(segue)



**Tabella 20.1** I manipolatori del C++. *(continua)*

| MANIPOLATORE | SCOPO | INPUT/OUTPUT |
|---|---|---|
| oct | Attiva il flag oct | Input e Output |
| resetioflags(fmtflags f) | Azzera i flag specificati in f | Input e Output |
| right | Attiva il flag right | Output |
| scientific | Attiva il flag scientific | Output |
| setbase(int base) | Imposta la base numerica | Output |
| setfill(int ch) | Imposta il carattere di riempimento | Output |
| setiosflags(fmtflags f) | Attiva i flag specificati in f | Input e output |
| setprecision(int p) | Imposta il numero di cifre di precisione | Output |
| setw(int w) | Imposta l'ampiezza di campo | Output |
| showbase | Attiva il flag showbase | Output |
| showpoint | Attiva il flag showpoint | Output |
| skipws | Attiva il flag skipws | Input |
| unitbuf | Attiva il flag unitbuf | Output |
| uppercase | Attiva il flag uppercase | Output |
| ws | Salta gli spazi vuoti iniziali | Input |

Per accedere ai manipolatori che richiedono l'uso di parametri (come ad esempio setw()) si deve includere nel programma l'header <iomanip>.
Ecco un esempio che illustra l'uso di alcuni manipolatori:

```
#include <iostream>
#include <iomanip>
using namespace std;

int main()
{
  cout << hex << 100 << endl;

  cout << setfill('?') << setw(10) << 2343.0;

  return 0;
}
```

Questo programma produce il seguente output:

```
64
???????2343
```

Si noti la posizione che i manipolatori occupano nella catena delle operazioni di I/O. Inoltre si noti che quando un manipolatore non richiede argomenti, come ad esempio endl() non è seguito da parentesi. Questo avviene perché all'operatore << viene passato l'indirizzo della funzione.

Per confronto, ecco una versione equivalente del programma precedente che ottiene gli stessi risultati utilizzando le funzioni membro della classe ios:

```
#include <iostream>
#include <iomanip>
using namespace std;

int main()
{
  cout.setf(ios::hex, ios::basefield);
  cout << 100 << "\n";  // 100 in esadecimale

  cout.fill('?');
  cout.width(10);
  cout << 2343.0;

  return 0;
}
```

Come suggerisce l'esempio, il vantaggio principale dell'uso dei manipolatori al posto delle funzioni membro di ios è il fatto che spesso consentono di produrre codice più compatto.

Il manipolatore setiosflags() consente di impostare direttamente i vari flag di formattazione relativi a uno stream. Ad esempio, questo programma utilizza setiosflags() per attivare i flag showbase e showpos:

```
#include <iostream>
#include <iomanip>
using namespace std;

int main()
{
  cout << setiosflags(ios::showpos);
  cout << setiosflags(ios::showbase);
  cout << 123 << " " << hex << 123;
  return 0;
}
```



Il manipolatore setiosflags() esegue la stessa operazione della funzione membro setf().

Uno dei manipolatori più interessanti è boolalpha che consente di introdurre i valori logici come true e false invece che come valori numerici. Ad esempio:

```
#include <iostream>
using namespace std;

int main()

  bool b;

  b = true;
  cout << b << " " << boolalpha << b << endl;

  cout << "Inserire un valore booleano: ";
  cin >> boolalpha >> b;
  cout << "Questo è il valore immesso: " << b;

  return 0;
}
```

Ecco quale potrebbe essere l'output del programma.

```
1 true
Inserire un valore booleano: false
Questo è il valore immesso: false
```

## 20.5 Overloading di << e >>

Come si è detto in precedenza, gli operatori << e >> sono stati modificati tramite overloading in C++ per eseguire operazioni di I/O dei tipi standard. È possibile eseguire un ulteriore overloading di questi operatori per far loro eseguire operazioni di I/O sui nuovi tipi.

Seguendo la terminologia del C++, l'operatore di output << viene chiamato *operatore di inserimento* poiché inserisce i caratteri in uno stream. Analogamente, l'operatore di input >> è chiamato *operatore di estrazione* poiché estrae i caratteri da uno stream. Le funzioni che eseguono l'overloading degli operatori di inserimento ed estrazione sono generalmente chiamati *inseritori* ed *estraitori*.

## Creazione di un inseritore

È piuttosto facile creare un inseritore per una nuova classe. Tutte le funzioni inseritore hanno la seguente forma generale:

```
ostream &operator<<(ostream &stream, tipo_classe oggetto)
{
  // corpo dell'inseritore
  return stream;
}
```

Si noti che la funzione restituisce l'indirizzo di uno stream di tipo ostream (si ricordi che ostream è una classe derivata da ios che supporta l'output). Inoltre, il primo parametro della funzione è l'indirizzo dello stream di output. Il secondo parametro è l'oggetto da inserire (può anche essere l'indirizzo dell'oggetto da inserire). L'ultima operazione che un inseritore deve eseguire prima di terminare è la restituzione dello stream. In questo modo l'inseritore potrà essere utilizzato in una lunga espressione di I/O.

All'interno di una funzione inseritore, è possibile inserire qualsiasi tipo di procedura e operazione. Ovvero le operazioni svolte da un inseritore possono essere scelte completamente dal programmatore. Tuttavia, per conservare uno stile di programmazione corretto, si dovranno limitare le operazioni svolte da un inseritore all'output di informazioni su uno stream. Ad esempio, è sconsigliabile definire come effetto collaterale dell'operazione di inserimento il calcolo di pi con 30 cifre decimali.

Per osservare un esempio di inseritore personalizzato, si proverà a creare un inseritore per oggetti di tipo phonebook:

```
class phonebook {
public:
  char name[80];
  int areacode;
  int prefix;
  int num;
  phonebook(char *n, int a, int p, int nm)
  {
    strcpy(name, n);
    areacode = a;
    prefix = p;
    num = nm;
  }
};
```



Questa classe memorizza il nome e il numero di telefono di una persona. Ecco come è possibile creare una funzione inseritore per oggetti di tipo phonebook.

```
// Visualizza il nome e il numero telefonico
ostream &operator<<(ostream &stream, phonebook o)
{
  stream << o.name << " ";
  stream << "(" << o.areacode << ") ";
  stream << o.prefix << "-" << o.num << "\n";

  return stream; // deve restituire lo stream
}
```

Ecco un breve programma che illustra l'uso della funzione inseritore per phonebook.

```
#include <iostream>
#include <cstring>
using namespace std;

class phonebook {
public:
  char name[80];
  int areacode;
  int prefix;
  int num;
  phonebook(char *n, int a, int p, int nm)
{
    strcpy(name, n);
    areacode = a;
    prefix = p;
    num = nm;
  }
};

// Visualizza il nome e il numero telefonico.
ostream &operator<<(ostream &stream, phonebook o)
{
  stream << o.name << " ";
  stream << "(" << o.areacode << ") ";
  stream << o.prefix << "-" << o.num << "\n";

  return stream; // deve restituire lo stream
}
```

```
int main()
{
  phonebook a("Teo", 111, 555, 1234);
  phonebook b("Alice", 312, 555, 5768);
  phonebook c("Tommaso", 212, 555, 9991);

  cout << a << b << c;

  return 0;
}
```

Il programma produce il seguente output:

```
Teo (111) 555-1234
Alice (312) 555-5768
Tommaso (212) 555-9991
```

Nel programma precedente, si noti che l'inseritore di phonebook non è un membro di phonebook. Anche se ciò potrebbe sembrare a prima vista errato, il motivo è semplice. Quando una funzione operatore di qualsiasi tipo è un membro di una classe, l'operando sinistro (passato implicitamente mediante this) è l'oggetto che genera la chiamata alla funzione operatore. Inoltre, questo oggetto è un *oggetto della classe* di cui la funzione operatore è un membro. Non vi è alcun modo di cambiare ciò. Se una funzione per l'overloading di un operatore è un membro di una classe, l'operando di sinistra deve essere un oggetto di tale classe. Ma quando si esegue l'overloading degli inseritori, l'operando di sinistra è uno stream e l'operando di destra è un oggetto della classe.

Pertanto, le funzioni che eseguono l'overloading degli operatori di inserimento non possono essere membri della classe su cui operano. Nel programma precedente le variabili name, areacode, prefix e num sono pubbliche perché devono essere accessibili da parte degli inseritori.

Il fatto che gli inseritori non possano essere membri della classe per la quale sono definiti sembra essere una grave svista del C++. Dato che gli inseritori non sono membri, come possono accedere agli elementi privati di una classe? Nel programma precedente, tutti i membri sono stati resi pubblici. Tuttavia, l'incapsulamento è un componente essenziale della programmazione a oggetti. Richiedere che tutti i dati che devono essere prodotti in output debbano essere resi pubblici è in conflitto con questo principio. Fortunatamente, vi è una soluzione a questo problema: rendere l'inseritore friend della classe. Questo consente di lasciare che uno stream sia il primo argomento dell'inseritore e nel contempo concede alla funzione l'accesso ai membri privati della classe su cui opera. Ecco l'aspetto del programma modificato tramutando l'inseritore in una funzione friend:



```
#include <iostream>
#include <cstring>
using namespace std;

class phonebook {
  // ora è privata
  char name[80];
  int areacode;
  int prefix;
  int num;
public:
  phonebook(char *n, int a, int p, int nm)
  {
    strcpy(name, n);
    areacode = a;
    prefix = p;
    num = nm;
  }
  friend ostream &operator<<(ostream &stream, phonebook o);
};

// Visualizza il nome e il numero telefonico.
ostream &operator<<(ostream &stream, phonebook o)
{
  stream << o.name << " ";
  stream << "(" << o.areacode << ") ";
  stream << o.prefix << "-" << o.num << "\n";

  return stream; // deve restituire lo stream
}

int main()
{
  phonebook a("Teo", 111, 555, 1234);
  phonebook b("Alice", 312, 555, 5768);
  phonebook c("Tommaso", 212, 555, 9991);

  cout << a << b << c;

  return 0;
}
```

Quando si definisce il corpo di una funzione inseritore, si deve ricordare di mantenerlo il più possibile generale. Ad esempio, l'inseritore mostrato nell'esempio precedente può essere utilizzato con qualsiasi stream poiché il corpo della funzio-

ne invia l'output su stream ovvero lo stream che ha richiamato l'inseritore. Anche se non sarebbe tecnicamente errato scrivere la riga:

```
stream << o.name << " ";
```

come

```
cout << o.name << " ";
```

questo avrebbe l'effetto di costringere a specificare cout come stream di output. La versione originale funzionerà con qualsiasi stream inclusi quelli connessi a file su disco. Anche se in determinate situazioni, specialmente quando si opera su dispositivi di output particolari, si preferirà specificare precisamente il nome dello stream di output, nella maggior parte dei casi questo è sconsigliabile. In generale, più flessibili saranno gli inseritori e maggiore sarà il loro valore.

**NOTA** *L'inseritore della classe phonebook funziona correttamente tranne quando il valore di num è qualcosa come 0034, nel qual caso gli zeri iniziali non vengono visualizzati. Per correggere questo errore, si può convertire num in una stringa o impostare il carattere di riempimento a "0" e utilizzare la funzione di formattazione width() per generare gli zeri iniziali. La soluzione è lasciata al lettore per esercizio.*

Prima di esaminare gli estrattori, è bene vedere un altro esempio di funzione inseritore. Un inseritore non deve necessariamente gestire solo testo. Un inseritore può essere utilizzato per produrre in output dati in qualsiasi forma. Ad esempio, un inseritore per una classe che faccia parte di un sistema CAD potrebbe produrre in output le istruzioni per il plotter. Un altro inseritore potrebbe generare immagini grafiche. Per provare a generare in output oggetti diversi dal comune testo, si esamini il seguente programma che disegna riquadri sullo schermo (poiché il C++ non definisce una libreria di operazioni grafiche, il programma traccia un riquadro utilizzando caratteri ma si è liberi di utilizzare elementi grafici se il sistema ne supporta l'uso).

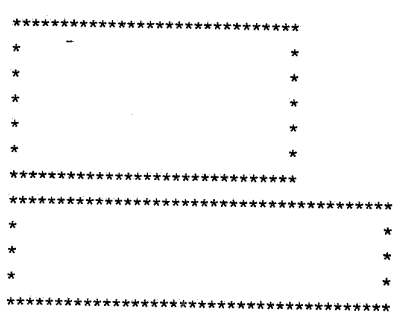
```
#include <iostream>
using namespace std;

class box {
  int x, y;
public:
  box(int i, int j) {x=i; y=j;}
  friend ostream &operator<<(ostream &stream, box o);
};
```



```
// Genera un riquadro.
ostream &operator<<(ostream &stream, box o)
{
  register int i, j;

  for(i=0; i<o.x; i++)
    stream << "*";

  stream << "\n";

  for(j=1; j<o.y-1; j++) {
    for(i=0; i<o.x; i++)
      if(i==0 || i==o.x-1) stream << "*";
      else stream << " ";
    stream << "\n";
  }

  for(i=0; i<o.x; i++)
     stream << "*";
  stream << "\n";

  return stream;
}

int main()
{
  box a(14, 6), b(30, 7), c(40, 5);

  cout << "Ecco alcuni riquadri:\n";
  cout << a << b << c;

  return 0;
}
```

Il programma visualizza i seguenti riquadri:

```
Ecco alcuni riquadri:
**************
*            *
*            *
*            *
*            *
**************
```



## Creazione di un estrattore

Gli estrattori sono complementari rispetto agli inseritori. La forma generale di una funzione estrattore è:

```
istream &operator>>(istream &stream, tipo_classe &oggetto)
{
  // corpo dell'estrattore
  return stream;
}
```

Gli estrattori restituiscono l'indirizzo di uno stream di tipo istream che corrisponde a uno stream di input. Anche il primo parametro deve essere l'indirizzo di uno stream di tipo istream. Si noti che il secondo parametro deve essere l'indirizzo di un oggetto della classe per la quale è definito l'estrattore. Questo consente all'operatore di input (estrattore) di modificare l'oggetto.

Proseguendo nell'esempio della classe phonebook, ecco come è possibile scrivere una funzione estrattore:

```
istream &operator>>(istream &stream, phonebook &o)
{
  cout << "Indicare il nome: ";
  stream >> o.name;
  cout << "Indicare il codice: ";
  stream >> o.areacode;
  cout << "Indicare il prefisso: ";
  stream >> o.prefix;
  cout << "Indicare il numero: ";
  stream >> o.num;
  cout << "\n";

  return stream;
}
```



Si noti che sebbene questa sia una funzione di input, produce l'output necessario per chiedere informazioni all'utente. Anche se lo scopo principale di un estrattore è l'input, questo può essere utilizzato per eseguire qualsiasi operazione necessaria per ottenere tale scopo. Tuttavia, come nel caso degli inseritori, è consigliabile mantenere le azioni eseguite da un estrattore direttamente correlate con l'input. In caso contrario si corre il rischio di perdere molto in termini di struttura e chiarezza.

Ecco un programma che illustra l'uso dell'estrattore di phonebook:

```
#include <iostream>
#include <cstring>
using namespace std;

class phonebook {
  char name[80];
  int areacode;
  int prefix;
  int num;
public:
  phonebook() { };
  phonebook(char *n, int a, int p, int nm)
  {
    strcpy(name, n);
    areacode = a;
    prefix = p;
    num = nm;
  }
  friend ostream &operator<<(ostream &stream, phonebook o);
  friend istream &operator>>(istream &stream, phonebook &o);
};

// Visualizza il nome e il numero telefonico.
ostream &operator<<(ostream &stream, phonebook o)
{
  stream << o.name << " ";
  stream << "(" << o.areacode << ") ";
  stream << o.prefix << "-" << o.num << "\n";

  return stream; // deve restituire lo stream
}
```

```
// Input del nome e del numero telefonico.
istream &operator>>(istream &stream, phonebook &o)
{
  cout << "Indicare il nome: ";
  stream >> o.name;
  cout << "Indicare il codice: ";
  stream >> o.areacode;
  cout << "Indicare il prefisso: ";
  stream >> o.prefix;
  cout << "Indicare il numero: ";
  stream >> o.num;
  cout << "\n";

  return stream;
}

int main()
{
  phonebook a;

  cin >> a;

  cout << a;

  return 0;
}
```

In realtà l'estrattore di phonebook non è perfetto poiché le istruzioni cout sono necessarie solo se lo stream di input è connesso a un dispositivo interattivo come la console (ovvero quando lo stream di input è cin). Se l'estrattore viene utilizzato su uno stream connesso a un file su disco, allora le istruzioni cout non risulteranno applicabili. Per divertimento si può provare a sopprimere le istruzioni cout tranne quando lo stream di input fa riferimento a cin. Ad esempio si può usare un'istruzione if come la seguente:

```
if(stream == cin) cout << "Inserire il nome" ";
```

Ora il messaggio verrà visualizzato solo se il dispositivo di output è lo schermo.

## 20.6 Creazione di funzioni di manipolazione

Oltre all'overloading degli operatori di inserimento ed estrazione, è possibile personalizzare ulteriormente il sistema di I/O C++ creando proprie funzioni manipolatori. I manipolatori personalizzati sono importanti per due motivi. Innanzi tutto, è possibile raggruppare una sequenza di operazioni di I/O in un unico manipolatore. Ad esempio, non è difficile trovare situazioni in cui la stessa sequenza di operazioni di I/O sia richiesta frequentemente in un programma. In questi casi si può utilizzare un manipolatore personalizzato, semplificando il codice sorgente ed evitando possibili errori. Un manipolatore personalizzato può anche essere utile quando si devono eseguire operazioni di I/O su un dispositivo non standard. Ad esempio, si potrebbe utilizzare un manipolatore per inviare codici di controllo a un particolare tipo di stampante oppure a un sistema di riconoscimento ottico.



I manipolatori personalizzati sono una delle funzionalità del C++ che supporta la programmazione a oggetti ma ne possono trarre beneficio anche programmi non a oggetti. Come si vedrà, i manipolatori personalizzati possono rendere più chiari ed efficienti i programmi che eseguono notevoli operazioni di I/O.

Vi sono due tipi principali di manipolatori: quelli che operano su stream di input e quelli che operano su stream di output. Oltre a queste due grandi categorie, vi è un'ulteriore suddivisione: manipolatori che richiedono un argomento e che non ne richiedono alcuno. Le procedure necessarie per creare un manipolatore parametrizzato variano da compilatore a compilatore e anche da una versione a un'altra dello stesso compilatore. Per questo motivo, per quanto riguarda la creazione di un manipolatore parametrizzato si rimanda alla documentazione del compilatore. Al contrario la creazione di manipolatori senza parametri è più semplice e rimane costante in tutti i compilatori. Questo sarà l'argomento della parte finale del capitolo.

Tutte le funzioni che realizzano manipolatori senza parametri per operazioni di output hanno la seguente struttura:

```
ostream &nome_manipolatore(ostream &stream)
{
  // codice
  return stream;
}
```

Si noti che il manipolatore restituisce l'indirizzo di uno stream di tipo ostream. Questo è necessario nel caso in cui il manipolatore venga utilizzato all'interno di un'espressione di I/O più estesa. È importante notare che anche se il manipolatore ha come unico argomento un indirizzo allo stream su cui opera, in realtà non viene utilizzato alcun argomento quando il manipolatore viene inserito in un'operazione di output.

Il primo semplice esempio crea un manipolatore chiamato sethex() che attiva il flag showbase e imposta l'output esadecimale.

```
#include <iostream>
#include <iomanip>
```

```
using namespace std;

// Un semplice manipolatore di output.
ostream &sethex(ostream &stream)
{
  stream.setf(ios::showbase);
  stream.setf(ios::hex, ios::basefield);

  return stream;
}

int main()
{
  cout << 256 << " " << sethex << 256;

  return 0;
}
```

Questo programma visualizza 256 0x100. Come si vede sethex è utilizzata all'interno di un'espressione di I/O così come avviene per i manipolatori standard.

I manipolatori personalizzati non devono necessariamente essere complessi per essere utili. Ad esempio, i semplici manipolatori la() e ra() visualizzano rispettivamente una freccia a sinistra e una freccia a destra:

```
#include <iostream>
#include <iomanip>
using namespace std;

// Freccia a destra
ostream &ra(ostream &stream)
{
  stream << "------> ";
  return stream;
}

// Freccia a sinistra
ostream &la(ostream &stream)
{
  stream << " <------";
  return stream;
}

int main()
{
  cout << "Valore massimo " << ra << 1233.23 << "\n";
```



```
  cout << "Valore medio " << ra << 567.66 << la;

  return 0;
}
```

Questo programma visualizza il seguente output:

```
Valore massimo ------> 1233.23
Valore medio ------> 567.66 <------
```

Se devono essere utilizzati con frequenza, questi semplici manipolatori consentono di risparmiare tempo e fatica.

L'utilizzo di un manipolatore di output è particolarmente utile per inviare codici a un dispositivo. Ad esempio, una stampante potrebbe accettare vari codici che cambiano le dimensioni o il font dei caratteri o che posizionano la testina di stampa in una particolare posizione. Se queste regolazioni devono essere eseguite frequentemente, si tratta di candidati perfetti per un manipolatore.

Tutte le funzioni manipolatori di input senza parametri hanno la seguente struttura:

```
istream &nome_manipolatore(istream &stream)
{
  // codice
  return stream;
}
```

Un manipolatore di input riceve l'indirizzo dello stream per il quale è stato richiamato. Questo stream dovrà poi essere restituito dal manipolatore.

Il seguente programma crea il manipolatore di input getpass() che emette un segnale acustico e chiede una password.

```
#include <iostream>
#include <cstring>
using namespace std;

// Un semplice manipolatore di input.
istream &getpass(istream &stream)
{
  cout << '\a';  // segnale acustico
  cout << "Immettere la password: ";

  return stream;
}
```

```
int main()
{
  char pw[80];

  do {
    cin >> getpass >> pw;
  } while (strcmp(pw, "password"));

  cout << "Login completato\n";

  return 0;
}
```

È importante che il manipolatore restituisca lo stream. In caso contrario, non potrebbe essere utilizzato in una serie di operazioni di input o di output.

Capitolo 21

# Operazioni di I/O su file in C++


21.1 **L'header <fstream> e le classi per i file**
21.2 **L'apertura e la chiusura di un file**
21.3 **La lettura e la scrittura di un file di testo**
21.4 **Le operazioni di I/O binarie e non formattate**
21.5 **Altre forme della funzione get()**
21.6 **La funzione getline()**
21.7 **Rilevamento della fine del file**
21.8 **La funzione ignore()**
21.9 **Le funzioni peek() e putback()**
21.10 **La funzione flush()**
21.11 **L'accesso diretto ai file**
21.12 **Lo stato delle operazioni di I/O**
21.13 **Personalizzazione delle operazioni di I/O sui file**


Anche se l'approccio C++ alle operazioni di I/O costituisce un sistema integrato, l'I/O su file è sufficientemente specializzato da essere in genere considerato come un caso speciale, soggetto a propri limiti e metodi. In parte, questo è dovuto al fatto che, quando si parla di file, si intende in genere file residenti su disco e i file su disco sono dotati di funzionalità e caratteristiche specifiche che non si riscontrano per altri dispositivi. Si deve però ricordare che l'I/O su disco è un caso speciale del sistema generale di I/O e che la maggior parte degli aspetti discussi in questo capitolo si applicano anche ad altri stream connessi ad altri tipi di dispositivi.

## 21.1 L'header <fstream> e le classi per i file

Per eseguire operazioni di I/O su file, si deve includere nel programma l'header <fstream>. Tale header definisce numerose classi, fra le quali, ifstream, ofstream e fstream. Queste classi derivano rispettivamente da istream, ostream e iostream.

Come si ricorderà, istream, ostream e iostream derivano dalla classe ios e quindi ifstream, ofstream e fstream. hanno accesso anche a tutte le operazioni definite da ios (discusse nel precedente capitolo). Un'altra classe utilizzata per le operazioni di I/O su file è filebuf che fornisce funzionalità di I/O di basso livello su stream. Normalmente si usa filebuf indirettamente attraverso le altre classi.

## 21.2 L'apertura e la chiusura di un file

In C++ si apre un file collegandolo a uno stream. Prima di poter aprire un file si dovrà pertanto aprire uno stream. Vi sono tre tipi di stream: stream di input, stream di output e stream di input/output. Per creare uno stream di input, si deve dichiarare uno stream della classe ifstream. Per creare uno stream di output lo si deve dichiarare della classe ostream. Gli stream che dovranno eseguire sia operazioni di input che operazioni di output devono essere dichiarati della classe fstream. Ad esempio, questo frammento di codice crea uno stream di input, uno stream di output e uno stream in grado di eseguire operazioni sia di input che di output.

```
ifstream in;  // input
ofstream out; // output
fstream io;   // input e output
```

Dopo aver creato uno stream, un modo per associarlo a un file consiste nell'uso della funzione open(). Questa funzione è un membro di ognuna delle tre classi di stream. Il suo prototipo è:

void ifstream::open(const char **nomefile*, ios::openmode *modalità* ios::in);
void ofstream::open(const char **nomefile*, ios::openmode *modalità* ios::out | ios::trunc);
void fstream::open(const char **nomefile*, ios::openmode *modalità* ios::in | ios::out);

Qui, *nomefile* è il nome del file, il quale può includere uno specificatore di percorso. Il valore di *modalità* determina il modo in cui deve essere aperto il file. *modalità* deve essere uguale a uno (o più) di questi valori definiti da openmode che è un'enumerazione definita da ios attraverso la classe base ios_base.

ios::app
ios::ate
ios::binary
ios::in
ios::out
ios::trunc

Per combinare due o più di questi valori si utilizza l'operatore OR.

La modalità ios::app fa in modo che tutto l'output venga aggiunto (append) alla fine del file. Questo valore può essere utilizzato solo per file che accettano operazioni di output. L'inclusione di ios::ate posiziona il puntatore alla fine del file aperto. Nonostante questo, le operazioni di I/O possono avvenire anche in qualsiasi altro punto del file.

Il valore ios::in specifica che il file è in grado di fornire dati per operazioni di input. Il valore ios::out specifica che il file è in grado di accettare dati in output.

Il valore ios::binary provoca l'apertura di un file in modalità binaria (normalmente vengono aperti in modalità binaria). Quando un file viene aperto in modalità testo, possono avvenire varie traduzioni dei caratteri, ad esempio la conversione di sequenze Carriage Return/Line Feed in codici di fine riga. Quando invece un file viene aperto in modalità binaria, non viene eseguita alcuna traduzione. Qualsiasi file, che contenga testo formattato o semplici dati binari, può essere aperto sia in modalità testo che in modalità binaria. L'unica differenza consiste nella traduzione eseguita sui caratteri.

Il valore ios::trunc provoca la distruzione del contenuto di un file precedente avente lo stesso nome e tronca la lunghezza del file a zero. Quando si crea uno stream di output con ofstream, un eventuale file preesistente verrà troncato a zero.

Il seguente frammento di codice apre in output un comune file.

```
ofstream out;

out.open("test", ios::out);
```

Ma raramente si vedrà una chiamata a open() di questo tipo, poiché il parametro *modalità* fornisce dei valori standard per ciascun tipo di stream. Per ifstream il valore standard di *modalità* è ios::in, per ofstream è ios::out | ios::trunc e per fstream è ios::out | ios::out. Pertanto, l'istruzione precedente avrà normalmente il seguente aspetto:

```
out.open("test"); // impostazione standard: normale file di output
```

**NOTA** *A seconda del compilatore impiegato, il parametro modalità per fstream::open() potrebbe non avere l'impostazione standard in | out. Dunque talvolta è necessario specificarli in modo esplicito.*

Se open() non ha successo e si valuta mystream in un'espressione booleana il suo valore sarà uguale a false. Pertanto, prima di utilizzare un file, si deve control-

lare che l'operazione di apertura abbia avuto successo. A tale scopo si può impiegare un'istruzione simile alla seguente:

```
if(!mystream) {
  cout << "Impossibile aprire il file.\n";
  // gestione dell'errore
}
```

Anche se è perfettamente corretto aprire un file utilizzando la funzione open(), la maggior parte delle volte questo non sarà necessario poiché le classi ifstream, ofstream e fstream prevedono funzioni costruttore che aprono automaticamente il file. Le funzioni costruttore hanno gli stessi parametri e gli stessi valori standard della funzione open(). Pertanto, è molto più comune vedere un file aperto nel seguente modo:

```
ifstream mystream("nomefile"); // apertura del file in input
```

Come si è detto, se per qualche motivo il file non potesse essere aperto, il valore della variabile associata allo stream sarà false. Pertanto, che si apra il file utilizzando una funzione costruttore o una chiamata esplicita a open(), si dovrà sempre controllare che il file sia stato effettivamente aperto andando a leggere il valore dello stream.

Per controllare se il file è stato aperto con successo si usa la funzione is_open() che è un membro di ifstream, ofstream e fstream. Tale funzione ha il seguente prototipo:

```
bool is_open();
```

La funzione restituisce true se lo stream è connesso a un file aperto e false in tutti gli altri casi. Ad esempio, il seguente frammento di codice controlla se mystream è aperto:

```
if( !mystream.is_open()) {
  cout << "File non aperto.\n";
  //...
```

Per chiudere un file si utilizza la funzione membro close(). Ad esempio, per chiudere il file collegato a uno stream chiamato mystream si utilizza la seguente istruzione:

```
mystream.close();
```

La funzione close() non richiede alcun parametro e non restituisce alcun valore.



## 21.3 La lettura e la scrittura di un file di testo

È molto facile eseguire operazioni di lettura o scrittura su un file di testo. Si devono semplicemente utilizzare gli operatori << e >> nello stesso modo visto per le operazioni di I/O su console, tranne per il fatto che invece di utilizzare cin e cout, si deve specificare uno stream precedentemente collegato a un file. Ad esempio, questo programma crea un breve file che contiene il nome e il prezzo di alcuni oggetti.

```
#include <iostream>
#include <fstream>
using namespace std;

int main()
{
  ofstream out("INVNTRY"); // output, comune file

  if(!out) {
    cout << "Impossibile aprire il file.\n";
    return 1;
  }

  out << "Radio " << 39.95 << endl;
  out << "Tostapane " << 19.95 << endl;
  out << "Mixer " << 24.80 << endl;

  out.close();
  return 0;
}
```

Il seguente programma legge il file creato dal programma precedente e ne visualizza il contenuto sullo schermo:

```
#include <iostream>
#include <fstream>
using namespace std;

int main()
{
  ifstream in("INVNTRY"); // input

  if(!in) {
    cout << "Impossibile aprire il file.\n";
    return 1;
  }
```

```
  char item[20];
  float cost;

  in >> item >> cost;
  cout << item << " " << cost << "\n";
  in >> item >> cost;
  cout << item << " " << cost << "\n";
  in >> item >> cost;
  cout << item << " " << cost << "\n";

  in.close();
  return 0;
}
```

In un certo senso, la lettura e la scrittura di file utilizzando gli operatori >> e << corrisponde all'uso delle funzioni C fprintf() e fscanf(). Le informazioni memorizzate nel file sono infatti nello stesso formato in cui vengono visualizzate sullo schermo.

Quello che segue è un altro esempio che mostra operazioni di I/O su disco. Questo programma legge una serie di stringhe immesse alla tastiera e le scrive su disco. Il programma termina quando l'utente immette un punto esclamativo. Per utilizzare il programma, si deve specificare sulla riga di comando il nome del file di output.

```
#include <iostream>
#include <fstream>
using namespace std;

int main(int argc, char *argv[])
{
  if(argc!=2) {
    cout << "Uso: output <nomefile>\n";
    return 1;
  }

  ofstream out(argv[1]); // output, comune file

  if(!out) {
    cout << "Impossibile aprire il file di output.\n";
    return 1;
  }

  char str[80];
  cout << "Scrittura di stringhe su disco. Digitare ! per concludere.\n";
```



```
  do {
    cout << ": ";
    gets(str);
    out << str << endl;
  } while (*str != '!');

  out.close();
  return 0;
}
```

Quando si legge un file di testo utilizzando l'operatore >> si deve ricordare che avverranno alcune traduzioni di caratteri. Ad esempio, verranno omessi i caratteri corrispondenti a spazi vuoti. Per evitare la traduzione dei caratteri, si deve aprire il file in modalità binaria e utilizzare le funzioni discusse nella prossima sezione.

Durante l'input, quando viene incontrato il codice di fine file, lo stream connesso con il file sarà uguale a false (questo fatto verrà discusso nella prossima sezione).

## 21.4 Le operazioni di I/O binarie e non formattate

Anche se leggere e scrivere file di testo formattati è molto semplice, non sempre questo è il modo più efficiente per gestire i file. Inoltre spesso capita di dover memorizzare dati binari non formattati e non testo. Questa sezione descrive le funzioni da impiegare.

Quando si eseguono operazioni binarie su un file, occorre assicurarsi di aprire il file utilizzando lo specificatore di modalità ios::binary. Anche se le funzioni non formattate possono operare anche su file aperti in modalità testo, in questo caso possono verificarsi delle traduzioni dei caratteri. Le traduzioni dei caratteri negano lo scopo delle operazioni su file binari.

### Caratteri e byte

Prima di iniziare l'esame delle operazioni di I/O non formattato, è importante chiarire un concetto importante. Per molti anni, le operazioni di I/O in C e C++ sono state "orientate ai byte". Questo è dovuto al fatto che un char è equivalente a un byte ed erano disponibili solo stream di char. Ma con l'avvento dei caratteri estesi (di tipo w_char_t) e i relativi stream, non si può più dire che le operazioni di I/O del linguaggio C++ siamo orientate ai caratteri. Naturalmente gli stream di char continuano a essere stream di byte e si può continuare a pensare in termini di

byte, specialmente quando si opera su dati differenti dal testo. Ma l'equivalenza fra un byte e un carattere non è più garantita.

Come si è detto nel Capitolo 20, tutti gli stream utilizzati in questo volume sono stream di char (che sono di gran lunga i più comuni). Essi semplificano anche la gestione non formattata dei file poiché uno stream di char stabilisce una corrispondenza uno-a-uno fra i byte e di caratteri a tutto vantaggio della lettura o scrittura di blocchi di dati binari.

## Le funzioni get() e put()

Per leggere e scrivere dati non formattati si possono utilizzare le funzioni membro get() e put(). Si tratta di funzioni che operano su caratteri, ovvero get() legge un carattere e put() scrive un carattere. naturalmente, se si è aperto un file in modalità binaria e si opera su uno stream di char (e non di wchar_t) queste funzioni leggono e scrivono byte di dati. La funzione get() può assumere diverse forme delle quali la più comune viene mostrata di seguito insieme alla corrispondente put().

istream &get(char *&carattere*);
ostream &put(char *carattere*);

La funzione get() legge un singolo carattere dallo stream chiamante e ne inserisce il valore in *carattere*. La funzione restituisce l'indirizzo dello stream. La funzione put() scrive *carattere* sullo stream e restituisce l'indirizzo dello stream.

Il seguente programma visualizza con la funzione get() il contenuto di un file di testo o binario.

```
#include <iostream>
#include <fstream>
using namespace std;

int main(int argc, char *argv[])
{
  char ch;

  if(argc!=2) {
    cout << "Uso: PR <nomefile>\n";
    return 1;
  }

  ifstream in(argv[1], ios::in | ios::binary);
  if(!in) {
```



```
    cout << "Impossibile aprire il file.";
    return 1;
  }

  while(in) { // in è uguale a 0 quando viene raggiunta la fine del file
    in.get(ch);
    if(in) cout << ch;
  }

  return 0;
}
```

Come si è detto nella sezione precedente, quando si raggiunge la fine del file, lo stream associato al file diviene uguale a false. Pertanto, quando viene raggiunta la fine del file, in diverrà uguale a false provocando l'uscita dal ciclo while.

Vi è anche un modo più compatto per codificare il ciclo che legge e visualizza il file:

```
while(in.get(ch))
  cout << ch;
```

Questo frammento di codice funziona poiché get() restituisce l'indirizzo dello stream in e alla fine del file in sarà uguale a false.

Il programma successivo utilizza put() per scrivere sul file CHARS tutti i caratteri compresi fra 0 e 255. Tutti sanno che i caratteri ASCII occupano solo la metà dei valori che è possibile inserire in un carattere di tipo char. Gli altri valori vengono normalmente chiamati *caratteri estesi* e includono simboli matematici e lettere accentate oppure specifiche di determinate nazioni (non tutti i sistemi prevedono l'uso di caratteri estesi).

```
#include <iostream>
#include <fstream>
using namespace std;

int main()
{
  int i;
  ofstream out("CHARS", ios::out | ios::binary);

  if(!out) {
    cout << "Impossibile aprire il file di output.\n";
    return 1;
  }
```

```
  // scrive su disco tutti i caratteri
  for(i=0; i<256; i++) out.put((char) i);

  out.close();
  return 0;
}
```

Può essere interessante osservare il contenuto di CHARS per conoscere i caratteri estesi disponibili sul proprio sistema.

## Le funzioni read() e write()

Un altro metodo utilizzabile per leggere e scrivere blocchi di dati binari prevede l'uso delle funzioni read() e write(). I rispettivi prototipi sono:

istream &read(char **buf*, streamsize *num*);
ostream &write(const char **buf*, streamsize *num*);

La funzione read() legge *num* caratteri dallo stream associato e li inserisce nel buffer puntato da *buf*. La funzione write() scrive *num* caratteri sullo stream associato prelevandoli dal buffer puntato da *buf*. Come si è detto in precedenza, streamsize è definito nella libreria C++ come una forma di intero.

Il seguente programma scrive su disco e poi rilegge una struttura.

```
#include <iostream>
#include <fstream>
#include <cstring>
using namespace std;

struct status {
  char name[80];
  float balance;
  unsigned long account_num;
};

int main()
{
  struct status acc;

  strcpy(acc.name, "Rodolfo Trentini");
  acc.balance = 1123.23;
  acc.account_num = 34235678;
```



```
  ofstream outbal("balance", ios::out | ios::binary);

  if(!outbal) {
    cout << "Impossibile aprire il file.\n";
    return 1;
  }

  outbal.write((unsigned char *) &acc, sizeof(struct status));
  outbal.close();

  // ora rilegge il file;

  ifstream inbal("balance", ios::in | ios::binary);

  if(!inbal) {
    cout << "Impossibile aprire il file.\n";
    return 1;
  }

  inbal.read((unsigned char *) &acc, sizeof(struct status));

  cout << acc.name << endl;
  cout << "Numero di conto: " << acc.account_num;
  cout.precision(2);
  cout.setf(ios::fixed);
  cout << endl << "Saldo: " << acc.balance;

  inbal.close();
  return 0;
}
```

Come si può vedere, per leggere o scrivere l'intera struttura basta una singola chiamata a read() o write(). Non è quindi necessario scrivere o leggere separatamente ogni singolo campo della struttura. Come si può vedere da questo esempio, il buffer può essere costituito da qualsiasi tipo di oggetto.

**NOTA** *Le conversioni di tipo presenti all'interno delle chiamata a read() e write() sono necessarie quando si opera su un buffer che non sia definito come un array di caratteri. A causa della stretta verifica di tipo del C++, un puntatore di un tipo non verrà automaticamente convertito in un puntatore di un altro tipo.*

Se viene raggiunta la fine del file prima della lettura di *num* caratteri, read() si fermerà e il buffer conterrà i soli caratteri disponibili. Per conoscere il numero di caratteri letti, si può utilizzare la funzione membro gcount() il cui prototipo è:

streamsize gcount( );

Tale funzione restituisce il numero di caratteri letti dall'ultima operazione di input binario. Il seguente programma mostra un altro esempio di utilizzo di read() e write() e illustra l'uso di gcount().

```
#include <iostream>
#include <fstream>
using namespace std;

int main()
{
  float fnum[4] = {99.75, -34.4, 1776.0, 200.1};
  int i;

  ofstream out("numeri", ios::out | ios::binary);
  if(!out) {
    cout << "Impossibile aprire il file.";
    return 1;
  }

  out.write((char *) &fnum, sizeof fnum);

  out.close();

  for(i=0; i<4; i++) // cancella l'array
    fnum[i] = 0.0;

  ifstream in("numeri", ios::in | ios::binary);
  in.read((char *) &fnum, sizeof fnum);

  // controlla quanti byte sono stati letti
  cout << in.gcount() << " byte letti\n";

  for(i=0; i<4; i++) // mostra i valori letti dal file
  cout << fnum[i] << " ";

  in.close();

  return 0;
}
```

Questo programma scrive su disco e quindi rilegge un array di valori in virgola mobile. Dopo la chiamata a read(), per determinare il numero di byte letti viene utilizzata gcount().



## 21.5 Altre forme della funzione get()

Oltre alla forma mostrata precedentemente, la funzione get() viene modificata tramite overloading in vari modi. Di seguito vengono illustrati i prototipi delle tre versioni più utilizzate:

```
istream &get(char *buf, streamsize num);
istream &get(char *buf, streamsize num, char delim);
int get( );
```

La prima forma legge caratteri inserendoli nell'array puntato da *buf* fermandosi dopo aver letto *num*-1 caratteri oppure finché non viene trovato il codice di fine riga o di fine file. Il codice nullo che conclude l'array puntato da *buf* verrà aggiunto automaticamente da get(). Se nello stream di input viene trovato il carattere di fine riga, questo *non* verrà estratto. Rimarrà nello stream fino all'operazione di input successiva.

La seconda forma legge caratteri inserendoli nell'array puntato da *buf* fermandosi dopo aver letto *num*-1 caratteri oppure finché non viene trovato il delimitatore, il codice di fine riga o il codice di fine file. Il codice nullo che conclude l'array puntato da *buf* verrà aggiunto automaticamente da get(). Se nello stream di input viene trovato il carattere delimitatore, questo *non* verrà estratto. Rimarrà nello stream fino all'operazione di input successiva.

La terza versione di get() restituisce il prossimo carattere letto dallo stream. Alla fine del file la funzione restituisce EOF. Questa forma di get() è simile alla funzione getc() del C.

## 21.6 La funzione getline()

Un'altra funzione membro di ogni classe stream di input che esegue operazioni di input è geline(). Ecco i suoi prototipi:

istream &getline(char **buf*, streamsize *num*);
istream &getline(char **buf*, streamsize *num*, char *delim*);

La prima forma legge caratteri inserendoli nell'array puntato da *buf* fermandosi dopo aver letto *num*-1 caratteri oppure finché non viene trovato il codice di fine riga o di fine file. Il codice nullo che conclude l'array puntato da *buf* verrà aggiunto automaticamente da getline(). Se nello stream di input viene trovato il carattere di fine riga, questo verrà estratto ma non verrà inserito nel buffer.

La seconda forma legge caratteri inserendoli nell'array puntato da *buf* fermandosi dopo aver letto *num*-1 caratteri oppure finché non viene trovato il

delimitatore, il codice di fine riga o il codice di fine file. Il codice nullo che conclude l'array puntato da *buf* verrà aggiunto automaticamente da get(). Se nello stream di input viene trovato il carattere delimitatore, questo verrà estratto ma non verrà inserito nel buffer.

Come si può vedere, queste due versioni di getline() sono praticamente identiche a get(buf, num) e get(buf, num, delim). Entrambe le funzioni leggono caratteri dallo stream di input e li inseriscono nell'array puntato da *buf* fino ad aver letto *num* caratteri o fino al raggiungimento del primo carattere *delim*. La differenza fra get() e getline() è che quest'ultima legge ed elimina il delimitatore dallo stream di input.

Ecco un programma che mostra l'uso della funzione getline(). Il programma legge il contenuto di un file di testo una riga per volta e visualizza sullo schermo le righe lette.

```
// Legge e visualizza un file di testo riga per riga.

#include <iostream>
#include <fstream>
using namespace std;

int main(int argc, char *argv[])
{
  if(argc!=2) {
    cout << "Uso: Display <nomefile>\n";
    return 1;
  }
  ifstream in(argv[1]); // input

  if(!in) {
    cout << "Impossibile aprire il file di input.\n";
    return 1;
  }

  char str[255];

  while(in) {
    in.getline(str, 255);  // il delimitatore standard è '\n'
    if (in) cout << str << endl;
  }

  in.close();

  return 0;
}
```



## 21.7 Rilevamento della fine del file

Per determinare il momento in cui si raggiunge la fine del file, si utilizza la funzione membro eof() il cui prototipo è:

```
bool eof( );
```

Questa funzione restituisce true quando viene raggiunta la fine del file e false in caso contrario.

Il seguente programma utilizza eof() per visualizzare il contenuto di un file in esadecimale e in ASCII.

```
/* Visualizza il contenuto del file specificato
   sia in ASCII che in esadecimale. */

#include <iostream>
#include <fstream>
#include <cctype>
#include <iomanip>
using namespace std;

int main(int argc, char *argv[])
{
  if(argc!=2) {
    cout << "Uso: Display <nomefile>\n";
    return 1;
  }

  ifstream in(argv[1], ios::in | ios::binary);

  if(!in) {
    cout << "Impossibile aprire il file di input.\n";
    return 1;
  }

  register int i, j;
  int count = 0;
  char c[16];

  cout.setf(ios::uppercase);
  while(!in.eof()) {
    for(i=0; i<16 && !in.eof(); i++) {
      in.get(c[i]);
    }
```

```
    if(i<16) i--; // si libera di eof

    for(j=0; j<i; j++)
      cout << setw(3) << hex << (int) c[j];
    for(; j<16; j++) cout << "   ";

    cout << "\t";
    for(j=0; j<i; j++)
      if(isprint(c[j])) cout << c[j];
      else cout << ".";

    cout << endl;

    count++;
    if(count==16) {
      count = 0;
      cout << "Premere Invio per continuare: ";
      cin.get();
      cout << endl;
    }
  }

  in.close();

  return 0;
}
```

Il programma produrrà un output simile al seguente:

```
2F 2A 20 44 69 73 70 6C 61 79 20 63 6F 6E 74 65 /* Display conte
6E 74 73 20 6F 66 20 73 70 65 63 69 66 69 65 64 nts of specified
20 66 69 6C 65  D  A 20 20 20 69 6E 20 62 6F 74 ±file..   in bot
68 20 41 53 43 49 49 20 61 6E 64 20 69 6E 20 68 h ASCII and in h
65 78 2E  D  A 2A 2F  D  A 23 69 6E 63 6C 75 64 ex...*/..#includ
65 20 3C 69 6F 73 74 72 65 61 6D 2E 68 3E  D  A e <iostream.h>..
23 69 6E 63 6C 75 64 65 20 3C 66 73 74 72 65 61 #include <fstrea
6D 2E 68 3E  D  A 23 69 6E 63 6C 75 64 65 20 3C m.h>..#include <
63 74 79 70 65 2E 68 3E  D  A 23 69 6E 63 6C 75 ctype.h>..#inclu
64 65 20 3C 69 6F 6D 61 6E 69 70 2E 68 3E  D  A de <iomanip.h>..
23 69 6E 63 6C 75 64 65 20 3C 73 74 64 69 6F 2E #include <stdio.
68 3E  D  A  D  A 6D 61 69 6E 28 69 6E 74 20 61 h>....main(int a
72 67 63 2C 20 63 68 61 72 20 2A 61 72 67 76 5B rgc, char *argv[
5D 29  D  A 7B  D  A 20 20 69 66 28 61 72 67 63 ]).{..  if(argc
21 3D 32 29 20 7B  D  A 20 20 20 20 63 6F 75 74 !=2) {..    cout
20 3C 3C 20 22 55 73 61 67 65 3A 20 44 69 73 70 ±<< "Uso: Disp
Premere Invio per continuare:
```



## 21.8 La funzione ignore()

La funzione membro ignore() consente di leggere e ignorare i caratteri presenti nello stream di input. Il suo prototipo è:

istream &ignore(streamsize *num*=1, int_type *delim*=EOF);

La funzione legge e ignora *num* caratteri (il valore standard è 1) o un numero di caratteri inferiore nel caso in cui venga incontrato il carattere specificato da *delim* (normalmente è il codice EOF). Se viene incontrato il delimitatore, questo non verrà eliminato dallo stream di input. Qui int_type è definito come un intero.

Questo programma legge il file TEST ignorando tutti i caratteri prima di uno spazio oppure un totale di dieci caratteri. Quindi il programma visualizza la parte rimanente del file.

```
#include <iostream>
#include <fstream>
using namespace std;

int main()
{
  ifstream in("test");

  if(!in) {
    cout << "Impossibile aprire il file.\n";
    return 1;
  }

  /* Ignora fino a 10 caratteri o fino a trovare
     il primo spazio. */
  in.ignore(10, ' ');
  char c;
  while(in) {
    in.get(c);
    if(in) cout << c;
  }

  in.close();
  return 0;
}
```

## 21.9 Le funzioni peek() e putback()

La funzione peek() consente di conoscere il prossimo carattere presente nello stream di input senza eliminarlo dallo stream. Il suo prototipo è:

```
int_type peek( );
```

La funzione restituisce il prossimo carattere presente nello stream o EOF nel caso in cui sia stata raggiunta la fine del file. Qui int_type è definito come un intero.

La funzione putback() reinserisce nello stream di input l'ultimo carattere letto. Il suo prototipo è:

```
istream &putback(char c);
```

dove *c* è l'ultimo carattere letto.

## 21.10 La funzione flush()

Quando vengono eseguite operazioni di output, non sempre i dati vengono immediatamente scritti sul dispositivo fisico connesso allo stream. In genere le informazioni vengono conservate in un buffer interno fino al suo completo riempimento. Solo allora il contenuto del buffer verrà scritto fisicamente sul disco. È però possibile richiedere la scrittura fisica su disco anche prima che il buffer sia completamente pieno. Il prototipo della funzione flush() è:

```
ostream &flush( );
```

Le chiamate a flush() possono essere utili quando un programma deve essere impiegato in ambienti instabili (ad esempio in casi in cui possono verificarsi cadute di tensione).

**NOTA** *Lo svuotamento del buffer può essere provocato anche dalla chiusura di un file o dalla naturale fine del programma.*

## 21.11 L'accesso diretto ai file

Nel sistema di I/O del C++, le operazioni di accesso diretto prevedono l'uso delle funzioni seekg() e seekp(). Le loro forme più comuni sono:



```
istream &seekg(off_type offset, seekdir origine);
ostream &seekp(off_type offset, seekdir origine);
```

off_type è un tipo definito da ios in grado di contenere il valore più grande che *offset* può assumere. seekdir è un'enumerazione, anch'essa definita in ios, che determina il punto di riferimento.

Il sistema di I/O del C++ gestisce due puntatori associati a un file. Uno è il *puntatore di lettura* che specifica la posizione nel file in cui avverrà la prossima operazione di input. L'altro è il *puntatore di scrittura* che specifica la posizione in cui avverrà la prossima operazione di output. Ogni volta che si verifica un'operazione di input o di output, l'appropriato puntatore viene automaticamente aggiornato. L'uso delle funzioni seekg() e seekp() consente di accedere a un file in modo non sequenziale.

La funzione seekg() sposta il puntatore di lettura del file a *offset* caratteri rispetto all'*origine* specificata la quale può corrispondere a uno dei seguenti valori:

| | |
|---|---|
| ios::beg | Inizio del file |
| ios::cur | Posizione corrente |
| ios::end | Fine del file |

La funzione seekp() sposta il puntatore di scrittura del file a *offset* caratteri rispetto all'*origine* specificata che deve corrispondere a uno dei tre valori precedenti.

Generalmente, le operazioni di I/O ad accesso diretto dovrebbero essere eseguite solo su fare aperti per operazioni binarie. La traduzione dei caratteri che può verificarsi nel caso dei file di testo, potrebbe alterare le richieste di posizionamento rispetto all'effettivo contenuto del file.

Il seguente programma mostra l'uso della funzione seekp(). Il programma consente di modificare un determinato carattere di un file. Si deve specificare un nome di file nella riga di comando seguito dal numero di caratteri corrispondente al carattere da modificare e seguito infine dal nuovo carattere. Si noti che il file viene aperto in lettura e scrittura.

```
#include <iostream>
#include <fstream>
#include <cstdlib>
using namespace std;

int main(int argc, char *argv[])
{
  if(argc!=4) {
    cout << "Uso: CHANGE <nomefile> <carattere> <char>\n";
    return 1;
```

```
  }

  fstream out(argv[1], ios::in | ios::out | ios::binary);
  if(!out) {
    cout << "Impossibile aprire il file.";
    return 1;
  }

  out.seekp(atoi(argv[2]), ios::beg);

  out.put(*argv[3]);
  out.close();

  return 0;
}
```

Ad esempio, per cambiare il dodicesimo byte del file TEST sostituendogli una Z, si deve utilizzare la seguente riga di comando:

```
change test 12 Z
```

Il programma seguente utilizza la funzione seekg() per visualizzare il contenuto di un file a partire dalla posizione specificata nella riga di comando.

```
#include <iostream>
#include <fstream>
#include <cstdlib>
using namespace std;

int main(int argc, char *argv[])
{
  char ch;

  if(argc!=3) {
    cout << "Uso: SHOW <nomefile> <posizione iniziale>\n";
    return 1;
  }

  ifstream in(argv[1], ios::in | ios::binary);
  if(!in) {
    cout << "Impossibile aprire il file.";
    return 1;
  }
```



```
  in.seekg(atoi(argv[2]), ios::beg);

  while(in.get(ch))
    cout << ch;

  return 0;
}
```

Il seguente programma utilizza sia seekp() che seekg() per invertire i primi <num> caratteri di un file.

```
#include <iostream>
#include <fstream>
#include <cstdlib>
using namespace std;

int main(int argc, char *argv[])
{
  if(argc!=3) {
    cout << "Uso: Reverse <nomefile> <num>\n";
    return 1;
  }

  fstream inout(argv[1], ios::in | ios::out | ios::binary);

  if(!inout) {
    cout << "Impossibile aprire il file di input.\n";
    return 1;
  }

  long e, i, j;
  char c1, c2;
  e = atol(argv[2]);

  for(i=0, j=e; i<j; i++, j--) {
    inout.seekg(i, ios::beg);
    inout.get(c1);
    inout.seekg(j, ios::beg);
    inout.get(c2);

    inout.seekp(i, ios::beg);
    inout.put(c2);
    inout.seekp(j, ios::beg);
    inout.put(c1);
  }
```

```
  inout.close();
  return 0;
}
```

Per utilizzare il programma si deve specificare il nome del file da invertire, seguito dal numero di caratteri da invertire. Ad esempio, per invertire i primi 10 caratteri del file TEST si deve utilizzare il comando:

```
reverse test 10
```

Se il file contiene le seguenti parole:

```
Questa è una prova.
```

al termine dell'operazione conterrà la frase:

```
u è atseuQna prova.
```

### Come conoscere la posizione corrente nel file

Per determinare la posizione corrente all'interno del file si devono utilizzare le funzioni:

```
pos_type tellg( );
pos_type tellp( );
```

pos_type è un tipo definito da ios in grado di contenere il valore più esteso che le funzioni possono restituire. Il valore restituito da tellg() e tellp() può essere utilizzato come argomento per le seguenti forme di seekg() e seekp().

```
istream &seekg(pos_type pos);
ostream &seekp(pos_type pos);
```

Queste funzioni consentono di salvare la posizione corrente nel file, eseguire altre operazioni sul file e di riportarsi sulla posizione precedentemente salvata.



## 21.12 Lo stato delle operazioni di I/O

Il sistema di I/O del C++ conserva una serie di informazioni di stato relative al risultato di ogni operazione di I/O. Lo stato corrente del sistema di I/O è contenuto in un oggetto di tipo iostate che è un'enumerazione definita in ios che contiene i seguenti membri:

| Nome | Significato |
|---|---|
| ios::goodbit | Non è impostato alcun bit di errore |
| ios::eofbit | 1 se viene incontrata la fine del file; 0 altrimenti |
| ios::failbit | 1 quando si verifica un errore di I/O non fatale (ammesso che esista); 0 altrimenti |
| ios::badbit | 1 quando si verifica un errore di I/O fatale; 0 altrimenti |

È possibile ottenere informazioni di stato sull'I/O in due modi. In primo luogo si può richiamare la funzione membro rdstate(). Il suo prototipo è:

```
iostate rdstate( );
```

La funzione restituisce lo stato corrente dei flag di errore. Come si può immaginare osservando l'elenco precedente, nel caso in cui non si verifichi alcun errore, rdstate() restituisce goodbit. In caso contrario verrà attivato un bit di errore.

Il seguente programma illustra l'uso di rdstate(). Il programma visualizza il contenuto di un file di testo. In caso di errore, il programma ne indica l'origine utilizzando checkstatus().

```
#include <iostream>
#include <fstream>
using namespace std;

void checkstatus(ifstream &in);

int main(int argc, char *argv[])
{
  if(argc!=2) {
    cout << "Uso: Display <nomefile>\n";
    return 1;
  }
```

```
  ifstream in(argv[1]);

  if(!in) {
    cout << "Impossibile aprire il file di input.\n";
    return 1;
  }

  char c;
  while(in.get(c)) {
    if (in) cout << c;
    checkstatus(in);
  }

  checkstatus(in);  // controlla lo stato finale
  in.close();
  return 0;
}

void checkstatus(ifstream &in)
{
  ios::iostate i;

  i = in.rdstate();

  if(i & ios::eofbit)
    cout << "Raggiunta la fine del file\n";
  else if(i & ios::failbit)
    cout << "Errore di I/O non fatale\n";
  else if(i & ios::badbit)
    cout << "Errore di I/O fatale\n";
}
```

Questo programma restituisce sempre almeno un "errore". Al termine del ciclo while, l'ultima chiamata a checkstatus() indica che è stata raggiunta la fine del file. Si troverà la funzione checkstatus() molto utile nei propri programmi.

L'altro modo in cui è possibile determinare se si è verificato un errore consiste nell'uso di una o più delle seguenti funzioni:

```
bool bad( );
bool eof( );
bool fail( );
bool good( );
```

La funzione bad() restituisce il valore logico vero quando è attivo il bit badbit.



La funzione eof() è stata già discussa in precedenza. La funzione fail() restituisce il valore logico vero quando è attivo il bit failbit. La funzione good restituisce il valore logico vero se non vi sono errori; in caso contrario restituisce il valore logico falso.

Quando si verifica un errore, per consentire al programma di continuare si dovrà reinizializzare lo stato di errore impostato. A tale scopo si utilizza la funzione clear() il cui prototipo è:

```
void clear(iostate flags=ios::goodbit);
```

Se *flags* è uguale a goodbit (impostazione standard) vengono riportati a zero tutti i flag di errore. Alternativamente si può impostare *flags* in modo da cancellare i soli flag di errore desiderati.

## 21.13 Personalizzazione delle operazioni di I/O sui file

Nel Capitolo 17 si sono apprese le tecniche di overloading degli operatori di inserimento ed estrazione rispetto a una data classe. In tale capitolo, si è parlato solo di operazioni di I/O da console. Ma poiché tutti gli stream C++ hanno lo stesso funzionamento, è possibile utilizzare la stessa funzione di inserimento o estrazione modificata tramite overloading per produrre il proprio output sullo schermo o su file. Il programma seguente rielabora l'esempio della rubrica del Capitolo 20 in modo da memorizzare un elenco di voci su disco. Il programma è molto semplice: consente di aggiungere nomi all'elenco o di visualizzare l'elenco sullo schermo. Tale programma legge e scrive i numeri telefonici tramite inseritori ed estrattori personalizzati. Si potrebbe provare a migliorare il programma in modo da ricercare un determinato numero o cancellare i numeri non desiderati.

```
#include <iostream>
#include <fstream>
#include <cstring>
using namespace std;

class phonebook {
  char name[80];
  char areacode[4];
  char prefix[4];
  char num[5];
public:
  phonebook() { };
```

```
  phonebook(char *n, char *a, char *p, char *nm)
  {
    strcpy(name, n);
    strcpy(areacode, a);
    strcpy(prefix, p);
    strcpy(num, nm);
  }
  friend ostream &operator<<(ostream &stream, phonebook o);
  friend istream &operator>>(istream &stream, phonebook &o);
};

// Visualizza il nome e il numero telefonico.
ostream &operator<<(ostream &stream, phonebook o)
{
  stream << o.name << " ";
  stream << "(" << o.areacode << ") ";
  stream << o.prefix << "-";
  stream << o.num << "\n";
  return stream; // deve restituire lo stream
}

// Input del nome e del numero telefonico.
istream &operator>>(istream &stream, phonebook &o)
{
  cout << "Indicare il nome: ";
  stream >> o.name;
  cout << "Indicare il codice: ";
  stream >> o.areacode;
  cout << "Indicare il prefisso: ";
  stream >> o.prefix;
  cout << "Indicare il numero: ";
  stream >> o.num;
  cout << "\n";
  return stream;
}

int main()
{
  phonebook a;
  char c;

  fstream pb("phone", ios::in | ios::out | ios::app);

  if(!pb) {
    cout << "Impossibile aprire il file della rubrica.\n";
    return 1;
```



```
  }

  for(;;) {
    do {
      cout << "1. Immissione dei numeri\n";
      cout << "2. Visualizzazione dei numeri\n";
      cout << "3. Uscita\n";
      cout << "\nScegliere un'opzione: ";
      cin >> c;
    } while(c<'1' || c>'3');

    switch(c) {
      case '1':
        cin >> a;
        cout << "Nuovo elemento: ";
        cout << a;  // visualizzazione
        pb << a;  // scrittura su disco
        break;
      case '2':
        char ch;
        pb.seekg(0, ios::beg);
        while(!pb.eof()) {
          pb.get(ch);
          if(!pb.eof()) cout << ch;
        }
        pb.clear();  // reinizializza eof
        cout << endl;
        break;
      case '3':
        pb.close();
        return 0;
    }
  }
}
```

Si noti che l'operatore << modificato tramite overloading può essere utilizzato per scrivere su un file su disco o sullo schermo senza alcuna modifica. Questa è una delle caratteristiche più importanti e utili dell'approccio alle operazioni di I/O seguito dal C++.

Capitolo 22

# L'identificazione run-time dei tipi e gli operatori cast

22.1 **L'identificazione run-time dei tipi (RTTI)**
22.2 **Gli operatori di conversione cast**
22.3 **L'operatore dynamic_cast**

Il C++ standard contiene due funzionalità che aiutano a supportare una moderna programmazione a oggetti: l'identificazione run-time dei tipi (abbreviata con RTTI) e i nuovi operatori di cast. Nessuno di questi elementi faceva parte delle specifiche originali del linguaggio C++ ma entrambi sono stati aggiunti per fornire un migliore supporto al polimorfismo run-time. Il meccanismo RTTI consente di identificare il tipo di un oggetto durante l'esecuzione del programma. Gli operatori di cast offrono un metodo di conversione più sicuro e controllato. Poiché uno degli operatori di cast, dynamic_cast, fa riferimento direttamente all'identificazione run-time dei tipi, è opportuno discutere entrambi gli argomenti nello stesso capitolo.

## 22.1 L'identificazione run-time dei tipi (RTTI)

L'identificazione run-time dei tipi può per qualcuno rappresentare una novità in quanto non è presente in alcun linguaggio non polimorfico, ad esempio il C. Nei linguaggi non polimorfici, non vi è alcuna necessità di avere informazioni run-time sui tipi poiché il tipo di un oggetto è già noto in fase di compilazione (ovvero viene specificato durante la scrittura del programma). Al contrario, nei linguaggi polimorfici come il C++, possono esistere situazioni in cui il tipo di un oggetto non è noto in fase di compilazione poiché l'esatta natura dell'oggetto rimane indeterminata fino all'esecuzione del programma. Come descritto nel Capitolo 17, il C++ implementa il polimorfismo tramite l'uso di gerarchie di classi, funzioni virtuali e puntatori alla classe base. Poiché i puntatori alla classe base possono essere utilizzati per puntare agli oggetti della classe base o a qualsiasi oggetto da essa derivato, non sempre è possibile sapere in anticipo quale tipo di oggetto sarà puntato da un puntatore alla classe base. Questa determinazione può essere eseguita solo tramite l'identificazione run-time dei tipi.

Per conoscere il tipo di un oggetto si utilizza typeid. Per utilizzare typeid si deve includere il file di intestazione <typeinfo>.

Normalmente typeid viene utilizzato nel seguente modo:

typeid(*oggetto*)

Qui *oggetto* è l'oggetto di cui si vuole conoscere il tipo. Può trattarsi di qualsiasi tipo, inclusi i tipi interni del linguaggio e le classi create dal programmatore. typeid restituisce l'indirizzo di un oggetto di tipo type_info che descrive il tipo di *oggetto*.

La classe type_info definisce i seguenti membri pubblici:

```
bool operator==(const type_info &ob);
bool operator!=(const type_info &ob);
bool before(const type_info &ob);
const char *name( );
```

Per eseguire confronti fra i tipi si utilizzano gli operatori == e != modificati tramite overloading. La funzione before() restituisce true se l'oggetto chiamante precede l'oggetto utilizzato come parametro nell'ordine di raccolta. Questa funzione ha principalmente un utilizzo interno. Il valore che restituisce non ha niente a che vedere con l'ereditarietà e le gerarchie di classi. La funzione name() restituisce un puntatore al nome del tipo.

Ecco un semplice esempio che utilizza typeid.

```
// Un semplice esempio che usa typeid.
#include <iostream>
#include <typeinfo>
using namespace std;

class myclass1 {
  // ...
};

class myclass2 {
  // ...
};

int main()
{
  int i, j;
  float f;
  char *p;
  myclass1 ob1;
  myclass2 ob2;
```



```
  cout << "Il tipo di i è: " << typeid(i).name();
  cout << endl;
  cout << "Il tipo di f è: " << typeid(f).name();
  cout << endl;
  cout << "Il tipo di p è: " << typeid(p).name();
  cout << endl;

  cout << "Il tipo di ob1 è: " << typeid(ob1).name();
  cout << endl;
  cout << "Il tipo di ob2 è: " << typeid(ob2).name();
  cout << "\n\n";

  if(typeid(i) == typeid(j))
    cout << "I tipi di i e j sono uguali";

  if(typeid(i) != typeid(f))
    cout << "I tipi di i e f sono differenti";

  if(typeid(ob1) != typeid(ob2))
    cout << "I tipi di ob1 e ob2 sono differenti";

  return 0;
}
```

Ecco l'output prodotto dal programma:

```
Il tipo di i è: int
Il tipo di f è: float
Il tipo di p è: char *
Il tipo di ob1 è: class myclass1
Il tipo di ob2 è: class myclass2

I tipi di i e j sono uguali
I tipi di i e f sono differenti
I tipi di ob1 e ob2 sono differenti
```

L'utilizzo più importante di typeid si verifica quando tale operatore viene applicato attraverso un puntatore a una classe base polimorfica. In questo caso, typeid restituisce automaticamente il tipo dell'oggetto effettivamente puntato, che può essere un oggetto appartenente alla classe base oppure un oggetto da essa derivato. Si ricordi che un puntatore a una classe base può puntare a oggetti della classe base oppure a uno qualsiasi degli oggetti da essa derivati. Pertanto, utilizzando typeid è possibile determinare run-time il tipo dell'oggetto puntato dal puntatore alla classe base. Questo principio è dimostrato dal seguente programma.

```
// Un esempio che usa typeid su una gerarchia di classi polimorfica.
#include <iostream>
#include <typeinfo>
using namespace std;

class Mammal {
public:
  virtual bool lays_eggs() { return false; } // Mammal è polimorfica
  // ...
};

class Cat: public Mammal {
public:
  // ...
};

class Platypus: public Mammal {
public:
  bool lays_eggs() { return true; }
  // ...
};

int main()
{
  Mammal *p, AnyMammal;
  Cat cat;
  Platypus platypus;

  p = &AnyMammal;
  cout << "p punta a un oggetto di tipo ";
  cout << typeid(*p).name() << endl;

  p = &cat;
  cout << "p punta a un oggetto di tipo ";
  cout << typeid(*p).name() << endl;

  p = &platypus;
  cout << "p punta a un oggetto di tipo ";
  cout << typeid(*p).name() << endl;

  return 0;
}
```

Ecco l'output prodotto dal programma:



```
p punta a un oggetto di tipo class Mammal
p punta a un oggetto di tipo class Cat
p punta a un oggetto di tipo class Platypus
```

Come si è detto, quando typeid viene applicato a un puntatore alla classe base di un tipo polimorfico, il tipo dell'oggetto puntato viene determinato run-time, come indicato dall'output prodotto dal programma.

In ogni caso, quando typeid viene applicato a un puntatore a una gerarchia di classi non polimorfica, allora si ottiene il tipo base del puntatore. Questo significa che non viene fatto alcuno sforzo per determinare l'oggetto cui il puntatore sta effettivamente puntando. Ad esempio si provi a trasformare in commento la parola riservata virtual che precede la funzione lays_eggs() in Mammal e a compilare ed eseguire il programma. Verrà visualizzato il seguente output.

```
p punta a un oggetto di tipo class Mammal
p punta a un oggetto di tipo class Mammal
p punta a un oggetto di tipo class Mammal
```

Poiché Mammal non è più una classe polimorfica, il tipo dell'oggetto sarà Mammal poiché questo è il tipo del puntatore.

Dato che typeid viene in genere applicato a un puntatore deindirizzato (ovvero un puntatore cui viene applicato l'operatore *), è stata creata una speciale estensione per gestire la situazione in cui il puntatore deindirizzato è nullo. In questo caso, typeid lancia l'eccezione bad_typeid.

Gli indirizzi di un oggetto di una gerarchia polimorfica di classe funzionano come i puntatori. Quando si applica typeid a un indirizzo di un oggetto di una classe polimorfica, typeid restituisce il tipo dell'oggetto cui si sta facendo effettivamente riferimento, il quale può essere un tipo derivato. La circostanza in cui si utilizza più frequentemente questa funzionalità è quando gli oggetti vengono passati alle funzioni per indirizzo. Ad esempio, nel seguente programma, la funzione WhatMammal() dichiara un parametro indirizzo a oggetti di tipo Mammal. Questo significa che a WhatMammal() possono essere passati riferimenti a oggetti di tipo Mammal o di qualsiasi classe derivata da Mammal. Quando a questo parametro viene applicato l'operatore typeid, viene restituito il tipo dell'oggetto effettivamente passato.

```
// Usa typeid con un indirizzo.
#include <iostream>
#include <typeinfo>
using namespace std;

class Mammal {
public:
```

```
  virtual bool lays_eggs() { return false; } // Mammal è polimorfica
  // ...
};

class Cat: public Mammal {
public:
  // ...
};

class Platypus: public Mammal {
public:
  bool lays_eggs() { return true; }
  // ...
};

 // Illustra l'uso di typeid con un parametro di tipo indirizzo.
void WhatMammal(Mammal &ob)
{
  cout << "ob è l'indirizzo di un oggetto di tipo ";
  cout << typeid(ob).name() << endl;
}

int main()
{
  Mammal AnyMammal;
  Cat cat;
  Platypus platypus;

  WhatMammal(AnyMammal);
  WhatMammal(cat);
  WhatMammal(platypus);

  return 0;
}
```

Ecco l'output prodotto dal programma:

```
ob è l'indirizzo di un oggetto di tipo class Mammal
ob è l'indirizzo di un oggetto di tipo class Cat
ob è l'indirizzo di un oggetto di tipo class Platypus
```

Vi è anche una seconda forma di typeid che accetta come argomento il nome di un tipo. Ecco l'uso di questa forma:

typeid(*nome-tipo*)



Ad esempio, la seguente istruzione è perfettamente lecita:

```
cout << typeid(int).name();
```

L'uso principale di questa forma di typeid consiste nell'ottenere un oggetto di tipo type_info che descrive il tipo specificato e che possa essere utilizzato in un'istruzione di confronto fra tipi. Ad esempio, questa forma di typeid indica che i gatti non amano l'acqua:

```
void WhatMammal(Mammal &ob)
{
  cout << "ob è l'indirizzo di un oggetto di tipo ";
  cout << typeid(ob).name() << endl;
  if(typeid(ob) == typeid(Cat))
    cout << "I gatti non amano l'acqua.";
}
```

## Una semplice applicazione dell'identificazione run-time dei tipi

Il seguente programma suggerisce le potenzialità del sistema RTTI. Nel programma, la funzione factory() crea istanze di vari tipi di oggetti derivati dalla classe Mammal (una funzione che produce oggetti viene spesso chiamata "fabbrica di oggetti" o "object factory"). Il tipo specifico dell'oggetto creato viene determinato dal risultato di una chiamata a rand(), il generatore di numeri casuali del C++. Pertanto non vi è alcun modo di sapere in anticipo quale tipo di oggetto verrà generato. Il programma crea 10 oggetti e conta il numero di mammiferi. Poiché una chiamata a factory() può generare qualsiasi tipo di mammifero, il programma determina il tipo dell'oggetto facendo affidamento su typeid.

```
// Illustra l'identificazione run-time dei tipi.
#include <iostream>
using namespace std;

class Mammal {
public:
  virtual bool lays_eggs() { return false; } // Mammal è polimorfica
  // ...
};

class Cat: public Mammal {
public:
```

```
  // ...
};

class Platypus: public Mammal {
public:
  bool lays_eggs() { return true; }
  // ...
};

class Dog: public Mammal {
public:
 // ...
};

// Una "fabbrica" di oggetti derivati da Mammal.
Mammal *factory()
{
  switch(rand() % 3 ) {
    case 0: return new Dog;
    case 1: return new Cat;
    case 2: return new Platypus;
  }
  return 0;
}

int main()
{
  Mammal *ptr; // puntatore alla classe base
  int i;
  int c=0, d=0, p=0;

  // genera e conta gli oggetti
  for(i=0; i<10; i++) {
    ptr = factory(); // generate an object

    cout << "L'oggetto è di tipo " << typeid(*ptr).name();
    cout << endl;

    // conteggio
    if(typeid(*ptr) == typeid(Dog)) d++;
    if(typeid(*ptr) == typeid(Cat)) c++;
    if(typeid(*ptr) == typeid(Platypus)) p++;
}

  cout << endl;
  cout << "Animali generati:\n";
```



```
  cout << "  Cani: " << d << endl;
  cout << "  Gatti: " << c << endl;
  cout << "  Ornitorinchi: " << p << endl;

  return 0;
}
```

Ecco l'output del programma:

```
L'oggetto è di tipo class Platypus
L'oggetto è di tipo class Platypus
L'oggetto è di tipo class Cat
L'oggetto è di tipo class Cat
L'oggetto è di tipo class Platypus
L'oggetto è di tipo class Cat
L'oggetto è di tipo class Dog
L'oggetto è di tipo class Dog
L'oggetto è di tipo class Cat
L'oggetto è di tipo class Platypus

Animali generati:
  Cani: 2
  Gatti: 4
  Ornitorinchi: 4
```

## typeid può essere applicato anche a classi template

L'operatore typeid può essere applicato anche a classi template. Il tipo di un oggetto che è un'istanza di una classe template è in parte determinato dai valori utilizzati per i suoi dati generici nel momento in cui l'oggetto viene istanziato. Due istanze della stessa classe template che sono create utilizzando dati differenti rappresentano pertanto tipi differenti. Ecco un semplice esempio:

```
// Uso di typeid con i template.
#include <iostream>
using namespace std;

template <class T> class myclass {
  T a;
public:
  myclass(T i) { a = i; }
  // ...
};
```

```
int main()
{
  myclass<int> o1(10), o2(9);
  myclass<double> o3(7.2);

  cout << "Il tipo di o1 è ";
  cout << typeid(o1).name() << endl;

  cout << "Il tipo di o2 è ";
  cout << typeid(o2).name() << endl;

  cout << "Il tipo di o3 è ";
  cout << typeid(o3).name() << endl;

  cout << endl;

  if(typeid(o1) == typeid(o2))
    cout << "o1 e o2 hanno lo stesso tipo\n";

  if(typeid(o1) == typeid(o3))
    cout << "Errore\n";
  else
    cout << "o1 e o3 hanno tipi differenti\n";

  return 0;
}
```

Ecco l'output prodotto dal programma.

```
Il tipo di ob1 è class myclass<int>
Il tipo di ob2 è class myclass<int>
Il tipo di o3 è class myclass<double>

o1 e o2 hanno lo stesso tipo
o1 e o3 hanno tipi differenti
```

Come si può vedere, anche se i due oggetti appartengono alla stessa classe template, se i loro dati parametrizzati non corrispondono, essi non rappresentano tipi equivalenti. Nel programma, o1 è di tipo myclass<int> e o3 è di tipo myclass<double>. Pertanto essi rappresentano tipi differenti.

L'identificazione run-time dei tipi non è una funzionalità che possa risultare utile in molti programmi. Tuttavia, quando si impiegano tipi polimorfici, consente di sapere quale tipo di oggetto si sta impiegando in una determinata situazione.



## 22.2 Gli operatori di conversione cast

Il linguaggio C++ definisce cinque operatori di cast. Il primo è rappresentato dalla forma tradizionale di conversione cast ereditata dal linguaggio C. Gli altri quattro sono stati aggiunti pochi anni fa. Tali operatori sono dynamic_cast, const_cast, reinterpret_cast e static_cast. Questi operatori offrono un maggiore controllo sulle conversioni di tipo.

## 22.3 L'operatore dynamic_cast

dynamic_cast è forse l'operatore di cast più importante. Tale operatore esegue una conversione run-time che verifica la validità della conversione. Se al momento dell'esecuzione di dynamic_cast la conversione è errata, allora l'operatore dynamic_cast non ha successo. Ecco la forma generale di dynamic_cast:

dynamic_cast<*tipo-destinazione*> (*espressione*)

Qui *tipo-destinazione* specifica il tipo di destinazione della conversione cast ed *espressione* è l'espressione che viene convertita nel nuovo tipo. Il tipo di destinazione deve essere un puntatore o un indirizzo e l'espressione convertita deve fornire un puntatore o un indirizzo. Pertanto dynamic_cast può essere utilizzato per convertire un tipo di puntatore in un altro o un tipo di indirizzo in un altro.

Lo scopo di dynamic_cast è quello di eseguire conversioni su tipi polimorfici. Ad esempio, date due classi polimorfiche chiamate B e D, con D derivata da B, un dynamic_cast può sempre convertire un puntatore D* in un puntatore B*. Questo è dovuto al fatto che un puntatore alla classe base può sempre puntare a un oggetto derivato. Ma dynamic_cast può convertire un puntatore D* in un puntatore B* solo se l'oggetto puntato è effettivamente un oggetto di tipo D. In generale dynamic_cast ha successo se il puntatore (o l'indirizzo) convertito è un puntatore (o un indirizzo) a un oggetto del tipo di destinazione o un oggetto derivato da tale tipo. In tutti gli altri casi la conversione non avrà successo. Se la conversione non ha successo, quando la conversione riguarda i puntatori, dynamic_cast produce il valore null mentre quando riguarda gli indirizzi viene lanciata un'eccezione bad_cast.

Ecco un semplice esempio. Si supponga che Base sia una classe polimorfica e che Derived derivi da Base.

```
Base *bp, b_ob;
Derived *dp, d_ob;

bp = &d_ob; // il puntatore alla classe base punta a un oggetto Derived
```

```
dp = dynamic_cast<Derived *> (bp); // Conversione nel tipo derivato
if(dp) cout << "Cast K";
```

Qui la conversione dal puntatore alla classe base bp in un puntatore alla classe derivata dp funziona perché bp sta in effetti puntando ad un oggetto Derived. Pertanto questo frammento di codice esegue la conversione. Ma nel successivo frammento, la conversione non ha successo poiché bp punta a un oggetto Base ed è vietato convertire un oggetto base in un oggetto derivato.

```
bp = &b_ob; // il puntatore alla classe base punta a un oggetto Base
dp = dynamic_cast<Derived *> (bp); // errore
if(!dp) cout << "Cast Fails";
```

In questo frammento di codice la conversione non ha successo.

Il seguente programma mostra le varie situazioni che possono essere gestite con dynamic_cast.

```
// Illustra l'uso di dynamic_cast.
#include <iostream>
using namespace std;

class Base {
public:
  virtual void f() { cout << "All'interno di base\n"; }
  // ...
};

class Derived : public Base {
public:
  void f() { cout << "All'interno di Derived\n"; }
;
}

int main()
{
  Base *bp, b_ob;
  Derived *dp, d_ob;

  dp = dynamic_cast<Derived *> (&d_ob);
  if(dp) {
    cout << "Conversione cast da Derived * a Derived * - OK.\n";
    dp->f();
}   else
    cout << "Errore\n";
```



```
  cout << endl;

  bp = dynamic_cast<Base *> (&d_ob);
  if(bp) {
    cout << "Conversione cast da Derived * a Base * - OK.\n";
    bp->f();
}   else
    cout << "Errore\n";

  cout << endl;

  bp = dynamic_cast<Base *> (&b_ob);
  if(bp) {
    cout << "Conversione cast da Base * a Base * - OK.\n";
    bp->f();
}   else
    cout << "Errore\n";

  cout << endl;

  dp = dynamic_cast<Derived *> (&b_ob);
  if(dp)
    cout << "Errore\n";
  else
    cout << "Conversione cast da Base * a Derived * - Errore.\n";

  cout << endl;

  bp = &d_ob; // bp punta a un oggetto Derived
  dp = dynamic_cast<Derived *> (bp);
  if(dp) {
    cout << "Conversione cast di bp in un Derived * - OK\n" <<
      "è un errore poiché bp in realtà punta\n" <<
      "a un oggetto Derived.\n";
    dp->f();
}   else
    cout << "Errore\n";

  cout << endl;

  bp = &b_ob; // bp punta a un oggetto Base
  dp = dynamic_cast<Derived *> (bp);
  if(dp)
    cout << "Errore";
  else {
    cout << "Ora converte bp in un Derived *\n" <<
```

```
      "è un errore poichè in realtà bp \n" <<
      "punta a un oggetto di tipo Base.\n";
}

  cout << endl;

  dp = &d_ob; // dp punta a un oggetto Derived
  bp = dynamic_cast<Base *> (dp);
  if(bp) {
    cout << "Conversione cast di dp in Base * - OK.\n";
    bp->f();
}   else
    cout << "Errore\n";

  return 0;
}
```

Ecco l'output prodotto dal programma.

```
Conversione cast da Derived * a Derived * - OK.
All'interno di Derived

Conversione cast da Derived * a Base * - OK.
All'interno di Derived

Conversione cast da Base * a Base * - OK.
All'interno di Base

Conversione cast da Base * a Derived * - Errore.

Conversione cast di bp in un Derived * OK
poichè bp in realtà punta
a un oggetto Derived.
All'interno di Derived

Ora converte bp in un Derived *
è un errore poichè bp in realtà punta
a un oggetto di tipo Base.

Conversione cast di dp in Base * - OK.
All'interno di Derived
```



## Sostituzione di typeid con dynamic_cast

L'operatore dynamic_cast può in alcuni casi essere utilizzato al posto di typeid. Ad esempio, si supponga ancora che Base sia una classe base polimorfica da cui deriva Derived. Il seguente frammento di codice assegna a dp l'indirizzo dell'oggetto puntato da bp se e solo se l'oggetto è veramente di tipo Derived.

```
Base *bp;
Derived *dp;
// ...
if(typeid(*bp) == typeid(Derived)) dp = (Derived *) bp;
```

In questo caso, per eseguire la conversione viene utilizzata un'operazione cast tradizionale. Tale operazione è sicura grazie al fatto che l'istruzione if controlla l'accettabilità del cast impiegando typeid prima di eseguire la conversione. Ma esiste un modo migliore per svolgere l'operazione: sostituire gli operatori typeid e l'istruzione if con la seguente istruzione contenente dynamic_cast.

```
dp = dynamic_cast<Derived *> (bp);
```

Poiché dynamic_cast ha successo solo se l'oggetto da convertire è un oggetto del tipo di destinazione o un oggetto di un tipo da esso derivato, dopo l'esecuzione di questa istruzione dp conterrà null o un puntatore a un oggetto di tipo Derived. Dato che dynamic_cast ha successo solo se la conversione cast è consentita, in queste situazioni può semplificare la logica del programma. Il seguente programma illustra il modo in cui dynamic_cast può essere utilizzato per sostituire typeid. L'operazione viene eseguita due volte, la prima con typeid e la seconda con dynamic_cast.

```
// Usa dynamic_cast al posto di typeid.
#include <iostream>
#include <typeinfo>
using namespace std;

class Base {
public:
  virtual void f()  {}
};

class Derived : public Base {
public:
  void derivedOnly() {
```

```
    cout << "È un oggetto di tipo Derived.\n";
  }
};

int main()
{
  Base *bp, b_ob;
  Derived *dp, d_ob;

  // ************************************
  // usa typeid
  // ************************************
  bp = &b_ob;
  if(typeid(*bp) == typeid(Derived)) {
    dp = (Derived *) bp;
    dp->derivedOnly();
  }
  else
    cout << "Conversione cast da Base a Derived fallita.\n";

  bp = &d_ob;
  if(typeid(*bp) == typeid(Derived)) {
    dp = (Derived *) bp;
    dp->derivedOnly();
  }
  else
    cout << "Errore, la conversione cast dovrebbe funzionare!\n";

  // ************************************
  // usa dynamic_cast
  // ************************************
  bp = &b_ob;
  dp = dynamic_cast<Derived *> (bp);
  if(dp) dp->derivedOnly();
  else
    cout << "Conversione cast da Base a Derived fallita.\n";

  bp = &d_ob;
  dp = dynamic_cast<Derived *> (bp);
  if(dp) dp->derivedOnly();
  else
    cout << "Errore, la conversione cast dovrebbe funzionare!\n";

  return 0;
}
```



Come si può vedere, l'uso di dynamic_cast semplifica la logica necessaria per convertire un puntatore alla classe base in un puntatore alla classe derivata. Ecco l'output del programma:

```
Conversione cast da Base a Derived fallita.
È un oggetto di tipo Derived.
Conversione cast da Base a Derived fallita.
È un oggetto di tipo Derived.
```

## Uso di dynamic_cast con classi template

L'operatore dynamic_cast può essere utilizzato anche con le classi template. Ad esempio,

```
// Illustra l'uso di dynamic_cast su classi template.
#include <iostream>
using namespace std;

template <class T> class Num {
protected:
  T val;
public:
  Num(T x) { val = x; }
  virtual T getval() { return val; }
  // ...
};

template <class T> class SqrNum : public Num<T> {
public:
  SqrNum(T x) : Num<T>(x)  { }
  T getval() { return val * val; }
};

int main()
{
  Num<int> *bp, numInt_ob(2);
  SqrNum<int> *dp, sqrInt_ob(3);
  Num<double> numDouble_ob(3.3);

  bp = dynamic_cast<Num<int> *> (&sqrInt_ob);
  if(bp) {
    cout << "Conversione cast da SqrNum<int>* a Num<int>* - OK.\n";
    cout << "Il valore è " << bp->getval() << endl;
  } else
```

```
    cout << "Errore\n";

  cout << endl;

  dp = dynamic_cast<SqrNum<int> *> (&numInt_ob);
  if(dp)
    cout << "Errore\n";
  else {
    cout << "Conversione cast da Num<int>* a SqrNum<int>* non consentita.\n";
    cout << "Non si può eseguire una conversione cast di un puntatore alla
classe base\n";
    cout << "in un puntatore a una classe derivata.\n";
  }
  cout << endl;

  bp = dynamic_cast<Num<int> *> (&numDouble_ob);
  if(bp)
    cout << "Errore\n";
  else
    cout << "Conversione cast impossibile da Num<double>* a Num<int>*.\n";
    cout << "Sono due tipi differenti.\n";

  return 0;
}
```

Ecco l'output del programma:

```
Conversione cast da SqrNum<int>* a Num<int>* - OK.
Il valore è 9

Conversione cast da Num<int>* a SqrNum<int>* non consentita.
Non si può eseguire una conversione cast di un puntatore alla classe base
in un puntatore a una classe derivata.

Conversione cast impossibile da Num<double>* a Num<int>*.
Sono due tipi differenti.
```

Ciò che questo esempio intende illustrare è il fatto che non è possibile utilizzare dynamic_cast per convertire un puntatore a un tipo di istanziazione template in un puntatore a un altro tipo di istanza. Si ricordi: l'esatto tipo di un oggetto di una classe template è determinato dal tipo dei dati utilizzati per creare l'istanza del template. Pertanto Num<double> e Num<int> sono due tipi differenti.



## L'operatore const_cast

L'operatore const_cast è utilizzato per rimuovere esplicitamente l'attributo const e/o volatile in una conversione cast. Il tipo di destinazione deve essere lo stesso del tipo di origine ad eccezione dell'eliminazione degli attributi const e/o volatile. Normalmente l'operatore const_cast viene utilizzato per rimuovere l'attributo const. Ecco la forma generale di const_cast.

const_cast<*tipo*> (*espressione*)

Qui, tipo specifica il tipo di destinazione della conversione ed espressione è l'espressione da convertire nel nuovo tipo. Il seguente programma illustra l'utilizzo di const_cast.

```
// Illustra l'uso di const_cast.
#include <iostream>
using namespace std;

void sqrval(const int *val)
{
  int *p;

  // elimina l'attributo const.
  p = const_cast<int *> (val);

  *p = *val * *val; // ora modifica l'oggetto tramite v
}

int main()
{
  int x = 10;

  cout << "x prima della chiamata: " << x << endl;
  sqrval(&x);
  cout << "x dopo la chiamata: " << x << endl;

  return 0;
}
```

Ecco di seguito l'output prodotto da questo programma:

```
x prima della chiamata: 10
x dopo la chiamata: 100
```

Come si può vedere, x è stato modificato da sqrval() anche se il parametro di sqrval() è stato specificato come un puntatore const.

L'operatore const_cast può essere utilizzato anche per eliminare l'attributo const da un indirizzo. Ad esempio ecco una rielaborazione del programma precedente che prevede un passaggio del valore tramite un indirizzo const.

```
// Usa const_cast su un indirizzo const.
#include <iostream>
using namespace std;

void sqrval(const int &val)
{
  // elimina l'attributo const su val
  const_cast<int &> (val) = val * val;
}

int main()
{
  int x = 10;

  cout << "x prima della chiamata: " << x << endl;
  sqrval(x);
  cout << "x dopo la chiamata: " << x << endl;

  return 0;
}
```

Questo programma produce lo stesso output del precedente. Anche questo programma funziona per il solo motivo che const_cast rimuove temporaneamente l'attributo const da val, consentendo dunque di utilizzarlo per assegnare un nuovo valore all'argomento chiamante (in questo caso x).

Occorre però notare che l'uso di const_cast per eliminare l'attributo const può essere pericoloso. Dunque è opportuno impiegarlo con cura.

Un ultima annotazione: solo const_cast può rimuovere l'attributo const. Tale operazione non può invece essere eseguita né con dynamic_cast, né con static_cast e neppure con reinterpret_cast.

## L'operatore static_cast

L'operatore static_cast esegue una conversione non polimorfica. Tale operatore può essere utilizzato per ogni conversione standard. Non verrà eseguita alcuna verifica run-time. La sua forma generale è:



static_cast<*tipo*> (*espressione*)

Qui, tipo specifica il tipo di destinazione della classe ed espressione è l'espressione da convertire nel nuovo tipo.

L'operatore static_cast sostituisce in generale la conversione cast tradizionale. Tale operatore non fa altro che eseguire una conversione cast non polimorfica. Ad esempio, il seguente programma converte un valore int in un double.

```
// Usa static_cast.
#include <iostream>
using namespace std;

int main()
{
  int i;

  for(i=0; i<10; i++)
    cout << static_cast<double> (i) / 3 << " ";

  return 0;
}
```

## L'operatore reinterpret_cast

L'operatore reinterpret_cast converte un tipo in un tipo radicalmente differente. Ad esempio può trasformare un puntatore in un intero e viceversa. Può essere utilizzato anche per eseguire conversioni fra tipi inerentemente incompatibili. La sua forma generale è:

reinterpret_cast<*tipo*> (*espressione*)

Qui tipo specifica di tipo di destinazione della conversione e espressione è l'espressione da convertire nel nuovo tipo.

L'uso dell'operatore reinterpret_cast è illustrato dal seguente esempio.

```
// Un esempio che usa reinterpret_cast.
#include <iostream>
using namespace std;

int main()
```

```
{
  int i;
  char *p = "Questa è una stringa";

  i = reinterpret_cast<int> (p); // conversione cast di un puntatore in un
intero

  cout << i;

  return 0;
}
```

Qui reinterpret_cast converte il puntatore p in un intero. Questa conversione rappresenta una modifica radicale di tipo e dunque è un buon esempio di utilizzo di reinterpret_cast.

Capitolo 23

# Namespace, funzioni di conversione e altri argomenti avanzati


23.1 **I namespace**
23.2 **Lo spazio dei nomi std**
23.3 **Creazione di funzioni di conversione**
23.4 **Funzioni membro const e mutable**
23.5 **Funzioni membro volatile**
23.6 **Costruttori espliciti**
23.7 **Uso della parola riservata asm**
23.8 **Specifiche di linking**
23.9 **Operazioni di I/O su array**
23.10 **Uso di array dinamici**
23.11 **Uso di I/O binario con stream basati su array**
23.12 **Riepilogo delle differenze esistenti fra C e C++**


Questo capitolo descrive l'uso degli spazi dei nomi (namespace) e altre funzionalità avanzate fra le quali le funzioni di conversione, i costruttori espliciti, le funzioni membro const e volatile, la parola riservata asm e le specifiche di linking. Il capitolo termina con una discussione sulle operazioni di I/O basate su array e un riepilogo delle differenze fra C e C++.

## 23.1 I namespace

I *namespace* (spazi dei nomi) sono stati brevemente introdotti in precedenza. Si tratta di un'aggiunta relativamente recente al linguaggio C++. Il loro scopo è quello di localizzare il nome degli identificatori per evitare collisioni fra i nomi. L'ambiente di programmazione C++ ha visto un'esplosione di nomi di variabili, funzioni e classi. Prima dell'invenzione di namespace, tutti questi nomi erano a caccia di un proprio angolino nello spazio dei nomi globale e questo portava alla nascita di vari conflitti. Ad esempio, se il programma definiva una funzione chia-

mata abs(), questa poteva (a seconda dei valori utilizzati per i parametri) sovrapporsi alla funzione omonima della libreria standard, poiché entrambi i nomi si sarebbero trovati nello stesso spazio dei nomi. Potevano sorgere collisioni fra nomi quando un programma utilizzava due o più librerie prodotte da terzi. In questo caso era possibile che un nome definito da una libreria entrasse in conflitto con lo stesso nome definito dall'altra libreria. La situazione poteva essere particolarmente problematica nel caso dei nomi delle classi. Ad esempio, se il programma definisce una classe chiamata ThreeDCircle e una libreria utilizzata dal programma definisce una classe avente lo stesso nome, questo porterebbe alla nascita di un conflitto.

La creazione della parola riservata namespace ha lo scopo di rispondere a questi problemi. Per il fatto che localizza al proprio interno la visibilità dei nomi, uno spazio dei nomi consente di utilizzare lo stesso nome in contesti differenti senza che questo provochi conflitti. Probabilmente chi si è più avvantaggiato nell'utilizzo di namespace è la libreria standard del C++. Prima di namespace, l'intera libreria C++ era definita nello spazio dei nomi globale (che ovviamente era l'unico disponibile). Dall'aggiunta della parola riservata namespace, la libreria C++ è definita all'interno di un proprio namespace chiamato std che riduce la possibilità che sorgano collisioni fra i nomi. Si può anche creare un proprio spazio dei nomi in modo da localizzare la visibilità dei nomi che si pensa possano provocare conflitti. Questo è particolarmente importante quando si creano librerie di classi o di funzioni.

## Le basi degli spazi di nomi

La parola riservata namespace consente di partizionare lo spazio globale di nomi creando una regione di dichiarazioni. In pratica uno spazio dei nomi definisce un campo di visibilità. Ecco la forma generale della parola riservata namespace:

```
namespace nome
  // dichiarazioni
```

Tutto ciò che è definito in un'istruzione namespace è all'interno del campo d'azione di tale namespace.

Ecco un esempio di namespace che localizza i nomi utilizzati per implementare una semplice classe di conto alla rovescia. Nello spazio dei nomi sono definite la classe counter che implementa il contatore e le variabili upperbound e lowerbound che contengono i limiti superiore e inferiore che si applicano a tutti i contatori.

```
namespace CounterNameSpace {
  int upperbound;
  int lowerbound;

  class counter {
    int count;
  public:
    counter(int n) {
      if(n <= upperbound) count = n;
      else count = upperbound;
    }

    void reset(int n) {
      if(n <= upperbound) count = n;
    }

    int run() {
      if(count > lowerbound) return count--;
      else return lowerbound;
    }
  };
}
```



Qui, upperbound, lowerbound e la classe counter fanno parte del campo d'azione definito dallo spazio dei nomi CounterNamespace.

All'interno di uno spazio dei nomi, gli identificatori dichiarati al suo interno possono essere utilizzati direttamente, senza qualificarli con il nome del namespace stesso. Ad esempio, all'interno di CounterNamespace, la funzione run() può far riferimento direttamente a lowerbound nell'istruzione seguente:

```
if(count > lowerbound) return count--;
```

Ma poiché namespace definisce un campo di visibilità, per far riferimento a oggetti dichiarati in un uno spazio dei nomi differente occorre utilizzare l'operatore di risoluzione del campo di azione.

Ad esempio, per assegnare il valore 10 a upperbound da parte di codice che si trova all'esterno di CounterNamespace, si deve utilizzare la seguente istruzione:

```
CounterNameSpace::upperbound = 10;
```

Oppure, per dichiarare un oggetto di tipo counter dall'esterno di CounterNamespace, si utilizza un'istruzione del seguente tipo:

```
CounterNameSpace::counter ob;
```

In generale, per accedere a un membro di uno spazio dei nomi dall'esterno di tale spazio dei nomi, occorre far precedere al nome del membro il nome dello spazio dei nomi seguito dall'operatore di risoluzione del campo d'azione.

Ecco un programma che mostra l'uso di CounterNamespace.

```
// Illustra l'uso di un namespace.
#include <iostream>
using namespace std;

namespace CounterNameSpace {
  int upperbound;
  int lowerbound;

  class counter {
     int count;
   public:
     counter(int n) {
       if(n <= upperbound) count  n;
       else count = upperbound;
     }

     void reset(int n) {
       if(n <= upperbound) count  n;
     }

     int run() {
       if(count > lowerbound) return count--;
       else return lowerbound;
     }
  };
}

int main()
{
  CounterNameSpace::upperbound = 100;
  CounterNameSpace::lowerbound = 0;

  CounterNameSpace::counter ob1(10);
  int i;

  do {
    i = ob1.run();
    cout << i << " ";
  } while(i > CounterNameSpace::lowerbound);
  cout << endl;
```



```
  CounterNameSpace::counter ob2(20);

  do {
    i = ob2.run();
    cout << i << " ";
  } while(i > CounterNameSpace::lowerbound);
  cout << endl;

  ob2.reset(100);
  CounterNameSpace::lowerbound = 90;
  do {
    i = ob2.run();
    cout << i << " ";
  } while(i > CounterNameSpace::lowerbound);

  return 0;
}
```

Si noti che la dichiarazione di un oggetto counter e i riferimenti a upperbound e lowerbound sono qualificati da CounterNamespace. Tuttavia, una volta che è stato dichiarato un oggetto di tipo counter, non è necessario qualificarlo ulteriormente o qualificare i suoi membri. Pertanto ob1.run() può essere richiamata direttamente in quanto il suo spazio dei nomi è già stato risolto.

## L'istruzione using

Come si può immaginare, se il programma include numerosi riferimenti ai membri di uno spazio dei nomi, il fatto di dover specificare il nome dello spazio dei nomi e l'operatore di risoluzione del campo d'azione ad ogni riferimento, può diventare fastidiosa. Per risolvere questo problema è stata introdotta la parola riservata using. Ecco la forma generale della parola riservata using:

using namespace *nome;*
using *nome*::*membro*;

Nella prima forma, nome specifica il nome del namespace cui si vuole accedere. Tutti i membri definiti all'interno dello spazio di nomi specificato diventano quindi immediatamente visibili (in pratica entrano a far parte dello spazio dei nomi corrente) e possono essere utilizzati senza alcuna qualifica. La seconda forma rende visibile un solo membro dello spazio dei nomi. Ad esempio, supponendo uno spazio dei nomi CounterNamespace come quello illustrato in precedenza, è possibile utilizzare le seguenti istruzioni using.

```
using CounterNameSpace::lowerbound; // diventa visibile solo lowerbound
lowerbound = 10; // OK perché lowerbound è visibile

using namespace CounterNameSpace; // diventano visibili tutti i membri
upperbound = 100; // OK perché ora sono visibili tutti i membri
```

Il seguente programma illustra l'uso di using rielaborando l'esempio del contatore sviluppato nella sezione precedente.

```
// Illustra l'uso di using.
#include <iostream>
using namespace std;

namespace CounterNameSpace {
  int upperbound;
  int lowerbound;

  class counter {
    int count;
  public:
    counter(int n) {
      if(n <= upperbound) count = n;
      else count = upperbound;
    }

    void reset(int n) {
      if(n <= upperbound) count = n;
    }

    int run() {
      if(count > lowerbound) return count--;
      else return lowerbound;
    }
  };
}

int main()
{
  // usa solo upperbound di CounterNameSpace
  using CounterNameSpace::upperbound;

  // ora non è necessario qualificare upperbound
  upperbound = 100;

  // ma è ancora necessario qualificare lowerbound ecc.
  CounterNameSpace::lowerbound = 0;

  CounterNameSpace::counter ob1(10);
  int i;

  do {
    i = ob1.run();
    cout << i << " ";
  } while(i > CounterNameSpace::lowerbound);
  cout << endl;

  // ora usa tutto CounterNameSpace
  using namespace CounterNameSpace;

  counter ob2(20);

  do {
    i = ob2.run();
    cout << i << " ";
  } while(i > lowerbound);
  cout << endl;

  ob2.reset(100);
  lowerbound = 90;
  do {
    i = ob2.run();
    cout << i << " ";
  } while(i > lowerbound);

  return 0;
}
```



Questo programma illustra un altro argomento molto importante: utilizzando uno spazio dei nomi non si chiude lo spazio dei nomi corrente. Quando si richiama uno spazio dei nomi, tutti i nomi in esso contenuti vengono aggiunti a quelli dello spazio dei nomi attualmente attivo. Pertanto, prima della fine del programma, sia std che CounterNamespace saranno stati aggiunti allo spazio dei nomi globale.

## Spazi dei nomi senza nome

Vi è anche un tipo particolare di spazio dei nomi, senza nome, che consente di creare identificatori univoci all'interno di un file. Gli spazi dei nomi senza nome sono anche detti spazi dei nomi anonimi. Ecco la loro forma generale:

```
namespace
  // dichiarazioni
```

Gli spazi dei nomi senza nome consentono di definire identificatori univoci noti solo nel campo d'azione di un determinato file. Questo significa che all'interno del file che contiene lo spazio dei nomi anonimo, i membri di tale spazio dei nomi possono essere utilizzati direttamente, senza alcuna qualifica mentre all'esterno del file gli identificatori risulteranno invisibili.

Gli spazi dei nomi senza nome eliminano la necessità di utilizzare il modificatore di classe static. Come descritto nel Capitolo 2, un modo per restringere il campo di visibilità di un nome globale al solo file in cui è dichiarato consiste nell'impiegare la parola riservata static. Ad esempio, si considerino i due file seguenti che fanno parte dello stesso programma.

Primo file

```
static int k;
void f1() {
  k = 99; // OK
}
```

Secondo file

```
extern int k;
void f2() {
k = 10; // errore
}
```

Poiché k è definita in Primo file, può essere utilizzata in Primo file. In Secondo file, k è specificato come extern e questo significa che non è definito, anche se ne sono noti il nome e il tipo. Quando questi due file vengono linkati, un tentativo di utilizzare k all'interno di Secondo file provoca un errore poiché k non è definito. Facendo precedere a k in Primo file il qualificatore static, il suo campo di visibilità viene ridotto unicamente al file e la variabile non risulterà visibile in Secondo file. Se il linguaggio C++ consente comunque di utilizzare dichiarazioni globali statiche, si può ottenere lo stesso effetto utilizzando uno spazio dei nomi senza nome. Ad esempio:

Primo file

```
namespace {
 int k;
}
void f1() {
 k = 99; // OK
}
```

Secondo file

```
extern int k;
void f2() {
k = 10; // errore
{
```

Anche in questo caso k è utilizzabile solo in Primo file. È preferibile utilizzare spazi di nomi anonimi piuttosto che il qualificatore static.



## Alcune opzioni degli spazi dei nomi

Possono esistere più dichiarazioni di spazi di nomi aventi lo stesso nome. Questo consente di suddividere uno spazio dei nomi su più file o anche in più aree dello stesso file. Ad esempio:

```
#include <iostream>
using namespace std;

namespace NS {
  int i;
}

// ...

namespace NS {
  int j;
}

int main()
{
  NS::i = NS::j = 10;

  // fa riferimento in particolare a NS
  cout << N::i * N::j << "\n";

  // usa il namespace NS
  using namespace NS;

  cout << i * j;

  return 0;
}
```

Questo programma produce il seguente output:

```
100
100
```

Qui, NS è stato suddiviso in due parti. Queste due parti costituiscono insieme lo spazio dei nomi NS.

Uno spazio dei nomi deve essere dichiarato all'esterno di ogni altro campo di visibilità. Questo significa che non è possibile dichiarare spazi di nomi contenuti, ad esempio, all'interno di una funzione. Vi è una sola eccezione: uno spazio dei

nomi può trovarsi all'interno di un altro spazio dei nomi. Si consideri il seguente programma:

```
#include <iostream>
using namespace std;

namespace NS1 {
  int i;
  namespace NS2  // un namespace nidificato
    int j;
  }
}

int main()
{
  NS1::i = 19;
  // NS2::j = 10; Errore, NS2 non è visibile
  NS1::NS2::j = 10; // questo va bene

  cout << N1::i << " "<< NNS1::NS2::j << "\n";

  // usa NS1
  using namespace NS1;

  /* Ora che NS1 è visibile,
     si può utilizzare NS2 per far riferimento a j. */
  cout << i * N2::j;

  return 0;
}
```

Questo programma produce il seguente output:

```
19 10
190
```

Qui lo spazio dei nomi NS2 è contenuto nello spazio dei nomi NS1. Pertanto, all'inizio del programma quando si fa riferimento a j occorre qualificarla con entrambi gli spazi dei nomi (NS1 e NS2). NS2 da solo non è sufficiente. Dopo l'istruzione:

```
using namespace NS1;
```



si può far riferimento direttamente a NS2 poiché l'istruzione using porta NS1 nel campo di visibilità.

In genere non è necessario creare spazi di nomi per programmi di piccole o medie dimensioni. Se invece si devono creare librerie riutilizzabili e si vuole garantire la più ampia trasportabilità, è opportuno considerare l'inserimento del codice in uno spazio dei nomi.

## 23.2 Lo spazio dei nomi std

Lo standard del linguaggio C++ definisce l'intera libreria standard come uno spazio dei nomi chiamato std. Questo è il motivo per il quale la maggior parte dei programmi di questo volume include la seguente istruzione:

```
using namespace std;
```

Questa provoca l'inserimento dello spazio dei nomi std nello spazio dei nomi corrente, dando accesso diretto al nome delle funzioni e delle classi definite nella libreria standard, senza dover ripetere continuamente la qualifica std::.

Naturalmente è possibile qualificare esplicitamente ogni nome con std:: se si desidera. Ad esempio il seguente programma non inserisce la libreria standard nello spazio dei nomi globale.

```
// Usa la qualifica esplicita di std::
#include <iostream>

int main()
{
  int val;

  std::cout << "Introdurre un numero: ";

  std::cin >> val;

  std::cout << "Ecco il numero: ";
  std::cout << std::hex << val;

  return 0;
}
```

Qui cout, cin e il manipolatore hex vengono qualificati esplicitamente con il loro spazio dei nomi. Questo significa che per scrivere sullo standard output si

deve specificare std::cout; per leggere dallo standard input si deve usare std::cin e per far riferimento al manipolatore esadecimale si deve usare std::hex.

Un motivo che può spingere a non portare la libreria standard C++ nello spazio dei nomi globale si verifica quando il programma utilizza la libreria in modo molto limitato. Quando invece il programma contiene centinaia di riferimenti a nomi contenuti nella libreria, è opportuno includere std nello spazio dei nomi corrente.

Se si usano solo pochi nomi della libreria standard, è più semplice specificare un'istruzione using per ciascun nome. Il vantaggio di questo approccio è il fatto che si può continuare a utilizzare tali nomi senza la qualifica std:: e senza portare l'intera libreria standard nello spazio dei nomi globale. Ad esempio:

```
// Porta solo alcuni nomi nel namespace globale
#include <iostream>

// ottiene l'accesso a cout, cin ed hex
using std::cout;
using std::cin;
using std::hex;

int main()
{
  int val;

  cout << "Introdurre un numero: ";

  cin >> val;
  cout << "Ecco il numero: ";
  cout << hex << val;
  return 0;
}
```

Qui, cin, cout ed hex possono essere utilizzati direttamente mentre gli altri nomi dello spazio dei nomi std non sono stati inseriti nello spazio dei nomi corrente.

Come si è detto, la libreria C++ originale è stata definita nello spazio dei nomi globale. Se si deve eseguire la conversione di vecchi programmi C++, si deve includere un'istruzione using namespace std oppure qualificare ogni riferimento a un membro della libreria con std::. Questo è particolarmente importante se si stanno sostituendo vecchi file header .H con le nuove versioni di tali file. Si ricordi che i vecchi header .H inseriscono il proprio contenuto nello spazio dei nomi globale; i nuovi file header inseriscono il loro contenuto nello spazio dei nomi std.



## 23.3 Creazione di funzioni di conversione

In alcune situazioni, si vuole usare un oggetto di una classe in un'espressione che prevede altri tipi di dati. In genere per eseguire questa operazione possono essere utilizzate delle funzioni operatore modificate tramite overloading. Tuttavia, in altri casi, ciò che si desidera è una semplice conversione di tipo dalla classe al tipo di destinazione. Per gestire questi casi, il linguaggio C++ consente di creare funzioni di conversione personalizzate. Una funzione di conversione converte la classe in un tipo compatibile con quello del resto dell'espressione. Ecco la forma generale di una funzione di conversione:

operator *tipo*( ) return *valore*;

Qui, tipo è il tipo di destinazione al quale si deve convertire la classe e valore è il valore della classe dopo la conversione. Le funzioni di conversione restituiscono dati di tipo tipo e non è possibile utilizzare altri specificatori di tipo. Inoltre non è possibile includere parametri. Le funzioni di conversione sono ereditate e possono essere virtuali.

La seguente funzione di conversione utilizza la classe stack sviluppata nel Capitolo 11. Si supponga di voler utilizzare gli oggetti di tipo stack all'interno di un'espressione intera. Inoltre si supponga che il valore di un oggetto stack utilizzato in un'espressione intera sia il numero di valori attualmente contenuti nello stack (questa situazione può essere utile se, ad esempio, si usano oggetti stack in una simulazione e si deve monitorare la rapidità con la quale lo stack si riempie). Un modo per risolvere questa situazione consiste nel convertire un oggetto di tipo stack in un intero che rappresenti il numero di elementi contenuti nello stack. Per ottenere questo risultato, si può usare una funzione di conversione avente il seguente aspetto:

```
operator int() { return tos; }
```

Ecco un programma che illustra il funzionamento delle funzioni di conversione:

```
#include <iostream>
using namespace std;

const int SIZE=100;

// crea la classe stack
class stack {
  int stck[SIZE];
  int tos;
public:
```

```
  stack() { tos=0; }
  void push(int i);
  int pop(void);
  operator int() { return tos; } // conversione di stack in int
};

void stack::push(int i)
{
  if(tos==SIZE) {
    cout << "Stack esaurito.\n";
    return;
  }
  stck[tos] = i;
  tos++;
}

int stack::pop()
{
  if(tos==0) {
    cout << "Underflow dello stack.\n";
    return 0;
  }
  tos--;
  return stck[tos];
}

int main()
{
  stack stck;
  int i, j;

  for(i=0; i<20; i++)  stck.push(i);

  j = stck; // converte in int

  cout << j << " elementi nello stack.\n";

  cout << SIZE - stck << " spazi disponibili.\n";
  return 0;
}
```

Il programma produce il seguente output:

```
20 elementi nello stack.
80 spazi disponibili.
```



Come si può vedere dal programma, quando un oggetto stack viene utilizzato in un'espressione intera, come j=stck, all'oggetto viene applicata la funzione di conversione. In questo specifico caso, la funzione di conversione restituisce il valore 20. La funzione di conversione viene chiamata anche quando stck viene sottratto da SIZE.

Ecco un altro esempio di una funzione di conversione. Questo programma crea una classe chiamata pwr() che conserva e calcola il risultato di alcuni numeri elevati a una potenza. Il risultato viene memorizzato come double. Fornendo una funzione di conversione nel tipo double e restituendo il risultato, si possono utilizzare oggetti di tipo pwr nelle espressioni che impiegano altri valori double.

```
#include <iostream>
using namespace std;

class pwr {
  double b;
  int e;
  double val;
public:
  pwr(double base, int exp);
  pwr operator+(pwr o) {
    double base;
    int exp;
    base = b + o.b;
    exp = e + o.e;

    pwr temp(base, exp);
    return temp;
  }
  operator double() { return val; } // converte in double
};

pwr::pwr(double base, int exp)
{
  b = base;
  e = exp;
  val = 1;
  if(exp==0) return;
  for( ; exp>0; exp--) val = val * b;
}

int main()
{
  pwr x(4.0, 2);
  double a;
```

```
  a = x; // converte in double
  cout << x + 100.2; // converte x in double e somma 100.2
  cout << "\n";

  pwr y(3.3, 3), z(0, 0);

  z = x + y;  // nessuna conversione
  a = z;  // converte in double
  cout << a;

  return 0;
}
```

Ecco l'output del programma.

```
116.2
20730.7
```

Come si può vedere, quando x viene utilizzata nell'espressione x+100.2, per produrre il valore double viene utilizzata la funzione di conversione. Si noti anche che nell'espressione x+y non viene applicata la funzione di conversione in quanto l'espressione considera solo oggetti di tipo pwr.

Come si può immaginare dagli esempi precedenti, vi sono molte situazioni in cui è conveniente creare una funzione di conversione per una classe. Spesso le funzioni di conversione offrono una sintassi più naturale quando si lavora con oggetti appartenenti a una classe e oggetti di un tipo standard C++. In particolare, nel caso della classe pwr, la disponibilità della conversione in double consente di utilizzare gli oggetti di tale classe nelle normali espressioni matematiche e questo semplifica la programmazione e la comprensibilità del programma.

È anche possibile creare funzioni di conversione differenti a seconda delle esigenze. Ad esempio si possono definire funzioni di conversione in double o in long. Verrà automaticamente applicata l'una o l'altra funzione a seconda del tipo di ciascuna espressione.

## 23.4 Funzioni membro const e mutable

Le funzioni membro di una classe possono essere dichiarate const e in questo caso this verrà trattato come un puntatore const. Pertanto tali funzioni non potranno modificare l'oggetto che le richiama. Inoltre un oggetto const non può richiamare una funzione membro non-const. Tuttavia, una funzione membro const può essere richiamata da oggetti const o non-const.



Per specificare che una funzione membro è const, si utilizza la forma illustrata nel seguente esempio.

```
class X {
  int some_var;
public:
  int f1() const; // funzione membro const
};
```

Come si può vedere, la parola riservata const segue la dichiarazione dei parametri della funzione.

Si dichiara una funzione membro come const per evitare che modifichi l'oggetto che la richiama. Ad esempio, si consideri il seguente programma.

```
/*
   Illustra l'uso delle funzioni membro const.
   Questo programma non è accettato dal compilatore.
*/
#include <iostream>
using namespace std;

class Demo {
  int i;
public:
  int geti() const {
    return i; // ok
}

  void seti(int x) const {
    i = x; // errore!
  }
};

int main()
{
  Demo ob;

  ob.seti(1900);
  cout << ob.geti();

  return 0;
}
```

Questo programma non può essere compilato poiché seti() è dichiarata const. Questo significa che non può modificare l'oggetto che l'ha richiamata. Poiché tenta di modificare i, il programma è interrotto. Al contrario, poiché geti() non tenta di modificare i, è perfettamente accettabile.

Talvolta si desidera che solo uno o più membri di una classe possano essere modificati da una funzione const ma che la funzione non possa modificare tutti gli altri membri. Si può ottenere questo effetto utilizzando la parola riservata mutable che consente di ignorare la dichiarazione const. Questo significa che un membro mutable può essere modificato anche da una funzione membro const. Ad esempio:

```
// Illustra l'uso di mutable.
#include <iostream>
using namespace std;

class Demo {
  mutable int i;
  int j;
public:
  int geti() const {
    return i; // ok
  }

  void seti(int x) const {
    i = x; // ora è OK.
  }

/* La funzione seguente non può essere compilata.
  void setj(int x) const {
    j = x; // ERRORE!
  }
*/
};

int main()
{
  Demo ob;

  ob.seti(1900);
  cout << ob.geti();

  return 0;
}
```



Qui, i è specificata come mutable e dunque può essere modificata dalla funzione seti(). Al contrario, j che non è mutable non può essere modificata da setj().

## 23.5 Funzioni membro volatile

Le funzioni membro di una classe possono essere dichiarate volatile e questo fa in modo che il puntatore this venga trattato come un puntatore volatile. Per specificare una funzione membro come volatile, si deve utilizzare la forma illustrata nel seguente esempio:

```
class X {
public:
  void f2(int a) volatile; // funzione membro volatile
};
```

## 23.6 Costruttori espliciti

Come si è detto nel Capitolo 12, ogni volta si ha un costruttore che richiede un solo argomento, per inizializzare l'oggetto si può usare ob(x) oppure ob = x. Il motivo è che quando si crea un costruttore che accetta un argomento si sta anche creando implicitamente una conversione dal tipo dell'argomento al tipo della classe. Ma talvolta si desidera che questa conversione automatica non abbia luogo. Per questo motivo il linguaggio C++ definisce la parola riservata explicit. Per comprendere il suo effetto, si consideri il seguente programma.

```
#include <iostream>
using namespace std;

class myclass {
  int a;
public:
  myclass(int x) { a = x; }
  int geta() { return a; }
};

int main()
{
 myclass ob = 4; // convertita automaticamente in myclass(4)

  cout << ob.geta();
```

```
  return 0;
}
```

Qui il costruttore di myclass accetta un parametro. Si faccia attenzione al modo in cui ob è dichiarata in main(). L'istruzione:

```
myclass ob = 4; // convertita automaticamente in myclass(4)
```

viene automaticamente convertita in una chiamata al costruttore di myclass dove l'argomento è 4. Pertanto l'istruzione precedente viene gestita dal compilatore come se fosse scritta nel seguente modo:

```
myclass ob(4);
```

Se non si vuole che avvenga questa conversione implicita, basta utilizzare lo specificatore esplicito il quale può essere applicato solo a costruttori. Il costruttore explicit verrà utilizzato solo quando l'inizializzazione utilizza la normale sintassi dei costruttori mentre non verrà impiegato per eseguire conversioni automatiche. Ad esempio, dichiarando come explicit il costruttore di myclass, non verrà utilizzata la conversione automatica. Ecco un esempio di myclass() dichiarata in modo explicit.

```
#include <iostream>
using namespace std;

class myclass {
  int a;
public:
  explicit myclass(int x) { a = x; }
  int geta() { return a; }
};
```

Ora saranno consentiti solo costruttori nella forma:

```
myclass ob(4);
```

mentre istruzioni come la seguente:

```
myclass ob = 4; // Errore
```

saranno errate.



## 23.7 Uso della parola riservata asm

Anche se il linguaggio C++ è molto flessibile e potente, vi sono alcune situazioni molto specializzate che il linguaggio non è in grado di gestire. Ad esempio il linguaggio C++ non contiene istruzioni in grado di disattivare gli interrupt. Per gestire queste situazioni, il linguaggio offre una sorta di via di fuga che consente di impiegare direttamente il codice assembler bypassando completamente il compilatore C++. Questa via di fuga è rappresentata dall'istruzione asm. Con asm è possibile inserire il codice assembler direttamente all'interno del programma C++. Questo codice assembler verrà compilato senza alcuna modifica e diverrà una parte integrante del codice del programma nel punto esatto in cui è stata inserita la parola riservata asm.

Ecco la forma generale dell'istruzione asm:

asm ("*op-code*")

Dove *op-code* è l'istruzione assembler che deve essere inserita nel programma. Molti compilatori consentono di impiegare anche le seguenti forme di asm:

asm *istruzione* ;
asm *istruzione newline*
asm
*sequenza di istruzioni*

Qui, *istruzione* e un'istruzione assembler. Data la natura di asm (che è strettamente legata all'implementazione) per informazioni sul suo uso è opportuno consultare la documentazione del compilatore.

Attualmente, il Microsoft Visual C++ utilizza a tale scopo la parola riservata __asm che ha un funzionamento simile ad asm.

Ecco un esempio semplice e sicuro d'uso della parola riservata asm:

```
#include <iostream>
using namespace std;

int main()
{
  asm int 5; // genera l'intertupt 5

  return 0;
}
```

Se utilizzato in DOS, questo programma genera un'istruzione INT 5 che provoca la stampa del contenuto dello schermo.

**ATTENZIONE** *Per impiegare l'istruzione asm è necessario conoscere approfonditamente il funzionamento del linguaggio assembler. Chiunque non sia particolarmente esperto in tale linguaggio non dovrebbe usare questa parola riservata per evitare di incorrere in gravi errori.*

## 23.8 Specifiche di linking

In C++ è possibile specificare il modo in cui una funzione viene linkata nel programma. Normalmente le funzioni vengono linkate secondo lo stile C++. Le specifiche di linking consentono di linkare una funzione utilizzando le forme previste da altri tipi di linguaggi. La forma generale dello specificatore di linking è:

extern "*linguaggio*" *prototipo-di-funzione*

Dove *linguaggio* indica il linguaggio desiderato. Tutti i compilatori C++ supportano il linking C e C++. Alcuni consentono anche di utilizzare gli specificatori Fortran, Pascal o BASIC (a tale proposito è opportuno consultare la documentazione del compilatore).

```
#include <iostream>
using namespace std;

extern "C" void myCfunc();

int main()
{
  myCfunc();

  return 0;
}

// Questa verrà linkata come una funzione C.
void myCfunc()
{
  cout << "Questa verrà linkata come una funzione C.\n";
}
```

**NOTA** *La parola riservata extern è una parte integrante delle specifiche di linking. Inoltre le specifiche di linking devono essere globali, ovvero non possono essere utilizzate all'interno di una funzione.*



Tramite questa forma è possibile specificare più di una funzione:

```
extern "linguaggio" {
 prototipi
}
```

## 23.9 Operazioni di I/O su array

Oltre alle operazioni di I/O su console e su file, il sistema di I/O a stream del C++ consente anche di eseguire operazioni di I/O su array. Le operazioni di I/O su array utilizzano array di caratteri come dispositivo di input, dispositivo di output o entrambi. Le operazioni di I/O su array vengono eseguite tramite normali stream C++. In pratica, alle operazioni di I/O su array si applica tutto ciò che già si sa sulle operazioni di I/O C++. L'unica differenza consiste nel fatto che il dispositivo collegato allo stream è un array di caratteri. Gli stream collegati ad array di caratteri vengono normalmente chiamati stream char *. Per utilizzare nei programmi le operazioni di I/O su array, si deve includere il file <strstream>.

**NOTA** *Le classi per stream basati su caratteri descritte in questa sezione sono sconsigliate dallo standard per il C++. Anche se rimangono valide, ne è sconsigliato l'uso per realizzare nuovo codice. Questa breve discussione è stata inclusa sono perché tali operazioni sono state ampiamente utilizzate.*

### Classi basate su array

Le classi di I/O su array sono istrstream, ostrstream e strstream. Queste classi sono utilizzate per creare rispettivamente stream di input, di output e di input/output. Inoltre la classe istrstream deriva da istream, la classe ostrstream deriva ostream e la classe strstream deriva da iostream. Pertanto tutte le classi basate su array derivano indirettamente da ios ed hanno accesso alle stesse funzioni membro delle normali classi di I/O.

### Creazione di uno stream di output su array

Per eseguire un'operazione di output su un array, si deve collegare tale array a uno stream utilizzando questo costruttore ostrstream:

ostrstream *ostr*(char **buf*, streamsize *size*, openmode *mode*=ios::out);

Qui, *buf* è un puntatore all'array e verrà utilizzato per raccogliere i caratteri scritti sullo stack *ostr*. Le dimensioni dell'array vengono passate tramite il parametro *size*. L'impostazione predefinita prevede che lo stream venga aperto per operazioni di normale output ma è possibile applicare tramite OR varie opzioni per creare la modalità desiderata. Ad esempio si può includere ios::app per fare in modo che l'output venga scritto alla fine delle informazioni contenute nell'array. Nella maggior parte dei casi viene impiegata l'impostazione predefinita.

Dopo aver aperto uno stream di output su array, tutto l'output inviato allo stream verrà inserito nell'array. Tuttavia non è possibile scrivere oltre i limiti dell'array. Un tentativo in questo senso provocherà un errore.

Ecco un semplice programma che dimostra il funzionamento di uno stream di output su array.

```
#include <strstream>
#include <iostream>
using namespace std;

int main()
{
  char str[80];

  ostrstream outs(str, sizeof(str));

  outs << "Operazioni di I/O su array in C++. ";
  outs << 1024 << hex << " ";
  outs.setf(ios::showbase);
  outs << 100 << ' ' << 99.789 << ends;

  cout << str;  // visualizza la stringa sulla console

  return 0;
}
```

Questo programma visualizza il seguente risultato:

```
Operazioni di I/O su array in C++. 1024 0x64 99.789
```

Si deve tenere in considerazione che outs è uno stream come un altro; dunque ha le stesse funzionalità di ogni altro tipo di stream di cui si è parlato in precedenza. L'unica differenza consiste nel fatto che il dispositivo collegato è un array di caratteri. Poiché outs è uno stream, è possibile utilizzare i manipolatori hex ed ends. Sono disponibili anche le funzioni membro di ostream, come ad esempio setf().



Questo programma chiude manualmente l'array utilizzando il manipolatore ends. Il fatto che l'array venga automaticamente chiuso con un NULL o meno dipende dall'implementazione, dunque è sempre opportuno eseguire una chiusura manuale dell'array.

Si può determinare il numero di caratteri inviati in output all'arresto richiamando la funzione membro pcount(). Tale funzione ha il seguente prototipo:

```
streamsize pcount( );
```

Il numero restituito da pcount() include anche il carattere di chiusura NULL, sempre che questo esista.

Il seguente programma illustra l'uso di pcount(). Il programma mostra che outs contiene 18 caratteri: 17 più il carattere nullo.

```
#include <strstream>
#include <iostream>
using namespace std;

int main()
{
  char str[80];

  ostrstream outs(str, sizeof(str));

  outs << "abcdefg ";
  outs << 27 << " " << 890.23;
  outs << ends;   // chiude la stringa con il carattere nullo

  cout << outs.pcount(); // numero di caratteri contenuti in outs

  cout << " " << str;

  return 0;
}
```

## Uso di un array di input

Per collegare uno stream di input a un array, si usa il seguente costruttore istrstream:

```
istrstream istr(const char *buf);
```

Qui, *buf* è un puntatore all'array che verrà utilizzato come origine dei caratteri ogni volta che verrà eseguito dell'input dallo stream *istr*. Il contenuto dell'array

puntato da *buf* deve essere chiuso dal carattere nullo. Ciononostante il carattere nullo finale non viene mai letto dall'array.

Ecco un programma di esempio che utilizza per l'input una stringa.

```
#include <iostream>
#include <strstream>
using namespace std;

int main()
{
  char s[] = "10 Hello 0x75 42.73 OK";

  istrstream ins(s);

  int i;
  char str[80];
  float f;

  // legge: 10 Hello
  ins >> i;
  ins >> str;
  cout << i << " " << str << endl;

  // legge 0x75 42.73 OK
  ins >> hex >> i;
  ins >> f;
  ins >> str;

  cout << hex << i << " " << f << " " << str;

  return 0;
}
```

Se si vuole utilizzare come input solo una parte di una stringa, si deve utilizzare la seguente forma del costruttore istrstream:

istrstream *istr*(const char **buf*, streamsize *size*);

In questo caso verranno utilizzati solo i primi *size* elementi dell'array puntato da *buf*. Questa stringa non deve essere chiusa dal carattere nullo poiché è il valore di *size* a determinare le sue dimensioni.

Gli stream collegati ad aree di memoria si comportano esattamente come quelli collegati ad altri dispositivi. Ad esempio, il seguente programma illustra la lettura del contenuto di un array di testo. Quando viene raggiunta la fine dell'array (corrispondente alla fine del file), ins sarà falso.



```
/* Questo programma illustra la lettura del contenuto
   di un array contenente testo. */
#include <iostream>
#include <strstream>
using namespace std;

int main()
{
  char s[] = "10.23 questo è testo <<>><<?!\n";

  istrstream ins(s);

  char ch;

  /* legge e visualizza il contenuto
     di un array di testo. */

  ins.unsetf(ios::skipws); // non salta gli spazi
  while (ins)  // false al raggiungimento della fine dell'array
    ins >> ch;
    cout << ch;
}

  return 0;
}
```

## Stream di input/output su array

Per creare uno stream basato su array che esegua operazioni di input e output, si utilizza la funzione costruttore strstream:

strstream *iostr*(char **buf*, streamsize *size*, openmode *mode* = ios::in | ios::out);

Qui *buf* punta alla stringa che verrà utilizzata per le operazioni di I/O. Il valore di *size* specifica le dimensioni dell'array. Il valore di *mode* determina la modalità di funzionamento dello stream *iostr*. Per normali operazioni di I/O, mode sarà iios::in | ios::out. Per operazioni di input, l'array deve essere chiuso dal carattere nullo.

Ecco un programma che utilizza un array per eseguire operazioni di input e output.

```
// Esegue sia l'input che l'output.
#include <iostream>
#include <strstream>
using namespace std;

int main()
{
  char iostr[80];

  strstream strio(iostr, sizeof(iostr), ios::in | ios::out);

  int a, b;
  char str[80];

  strio << "10 20 test ";
  strio >> a >> b >> str;
  cout << a << " " << b << " " << str << endl;

  return 0;
}
```

Questo programma scrive sull'array e poi ne rilegge il contenuto.

## 23.10 Uso di array dinamici

Negli esempi precedenti, quando si è collegato uno stream a un array di output, al costruttore di ostrstream venivano passati l'array e le sue dimensioni. Questo approccio funziona quando si conosce il numero massimo di caratteri che verranno inviati all'array. Ma cosa accade se non si conoscono le dimensioni esatte che dovrà assumere l'array di output? La soluzione a questo problema è rappresentata dalla seconda forma del costruttore di ostrstream:

```
ostrstream( );
```

Quando si usa questo costruttore, ostrstream crea e gestisce un array allocato in modo dinamico le cui dimensioni crescono automaticamente in modo da accogliere l'output inviato.

Per accedere all'array allocato dinamicamente si deve usare una seconda funzione, chiamata str(), che ha il seguente prototipo:

```
char *str( );
```



Questa funzione "congela" l'array e restituisce un puntatore ad esso. Il puntatore restituito da str() consente di accedere all'array dinamico come se fosse una stringa. Dopo che un array dinamico viene congelato, non può più essere utilizzato per operazioni di output a meno che non venga "scongelato". Pertanto è opportuno non congelare l'array mentre gli vengono inviati caratteri.

Ecco un programma che utilizza un array dinamico.

```
#include <strstream>
#include <iostream>
using namespace std;

int main()
{
  char *p;

  ostrstream outs;  // alloca dinamicamente un array

  outs << "Operazioni di I/O su array in C++ ";
  outs << -10 << hex << " ";
  outs.setf(ios::showbase);
  outs << 100 << ends;

  p = outs.str(); // Congela il buffer dinamico
                  // e restituisce un puntatore ad esso.

  cout << p;

  return 0;
}
```

Le operazioni di I/O su array dinamici possono essere utilizzate anche con la classe strstream che consente di eseguire simultaneamente operazioni di input e output.

Per congelare o scongelare un array dinamico si richiama la funzione freeze() il cui prototipo è:

```
void freeze(bool action = true);
```

Se *action* è true, l'array viene congelato mentre se è false, l'array viene scongelato.

## 23.11 Uso di I/O binario con stream basati su array

Come si è detto, le operazioni di I/O basate su array offrono tutte le funzionalità delle normali operazioni di I/O. Pertanto gli array collegati a stream possono contenere anche informazioni binarie. Per determinare la fine dell'array quando si leggono informazioni binarie si deve utilizzare la funzione eof(). Ad esempio, il seguente programma mostra come leggere il contenuto di un qualsiasi array (binario o di testo) utilizzando la funzione get().

```
#include <iostream>
#include <strstream>
using namespace std;

int main()
{
  char *p = "questo è testo";

  istrstream ins(p);

  char ch;

  // legge e visualizza informazioni binarie
  while (!ins.eof()) {
    ins.get(ch);
    cout << hex << (int) ch << ' ';
  }

  return 0;
}
```

In questo esempio, i valori formati da \1\2\3 e così via sono valori non stampabili.

Per produrre in output caratteri binari si usa la funzione put(). Per leggere buffer di dati binari si usa la funzione membro read(). Per scrivere buffer di dati binari si usa la funzione write().

## 23.12 Riepilogo delle differenze esistenti fra C e C++

Lo standard C++ è fondamentalmente un'estensione dello standard C e praticamente ogni programma C è anche un programma C++. Tuttavia esistono alcune differenze che sono state l'argomento della Parte 1 e della Parte 2 di questo volume. Ecco le differenze più importanti.



In C++ le variabili locali possono essere dichiarate in qualsiasi punto di un blocco. In C devono essere dichiarate all'inizio di un blocco, prima di ogni istruzione "attiva".

In C, una funzione dichiarata nel seguente modo:

```
int f();
```

non dice nulla sui parametri della funzione. Quando fra le parentesi non viene specificato nulla, in C significa che non viene stabilito nulla rispetto ai parametri della funzione. Potrebbero esservi parametri oppure no. In C++ una dichiarazione di funzione come questa significa che la funzione non ha parametri. Pertanto in C++ queste due dichiarazioni sono equivalenti:

```
int f();

int f(void);
```

In C++, void in un elenco di parametri è opzionale. Molti programmatori C++ includono void per indicare a chiunque leggerà il programma che una funzione non usa parametri; tuttavia la presenza di void, tecnicamente, non è necessaria.

In C++ tutte le funzioni devono avere un prototipo. In C questo non è un requisito (anche se è buona abitudine specificare tutti i prototipi anche in un programma C).

Una differenza piccola ma potenzialmente importante fra C e C++ è il fatto che in C una costante carattere viene trasformata automaticamente in un intero. In C++ questo non avviene.

In C non è un errore dichiarare più volte una variabile globale, anche se questa non è considerata una buona pratica di programmazione. In C++ questo rappresenta un errore.

In C un identificatore deve avere almeno 31 caratteri significativi. In C++ tutti i caratteri sono significativi. Tuttavia, per motivi pratici, è opportuno evitare di impiegare identificatori così lunghi.

In C è possibile (anche se raro) richiamare main() dal programma. Il linguaggio C++ non consente questa operazione.

In C non si può richiedere l'indirizzo di una variabile register. In C++ questo è consentito.

In C, quando non viene indicato alcuno specificatore di tipo nelle estensioni di dichiarazione, viene utilizzato per default il tipo int. Questa regola non viene più applicata in C++ (e probabilmente verrà eliminata anche dalle prossime versione del C).

Capitolo 24

# Introduzione alla libreria STL


24.1 **Introduzione all'uso della libreria STL**
24.2 **Le classi container**
24.3 **Funzionamento generale**
24.4 **I vettori**
24.5 **Le liste**
24.6 **Le mappe**
24.7 **Gli algoritmi**
24.8 **Uso degli oggetti funzione**
24.9 **La classe string**
24.10 **Commenti finali sulla libreria STL**


Questo capitolo introduce un argomento che da molti viene considerato la più importante aggiunta al linguaggio C++ degli ultimi anni: la libreria STL (Standard Template Library). La creazione della libreria STL è stato uno degli sforzi più ingenti svoltisi durante la standardizzazione del C++. Tale libreria fornisce classi template di utilizzo generale, funzioni che implementano algoritmi e strutture dati di uso comune comprendendo, ad esempio, il supporto di vettori, liste, code e stack. Inoltre la libreria STL definisce varie routine di accesso a tali elementi. Poiché la libreria STL si basa su classi template, gli algoritmi e le strutture dati possono essere applicate praticamente a ogni tipo di dati. La libreria STL è un elemento software complesso che utilizza alcune delle funzionalità più avanzate del linguaggio C++. Per comprendere e utilizzare al meglio la libreria STL è dunque opportuno conoscere al meglio il linguaggio C++ e soprattutto il funzionamento dei puntatori, degli indirizzi e dei template. Francamente, la sintassi template che descrive la libreria può inizialmente intimorire (anche se il tutto sembra molto più complicato di quanto non sia in realtà). Anche se in questo capitolo non vengono introdotti argomenti più complessi di quelli trattati in precedenza, non ci si deve sorprendere o scoraggiare se inizialmente si trova che l'argomento sembra essere confuso. Basta pazientare e studiare gli esempi evitando che la sintassi poco familiare distragga dalla semplicità di base della libreria.

Lo scopo di questo capitolo è quello di presentare una panoramica della libreria STL partendo dalla filosofia progettuale, dall'organizzazione, dagli elementi

di base e dalle tecniche di programmazione. Poiché la libreria STL è molto estesa, non è possibile discuterne in un capitolo tutte le funzionalità. Per questo motivo la Parte quarta rappresenta una guida di riferimento completa alla libreria.

Questo capitolo descrive anche una delle nuove classi più importanti del C++: la classe string. Tale classe definisce un tipo di dati che consente di lavorare con stringhe di caratteri come con qualsiasi altro tipo di dati, ovvero tramite gli operatori. La classe string è strettamente correlata alla libreria STL.

## 24.1 Introduzione all'uso della libreria STL

Anche se la libreria STL è molto estesa e la sua sintassi può inizialmente spaventare, è molto facile utilizzarla se si comprende il modo in cui è costruita e quali sono gli elementi da essa impiegati. Pertanto, prima di offrire degli esempi di codice, è opportuno presentare una panoramica della libreria STL.

Il nucleo della libreria STL è costituito da tre elementi: *container*, *algoritmi* e *iteratori*. Questi elementi collaborano fra loro per fornire soluzioni pronte all'uso per vari problemi di programmazione.

### I container

I *container* sono oggetti che contengono altri oggetti; la libreria offre container di vari tipi. Ad esempio, la classe vector definisce un array dinamico, deque crea una coda a doppio concatenamento e list fornisce una lista lineare. Questi container sono chiamati container sequenziali poiché nella terminologia STL una sequenza è una lista lineare. Oltre ai container di base, la libreria definisce anche dei container associativi che consentono di ricercare in modo efficiente un valore sulla base di una chiave. Ad esempio, un container map fornisce l'accesso a valori tramite chiavi univoche. Pertanto una mappa memorizza coppie chiave/valore e consente di trovare un valore sulla base della sua chiave.

Ogni classe container definisce una serie di funzioni specifiche per il container stesso. Ad esempio, un container per liste include le funzioni di inserimento, cancellazione e unione degli elementi. Un container stack include le funzioni di inserimento (push) ed estrazione (pop) dei valori.

### Gli algoritmi

Gli *algoritmi* operano su container. Essi rappresentano il mezzo tramite il quale è possibile manipolare il contenuto dei container. Fra le funzionalità offerte dagli algoritmi vi è dell'inizializzazione, l'ordinamento, la ricerca e la trasformazione del contenuto dei container. Molti algoritmi operano su un intervallo di elementi di un container.



### Gli iteratori

Gli *iteratori* sono oggetti strettamente imparentati con i puntatori. Essi danno la possibilità di attraversare il contenuto di un container un po' come un puntatore consente di attraversare un array. Vi sono cinque tipi di iteratori.

| ITERATORE | ACCESSI CONSENTITI |
|---|---|
| Accesso diretto | Memorizzazione e lettura dei valori. L'accesso agli elementi può avvenire in modo diretto. |
| Bidirezionale | Memorizzazione e lettura dei valori. Spostamento in avanti e all'indietro. |
| Avanti | Memorizzazione e lettura dei valori. Spostamento solo in avanti. |
| Input | Lettura ma non memorizzazione dei valori. Spostamento solo in avanti. |
| Output | Memorizzazione ma non lettura dei valori. Spostamento solo in avanti. |

In generale, un iteratore che ha maggiori capacità di accesso può essere utilizzato in luogo di uno che è dotato di minori capacità di accesso. Ad esempio, al posto di un iteratore di input si può utilizzare un iteratore in avanti.

Gli iteratori vengono gestiti come i puntatori. Dunque è possibile incrementarli e decrementarli. Agli iteratori si può applicare l'operatore *. Gli iteratori vengono dichiarati utilizzando il tipo iterator definito dai vari container.

La libreria STL supporta anche gli iteratori inversi. Gli iteratori inversi sono iteratori bidirezionali o ad accesso diretto che si muovono lungo una sequenza in direzione inversa. Pertanto, se un iteratore inverso punta alla fine di una sequenza, incrementando tale iteratore si farà in modo che esso punti al penultimo elemento.

Quando parla dei vari tipi di iteratori nelle descrizioni dei template, questo volume usa i seguenti termini.

| TERMINE | DESCRIZIONE |
|---|---|
| `BiIter` | Iteratore bidirezionale |
| `ForIter` | Iteratore in avanti |
| `InIter` | Iteratore di input |
| `OutIter` | Iteratore di output |
| `RandIter` | Iteratore di accesso diretto |

### Altri elementi del libreria STL

Oltre ai container, agli algoritmi e agli iteratori, la libreria STL si basa su molti altri componenti standard. Fra i principali vi sono gli allocatori, i predicati, le funzioni di confronto e ogni oggetto funzione.

Per ogni container è definito un allocatore. Gli allocatori gestiscono l'allocazione della memoria per un container. L'allocatore standard è un oggetto della classe allocator ma per applicazioni particolari è possibile definire allocatori specifici. Per la maggior parte degli usi si impiega l'allocatore standard.

Molti degli algoritmi e container usano tipi speciali di funzioni chiamati predicati. Vi sono due tipi di predicati: unari e binari. Un predicato *unario* accetta un argomento mentre un predicato *binario* ne accetta due. Queste funzioni restituiscono un risultato true o false. Le condizioni che provocano la restituzione di un valore true o false sono definite dal programmatore. Nella parte rimanente del capitolo, quando è necessario utilizzare una funzione predicato unaria, si fa riferimento al tipo UnPred. Quando invece è necessario un predicato binario, verrà utilizzato il tipo BinPred. In un predicato binario, gli argomenti sono sempre in ordine (primo, secondo). Per predicati unari e binari gli argomenti contengono valori appartenenti al tipo degli oggetti contenuti nel container.

Alcuni algoritmi e alcune classi usano un tipo particolare di predicato binario che confronta due elementi. Le funzioni di confronto restituiscono true se il primo argomento è minore del secondo. Le funzioni di confronto saranno identificate dal tipo Comp.

Oltre agli header richiesti dalle varie classi STL, la libreria standard C++ include i file header <utility> e <functional> che forniscono il supporto per la libreria STL. Ad esempio, la classe template pair che contiene una coppia di valori è definita in <utility>. Si proverà a utilizzare pair più avanti in questo stesso capitolo.

I template contenuti in <functional> aiutano a costruire oggetti che definiscono operator(). Si tratta di oggetti funzione che possono essere utilizzati al posto dei puntatori a funzione. In <functional> sono dichiarati vari oggetti funzione. Eccone alcuni:

| | | | | |
|---|---|---|---|---|
| plus | minus | multiplies | divides | modulus |
| negate | equal_to | not_equal_to | greater | greater_equal |
| less | less_equal | logical_and | logical_or | logical_not |

L'oggetto funzione probabilmente più utilizzato è less che determina quando un oggetto è minore di un altro. Gli oggetti funzione possono essere utilizzati al posto dei puntatori a funzione negli algoritmi STL descritti più avanti. Utilizzando oggetti funzione al posto dei puntatori a funzione, la libreria STL è in grado di generare codice più efficiente.



Due entità presenti nella libreria STL sono i *binder* e i *negatori*. Un binder collega un argomento a un oggetto funzione. Un negatore restituisce il complemento di un predicato.

Un ultimo termine da introdurre è *adattatore*. In termini STL, un adattatore trasforma una cosa in un'altra. Ad esempio, il container queue (che crea una coda standard) è un adattatore per il container deque.

## 24.2 Le classi container

Come si è detto, i container sono oggetti STL che memorizzano più dati. La Tabella 24.1 elenca i container definiti dalla libreria STL. In tale tabella sono anche indicati i file header necessari per utilizzare ciascun container. Anche la classe string, che gestisce stringhe di caratteri, è un container ma verrà discusso più avanti in questo stesso capitolo.

Poiché il nome del tipo generico nella dichiarazione di una classe template è arbitrario, le classi container dichiarano delle versioni di questi tipi tramite typedef. In questo modo il nome dei tipi assume un significato concreto.

**Tabella 24.1** I container definiti dalla libreria STL.

| CONTAINER | DESCRIZIONE | HEADER RICHIESTO |
|---|---|---|
| `bitset` | Un gruppo di bit | `<bitset>` |
| `deque` | Una coda a doppio concatenamento | `<deque>` |
| `list` | Una lista lineare | `<list>` |
| `map` | Una mappa chiave/valore in cui a una chiave è associato un solo valore | `<map>` |
| `multimap` | Una mappa chiave/valore in cui a una chiave possono essere associati due o più valori | `<list>` |
| `multiset` | Un insieme in cui ogni elemento non necessariamente è univoco | `<set>` |
| `priority_queue` | Una coda a priorità | `<queue>` |
| `queue` | Una coda | `<queue>` |
| `set` | Un insieme in cui ogni elemento è univoco | `<set>` |
| `stack` | Uno stack | `<stack>` |
| `vector` | Un array dinamico | `<vector>` |

Ecco alcuni dei nomi più utilizzati fra quelli definiti.

| | |
|---|---|
| size_type | Un tipo di intero |
| reference | L'indirizzo di un elemento |
| const_reference | L'indirizzo cconst di un elemento |
| iterator | Un iteratore |
| const_iterator | Un iteratore const |
| reverse_iterator | Un iteratore inverso |
| const_reverse_iterator | Un iteratore const inverso |
| value_type | Il tipo del valore contenuto in un container |
| allocator_type | Il tipo dell'allocatore |
| key_type | Il tipo di una chiave |
| key_compare | Il tipo di una funzione che confronta due chiavi |
| value_compare | Il tipo di una funzione che confronta due valori |

## 24.3 Funzionamento generale

Anche se le operazioni interne svolte dalla libreria STL sono molto sofisticate, l'uso della libreria è piuttosto semplice. Innanzitutto si deve decidere il tipo di container che si vuole usare. Ognuno di essi offre vantaggi e costringe a scendere a compromessi. Ad esempio, un vettore è ottimo quando è necessario usare una specie di array con accesso diretto e quando non vengono eseguite troppe operazioni di inserimento o cancellazione. Una lista offre agevoli operazioni di inserimento e cancellazione al prezzo di una certa lentezza. Una mappa fornisce un container associativo ma questo ha un costo in termini elaborativi.

Dopo aver scelto un container, per aggiungervi elementi, per accedere o modificare tali elementi o per cancellare elementi si devono usare le sue funzioni membro. Ad eccezione di bitset, un container cresce automaticamente e si riduce quando gli elementi vengono rimossi.

Gli elementi possono essere aggiunti e rimossi da un container in vari modi. Ad esempio sia i container sequenziali (vector, list e deque) che i container associativi (map, multimap, set e multiset) forniscono una funzione membro chiamata insert() che inserisce elementi in un container e una funzione erase() che rimuove gli elementi da un container. I container sequenziali forniscono anche le funzioni push_back() e push_front() che aggiungono un elemento rispettivamente alla fine o all'inizio di un container. Queste funzioni sono probabilmente il modo più comune con il quale vengono aggiunti elementi a un container sequenziale. Per rimuovere singoli elementi da un container sequenziale si usano le funzioni pop_front() e pop_back() che, rispettivamente, rimuovono elementi dall'inizio o dalla fine del container.



Uno dei modi più comuni per accedere agli elementi di un container è attraverso un iteratore. I container sequenziali e associativi forniscono le funzioni membro begin() ed end() che restituiscono gli iteratori all'inizio e alla fine del container. Questi iteratori sono molto utili per accedere al contenuto del container. Ad esempio per scorrere un container si può ottenere un iteratore al suo inizio con begin() e poi incrementare l'iteratore finché il suo valore non raggiunge end().

I container associativi forniscono la funzione find() che consente di individuare un elemento del container a partire dalla sua chiave. Dato che i container associativi associano a una chiave il suo valore, find() rappresenta il modo in cui vengono normalmente individuati gli elementi contenuti in un container associativo.

Poiché vector è un array dinamico, supporta la sintassi di accesso agli elementi tramite indicizzazione.

Dopo aver creato il container contenente informazioni, questo potrà essere manipolato tramite uno o più algoritmi. Gli algoritmi non solo consentono di modificare il contenuto di un container in un modo predefinito ma consentono anche di trasformare un tipo di sequenza in un altro.

Nelle prossime sezioni si imparerà ad applicare queste tecniche generali a tre diversi container: vector, list e map. Se si impara il funzionamento di questi container, non vi dovrebbero essere problemi a impiegare anche gli altri.

## 24.4 I vettori

vector è probabilmente il container generico più utilizzato. La classe vector supporta un array dinamico. Si tratta di un array le cui dimensioni crescono automaticamente. Come si sa, in C++ le dimensioni di un array sono fissate in fase di compilazione. Anche se questo è di gran lunga il modo più efficiente per implementare gli array, è anche il più restrittivo in quanto le dimensioni dell'array non possono essere modificate run-time in modo da adattarsi alle condizioni di funzionamento del programma. Un vettore risolve questo problema allocando la memoria ogni volta che viene richiesta. Nonostante il fatto che i vettori abbiano una natura dinamica, è possibile utilizzarli impiegando la normale notazione degli array.

Ecco come è specificata la classe vector:

```
template <class T, class Allocator = allocator<T>>class vector
```

Qui, T è il tipo di dati memorizzato e Allocatore specifica l'allocatore che, normalmente, è l'allocatore standard. La classe vector ha i seguenti costruttori:

```
explicit vector(const Allocator &a = Allocator( ) );
explicit vector(size_type num, const T &val = T( ),
                const Allocator &a = Allocator( ));
```

```
vector(const vector<T, Allocator> &ob);
template <class InIter> vector(InIter start, InIter end,
                const Allocator &a = Allocator( ));
```

La prima forma costruisce un vettore vuoto. La seconda forma costruisce un vettore contenente *num* elementi con il valore *val*. Per *val* si può indicare un valore standard. La terza forma costruisce un vettore che contiene gli stessi elementi di *ob*. La quarta forma costruisce un vettore che contiene gli elementi dell'intervallo specificato dagli iteratori *start* ed *end*.

Ogni oggetto che verrà memorizzato in un vettore deve definire un costruttore standard. Inoltre deve definire le operazioni < e ==. Alcuni compilatori possono chiedere che vengano definiti anche altri operatori di confronto. Poiché si tratta di un dettaglio implementativo, per avere informazioni precise si deve consultare la documentazione del compilatore. Tutti i tipi interni soddisfano automaticamente questi requisiti.

Anche se la sintassi template può sembrare piuttosto complessa, non è difficile dichiarare un vettore. Ecco alcuni esempi:

```
vector<int> iv;              // crea un vettore di int di lunghezza zero
vector<char> cv(5);          // crea un vettore di char di 5 elementi
vector<char> cv(5, 'x'); // inizializza un vettore di char di 5 elementi
vector<int> iv2(iv);     // crea un vettore di int da un altro vettore di int
```

Per la classe vector sono definiti i seguenti operatori di confronto:

```
==, <, <=, !=, >, >=
```

Per vector è definito anche l'operatore di specificazione dell'indice. Questo consente di accedere agli elementi di un vettore utilizzando la normale notazione per gli array.

La Tabella 24.2 mostra molte delle funzioni membro definite da vector. La Parte quarta di questo volume contiene un elenco completo delle classi STL. Fra le funzioni membro più utilizzate vi sono size(), begin(), end(), push_back(), insert() ed erase(). La funzione size() restituisce le dimensioni di un vettore. Questa funzione è utile poiché consente di determinare run-time le dimensioni di un vettore. Si ricordi che le dimensioni di un vettore possono crescere a seconda delle esigenze e pertanto devono essere determinate durante l'esecuzione e non durante la compilazione.

La funzione begin() restituisce un iteratore all'inizio del vettore. La funzione end() restituisce un iteratore alla fine del vettore. Come si è detto gli iteratori sono simili ai puntatori e grazie alle funzioni begin() ed end() si può ottenere con rapidità un iteratore all'inizio o alla fine del vettore.



La funzione push_back() inserisce un valore alla fine del vettore. Se necessario, la lunghezza del vettore viene incrementata in modo da contenere il nuovo elemento. Utilizzando la funzione insert() è possibile inserire gli elementi nella parte mediana del vettore. Un vettore può anche essere inizializzato. In ogni caso, una volta che un vettore contiene degli elementi, diventa possibile eseguire accessi o modifiche utilizzando l'operatore [ ]. Per rimuovere elementi da un vettore si usa la funzione erase().

**Tabella 24.2** Alcune delle funzioni membro più utili definite da vector.

| MEMBRO | DESCRIZIONE |
|---|---|
| `reference back( );`<br>`const_reference back( ) const;` | Restituisce l'indirizzo dell'ultimo elemento di un vettore. |
| `iterator begin( );`<br>`const_iterator begin( ) const;` | Restituisce un iteratore al primo elemento di un vettore. |
| `void clear( );` | Rimuove tutti gli elementi dal vettore. |
| `bool empty( ) const;` | Restituisce true se il vettore chiamante è vuoto, altrimenti restituisce false. |
| `iterator end( );`<br>`const_iterator end( ) const;` | Restituisce un iteratore alla fine del vettore. |
| `iterator erase(iterator i);` | Rimuove l'elemento puntato da *i*. Restituisce un iteratore all'elemento successivo a quello rimosso. |
| `iterator erase(iterator start, iterator end);` | Rimuove gli elementi compresi fra *start* ed *end*. Restituisce un iteratore all'elemento che segue l'ultimo elemento rimosso. |
| `reference front( );`<br>`const_reference front( ) const;` | Restituisce l'indirizzo del primo elemento di un vettore. |
| `iterator insert(iterator i, const T &val);` | Inserisce *val* immediatamente prima dell'elemento specificato da *i*. Restituisce un iteratore all' elemento. |
| `void insert(iterator i, size_type num,`<br>`const T & val)` | Inserisce *num* copie di *val* immediatamente prima dell'elemento specificato da *i*. |
| `template <class InIter>`<br>`void insert(iterator i, InIter start,`<br>`InIter end);` | Inserisce la sequenza definita da *start* ed *end* immediatamente prima dell'elemento specificato da *i*. |
| `reference operator[ ](size_type i) const;`<br>`const_reference operator[ ](size_type i)`<br>`const;` | Restituisce l'indirizzo dell'elemento specificato da *i*. |
| `void pop_back( );` | Rimuove l'ultimo elemento di un vettore. |
| `void push_back(const T &val);` | Aggiunge alla fine del vettore un elemento con il valore specificato da *val*. |
| `size_type size( ) const;` | Restituisce il numero di elementi contenuti in un vettore. |

Ecco un breve esempio che illustra le operazioni di base di un vettore.

```
// Illustra l'uso di un vettore.
#include <iostream>
#include <vector>
#include <cctype>
using namespace std;

int main()
{
  vector<char> v(10); // crea un vettore di lunghezza 10
  int i;

  // visualizza le dimensioni originali di v
  cout << "Dimensioni = " << v.size() << endl;

  // assegna un valore agli elementi di un vettore
  for(i=0; i<10; i++) v[i]  i + 'a';

  // visualizza il contenuto del vettore
  cout << "Contenuto =";
  for(i=0; i<v.size(); i++) cout << v[i] << " ";
  cout << "\n\n";

  cout << "Espansione del vettore\n";
  /* inserisce nuovi elementi alla fine del vettore,
     il quale cresce in modo appropriato */
  for(i=0; i<10; i++) v.push_back(i + 10 + 'a');

  // visualizza le attuali dimensioni di v
  cout << "Attuali dimensioni = " << v.size() << endl;

  // visualizza il contenuto del vettore
  cout << "Contenuto =\n";
  for(i=0; i<v.size(); i++) cout << v[i] << " ";
  cout << "\n\n";

  // cambia il contenuto del vettore
  for(i=0; i<v.size(); i++) v[i]  toupper(v[i]);
  cout << "Contenuto modificato =\n";
  for(i=0; i<v.size(); i++) cout << v[i] << " ";
  cout << endl;

  return 0;
}
```



Ecco l'output del programma:

```
Dimensioni = 10
Contenuto =
a b c d e f g h i j

Espansione del vettore
Attuali dimensioni = 20
Contenuto =
a b c d e f g h i j k l m n o p q r s t

Contenuto modificato =
A B C D E F G H I J K L M N O P Q R S T
```

Si osservi attentamente questo programma. In main() viene creato un vettore di caratteri chiamato v con capacità iniziale pari a 10. Pertanto inizialmente v contiene 10 elementi. Questo è confermato dalla chiamata alla funzione membro size(). Successivamente questi 10 elementi vengono inizializzati con i caratteri da "a" a "j" e quindi viene visualizzato il contenuto di v. Si noti che viene impiegata la normale notazione per gli array. Quindi alla fine del vettore v vengono aggiunti altri 10 elementi utilizzando la funzione push_back(). Questo provoca una crescita di v e come si può vedere dall'output, le sue dimensioni raggiungono il valore 20. Infine il valore degli elementi di v viene modificato utilizzando la normale notazione per array.

Questo programma ha anche un altro punto interessante. Si noti che i cicli che visualizzano il contenuto di v usano come limite il valore v.size(). Un vantaggio dei vettori rispetto agli array è il fatto che è possibile conoscere le loro dimensioni correnti. Come si può immaginare questa possibilità è utile in molte situazioni.

## Accesso a un vettore tramite un iteratore

Come si sa, gli array e i puntatori sono strettamente correlati in C++. Per accedere a un array si può utilizzare il suo indice oppure un puntatore. Analogamente in STL vi è una stretta correlazione fra vettori e iteratori. Si può accedere ai membri di un vettore utilizzando l'indice oppure un iteratore. Questa possibilità è illustrata dal seguente esempio:

```
// Accede agli elementi of un vettore tramite un iteratore.
#include <iostream>
#include <vector>
#include <cctype>
using namespace std;
```

```
int main()
{
  vector<char> v(10); // crea un vettore di lunghezza 10
  vector<char>::iterator p; // crea un iteratore
  int i;

  // assegna un valore agli elementi del vettore
  p = v.begin();
  i = 0;
  while(p != v.end()) {
    *p = i + 'a';
    p++;
    i++;
  }

  // visualizza il contenuto del vettore
  cout << "Contenuto originale =\n";
  p = v.begin();
  while(p != v.end()) {
    cout << *p << " ";
    p++;
  }
  cout << "\n\n";

  // cambia il contenuto del vettore
  p = v.begin();
  while(p != v.end()) {
    *p = toupper(*p);
    p++;
  }

  // visualizza il contenuto del vettore
  cout << "Contenuto modificato =\n";
  p = v.begin();
  while(p != v.end()) {
    cout << *p << " ";
    p++;
  }
  cout << endl;

  return 0;
}
```



Ecco l'output del programma:

```
Contenuto originale =
a b c d e f g h i j

Contenuto modificato =
A B C D E F G H I J
```

Nel programma, si noti il modo in cui viene dichiarato l'iteratore p. Il tipo iterator è definito dalle classi container. Pertanto per ottenere un iteratore per un determinato container si deve utilizzare una dichiarazione simile a quella illustrata nell'esempio: basta qualificare iterator con il nome del container. Nel programma, p è inizializzato in modo da puntare all'inizio del vettore grazie alla funzione membro begin(). Questo iteratore può quindi essere utilizzato per accedere al vettore un elemento alla volta incrementandolo a piacere. Questo processo è analogo al modo in cui si usa un puntatore per accedere agli elementi di un array. Per determinare il momento in cui viene raggiunta la fine dell'array si impiega la funzione membro end(). Questa funzione restituisce un iteratore all'elemento successivo rispetto all'ultimo elemento del vettore. Pertanto, quando p è uguale a end(), è stata raggiunta la fine dell'array.

## Inserimento e cancellazione degli elementi in un vettore

La funzione insert() consente di inserire elementi all'interno del vettore. Per rimuovere elementi si usa erase(). Il seguente programma mostra l'uso di queste due funzioni:

```
// Illustra l'uso di insert ed erase.
#include <iostream>
#include <vector>
using namespace std;

int main()
{
  vector<char> v(10);
  vector<char> v2;
  char str[] = "<Vector>";
  int i;

  // inizializza v
  for(i=0; i<10; i++) v[i] = i + 'a';
```

```
// copia in v2 i caratteri contenuti in str
for(i=0; str[i]; i++) v2.push_back(str[i]);

// visualizza il contenuto originale del vettore
cout << "Contenuto originale di v:\n";
for(i=0; i<v.size(); i++) cout << v[i] << " ";
cout << "\n\n";

vector<char>::iterator p = v.begin();
p += 2; // punta al terzo elemento

// inserisce 10 'X' in v
v.insert(p, 10, 'X');

// visualizza il contenuto dopo l'inserimento
cout << "Dimensioni dopo l'inserimento delle X  " << v.size() << endl;
cout << "Contenuto dopo l'inserimento =\n";
for(i=0; i<v.size(); i++) cout << v[i] << " ";
cout << "\n\n";

// rimozione degli elementi
p = v.begin();
p += 2; // punta al terzo elemento
v.erase(p, p+10); // rimuove i 10 elementi successivi

// visualizza il contenuto dopo la cancellazione
cout << "Dimensioni dopo la cancellazione = " << v.size() << endl;
cout << "Contenuto dopo la cancellazione =\n";
for(i=0; i<v.size(); i++) cout << v[i] << " ";
cout << "\n\n";

// Inserisce v2 in v
v.insert(p, v2.begin(), v2.end());
cout << "Dimensioni dopo l'inserimento di v2 =";
cout << v.size() << endl;
cout << "Contenuto dopo l'inserimento =\n";
for(i=0; i<v.size(); i++) cout << v[i] << " ";
cout << endl;

return 0;
}
```

Questo programma produce il seguente output:

```
Contenuto originale di v:
a b c d e f g h i j
```



```
Dimensioni dopo l'inserimento delle X = 20
Contenuto dopo l'inserimento =
a b X X X X X X X X X X c d e f g h i j

Dimensioni dopo la cancellazione = 10
Contenuto dopo la cancellazione =
a b c d e f g h i j

Dimensioni dopo l'inserimento di v2 =18
Contenuto dopo l'inserimento =
a b < e c t o r > c d e f g h i j
```

Questo programma mostra due forme di insert(). La prima volta che viene utilizzata, la funzione inserisce in v 10 "X". La seconda volta inserisce in v il contenuto di un secondo vettore, v2. Questo secondo uso è il più interessante. Tale forma richiede come argomenti tre iteratori. Il primo specifica il punto di inserimento nel container chiamante. Gli ultimi due puntano all'inizio e alla fine della sequenza da inserire.

## Memorizzazione di oggetti in un vettore

I precedenti esempi si occupavano dell'inserimento in un vettore di elementi appartenenti a tipi standard, ma è anche possibile creare vettori di oggetti appartenenti a classi. I vettori possono contenere ogni tipo di oggetti, compresi quelli delle classi del programma. Ecco un esempio che utilizza un vettore per memorizzare oggetti che contengono delle misurazioni di temperatura eseguite nell'arco di una settimana. Si noti che DailyTemp definisce il costruttore standard e che vengono fornite le versioni modificate tramite overloading degli operatori < e ==. Si ricordi che a seconda del modo in cui il compilatore implementa la libreria STL, possono essere definiti anche altri operatori di confronto.

```
// memorizza un oggetto di tipo class in un vettore.
#include <iostream>
#include <vector>
#include <cstdlib>
using namespace std;

class DailyTemp
  int temp;
public:
  DailyTemp() { temp = 0; }
  DailyTemp(int x) { temp = x; }
```

```
  DailyTemp &operator=(int x) {
    temp = x; return *this;
  }

  double get_temp() { return temp; }
};

bool operator<(DailyTemp a, DailyTemp b)
{
  return a.get_temp() < b.get_temp();
}

bool operator==(DailyTemp a, DailyTemp b)
{
  return a.get_temp() == b.get_temp();
}

int main()
{
  vector<DailyTemp> v;
  int i;

  for(i=0; i<7; i++)
    v.push_back(DailyTemp(60 + rand()%30));

  cout << "Temperature Farenheit:\n";
  for(i=0; i<v.size(); i++)
    cout << v[i].get_temp() << " ";

  cout << endl;

  // Conversione da gradi Farenheit a gradi Centigradi
  for(i=0; i<v.size(); i++)
    v[i] = (v[i].get_temp()-32) * 5/9 ;

  cout << "Temperature in gradi Centigradi:\n";
  for(i=0; i<v.size(); i++)
    cout << v[i].get_temp() << " ";

  return 0;
}
```

Ecco l'output del programma:

```
Temperature Farenheit:
71 77 64 70 89 64 78
```



```
Temperature in gradi Centigradi:
21 25 17 21 31 17 25
```

I vettori offrono grandi potenzialità, sicurezza e flessibilità ma sono meno efficienti dei normali array. Pertanto, per i compiti di programmazione più semplici, in genere si preferisce utilizzare normali array; tuttavia vi sono situazioni in cui i vantaggi dei vettori ripagano di gran lunga i loro costi in termini di efficienza.

## 24.5 Le liste

La classe list supporta una lista lineare bidirezionale. A differenza di un vettore che supporta accessi diretti, una lista offre solo accessi sequenziali. Poiché le liste sono bidirezionali, l'accesso può avvenire in entrambi i sensi di scorrimento.

Una lista ha la seguente specifica template:

```
template <class T, class Allocator = allocator<T>> class list
```

Qui, T è il tipo dei dati memorizzati nella lista. L'allocatore è specificato da Allocator (che normalmente è l'allocatore standard). Ecco i suoi costruttori:

```
explicit list(const Allocator &a = Allocator( ) );
explicit list(size_type num, const T &val = T ( ),
              const Allocator &a = Allocator( ));
list(const list<T, Allocator> &ob);
template <class InIter>list(InIter start, InIter end,
              const Allocator &a = Allocator( ));
```

La prima forma costruisce una lista vuota. La seconda forma costruisce una lista contenente *num* elementi di valore *valore* e consente anche di utilizzare valori standard. La terza forma costruisce una lista che contiene gli stessi elementi di *ob*. La quarta forma costruisce una lista che contiene gli elementi nell'intervallo specificato dagli iteratori *start* ed *end*.

Per *list* sono definiti i seguenti operatori di confronto:

```
==, <, <=, !=, >, >=
```

La Tabella 24.3 mostra alcune delle funzioni membro più utilizzate di list. Come nel caso dei vettori, per inserire elementi nella lista si usa la funzione push_back(). Per inserire elementi all'inizio della lista si usa push_front(). Un elemento può anche essere inserito in mezzo alla lista con insert(). Per unire due liste si usa splice() e per inserire una lista in un'altra si usa merge().

**Tabella 24.3** Alcune delle funzioni membro più utilizzate di list.

| MEMBRO | DESCRIZIONE |
|---|---|
| `reference back( );`<br>`const_reference back( ) const;` | Restituisce l'indirizzo dell'ultimo elemento della lista. |
| `iterator begin( );`<br>`const_iterator begin( ) const;` | Restituisce un iteratore al primo elemento della lista. |
| `void clear( );` | Rimuove tutti gli elementi dalla lista. |
| `bool empty( ) const;` | Restituisce true se la lista chiamante è vuota, altrimenti restituisce false. |
| `iterator end( );`<br>`const_iterator end( ) const;` | Restituisce un iteratore alla fine della lista. |
| `iterator erase(iterator i);` | Rimuove l'elemento puntato da *i*. Restituisce un iteratore all'elemento successivo a quello rimosso. |
| `iterator erase(iterator start,`<br>`iterator end);` | Rimuove gli elementi nell'intervallo compreso fra *start* ed *end*. Restituisce un iteratore all'elemento che segue l'ultimo elemento rimosso. |
| `reference front( );`<br>`const_reference front( ) const;` | Restituisce l'indirizzo del primo elemento della lista. |
| `iterator insert(iterator i, const T &val);` | Inserisce *val* immediatamente prima dell'elemento specificato da *i*. Restituisce un iteratore all'elemento. |
| `void insert(iterator i, size_type num,`<br>`const T &val)` | Inserisce *num* copie di *val* immediatamente prima dell'elemento specificato da *i*. |
| `template <class InIter>`<br>`void insert(iterator i,`<br>`InIter start, InIter end);` | Inserisce la sequenza definita da *start* ed *end* immediatamente prima dell'elemento specificato da *i*. |
| `void merge(list<T, Allocator> &ob);`<br>`template <class Comp>`<br>`void merge(<list<T, Allocator> &ob,`<br>`Comp cmpfn);` | Unisce la lista ordinata contenuta in *ob* alla lista ordinata chiamante. Il risultato è ordinato. Dopo l'unione, la lista contenuta in *ob* è vuota. Nella seconda forma, può essere specificata una funzione di confronto che determina quando un elemento è minore di un altro. |
| `void pop_back( );` | Rimuove l'ultimo elemento della lista. |
| `void pop_front( );` | Rimuove il primo elemento della lista. |
| `void push_back(const T &val);` | Aggiunge alla fine della lista un elemento con il valore specificato da *val*. |
| `void push_front(const T &val);` | Aggiunge alla fine della lista un elemento con il valore specificato da *val*. |
| `void remove(const T &val);` | Rimuove dalla lista gli elementi con il valore *val*. |
| `void reverse( );` | Inverte la lista chiamante. |

*(segue)*



**Tabella 24.3** Alcune delle funzioni membro più utilizzate di list. *(continua)*

| MEMBRO | DESCRIZIONE |
|---|---|
| `size_type size( ) const;` | Restituisce il numero di elementi contenuti nella lista. |
| `void sort( );`<br>`template <class Comp>`<br>`void sort(Comp cmpfn);` | Ordina la lista. La seconda forma ordina la lista con la funzione di confronto *fn* per determinare quando un elemento è minore di un altro. |
| `void splice(iterator i,`<br>`list<T, Allocator> &ob);` | Il contenuto di *ob* viene inserito nella lista chiamante nella posizione puntata da *i*. Dopo l'operazione, *ob* è vuoto. |
| `void splice(iterator i,`<br>`list<T, Allocator> &ob,`<br>`iterator el);` | L'elemento puntato da *el* viene rimosso dalla lista *ob* e memorizzato nella lista chiamante nella posizione puntata da *i*. |
| `void splice(iterator i,`<br>`list<T, Allocator> &OB,`<br>`ITERATOR start, iterator end);` | L'intervallo definito da *start* ed *end* viene rimosso da *ob* e inserito nella lista chiamante a partire dalla posizione puntata da *i*. |

I dati che dovranno essere contenuti nella lista devono definire un costruttore standard. Inoltre devono definire i vari operatori di confronto. Al momento attuale, i requisiti per gli oggetti che verranno memorizzati in una lista variano da compilatore a compilatore dunque è opportuno controllarne la documentazione.

Ecco un semplice esempio d'uso di una lista.

```
// Elementi di base delle liste.
#include <iostream>
#include <list>
using namespace std;

int main()
{
  list<int> lst; // crea una lista vuota
  int i;

  for(i=0; i<10; i++) lst.push_back(i);

  cout << "Dimensioni = " << lst.size() << endl;

  cout << "Contenuto = ";
  list<int>::iterator p = lst.begin();
  while(p != lst.end()) {
    cout << *p << " ";
    p++;
  }
  cout << "\n\n";
```

```
  // cambia il contenuto della lista
  p = lst.begin();
  while(p != lst.end()) {
    *p = *p + 100;
    p++;
  }

  cout << "Contenuto modificato = ";
  p = lst.begin();
  while(p != lst.end()) {
    cout << *p << " ";
    p++;
  }

  return 0;
}
```

Ecco l'output prodotto dal programma:

```
Dimensioni = 10
Contenuto = 0 1 2 3 4 5 6 7 8 9

Contenuto modificato = 100 101 102 103 104 105 106 107 108 109
```

Questo programma crea una lista di interi. Innanzitutto viene creato un oggetto list vuoto. Poi nella lista vengono inseriti dieci interi. L'operazione viene eseguita con la funzione push_back() che inserisce ogni nuovo valore alla fine della lista esistente. Poi vengono visualizzati sia le dimensioni che il contenuto della lista. La lista viene visualizzata tramite il seguente frammento di codice:

```
list<int>::iterator p = lst.begin();
while(p != lst.end()) {
  cout << *p << " ";
  p++;
}
```

Qui l'iteratore p viene inizializzato per puntare all'inizio della lista. Ad ogni ciclo p viene incrementato e dunque punta all'elemento successivo. Il ciclo termina quando p punta alla fine della lista. Questo frammento di codice è praticamente identico a quello utilizzato per il vettore. I cicli come questo sono molto comuni nel codice STL e il fatto che gli stessi costruttori possano essere utilizzati per accedere a container di tipo differente fa parte della potenza della libreria STL.



## La funzione end()

Ora è giunto il momento di enfatizzare un attributo inaspettato della funzione container end(). La funzione end() non restituisce un puntatore all'ultimo elemento del container ma all'elemento successivo. Pertanto l'ultimo elemento del container è puntato da end() - 1. Questo consente di scrivere algoritmi molto efficienti che attraversano tutti gli elementi di un container incluso l'ultimo. Quando l'iteratore avrà raggiunto lo stesso valore di end(), si saprà che sono stati attraversati tutti gli elementi. Occorre tenere conto di ciò in quanto non è una caratteristica intuitiva di end(). Ad esempio, si consideri il seguente programma che visualizza una lista prima in avanti e poi all'indietro.

```
// Uso di end().
#include <iostream>
#include <list>
using namespace std;

int main()
{
  list<int> lst; // crea una lista vuota
  int i;

  for(i=0; i<10; i++) lst.push_back(i);

  cout << "Lista visualizzata in avanti =\n";
  list<int>::iterator p = lst.begin();
  while(p != lst.end()) {
    cout << *p << " ";
    p++;
  }
  cout << "\n\n";

  cout << "Lista visualizzata all'indietro =";
  p = lst.end();
  while(p != lst.begin()) {
    p--; // decrementa il puntatore
    cout << *p << " ";
  }

  return 0;
}
```

Ecco l'output prodotto dal programma:

```
Lista visualizzata in avanti =
0 1 2 3 4 5 6 7 8 9

Lista visualizzata all'indietro =
9 8 7 6 5 4 3 2 1 0
```

Il codice che visualizza la lista in avanti dovrebbe essere già noto. Ma si faccia attenzione al codice che visualizza la lista in ordine inverso. Innanzitutto all'iteratore p viene assegnata con end() la posizione finale della lista. Poiché end() restituisce un iteratore che punta all'oggetto successivo all'ultimo memorizzato nella lista, è necessario decrementare p. Questo è il motivo per cui p viene decrementato prima dell'istruzione cout contenuta nel ciclo. Si ricordi: end() non restituisce un puntatore all'ultimo oggetto della lista ma all'oggetto successivo.

## Le funzioni push_front() e push_back()

Si può costruire una lista aggiungendo elementi da entrambe le estremità. Finora gli elementi erano stati aggiunti solo alla fine, impiegando push_back(). Per aggiungere elementi all'inizio della lista si usa push_front(). Ad esempio:

```
/* Differenze fra
   push_back() e push_front(). */
#include <iostream>
#include <list>
using namespace std;

int main()
{
  list<int> lst1, lst2;
  int i;

  for(i=0; i<10; i++) lst1.push_back(i);
  for(i=0; i<10; i++) lst2.push_front(i);

  list<int>::iterator p;

  cout << "Contenuto di lst1:\n";
  p = lst1.begin();
  while(p != lst1.end()) {
    cout << *p << " ";
    p++;
  }

  cout << "\n\n";

  cout << "Contenuto di lst2:";
  p = lst2.begin();
  while(p != lst2.end()) {
    cout << *p << " ";
    p++;
  }

  return 0;
}
```



Ecco l'output prodotto dal programma:

```
Contenuto di lst1:
0 1 2 3 4 5 6 7 8 9

Contenuto di lst2:
9 8 7 6 5 4 3 2 1 0
```

Poiché lst2 viene costruita inserendo gli elementi nella parte frontale, viene prodotta una lista in ordine inverso rispetto a lst1 nella quale gli elementi venivano inseriti via via alla fine.

## Ordinamento di una lista

Una lista può essere ordinata richiamando la funzione membro sort(). Il seguente programma crea una lista di interi casuali e quindi la ordina.

```
// Ordinamento di una lista.
#include <iostream>
#include <list>
#include <cstdlib>
using namespace std;

int main()
{
  list<int> lst;
  int i;

  // crea una lista di interi casuali
  for(i=0; i<10; i++)
    lst.push_back(rand());
```

```
  cout << "Contenuto originale =\n";
  list<int>::iterator p = lst.begin();
  while(p != lst.end()) {
    cout << *p << " ";
    p++;
  }
  cout << endl << endl;

  // ordina la lista
  lst.sort();

  cout << "Contenuto ordinato =\n";
  p = lst.begin();
  while(p != lst.end()) {
    cout << *p << " ";
    p++;
  }

  return 0;
}
```

Ecco un ipotetico output prodotto dal programma:

```
Contenuto originale =
41 18467 6334 26500 19169 15724 11478 29358 26962 24464

Contenuto ordinato =
41 6334 11478 15724 18467 19169 24464 26500 26962 29358
```

## Unione di due liste

Una lista ordinata può essere unita a un'altra. Il risultato è una lista ordinata che contiene gli oggetti presenti nelle due liste di partenza. La nuova lista occuperà la lista chiamante mentre la seconda lista rimarrà vuota. Il seguente esempio unisce due liste. La prima lista contiene i numeri pari compresi fra 0 e 9 e la seconda lista contiene i numeri dispari (sempre compresi fra 0 e 9). L'unione delle due liste produce la sequenza 0 1 2 3 4 5 6 7 8 9.

```
// Unione di due liste.
#include <iostream>
#include <list>
using namespace std;
```



```
int main()
{
  list<int> lst1, lst2;
  int i;

  for(i=0; i<10; i+2) lst1.push_back(i);
  for(i=1; i<11; i+2) lst2.push_back(i);

  cout << "Contenuto di lst1:\n";
  list<int>::iterator p = lst1.begin();
  while(p != lst1.end()) {
    cout << *p << " ";
    p++;
  }
  cout << endl << endl;

  cout << "Contenuto di lst2:\n";
  p = lst2.begin();{
  while(p != lst2.end()) {
    cout << *p << " ";
    p++;
  }
  cout << endl << endl;

  // unisce le due liste
  lst1.merge(lst2);
  if(lst2.empty())
    cout << "Ora lst2 è vuota";

  cout << "Contenuto di lst1 dopo merge:\n";
  p = lst1.begin();
  while(p != lst1.end()) {
    cout << *p << " ";
    p++;
  }

  return 0;
}
```

Ecco l'output prodotto dal programma:

```
Contenuto di lst1:
0 2 4 6 8

Contenuto di lst2:
1 3 5 7 9
```

```
Ora lst2 è vuota
Contenuto di lst1 dopo merge:
0 1 2 3 4 5 6 7 8 9
```

Un'ultima nota su questo esempio; si noti l'uso della funzione empty(). Tale funzione restituisce true se il container chiamante è vuoto. Poiché empty() rimuove tutti gli elementi dalla seconda lista, dopo il completamento dell'unione essa risulterà vuota, come si può notare dall'output del programma.

## Memorizzazione di oggetti in una lista

Ecco un esempio che utilizza una lista per memorizzare oggetti di tipo myclass. Si noti che gli operatori <, >, != e == sono modificati tramite overloading per operare su oggetti di tipo myclass. Questi sono gli operatori richiesti dal compilatore Microsoft Visual C++ (il compilatore impiegato per eseguire il test degli esempi STL presentati in questo capitolo). Altri compilatori potrebbero richiedere l'overloading anche di altri operatori. La libreria STL utilizza queste funzioni per determinare l'ordinamento e l'uguaglianza degli oggetti del container. Anche se la lista non è un container ordinato, è comunque necessario avere la possibilità di confrontare gli elementi durante le operazioni di ricerca, ordinamento o unione.

```
// Memorizzazione di oggetti in una lista.
#include <iostream>
#include <list>
#include <cstring>
using namespace std;

class myclass {
  int a, b;
  int sum;
public:
  myclass() { a = b = 0; }
  myclass(int i, int j) {
    a = i;
    b = j;
    sum = a + b;
  }
  int getsum() { return sum; }

  friend bool operator<(const myclass &o1,
                        const myclass &o2);
  friend bool operator>(const myclass &o1,
                        const myclass &o2);
```



```
  friend bool operator==(const myclass &o1,
                         const myclass &o2);
  friend bool operator!=(const myclass &o1,
                         const myclass &o2);
};

bool operator<(const myclass &o1, const myclass &o2)
{
  return o1.sum < o2.sum;
}

bool operator>(const myclass &o1, const myclass &o2)
{
  return o1.sum > o2.sum;
}

bool operator==(const myclass &o1, const myclass &o2)
{
  return o1.sum == o2.sum;
}

bool operator!=(const myclass &o1, const myclass &o2)
{
  return o1.sum != o2.sum;
}

int main()
{
  int i;

  // crea la prima lista
  list<myclass> lst1;
  for(i=0; i<10; i++) lst1.push_back(myclass(i, i));

  cout << "Prima lista = ";
  list<myclass>::iterator p = lst1.begin();
  while(p != lst1.end()) {
    cout << p->getsum() << " ";
    p++;
  }
  cout << endl;

  // crea la seconda lista
  list<myclass> lst2;
  for(i=0; i<10; i++) lst2.push_back(myclass(i*2, i*3));
```

```
  cout << "Seconda lista = ";
  p = lst2.begin();
  while(p != lst2.end()) {
    cout << p->getsum() << " ";
    p++;
  }
  cout << endl;

  // ora unisce lst1 e lst2
  lst1.merge(lst2);

  // visualizza il risultato
  cout << "Risultato dell'unione = ";
  p = lst1.begin();
  while(p != lst1.end()) {
    cout << p->getsum() << " ";
    p++;
  }

  return 0;
}
```

Il programma crea due liste di oggetti myclass e poi ne visualizza il contenuto. Quindi unisce le due liste e visualizza il risultato. Ecco l'output prodotto dal programma:

```
Prima lista = 0 2 4 6 8 10 12 14 16 18
Seconda lista = 0 5 10 15 20 25 30 35 40 45
Risultato dell'unione = 0 0 2 4 5 6 8 10 10 12 14 15 16 18 20 25 30 35 40 45
```

## 24.6 Le mappe

La classe map supporta un container associativo in cui viene realizzata una mappa fra chiavi univoche e valori. In pratica una chiave è semplicemente un nome assegnato a un valore. Successivamente diventa possibile recuperare il valore utilizzando la relativa chiave. Pertanto in generale, una mappa è un elenco di coppie chiave/valore. La potenza della mappa è legata al fatto che consente di ricercare un valore sulla base della sua chiave. Ad esempio si potrebbe definire una mappa che utilizza come chiave il nome di una persona e memorizza come valore il numero di telefono di quella persona. I container associativi stanno diventando sempre più popolari nel campo della programmazione.



Come si è detto, una mappa può contenere solo chiavi univoche, non è consentito creare chiavi duplicate. Per creare una mappa che consente l'impiego di chiavi multiple si deve utilizzare una multi-mappa.

Il container map ha la seguente specifica template.

```
template <class Key, class T, class Comp  less<Key>,
          class Allocator = allocator<T>> class map
```

Qui Key è il tipo di dati delle chiavi, T è il tipo di dati dei valori memorizzati (ovvero mappati) e Comp è una funzione che confronta le due chiavi. Normalmente si tratta della funzione oggetto di servizio standard less(). Allocator è l'allocatore (che normalmente è allocator).

Una mappa ha i seguenti costruttori:

```
explicit map(const Comp &cmpfn = Comp( ),
             const Allocator &a = Allocator( ) );
map(const map<Key, T, Comp, Allocator> &ob);
template <class InIter> map(InIter start, InIter end,
             const Comp &cmpfn = Comp( ), const Allocator &a =
              Allocator( ));
```

La prima forma costruisce una mappa vuota. La seconda forma costruisce una mappa che contiene gli stessi elementi di *ob*. La terza forma costruisce una mappa che contiene gli elementi dell'intervallo specificato dagli iteratori *start* ed *end*. La funzione specificata da *cmpfn*, se presente, determina l'ordinamento della mappa.

In generale, ogni oggetto utilizzato come chiave deve definire un costruttore standard per eseguire l'overloading di tutti gli operatori di confronto necessari.

Per map sono definiti i seguenti operatori di confronto.

```
==, <, <=, !=, >, >=
```

La Tabella 24.4 mostra numerose funzioni membro di map. Nelle descrizioni, key_type è il tipo della chiave e value_type rappresenta pair<Key, T>.

Le coppie chiave/valore vengono memorizzate in una mappa come oggetti di tipo pair che ha la seguente specificazione template.

```
template <class Ktype, class type> struct pair {
  typedef Ktype first_type; // type of key
  typedef Vtype second_type; // type of value
  Ktype first; // contains the key
  Vtype second; // contains the value
```

```
  // costruttori
  pair();
  pair(const Ktype &k, const Vtype &v);
  template<class , class > pair(const<A, B> &ob);
}
```

Come si può dedurre dai commenti, il valore della chiave contiene la chiave stessa e il valore di value contiene il valore associato a tale chiave.

**Tabella 24.4** Le funzioni più utilizzate della container map.

| MEMBRO | DESCRIZIONE |
|---|---|
| `iterator begin( );`<br>`const_iterator begin( ) const;` | Restituisce un iteratore al primo elemento della mappa. |
| `void clear( );` | Rimuove tutti gli elementi dalla mappa. |
| `size_type count(const key_type &k) const;` | Restituisce il numero di volte che *k* si presenta nella mappa (1 o zero). |
| `bool empty( ) const;` | Restituisce true se la mappa chiamante è vuota, altrimenti restituisce false. |
| `iterator end( );`<br>`const_iterator end( ) const;` | Restituisce un iteratore alla fine della mappa. |
| `void erase(iterator i);` | Rimuove l'elemento puntato da *i*. |
| `void erase(iterator start, iterator end);` | Rimuove gli elementi nell'intervallo compreso fra *start* ed *end*. |
| `size_type erase(const key_type &k)` | Rimuove dalla mappa gli elementi la cui chiave è *k*. |
| `iterator find(const key_type &k);`<br>`const_iterator find(const key_type &k) const;` | Restituisce un iteratore alla chiave specificata. Se la chiave non viene trovata, allora viene restituito un iteratore alla fine della mappa. |
| `iterator insert(iterator i, const value_type &val);` | Inserisce *val* al posto o dopo l'elemento specificato da *i*. Restituisce un iteratore all'elemento. |
| `template <class InIter>`<br>`void insert(InIter start, InIter end)` | Inserisce un intervallo di elementi. |
| `pair<iterator, bool>`<br>`insert(const value_type &val);` | Inserisce *val* nella mappa chiamante. Restituisce un iteratore all'elemento. L'elemento viene inserito solo se non esiste ancora nella mappa. Se l'elemento viene inserito, restituisce, pair<iterator, true>, altrimenti restituisce pair<iterator, false>. |
| `reference operator[ ](const key_type &i)` | Restituisce l'indirizzo dell'elemento specificato da *i*. Se questo elemento non esiste viene inserito. |
| `size_type size( ) const;` | Restituisce il numero di elementi attualmente presenti nella lista. |



Si può costruire una coppia utilizzando uno dei costruttori di pair oppure make_pair(), che costituisce un oggetto pair sulla base del tipo degli oggetti utilizzati come parametri. make_pair() è una funzione generica che ha il seguente prototipo.

template <class *Ktype*, class *Vtype*>
pair<*Ktype*, *Vtype*> make_pair(const *Ktype &k*, const *Vtype &v*);

Come si può vedere, make_pair() restituisce un oggetto pair costituito dai valori dei tipi specificati da Ktype e Vtype. Il vantaggio di make_pair() è il fatto che il tipo degli oggetti memorizzati viene determinato automaticamente dal compilatore e dunque non deve essere specificato esplicitamente dal programmatore.

Il seguente programma illustra gli elementi di base dell'uso di una mappa. In particolare vengono utilizzate coppie chiave/valore che mostrano il mappaggio fra le lettere maiuscole e il corrispondente codice ASCII. Pertanto la chiave è un carattere e il valore è un intero. Le coppie chiave/valore memorizzate sono:

| | |
|---|---|
| A | 65 |
| B | 66 |
| C | 67 |

e così via. Dopo aver memorizzarlo le coppie, viene richiesta una chiave (ovvero una lettera compresa fra A e Z) e quindi viene visualizzato il codice ASCII di tale lettera.

```
// Uso delle mappe.
#include <iostream>
#include <map>
using namespace std;

int main()
{
  map<char, int> m;
  int i;

  // inserisce le coppie nella mappa
  for(i=0; i<26; i++) {
    m.insert(pair<char, int>('A'+i, 65+i));
  }

  char ch;
  cout << "Introdurre la chiave: ";
  cin >> ch;
```

```
  map<char, int>::iterator p;

  // trova il valore sulla base della chiave immessa
  p = m.find(ch);
  if(p != m.end())
    cout << "Il suo valore ASCII è " << p->second;
  else
    cout << "La chiave non è presente nella mappa.\n";

  return 0;
}
```

Si noti l'uso della classe template pair per costruire le coppie chiave/valore. Il tipo di dati specificato da pair deve corrispondere a quello della mappa in cui sono state inserite le coppie.

Dopo aver inizializzato la mappa con le chiavi e i valori, è possibile ricercare un valore partendo dalla sua chiave, utilizzando la funzione find() che restituisce un iteratore all'elemento corrispondente oppure (nel caso la chiave non venga trovata) alla fine della mappa. Quando la chiave viene trovata, il valore associato alla chiave è contenuto nel membro second di pair.

Nell'esempio, le coppie chiave/valore venivano costruite in modo esplicito, utilizzando pair<char, int>. Anche se questo approccio non ha nulla di sbagliato, spesso è più facile utilizzare make_pair() che costruisce un oggetto pair sulla base del tipo dei dati utilizzati come parametri. Ad esempio, dato il programma precedente, anche questa riga di codice inserisce in m coppie chiave/valore.

```
m.insert(make_pair((char)('A'+i), 65+i));
```

Qui la conversione cast in char è necessaria per eliminare la conversione automatica in int che viene eseguita quando i viene sommato a "A". Altrimenti la determinazione del tipo è automatica.

## Memorizzazione di oggetti in una mappa

Come per tutti gli altri container, anche una mappa può essere utilizzata per memorizzare oggetti appartenenti a classi create dall'utente. Ad esempio, il seguente programma crea un semplice elenco telefonico. In pratica crea una mappa di nomi e dei rispettivi numeri telefonici. A tale scopo crea due classi chiamate name e number. Dato che una mappa gestisce un elenco ordinato di chiavi, il programma definisce anche l'operatore < per oggetti di tipo name. In generale si deve definire l'operatore < per tutte le classi che verranno utilizzate come chiave (alcuni compilatori possono richiedere la definizione anche di altri operatori di confronto).



```
// Usa una mappa per creare una rubrica telefonica.
#include <iostream>
#include <map>
#include <cstring>
using namespace std;

class name {
  char str[40];
public:
  name() { strcpy(str, ""); }
  name(char *s) { strcpy(str, s); }
  char *get() { return str; }

};

// Definisce l'operatore "minore di".
bool operator<(name a, name b)
{
   return strcmp(a.get(), b.get()) < 0;
}

class phoneNum {
  char str[80];
public:
  phoneNum() { strcmp(str, ""); }
  phoneNum(char *s) { strcpy(str, s); }
  char *get() { return str; }
};


int main()
{
  map<name, phoneNum> directory;

 // inserisce nella mappa nomi e numeri
  directory.insert(pair<name, phoneNum>(name("Tommaso"),
                   phoneNum("555-4533")));
  directory.insert(pair<name, phoneNum>(name("Cristina"),
                   phoneNum("555-9678")));
  directory.insert(pair<name, phoneNum>(name("Giovanni"),
            phoneNum("555-8195")));
  directory.insert(pair<name, phoneNum>(name("Rachele"),
                   phoneNum("555-0809")));

  // dato un nome, trova il numero
  char str[80];
```

```
  cout << "Introdurre un nome: ";
  cin >> str;

  map<name, phoneNum>::iterator p;

  p = directory.find(name(str));
  if(p != directory.end())
    cout << "Numero telefonico: " <<   p->second.get();
  else
    cout << "Nome non presente.\n";

  return 0;
}
```

Ecco un esempio d'uso del programma:

```
Introdurre un nome: Rachele
Numero telefonico: 555-0809.
```

Nel programma, ogni voce della mappa è un array di caratteri contenente una stringa chiusa dal carattere nullo. Più avanti nel capitolo si vedrà un modo più semplice per scrivere questo programma impiegando il tipo standard string.

## 24.7 Gli algoritmi

Mentre i container forniscono il supporto per le proprie operazioni di base, gli algoritmi forniscono azioni più estese o complesse. Inoltre essi consentono di lavorare contemporaneamente su due tipi differenti. Per avere accesso agli algoritmi STL si deve includere nel programma l'header <algorithm>.

La libreria STL definisce un gran numero di algoritmi che sono riepilogati nella Tabella 24.5. Tutti gli algoritmi sono funzioni template. Questo significa che possono essere applicati a ogni tipo di container. Gli algoritmi STL verranno trattati nella Parte quarta di questo volume; le prossime sezioni mostrano solo alcuni esempi del loro uso.

### Conteggio

Una delle operazioni più semplici che è possibile eseguire su una sequenza è il conteggio. Per fare ciò si usa count() o count_if(). La loro forma generale è la seguente:



```
template <class InIter, class T>
    size_t count(InIter start, InIter end, const T &val);
template <class InIter, class UnPred>
    size_t count_if(InIter start, InIter end, UnPred pfn);
```

**Tabella 24.5** Gli algoritmi della libreria STL.

| ALGORITMO | USO |
|---|---|
| adjacent_find | Ricerca elementi adiacenti in una sequenza e restituisce un iteratore alla prima corrispondenza. |
| binary_search | Esegue una ricerca binaria su una sequenza ordinata. |
| copy | Copia una sequenza. |
| copy_backward | Come copy( ) tranne per il fatto che prima sposta gli elementi dalla fine della sequenza. |
| count | Restituisce il numero di elementi della sequenza. |
| count_if | Restituisce il numero di elementi della sequenza che soddisfano un predicato. |
| equal | Determina se due intervalli coincidono. |
| equal_range | Restituisce un intervallo in cui un elemento può essere inserito in una sequenza senza alterarne l'ordine. |
| fill e fill_n | Riempie un intervallo con il valore specificato. |
| find | Ricerca in un intervallo un valore e restituisce un iteratore alla prima occorrenza dell'elemento. |
| find_end | Ricerca in un intervallo una sottosequenza. Restituisce un iteratore alla fine della sottosequenza all'interno dell'intervallo. |
| find_first_of | Trova in una sequenza il primo elemento che corrisponde a un elemento di un intervallo. |
| find_if | Ricerca in un intervallo un elemento per il quale un predicato unario definito dall'utente restituisce true. |
| for_each | Applica una funzione a un intervallo di elementi. |
| generate e generate_n | Assegna agli elementi di un intervallo i valori prodotti da una funzione di generazione. |
| includes | Determina se una sequenza include tutti gli elementi di un'altra sequenza. |
| inplace_merge | Unisce un intervallo con un altro intervallo. Entrambi gli intervalli devono essere ordinati in senso ascendente. La sequenza risultante è ordinata. |
| iter_swap | Scambia i valori puntati dai due iteratori forniti come argomenti. |
| lexicographical_compare | Confronta alfabeticamente due sequenze. |
| lower_bound | Trova il primo punto della sequenza che non sia inferiore a un determinato valore. |

(segue)

**Tabella 24.5** Gli algoritmi della libreria STL. *(continua)*

| ALGORITMO | USO |
|---|---|
| make_heap | Costruisce uno heap da una sequenza. |
| max | Restituisce il maggiore di due valori. |
| max_element | Restituisce un iteratore all'elemento maggiore di un intervallo. |
| merge | Unisce due sequenze ordinate, inserendo il risultato in una terza sequenza. |
| min | Restituisce il minore di due valori. |
| min_element | Restituisce un iteratore all'elemento minore di un intervallo. |
| mismatch | Trova la prima differenza fra gli elementi di due sequenze. Restituisce gli iteratori ai due elementi. |
| next_permutation | Costruisce la successiva permutazione di una sequenza. |
| nth_element | Dispone una sequenza in modo che tutti gli elementi minori dell'elemento *E* si trovino prima di tale elemento e che tutti gli elementi maggiori di *E* seguano tale elemento. |
| partial_sort | Ordina un intervallo. |
| partial_sort_copy | Ordina un intervallo e poi copia il numero di elementi necessari per riempire la sequenza risultante. |
| partition | Dispone una sequenza in modo che tutti gli elementi per i quali un predicato restituisce true precedano quelli per cui il predicato restituisce false. |
| pop_heap | Scambia il primo elemento con il penultimo e poi ricostruisce lo heap. |
| prev_permutation | Costruisce la permutazione precedente di una sequenza. |
| push_heap | Inserisce un elemento alla fine dello heap. |
| random_shuffle | Casualizza una sequenza. |
| remove, remove_if, remove_copy e remove_copy_if | Rimuovono elementi dall'intervallo specificato. |
| replace, replace_copy, replace_if e replace_copy_if | Sostituisce gli elementi di un intervallo. |
| reverse e reverse_copy | Inverte l'ordine di un intervallo. |
| rotate e rotate_copy | Ruota a sinistra gli elementi in un intervallo. |
| search | Ricerca una sottosequenza in una sequenza. |
| search_n | Ricerca in una sequenza il numero specificato di elementi simili. |
| set_difference | Produce una sequenza che contiene la differenza fra due insiemi ordinati. |
| set_intersection | Produce una sequenza che contiene l'intersezione di due insiemi ordinati. |

(segue)



**Tabella 24.5** Gli algoritmi della libreria STL. *(continua)*

| ALGORITMO | USO |
|---|---|
| set_symmetric_difference | Produce una sequenza che contiene la differenza simmetrica fra due insiemi ordinati. |
| set_union | Produce una sequenza che contiene l'unione di due insiemi ordinati. |
| sort | Ordina un intervallo. |
| sort_heap | Ordina uno heap in un determinato intervallo. |
| stable_partition | Crea una sequenza in modo che tutti gli elementi per i quali un predicato restituisce true precedano quelli per cui il predicato restituisce false. Il partizionamento è stabile. Viene pertanto conservato l'ordinamento relativo della sequenza. |
| stable_sort | Ordina un intervallo. L'ordinamento è stabile. Pertanto gli elementi uguali non verranno spostati. |
| swap | Scambia due valori. |
| swap_ranges | Scambia gli elementi di un intervallo. |
| transform | Applica una funzione a un intervallo di elementi e memorizza il risultato in una nuova sequenza. |
| unique e unique_copy | Elimina gli elementi duplicati da un intervallo. |
| upper_bound | Trova l'ultimo punto della sequenza che non sia maggiore di un determinato valore. |

L'algoritmo count() restituisce il numero di elementi della sequenza compresi fra *start* ed *end* e corrispondenti a *val*. L'algoritmo count_if() restituisce il numero di elementi della sequenza compresi fra *start* ed *end* per i quali il predicato unario *pfn* restituisce il valore booleano true.

Il seguente programma illustra l'uso di count().

```
// Illustra l'uso di count().
#include <iostream>
#include <vector>
#include <cstdlib>
#include <algorithm>
using namespace std;

int main()
{
  vector<bool> v;
  int i;

  for(i=0; i < 10; i++) {
    if(rand() % 2) v.push_back(true);
    else v.push_back(false);
  }
```

```
  cout << "Sequenza =\n";
  for(i=0; i<v.size(); i++)
    cout << boolalpha << v[i] << " ";
  cout << endl;

  i = count(v.begin(), v.end(), true);
  cout << i << " elementi true.\n";

  return 0;
}
```

Il programma visualizza il seguente output:

```
Sequenza =
true true false false true false false false false false
3 elementi true.
```

Il programma inizia creando un vettore costituito da valori true e false generati casualmente. Quindi usa count() per conteggiare il numero di valori true.

Il seguente programma illustra l'uso di count_if(). Viene creato un vettore contenente i numeri da 1 a 19. Quindi conta i valori divisibili per 3. A tale scopo viene creato un predicato unario chiamato dividesBy3() che restituisce true quando l'argomento è divisibile per 3.

```
// Illustra l'uso di count_if().
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

/* Questo è un predicato unario che determina
   se il numero è divisibile per 3. */
bool dividesBy3(int i)
{
  if((i%3) == 0) return true;

  return false;
}

int main()
{
  vector<int> v;
  int i;
```



```
  for(i=1; i < 20; i++) v.push_back(i);

  cout << "Sequenza =\n";
  for(i=0; i<v.size(); i++)
    cout << v[i] << " ";
  cout << endl;

  i = count_if(v.begin(), v.end(), dividesBy3);
  cout << i << " numeri sono divisibili per 3.\n";

  return 0;
}
```

Questo programma produce il seguente output:

```
Sequenza =
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
6 numeri sono divisibili per 3.
```

Si deve notare il modo in cui è realizzato il predicato unario dividesBy3. Tutti i predicati unari ricevono come parametro un oggetto dello stesso tipo di quello contenuto nel container sul quale il predicato sta operando. Il predicato deve restituire un risultato true o false sulla base del valore dell'oggetto.

## Rimozione e sostituzione degli elementi

Talvolta è utile generare una nuova sequenza che è costituita da un numero ben deciso di elementi tratti da una sequenza originaria. Un algoritmo di questo tipo è remove_copy(). Ecco la sua forma generale:

```
template <class InIter, class OutIter, class T>
  OutIter remove_copy(InIter start, InIter end,
                      OutIter result, const T &val);
```

L'algoritmo remove_copy() copia gli elementi dall'intervallo specificato, rimuovendo quelli uguali a val. Il risultato viene inserito nella sequenza puntata da result e quindi viene restituito un iteratore che punta alla fine del risultato. Le dimensioni del container di destinazione devono essere sufficienti per contenere il risultato.

Per copiare una sequenza in un'altra sostituendo un elemento con un altro, si usa replace_copy().

La sua forma generale è:

template <class InIter, class OutIter, class T>
OutIter replace_copy(InIter *start*, InIter *end*,
OutIter *result*, const T *&old*, const T *&new*);

L'algoritmo replace_copy() copia gli elementi dall'intervallo specificato, sostituendo gli elementi uguali a old con new. Il risultato viene inserito nella sequenza puntata da result e quindi viene restituito un iteratore alla fine del risultato. Le dimensioni del container di destinazione devono essere sufficienti per contenere il risultato.

Il seguente programma illustra l'uso di remove_copy() e replace_copy(). In particolare il programma crea una sequenza di caratteri. Quindi rimuove dalla sequenza tutti gli spazi e li sostituisce con il carattere ":".

```
// Illustra l'uso di remove_copy e replace_copy.
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

int main()
{
  char str[] = "Programmare in STL è fantastico.";
  vector<char> v, v2(30);
  int i;

  for(i=0; str[i]; i++) v.push_back(str[i]);

  // **** illustra l'uso di remove_copy ****
  cout << "Sequenza di input =\n";
  for(i=0; i<v.size(); i++) cout << v[i];
  cout << endl;

  // rimozione degli spazi
  remove_copy(v.begin(), v.end(), v2.begin(), ' ');

  cout << "Risultato dopo la rimozione degli spazi =\n";
  for(i=0; i<v2.size(); i++) cout << v2[i];
  cout << endl << endl;

  // **** ora illustra l'uso di replace_copy ****
  cout << "Sequenza di input =\n";
  for(i=0; i<v.size(); i++) cout << v[i];
  cout << endl;

  // sostituisce gli spazi con il carattere due punti
  replace_copy(v.begin(), v.end(), v2.begin(), ' ', ':');

  cout << "Risultato dopo la sostituzione degli spazi con caratteri di due-
punti =\n";
  for(i=0; i<v2.size(); i++) cout << v2[i];
  cout << endl << endl;

  return 0;
}
```



Ecco l'output prodotto dal programma:

```
Sequenza di input =
Programmare in STL è fantastico.
Risultato dopo la rimozione degli spazi =
ProgrammareinSTLèfantastico.

Sequenza di input =
Programmare in STL è fantastico.
Risultato dopo la sostituzione degli spazi con caratteri di due-punti =
Programmare:in:STL:è:fantastico.
```

## Inversione di una sequenza

Uno degli algoritmi più utilizzati è reverse() che inverte una sequenza. La sua forma generale è:

template <class BiIter> void reverse(BiIter *start*, BiIter *end*);

L'algoritmo reverse() inverte l'ordine dell'intervallo degli elementi conmpresi fra *start* ed *end*. Il seguente programma illustra l'uso di reverse():

```
// Illustra l'uso di reverse.
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

int main()
```

```
{
  vector<int> v;
  int i;

  for(i=0; i<10; i++) v.push_back(i);

  cout << "Situazione iniziale = ";
  for(i=0; i<v.size(); i++) cout << v[i] << " ";
  cout << endl;

  reverse(v.begin(), v.end());

  cout << "Dopo l'inversione = ";
  for(i=0; i<v.size(); i++) cout << v[i] << " ";

  return 0;
}
```

Ecco l'output prodotto dal programma:

```
Situazione iniziale = 0 1 2 3 4 5 6 7 8 9
Dopo l'inversione = 9 8 7 6 5 4 3 2 1 0
```

## Trasformazione di una sequenza

Uno degli algoritmi più interessanti è transform() che modifica ciascun elemento di un intervallo sulla base di una funzione fornita a parte. L'algoritmo transform() ha le due forme generali seguenti:

```
template <class InIter, class OutIter, class Func)
   OutIter transform(InIter start, InIter end, OutIter result,
                     Func unaryfunc);
template <class InIter1, class InIter2, class OutIter, class Func)
   OutIter transform(InIter1 start1, InIter1 end1, InIter2 start2,
                     OutIter result, Func binaryfunc);
```

L'algoritmo transform() applica una funzione a un intervallo di elementi e memorizza il risultato in result. Nella prima forma, l'intervallo è specificato da *start* ed *end*. *unaryfunc* è la funzione da applicare. Questa funzione riceve come parametro il valore di un elemento e deve restituire la sua trasformazione. Nella seconda forma, la trasformazione viene applicata utilizzando una funzione operatore binaria che riceve come primo parametro il valore dell'elemento della sequenza che deve essere trasformato e come secondo parametro un elemento della



seconda sequenza. Entrambe le versioni restituiscono un iteratore che punta alla fine della sequenza prodotta.

Il seguente programma utilizza una semplice funzione di trasformazione chiamata reciprocal() che crea il reciproco di un numero. Si noti che la sequenza risultante viene memorizzata nella stessa lista contenente la sequenza originale.

```
// Uso dell'algoritmo transform.
#include <iostream>
#include <list>
#include <algorithm>
using namespace std;

// Una semplice funzione di trasformazione.
double reciprocal(double i) {
  return 1.0/i; // restituisce il reciproco
}

int main()
{

  list<double> vals;
  int i;

  // inserisce i valori in una lista
  for(i=1; i<10; i++) vals.push_back((double)i);

  cout << "Contenuto originale di vals =\n";
  list<double>::iterator p = vals.begin();
  while(p != vals.end()) {
    cout << *p << " ";
    p++;
  }

  cout << endl;

  // trasforma vals
  p = transform(vals.begin(), vals.end(),
                vals.begin(), reciprocal);

  cout << "Contenuto trasformato di vals =\n";
  p = vals.begin();
  while(p != vals.end()) {
    cout << *p << " ";
    p++;
  }
```

```
  return 0;
}
```

Ecco l'output prodotto dal programma:

```
Contenuto originale di vals =
1 2 3 4 5 6 7 8 9
Contenuto trasformato di vals =
1 0.5 0.333333 0.25 0.2 0.166667 0.142857 0.125 0.111111
```

Come si può vedere, ogni valore di vals è stato trasformato nel suo reciproco.

## 24.8 Uso degli oggetti funzione

Come si è detto all'inizio del capitolo, la libreria STL supporta (e utilizza pesantemente) gli oggetti funzione. Come si ricorderà gli oggetti funzione sono semplicemente classi che definiscono operator(). La libreria STL fornisce numerosi oggetti funzione già pronti, ad esempio less, minus e così via. Inoltre consente di definire i propri oggetti funzione. Per la verità la descrizione di tutti gli elementi riguardanti la creazione e l'uso degli oggetti funzione non rientra negli scopi di questo volume. Ma fortunatamente, come si è visto negli esempi precedenti, è possibile utilizzare la libreria STL senza dover creare neppure un oggetto funzione. Tuttavia, poiché gli oggetti funzione sono un ingrediente fondamentale della libreria STL, è importante quantomeno sapere a grandi linee il loro funzionamento.

### Oggetti funzione unari e binari

Così come esistono predicati unari e binari, esistono oggetti funzione unari e binari. Un oggetto funzione unario richiede un argomento mentre un oggetto funzione binario ne richiede due. Si deve utilizzare un oggetto del tipo richiesto. Ad esempio, se un algoritmo si attende un oggetto funzione binario, occorre passargli un oggetto funzione binario.

### Uso degli oggetti funzione forniti

La libreria STL fornisce un ricco assortimento di oggetti funzione. Ecco gli oggetti funzione binari forniti:

| | | | | |
|---|---|---|---|---|
| plus | minus | multiplies | divides | modulus |
| equal_to | not_equal_to | greater | greater_equal | less |



| | | |
|---|---|---|
| less_equal | logical_and | logical_or |

Ed ecco gli oggetti funzione unari:

| | |
|---|---|
| logical_not | negate |

Gli oggetti funzione svolgono le operazioni specificate dal loro nome. L'unico oggetto funzione il cui nome è poco chiaro è negate che inverte il segno del suo argomento.

Gli oggetti funzione forniti sono classi template che eseguono l'overloading di operator(), che restituisce il risultato dell'operazione specificata sul tipo di dati selezionato. Ad esempio, per richiamare l'oggetto funzione binario plus() si usa la seguente forma:

```
plus<float>()
```

Gli oggetti funzione forniti usano l'header <functional>.

Si può partire da un semplice esempio. Il seguente programma usa l'algoritmo transform() (descritto nella sezione precedente) e l'oggetto funzione negate() per invertire il segno di una lista di valori.

```
// Usa un oggetto funzione unario.
#include <iostream>
#include <list>
#include <functional>
#include <algorithm>
using namespace std;

int main()
{
  list<double> vals;
  int i;

  // inserisce i valori in una lista
  for(i=1; i<10; i++) vals.push_back((double)i);

  cout << "Contenuto originale di vals =\n";
  list<double>::iterator p = vals.begin();
  while(p != vals.end()) {
    cout << *p << " ";
    p++;
  }
  cout << endl;
```

```
  // usa l'oggetto funzione negate
  p = transform(vals.begin(), vals.end(),
                vals.begin(),
                negate<double>()); // richiama l'oggetto funzione

  cout << "Contenuto negato di vals =\n";
  p = vals.begin();
  while(p != vals.end()) {
    cout << *p << " ";
    p++;
  }

  return 0;
}
```

Il programma produce il seguente output:

```
Contenuto originale di vals =
1 2 3 4 5 6 7 8 9
Contenuto negato di vals =
-1 -2 -3 -4 -5 -6 -7 -8 -9
```

Nel programma, si noti il modo in cui viene richiamata negate(). Poiché vals è una lista di valori double, negate() viene richiamata utilizzando negate<double>(). L'algoritmo transform() richiama automaticamente negate() per ciascun elemento della sequenza. Pertanto l'unico parametro che negate() riceve come argomento è un elemento della sequenza.

Il prossimo programma illustra l'uso dell'oggetto funzione binario divides(). Questo programma crea due liste di valori double e poi divide l'uno per l'altro. Questo programma utilizza la forma binaria dell'algoritmo transform().

```
// Uso di un oggetto funzione binario.
#include <iostream>
#include <list>
#include <functional>
#include <algorithm>
using namespace std;

int main()
{
  list<double> vals;
  list<double> divisors;
  int i;
```



```
  // inserisce i valori in una lista
  for(i=10; i<100; i+10) vals.push_back((double)i);
  for(i=1; i<10; i++) divisors.push_back(3.0);

  cout << "Contenuto originale di vals =\n";
  list<double>::iterator p = vals.begin();
  while(p != vals.end()) {
    cout << *p << " ";
    p++;
  }

  cout << endl;

  // trasforma vals
  p = transform(vals.begin(), vals.end(),
                divisors.begin(), vals.begin(),
                divides<double>()); // richiama l'oggetto funzione

  cout << "Contenuto di vals diviso per 3 =\n";
  p = vals.begin();
  while(p != vals.end()) {
    cout << *p << " ";
    p++;
  }

  return 0;
}
```

Ecco l'output del programma:

```
Contenuto originale di vals =
10 20 30 40 50 60 70 80 90
Contenuto di vals diviso per 3 =
3.33333 6.66667 10 13.3333 16.6667 20 23.3333 26.6667 30
```

In questo caso, l'oggetto funzione binario divides() divide gli elementi della prima sequenza con i corrispondenti elementi della seconda sequenza. Pertanto divides() riceve gli argomenti nel seguente ordine:

```
divides(first, second)
```

Questo ordine può essere generalizzato. Ogni volta che viene utilizzato un oggetto funzione binario, i suoi argomenti vengono ordinati in questo modo.

## Creazione di un oggetto funzione

Oltre a utilizzare gli oggetti funzione forniti, è anche possibile crearne di nuovi. A tale scopo si deve semplicemente creare una classe che esegue l'overloading della funzione operator(). Tuttavia, per ottenere la massima flessibilità è opportuno utilizzare come classe base dell'oggetto funzione una delle seguenti classi definite dalla libreria STL.

```
template <class Argument, class Result> struct unary_function {
  typedef Argument argument_type;
  typedef Result result_type;
};

template <class Argument1, class Argument2, class Result>
struct binary_function {
  typedef Argument1 first_argument_type;
  typedef Argument2 second_argument_type;
  typedef Result result_type;
};
```

Queste classi template forniscono nomi di tipi concreti per i tipi generici utilizzati dall'oggetto funzione. Anche se questo tecnicamente è solo una comodità, si tratta di elementi molto utilizzati nella creazione di oggetti funzione.

Il seguente programma mostra la creazione di un oggetto funzione personalizzato. Il programma converte in un oggetto funzione la funzione reciprocal() (utilizzata per illustrare in precedenza il funzionamento dell'algoritmo transform()).

```
// Crea l'oggetto funzione reciprocal.
#include <iostream>
#include <list>
#include <functional>
#include <algorithm>
using namespace std;

// Un semplice oggetto funzione.
class reciprocal: unary_function<double, double> {
public:
  result_type operator()(argument_type i)
  {
    return (result_type) 1.0/i; // restituisce il reciproco
  }
};
```



```
int main()
{
  list<double> vals;
  int i;

  // inserisce i valori in una lista
  for(i=1; i<10; i++) vals.push_back((double)i);

  cout << "Contenuto originale di vals =\n";
  list<double>::iterator p = vals.begin();
  while(p != vals.end()) {
    cout << *p << " ";
    p++;
  }
  cout << endl;

  // usa l'oggetto funzione reciprocal
  p = transform(vals.begin(), vals.end(),
                vals.begin(),
                reciprocal()); // richiama l'oggetto funzione

  cout << "Contenuto trasformato di vals =\n";
  p = vals.begin();
  while(p != vals.end()) {
    cout << *p << " ";
    p++;
  }

  return 0;
}
```

Si devono notare due aspetti importanti di reciprocal(). Innanzitutto eredita la classe base unary_function(). Questo le dà accesso ai tipi argument_type e result_type. In secondo luogo definisce operator() in modo che restituisca il reciproco del suo argomento. In generale, per creare un oggetto funzione, basta ereditare dalla classe base corretta ed eseguire l'overloading di operator(). Dunque in realtà è tutto molto semplice.

## Uso dei binder

Quando si usa un oggetto funzione binario, è possibile collegare un valore a uno dei suoi argomenti. Questa possibilità può essere utile in molte situazioni. Ad esempio si potrebbe voler rimuovere da una sequenza tutti gli elementi maggiori di un determinato valore, ad esempio 8. A tale scopo occorre un modo per colle-

gare il numero 8 all'operando di destra dell'oggetto funzione greater(). Pertanto si vuole che greater() esegua il confronto:

```
val > 8
```

per ciascun elemento della sequenza. Per ottenere questo risultato, la libreria STL fornisce il meccanismo dei binder.

Esistono due binder: bind2nd() e bind1st(). Essi hanno la seguente forma generale:

bind1st(*binfunc_obj, value*)
bind2nd(*binfunc_obj, value*)

Qui, binfunc_obj è un oggetto funzione binario. bind1st() restituisce un oggetto funzione unario il cui operando di sinistra di binfunc_obj è collegato a value. bind2nd() restituisce un oggetto funzione unario il cui operando di destra di binfunc_obj è collegato a value. bind2nd() è il binder più comunemente utilizzato. In entrambi casi il risultato di un binder è un oggetto funzione unario che impiega la connessione al valore specificato. Per illustrare l'uso di un binder, verrà utilizza l'algoritmo remove_if(). Tale algoritmo rimuove da una sequenza gli elementi sulla base del risultato di un predicato. Ecco il suo prototipo:

```
template <class ForIter, class UnPred>
     ForIter remove_if(ForIter start, ForIter end, UnPred func);
```

L'algoritmo rimuove dalla sequenza gli elementi definiti da *start* ed *end* nel caso in cui il predicato unario definito da *func* sia *true*. L'algoritmo restituisce un puntatore alla nuova fine della sequenza, che riflette così la cancellazione degli elementi.

Il seguente programma rimuove da una sequenza tutti i valori maggiori di 8. Poiché il predicato richiesto da remove_if() è unario, non si può semplicemente utilizzare l'oggetto funzione greater() così come è poiché greater() è un oggetto binario. Al contrario si deve collegare il valore 8 al secondo parametro di greater() impiegando il binder bind2nd() nel modo illustrato dal programma.

```
// Illustra l'uso di bind2nd().
#include <iostream>
#include <list>
#include <functional>
#include <algorithm>
using namespace std;

int main()
{
  list<int> lst;
  list<int>::iterator p, endp;

  int i;

  for(i=1; i < 20; i++) lst.push_back(i);

  cout << "Sequenza originale:\n";
  p = lst.begin();
  while(p != lst.end()) {
    cout << *p << " ";
    p++;
  }
  cout << endl;

  endp = remove_if(lst.begin(), lst.end(),
                   bind2nd(greater<int>(), 8));

  cout << "Sequenza risultante:\n";
  p = lst.begin();
  while(p != endp) {
    cout << *p << " ";
    p++;
  }

  return 0;
}
```

Ecco l'output prodotto dal programma:

```
Sequenza originale:
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
Sequenza risultante:
1 2 3 4 5 6 7 8
```

Si consiglia di sperimentare l'uso di questo programma, provando con oggetti funzione differenti e collegando valori differenti. Come si scoprirà, i binder estendono le potenzialità della libreria STL in vari modi.

Un'ultima annotazione: strettamente connesso ai binder vi è un oggetto chiamato negatore. I negatori sono not1() e not2() e restituiscono la negazione (ovvero il complemento) di ciò che viene fornito come predicato. Ecco la loro forma generale:

not1(*unary_predicate*)
not2(*binary_predicate*)

Ad esempio, se si sostituisce la riga:

```
endp = remove_if(lst.begin(), lst.end(),
                 not1(bind2nd(greater<int>(), 8)));
```

nel programma precedente, verranno rimossi da lst tutti gli elementi che non sono maggiori di 8.

## 24.9 La classe string

Come si sa, il linguaggio C++ non supporta direttamente un tipo per le stringhe. Tuttavia fornisce due modi per gestire le stringhe: innanzitutto si può usare un normale array di caratteri chiuso dal carattere nullo, che ormai dovrebbe essere familiare. In genere questo tipo di array è chiamato "stringa C". Il secondo modo è costituito dalla classe string; quest'ultimo sarà l'approccio che verrà esaminato ora.

In realtà la classe string è una specializzazione di una classe template più generale chiamata basic_string. Infatti esistono due specializzazioni di basic_string: string che supporta stringhe di caratteri da 8 bit e wstring che supporta stringhe di caratteri estesi. Poiché in programmazione vengono utilizzati prevalentemente caratteri a 8 bit, in questa sezione verrà esaminata solo la classe string.

Prima di parlare della classe string è importante comprendere perché fa parte della libreria C++. Le classi standard non sono state aggiunte al linguaggio C++ in modo casuale. Infatti ogni nuova aggiunta è stata accompagnata da grandi discussioni e dibattiti. Poiché il linguaggio C++ conteneva già un certo supporto per le stringhe nella forma degli array di caratteri chiusi dal carattere nullo, a prima vista potrebbe sembrare che l'inclusione della classe string rappresenti un'eccezione a questa regola. La verità è molto diversa, ecco perché: le stringhe chiuse dal carattere nullo non possono essere manipolate da nessuno degli operatori standard del linguaggio C++ né possono entrare nelle normali espressioni C++. Ad esempio, si consideri il seguente frammento di codice:

```
char s1[80], s2[80], s3[80];

s1 = "Alfa"; // non si può fare
s2 = "Beta"; // non si può fare
s3 = s1 + s2; // errore, non consentito
```



Come si può vedere dai commenti, in C++ non è possibile utilizzare l'operatore di assegnamento per dare un nuovo valore a un array di caratteri (ad eccezione solo dell'inizializzazione) né è possibile utilizzare l'operatore + per concatenate due stringhe. Queste operazioni devono essere scritte utilizzando apposite funzioni della libreria:

```
strcpy(s1, "Alfa");
strcpy(s2, "Beta");
strcpy(s3, s1);
strcat(s3, s2);
```

Poiché gli array di caratteri chiusi dal carattere nullo non sono tecnicamente tipi, non è possibile applicargli gli operatori C++. Questo complica anche la più rudimentale operazione sulle stringhe. Soprattutto è l'impossibilità di operare su stringhe chiuse dal carattere nullo con gli operatori C++ standard che ha portato allo sviluppo di una classe per stringhe standard. Si ricordi che quando si definisce una classe in C++, si definisce a tutti gli effetti un nuovo tipo di dati che può essere integrato completamente nell'ambiente C++. Questo significa ovviamente che gli operatori possono essere modificati in modo da operare sulla nuova classe. Pertanto, aggiungendo una classe standard per stringhe diventa possibile gestire le stringhe come si fa con ogni altro tipo di dati: tramite gli operatori.

Vi è anche un altro motivo che ha spinto a realizzare una classe standard per le stringhe: la sicurezza. Nelle mani di un programmatore inesperto o poco accorto, è molto facile superare i limiti di un array contenente una stringa chiusa dal carattere nullo. Ad esempio, si consideri la funzione strcpy() che esegue la copia di una stringa. Questa funzione non controlla in alcun modo se vengono superati i limiti dell'array di destinazione. Se l'array di origine contiene più caratteri di quanti possano essere inseriti nell'array di destinazione, può nascere un errore del programma o un blocco di sistema. Come si vedrà, la classe standard per le stringhe previene questo errore.

Nell'analisi finale, vi sono tre motivi che hanno spinto a includere nello standard una classe per le stringhe: l'uniformità (ora la stringa è un tipo), la comodità (si possono usare gli operatori standard del linguaggio) e la sicurezza (non possono essere superati i limiti dell'array). Si deve tenere in considerazione che non vi è alcun motivo che costringa ad abbandonare completamente le normali stringhe chiuse dal carattere nullo che rappresentano tuttora il modo più efficiente per implementare le stringhe. Tuttavia, quando la velocità non è un problema, la possibilità di usare la nuova classe string dà accesso a un modo più sicuro e integrato di gestire le stringhe.

Anche se tradizionalmente questo argomento non è collegato alla libreria STL, la classe string è non è altro che un nuovo container definito dal C++. Questo significa che supporta gli algoritmi descritti nella sezione precedente. Tuttavia le

stringhe offrono anche altre funzionalità. Per avere accesso alla classe string si deve includere nel programma l'header <string>.

La classe string è molto molto estesa ed è dotata di numerosi costruttori e funzioni membro. Inoltre molte funzioni membro sono dotate di varie forme create tramite overloading. Per questo motivo non è possibile esaminare in un capitolo l'intero contenuto di string. Dunque si parlerà semplicemente delle sue caratteristiche più utilizzate. Dopo aver compreso il funzionamento della classe string, non sarà difficile esplorare tutte le altre funzionalità di cui essa è dotata.

La classe string supporta numerosi costruttori. Ecco i prototipi dei tre costruttori più utilizzati:

```
string( );
string(const char *str);
string(const string &str);
```

La prima forma crea un oggetto string vuoto. La seconda forma crea un oggetto string utilizzando come base una stringa chiusa dal carattere nullo puntata da str. Questa forma fornisce una conversione da una stringa chiusa dal carattere nullo alla classe string. La terza forma crea una stringa partire da un'altra stringa.

Per gli oggetti string sono definiti numerosi operatori fra i quali:

| OPERATORE | SIGNIFICATO |
|---|---|
| = | Assegnamento |
| + | Concatenamento |
| += | Concatenamento e assegnamento |
| == | Uguaglianza |
| != | Disuguaglianza |
| < | Minore di |
| <= | Minore o uguale a |
| > | Maggiore di |
| >= | Maggiore o uguale a |
| [] | Indici |
| << | Output |
| >> | Input |



Questi operatori consentono di utilizzare gli oggetti stringa nelle normali espressioni ed evitano di dover richiamare funzioni come strcopy() o strcat(). In generale è possibile utilizzare nelle espressioni gli oggetti string e le normali stringhe chiuse dal carattere nullo. Ad esempio a un oggetto string si può assegnare una stringa chiusa dal carattere nullo.

L'operatore + concatena due oggetti string oppure un oggetto string e una stringa C. Dunque sono consentite le seguenti varianti:

```
string + string
stringa + stringa C
stringa C + string
```

L'operatore + può essere utilizzato anche per concatenate un carattere alla fine di una stringa.

La classe string definisce la costante npos uguale a -1. Questa costante rappresenta la lunghezza massima di una stringa.

La classe per stringhe C++ semplifica straordinariamente la gestione delle stringhe. Ad esempio, utilizzando oggetti string è possibile impiegare l'operatore di assegnamento per assegnare a una stringa una stringa quotata, l'operatore + per concatenate le stringhe e gli operatori di confronto per confrontare le stringhe. Queste operazioni sono illustrate dal seguente programma:

```
// Un breve esempio d'uso delle stringhe.
#include <iostream>
#include <string>
using namespace std;

int main()
{
  string str1("Alfa");
  string str2("Beta");
  string str3("Omega");
  string str4;

  // assegna una stringa
  str4 = str1;
  cout << str1 << "\n" << str3 << "\n";

  // concatena due stringhe
  str4 = str1 + str2;
  cout << str4 << "\n";

  // concatena una stringa e una stringa C
  str4 = str1 + " = " + str3;
```

```
  cout << str4 << "\n";

  // confronta le stringhe
  if(str3 > str1) cout << "str3 > str1\n";
  if(str3 == str1+str2)
    cout << "str3 == str1+str2\n";

  /* A un oggetto stringa può essere assegnata
     anche una normale stringa. */
  str1 = "Questa è una stringa chiusa dal carattere nullo.\n";
  cout << str1;

  // crea un oggetto stringa a partire da un altro oggetto stringa
  string str5(str1);
  cout << str5;

  // legge una stringa
  cout << "Introdurre una stringa: ";
  cin >> str5;
  cout << str5;

  return 0;
}
```

Il programma produce il seguente output:

```
Alfa
Omega
AlfaBeta
Alfa - Omega
str3 > str1
Questa è una stringa chiusa dal carattere nullo.
Questa è una stringa chiusa dal carattere nullo.
Introdurre una stringa: STL
STL
```

Si noti la facilità di gestione della stringa. Ad esempio, per concatenare le stringhe si usa l'operatore + e per confrontare due stringhe si usa >. Per ottenere queste stesse operazioni con le stringhe C, sarebbe necessario ricorrere alle funzioni strcat() e strcmp(). Poiché gli oggetti string del linguaggio C++ possono essere utilizzati tranquillamente insieme alle normali stringhe C, non vi è alcuno svantaggio a impiegarle nel programma, anzi si otterranno notevoli vantaggi.

Vi è un'ultima cosa da notare nel programma precedente: le dimensioni delle stringhe non vengono specificate. Gli oggetti di tipo string possono essere



dimensionati automaticamente per contenere la stringa fornita. Pertanto, quando si assegnano o concatenano le stringhe, le dimensioni della stringa di destinazione cresceranno automaticamente per accogliere la nuova stringa. Dunque non è possibile superare i limiti di una stringa. Questo aspetto dinamico degli oggetti string è una delle caratteristiche che la distinguono rispetto alle stringhe C che sono soggette a errori dovuti al superamento dei limiti dell'array.

## Alcune delle funzioni membro di string

Mentre gli operatori consentono di eseguire sulle stringhe le operazioni più semplici, quando occorre eseguire operazioni più complesse vengono impiegate apposite funzioni membro. La classe string contiene un numero elevatissimo di funzioni membro e dunque non è possibile descriverle tutte. In questo capitolo verranno esaminate solo le più comuni.

### Manipolazione di base delle stringhe

Per assegnare una stringa a un'altra si usa la funzione assign(). Ecco due delle sue forme:

```
string &assign(const string &strob, size_type start, size_type num);
string &assign(const char *str, size_type num);
```

Nella prima forma, all'oggetto chiamante vengono assegnati *num* caratteri della stringa *strob* a partire dall'indice specificato da *start*. Nella seconda forma, all'oggetto chiamante vengono assegnati i primi *num* caratteri della stringa C *str*. Entrambe le funzioni restituiscono l'indirizzo dell'oggetto chiamante. Naturalmente è molto più comodo utilizzare = per assegnare una stringa a un'altra. La funzione assign() viene impiegata solamente per assegnare una parte di una stringa.

La funzione membro append() aggiunge a una stringa una parte di un'altra stringa. Ecco due delle sue forme:

```
string &append(const string &strob, size_type start, size_type num);
string &append(const char *str, size_type num);
```

Qui, all'oggetto chiamante vengono aggiunti *num* caratteri tratti da *strob* a partire dall'indice specificato da *start*. Nella seconda forma all'oggetto chiamante vengono aggiunti i primi *num* caratteri della stringa C *str*. In entrambi i casi viene restituito l'indirizzo dell'oggetto chiamante. Naturalmente è molto più facile utilizzare l'operatore + per aggiungere un'intera stringa a un'altra e dunque si usa la funzione append() solo quando si deve aggiungere una parte della stringa.

Per inserire o sostituire i caratteri di una stringa si usano le funzioni insert() e replace(). Ecco i loro prototipi più utilizzati:

```
string &insert(size_type start, const string &strob);
string &insert(size_type start, const string &strob,
               size_type insStart, size_type num);
string &replace(size_type start, size_type num, const string &strob);
string &replace(size_type start, size_type orgNum, const string &strob,
                size_type replaceStart, size_type replaceNum);
```

La prima forma di insert() inserisce *strob* nella stringa chiamante all'indice specificato da *start*. La seconda forma di insert() inserisce nella stringa chiamante all'indice specificato da *start num* carattere tratti dalla stringa strob a partire da *insStart*.

La prima forma di replace() sostituisce *num* caratteri dalla stringa chiamante a partire da *start* con il contenuto della stringa *strob*. La seconda forma sostituisce *orgNum* caratteri a partire da *start* con *replaceNum* caratteri della stringa specificata da *strob* partendo da *replaceStart*. In entrambi i casi viene restituito l'indirizzo dell'oggetto chiamante.

Per rimuovere caratteri da una stringa si usa erase(). Ecco una delle sue forme più comuni:

```
string &erase(size_type start = 0, size_type num = npos);
```

Tale funzione rimuove dalla stringa chiamante *num* caratteri a partire da *start*. La funzione restituisce l'indirizzo della stringa chiamante.

Il seguente programma illustra l'uso delle funzioni insert(), erase() e replace().

```
// Illustra l'uso di insert(), erase() e replace().
#include <iostream>
#include <string>
using namespace std;

int main()
{
  string str1("La potenza delle stringhe C++.");
  string str2("Le potenzialità della libreria STL");

  cout << "Le stringhe iniziali\n";
  cout << "str1: " << str1 << endl;
  cout << "str2: " << str2 << "\n\n";

  // illustra l'uso di insert()
  cout << "Inserisce str2 in str1:\n";
  str1.insert(6, str2);
  cout << str1 << "\n\n";

  // illustra l'uso di erase()
  cout << "Rimuove 34 caratteri da str1:\n";
  str1.erase(6, 34);
  cout << str1 <<"\n\n";

  // illustra l'uso di replace
  cout << "Sostituisce 8 caratteri di str1 con str2:\n";
  str1.replace(7, 8, str2);
  cout << str1 << endl;

  return 0;
}
```

Ecco l'output prodotto da questo programma:

```
Le stringhe iniziali
str1: La potenza delle stringhe C++.
str2: Le potenzialità della libreria STL

Inserisce str2 in str1:
La potLe potenzialità della libreria STLenza delle stringhe C++.

Rimuove 34 caratteri da str1:
La potenza delle stringhe C++.

Sostituisce 8 caratteri di str1 con str2:
La poteLe potenzialità della libreria STLe stringhe C++.
```

### Ricerca di una stringa

La classe string fornisce numerose funzioni membro che consentono di eseguire ricerche all'interno di una stringa. Fra di esse vi sono find() e rfind(). Ecco i prototipi delle più comuni versioni di queste funzioni:

```
size_type find(const string &strob, size_type start=0) const;
size_type rfind(const string &strob, size_type start=npos) const;
```

Partendo da *start*, find() ricerca nella stringa chiamante la prima occorrenza della stringa contenuta in *strob*. Se la trova restituisce l'indice in cui è stata trovata la stringa. Se invece la stringa non viene trovata, restituisce *npos*. rfind() è l'opposto di find(). A partire da *start* ricerca all'inverso nella stringa chiamante la prima occorrenza della stringa contenuta in *strob* (ovvero ricerca l'ultima occorrenza di *strob* nella stringa chiamante). Se la trova, rfind() restituisce l'indice in cui è stata trovata la corrispondenza, altrimenti restituisce *npos*.

Ecco un breve esempio che utilizza find() e rfind().

```
#include <iostream>
#include <string>
using namespace std;

int main()
{
  int i;
  string s1 =
    "Veloce di mente, forte di corpo, puro di cuore";
  string s2;

  i = s1.find("veloce");
  if(i!=string::npos) {
    cout << "Trovato in " << i << endl;
    cout << "La stringa rimanente è:\n";
    s2.assign(s1, i, s1.size());
    cout << s2;
  }
  cout << "\n\n";

  i = s1.find("forte");
  if(i!=string::npos) {
    cout << "Trovato in " << i << endl;
    cout << "La stringa rimanente è:\n";
    s2.assign(s1, i, s1.size());
    cout << s2;
  }
  cout << "\n\n";

  i = s1.find("puro");
  if(i!=string::npos) {
    cout << "Trovato in " << i << endl;
    cout << "La stringa rimanente è:\n";
    s2.assign(s1, i, s1.size());
    cout << s2;
  }
  cout << "\n\n";

  // trova l'ultimo "di"
  i = s1.rfind("di");
  if(i!=string::npos) {
    cout << "Trovato in " << i << endl;
    cout << "La stringa rimanente è:\n";
    s2.assign(s1, i, s1.size());
    cout << s2;
  }
```



```
  return 0;
}
```

Ecco l'output prodotto dal programma:

```
Trovato in 0
La stringa rimanente è:
Veloce di mente, forte di corpo, puro di cuore

Trovato in 18
La stringa rimanente è:
forte di corpo, puro di cuore

Trovato in 34
La stringa rimanente è:
puro di cuore

Trovato in 39
La stringa rimanente è:
di cuore
```

### Confronto fra stringhe

Per confrontare l'intero contenuto di una stringa con il contenuto di un'altra, in genere vengono impiegati gli operatori relazionali descritti in precedenza. Ma se si deve confrontare solo una porzione di una stringa, è necessario impiegare la funzione membro compare(), descritta di seguito:

int compare(size_type *start*, size_type *num*, const string &*strob*) const;

Qui il confronto viene eseguito fra *num* caratteri tratti da *strob* a partire da *start* e la stringa chiamante. Se la stringa chiamante è minore, compare() restituisce un valore minore di 0. Se la stringa chiamante è maggiore di *strob*, restituisce un valore maggiore di 0. Se *strob* e la stringa chiamante sono uguali, compare() restituisce il valore 0.

### Come ottenere una stringa chiusa dal carattere nullo

Anche se gli oggetti string sono estremamente utili, in alcuni casi occorre ottenere la versione C di una stringa. Ad esempio, si potrebbe utilizzare un oggetto string per costruire il nome di un file. Tuttavia, quando si apre un file, occorre specificare un puntatore a una normale stringa C. Per risolvere questo problema, si può utilizzare la funzione membro c_str(). Ecco il suo prototipo:

const char *c_str( ) const;

Questa funzione restituisce un puntatore alla versione chiusa dal carattere nullo della stringa contenuta nell'oggetto string chiamante. La stringa chiusa dal carattere nullo non deve essere modificata. Inoltre non si può garantire che sia valida dopo che sull'oggetto string sono state eseguite altre operazioni.

## Le stringhe sono container

La classe string risponde a tutti i requisiti dei container. Pertanto supporta tutte le funzioni dei container, come begin(), end() e size(). Inoltre supporta gli iteratori. Pertanto un oggetto string può essere manipolato anche tramite algoritmi STL. Ecco un semplice esempio:

```
// le stringhe come container.
#include <iostream>
#include <string>
#include <algorithm>
using namespace std;

int main()
{
  string str1("La gestione delle stringhe in C++ è semplice");
  string::iterator p;
  int i;

  // usa size()
  for(i=0; i<str1.size(); i++)
    cout << str1[i];
  cout << endl;

  // usa un iteratore
  p = str1.begin();
  while(p != str1.end())
    cout << *p++;
  cout << endl;

  // usa l'algoritmo count()
  i = count(str1.begin(), str1.end(), 'i');
  cout << "Vi sono " << i << " 'i' in str1\n";

  // usa transform() per trasformare la stringa in lettere maiuscole
  transform(str1.begin(), str1.end(), str1.begin(),
            toupper);
  p = str1.begin();
  while(p != str1.end())
    cout << *p++;
  cout << endl;

  return 0;
}
```



Ecco l'output del programma:

```
La gestione delle stringhe in C++ è semplice
La gestione delle stringhe in C++ è semplice
Vi sono 4 'i' in str1
LA GESTIONE DELLE STRINGHE IN C++ È SEMPLICE
```

## Inserimento di stringhe in altri container

Anche se string è un container, gli oggetti di tipo string vengono comunemente memorizzati in altri container STL, come ad esempio le mappe o le liste. Ad esempio, ecco un modo migliore per scrivere il programma per rubriche telefoniche descritto in precedenza. In questo caso al posto delle stringhe C vengono utilizzati oggetti di tipo string.

```
// Usa una mappa di stringhe per creare una rubrica telefonica.
#include <iostream>
#include <map>
#include <string>
using namespace std;

int main()
{
  map<string, string> directory;

  directory.insert(pair<string, string>("Tommaso", "555-4533"));
  directory.insert(pair<string, string>("Cristina", "555-9678"));
  directory.insert(pair<string, string>("Giovanni", "555-8195"));
  directory.insert(pair<string, string>("Rachele", "555-0809"));

  string s;
  cout << "Introdurre un nome: ";
  cin >> s;

  map<string, string>::iterator p;

  p = directory.find(s);
```

```
  if(p != directory.end())
    cout << "Numero telefonico: " << p->second;
  else
    cout << "Nome non presente.\n";

  return 0;
}
```

## 24.10 Commenti finali sulla libreria STL

La libreria STL è diventata un elemento fondamentale del linguaggio C++. Dunque ora molte operazioni di programmazione vengono normalmente svolte impiegando tale libreria. STL combina ottime doti di potenzialità e flessibilità e, seppure la sua sintassi sia un po' complessa, il suo uso è molto semplice. Nessun programmatore C++ può ignorare la libreria STL poiché essa giocherà in futuro un ruolo sempre più importante nella realizzazione dei programmi.

Parte terza

# LA LIBRERIA DI FUNZIONI STANDARD

Il C++ definisce due tipi di librerie. La prima è la memoria delle funzioni standard. Questa libreria è costituita da funzioni indipendenti di utilizzo generale che non fanno parte di nessuna classe. Il C++ eredita la libreria di funzioni dal C. La seconda è la libreria di classi per la programmazione a oggetti. La Parte terza di questo volume costituisce una guida di riferimento per la libreria di funzioni standard. La Parte quarta descrive la libreria delle classi.

La libreria delle funzioni può essere suddivisa nelle seguenti categorie:

- funzioni di I/O;
- funzioni per la gestione di stringhe e caratteri;
- funzioni matematiche;
- funzioni per l'ora, la data e la localizzazione;
- funzioni di allocazione dinamica della memoria;
- funzioni varie;
- funzioni per caratteri estesi

L'ultima categoria è stata aggiunta al C standard nel 1995 e successivamente è stata incorporata nel C++. Essa contiene gli equivalenti a caratteri estesi (wchar_t) per molte funzioni di libreria. Onestamente l'uso della libreria per caratteri estesi è molto limitato e il linguaggio C++ fornisce un modo migliore per gestire ambienti a caratteri estesi ma per completezza l'argomento verrà descritto nel Capitolo 31.

Un'ultima annotazione: tutti i compilatori forniscono molte più funzioni rispetto a quelle definite dallo standard sia per il C che per il C++. In genere queste funzioni aggiuntive si occupano dell'interfacciamento con il sistema operativo e di altre operazioni relative all'ambiente. Per informazioni si rimanda alla documentazione del compilatore.

Capitolo 25

# Le funzioni di I/O basate sul C

Questo capitolo descrive le funzioni di I/O tipiche del linguaggio C. Queste funzioni sono definite sia nel C standard che nel C++ standard. Anche se quando si deve realizzare un nuovo programma in genere si utilizza il sistema di I/O a oggetti C++, in realtà nulla impedisce di utilizzare in un programma C++ le funzioni di I/O C, quando questo sembri appropriato. Le funzioni contenute in questo capitolo sono state specificate innanzitutto dallo standard ANSI per il C e vengono generalmente chiamate "il sistema di I/O ANSI C".

L'header associato alle funzioni di I/O C è <stdio> (un programma C deve utilizzare il file header <stdio.h>). Questo header definisce varie macro e vari tipi utilizzati dal file system. Il tipo più importante è FILE, utilizzato per dichiarare un puntatore a file. Gli altri tipi sono size_t e fpos_t. Il tipo size_t (corrispondente in genere a qualche forma di intero senza segno) definisce un oggetto in grado di contenere le dimensioni del più grande file consentito dal sistema operativo. Il tipo fpos_t definisce un oggetto che può contenere tutte le informazioni necessarie per specificare in modo univoco ogni posizione all'interno di un file. La macro più utilizzata fra quelle definite nei file header è EOF il cui valore indica la fine del file.

Molte delle funzioni di I/O in caso di errore impostano la variabile intera globale standard errno. Il programma può controllare questa variabile per ottenere informazioni più approfondite sull'errore. I valori che possono essere contenuti in errno dipendono dall'implementazione.

Per una panoramica sul sistema di I/O C, consultare i Capitoli 8 e 9 nella Parte prima.

**NOTA** *Questo capitolo descrive le funzioni di I/O a caratteri. Si tratta di funzioni originariamente definite per il C e C++ standard e sono anche quelle di gran lunga più utilizzate. Nel 1995, a questo gruppo sono state aggiunte numerose funzioni per caratteri estesi (wchar_t) per verranno descritte brevemente nel Capitolo 31.*

## clearerr()

```
#include <cstdio>
void clearerr(FILE *stream);
```

La funzione clearerr() esegue il reset (ovvero imposta a 0) del flag per errori associato allo stream. Viene anche eseguito il reset dell'indicatore di fine file.

I flag di errore degli stream vengono inizialmente impostati a 0 da una chiamata eseguita con successo alla funzione fopen(). Quando si verifica un errore, i flag rimangono attivi fino a un'esplicita chiamata a clearerr() oppure fino all'esecuzione di rewind().

Gli errori su file possono verificarsi per varie ragioni, molte delle quali dipendono dal sistema. L'esatta natura dell'errore può essere determinata richiamando perror() che visualizza l'errore verificatosi.

Le funzioni correlate sono feof(), ferror() e perror().

## fclose()

```
#include <cstdio>
int fclose(FILE *stream);
```

La funzione fclose() chiude il file associato a stream e ne vuota il buffer. Dopo aver eseguito fclose(), stream non è più connesso al file e ogni buffer allocato dinamicamente viene deallocato.

Se fclose() ha successo, restituisce il valore 0, altrimenti restituisce EOF. Il tentativo di chiudere un file già chiuso produce un errore. Anche la rimozione del dispositivo di memorizzazione prima di chiudere un file genera un errore, così come la mancanza di spazio su disco.

Le funzioni correlate sono fopen(), freopen() e fflush().

## feof()

```
#include <cstdio>
int feof(FILE *stream);
```

La funzione feof() controlla la posizione dell'indicatore di posizione del file per determinare se è stata raggiunta la fine del file associato allo stream. Se l'indicatore di posizione del file è alla fine del file, viene restituito un valore diverso da 0, altrimenti viene restituito il valore 0.

Dopo che è stata raggiunta la fine del file, tutte le successive operazioni di lettura restituiscono EOF finché non verrà richiamata rewind() o finché la posizione corrente all'interno del file non verrà spostata con fseek().



La funzione feof() è particolarmente utile quando si opera su file binari poiché l'indicatore di fine file è anche un intero binario valido. È opportuno eseguire chiamate esplicite a feof() piuttosto che controllare il valore restituito da getc(); nel caso di file binari, questo è l'unico modo per garantire con sicurezza se è stata raggiunta la fine del file.

Le funzioni correlate sono clearerr(), ferror(), perror(), putc() e getc().

## ferror()

```
#include <cstdio>
int ferror(FILE *stream);
```

La funzione ferror() controlla se si è verificato un errore sul file associato allo stream specificato. Se il valore restituito è 0, significa che non si è verificato alcun errore, mentre un valore diverso da 0 indica che si è verificato un errore.

I flag di errore associati a stream rimangono attivi finché il file non viene chiuso o finché non viene richiamata la funzione clearerr().

Per determinare l'esatta natura dell'errore si usa la funzione perror().

Le funzioni correlate sono clearerr ), feof() e perror().

## fflush()

```
#include <cstdio>
int fflush(FILE *stream);
```

Se stream è associato a un file aperto in scrittura, una chiamata a fflush() provoca la scrittura fisica sul file del contenuto del buffer di output. Se stream punta a un file di input, il contenuto del buffer di input viene cancellato. In entrambi i casi, il file rimane aperto.

Se il valore restituito è zero, l'operazione è stata completata con successo; EOF indica che si è verificato un errore di scrittura.

I buffer vengono svuotati automaticamente alla normale terminazione del programma oppure quando sono completamente pieni. Anche la chiusura del file provoca lo svuotamento del suo buffer.

Le funzioni correlate sono fclose(), fopen(), fread(), fwrite(), getc() e putc().

## fgetc()

```
#include <cstdio>
int fgetc(FILE *stream);
```

La funzione fgetc() restituisce il prossimo carattere contenuto nello stream di input a partire dalla posizione corrente e incrementa l'indicatore di posizione del file. Il carattere viene letto come un unsigned char che viene convertito in un intero.

Quando viene raggiunta la fine del file, fgetc() restituisce EOF. Ma poiché EOF è un valore intero valido, quando si lavora su file binari, per controllare se è stata raggiunta la fine del file si deve usare feof(). fgetc() restituisce EOF anche se incontra un errore. Per controllare se si sono verificati errori quando si opera su un file binario, si deve usare ferror().

Le funzioni correlate sono fputc(), getc(), putc() e fopen().

## fgetpos()

```
#include <cstdio>
int fgetpos(FILE *stream, fpos_t *position);
```

La funzione fgetpos() memorizza il valore corrente dell'indicatore di posizione del file nell'oggetto puntato da *position*. L'oggetto puntato da *position* deve essere di tipo fpos_t. Il valore così memorizzato può essere utile solo nelle successive chiamate a fsetpos().

Se si verifica un errore, fgetpos() restituisce un valore diverso da 0, altrimenti restituisce 0.

Le funzioni correlate sono fsetpos(), fseek() e ftell().

## fgets()

```
#include <cstdio>
char *fgets(char *str, int num, FILE *stream);
```

La funzione fgets() legge fino a *num* - 1 caratteri da stream e li inserisce nell'array di caratteri puntato da *str*. I caratteri vengono letti fino a che non viene ricevuto un codice di fine riga o EOF o finché non viene raggiunto il limite specificato. Dopo la lettura dei caratteri, nell'array viene inserito un codice nullo immediatamente dopo l'ultimo carattere letto. Il carattere di fine riga verrà mantenuto ed entrerà a far parte dell'array puntato da *str*.

In caso di successo, fgets() restituisce *str*; in caso di fallimento restituisce un puntatore nullo. Se si verifica un errore di lettura, il contenuto dell'array puntato da *str* sarà indeterminato. Poiché il puntatore nullo viene restituito sia quando si verifica un errore sia alla fine del file, per determinare ciò che è accaduto si deve usare feof() o ferror().

Le funzioni correlate sono fputs(), fgetc(), gets() e puts().



## fopen()

```
#include <cstdio>
FILE *fopen(const char *fname, const char *mode);
```

La funzione fopen() apre un file il cui nome è puntato da *name* e restituisce lo stream che gli viene associato. Le operazioni consentite sul file sono definite dal valore di *mode*.

I valori consentiti per *mode* sono elencati nella Tabella 25.1. Il nome del file deve essere una stringa di caratteri contenente un nome di file valido così come è definita dal sistema operativo e può includere il percorso (sempre che l'ambiente supporti questa possibilità).

Se fopen() riesce ad aprire il file specificato, restituisce un puntatore FILE. Se il file non può essere aperto, restituisce un puntatore nullo.

Come si può vedere nella tabella, un file può essere aperto in modalità testo o in modalità binaria. In modalità testo si verificano alcune traduzioni dei caratteri. Ad esempio i codici di fine riga possono essere convertiti in sequenze carriage Return / Linee Feed. Sui file binari tali traduzioni non avvengono.

**Tabella 25.1** I valori consentiti per il parametro mode di fopen().

| MODALITÀ | SIGNIFICATO |
|---|---|
| "r" | Apre un file di testo in lettura. |
| "w" | Crea un file di testo in scrittura. |
| "a" | Aggiunta a un file di testo. |
| "rb" | Apre un file binario in lettura. |
| "wb" | Crea un file binario in scrittura. |
| "ab" | Aggiunta a un file binario. |
| "r+" | Apre un file di testo in lettura/scrittura. |
| "w+" | Crea un file di testo in lettura/scrittura. |
| "a+" | Apre un file di testo in lettura/scrittura. |
| "rb+" o "r+b" | Apre un file binario in lettura/scrittura. |
| "wb+" o "w+b" | Crea un file binario in lettura/scrittura. |
| "ab+" o "a+b" | Apre un file binario in lettura/scrittura. |

Il seguente frammento di codice mostra il metodo corretto per aprire un file:

```
FILE *fp;

if ((fp = fopen("test", "w"))==NULL) {
  printf("Impossibile aprire il file.\n");
  exit(1);
}
```

Questo metodo rileva eventuali errori di apertura di un file, ad esempio l'apertura di un disco protetto in scrittura o completamente pieno. Per indicare un errore viene utilizzato NULL poiché nessun puntatore a file avrà mai tale valore.

Se si usa fopen() per aprire un file in output, un eventuale file preesistente avente tale nome verrà cancellato e sostituito. Se non esiste un file con tale nome, verrà creato. L'apertura in lettura di un file richiede naturalmente che il file esista. Se non esiste verrà restituito un errore. Per aggiungere dati alla fine di un file si deve usare la modalità "a". Se il file non esiste verrà creato.

Quando si accede a un file aperto per operazioni di lettura e scrittura, non si può far seguire a un'operazione di output un'operazione di input senza prima richiamare fflush(), fseek(), fsetpos() o rewind(). Inoltre non si può far seguire a un'operazione di input un'operazione di output senza una chiamata alle funzioni appena menzionate.

Le funzioni correlate sono fclose(), fread(), fwrite(), putc() e getc().

## fprintf()

```
#include <cstdio>
int fprintf(FILE *stream, const char *format, ...);
```

La funzione fprintf() produce in output in *stream* il valore degli elementi contenuti nella lista degli argomenti e nel modo specificato dalla stringa *format*. Il valore restituito è il numero di caratteri effettivamente prodotti in output. Se si verifica un errore, il valore restituito sarà negativo.

Gli argomenti possano andare da 0 a un numero massimo definito dal sistema.

Le operazioni della stringa e dei comandi di controllo del formato sono identiche a quelle della funzione printf(); per una descrizione approfondita, si consulti tale funzione.

Le funzioni correlate sono printf() e fscanf().



## fputc()

```
#include <cstdio>
int fputc(int ch, FILE *stream);
```

La funzione fputc() scrive il carattere *ch* nello stream specificato e alla posizione specificata e poi fa avanzare l'indicatore di posizione del file. Anche se, per motivi storici, *ch* è dichiarato come un int, fputc() lo converte in un unsigned char. Poiché al momento della chiamata tutti i caratteri specificati come argomenti vengono promossi a interi, in genere come argomenti vengono utilizzati dei caratteri. Quando invece viene usato un intero, i byte di ordine superiore verranno semplicemente ignorati.

fputc() restituisce il valore del carattere scritto. Quando si verifica un errore, restituisce EOF. Quando i file sono aperti per operazioni binarie, EOF può essere anche un carattere valido e per sapere se si è effettivamente verificato un errore si deve utilizzare la funzione ferror().

Le funzioni correlate sono fgetc(), fopen(), fprintf(), fread() e fwrite().

## fputs()

```
#include <cstdio>
int fputs(const char *str, FILE *stream);
```

La funzione fputs() scrive sullo stream specificato il contenuto della stringa puntata da *str*. Il terminatore nullo non viene scritto.

In caso di successo la funzione fputs() restituisce un valore non negativo mentre in caso di errore restituisce EOF. Se lo stream è aperto in modalità testo, possono verificarsi alcune traduzioni sui caratteri. Questo significa che potrebbe non esservi una corrispondenza uno-a-uno fra la stringa e il file. Se invece lo stream viene aperto in modalità binaria, non verrà applicata alcuna traduzione e vi sarà una corrispondenza uno-a-uno fra la stringa e il file.

Le funzioni correlate sono fgets(), gets(), puts(), fprintf() e fscanf().

## fread()

```
#include <cstdio>
size_t fread(void *buf, size_t size, size_t count,
             FILE *stream);
```

La funzione fread() restituisce *count* oggetti, ognuno dei quali avrà dimensioni pari a *size* byte; le informazioni vengono lette da *stream* e inseriti nell'array puntato da *buf*. L'indicatore di posizione del file viene fatto avanzare del numero di caratteri letti.

La funzione fread() restituisce il numero di elementi effettivamente letti. Se è stato letto un numero di elementi inferiore a quelli richiesti nella chiamata, vuol dire che si è verificato un errore oppure che è stata raggiunta la fine del file. Per determinare ciò che è accaduto si usa feof() o ferror().

Se lo stream viene aperto per operazioni di testo, possono verificarsi alcune traduzioni dei caratteri come la conversione di sequenze Carriage return / Linee Feed in codici di fine riga.

Le funzioni correlate sono fwrite(), fopen(), fscanf(), fgetc() e getc().

## freopen()

```
#include <cstdio>
FILE *freopen(const char *fname, const char *mode,
              FILE *stream);
```

La funzione freopen() associa uno stream esistente a un altro file. Il nome del nuovo file è puntato da *fname* e la modalità di accesso è puntata da mode; lo stream da riassegnare è puntato da stream. La stringa *mode* usa lo stesso formato di fopen(); una discussione più approfondita si trova nella descrizione di fopen().

Quando viene richiamata, freopen() cerca innanzitutto di chiudere il file cui è attualmente associato lo stream. Se il tentativo di chiusura non ha successo, la funzione freopen() continua comunque a tentare di aprire l'altro file.

In caso di successo la funzione freopen() restituisce un puntatore allo stream mentre in caso contrario restituisce un puntatore nullo.

freopen() viene principalmente utilizzata per redirigere i file definiti dal sistema (stdin, stdout e stderr) in qualche altro file.

Le funzioni correlate sono fopen() e fclose().

## fscanf()

```
#include <cstdio>
int fscanf(FILE *stream, const char *format, ...);
```

La funzione fscanf() funziona esattamente come scanf() tranne per il fatto che legge le informazioni dallo stream specificato invece che da stdin. Per informazioni consultare la descrizione di scanf().



La funzione fscanf() restituisce il numero di argomenti cui è stato assegnato un valore. Questo numero non include i campi saltati. Quando viene restituito il valore EOF significa che prima del primo assegnamento si è verificato un errore.

Le funzioni correlate sono scanf() e fprintf().

## fseek()

```
#include <cstdio>
int fseek(FILE *stream, long offset, int origin);
```

La funzione fseek() imposta l'indicatore di posizione del file associato allo stream secondo il valore di *offset* e di *origin*. Il suo scopo è quello di supportare operazioni di I/O ad accesso diretto. Il valore *offset* è il numero di byte rispetto a *origin* in cui occorre portarsi. Il valore di *origin* deve essere una delle seguenti macro (definite in <cstdio>):

| NOME | SIGNIFICATO |
|---|---|
| SEEK_SET | Posizionamento rispetto all'inizio del file. |
| SEEK_CUR | Posizionamento rispetto alla posizione corrente. |
| SEEK_END | Posizionamento rispetto alla fine del file. |

Quando viene restituito il valore 0, fseek() ha avuto successo. Un valore diverso da 0 indica che si è verificato un errore.

Si può usare fseek() per spostare l'indicatore di posizione in qualsiasi punto del file, anche oltre la fine. Al contrario è un errore tentare di posizionare l'indicatore prima dell'inizio del file.

La funzione fseek() cancella il flag di fine file associato allo stream specificato. Inoltre annulla ogni eventuale ungetc() eseguita sullo stesso stream (vedere la funzione ungetc()).

Le funzioni correlate sono ftell(), rewind(), fopen(), fgetpos() e fsetpos().

## fsetpos()

```
#include <cstdio>
int fsetpos(FILE *stream, const fpos_t *position);
```

La funzione fsetpos() sposta l'indicatore di posizione del file nel punto specificato dall'oggetto puntato da *position*. Questo valore deve essere stato ottenuto

in precedenza con una chiamata a fgetpos(). Dopo aver eseguito fsetpos(), l'indicatore di fine file viene reinizializzato. Inoltre viene annullato ogni precedente chiamata a ungetc().

Se fsetpos() non ha successo, restituisce un valore diverso da 0. In caso di successo restituisce 0.

Le funzioni correlate sono fgetpos(), fseek() e ftell().

## ftell()

```
#include <cstdio>
long ftell(FILE *stream);
```

La funzione ftell() restituisce il valore corrente dell'indicatore di posizione del file nello stream specificato. Nel caso di stream binari, il valore è la posizione (in termini di byte) in cui è posizionato l'indicatore rispetto all'inizio del file. Per stream di testo, questo valore non ha significato se non come argomento di fseek(), poiché le traduzioni apportate al contenuto del file (ad esempio la trasformazione dei codici di fine riga in Carriage Return / Line Feed) può alterare le dimensioni apparenti del file.

La funzione ftell() restituisce in caso d'errore il valore -1. Se lo stream è incapace di eseguire posizionamenti diretti (ad esempio nel caso di un modem) il valore restituito è indefinito.

Le funzioni correlate sono fseek() e fgetpos().

## fwrite()

```
#include <cstdio>
size_t fwrite(const void *buf, size_t size,
              size_t count, FILE *stream);
```

La funzione fwrite() scrive nello stream *num* oggetti ognuno dei quali ha lunghezza pari a *size* byte tratti dall'array di caratteri puntato da *buf*. L'indicatore di posizione del file viene fatto avanzare del numero di caratteri scritti.

La funzione fwrite() restituisce il numero di elementi effettivamente scritti che, se la funzione ha successo, sarà uguale al numero di oggetti richiesti. Quando viene scritto un numero di elementi inferiore rispetto a quelli richiesti, si è verificato un errore. Per gli stream di testo, possono verificarsi varie traduzioni di caratteri ma queste non avranno effetto sul valore restituito.

Le funzioni correlate sono fread(), fscanf(), getc() e fgetc().



## getc()

```
#include <cstdio>
int getc(FILE *stream);
```

La funzione getc() restituisce il carattere successivo tratto dallo stream di input e incrementa l'indicatore di posizione del file. Il carattere viene letto come un unsigned char che viene convertito in un intero. Quando viene raggiunta la fine del file, getc() restituisce EOF. Ma poiché EOF è un valore intero valido, quando si lavora su file binari è necessario controllare che non sia stata raggiunta la fine del file tramite la funzione feof(). getc() restituisce EOF anche quando si verifica un errore. Quando si lavora su file binari, per controllare quando si è verificato un errore si deve usare ferror().

Le funzioni getc() e fgetc() sono identiche e nella maggior parte delle implementazioni la funzione getc() è definita come una semplice macro nel modo seguente:

```
#define getc(fp) fgetc(fp)
```

Questo fa in modo che alla macro getc() sia sostituita la funzione fgetc().

Le funzioni correlate sono fputc(), fgetc(), putc() e fopen().

## getchar()

```
#include <cstdio>
int getchar(void);
```

La funzione getchar() restituisce il prossimo carattere contenuto in stdin. Il carattere viene letto come un unsigned char che viene convertito in un intero.

Se viene raggiunta la fine del file, getchar() restituisce EOF. Ma poiché EOF è un valore intero valido, quando si lavora su file binari si deve controllare se è stata raggiunta la fine del file utilizzando la funzione feof(). Anche quando si verifica un errore getchar() restituisce EOF. Per controllare quando si è verificato un errore, nel caso di file binari si deve usare la funzione ferror().

Spesso la funzione getchar() è implementata come una macro.

Le funzioni correlate sono fputc(), fgetc(), putc() e fopen().

## gets()

```
#include <cstdio>
char *gets(char *str);
```

La funzione gets() legge i caratteri da stdin e li inserisce nell'array di caratteri puntato da *str*. I caratteri vengono letti fino al codice newline successivo o fino alla fine del file. Il carattere di fine riga non entra a far parte della stringa ma viene trasformato nel codice nullo di fine stringa.

In caso di successo, gets() restituisce *str* e in caso di fallimento restituisce un puntatore nullo. Se si verifica un errore, il contenuto dell'array puntato da *str* sarà indeterminato. Poiché viene restituito un puntatore nullo sia in caso di errore che in caso di raggiungimento della fine del file, per determinare cosa è accaduto si deve usare feof() o ferror().

Non vi è alcun modo per limitare il numero di caratteri letti da gets() e dunque è necessario fare in modo che non vengano superati i limiti dell'array puntato da *str*.

Le funzioni correlate sono fputs(), fgetc(), fgets() e puts().

## perror()

```
#include <cstdio>
void perror(const char *str);
```

La funzione perror() mappa il valore della variabile globale errno in una stringa e scrive la stringa in stderr. Se il valore di *str* non è nullo, verrà scritto, seguito da un due punti e da un messaggio definito dall'implementazione.

## printf()

```
#include <cstdio>
int printf(const char *format, ...);
```

La funzione printf() scrive su stdout la lista degli argomenti specificata dalla stringa puntata da formato.

La stringa puntata da formato è costituita da due tipi di elementi. Il primo è composto dai caratteri che verranno stampati sullo schermo. Il secondo tipo contiene gli specificatori di formato che definiscono il modo in cui vengono visualizzati gli argomenti. Uno specificatore di formato inizia con il segno percentuale ed è seguito dal codice di formato. Vi deve essere esattamente lo stesso numero di argomenti e lo stesso numero di specificatori di formato e gli specificatori di formato e gli argomenti devono essere ordinati allo stesso modo. Ad esempio, la seguente istruzione visualizza la frase "Hi c 10 there!".

```
printf("Hi %c %d %s", 'c', 10, "there!");
```



Se gli argomenti sono insufficienti rispetto agli specificatori di formato, l'output sarà indefinito. Se vi sono più argomenti che specificatori, gli argomenti in eccesso vengono eliminati. Gli specificatori di formato sono elencati nella Tabella 25.2.

La funzione printf() restituisce il numero di caratteri effettivamente stampati. Quando viene restituito un numero negativo, significa che si è verificato un errore.

Ai codici di formato possono essere associati dei modificatori che specificano la larghezza del campo, la precisione e l'allineamento. Un intero posizionato fra il segno % e il codice di formattazione costituisce uno *specificatore di larghezza minima del campo*. Questo ha l'effetto di inserire degli spazi o il numero "0" per garantire la larghezza minima del campo. Se la stringa o il numero è maggiore del minimo specificato, verrà stampato completamente. Normalmente lo spazio vuoto sarà occupato dal carattere spazio. Se si vuole utilizzare il numero "0", si deve posizionare uno "0" prima dello specificatore di larghezza del campo. Ad esem-

**Tabella 25.2** Gli specificatori di formato di printf().

| CODICE | FORMATO |
|---|---|
| %c | Carattere |
| %d | Intero decimale con segno |
| %i | Intero decimale con segno |
| %e | Notazione scientifica (e minuscola) |
| %E | Notazione scientifica (E maiuscola) |
| %f | Numero in virgola mobile |
| %g | Usa il formato più breve fra %e (minuscola) o %f |
| %G | Usa il formato più breve fra %E (maiuscola) o %f |
| %o | Ottale senza segno |
| %s | Stringa di caratteri |
| %u | Intero decimale senza segno |
| %x | Esadecimale senza segno (lettere minuscole) |
| %X | Esadecimale senza segno (lettere maiuscole) |
| %p | Visualizza un puntatore |
| %n | L'argomento associato è un puntatore a un intero nel quale viene inserito il numero di caratteri scritti finora |
| %% | Stampa il segno % |

pio il codice %05d farà precedere ai numeri di meno di 5 cifre una serie di caratteri "0" in modo che la larghezza totale del campo sia pari a 5.

Il significato del modificatore di precisione dipende dal codice di formattazione che viene modificato. Per aggiungere un modificatore di precisione si deve posizionare un punto decimale seguito dal valore della precisione, dopo lo specificatore di larghezza del campo. Per i formati e, E e F il modificatore di precisione determina il numero di cifre decimali visualizzate. Ad esempio il codice %10.4f visualizza un numero in 10 spazi con 4 cifre decimali. Quanto il modificatore di precisione viene applicato al codice di formattazione g o G, determina il numero massimo di cifre significative visualizzate. Quando viene applicato agli interi, il modificatore di precisione specifica il numero minimo di cifre da visualizzare. Se necessario verranno aggiunti caratteri "0" iniziali.

Normalmente l'output è allineato a destra: se la larghezza del campo è maggiore rispetto ai dati stampati, i dati vengono accostati al bordo destro del campo. Per richiedere l'allineamento a sinistra dei dati basta inserire il segno - appena dopo il segno %. Ad esempio, il codice %-10.2f allinea a sinistra un numero in virgola mobile con due cifre decimali in un campo che si estende per 10 caratteri.

Vi sono due modificatori di formato che consentono a printf() di visualizzare interi short e long. Questi modificatori possono essere applicati agli specificatori d, i, o, u e x. Il modificatore l (la lettera elle) dice a printf() che deve essere stampato un tipo long. Ad esempio, %ld chiede la visualizzazione di un intero long. Il modificatore h chiede a printf() di visualizzare un intero short. Pertanto il codice %hu indica dati di tipo short unsigned int.

Se si usa un moderno compilatore che supporta i caratteri estesi, introdotti nel 1995, allora si può usare il modificatore l con lo specificatore c per indicare l'uso di caratteri estesi di tipo wchar_t. Si può utilizzare il modificatore l anche con il comando di formattazione s per indicare una stringa di caratteri estesi.

Il modificatore L può precedere i comandi per numeri in virgola mobile e, f e g e indica che deve essere stampato un long double.

Il comando %n chiede di inserire in una variabile intera (il cui puntatore è specificato nell'elenco degli argomenti) il numero di caratteri che sono stati scritti fino al momento in cui è stato incontrato tale codice. Ad esempio, questo frammento di codice visualizza il numero 18 dopo la frase "Questa è una prova":

```
int i;

printf("Questa è una prova%n", &i);
printf("%d", i);
```

Se utilizzato con alcuni codici di formattazione di printf(), il carattere # ha un significato speciale. Quando il carattere # precede un codice g, G, f, e o E, significa che è richiesto l'inserimento del punto decimale, anche se il numero non contiene cifre decimali. Prima dei codici di formattazione x e X, il codice # provoca la visualizzazione del numero esadecimale con il prefisso 0x. Prima del codice di formattazione o, il modificatore # provoca la visualizzazione di un valore ottale con il prefisso 0. Il modificatore # può essere applicato solo a questi specificatori di formato. Gli specificatori di larghezza minima del campo e di precisione possono essere forniti anche come argomenti di printf() oltre che come costanti. A tale scopo si deve utilizzare come segnaposto il carattere *. Quando viene esaminata la stringa di formattazione, la funzione printf() assegna ad ogni asterisco l'argomento nell'ordine in cui si presenta.

Le funzioni correlate sono scanf() e fprintf().



## putc()

```
#include <cstdio>
int putc(int ch, FILE *stream);
```

La funzione putc() scrive sullo stream il carattere contenuto nel byte meno significativo di ch. Poiché al momento della chiamata gli argomenti a caratteri vengono promossi in interi, si può tranquillamente utilizzare caratteri come argomenti di putc().

La funzione putc() restituisce i caratteri scritti con successo oppure EOF in caso di errore. Se lo stream di output è stato aperto in modalità binaria, EOF è un valore valido per ch. Questo significa che si deve usare ferror() per determinare se si è verificato un errore.

Le funzioni correlate sono fgetc(), fputc(), getchar() e putchar().

## putchar()

```
#include <cstdio>
int putchar(int ch);
```

La funzione putchar() scrive il carattere contenuto nel byte meno significativo di *ch* su stdout. In pratica la funzione è equivalente a putc(ch, stdout). Poiché al momento della chiamata gli argomenti a caratteri vengono promossi in interi, si può tranquillamente utilizzare caratteri come argomenti di putchar().

La funzione putchar() restituisce i caratteri scritti con successo oppure EOF in caso di errore. Se lo stream di output è stato aperto in modalità binaria, EOF è un valore valido per *ch*. Questo significa che si deve usare ferror() per determinare se si è verificato un errore.

La funzione correlata è putc().

## puts()

```
#include <cstdio>
int puts(const char *str);
```

La funzione puts() scrive la stringa puntata da *str* sul dispositivo di output standard. Il terminatore nullo viene tradotto nel codice di fine riga.

La funzione puts() restituisce un valore non negativo oppure EOF in caso di insuccesso.

Le funzioni correlate sono putc(), gets() e printf().

## remove()

```
#include <cstdio>
int remove(const char *fname);
```

La funzione remove() cancella il file specificato da *fname*. In caso di successo restituisce 0 mentre in caso di errore restituisce un valore diverso da zero.

Una funzione correlata è rename().

## rename()

```
#include <cstdio>
int rename(const char *oldfname, const char *newfname);
```

La funzione rename() cambia il nome del file specificato da *oldfname* assegnandogli il nome *newfname*. Il nuovo nome non deve essere già presente nella stessa directory.

In caso di successo la funzione restituisce il valore 0 mentre in caso di errore restituisce il valore 0.

Una funzione correlata è remove().

## rewind()

```
#include <cstdio>
void rewind(FILE *stream);
```

La funzione rewind() porta l'indicatore di posizione del file all'inizio dello stream specificato. Inoltre cancella i flag di fine file e di errore associati a stream. Non restituisce alcun valore.

Una funzione correlata è fseek().



## scanf()

```
#include <cstdio>
int scanf(const char *format, ...);
```

La funzione scanf() è una routine di input di carattere generale che legge informazioni dallo stream stdin e le memorizza nelle variabili puntate dalla lista degli argomenti. Può leggere tutti i tipi di dati standard e convertirli automaticamente nel formato interno corretto.

La stringa controllo puntata da *format* è costituita da tre classificazioni di caratteri:

specificatori di formato;
caratteri di spaziatura;
caratteri non di spaziatura.

Gli specificatori di formato di input iniziano con il segno % e dicono a scanf() quale tipo di dati deve essere letto. Gli specificatori di formato sono elencati nella Tabella 25.3.

**Tabella 25.3** Gli specificatori di formato di scanf().

| CODICE | SIGNIFICATO |
|---|---|
| %c | Legge un singolo carattere. |
| %d | Legge un intero decimale. |
| %i | Legge un intero. |
| %e | Legge un numero in virgola mobile. |
| %f | Legge un numero in virgola mobile. |
| %g | Legge un numero in virgola mobile. |
| %o | Legge un numero ottale. |
| %s | Legge una stringa. |
| %x | Legge un numero esadecimale. |
| %p | Legge un puntatore. |
| %n | Riceve un valore intero uguale al numero di caratteri scritti finora. |
| %u | Legge un intero senza segno. |
| %[ ] | Scansione di un insieme di caratteri. |
| %% | Legge un segno percentuale. |

Ad esempio %s legge una stringa e %d legge un intero. La stringa di formattazione viene letta da sinistra a destra e gli specificatori vengono associati uno dopo l'altro agli elementi che costituiscono la lista degli argomenti.

Per leggere un intero si deve specificare il codice l (la lettera elle) davanti allo specificatore di formato. Per leggere un intero short si deve porre una h davanti allo specificatore di formato. Questi modificatori possono essere utilizzati nei codici di formattazione d, i, o, u e x.

Normalmente gli specificatori f, e e g chiedono a scanf() di assegnare i dati a un float. Se si pone la l davanti a uno di questi specificatori, scanf() assegna i dati a un double. La lettera L dice a scanf() che la variabile che riceve i dati è un long double.

Se si usa un moderno compilatore che supporta i caratteri estesi, introdotti nel 1995, allora si può usare il modificatore l con lo specificatore c per indicare l'uso di caratteri estesi di tipo wchar_t. Si può utilizzare il modificatore l anche con il comando di formattazione s per indicare una stringa di caratteri estesi. La lettera l può essere utilizzata anche per modificare uno scanset in modo da richiedere l'uso di caratteri estesi.

Un carattere di spaziatura nella stringa di formattazione fa in modo che scanf() salti uno o più spazi nella stringa di input. Un carattere di spaziatura può essere un vero e proprio carattere di spazio, un carattere di tabulazione o un carattere di fine riga. In pratica un carattere vuoto nella screen di controllo fa in modo che scanf() legga senza memorizzare un qualsiasi numero (anche 0) di caratteri di spaziatura fino al successivo carattere non di spaziatura.

Un carattere non di spaziatura nello stream di formattazione fa in modo che scanf() legga e scarichi il carattere corrispondente. Ad esempio il codice %d,%d fa in modo che scanf() legga un intero, legga ed elimini una virgola e infine legga un altro intero. Se il carattere specificato non viene trovato, scanf() ha termine.

Tutte le variabili utilizzate per ricevere valori tramite scanf() devono essere passate per indirizzo. Questo significa che tutti gli argomenti devono essere puntatori.

Gli elementi di input devono essere separati da uno spazio, un codice di tabulazione o dal carattere di fine riga. I segni di punteggiatura come le virgole i punti e virgola e così via non contano come separatori. Questo significa che:

```
scanf("%d%d", &r, &c);
```

accetterà un input 10 20 ma non un input 10,20.

Un asterisco posizionato dopo il segno % e prima del codice di formattazione consente di leggere i dati del tipo specificato ma sopprime l'assegnamento. Pertanto il comando:

```
scanf("%d%*c%d", &x, &y);
```



dato l'input 10/20 inserisce il valore 10 in x, ignora il segno di divisione e assegna il valore 20 a y.

I comandi di formattazione possono specificare un modificatore che indica la lunghezza massima di un campo. Si tratta di un numero intero posizionato fra il segno % e il codice di formattazione per limitare il numero di caratteri letti per ciascun campo. Ad esempio, se nella variabile address non si vogliono leggere più di 20 caratteri, si deve utilizzare la seguente forma:

```
scanf("%20s", address);
```

Se lo stream di input è maggiore di 20 caratteri, una successiva chiamata di input inizierà da dove questa chiamata è terminata. L'input di un campo può terminare prima del raggiungimento della lunghezza massima del campo quando viene incontrato uno spazio vuoto. In questo caso scanf() si porta sul campo successivo.

Anche se gli spazi, i codici di tabulazione e i codici di fine riga sono utilizzati come separatori fra i campi, quando si legge un singolo carattere questi codici verranno letti come qualsiasi altro carattere. Ad esempio, se lo stream di input contiene x y,

```
scanf("%c%c%c", &a, &b, &c);
```

restituisce il carattere x in a, uno spazio in b e il carattere y in c.

Attenzione: ogni altro carattere della stringa di controllo (inclusi gli spazi, i caratteri di tabulazione e i codici di fine riga) vengono utilizzati per individuare ed eliminare i caratteri dallo stream di input. Dunque verranno eliminati tutti i caratteri corrispondenti. Ad esempio, dato lo stream di input 10t20,

```
scanf("%dt%d", &x, &y);
```

inserisce 10 in x e 20 in y. La t viene eliminata per la presenza della lettera t nella stringa di controllo.

Un'altra funzionalità di scanf() è lo *scanset*. Uno scanset definisce l'insieme di caratteri che verranno letti da scanf() e assegnati al corrispondente array di caratteri. Uno scanset viene definito inserendo i caratteri da considerare fra parentesi quadre. La parentesi quadra aperta deve essere preceduta dal segno %. Ad esempio, il seguente scanset chiede a scanf() di leggere i soli caratteri A, B e C:

```
%[ABC]
```

Quando viene utilizzato uno scanset, scanf() continua la leggere i caratteri e ad inserirli nel corrispondente array di caratteri finché non viene incontrato un

carattere non presente nello scanset. La variabile corrispondente deve essere un puntatore a un array di caratteri. All'uscita di scanf(), l'array conterrà una stringa chiusa dal carattere nullo e contenente i caratteri letti.

Si può anche specificare uno scanset inverso quando il primo carattere dello scanset è ^. Questo significa che scanf() accetterà tutti i caratteri non definiti dallo scanset.

Per specificare un intervallo basta utilizzare un trattino. Ad esempio, la seguente istruzione chiede a scanf() di accettare tutti i caratteri fra A e Z.

```
%[A-Z]
```

È importante ricordare che uno scanset distingue fra lettere maiuscole e minuscole. Pertanto se si vogliono leggere lettere maiuscole e minuscole occorre specificarle esplicitamente.

La funzione scanf() restituisce un numero equivalente al numero di campi cui è stato assegnato un valore. Questo numero non comprende i campi letti ma non assegnati a causa della presenza del modificatore *. Se prima della lettura del primo campo si è verificato un errore, la funzione restituisce EOF.

Le funzioni correlate sono printf() e fscanf().

## setbuf()

```
#include <cstdio>
void setbuf(FILE *stream, char *buf);
```

La funzione setbuf() consente di specificare il buffer per lo stream oppure, quando *buf* è nullo, consente di disattivare la bufferizzazione. Se deve essere specificato un buffer definito dal programmatore, questo deve essere lungo *BUFSIZ* caratteri. *BUFSIZ* è definito in <cstdio>.

La funzione setbuf() non restituisce alcun valore.

Le funzioni correlate sono fopen(), fclose() e setvbuf().

## setvbuf()

```
#include <cstdio>
int setvbuf(FILE *stream, char *buf, int mode, size_t size);
```

La funzione setvbuf() consente al programmatore di specificare il buffer, le sue dimensioni e la sua modalità di funzionamento. Per bufferizzare le operazioni di I/O verrà utilizzato l'array di caratteri puntato da *buf*. Le dimensioni del buffer



sono definite da *size* mentre *mode* determina la modalità di bufferizzazione. Se *buf* è nullo, setvbuf() alloca un proprio buffer.

Per *mode* si possono usare i valori _IOFBF, _IONBF e _IOLBF. Tali valori sono definiti in <cstdio>. Quando *mode* è uguale a _IOFBF, viene eseguita una bufferizzazione completa. Se *mode* è _IOLBF, lo stream verrà bufferizzato riga per riga, ovvero il buffer verrà svuotato ogni volta che nello stream di output viene scritto il codice di fine riga; per gli stream di input, una richiesta di input leggerà tutti i caratteri fino al codice di fine riga. In entrambi i casi il buffer sarà comunque svuotato una volta raggiunta la massima capacità. Quando *mode* è _IONBF, non viene eseguita alcuna bufferizzazione.

Il valore di *size* deve essere maggiore di 0.

In caso di successo la funzione setvbuf() restituisce 0 mentre in caso di errore restituisce un valore diverso da zero.

Una funzione correlata è setbuf().

## sprintf()

```
#include <cstdio>
int sprintf(char *buf, const char *format, ...);
```

La funzione sprintf() è identica a printf() tranne per il fatto che l'output viene inserito nell'array puntato da *buf* invece di essere scritto sullo schermo. Per informazioni consultare printf().

Il valore restituito è uguale al numero di caratteri effettivamente inseriti nell'array.

Le funzioni correlate sono printf() e fsprintf().

## sscanf()

```
#include <cstdio>
int sscanf(const char *buf, const char *format, ...);
```

La funzione sscanf() è identica a scanf() tranne per il fatto che i dati vengono letti dall'array puntato da *buf* invece che da stdin. Per informazioni consultare scanf().

Il valore restituito è uguale al numero delle variabili cui è stato assegnato un valore. Questo numero non include i campi saltati per l'uso del modificatore *. Il valore 0 significa che non è stato assegnato il valore ad alcun campo mentre EOF indica che prima del primo assegnamento si è verificato un errore.

Le funzioni correlate sono scanf() e fscanf().

## tmpfile()

```
#include <cstdio>
FILE *tmpfile(void);
```

La funzione tmpfile() apre in aggiornamento un file temporaneo e restituisce un puntatore allo stream. La funzione utilizza automaticamente un nome di file univoco per evitare conflitti con i file preesistenti.

In caso di fallimento la funzione tmpfile() restituisce un puntatore nullo, altrimenti restituisce un puntatore allo stream.

Il file temporaneo creato da tmpfile() viene rimosso automaticamente alla chiusura del file o al termine del programma.

Una funzione correlata è tmpnam().

## tmpnam()

```
#include <cstdio>
char *tmpnam(char *name);
```

La funzione tmpnam() genera un nome di file univoco e lo memorizza nell'array puntato da *name*. Lo scopo principale di tmpnam() è quello di generare un nome di file temporaneo diverso rispetto ad ogni altro file contenuto nella directory corrente.

La funzione può essere richiamata fino a TMP_MAX volte. TMP_MAX è definito in <cstdio> ed equivale ad almeno 25. Ogni volta che viene richiamata, tmpnam() genera un nuovo nome di file temporaneo.

In caso di successo viene restituito un puntatore a *name*, altrimenti viene restituito un puntatore nullo.

Una funzione correlata è tmpfile().

## ungetc()

```
#include <cstdio>
int ungetc(int ch, FILE *stream);
```

La funzione ungetc() reinserisce nello stream il carattere specificato dal byte di ordine inferiore di *ch*. Questo carattere verrà ottenuto dalla successiva operazione di lettura dallo stream. Una chiamata a fflush() o fseek() annulla un'operazione ungetc() ed elimina il carattere.

In genere è consentito il reinserimento di un solo carattere mentre alcune implementazioni consentono di reinserire nella stringa un maggior numero caratteri.



Non si può eseguire ungetc() su EOF.

Una chiamata a ungetc() cancella il flag di fine file associato allo stream specificato. Il valore dell'indicatore di posizione del file per uno stream di testo è indefinito finché non vengono letti tutti i caratteri reinseriti e allora sarà uguale al valore che aveva prima della chiamata a ungetc(). Per gli stream binari, ogni chiamata a ungetc() decrementata l'indicatore di posizione del file.

La funzione restituisce il carattere *ch* in caso di successo e EOF in caso di errore.

Una funzione correlata è getc().

## vprintf(), vfprintf() e vsprintf()

```
#include <cstdarg>
#include <cstdio>
int vprintf(char *format, va_list arg_ptr);
int vfprintf(FILE *stream, const char *format,
             va_list arg_ptr);
int vsprintf(char *buf, const char *format,
             va_list arg_ptr);
```

Le funzioni vprintf(), vfprintf() e vsprintf() hanno funzionalità equivalenti, rispettivamente, a printf(), fprintf() e sprintf() tranne per il fatto che la lista degli argomenti è sostituita da un puntatore. Questo puntatore deve essere tipo va_list, definito nell'header <cstarg> (o nel file header C <stdarg.h>).

Le funzioni correlate sono va_arg(), va_start() e va_end().

Capitolo 26

# Le funzioni per stringhe e caratteri

La libreria di funzioni standard contiene una ricca varietà di funzioni per la manipolazione delle stringhe e dei caratteri. Le funzioni per stringhe operano su un array di caratteri chiuso dal carattere nullo e richiedono l'impiego dell'header <cstring>. Le funzioni per caratteri usano l'header <cctype>. I programmi C devono usare i file header <string.h> e <ctype.h>.

Poiché il linguaggio C/C++ non prevede verifiche sul raggiungimento dei limiti per le operazioni sugli array, sarà responsabilità del programmatore evitare di fuoriuscire dai limiti stabiliti per gli array. Se si dimentica questa precauzione, si provocherà molto probabilmente il blocco del programma.

In C/C++ un *carattere stampabile* è un carattere che può essere visualizzato su un terminale. In genere si tratta dei caratteri compresi fra lo spazio (0x20) e il carattere tilde (0xFE). I *caratteri di controllo* hanno un valore compreso fra 0 e 0x1F più il codice DEL (0x7F).

Per motivi storici, i parametri delle funzioni a caratteri sono interi ma viene utilizzato solo il byte di ordine inferiore; le funzioni per caratteri convertono automaticamente i loro argomenti in unsigned char. Tuttavia è anche possibile richiamare queste funzioni con argomenti char in quanto la chiamata a funzione promuove automaticamente i caratteri in interi.

L'header <cstring> definisce il tipo size_t che è in pratica la stessa cosa di unsigned.

Questo capitolo descrive le sole funzioni che operano su caratteri di tipo char. Si tratta delle funzioni originariamente definite dal C e dal C++ standard e sono anche quelle più ampiamente utilizzate e supportate. Le funzioni per caratteri estesi che operano su caratteri di tipo wchar_t verranno discusse nel Capitolo 31.

## isalnum()

```
#include <cctype>
int isalnum(int ch);
```

La funzione isalnum() restituisce un numero diverso da 0 quando il suo argomento è una lettera alfabetica o una cifra. Se il carattere non è alfanumerico, restituisce 0.

Le funzioni correlate sono isalpha(), iscntrl(), isdigit(), isgraph(), isprint(), ispunct() e isspace().

## isalpha()

```
#include <cctype>
int isalpha(int ch);
```

La funzione isalpha() restituisce un valore diverso da zero se ch è una lettera dell'alfabeto, altrimenti restituisce il valore 0. Cosa si debba intendere per lettera dell'alfabeto dipende da lingua a lingua ma in genere si tratta delle lettere maiuscole e minuscole comprese fra A e Z.

Le funzioni correlate sono isalnum(), iscntrl(), isdigit(), isgraph(), isprint(), ispunct() e isspace().

## iscntrl()

```
#include <cctype>
int iscntrl(int ch);
```

La funzione iscntrl() restituisce un valore diverso da 0 se *ch* è compreso fra 0 e 0x1F o se è uguale a 0x7F (DEL); altrimenti restituisce il valore 0.

Le funzioni correlate sono isalnum(), isalpha(), isdigit(), isgraph(), isprint(), ispunct() e isspace().

## isdigit()

```
#include <cctype>
int isdigit(int ch);
```

La funzione isdigit() restituisce un valore diverso da zero se *ch* è una cifra, ovvero se è un carattere compreso fra 0 e 9. Altrimenti restituisce il valore 0.

Le funzioni correlate sono isalnum(), isalpha(), iscntrl(), isgraph(), isprint(), ispunct() e isspace().

## isgraph()

```
#include <cctype>
int isgraph(int ch);
```

La funzione isgraph() restituisce un valore diverso da zero se *ch* è un carattere stampabile diverso da uno spazio, altrimenti restituisce 0. In generale i caratteri stampabili sono quelli compresi fra 0x21 e 0x7E.

Le funzioni correlate sono isalnum(), isalpha(), iscntrl(), isdigit(), isprint(), ispunct() e isspace().

## islower()

```
#include <cctype>
int islower(int ch);
```

La funzione islower() restituisce un valore diverso da zero se *ch* è una lettera , altrimenti restituisce 0.

Una funzione correlata è isupper().

## isprint()

```
#include <cctype>
int isprint(int ch);
```

La funzione isprint() restituisce un valore diverso da zero se *ch* è un carattere stampabile, incluso lo spazio, altrimenti restituisce il valore 0. I caratteri stampabili si trovano spesso nell'intervallo fra 0x20 e 0x7E.

Le funzioni correlate sono isalnum(), isalpha(), iscntrl(), isdigit(), isgraph(), ispunct() e isspace().

## ispunct()

```
#include <cctype>
int ispunct(int ch);
```

La funzione ispunct() restituisce un valore diverso da 0 se *ch* è un segno di punteggiatura, altrimenti restituisce 0. Con "segno di punteggiatura" si intendono tutti i caratteri stampabili che non sono né alfanumerici né spazi.

Le funzioni correlate sono isalnum(), isalpha(), iscntrl(), isdigit(), isgraph() e isspace().

## isspace()

```
#include <cctype>
int isspace(int ch);
```

La funzione isspace() restituisce un valore diverso da zero se *ch* è uno spazio, una tabulazione orizzontale, una tabulazione verticale, un codice di avanzamento pagina, un codice Carriage Return o un carattere di fine riga; altrimenti restituisce 0.

Le funzioni correlate sono isalnum(), isalpha(), iscntrl(), isdigit(), isgraph() e ispunct().

## isupper()

```
#include <cctype>
int isupper(int ch);
```

La funzione isupper() restituisce un valore diverso da 0 se *ch* è una lettera maiuscola, altrimenti restituisce 0.

Una funzione correlata è islower().

## isxdigit()

```
#include <cctype>
int isxdigit(int ch);
```

La funzione isxdigit() restituisce per valore diverso da zero se *ch* è una cifra esadecimale, altrimenti restituisce 0. Una cifra esadecimale si trova negli intervalli A-F, a-f e 0-9.

Le funzioni correlate sono isalnum(), isalpha(), iscntrl(), isdigit(), isgraph(), ispunct() e isspace().

## memchr()

```
#include <cstring>
void *memchr(const void *buffer, int ch, size_t count);
```



La funzione memchr() ricerca nell'array puntato da buffer la prima occorrenza di *ch* nei primi *count* caratteri.

La funzione memchr() restituisce un puntatore alla prima occorrenza di *ch* in buffer; se *ch* non viene trovato restituisce un puntatore nullo.

Le funzioni correlate sono memcpy() e isspace().

## memcmp()

```
#include <cstring>
int memcmp(const void *buf1, const void *buf2, size_t count);
```

La funzione memcmp() confronta i primi count caratteri degli array puntati da *buf1* e *buf2*.

La funzione memcmp() restituisce un intero da interpretare nel seguente modo:

| VALORE | SIGNIFICATO |
|---|---|
| Minore di 0 | *buf1* è minore di *buf2* |
| 0 (zero) | *buf1* è uguale a *buf2* |
| Maggiore di 0 | *buf1* è maggiore di *buf2* |

Le funzioni correlate sono memchr(), memcpy() e strcmp().

## memcpy()

```
#include <cstring>
void *memcpy(void *to, const void *from, size_t count);
```

La funzione memcpy() copia *count* caratteri dall'array puntato da *from* all'array puntato da *to*. Se gli array si sovrappongono, il comportamento di memcpy() è indefinito.

La funzione memcpy() restituisce un puntatore a *to*.

Una funzione correlata è memmove().

## memmove()

```
#include <cstring>
void *memmove(void *to, const void *from, size_t count);
```

La funzione memmove() copia *count* caratteri dall'array puntato da *from* all'array puntato da *to*. Se gli array si sovrappongono, la copia verrà eseguita correttamente, inserendo il contenuto corretto in *to* ma modificando anche *from*.

La funzione memmove() restituisce un puntatore a *to*.

Una funzione correlata è memcpy().

## memset()

```
#include <cstring>
void *memset(void *buf, int ch, size_t count);
```

La funzione memset() copia il byte di ordine inferiore di *ch* nei primi *count* caratteri dell'array puntato da *buf* e restituisce *buf*.

L'uso più comune di memset() consiste nell'inizializzazione di una regione di memoria con un valore ben noto.

Le funzioni correlate sono memcmp(), memcpy() e memmove().

## strcat()

```
#include <cstring>
char *strcat(char *str1, const char *str2);
```

La funzione strcat() concatena una copia di *str2* a *str1* e chiude *str1* con un carattere nullo. Il terminatore nullo che originariamente chiudeva *str1* viene sovrascritto dal primo carattere di *str2*. La stringa *str2* non viene modificata dall'operazione. Se gli array si sovrappongono, il comportamento di strcat() è indefinito.

La funzione strcat() restituisce *str1*.

Si ricordi che non vengono eseguite verifiche dei limiti e dunque è responsabilità del programmatore assicurarsi che *str1* abbia dimensioni sufficienti per contenere sia il suo originale contenuto che il contenuto di *str2*.

Le funzioni correlate sono strchr(), strcmp() e strcpy().

## strchr()

```
#include <cstring>
char *strchr(const char *str, int ch);
```

La funzione strchr() restituisce un puntatore alla prima occorrenza del byte di ordine inferiore di *ch* nella stringa puntata da *str*.



Se non viene trovata una corrispondenza la funzione restituisce un puntatore nullo.

Le funzioni correlate sono strpbrk(), strspn(), strstr() e strtok().

## strcmp()

```
#include <cstring>
int strcmp(const char *str1, const char *str2);
```

La funzione strcmp() confronta in modo lessicale due stringhe e restituisce un intero sulla base del risultato, come indicato nella seguente tabella.

| VALORE | SIGNIFICATO |
|---|---|
| Minore di 0 | *str1* è minore di *str2* |
| 0 (zero) | *str1* è uguale a *str2* |
| Maggiore di 0 | *str1* è maggiore di *str2* |

Le funzioni correlate sono strchr(), strcpy() e strcmp().

## strcoll()

```
#include <cstring>
int strcoll(const char *str1, const char *str2);
```

La funzione strcoll() confronta la stringa puntata da *str1* con quella puntata da *str2*. Il confronto viene eseguito secondo quanto stabilito nella funzione setlocale() (per informazioni consultare la parte relativa a tale funzione).

La funzione strcoll() restituisce un intero interpretabile nel seguente modo:

| VALORE | SIGNIFICATO |
|---|---|
| Minore di 0 | *str1* è minore di *str2* |
| 0 (zero) | *str1* è uguale a *str2* |
| Maggiore di 0 | *str1* è maggiore di *str2* |

Le funzioni correlate sono memcmp() e strcmp().

## strcpy()

```
#include <cstring>
char *strcpy(char *str1, const char *str2);
```

La funzione strcpy() viene utilizzata per copiare in *str1* il contenuto di *str2*. *str2* deve essere un puntatore a una stringa chiusa dal carattere nullo. La funzione strcpy() restituisce un puntatore a *str1*.

Se *str1* e *str2* si sovrappongono, il comportamento di strcpy() è indefinito.

Le funzioni correlate sono memcpy(), strchr(), strcmp() e strncmp().

## strcspn()

```
#include <cstring>
size_t strcspn(const char *str1, const char *str2);
```

La funzione strcspn() restituisce la lunghezza della sottostringa iniziale della stringa puntata da str1 che è composta dai soli caratteri non contenuti nella stringa puntata da *str2*. In altre parole, strcspn() restituisce l'indice del primo carattere della stringa puntata da *str1* che corrisponde a uno qualsiasi dei caratteri contenuti nella stringa puntata da *str2*.

Le funzioni correlate sono strrchr(), strpbrk(), strstr() e strtok().

## strerror()

```
#include <cstring>
char *strerror(int errnum);
```

La funzione strerror() restituisce un puntatore a una stringa definita dall'implementazione associata al valore di *errnum*. In nessun caso si deve modificare la stringa.

## strlen()

```
#include <cstring>
size_t strlen(const char *str);
```

La funzione strlen() restituisce la lunghezza della stringa chiusa dal carattere nullo puntata da *str*. Il terminatore nullo non viene contato.

Le funzioni correlate sono memcpy(), strchr(), strcmp() e strncmp().



## strncat()

```
#include <cstring>
char *strncat(char *str1, const char *str2, size_t count);
```

La funzione strncat() concatena non più di *count* caratteri della stringa puntata da *str2* alla stringa puntata da *str1* e chiude *str1* con il carattere nullo. Il terminatore nullo che originariamente chiudeva *str1* viene sovrascritto dal primo carattere di *str2*. La stringa *str2* non viene modificata dall'operazione. Se le stringhe si sovrappongono, il comportamento è indefinito.

La funzione strncat() restituisce *str1*.

Si ricordi che non vengono eseguite verifiche sui limiti e dunque è responsabilità del programmatore assicurarsi che le dimensioni di *str1* siano sufficienti per contenere il suo originale contenuto e anche *str2*.

Le funzioni correlate sono strcat(), strnchr(), strncmp() e strncpy().

## strncmp()

```
#include <cstring>
int strncmp(const char *str1, const char *str2, size_t count);
```

La funzione strncmp() confronta in modo lessicale non più di *count* caratteri di due stringhe e restituisce un intero sulla base del risultato, come indicato nella seguente tabella.

| VALORE | SIGNIFICATO |
|---|---|
| Minore di 0 | *str1* è minore di *str2* |
| 0 (zero) | *str1* è uguale a *str2* |
| Maggiore di 0 | *str1* è maggiore di *str2* |

Se nelle stringhe vi sono meno di *count* caratteri, il confronto termina quando viene incontrato il primo carattere nullo.

Le funzioni correlate sono strcmp(), strnchr() e strncpy().

## strncpy()

```
#include <cstring>
char *strncpy(char *str1, const char *str2, size_t count);
```

La funzione strncpy() copia fino a *count* caratteri dalla stringa puntata da *str2* alla stringa puntata da *str1*. *str2* deve essere un puntatore a una stringa chiusa dal carattere nullo.

Se *str1* e *str2* si sovrappongono, il comportamento di strncpy() è indefinito.

Se la stringa puntata da *str2* contiene meno di *count* caratteri, alla fine della stringa *str1* verranno aggiunti dei caratteri nulli fino a copiare *count* caratteri

Al contrario, se la stringa puntata da *str2* è più lunga di *count* caratteri, la stringa risultante puntata da *str1* sarà chiusa dal carattere nullo.

La funzione strncpy() restituisce un puntatore a *str1*.

Le funzioni correlate sono memcpy(), strchr(), strncat() e strncmp().

## strpbrk()

```
#include <cstring>
char *strpbrk(const char *str1, const char *str2);
```

La funzione strpbrk() restituisce un puntatore al primo carattere della stringa puntata da *str1* che corrisponde a un qualsiasi carattere della stringa puntata da *str2*. I terminatori nulli non vengono inclusi. Se non vi è alcuna corrispondenza la funzione restituisce un puntatore nullo.

Le funzioni correlate sono strspn(), strrchr(), strstr() e strtok().

## strrchr()

```
#include <cstring>
char *strrchr(const char *str, int ch);
```

La funzione strrchr() restituisce un puntatore all'ultima occorrenza del byte di ordine inferiore di *ch* nella stringa puntata da *str*. Se non viene trovata alcuna corrispondenza, strrchr() restituisce un puntatore nullo.

Le funzioni correlate sono strpbrk(), strspn(), strstr() e strtok().

## strspn()

```
#include <cstring>
size_t strspn(const char *str1, const char *str2);
```

La funzione strspn() restituisce la lunghezza della sottostringa iniziale della stringa puntata da *str1* che è composta dai soli caratteri contenuti nella stringa puntata da *str2*. In altre parole, strspn() restituisce l'indice del primo carattere della stringa puntata *str1* che non corrisponde ad alcuno dei caratteri contenuti nella stringa puntata da *str2*.



Le funzioni correlate sono strpbrk(), strrchr(), strstr() e strtok().

## strstr()

```
#include <cstring>
char *strstr(const char *str1, const char *str2);
```

La funzione strstr() restituisce un puntatore alla prima occorrenza della stringa puntata da *str2* all'interno della stringa puntata da *str1*. Se non viene trovata alcuna corrispondenza, la funzione strstr() restituisce un puntatore nullo.

Le funzioni correlate sono strchr(), strcspn(), strpbrk(), strspn(), strtok() e strrchr().

## strtok()

```
#include <cstring>
char *strtok(char *str1, const char *str2);
```

La funzione strtok() restituisce un puntatore al token successivo nella stringa puntata da *str1*. i caratteri che compongono la stringa puntata da *str2* sono i delimitatori che separano i token. Se non vi sono token viene restituito un puntatore nullo.

Per trasformare una stringa in token la prima chiamata a strtok() deve fare in modo che *str1* punti alla stringa da trasformare in token. Le chiamate successive devono usare per *str1* un puntatore nullo. In questo modo è possibile ridurre in token l'intera stringa.

È possibile usare delimitatori differenti per ciascuna chiamata a strtok().

Le funzioni correlate sono strchr(), strcspn(), strpbrk(), strrchr() e strspn().

## strxfrm()

```
#include <cstring>
size_t strxfrm(char *str1, const char *str2, size_t count);
```

La funzione strxfrm() trasforma i primi *count* caratteri della stringa puntata da *str2* in modo che possano essere utilizzati dalla funzione strcmp() e inserisce il risultato nella stringa puntata da *str1*. Dopo la trasformazione, i risultati di strcmp() su *str1* e strcoll() sulla stringa originale puntata da *str2* saranno identici.

La funzione strxfrm() restituisce la lunghezza della stringa trasformata.

Una funzione correlata è strcoll().

### tolower()

```
#include <cctype>
int tolower(int ch);
```

La funzione tolower() restituisce l'equivalente minuscolo di *ch* (se *ch* è una lettera); altrimenti *ch* viene restituito così come è.

Una funzione correlata è toupper().

### toupper()

```
#include <cctype>
int toupper(int ch);
```

La funzione toupper() restituisce l'equivalente maiuscolo di *ch* (se *ch* è una lettera); altrimenti *ch* viene restituito così come è.

Una funzione correlata è tolower().

Capitolo 27

# Le funzioni matematiche

La libreria delle funzioni standard contiene numerose funzioni matematiche che rientrano nelle seguenti categorie:

- funzioni trigonometriche;
- funzioni iperboliche;
- funzioni esponenziali e logaritmiche;
- funzioni varie

Tutte le funzioni matematiche richiedono l'impiego dell'header <cmath> (i programmi C devono usare il file header <math.h>). Oltre a dichiarare le funzioni matematiche, questo header definisce la macro HUGE_VAL. Anche le macro EDOM ed ERANGE sono molto impiegate dalle funzioni matematiche. Queste macro sono definite nell'header <cerrno> (o nel file header <errno.h>). Se un argomento di una funzione matematica non è nel dominio richiesto, viene restituito un valore definito dall'implementazione e alla variabile intera globale errno viene assegnato il valore EDOM. Se una routine produce un risultato troppo esteso per essere rappresentato, si verifica un overflow. In questo caso la routine restituisce HUGE_VAL e a errno viene assegnato ERANGE. Se si verifica un underflow, la funzione restituisce il valore 0 e assegna a errno il valore ERANGE.

Tutti gli angoli sono espressi in radianti.

Originariamente le funzioni matematiche erano specificate in modo da operare su valori di tipo double ma il C++ standard ha aggiunto delle versioni modificate tramite overloading in grado di accettare esplicitamente valori di tipo float e long double. Per il resto l'operatività delle funzioni non risulta modificata.

### acos()

```
#include <cmath>
float acos(float arg);
```

```
double acos(double arg);
long double acos(long double arg);
```

La funzione acos() restituisce l'arco coseno di *arg*. L'argomento di acos() deve essere compreso tra –1 e 1; altrimenti si verifica un errore di dominio.

Le funzioni correlate sono asin(), atan(), atan2(), sin(), cos(), tan(), sinh(), cosh() e tanh().

## asin()

```
#include <cmath>
float asin(float arg);
double asin(double arg);
long double asin(long double arg);
```

La funzione asin() restituisce l'arco seno di *arg*. L'argomento di asin() deve essere compreso tra –1 e 1; altrimenti si verifica un errore di dominio.

Le funzioni correlate sono acos(), atan(), atan2(), sin(), cos(), tan(), sinh(), cosh() e tanh().

## atan()

```
#include <cmath>
float atan(float arg);
double atan(double arg);
long double atan(long double arg);
```

La funzione atan() restituisce l'arco tangente di *arg*.

Le funzioni correlate sono asin(), acos(), atan2(), tan(), cos(), sin(), sinh(), cosh() e tanh().

## atan2()

```
#include <cmath>
float atan2(float y, float x);
double atan2(double y, double x);
long double atan2(long double y, long double x);
```

La funzione atan2() restituisce l'arco tangente di *y/x*. Per calcolare il quadrante del valore restituito utilizza il segno dei suoi argomenti.



Le funzioni correlate sono asin(), acos(), atan(), tan(), cos(), sin(), sinh(), cosh() e tanh().

## ceil()

```
#include <cmath>
float ceil(float num);
double ceil(double num);
long double ceil(long double num);
```

La funzione ceil() restituisce il più piccolo intero (rappresentato come un valore in virgola mobile) non minore di *num*. Ad esempio, dato 1.02, ceil() restituisce 2.0. Dato –1.02, ceil() restituisce –1.

Le funzioni correlate sono floor() e fmod().

## cos()

```
#include <cmath>
float cos(float arg);
double cos(double arg);
long double cos(long double arg);
```

La funzione cos() restituisce il coseno di *arg*. Il valore di *arg* deve essere espresso in radianti.

Le funzioni correlate sono asin(), acos(), atan2(), atan(), tan(), sin(), sinh(), cos() e tanh().

## cosh()

```
#include <cmath>
float cosh(float arg);
double cosh(double arg);
long double cosh(long double arg);
```

La funzione cosh() restituisce il coseno iperbolico di *arg*.

Le funzioni correlate sono asin(), acos(), atan2(), atan(), tan(), sin(), cosh() e tanh().

## exp()

```
#include <cmath>
float exp(float arg);
```

```
double exp(double arg);
long double exp(long double arg);
```

La funzione exp() restituisce il logaritmo natuale in base *e* elevato alla potenza *arg*.

Una funzione correlata è log( ).

## fabs()

```
#include <cmath>
float fabs(float num);
double fabs(double num);
long double fabs(long double num);
```

La funzione fabs() restituisce il valore assoluto di *num*.

Una funzione correlata è abs().

## floor()

```
#include <cmath>
float floor(float num);
double floor(double num);
long double floor(long double num);
```

La funzione floor() restituisce il massimo intero (rappresentato come un valore in virgola mobile) non maggiore di *num*. Ad esempio, dato 1.02, floor() restituisce 1.0. Dato –1.02, floor() restituisce –2.0.

Le funzioni correlate sono fceil() e fmod().

## fmod()

```
#include <cmath>
float fmod(float x, float y);
double fmod(double x, double y);
long double fmod(long double x, long double y);
```

La funzione fmod() restituisce il resto di *x/y*.

Le funzioni correlate sono ceil(), floor() e fabs().



## frexp()

```
#include <cmath>
float frexp(float num, int *exp);
double frexp(double num, int *exp);
long double frexp(long double num, int *exp);
```

La funzione frexp() decompone il numero *num* in una mantissa compresa fra 0.5 e 1 e un esponente intero tale che *num* = *mantissa* * $2^{exp}$. La mantissa viene restituita dalla funzione e l'esponente viene memorizzato nella variabile puntata da *exp*.

Una funzione correlata è ldexp().

## ldexp()

```
#include <cmath>
float ldexp(float num, int exp);
double ldexp(double num, int exp);
long double ldexp(long double num, int exp);
```

La funzione ldexp() restituisce il valore di *num* * $2^{exp}$. Nel caso di overflow viene restituito il valore HUGE_VAL.

Le funzioni correlate sono frexp() e modf().

## log()

```
#include <cmath>
float log(float num);
double log(double num);
long double log(long double num);
```

La funzione log( ) restituisce il logaritmo natulae di *num*. Se *num* è negativo si verifica un errore di dominio mentre se l'argomento è 0 si verifica un errore di intervallo.

Una funzione correlata è log10().

## log10()

```
#include <cmath>
float log10(float num);
```

```
double log10(double num);
long double log10(long double num);
```

La funzione log10() restituisce il logaritmo in base 10 di *num*. Se *num* è negativo si verifica un errore di dominio mentre se l'argomento è 0 si verifica un errore di intervallo.

Una funzione correlata è log( ).

## modf()

```
#include <cmath>
float modf(float num, float *i);
double modf(double num, double *i);
long double modf(long double num, long double *i);
```

La funzione modf()decompone *num* nelle sue parti intera e frazionale. Restituisce la parte frazionale e inserisce la parte intera nella variabile puntata da *i*.

Le funzioni correlate sono frexp() e ldexp().

## pow()

```
#include <cmath>
float pow(float base, float exp);
float pow(float base, int exp);
double pow(double base, double exp);
double pow(double base, int exp);
long double pow(long double base, long double exp);
long double pow(long double base, int exp);
```

La funzione pow() restituisce *base* elevata alla potenza *exp* (*base*$^{exp}$). Se *base* è uguale a 0 e *exp* è minore o uguale a 0 si verifica un errore di dominio. Questo accade anche se *base* è negativa e *exp* non è un intero. Un overflow produce un errore di intervallo.

Le funzioni correlate sono exp(), log( ) e sqrt().

## sin()

```
#include <cmath>
float sin(float arg);
double sin(double arg);
long double sin(long double arg);
```



La funzione sin() restituisce the sine di *arg*. Il valore di *arg* deve essere espresso in radianti.

Le funzioni correlate sono asin(), acos(), atan2(), atan(), tan(), cos(), sinh(), cosh() e tanh().

## sinh()

```
#include <cmath>
float sinh(float arg);
double sinh(double arg);
long double sinh(long double arg);
```

La funzione sinh() restituisce il seno iperbolico di *arg*.

Le funzioni correlate sono asin(), acos(), atan2(), atan(), tan(), cos(), tanh(), cosh() e sin().

## sqrt()

```
#include <cmath>
float sqrt(float num);
double sqrt(double num);
long double sqrt(long double num);
```

La funzione sqrt() restituisce la radice quadrata di *num*. Se viene richiamata con un argomento negativo, si verifica un errore di dominio.

Le funzioni correlate sono exp(), log( ) e pow().

## tan()

```
#include <cmath>
float tan(float arg);
double tan(double arg);
long double tan(long double arg);
```

La funzione tan() restituisce la tangente *arg*. Il valore di *arg* deve essere espresso in radianti.

Le funzioni correlate sono acos(), asin(), atan(), atan2(), cos(), sin(), sinh(), cosh() e tanh().

## tanh()

```
#include <cmath>
float tanh(float arg);
double tanh(double arg);
long double tanh(long double arg);
```

La funzione tanh() restituisce la tangente iperbolica di *arg*.

Le funzioni correlate sono acos(), asin(), atan(), atan2(), cos(), sin(), cosh(), sinh() e tan().

Capitolo 28

# Le funzioni per le date, le ore e la localizzazione

La libreria delle funzioni standard definisce numerose funzioni che si occupano di date e ore. Inoltre definisce funzioni che gestiscono tutte le informazioni geopolitiche associate a un programma.

Le funzioni per date e ore richiedono l'uso dell'header <ctime> (un programma C deve usare il file header <time.h>). Questo header definisce tre tipi correlati con il tempo: clock_t, time_t e tm. I tipi clock_t e time_t consentono di rappresentare l'ora e la data di sistema sotto forma di interi (la cosiddetta *ora di calendario*). La struttura tm contiene la data e l'ora suddivise nei rispettivi elementi ed è definita nel seguente modo:

```
struct tm {
  int tm_sec;  /* secondi, 0-61 */
  int tm_min;  /* minuti, 0-59 */
  int tm_hour; /* ore, 0-23 */
  int tm_mday; /* giorno del mese, 1-31 */
  int tm_mon;  /* mese da Gennaio, 0-11 */
  int tm_year; /* anno dal 1900 */
  int tm_wday; /* giorni da Domenica, 0-6 */
  int tm_yday; /* giorni dal primo Gennaio, 0-365 */
  int tm_isdst /* indicatore
                  ora legale */
}
```

Il valore di tm_isdst è positivo quando è attiva l'ora legale, 0 se non è attiva l'ora legale e negativa se non sono disponibili informazioni. Questa forma dell'ora e della data è chiamata *ora suddivisa*.

Inoltre <ctime> definisce la macro CLOCKS_PER_SEC che equivale al numero di impulsi di sistema al secondo.

Le funzioni per la gestione delle informazioni geopolitiche d'ambiente richiedono l'header <locale> (un programma C deve usare il file header <locale.h>).

## asctime()

```
#include <ctime>
char *asctime(const struct tm *ptr);
```

La funzione asctime() restituisce un puntatore a una stringa che contiene le informazioni memorizzate nella struttura puntata da *ptr* convertite nella seguente forma:

```
giorno mese data ore:minuti:secondi anno \n\0
```

Ad esempio:

```
Wed Jun 19 12:05:34 1999
```

Il puntatore a struttura passato a asctime() viene in genere ottenuto da localtime() o gmtime().

Il buffer utilizzato da asctime() per contenere la stringa di output formattato è un array di caratteri allocato in modo statico che viene sovrascritta ogni volta che viene richiamata la funzione asctime(). Per salvare il contenuto della stringa occorre copiarla altrove.

Le funzioni correlate sono localtime(), gmtime(), time() e ctime().

## clock()

```
#include <ctime>
clock_t clock(void);
```

La funzione clock() restituisce un valore che approssima la durata dell'esecuzione del programma. Per trasformare il valore in secondi, occorre dividerlo per CLOCKS_PER_SEC. Se l'ora non è disponibile viene restituito il valore -1.

Le funzioni correlate sono time(), asctime() e ctime().

## ctime()

```
#include <ctime>
char *ctime(const time_t *time);
```

La funzione ctime() restituisce un puntatore a una stringa nella forma:

*giorno mese anno ore:minuti:secondi anno*



dato un puntatore all'ora di calendario. L'ora di calendario viene in genere ottenuta tramite una chiamata a time().

Il buffer utilizzato da ctime() per contenere la stringa di output formattato è un array di caratteri allocato in modo statico che viene sovrascritta ogni volta che viene richiamata la funzione ctime(). Per salvare il contenuto della stringa occorre copiarla altrove.

Le funzioni correlate sono localtime(), gmtime(), time() e asctime().

## difftime()

```
#include <ctime>
double difftime(time_t time2, time_t time1);
```

La funzione difftime() restituisce la differenza in secondi fra *time1* e *time2*, ovvero *time2 - time1*.

Le funzioni correlate sono localtime(), gmtime(), time(), asctime().

## gmtime()

```
#include <ctime>
struct tm *gmtime(const time_t *time);
```

La funzione gmtime() restituisce un puntatore alla forma suddivisa di *time* in una struttura *tm*. L'ora è rappresentata in termini di tempo UTC (Coordinated Universal Time) che corrisponde in pratica all'ora mediana di Greenwich. Il valore *time* è in genere ottenuto tramite una chiamata a time(). Se il sistema non supporta l'ora UTC viene restituito il valore NULL.

La struttura usata da gmtime() per contenere l'ora così suddivisa viene allocata in modo statico ed è sovrascritta ogni volta che viene richiamata la funzione. Dunque per salvare il contenuto della struttura occorre copiarla altrove.

Le funzioni correlate sono localtime(), time() e asctime().

## localeconv()

```
#include <clocale>
struct lconv *localeconv(void);
```

La funzione localeconv() restituisce un puntatore a una struttura di tipo *lconv* che contiene varie informazioni geopolitiche d'ambiente relative alla formattazione dei valori numerici. La struttura *lconv* è organizzata nel seguente modo:

```
struct lconv {
  char *decimal_point;      /* carattere di punto decimale
                               per valori non monetari */
  char *thousands_sep;      /* separatore delle migliaia
                               per valori non monetari */
  char *grouping;           /* specifica il raggruppamento per
                               valori non monetari */
  char *int_curr_symbol;    /* simbolo di valuta internazionale */
  char *currency_symbol;    /* simbolo di valuta locale */
  char *mon_decimal_point; /* carattere di punto decimale per
                               valori monetari */
  char *mon_thousands_sep; /* separatore delle migliaia per
                               valori monetari */
  char *mon_grouping;       /* specifica il raggruppamento per
                               valori monetari */
  char *positive_sign;      /* indicatore di valori positivi per
                               valori monetari */
  char *negative_sign;      /* indicatore di valori negativi per
                               valori monetari */
  char int_frac_digits;     /* numero di cifre visualizzate a
                               destra del punto decimale per
                               valori monetari visualizzati con
                               il formato internazionale */
  char frac_digits;         /* numero di cifre visualizzate a
                               destra del punto decimale per
                               valori monetari visualizzati con
                               il formato locale */
  char p_cs_precedes;       /* 1 se il simbolo della valuta precede
                               un valore positivo, 0 se il simbolo
                               della valuta segue il valore */
  char p_sep_by_space;      /* 1 se il simbolo della valuta è
                               separato dal valore con uno spazio,
                               0 altrimenti */
  char n_cs_precedes;       /* 1 se il simbolo della valuta precede
                               un valore negativo, 0 se il simbolo
                               della valuta segue il valore */
  char n_sep_by_space;      /* 1 se il simbolo della valuta è
                               separato da un valore negativo
                               con uno spazio, 0 se il
                               simbolo della valuta segue il valore */
  char p_sign_posn;         /* indica la posizione del
                               simbolo di valore positivo */
  char n_sign_posn;         /* indica la posizione del
                               simbolo di valore negativo */
}
```



La funzione localeconv() restituisce un puntatore alla struttura *lconv*. Non si deve modificare il contenuto di questa struttura. Per informazioni specifiche sulla struttura *lconv*, consultare la documentazione del compilatore.

La funzione correlata è setlocale().

## localtime()

```
#include <ctime>
struct tm *localtime(const time_t *time);
```

La funzione localtime() restituisce un puntatore a una forma suddivisa di *time* in una struttura *tm*. L'ora è rappresentata in termini locali. Il valore *time* è in genere ottenuto tramite una chiamata a time().

La struttura usata da localtime() per contenere l'ora così suddivisa viene allocata in modo statico ed è sovrascritta ogni volta che viene richiamata la funzione. Dunque per salvare il contenuto della struttura occorre copiarla altrove.

Le funzioni correlate sono gmtime(), time() e asctime().

## mktime()

```
#include <ctime>
time_t mktime(struct tm *time);
```

La funzione mktime() restituisce un'ora di calendario equivalente all'ora suddivisa presente nella struttura puntata da *time*. Gli elementi tm_wday e tm_yday vengono impostati dalla funzione e dunque non devono essere definiti al momento della chiamata. Se mktime() non può rappresentare le informazioni come un'ora valida, restituisce -1.

Le funzioni correlate sono time(), gmtime(), asctime() e ctime().

## setlocale()

```
#include <clocale>
char *setlocale(int type, const char *locale);
```

La funzione setlocale() consente di interrogare o impostare alcuni parametri relativi all'ambiente geopolitico di esecuzione del programma. Se *locale* è null, setlocale() restituisce un puntatore alla stringa di localizzazione corrente. Altrimenti setlocale() tenta di utilizzare la stringa specificata da *locale* per impostare i parametri locali nel modo specificato da *type*. Per informazioni sulle stringhe di localizzazione, consultare la documentazione del compilatore.

Al momento della chiamata, *type* deve essere una delle seguenti macro:

LC_ALL
LC_COLLATE
LC_CTYPE
LC_MONETARY
LC_NUMERIC
LC_TIME

LC_ALL fa riferimento a tutte le categorie di localizzazione. LC_COLLATE riguarda le operazioni della funzione strcoll(). LC_CTYPE modifica l'operatività delle funzioni per caratteri. LC_MONETARY determina il formato monetario. LC_NUMERIC cambia il carattere di punto decimale per le funzioni di input e output formattato. Infine LC_TIME determina il comportamento della funzione strftime(). La funzione setlocale() restituisce un puntatore una stringa associata al parametro *type*.

Le funzioni correlate sono localeconv(), time(), strcoll() e strftime().

## strftime()

```
#include <ctime>
size_t strftime(char *str, size_t maxsize, const char *fmt,
                const struct tm *time);
```

La funzione strftime() inserisce nella stringa puntata da *str* le informazioni relative alla data e all'ora e anche altre informazioni utilizzando comandi di formattazione contenuti nella stringa puntata da *fmt* impiegando l'ora specificata in *time*. In *str* verrà inserito un massimo di *maxsize* caratteri.

La funzione strftime() si comporta un po' come sprintf() in quanto riconosce una serie di comandi di formattazione che iniziano con il segno di percentuale (%) e inserisce l'output formattato in una stringa. I comandi di formattazione vengono utilizzati per specificare il modo in cui le informazioni devono essere rappresentate in *str*. Ogni altro carattere presente nella stringa di formattazione verrà inserito in *str* senza alcuna modifica. La data e l'ora fanno riferimento all'ora locale. Nella seguente tabella sono elencati i comandi di formattazione. Si noti che la maggior parte dei comandi distingue fra lettere maiuscole e minuscole.

La funzione strftime() restituisce il numero di caratteri inseriti nella stringa puntata da *str* oppure 0 se si verifica un errore.



| COMANDO | SOSTITUZIONE |
|---|---|
| %a | Giorno della settimana abbreviato |
| %A | Giorno della settimana completo |
| %b | Nome del mese abbreviato |
| %B | Nome del mese completo |
| %c | Stringa standard per la data e l'ora |
| %d | Giorno del mese (1-31) |
| %H | Ora (0-23) |
| %I | Ora (1-12) |
| %j | Giorno dell'anno (1-366) |
| %m | Mese (1-12) |
| %M | Minuto (0-59) |
| %p | Equivalente locale di AM o PM |
| %S | Secondo (0-59) |
| %U | Settimana (0-53); il primo giorno è domenica |
| %w | Giorno della settimana (0-6); domenica = 0 |
| %W | Settimana (0-53); il primo giorno è lunedì |
| %x | Stringa standard per la data |
| %X | Stringa standard per l'ora |
| %y | Anno senza secolo (0-99) |
| %Y | Anno con secolo (in decimale) |
| %Z | Nome fuso orario |
| %% | Segno di percentuale |

Le funzioni correlate sono time(), localtime() e gmtime().

## time()

```
#include <ctime>
time_t time(time_t *time);
```

La funzione time() restituisce l'ora di calendario corrente del sistema. Se il sistema non supporta l'ora, restituisce -1.

La funzione time() può essere richiamata con un puntatore nullo oppure con un puntatore a una variabile di tipo *time_t*. Nel secondo caso alla variabile verrà assegnata anche l'ora di calendario.

Le funzioni correlate sono localtime(), gmtime(), strftime() e ctime().

Capitolo 29

# Le funzioni di allocazione dinamica della memoria

Questo capitolo descrive le funzioni di allocazione dinamica della memoria, così come vengono ereditate dal linguaggio C. Il nucleo di tali funzioni è costituito da malloc() e free(). Ogni volta che si richiama malloc(), viene allocata una porzione della memoria libera. Ogni volta che si richiama free() la memoria viene restituita al sistema. La regione da cui viene tratta l'area di memoria si chiama heap. I prototipi delle funzioni di allocazione dinamica si trovano in <cstdlib>. Un programma C deve usare il file header <stdlib.h>.

Tutti i compilatori C++ includono almeno le seguenti funzioni di allocazione dinamica della memoria: calloc(), malloc(), free() e realloc(). Quasi certamente il compilatore fornirà molte varianti di queste funzioni in modo da rispondere a varie esigenze e da adattarsi alle caratteristiche dell'ambiente operativo. A tale proposito si rimanda alla documentazione del compilatore.

Anche se il linguaggio C++ supporta le funzioni di allocazione dinamica della memoria qui descritte, in genere esse non vengono utilizzate nei programmi C++. Questo è dovuto al fatto che il linguaggio C++ definisce appositi operatori di allocazione dinamica della memoria chiamati new e delete. Questi presentano notevoli vantaggi rispetto alle funzioni di allocazione dinamica della memoria. Innanzitutto new alloca automaticamente la quantità di memoria corretta per il tipo di dati specificato. In secondo luogo restituisce un puntatore del tipo corretto. Infine sia new che delete possono essere modificati tramite overloading. Visti questi vantaggi rispetto alle funzioni di allocazione dinamica della memoria del C, si consiglia di utilizzare gli operatori al posto delle funzioni.

## calloc()

```
#include <cstdlib>
void *calloc(size_t num, size_t size);
```

La funzione calloc() alloca un'area di memoria le cui dimensioni sono uguali a *num* * *size*. Pertanto calloc() alloca un'area di memoria sufficiente per un array di *num* oggetti di dimensioni *size*. La funzione calloc() restituisce un puntatore al primo byte della regione allocata. Se la memoria è insufficiente per soddisfare la richiesta, viene restituito un puntatore nullo. È sempre importante verificare che il valore restituito non sia nullo prima di tentare di usarlo.

Le funzioni correlate sono free(), malloc() e realloc().

## free()

```
#include <cstdlib>
void free(void *ptr);
```

La funzione free() restituisce all'heap la memoria puntata da *ptr*. L'area tornerà ad essere disponibile per ogni successiva operazione di allocazione.

È fondamentale che free() venga richiamata con il puntatore precedentemente allocato utilizzando una delle funzioni di allocazione dinamica (calloc() o malloc()). L'uso di un puntatore non valido provoca con ogni probabilità la distruzione del meccanismo di gestione della memoria e pertanto un blocco del sistema.

Le funzioni correlate sono calloc(), malloc() e realloc().

## malloc()

```
#include <cstdlib>
void *malloc(size_t size);
```

La funzione malloc() restituisce un puntatore al primo byte di una regione di memoria di dimensioni pari a *size* allocata nello heap. Se la memoria è insufficiente per soddisfare la richiesta, malloc() restituisce un puntatore nullo. È sempre importante verificare che il valore restituito non sia nullo prima di tentare di utilizzare il puntatore. Il tentativo di utilizzare un puntatore NULL provoca generalmente un blocco del sistema.

Le funzioni correlate sono free(), realloc() e calloc().

## realloc()

```
#include <cstdlib>
void *realloc(void *ptr, size_t size);
```

La funzione realloc() cambia le dimensioni dell'area di memoria precedentemente allocata da *ptr* in modo da ottenere uno spazio pari a *size*. Il valore di *size*



può essere maggiore o minore del precedente. realloc() restituisce un puntatore al nuovo blocco di memoria poiché un aumento delle dimensioni può richiedere che realloc() sposti l'intero blocco. In questo caso il contenuto del vecchio blocco viene copiato nel nuovo blocco senza alcuna perdita di informazioni.

Se *ptr* è nullo, realloc() alloca semplicemente *size* byte di memoria e restituisce un puntatore a tale area. Se *size* è 0, la memoria puntata da *ptr* viene liberata.

Se la memoria disponibile nello heap è insufficiente per allocare *size* byte, la funzione restituisce un puntatore nullo e il blocco originale rimane non modificato.

Le funzioni correlate sono free(), malloc() e calloc().

Capitolo 30

# Le funzioni di servizio

La libreria delle funzioni standard definisce numerose funzioni che forniscono vari tipi di servizi. Fra di esse vi sono funzioni di conversione, funzioni per l'elaborazione di liste di argomenti di lunghezza variabile, funzioni di ordinamento e ricerca e funzioni per la generazione di numeri casuali. La maggior parte delle funzioni descritte in questo capitolo richiede l'uso dell'header <cstdlib> (un programma C dovrà utilizzare il file header <stdlib.h>). In questo header sono definiti div_t e ldiv_t che sono i tipi dei valori restituiti rispettivamente da div() e ldiv(). Inoltre viene definito il tipo size_t che è il tipo del valore unsigned restituito da sizeof. Inoltre sono definite le seguenti macro:

| MACRO | SIGNIFICATO |
|---|---|
| NULL | Un puntatore nullo |
| RAND_MAX | Il valore massimo che può essere restituito dalla funzione rand(). |
| EXIT_FAILURE | Il valore restituito al processo chiamante quando il programma è terminato senza successo. |
| EXIT_SUCCESS | Il valore restituito al processo chiamante se il programma è terminato con successo. |

Per le funzioni che impiegano un header diverso da <cstdio> è stato indicato l'header da includere.

## abort()

```
#include <cstdlib>
void abort(void);
```

La funzione abort() provoca l'immediata chiusura anormale del programma. Generalmente i buffer non vengono svuotati su file. In alcuni ambienti,

abort() restituisce al processo chiamante (in genere il sistema operativo) un valore definito dall'implementazione per indicare che il programma è terminato con un insuccesso.

Le funzioni correlate sono exit() e atexit().

## abs()

```
#include <cstdlib>
int abs(int num);
long abs(long num);
double abs(double num);
```

La funzione abs() restituisce il valore assoluto di *num*. La versione long di abs() è uguale a labs(). La versione double di abs() è uguale a fabs().

Una funzione correlata è labs().

## assert()

```
#include <cassert>
void assert(int exp);
```

La macro assert(), definita nell'header <cassert> scrive le informazioni di errore su stderr e quindi chiude l'esecuzione del programma se l'espressione *exp* restituisce 0. Altrimenti assert() non fa nulla. Anche se l'output è definito dall'implementazione, la maggior parte dei compilatori usa un messaggio simile al seguente:

Assertion failed: <*expression*>, file <*file*>, line <*linenum*>

La macro assert() viene generalmente utilizzata per verificare che il programma stia operando in modo corretto e l'espressione è organizzata in modo da fornire true solo quando non si è verificato alcun errore.

Non è necessario rimuovere le istruzioni assert() dal codice sorgente dopo il debug del programma: basta definire la macro NDEBUG e tutte le macro assert() verranno ignorate.

Una funzione correlata è abort().

## atexit()

```
#include <cstdlib>
int atexit(void (*func)(void));
```



La funzione atexit() fa in modo che alla normale terminazione del programma venga richiamata la funzione puntata da *func*. Se la funzione è registrata con successo come funzione di chiusura del programma, la funzione atexit() restituisce il valore 0, altrimenti restituisce un valore diverso da 0.

Si possono definire almeno 32 funzioni di chiusura del programma, le quali verranno richiamate in ordine inverso rispetto alla loro definizione.

Le funzioni correlate sono exit() e abort().

## atof()

```
#include <cstdlib>
double atof(const char *str);
```

La funzione atof() converte la stringa puntata da *str* in un valore double. La stringa deve contenere un numero in virgola mobile valido. Se questo non avviene, il valore restituito è indefinito.

Il numero può essere concluso da qualsiasi carattere che non possa costituire un numero in virgola mobile. Ad esempio può trattarsi di uno spazio vuoto, di un segno di punteggiatura (ad eccezione del punto) o di un carattere (diverso da e o E). Questo significa che se atof() viene richiamata sulla stringa "100.00SALVE", verrà restituito il valore 100.00.

Le funzioni correlate sono atoi() e atol().

## atoi()

```
#include <cstdlib>
int atoi(const char *str);
```

La funzione atoi() converte la stringa puntata da *str* in un valore int. La stringa deve contenere un numero intero valido. Se questo non avviene, il valore restituito è indefinito; tuttavia molte implementazioni restituiscono il valore 0.

Il numero può essere concluso da qualsiasi carattere che non possa costituire un numero intero. Ad esempio può trattarsi di uno spazio vuoto, di un segno di punteggiatura o di un carattere. Questo significa che se atoi() viene richiamata sulla stringa "123.23", verrà restituito il valore 123 e ".23" verrà ignorato.

Le funzioni correlate sono atof() e atol().

## atol()

```
#include <cstdlib>
long atol(const char *str);
```

La funzione atol() converte la stringa puntata da *str* in un valore long. La stringa deve contenere un numero long valido. Se questo non avviene, il valore restituito è indefinito; tuttavia molte implementazioni restituiscono il valore 0.

Il numero può essere concluso da qualsiasi carattere che non possa costituire un numero intero. Ad esempio può trattarsi di uno spazio vuoto, di un segno di punteggiatura o di un carattere. Questo significa che se atol() viene richiamata sulla stringa "123.23", verrà restituito il valore 123L e ".23" verrà ignorato.

Le funzioni correlate sono atof() e atoi().

## bsearch()

```
#include <cstdlib>
void *bsearch(const void *key, const void *buf,
              size_t num, size_t size,
              int (*compare)(const void *, const void *));
```

La funzione bsearch() esegue una ricerca binaria sull'array ordinato puntato da *buf* e restituisce un puntatore al primo membro che corrisponde alla chiave puntata da *key*. *num* indica il numero di elementi dell'array mentre *size* indica le dimensioni in byte di ciascun elemento.

Per confrontare un elemento dell'array con la chiave si usa la funzione puntata da compare(). La funzione compare() deve avere la seguente forma:

int *func_name*(const void **arg1*, const void **arg2*);

Inoltre la funzione deve restituire un valore secondo le regole descritte dalla seguente tabella.

| CONFRONTO | VALORE RESTITUITO |
|---|---|
| *arg1* è minore di *arg2* | Minore di 0 |
| *arg1* è uguale ad *arg2* | 0 |
| *arg1* è maggiore di *arg2* | Maggiore di 0 |

L'array deve essere ordinato in senso ascendente e l'indirizzo più basso deve contenere l'elemento inferiore.

Se l'array non contiene la chiave, bsearch() restituisce un puntatore nullo.

Una funzione correlata è qsort().



## div()

```
#include <cstdlib>
div_t div(int numerator, int denominator);
ldiv_t div(long numerator, long denominator);
```

La versione int della funzione div() restituisce il quoziente e per il resto dell'operazione numeratore / denominatore in una struttura di tipo div_t. La versione long di div() ha le stesse caratteristiche della funzione ldiv().

La struttura div_t contiene quanto meno i due campi seguenti:

```
int quot; /* quoziente */
int rem;  /* resto */
```

La struttura ldiv_t contiene quanto meno i due campi seguenti:

```
long quot; /* quoziente */
long rem;  /* resto */
```

Una funzione correlata è ldiv().

## exit()

```
#include <cstdlib>
void exit(int exit_code);
```

La funzione exit() provoca l'immediata chiusura (normale) del programma. Al processo chiamante (normalmente il sistema operativo) viene passato il valore exit_code, sempre che l'ambiente operativo supporti l'operazione. Per convenzione, se il valore di exit_code è 0 o EXIT_SUCCESS si presume che il programma sia terminato naturalmente. Un valore diverso da 0 oppure EXIT_FAILURE indica un errore definito dall'implementazione.

Le funzioni correlate sono atexit() e abort().

## getenv()

```
#include <cstdlib>
char *getenv(const char *name);
```

La funzione getenv() restituisce un puntatore alle informazioni d'ambiente associate alla stringa puntata da *name* all'interno della tabella delle informazioni

d'ambiente definite dall'implementazione. La stringa restituita non deve mai essere modificata dal programma.

L'ambiente di un programma può includere percorsi e dispositivi. L'esatta natura di questi dati è definita dall'implementazione. Per informazioni è opportuno consultare la documentazione del compilatore.

Se viene eseguita una chiamata a getenv() con un argomento non corrispondente ai dati d'ambiente, getenv() restituisce un puntatore nullo.

Una funzione correlata è system().

## labs()

```
#include <cstdlib>
long labs(long num);
```

La funzione labs() restituisce il valore assoluto di *num*.

Una funzione correlata è abs().

## ldiv()

```
#include <cstdlib>
ldiv_t ldiv(long numerator, long denominator);
```

La funzione ldiv() restituisce il quoziente e il resto della divisione numeratore / denominatore.

La struttura ldiv_t contiene quanto meno i due campi seguenti:

```
long quot; /* quoziente */
long rem;  /* resto */
```

Una funzione correlata è div().

## longjmp()

```
#include <csetjmp>
void longjmp(jmp_buf envbuf, int status);
```

La funzione longjmp() fa in modo che l'esecuzione del programma riprenda dal punto dell'ultima chiamata a setjmp(). Queste due funzioni forniscono un mezzo per saltare fuori dalle funzioni. Si noti che è necessario impiegare l'header <csetjmp>.



La funzione longjmp() esegue il reset dello stack allo stato descritto da *envbuf* che deve essere stato impostato tramite una chiamata a setjmp(). Questo fa in modo che l'esecuzione del programma riprenda dall'istruzione che seguiva la chiamata a setjmp(). Pertanto il computer viene "ingannato" poiché riterrà di non aver mai lasciato la funzione che ha richiamato setjmp(). In pratica la funzione longjmp() esegue un salto nel tempo e nello spazio (di memoria) a un punto del programma senza seguire il normale processo di uscita dalle funzioni.

Il buffer *envbuf* è di tipo jmp_buf, definito nell'header <csetjmp>. Prima di richiamare longjmp(), il buffer deve essere stato impostato tramite una chiamata a setjmp().

Il valore di status diviene il valore restituito da longjmp() e può essere interrogato per determinare il luogo di provenienza del salto. L'unico valore non consentito è 0.

L'uso più comune di longjmp() consiste nell'uscire da un insieme molto profondo di routine quando si verifica un errore.

Una funzione correlata è setjmp().

## mblen()

```
#include <cstdlib>
int mblen(const char *str, size_t size);
```

La funzione mblen() restituisce la lunghezza in byte di un carattere multibyte puntato da *str*. Vengono esaminati solo i primi *size* caratteri. In caso di errore la funzione restituisce -1.

Se *str* è nulla e i caratteri multibyte hanno una codifica che dipende dallo stato, la funzione restituisce un valore diverso da 0. In caso contrario restituisce 0.

Le funzioni correlate sono mbtowc() e wctomb().

## mbstowcs()

```
#include <cstdlib>
size_t mbstowcs(wchar_t *out, const char *in, size_t size);
```

La funzione mbstowcs() converte la stringa multibyte puntata da *in* in una stringa di caratteri estesi e inserisce il risultato nell'array puntato da *out*. In *out* verranno memorizzati solo *size* byte.

La funzione mbstowcs() restituisce il numero di caratteri multibyte convertiti. In caso di errore la funzione restituisce -1.

Le funzioni correlate sono wcstombs() e mbtowc().

## mbtowc()

```
#include <cstdlib>
int mbtowc(wchar_t *out, const char *in, size_t size);
```

La funzione mbtowc() converte il carattere multibyte contenuto nell'array puntato da *in* nell'equivalente a caratteri estesi e inserisce il risultato nell'oggetto puntato da *out*. Verranno esaminati solo *size* caratteri.

Questa funzione restituisce il numero inserito in *out*. in caso di errore la funzione restituisce -1. Se *in* è nullo e i caratteri multibyte hanno dipendenze di stato, allora mbtowc() restituisce un valore diverso da 0; altrimenti la funzione restituisce il valore 0.

Le funzioni correlate sono mblen(), wctomb().

## qsort()

```
#include <cstdlib>
void qsort(void *buf, size_t num, size_t size,
           int (*compare) (const void *, const void *));
```

La funzione qsort() ordina l'array puntato da *buf* utilizzando l'algoritmo Quicksort (sviluppato da Hoare). Il Quicksort è il miglior algoritmo di ordinamento di carattere generale. Al termine dell'algoritmo l'array sarà ordinato. Il numero di elementi dell'array è specificato da *num* e le dimensioni in byte di ciascun elemento sono descritte da *size*.

Per confrontare un elemento dell'array con la chiave si usa *compare*. *compare* deve avere la seguente forma:

int *func_name*(const void **arg1*, const void **arg2*);

Il codice restituito deve seguire le regole contenute nella tabella.

| CONFRONTO | VALORE RESTITUITO |
|---|---|
| *arg1* è minore di *arg2* | Minore di 0 |
| *arg1* è uguale ad *arg2* | 0 |
| *arg1* è maggiore di *arg2* | Maggiore di 0 |

L'array è viene ordinato in senso ascendente e dunque l'indirizzo più basso conterrà l'elemento inferiore.

Una funzione correlata è bsearch().



## raise()

```
#include <csignal>
int raise(int signal);
```

La funzione raise() invia al programma in esecuzione il segnale specificato da *signal*. In caso di successo restituisce il valore 0 e in caso contrario restituisce un valore diverso da 0. La funzione utilizza l'header <csignal>.

Il C++ standard definisce i seguenti segnali. Naturalmente il compilatore è libero di fornire anche altri segnali.

| MACRO | SIGNIFICATO |
|---|---|
| SIGABRT | Errore terminazione |
| SIGFPE | Errore virgola mobile |
| SIGILL | Istruzione errata |
| SIGINT | L'utente ha premuto CTRL-C |
| SIGSEGV | Accesso illegale alla memoria |
| SIGTERM | Programma terminato |

Una funzione correlata è signal().

## rand()

```
#include <cstdlib>
int rand(void);
```

La funzione rand() genera una sequenza di numeri pseudocasuali. Ad ogni chiamata, rand() restituisce un intero compreso fra 0 e RAND_MAX.

Una funzione correlata è srand().

## setjmp()

```
#include <csetjmp>
int setjmp(jmp_buf envbuf);
```

La funzione setjmp() salva in *buf* il contenuto dello stack di sistema in modo che successivamente possa essere utilizzato da longjmp(). La funzione usa l'header <csetjmp>.

Alla chiamata la funzione setjmp() restituisce 0. Però durante l'esecuzione longjmp() passa un argomento a setjmp() ed è questo valore (sempre diverse da 0) che sarà il valore di setjmp() dopo una chiamata a longjmp(). Per informazioni consultare la descrizione della funzione longjmp().

Una funzione correlata è longjmp().

## signal()

```
#include <csignal>
void (*signal(int signal, void (*func)(int))) (int);
```

La funzione signal() registra la funzione puntata da *func* come gestore del segnale specificato da *signal*. Pertanto quando il programma riceverà il segnale *signal*, richiamerà la funzione puntata da *func*.

Il valore di *func* può essere l'indirizzo di una funzione di gestione dei segnali o una delle seguenti macro definite in <csignal>.

| MACRO | SIGNIFICATO |
|---|---|
| SIG_DFL | Usa il sistema di segnalazione standard |
| SIG_IGN | Ignora il segnale |

Se viene utilizzato per indirizzo di funzione, alla ricezione del segnale verrà eseguito il gestore specificato.

In caso di successo, signal() restituisce l'indirizzo della funzione definita in precedenza per il segnale specificato. In caso di errore viene restituito SIG_ERR (definito in <csignal>).

Una funzione correlata è raise().

## srand()

```
#include <cstdlib>
void srand(unsigned seed);
```

La funzione srand() definisce un punto iniziale per la sequenza generata da rand(), la funzione che restituisce numeri pseudocasuali.

srand() viene generalmente utilizzata per consentire a più programmi di utilizzare sequenze differenti di numeri pseudocasuali specificando punti di avvio differenti. Nel contempo si può anche usare srand() per generare ripetutamente la stessa sequenza di numeri pseudocasuali, richiamando lo stesso seme prima di iniziare la sequenza.



La funzione correlata è rand().

## strtod()

```
#include <cstdlib>
double strtod(const char *start, char **end);
```

La funzione strtod() converte una stringa che rappresenta un numero (*start*) in un double e quindi restituisce il risultato.

La funzione strtod() opera nel seguente modo. Innanzitutto salta eventuali spazi vuoti contenuti nella stringa puntata da *start*. Poi legge ogni carattere che costituisce il numero. La lettura dei caratteri termina quando viene rilevato un carattere che non può far parte di un numero in virgola mobile. Può trattarsi di uno spazio vuoto, di un segno di punteggiatura (a parte il punto) e di caratteri diversi da e o E. Al termine *end* punta all'eventuale parte rimanente della stringa originaria. Questo significa che se strtod() viene richiamata sul sulla stringa "100.00 elementi", restituirà il valore 100.00 mentre *end* punterà allo spazio che precede la parola "elementi".

Se non viene eseguita alcuna conversione, la funzione restituisce 0. Se si verifica un overflow, strtod() restituisce HUGE_VAL (o -HUGE_VAL per indicare un overflow negativo) e imposta a ERANGE la variabile globale errno per indicare un errore relativo all'intervallo. Se si verifica un underflow, strtod() restituisce il valore 0 e alla variabile globale errno viene assegnato il valore ERANGE.

Una funzione correlata è atof().

## strtol()

```
#include <cstdlib>
long strtol(const char *start, char **end,
            int radix);
```

La funzione strtol() converte una stringa che rappresenta un numero (*start*) in un long e quindi restituisce il risultato. La base del numero è determinata da *radix*. Se *radix* è zero, la base è determinata dalle regole che determinano le caratteristiche della costante.

La funzione strtol() opera nel seguente modo. Innanzitutto salta eventuali spazi vuoti contenuti nella stringa puntata da *start*. Poi legge ogni carattere che costituisce il numero. La lettura dei caratteri termina quando viene rilevato un carattere che non può far parte di un numero long. Può trattarsi di uno spazio vuoto, di un segno di punteggiatura e di caratteri. Al termine *end* punta all'eventuale parte rimanente della stringa originaria. Questo significa che se strtol() viene richiama-

ta sul sulla stringa "100.00 elementi", restituirà il valore 100L mentre *end* punterà allo spazio che precede la parola "elementi".

Se il risultato non è rappresentato da un intero long, la funzione restituisce LONG_MAX o LONG_MIN e imposta a ERANGE la variabile globale errno per indicare un errore relativo all'intervallo. Se non si verifica nessuna conversione strtol() restituisce il valore 0.

Una funzione correlata è atol().

## strtoul()

```
#include <cstdlib>
unsigned long strtoul(const char *start, char **end,
                      int radix);
```

La funzione strtoul() converte una stringa che rappresenta un numero (*start*) in un unsigned long e quindi restituisce il risultato. La base del numero è determinata da *radix*. Se *radix* è zero, la base è determinata dalle regole che determinano le caratteristiche della costante. Se radix viene specificata deve essere compresa fra 2 e 36.

La funzione strtoul() opera nel seguente modo. Innanzitutto salta eventuali spazi vuoti contenuti nella stringa puntata da *start*. Poi legge ogni carattere che costituisce il numero. La lettura dei caratteri termina quando viene rilevato un carattere che non può far parte di un numero unsigned long. Può trattarsi di uno spazio vuoto, di un segno di punteggiatura e di caratteri. Al termine *end* punta all'eventuale parte rimanente della stringa originaria. Questo significa che se strtoul() viene richiamata sul sulla stringa "100.00 elementi", restituirà il valore 100L mentre *end* punterà allo spazio che precede la parola "elementi".

Se il risultato non è rappresentato da un intero unsigned long, la funzione restituisce ULONG_MAX e imposta a ERANGE la variabile globale errno per indicare un errore relativo all'intervallo. Se non si verifica nessuna conversione strtol() restituisce il valore 0.

Una funzione correlata è strtol().

## system()

```
#include <cstdlib>
int system(const char *str);
```

La funzione system() invia la stringa puntata da *str* come comando per l'interprete dei comandi del sistema operativo.



Se system() viene richiamata con un puntatore nullo, allora restituisce un valore diverso da 0 se l'interprete dei comandi è presente e 0 altrimenti (alcuni programmi C++ vengono eseguiti in sistemi dedicati che non usano sistemi operativi ed interpreti di comandi dunque non si può sempre presumere che sia presente un interprete dei comandi). Il valore restituito da system() è definito dall'implementazione. Comunque quando il comando viene eseguito con successo la funzione restituisce generalmente il valore 0 mentre in tutti gli altri casi restituisce un valore diverso da 0.

Una funzione correlata è exit().

## va_arg(), va_start() e va_end()

```
#include <cstdarg>
type va_arg(va_list argptr, type);
void va_end(va_list argptr);
void va_start(va_list argptr, last_parm);
```

Le macro va_arg(), va_start() e va_end() collaborano per consentire il passaggio a una funzione di un numero variabile di argomenti. L'esempio più comune di una funzione che accetta un numero variabile di argomenti è printf(). Il tipo va_list è definito da <cstdarg>.

La procedura generale per creare una funzione che può accettare un numero variabile di argomenti è la seguente. La funzione deve avere almeno un parametro noto (ma può anche averne di più) il quale deve precedere la lista dei parametri variabili. L'ultimo parametro noto si chiama last_parm. Il nome di last_parm viene utilizzato come secondo parametro in una chiamata a va_start(). Prima di poter accedere ai parametri variabili, occorre inizializzare il puntatore agli argomenti *argptr* tramite una chiamata a va_start(). Successivamente i parametri vengono restituiti tramite chiamate a va_arg() dove *type* è il tipo del parametro successivo. Infine, una volta che tutti i parametri sono stati letti e prima di uscire dalla funzione, una chiamata a va_end() assicura che lo stack venga ripristinato in modo corretto. Se si dimentica di richiamare va_end() è probabile che si verifichi un blocco del programma.

Una funzione correlata è vprintf().

## wcstombs()

```
#include <cstdlib>
size_t wcstombs(char *out, const wchar_t *in, size_t size);
```

La funzione wcstombs() converte l'array di caratteri estesi puntato da *in* nel suo equivalente *multibyte* e inserisce il risultato nell'array puntato da *out*. Vengono convertiti solo i primi *size* byte di *in*. La conversione termina quando viene incontrato il terminatore nullo.

In caso di successo, wcstombs() restituisce il numero di byte convertiti. In caso di insuccesso restituisce -1.

Le funzioni correlate sono wctomb() e mbstowcs().

### wctomb()

```
#include <cstdlib>
int wctomb(char *out,  wchar_t in);
```

La funzione wctomb() converte il carattere esteso *in* nel suo equivalente *multibyte* e inserisce il risultato nell'oggetto puntato da *out*. L'array puntato da *out* deve essere lungo MB_CUR_MAX caratteri.

In caso di successo, wctomb() restituisce il numero di byte contenuti nel carattere *multibyte*. In caso di insuccesso restituisce -1.

Se *out* è nullo, allora wctomb() restituisce un valore diverso da 0 nel caso in cui il carattere *multibyte* abbia stabilito delle dipendenze e 0 in caso contrario.

Le funzioni correlate sono wcstombs() e mbtowc().

Capitolo 31

# Le funzioni per caratteri estesi


31.1 **Le funzioni di classificazione per caratteri estesi**

31.2 **Le funzioni di I/O per caratteri estesi**

31.3 **Funzioni per stringhe di caratteri estesi**

31.4 **Funzioni di conversione per stringhe di caratteri estesi**

31.5 **Funzioni per array di caratteri estesi**

31.6 **Funzioni per la conversione di caratteri multibyte ed estesi**


Nel 1995, al C standard è stata aggiunta una serie di funzioni per caratteri estesi che successivamente sono state adottate dal C++ standard. Le funzioni per caratteri estesi operano su caratteri di tipo wchar_t che occupano 16 bit. Queste funzioni sono in genere analoghe alle equivalenti funzioni per char. Ad esempio, la funzione iswspace() è la versione a carattere estesi di isspace(). In generale le funzioni per caratteri estesi usano lo stesso nome delle equivalenti funzioni per char, con l'aggiunta della lettera "w".

Le funzioni per caratteri estesi usano due header: <cwchar> e <cwctype>. Sono supportati anche il file header C <wchar.h> e <wctype.h>.

L'header <cwctype> definisce i tipi wint_t, wctrans_t e wctype_t. Molte delle funzioni per caratteri estesi ricevono come parametro un carattere esteso. Il tipo di questo parametro è wint_t. Tale parametro è in grado di contenere un carattere esteso. L'uso del tipo wint_t nelle funzioni per caratteri estesi è simile all'uso di int nelle funzioni per char. I tipi wctrans_t e wctype_t sono dedicati a oggetti utilizzati rispettivamente per rappresentare un mappaggio dei caratteri (ovvero una traduzione) e la classificazione di un carattere. L'indicatore di fine file per caratteri estesi è definito come WEOF.

Oltre a definire win_t, l'header <cwchar> definisce il tipo mstate_t che descrive un oggetto che contiene lo stato di una conversione da multibyte a caratteri estesi. L'header <cwchar> definisce anche le macro NULL, WEIF, WCHAR_MAX e WCHAR_MIN. Le ultime due macro definiscono il valore massimo e minimo che può essere contenuto in un oggetto di tipo wchar_t.

Anche se il supporto che la libreria delle funzioni standard offre per i caratteri estesi è piuttosto ampio, in realtà queste funzioni vengono utilizzate raramente.

Un motivo è il fatto che il sistema di I/O e le librerie di classi del C++ standard forniscono un supporto per caratteri normali ed estesi tramite l'impiego di classi template. Inoltre, non vi è un grande interesse nella realizzazione di programmi che impiegano caratteri estesi. Naturalmente la situazione potrebbe cambiare nel futuro.

Poiché la maggior parte delle funzioni per caratteri estesi hanno un equivalente per char e per il fatto che non sono molto utilizzate dai programmatori C++, ne verrà fornita solo una breve introduzione.

## 31.1 Le funzioni di classificazione per caratteri estesi

L'header <cwctype> fornisce i prototipi per le funzioni per caratteri estesi che supportano la classificazione dei caratteri. Queste funzioni suddividono in categorie i caratteri estesi oppure convertono i caratteri da lettere minuscole a maiuscole e viceversa. La Tabella 31.1 elenca queste funzioni insieme alle corrispondenti funzioni per char descritte nel Capitolo 26.

**Tabella 31.1** Le funzioni di classificazione per caratteri estesi.

| FUNZIONE | EQUIVALENTE PER CHAR |
|---|---|
| `int iswalnum(wint_t ch)` | `isalnum()` |
| `int iswalpha(wint_t ch)` | `isalpha()` |
| `int iswcntrl(wint_t ch)` | `iscntrl()` |
| `int iswdigit(wint_t ch)` | `isdigit()` |
| `int iswgraph(wint_t ch)` | `isgraph()` |
| `int iswlower(wint_t ch)` | `islower()` |
| `int iswprint(wint_t ch)` | `isprint()` |
| `int iswpunct(wint_t c)` | `ispunct()` |
| `int iswspace(wint_t ch)` | `isspace()` |
| `int iswupper(wint_t ch)` | `isupper()` |
| `int iswxdigit(wint_t ch)` | `isxdigit()` |
| `wint_t tolower(wint_t ch)` | `tolower()` |
| `wint_t toupper(wint_t ch)` | `toupper()` |



Oltre alle funzioni elencate nella Tabella 31.1, <cwctype> definisce le seguenti funzioni che forniscono un metodo aperto per la classificazione dei caratteri.

```
wctype_t wctype(const char *attr);
int iswctype(wint_t ch, wctype_t attr_ob);
```

La funzione wctype() restituisce un valore che può essere passato al parametro *attr_ob* di iswctype(). La stringa puntata da *attr* specifica una proprietà che deve essere in possesso del carattere. Il valore di *attr_ob* viene utilizzato per determinare se *ch* è un carattere che ha tale proprietà. In caso affermativo, iswctype() restituisce un valore diverso da 0, altrimenti restituisce 0. Le stringhe di proprietà della tabella seguente sono definite per tutti gli ambienti operativi.

| | | | |
|---|---|---|---|
| alnum | alpha | cntrl | digit |
| graph | lower | print | punct |
| space | upper | xdigit | |

Il seguente programma illustra l'uso delle funzioni wctype() e iswctype().

```
#include <iostream>
#include <cwctype>
using namespace std;

int main()
{
  wctype_t x;

  x = wctype("spazio");

  if(iswctype(L' ', x))
    cout << " È uno spazio.\n";

  return 0;
}
```

Questo programma visualizza la stringa "È uno spazio.".

<cwctype> definisce anche le funzioni wctrans() e towctrans():

```
wctrans_t wctrans(const char *mapping);
wint_t towctrans(wint_t ch, wctrans_t mapping_ob);
```

La funzione wctrans() restituisce un valore che può essere passato al parametro mapping_ob di towctrans(). Qui la stringa puntata da *mapping* specifica un

mappaggio di un carattere in un altro. Questo valore può essere utilizzato da iswctrans() per mappare *ch*. Quindi viene restituito il valore mappato. Le seguenti stringhe di mappaggio sono supportate da tutti gli ambienti di esecuzione.

tolower toupper

Ecco un breve esempio che illustra l'uso di wctrans() e towctrans().

```
#include <iostream>
#include <cwctype>
using namespace std;

int main()
{
  wctrans_t x;

  x = wctrans("tolower");

  wchar_t ch = towctrans(L'W', x);
  cout << (char) ch;

  return 0;
}
```

Questo programma visualizza la minuscola "w".

## 31.2 Le funzioni di I/O per caratteri estesi

Molte delle funzioni di I/O descritte nel Capitolo 25 hanno anche un'implementazione per caratteri estesi. Tali unioni sono elencate nella Tabella 31.2. Le funzioni di I/O per caratteri estesi usano l'header <cwchar>. Si noti che swprintf() e wswprintf() richiedono un parametro aggiuntivo non richiesto dalle equivalenti funzioni per char.

Oltre alle funzioni elencate in questa tabella, è stata aggiunta la seguente funzione di I/O specifica per caratteri estesi:

```
int fwide(FILE *stream, int how);
```

Se *how* è positiva, fwide() lo trasforma in uno stream di caratteri estesi. Se *how* è negativa, fwide() lo trasforma in uno stream di char. Se *how* è uguale a zero, lo stream non viene alterato. Se lo stream ha già le caratteristiche desiderate, non viene alterato. La funzione restituisce un valore positivo se lo stream usa caratteri estesi, negativo se usa char e 0 in caso di errore. Le caratteristiche di uno stream sono determinate dal suo primo uso.



**Tabella 31.2** Le funzioni di I/O per caratteri estesi.

| FUNZIONE | CHAR EQUIVALENTE |
|---|---|
| `win_t fgetwc(FILE *stream)` | `fgetc()` |
| `wchar_t *fgetws(wchar_t *str, int num, FILE *stream)` | `fgets()` |
| `wint_t fputwc(wchar_t ch, FILE *stream)` | `fputc()` |
| `int fputws(const wchar_t *str, FILE *stream)` | `fputs()` |
| `int fwprintf(FILE *stream, const wchar_t fmt, ...)` | `fprintf()` |
| `int fwscanf(FILE *stream, const wchar_t fmt, ...)` | `fscanf()` |
| `wint_t getwc(FILE *stream)` | `getc()` |
| `wint_t getwchar( )` | `getchar()` |
| `wint_t putwc(wchar_t ch, FILE *stream)` | `putc()` |
| `wint_t putwchar(wchar_t ch)` | `putchar()` |
| `int swprintf(wchar_t *str, size_t num, const wchar_t *fmt, ...)` | `sprintf()` Si noti l'aggiunta del parametro *num*, che limita il numero di caratteri scritti su *str*. |
| `int swscanf(const wchar_t *str, const wchar_t *fmt, ...)` | `sscanf()` |
| `wint_t ungetwc(wint_t ch, FILE *stream)` | `ungetc()` |
| `int vfwprintf(FILE *stream,const wchar_t fmt, va_list arg)` | `vfprintf()` |
| `int vswprintf(wchar_t *str, size_t num, const wchar_t *fmt, va_list arg)` | `vsprintf()` Si noti l'aggiunta del parametro *num*, che limita il numero di caratteri scritti su *str*. |
| `int vwprintf(const wchar_t *fmt, va_list arg)` | `vprintf()` |
| `int wprintf(const wchar_t *fmt, ...)` | `printf()` |
| `int wscanf(const wchar_t *fmt, ...)` | `scanf()` |

## 31.3 Funzioni per stringhe di caratteri estesi

L'header <cwchar> definisce anche le versioni a caratteri estesi delle funzioni per la manipolazione delle stringhe descritte nel Capitolo 27. Si noti che wcstok() richiede un parametro in più rispetto alla funzione char corrispondente.

**Tabella 31.3** Funzioni per stringhe di caratteri estesi.

| FUNZIONE | CHAR EQUIVALENTE |
|---|---|
| `wchar_t *wcscat(wchar_t *str1, const wchar_t *str2)` | `strcat()` |
| `wchar_t *wcschr(const wchar_t *str, wchar_t ch)` | `strchr()` |
| `int wcscmp(const wchar_t *str1, const wchar_t *str2)` | `strcmp()` |
| `int wcscoll(const wchar_t *str1, const wchar_t *str2)` | `strcoll()` |
| `size_t wcscspn(const wchar_t *str1, const wchar_t *str2)` | `strcspn()` |
| `wchar_t *wcscpy(wchar_t *str1, const wchar_t *str2)` | `strcpy()` |
| `size_t wcslen(const wchar_t *str)` | `strlen()` |
| `wchar_t *wcsncpy(wchar_t *str1, const wchar_t str2, size_t num)` | `strncpy()` |
| `wchar_t *wcsncat(wchar_t *str1, const wchar_t str2, size_t num)` | `strncat()` |
| `int wcsncmp(const wchar_t *str1,const wchar_t *str2, size_t num)` | `strncmp()` |
| `wchar_t *wcspbrk(const wchar_t *str1,const wchar_t *str2)` | `strpbrk()` |
| `wchar_t *wcsrchr(const wchar_t *str, wchar_t ch)` | `strrchr()` |
| `size_t wcsspn(const wchar_t *str1,const wchar_t str2)` | `strspn()` |
| `wchar_t *wcstok(wchar_t *str1, const wchar_t *str2, wchar_t **endptr)` | `strtok()` Qui, *endptr* è un puntatore che contiene le informazioni necessarie per continuare il processo di estrazione dei token. |
| `wchar_t *wcsstr(const wchar_t *str1, const wchar_t *str2)` | `strstr()` |
| `size_t wcsxfrm(wchar_t *str1, const wchar_t *str2,size_t num)` | `strxfrm()` |



## 31.4 Funzioni di conversione per stringhe di caratteri estesi

Le funzioni elencate nella Tabella 31.4 costituiscono le versioni per caratteri estesi delle normali funzioni per la conversione di valori numerici e temporali. Queste funzioni usano l'header <cwchar>.

**Tabella 31.4** Le funzione di conversione per caratteri estesi.

| FUNZIONE | CHAR EQUIVALENTE |
|---|---|
| `size_t wcsftime(wchar_t *str, size_t max, const wchar_t *fmt, const struct tm *ptr)` | `strftime()` |
| `double wcstod(const wchar_t *start, wchar_t **end);` | `strtod()` |
| `long wcstol(const wchar_t *start, wchar_t **end, int radix)` | `strtol()` |
| `unsigned long wcstoul(const wchar_t *start, wchar_t **end, int radix)` | `strtoul()` |

## 31.5 Funzioni per array di caratteri estesi

Le funzioni per la manipolazione di array come memcpy() hanno una funzione equivalente per caratteri estesi. Tali funzioni sono elencate nella Tabella 31.5. Questo gruppo di funzioni usa l'header <cwchar>.

**Tabella 31.5** Le funzioni per array di caratteri estesi.

| FUNZIONE | CHAR EQUIVALENTE |
|---|---|
| `wchar_t *wmemchr(const wchar_t *str, wchar_t ch, size_t num)` | `memchr()` |
| `int wmemcmp(const wchar_t *str1,const wchar_t *str2, size_t num)` | `memcmp()` |
| `wchar_t *wmemcpy(wchar_t *str1, const wchar_t *str2, size_t num)` | `memcpy()` |
| `wchar_t *wmemmove(wchar_t *str1, const wchar_t *str2, size_t num)` | `memmove()` |
| `wchar_t *wmemset(wchar_t *str, wchar_t ch, size_t num)` | `memset()` |

## 31.6 Funzioni per la conversione di caratteri multibyte ed estesi

La libreria delle funzioni standard C++ fornisce varie funzioni che supportano la conversione fra *multibyte* e caratteri estesi. Queste funzioni, elencate nella Tabella 31.6 usano l'header <cwchar>. Molte di esse sono le versioni riavviabili delle normali funzioni per *multibyte*. La versione riavviabile utilizza le informazioni di stato passate in un parametro di tipo mbstate_t. Se il parametro è nullo, la funzione fornisce un proprio oggetto mbstate_t.

**Tabella 31.6** Funzioni di conversione fra caratteri estesi e multibyte.

| FUNZIONE | DESCRIZIONE |
|---|---|
| `win_t btowc(int ch)` | Converte *ch* nell'equivalente a caratteri estesi e restituisce il risultato. In caso di errore o se *ch* non è un carattere *multibyte* di un solo byte, restituisce `WEOF`. |
| `size_t mbrlen(const char *str, size_t num, mbstate_t *state)` | Versione riavviabile di `mblen()`, come descritto da *state*. Restituisce un valore positivo che indica la lunghezza del successivo carattere *multibyte*. Se il carattere successivo è nullo, restituisce 0. In caso di errore restituisce un valore negativo. |
| `size_t mbrtowc(wchar_t *out, const char *in, size_t num, mbstate_t *state)` | Versione riavviabile di `mbtowc()` come descritto da *state*. Restituisce un valore positivo che indica la lunghezza del successivo carattere *multibyte*. Restituisce 0 se il carattere successivo è nullo. In caso di errore restituisce un valore negativo. Se si verifica un errore, a `errno` viene assegnata la macro `EILSEQ`. |
| `int mbsinit(const mbstate_t *state)` | Restituisce `true` se *state* rappresenta uno stato iniziale della conversione. |
| `size_t mbsrtowcs(wchar_t *out, const char **in, size_t num, mbstate_t state)` | Versione riavviabile di `mbstowcs()` come descritto da *state*. Inoltre `mbsrtowcs()` differisce da `mbstowcs()` per il fatto che *in* è un puntatore indiretto all'array di origine. Se si verifica un errore, a `errno` viene assegnata la macro `EILSEQ`. |
| `size_t wcrtomb(char *out, wchar_t ch, mbstate_t *state)` | Versione riavviabile di `wctomb()` come descritto da *state*. Se si verifica un errore, a `errno` viene assegnata la macro `EILSEQ`. |
| `size_t wcsrtombs(char *out, const wchar_t **in, size_t num, mbstate_t *state)` | Versione riavviabile di `wcstombs()` come descritto da state. Inoltre `wcsrtombs()` differisce da `wcstombs()` per il fatto che *in* è un puntatore indiretto all'array di origine. Se si verifica un errore, a `errno` viene assegnata la macro `EILSEQ`. |
| `int wctob(wint_t ch)` | Converte *ch* nel suo equivalente *multibyte* a un solo byte. In caso di errore restituisce `EOF`. |

Parte quarta

# LA LIBRERIA DI CLASSI STANDARD DEL C++

Lo standard per il C++ definisce un insieme esteso di classi che offre il supporto per un gran numero di attività molto comuni, come le operazioni di I/O, la manipolazione delle stringhe e l'elaborazione numerica. La libreria delle classi si aggiunge alla libreria delle funzioni descritta nella Parte terza. La libreria delle classi costituisce una parte fondamentale del linguaggio C++ e definisce in larga misura il suo carattere. Nonostante le sue dimensioni, la libreria delle classi è facile da impiegare in quanto è organizzata attorno a principi di programmazione a oggetti.

La libreria del C++ standard è piuttosto estesa e una descrizione approfondita di tutte le sue classi, funzionalità, attributi e dettagli implementativi non rientra negli scopi di questo volume (una descrizione completa della libreria di classi occuperebbe da sola un intero volume). Tuttavia, anche se la maggior parte delle classi è di utilizzo generale, alcune di esse sono dedicate principalmente agli sviluppatori di compilatori o ai programmatori che implementano estensioni di varia natura. Pertanto questa sezione descrive le sole parti della libreria che vengono normalmente utilizzate in un'applicazione. Chi utilizza la libreria per eseguire operazioni molto specializzate potrà acquisire una copia dello standard del linguaggio C++ che contiene la descrizione tecnica dell'intera libreria di classi.

Capitolo 32

# Le classi di I/O del C++ standard


32.1 **Le classi di I/O**

32.2 **Gli header di I/O**

32.3 **I flag di formattazione e i manipolatori di I/O**

32.4 **I tipi del sistema di I/O del C++ standard**

32.5 **Overloading degli operatori < e >**

32.6 **Le funzioni di I/O di utilizzo generale**


Questo capitolo descrive la libreria di classi di I/O del C++ standard. Come si è detto nella Parte seconda, vengono comunemente utilizzate due versioni della libreria di I/O del C++. La prima è la libreria "vecchio stile" che non è definita dallo standard. La seconda è rappresentata dal sistema di I/O standard C++ a template. Poiché la nuova libreria di I/O è fondamentalmente un sovrainsieme della precedente, in questo capitolo si parlerà unicamente delle sue nuove funzionalità.

**NOTA** *Per una panoramica delle funzionalità di I/O del C++, consultare i Capitoli 20 e 21.*

## 32.1 Le classi di I/O

Il sistema di I/O del C++ standard si basa su una gerarchia piuttosto complessa di classi template. Ecco le classi impiegate:

| CLASSE | SCOPO |
|---|---|
| `basic_ios` | Fornisce operazioni di I/O di carattere generale |
| `basic_streambuf` | Supporto di basso livello per le operazioni di I/O |
| `basic_istream` | Supporto per le operazioni di input |

*(segue)*

*(continua)*

| CLASSE | SCOPO |
|---|---|
| basic_ostream | Supporto per le operazioni di input |
| basic_iostream | Supporto per le operazioni di input |
| basic_filebuf | Supporto di basso livello per le operazioni di I/O su file |
| basic_ifstream | Supporto per le operazioni di input da file |
| basic_ofstream | Supporto per le operazioni di output su file |
| basic_fstream | Supporto per le operazioni di input/output su file |
| basic_stringbuf | Supporto di basso livello per le operazioni di I/O su stringhe |
| basic_istringstream | Supporto per le operazioni di input da stringhe |
| basic_ostringstream | Supporto per le operazioni di output su stringhe |
| basic_stringstream | Supporto per le operazioni di input/output su stringhe |

All'interno di questa gerarchia di classi di I/O si situa anche la classe non-template ios_base. Questa classe fornisce le definizioni di vari elementi del sistema di I/O.

Il sistema di I/O del linguaggio C++ si basa su due gerarchie di classi template correlate ma differenti. La prima deriva dalla classe di I/O a basso livello chiamata basic_streambuf. Questa classe fornisce le operazioni di input e output a basso livello e fornisce il supporto di base per l'intero sistema di I/O. Le classi basic_filebuf e basic_stringbuf derivano entrambe da basic_streambuf. Se non si devono eseguire operazioni di I/O molto avanzate, è probabile che non si debba usare direttamente basic_streambuf o le sue sottoclassi.

La gerarchia di classi che viene normalmente utilizzata deriva da basic_ios. Questa è una classe di I/O di alto livello che fornisce alle operazioni di I/O su stream le funzionalità di formattazione, di controllo degli errori e le informazioni sullo stato. La classe basic_ios viene utilizzata come base per varie classi derivate, fra le quali basic_istream, basic_ostream e basic_iostream. Queste classi sono utilizzate rispettivamente per creare stream per operazioni di input, output e input/output. In particolare, da basic_istream derivano le classi basic_ifstream e basic_istringstream, da basic_ostream derivano le classi basic_ofstream e basic_ostringstream e da basic_iostream derivano le classi basic_fstream e basic_stringstream. Una classe base di basic_ios è ios_base. Pertanto ogni classe derivata da basic_ios ha accesso anche ai membri di ios_base.

Le classi di I/O sono parametrizzate per il tipo di caratteri su cui operano e per le caratteristiche associate a tali caratteri. Ad esempio ecco la specifica template per basic_ios:



```
template <class CharType, class Attr = char_traits<CharType>>
  class basic_ios: public ios_base
```

Qui CharType specifica il tipo dei caratteri (ad esempio char o wchar_t) e Attr specifica un tipo che descrive i suoi attributi. Il tipo generico char_traits è una classe di servizio che definisce gli attributi associati a un carattere.

Come si è detto nel Capitolo 20, la libreria di I/O crea due specializzazioni delle gerarchie di classi template appena descritte: una per caratteri a 8 bit e una per caratteri estesi. Ecco l'elenco completo delle corrispondenze fra classi template, per caratteri e per caratteri estesi.

| CLASSE TEMPLATE | CLASSE PER CARATTERI | CLASSE PER CARATTERI ESTESI |
|---|---|---|
| basic_ios | ios | wios |
| basic_istream | istream | wistream |
| basic_ostream | ostream | wostream |
| basic_iostream | iostream | wiostream |
| basic_ifstream | ifstream | wifstream |
| basic_ofstream | ofstream | wofstream |
| basic_fstream | fstream | wfstream |
| basic_istringstream | istringstream | wistringstream |
| basic_ostringstream | ostringstream | wostringstream |
| basic_stringstream | stringstream | wstringstream |
| basic_streambuf | streambuf | wstreambuf |
| basic_filebuf | filebuf | wfilebuf |
| basic_stringbuf | stringbuf | wstringbuf |

Poiché la maggior parte dei programmatori utilizza operazioni di I/O a caratteri, questi saranno i nomi più utilizzati in questo capitolo. Pertanto quando si fa riferimento alle classi di I/O si indicherà semplicemente il nome della classe per caratteri piuttosto che il nome template interno. Ad esempio, in questo capitolo si userà il nome ios invece di basic_ios, istream invece di basic_istream e fstream invece di basic_fstream. Si ricordi sempre che esistono funzioni parallele per stream di caratteri estesi che funzionano esattamente come quelle che verranno descritte.

## 32.2 Gli header di I/O

Il sistema di I/O del C++ standard utilizza vari header:

| HEADER | UTILIZZO |
|---|---|
| <fstream> | I/O su file |
| <iomanip> | Manipolatori di I/O parametrizzati |
| <ios> | Supporto di base per le operazioni di I/O |
| <iosfwd> | Dichiarazioni forward usate dal sistema di I/O |
| <iostream> | I/O generale |
| <istream> | Supporto di base per l'input |
| <ostream> | Supporto di base per l'output |
| <sstream> | Stream basati su stringhe |
| <streambuf> | Supporto di I/O a basso livello |

Molti di questi header sono utilizzati internamente dal sistema di I/O. In genere il programma include i soli header <iostream>, <fstream>, <sstream> o <iomanip>.

## 32.3 I flag di formattazione e i manipolatori di I/O

A ogni stream è associato un insieme di flag di formattazione che controllano il modo in cui le informazioni vengono formattate. La classe ios_base dichiara un'enumerazione bitmask chiamata fmtflags in cui sono definiti i seguenti valori:

| | | | |
|---|---|---|---|
| adjustfield | basefield | boolalpha | dec |
| fixed | floatfield | hex | internal |
| left | oct | right | scientific |
| showbase | showpoint | showpos | skipws |
| unitbuf | uppercase | | |

Questi valori sono utilizzati per impostare o cancellare i flag di formattazione utilizzando funzioni come setf() e unsetf(). Per una descrizione dettagliata di questi flag si consulti il Capitolo 20.

Oltre a impostare o cancellare direttamente i flag di formattazione è anche possibile modificare i parametri di formattazione di uno stream impiegando apposite funzioni chiamate manipolatori che possono essere incluse in un'espressione di I/O. Questi manipolatori standard sono elencati nella seguente tabella:



| MANIPOLATORE | SCOPO | INPUT/OUTPUT |
|---|---|---|
| boolalpha | Attiva il flag boolapha. | Input/Output |
| dec | Attiva il flag dec. | Input/Output |
| endl | Produce in output un carattere di fine riga e svuota lo stream. | Output |
| ends | Produce in output un carattere nullo. | Output |
| fixed | Attiva il flag fixed. | Output |
| flush | Svuota uno stream. | Output |
| hex | Attiva il flag hex. | Input/Output |
| internal | Attiva il flag internal. | Output |
| left | Attiva il flag left. | Output |
| noboolalpha | Disattiva il flag boolalpha. | Input/Output |
| noshowbase | Disattiva il flag showbase. | Output |
| noshowpoint | Disattiva il flag showpoint. | Output |
| noshowpos | Disattiva il flag showpos. | Output |
| noskipws | Disattiva il flag skipws. | Input |
| nounitbuf | Disattiva il flag unitbuf. | Output |
| nouppercase | Disattiva il flag uppercase. | Output |
| oct | Attiva il flag oct. | Input/Output |
| resetiosflags (fmtflags f) | Disattiva i flag specificati in f. | Input/Output |
| right | Attiva il flag right. | Output |
| scientific | Attiva il flag scientific. | Output |
| setbase(int base) | Imposta la base numerica. | Input/Output |
| setfill(int ch) | Imposta il carattere di riempimento. | Output |
| setiosflags(fmtflags f) | Attiva i flag specificati in f. | Input/output |
| setprecision (int p) | Imposta le cifre di precisione. | Output |
| setw(int w) | Imposta la larghezza del campo. | Output |

(segue

*(continua)*

| MANIPOLATORE | SCOPO | INPUT/OUTPUT |
|---|---|---|
| showbase | Attiva il flag showbase. | Output |
| showpoint | Attiva il flag showpoint. | Output |
| showpos | Attiva il flag showpos. | Output |
| skipws | Attiva il flag skipws. | Input |
| unitbuf | Attiva il flag unitbuf. | Output |
| uppercase | Attiva il flag uppercase. | Output |
| ws | Salta gli spazi iniziali. | Input |

Per utilizzare un manipolatore dotato di parametri, si deve includere <iomanip>.

## 32.4 I tipi del sistema di I/O del C++ standard

Oltre al tipo fmtflags appena descritto, il sistema di I/O del C++ standard definisce molti altri tipi.

### I tipi streamsize e streamoff

Un oggetto di tipo streamsize è in grado di contenere il massimo numero di byte che verranno trasferiti in un'operazione di I/O. In genere si tratta di una qualche forma di intero. Un oggetto di tipo streamoff è in grado di contenere un valore che indica una posizione di offset all'interno di uno stream. In genere si tratta di una qualche forma di intero. Questi tipi sono definiti nell'header <ios> che viene incluso automaticamente dal sistema di I/O.

### I tipi streampos e wstreampos

Un oggetto di tipo streampos è in grado di contenere un valore che rappresenta una posizione in uno stream di char. Il tipo wstreampos è in grado di contenere un valore che rappresenta una posizione in uno stream di wchar_t. Questi tipi sono definiti nell'header <iosfwd> che viene incluso automaticamente dal sistema di I/O.



### I tipi pos_type e off_type

I tipi pos_type e off_type creano oggetti (normalmente interi) in grado di contenere valori che rappresentano rispettivamente la posizione e l'offset all'interno di uno stream. Questi tipi sono definiti da ios (e altre classi) e sono in genere dello stesso tipo di streamoff o streampos (o dei loro equivalenti per caratteri estesi).

### Il tipo openmode

Il tipo openmode è definito da ios_base e descrive il modo in cui viene aperto un file. Può assumere uno o più dei seguenti valori.

| | |
|---|---|
| app | Aggiunta alla fine del file. |
| ate | Posizionamento alla fine del file per operazioni di creazione. |
| binary | Apertura del file per operazioni binarie. |
| in | Apertura del file in input. |
| out | Apertura del file in output. |
| trunc | Cancellazione di un file. |

Questi valori possono essere combinati con l'operatore OR.

### Il tipo iostate

Un oggetto di tipo iostate descriverà lo stato di uno stream di I/O; si tratta di un'enumerazione definita da ios_base che include i seguenti membri.

| NOME | SIGNIFICATO |
|---|---|
| goodbit | Nessun errore. |
| eofbit | Raggiunta la fine del file. |
| failbit | Errore di I/O non fatale. |
| badbit | Errore di I/O fatale. |

### Il tipo seekdir

Il tipo seekdir descrive il modo in cui verrà eseguita un'operazione di accesso diretto al file. Il tipo seekdir è definito in ios_base e i valori consentiti sono:

| | |
|---|---|
| beg | Inizio del file. |
| cur | Posizione corrente. |
| end | Fine del file. |

### La classe failure

La classe ios_base definisce il tipo per eccezioni failure che funge da classe base per tutte le eccezioni che possono essere lanciate dal sistema di I/O. Tale classe eredita exception (la classe standard per le eccezioni). La classe failure ha il seguente costruttore:

```
explicit failure(const string &str);
```

Qui *str* è un messaggio che descrive l'errore. Questo messaggio può essere ottenuto da un oggetto failure richiamando la sua funzione what():

```
virtual const char *what() const throw();
```

## 32.5 Overloading degli operatori < e >

Gli operatori < e/o > sono modificati tramite overloading nelle seguenti classi rispetto a tutti i tipi di dati standard.

```
basic_istream
basic_ostream
basic_iostream
```

Dunque ogni classe derivata da quelle elencate erediterà questi operatori.

## 32.6 Le funzioni di I/O di utilizzo generale

La parte rimanente di questo capitolo descrive le funzioni di I/O di utilizzo generale fornite dal C++ standard. Come si è detto, il sistema di I/O del C++ standard si basa su una complessa gerarchia di classi template. Molti dei membri delle classi di basso livello non sono utilizzati nella programmazione di applicazioni e pertanto non verranno descritti.



### bad()

```
#include <iostream>
bool bad() const;
```

La funzione bad() è un membro di ios.
La funzione bad() restituisce true quando nello stream associato si è verificato un errore di I/O fatale, altrimenti restituisce false.
La funzione correlata è rdstate().

### clear()

```
#include <iostream>
void clear(iostate flags = goodbit);
```

La funzione clear() è un membro di ios.
La funzione clear() cancella i flag di stato associati a uno stream. Se *flags* è goodbit (come nell'impostazione standard) allora tutti i flag di errore vengono cancellati (riportati a 0). Altrimenti ai flag di stato viene assegnato il valore specificato in flags.
La funzione correlata è rdstate().

### eof()

```
#include <iostream>
bool eof() const;
```

La funzione eof() è un membro di ios.
La funzione eof() restituisce true quando si raggiunge la fine del file di input, altrimenti restituisce false.
Le funzioni correlate sono bad(), fail(), good(), rdstate() e clear().

### exceptions()

```
#include <iostream>
iostate exceptions() const;
void exceptions(iostate flags);
```

La funzione exceptions() è un membro di ios.
La prima forma restituisce un oggetto iostate che indica quali flag hanno provocato l'eccezione. La seconda forma imposta questi valori.

La funzione correlata è rdstate().

## fail()

```
#include <iostream>
bool fail() const;
```

La funzione fail() è un membro di ios.

La funzione fail() restituisce true se sullo stream si è verificato un errore di I/O, altrimenti restituisce false.

Le funzioni correlate sono good(), eof(), bad(), clear() e rdstate().

## fill()

```
#include <iostream>
char fill() const;
char fill(char ch);
```

La funzione fill() è un membro di ios.

Normalmente quando si deve riempire un campo, viene utilizzato come carattere di riempimento lo spazio. La funzione fill() consente di specificare un altro carattere di riempimento. La funzione restituisce il carattere di riempimento precedente.

Per ottenere il carattere di riempimento corrente basta usare la prima forma di fill().

Le funzioni correlate sono precision() e width().

## flags()

```
#include <iostream>
fmtflags flags() const;
fmtflags flags(fmtflags f);
```

La funzione flags() è un membro di ios (ereditato da ios_base).

La prima forma di flags() restituisce semplicemente i flag di formattazione attualmente associati allo stream.

La seconda forma di flags() imposta tutti i flag di formattazione associati a uno stream secondo quanto specificato da *f*. Quando si usa questa versione, il modello di bit presente in *f* viene copiato nei flag di formattazione associati allo stream. Anche questa versione restituisce l'impostazione precedente.

Le funzioni correlate sono unsetf() e setf().



## flush()

```
#include <iostream>
ostream &flush();
```

La funzione flush() è un membro di ostream.

La funzione flush() provoca la scrittura fisica del buffer connesso allo stream di output. La funzione restituisce l'indirizzo dello stream.

Le funzioni correlate sono put() e write().

## fstream(), ifstream() e ofstream()

```
#include <fstream>
fstream();
explicit fstream(const char *filename,
                 ios::openmode mode = ios::in | ios::out);

ifstream();
explicit ifstream(const char *filename, ios::openmode mode=ios::in);

ofstream();
explicit ofstream(const char *filename,
                  ios::openmode mode=ios::out | ios::trunc);
```

Le funzioni fstream(), ifstream() e ofstream() sono i costruttori delle classi fstream, ifstream e ofstream.

Le versioni di fstream(), ifstream() e ofstream() che non richiedono parametri creano uno stream che non è associato ad alcun file. Questo stream può successivamente essere collegato a un file con open().

Le versioni di fstream(), ifstream() e ofstream() che accettano come primo parametro il nome di un file sono quelle più utilizzate nei programmi applicativi. Anche se è corretto aprire un file utilizzando la funzione open(), in genere si preferisce usare i costruttori di fstream, ifstream e ofstream che aprono automaticamente il file al momento in cui viene creato lo stream. Le funzioni costruttore hanno gli stessi parametri e le stesse impostazioni standard della funzione open() (per informazioni si consulti alla parte relativa a tale funzione). Ad esempio, ecco il modo in cui viene normalmente aperto un file:

```
ifstream mystream("myfile");
```

Se per qualsiasi motivo il file non potesse essere aperto, il valore della variabile associata allo stream sarà false. Pertanto sia che si apra il file con la funzione costruttore sia che si usi una chiamata esplicita a open(), sarà necessario confer-

mare che il file sia stato effettivamente aperto controllando il valore dello stream.

Le funzioni correlate sono close() e open().

## gcount()

```
#include <iostream>
streamsize gcount() const;
```

La funzione gcount() è un membro di istream.

La funzione gcount() restituisce il numero di caratteri letti dall'ultima operazioni di input.

Le funzioni correlate sono get(), getline() e read().

## get()

```
#include <iostream>
int get();
istream &get(char &ch);
istream &get(char *buf, streamsize num);
istream &get(char *buf, streamsize num, char delim);
istream &get(streambuf &buf);
istream &get(streambuf &buf, char delim);
```

La funzione get() è un membro di istream.

In generale, get() legge i caratteri da uno stream di input. La forma senza parametri di get() legge dallo stream un singolo carattere e ne restituisce il valore.

get(char &ch) legge dallo stream un carattere e ne inserisce il valore in *ch*. Quindi restituisce l'indirizzo dello stream.

get(char *buf, streamsize *num*) legge dei caratteri nell'array puntato da *buf* finché non vengono letti *num* - 1 caratteri, finché non viene trovato un codice di fine riga o fino alla fine del file. L'array puntato da *buf* verrà chiuso dal carattere nullo. Se nella stringa viene rilevato il carattere di fine riga, questo non verrà estratto. Pertanto rimarrà nello stream fino all'operazione di input successiva. La funzione restituisce l'indirizzo dello stream.

get(char *buf, streamsize *num*, char *delim*) legge dei caratteri nell'array puntato da *buf* finché non vengono letti *num* - 1 caratteri, finché non viene incontrato il delimitatore *delim* o fino alla fine del file. L'array puntato da *buf* verrà chiuso dal carattere nullo. Se nella stringa viene rilevato il delimitatore, questo non verrà estratto e rimarrà nello stream fino all'operazione di input successiva. La funzione restituisce l'indirizzo dello stream.



get(streambuf &buf) legge i caratteri dallo stream di input all'oggetto streambuf. I caratteri vengono letti fino al codice di fine riga o fino alla fine del file. La funzione restituisce l'indirizzo dello stream. Un eventuale carattere di fine riga presente dello stream non verrà estratto.

get(streambuf &buf, char *delim*) legge i caratteri dallo stream di input all'oggetto streambuf. I caratteri vengono letti fino al delimitatore o fino alla fine del file. La funzione restituisce l'indirizzo dello stream. Un eventuale delimitatore presente dello stream non verrà estratto.

Le funzioni correlate sono put(), read() e getline().

## getline()

```
#include <iostream>
istream &getline(char *buf, streamsize num);
istream &getline(char *buf, streamsize num, char delim);
```

La funzione getline() è un membro di istream.

getline(char *buf, streamsize *num*) legge dei caratteri nell'array puntato da *buf* finché non vengono letti *num* - 1 caratteri, finché non viene trovato un codice di fine riga o fino alla fine del file. L'array puntato da *buf* verrà chiuso dal carattere nullo. Se nella stringa viene rilevato il carattere di fine riga, questo verrà estratto ma non verrà inserito in *buf*. La funzione restituisce l'indirizzo dello stream.

getline(char *buf, streamsize *num*, char *delim*) legge dei caratteri nell'array puntato da *buf* finché non vengono letti *num* - 1 caratteri, finché non viene incontrato il delimitatore *delim* o fino alla fine del file. L'array puntato da *buf* verrà chiuso dal carattere nullo. Se nella stringa viene rilevato il delimitatore, questo verrà estratto ma non verrà inserito in *buf*. La funzione restituisce l'indirizzo dello stream.

Le funzioni correlate sono get() e read().

## good()

```
#include <iostream>
bool good() const;
```

La funzione good() è un membro di ios.

La funzione good() restituisce true se nello stream non si è verificato alcun errore di I/O, altrimenti restituisce false.

Le funzioni correlate sono bad(), fail(), eof(), clear() e rdstate().

## ignore()

```
#include <iostream>
istream &ignore(streamsize num = 1, int delim = EOF);
```

La funzione ignore() è un membro di istream.

La funzione ignore() legge ed elimina i caratteri dallo stream di input. I caratteri vengono letti ed eliminati finché non vengono letti *num* caratteri (normalmente *num* è uguale a 1) o finché non viene incontrato il carattere specificato da *delim* (normalmente EOF). Se viene incontrato il delimitatore, questo verrà rimosso dallo stream di input. La funzione restituisce l'indirizzo dello stream.

Le funzioni correlate sono get() e getline().

## open()

```
#include <fstream>
 void fstream::open(const char *filename,
                    ios::openmode mode = ios::in | ios:: out);
void ifstream::open(const char *filename,
                    ios::openmode mode = ios::in);
void ofstream::open(const char *filename,
                    ios::openmode mode = ios:: out | ios::trunc);
```

La funzione open() è un membro di fstream, ifstream e ofstream.

La funzione open() associa un file a uno stream. Qui *filename* è il nome del file che può includere uno specificatore di percorso. La modalità di apertura del file è determinata da *mode* che può assumere uno o più dei seguenti valori:

ios::app
ios::ate
ios::binary
ios::in
ios::out
ios::trunc

Per combinare due o più valori si usa l'operatore OR.

Con ios::app tutto l'output verrà aggiunto alla fine del file. Questo valore può essere utilizzato solo su file in grado di accettare operazioni di output. ios::ate provoca il posizionamento alla fine del file subito dopo l'apertura. Nonostante questo posizionamento iniziale alla fine del file, le operazioni di I/O possono comunque verificarsi in qualsiasi punto del file.



Il valore ios::binary fa in modo che il file venga aperto per operazioni di I/O binarie. Normalmente il file viene aperto in modalità testo.

Il valore ios::in specifica che il file è utilizzabile per operazioni di input. Il valore ios::out specifica che il file è disponibile per operazioni di output. Tuttavia la creazione di uno stream ifstream presuppone l'impiego di operazioni di input, la creazione uno stream ofstream prevede operazioni di output e l'apertura di un file con fstream prevede operazioni sia di input che di output.

Il valore ios::trunc fa in modo che il contenuto di un file preesistente venga distrutto; la lunghezza del file viene troncata a 0.

In ogni caso, se open() termina senza successo lo stream sarà false. Pertanto, prima di usare un file è opportuno assicurarsi che l'operazione di apertura abbia avuto successo.

Le funzioni correlate sono close(), fstream(), ifstream() e ofstream().

## peek()

```
#include <iostream>
int peek();
```

La funzione peek() è un membro di istream.

La funzione peek() restituisce il carattere successivo dello stream oppure EOF quando ci si trova alla fine del file. Questa funzione non estrae mai il carattere dallo stream.

La funzione correlata è get().

## precision()

```
#include <iostream>
streamsize precision() const;
streamsize precision(streamsize p);
```

La funzione precision() è un membro di ios (ereditata da ios_base).

Normalmente quando si produce in output un valore in virgola mobile, vengono utilizzate 6 cifre di precisione. Se si usa la seconda forma di precision() si può impiegare il valore di precisione specificato da *p*. Viene restituito il valore precedente.

Le funzioni correlate sono width() e fill().

## put()

```
#include <iostream>
ostream &put(char ch);
```

La funzione put() è un membro di ostream.

La funzione put() scrive *ch* sullo stream di output. Restituisce l'indirizzo dello stream.

Le funzioni correlate sono write() e get().

## putback()

```
#include <iostream>
istream &putback(char ch);
```

La funzione putback() è un membro di istream.

La funzione putback() restituisce *ch* allo stream di input.

La funzione correlata è peek().

## rdstate()

```
#include <iostream>
iostate rdstate() const;
```

La funzione rdstate() è un membro di ios.

La funzione rdstate() restituisce lo stato dello stream. Il sistema di I/O del C++ gestisce le informazioni di stato per ogni operazione di I/O per ciascuno stream attivo. Lo stato corrente di uno stream è contenuto in un oggetto di tipo iostate in cui sono definiti i seguenti flag.

| NOME | SIGNIFICATO |
|---|---|
| goodbit | Nessun errore. |
| eofbit | Raggiunta la fine del file. |
| failbit | Errore di I/O non fatale. |
| badbit | Errore di I/O fatale. |

Questi flag sono enumerati in ios (tramite ios_base).

Se non si è verificato alcun errore, rdstate() restituisce goodbit, altrimenti imposta un bit di errore.



Le funzioni correlate sono eof(), good(), bad(), clear(), setstate() e fail().

## read()

```
#include <iostream>
istream &read(char *buf, streamsize num);
```

La funzione read() è un membro di istream.

La funzione read() legge *num* byte dallo stream di input e li inserisce nel buffer puntato da *buf*. Se prima di leggere *num* caratteri viene raggiunta la fine del file, read() si ferma, imposta failbit e lascia nel buffer i caratteri disponibili (vedere gcount()). read() restituisce l'indirizzo dello stream.

Le funzioni correlate sono gcount(), readsome(), get(), getline() e write().

## readsome()

```
#include <iostream>
streamsize readsome(char *buf, streamsize num);
```

La funzione readsome() è un membro di istream.

La funzione readsome() legge *num* byte dallo stream di input associato e li inserisce nel buffer puntato da *buf*. Se lo stream contiene meno di *num* caratteri, verranno letti i caratteri disponibili. readsome() restituisce il numero di caratteri letti. La differenza fra read() e readsome() consiste nel fatto che quest'ultima non imposta failbit quando legge meno di *num* caratteri.

Le funzioni correlate sono gcount(), read() e write().

## seekg() e seekp()

```
#include <iostream>
istream &seekg(off_type offset, ios::seekdir origin)
istream &seekg(pos_type position);

ostream &seekp(off_type offset, ios::seekdir origin);
ostream &seekp(pos_type position);
```

La funzione seekg() è un membro di istream e la funzione seekp() è un membro di ostream.

Nel sistema di I/O del C++, le funzioni seekg() e seekp() consentono di eseguire operazioni di accesso diretto ai file. In particolare il sistema di I/O C++ gestisce due puntatori per ogni file. Uno è il puntatore che specifica la posizione

in cui avverrà la prossima operazione di input e l'altro è il puntatore che specifica dove avverrà la prossima operazione di output. Ogni volta che viene eseguita un'operazione di input o di output, viene fatto avanzare il puntatore appropriato. Le funzioni seekg() e seekp() consentono di accedere al file in modo non sequenziale.

La versione a due parametri di seekg() sposta il puntatore di lettura a *offset* byte dalla posizione specificata da *origin*. La versione a due parametri di seekp() sposta il puntatore di scrittura a offset byte rispetto alla posizione specificata da *origin*. Il parametro *offset* è di tipo off_type, un tipo in grado di contenere il più esteso valore valido che può essere inserito in *offset*.

Il parametro *origin* è di tipo seekdir, un'enumerazione contenente i seguenti valori:

| | |
|---|---|
| ios::beg | Posizionamento dall'inizio. |
| ios::cur | Posizionamento dalla posizione corrente. |
| ios::end | Posizionamento dalla fine. |

La versione a un solo parametro di seekg() e seekp() porta il puntatore del file alla posizione specificata. Questo valore deve essere ottenuto precedentemente con una chiamata, rispettivamente, a tellg() o tellp(). pos_type è un tipo in grado di contenere il più esteso valore valido che può essere inserito in *position*. Queste funzioni restituiscono l'indirizzo dello stream.

Le funzioni correlate sono tellg() e tellp().

## setf()

```
#include <iostream>
fmtflags setf(fmtflags flags);
fmtflags setf(fmtflags flags1, fmtflags flags2);
```

La funzione setf() è un membro di ios (ereditata da ios_base).

La funzione setf() imposta i flag di formattazione di uno stream. Per informazioni sui flag di formattazione consultare la prima parte di questo capitolo.

La prima versione di setf() attiva i flag di formattazione specificati da *flags* (tutti gli altri flag non vengono modificati). Ad esempio, per attivare il flag showpos per cout si può usare la seguente istruzione:

```
cout.setf(ios::showpos);
```

Per impostare più flag si può utilizzare l'operatore OR.

È importante comprendere che una chiamata a setf() fa riferimento a un determinato stream. Non ha senso richiamare setf() se non in relazione a uno stream. In altre parole in C++ non esistono flag di formattazione globali. Ogni stream gestisce in modo indipendente i propri flag di formattazione.



La seconda versione di setf() modifica i soli flag che sono impostati in *flags2*. I flag corrispondenti vengono prima azzerati e poi impostati secondo quanto indicato da flags1. Anche se flags1 contiene altri flag, verranno modificati i soli flag specificati da *flags2*.

Entrambe le versioni di setf() restituiscono l'impostazione precedente dei flag dello stream.

Le funzioni correlate sono unsetf() e flags().

## setstate()

```
#include <iostream>
void setstate(iostate flags) const;
```

La funzione setstate() è un membro di ios.

La funzione setstate() imposta lo stato dello stream secondo quanto indicato da flags. Per informazioni consultare rdstate().

Le funzioni correlate sono clear() e rdstate().

## str()

```
#include <sstream>
string str() const;
void str(string &s)
```

La funzione str() è un membro di stringstream, istringstream e ostringstream.

La prima forma della funzione str() restituisce un oggetto *string* con l'attuale contenuto dello stream di stringhe.

La seconda forma, elimina la stringa attualmente contenuta nello stream di stringhe e la sostituisce con la stringa puntata da *s*.

Le funzioni correlate sono get() e put().

## stringstream(), istringstream() e ostringstream()

```
#include <sstream>
explicit stringstream(ios::openmode mode = ios::in | ios::out);
explicit stringstream(const string &str,
                      ios::openmode mode =ios::in | ios::out);
```

```
explicit istringstream(ios::openmode mode=ios::in);
explicit istringstream(const string str, ios::openmode mode=ios::in);
explict ostringstream(ios::openmode mode=ios::out);
explict ostringstream(const string str, ios::openmode
mode=ios::out);
```

Le funzioni stringstream(), istringstream() e ostringstream() sono rispettivamente i costruttori delle classi stringstream, istringstream e ostringstream. These construct streams that are tied to strings.

Le versioni di stringstream(), istringstream() e ostringstream() che specificano solo il parametro *openmode* creano stream vuoti. Le versioni che accettano un parametro *string* inizializzano lo stream di stringhe.

Ecco un esempio che illustra l'uso di uno stream di stringhe.

```
// Illustra l'uso di string streams.
#include <iostream>
#include <sstream>
using namespace std;

int main()
{
  stringstream s("Questa è la stringa iniziale.");

  // legge la stringa
  string str = s.str();
  cout << str << endl;

  // output sulla stringa
  s << "Numeri: " << 10 << " " << 123.2;

  int i;
  double d;
  s >> str >> i >> d;
  cout << str << " " << i << " " << d;

  return 0;
}
```

Ecco l'output prodotto dal programma:

```
Questa è la stringa iniziale.
Numeri: 10 123.2
```

La funzione correlata è str().



## sync_with_stdio()

```
#include <iostream>
bool sync_with_stdio(bool sync = true );
```

Una chiamata a sync_with_stdio() consente di utilizzare in sistema di I/O C insieme al sistema di I/O C++. Per disattivare la sincronizzazione di stdio si deve passare a sync_with_stdio() il valore false. Viene restituita l'impostazione precedente: true per la sincronizzazione e false per l'assenza di sincronizzazione. Normalmente gli stream standard sono sincronizzati. Questa funzione è affidabile solo se viene richiamata prima di ogni altra operazione di I/O.

## tellg() e tellp()

```
#include <iostream>
pos_type tellg();
pos_type tellp();
```

La funzione tellg() è un membro di istream e tellp() è un membro di ostream.

Il sistema di I/O C++ gestisce due puntatori per ogni file. Uno è il puntatore che specifica la posizione in cui avverrà la prossima operazione di input e l'altro è il puntatore che specifica dove avverrà la prossima operazione di output. Ogni volta che viene eseguita un'operazione di input o di output, viene fatto avanzare il puntatore appropriato. Per determinare la posizione corrente del puntatore di lettura si usa tellg() mentre per determinare la posizione del puntatore di scrittura si usa tellp().

pos_type è un tipo in grado di contenere il massimo valore che può essere restituito da queste funzioni.

Il valore restituito da tellg() e tellp() può essere utilizzato come parametro per seekg() e seekp().

Le funzioni correlate sono seekg() e seekp().

## unsetf()

```
#include <iostream>
void unsetf(fmtflags flags);
```

La funzione unsetf() è un membro di ios (ereditata da ios_base).

La funzione unsetf() cancella uno o più flag di formattazione. Vengono cancellati tutti i flag specificati da *flags* (tutti gli altri flag non vengono influenzati).

Le funzioni correlate sono setf() e flags().

### width()

```
#include <iostream>
streamsize width() const;
streamsize width(streamsize w);
```

La funzione width() è un membro di ios (ereditata da ios_base).

La prima forma di width() fornisce l'attuale larghezza del campo. La seconda imposta la larghezza del campo. In questo caso *w* diviene la larghezza del campo e viene restituita la precedente larghezza.

Le funzioni correlate sono precision() e fill().

### write()

```
#include <iostream>
ostream &write(const char *buf, streamsize num);
```

La funzione write() è un membro di ostream.

La funzione write() scrive *num* byte sullo stream di output associato al buffer puntato da *buf*. La funzione restituisce l'indirizzo dello stream.

Le funzioni correlate sono read() e put().

Capitolo 33

# Le classi container STL

Questo capitolo descrive le classi che implementano i container definiti dalla libreria STL. I container sono quella parte della libreria che fornisce un sistema di memorizzazione per altri oggetti. Oltre a fornire la memoria necessaria per gli oggetti, i container definiscono il meccanismo di accesso agli oggetti contenuti nel container. Pertanto i container sono dispositivi di memorizzazione ad alto livello.

**NOTA** *Per una panoramica della libreria STL, consultare il Capitolo 24.*

Nella descrizione dei container, verranno impiegate le seguenti convenzioni. Quando si fa riferimento genericamente ai vari tipi di iteratori, nel volume vengono impiegati i seguenti termini.

| TERMINE | SIGNIFICATO |
|---|---|
| `BiIter` | Iteratore bidirezionale |
| `ForIter` | Iteratore in avanti |
| `InIter` | Iteratore di input |
| `OutIter` | Iteratore di output |
| `RandIter` | Iteratore di accesso diretto |

Quando è richiesta una funzione con predicato unario, si farà riferimento a UnPred. Quando è necessario un predicato binario, verrà utilizzato il tipo BinPred. In un predicato binario gli argomenti sono sempre nello stesso ordine della funzione che richiama il predicato. Per i predicati unari e binari, gli argomenti contengono il valore del tipo degli oggetti da memorizzare nel container.

Le funzioni di confronto saranno indicate dal tipo Comp.

Un'ultima annotazione: nella descrizione seguente, quando si dice che un iteratore punta alla fine di un container si intende che l'iteratore punta appena dopo l'ultimo oggetto del container.

La libreria STL definisce le seguenti classi container.

| CONTAINER | DESCRIZIONE | HEADER RICHIESTO |
|---|---|---|
| `bitset` | Un insieme di bit. | `<bitset>` |
| `deque` | Una coda a doppio concatenamento. | `<deque>` |
| `list` | Una lista lineare. | `<list>` |
| `map` | Memorizza coppie chiave/valore in cui a ogni chiave è associato un solo valore. | `<map>` |
| `multimap` | Memorizza coppie chiave/valore in cui a ogni chiave possono essere associati due o più valori. | `<map>` |
| `multiset` | Un insieme in cui ogni elemento non è necessariamente univoco. | `<set>` |
| `priority_queue` | Una coda a priorità. | `<queue>` |
| `queue` | Una coda. | `<queue>` |
| `set` | Un insieme in cui ogni elemento è univoco. | `<set>` |
| `stack` | Uno stack. | `<stack>` |
| `vector` | Un array dinamico. | `<vector>` |

Le prossime sezioni si occuperanno di ogni singolo container. Poiché i container sono implementati utilizzando classi template, verranno utilizzati vari tipi fittizi. Nelle descrizioni seguenti, il tipo generico T rappresenta il tipo dei dati memorizzati dal container.

Poiché il nome dei tipi fittizi in una classe template è arbitrario, le classi container dichiarano delle versioni di questi tipi create con typedef. Questo rende più concreti i nomi dei tipi. Ecco i nomi typedef utilizzati dalle classi container.

| | |
|---|---|
| size_type | Un tipo intero equivalente più o meno a size_t. |
| reference | L'indirizzo di un elemento. |
| const_reference | L'indirizzo const di un elemento. |
| difference_type | Può rappresentare la differenza fra due indirizzi. |
| iterator | Un iteratore. |



| | |
|---|---|
| const_iterator | Un iteratore const. |
| reverse_iterator | Un iteratore inverso. |
| const_reverse_iterator | Un iteratore const inverso. |
| value_type | Il tipo di un valore contenuto in un container (equivale al tipo generico T). |
| allocator_type | Il tipo dell'allocatore. |
| key_type | Il tipo di una chiave. |
| key_compare | Il tipo di una funzione che confronta due chiavi. |
| mapped_type | Il tipo del valore contenuto in una mappa (equivale al tipo generico T). |
| value_compare | Il tipo di una funzione che confronta due valori. |
| value_type | Il tipo dei valori su cui si opera (equivale al tipo generico T). |
| pointer | Il tipo di un puntatore. |
| const_pointer | Il tipo di un puntatore const. |
| container_type | Il tipo di un container. |

## La classe bitset

La classe bitset supporta le operazioni su insiemi di bit. La sua specifica template è:

```
template <size_t <N>> class bitset;
```

Qui, *N* specifica la lunghezza del bitset in bit. La classe ha il seguente costruttore:

```
bitset( );

bitset(unsigned long bits);

explicit bitset(const string &s, size_t i = 0, size_t num = npos);
```

La prima forma costituisce un bitset vuoto. La seconda forma costruisce un bitset i cui bit vengono impostati secondo quanto specificato in *bits*. La terza forma costruisce un bitset utilizzando la stringa *s* a partire da *x*. La stringa deve contenere unicamente i numeri 1 e 0. Possono essere usate solo *num* o *s*.size() - *i* valori (il minore dei due). La costante *npos* è un valore sufficientemente esteso per descrivere la lunghezza massima di *s*.

Per bitset sono definiti gli operatori di I/O << e >>.
La classe bitset contiene le seguenti funzioni membro.

| MEMBRO | DESCRIZIONE |
|---|---|
| `bool any( ) const;` | Restituisce true se un bit nel bitset chiamante è uguale a 1; altrimenti restituisce false. |
| `size_type count( ) const;` | Restituisce il numero di bit a 1. |
| `bitset<N> &flip( );` | Inverte lo stato di tutti i bit contenuti nel bitset chiamante e restituisce `*this`. |
| `bitset<N> &flip(size_t i);` | Inverte il bit nella posizione *i* nel bitset chiamante e restituisce `*this`. |
| `bool none( ) const;` | Restituisce true se nel bitset chiamante nessun bit è uguale a 1. |
| `bool operator !=(const bitset<N> &op2) const;` | Restituisce true se il bitset chiamante differisce da quello specificato dall'operatore di destra, *op2*. |
| `bool operator ==(const bitset<N> &op2) const;` | Restituisce true se il bitset chiamante è uguale a quello specificato dall'operatore di destra, *op2*. |
| `bitset<N> &operator &=(const bitset<N> &op2);` | Esegue l'AND su ogni bit del bitset chiamante con i corrispondenti bit di *op2* e lascia il risultato nel bitset chiamante. Restituisce `*this`. |
| `bitset<N> &operator ^=(const bitset<N> &op2);` | Esegue lo XOR su ogni bit del bitset chiamante con il bit corrispondente in *op2* e lascia il risultato nel bitset chiamante. Restituisce `*this`. |
| bitset<N> &operator \|=(const bitset<N> &*op2*); | Esegue l'OR di ogni bit del bitset chiamante con il bit corrispondente in *op2* e lascia il risultato nel bitset chiamante. Restituisce `*this`. |
| bitset<N> &operator ~=( ) const; | Inverte lo stato di tutti i bit del bitset chiamante e restituisce il risultato. |
| bitset<N> &operator <<=(size_t *num*); | Esegue uno scorrimento a sinistra di ogni bit del bitset chiamante di *num* posizioni e lascia il risultato nel bitset chiamante. Restituisce `*this`. |
| bitset<N> &operator >>=(size_t *num*); | Esegue uno scorrimento a destra di ogni bit del bitset chiamante di *num* posizioni e lascia il risultato nel bitset chiamante. Restituisce `*this`. |
| reference operator [ ](size_type *i*); | Restituisce l'indirizzo del bit *i* del bitset chiamante. |
| bitset<N> &reset( ); | Cancella tutti i bit del bitset chiamante e restituisce `*this`. |
| bitset<N> &reset(size_t *i*); | Cancella il bit nella posizione *i* del bitset chiamante e restituisce `*this`. |
| bitset<N> &set( ); | Assegna 1 a tutti i bit del bitset chiamante e restituisce `*this`. |
| bitset<N> &set(size_t *i*, int *val* = 1); | Assegna al bit nella posizione *i* il valore specificato da *val* nel bitset chiamante e restituisce `*this`. Ogni valore diverso da zero contenuto in *val* sarà considerato uguale a 1. |

*(segue)*



*(continua)*

| MEMBRO | DESCRIZIONE |
|---|---|
| size_t size( ) const; | Restituisce il numero di bit che il bitset può contenere. |
| bool test(size_t *i*) const; | Restituisce lo stato del bit nella posizione *i*. |
| string to_string( ) const; | Restituisce una stringa che contiene una rappresentazione del modello di bit del bitset chiamante. |
| unsigned long to_ulong( ) const; | Converte il bitset chiamante in un unsigned long int. |

## La classe deque

La classe deque supporta una coda a doppio concatenamento. La sua specifica template è:

```
template <class T, class Allocator = allocator<T>> class deque
```

Qui T è il tipo dei dati memorizzati nella deque. La classe ha i seguenti costruttori:

```
explicit deque(const Allocator &a = Allocator( ) );

explicit deque(size_type num, const T &val = T ( ),
    const Allocator &a = Allocator( ));

deque(const deque<T, Allocator> &ob);

template <class InIter> deque(InIter start, InIter end,
    const Allocator &a = Allocator( ));
```

La prima forma costruisce una deque vuota. La seconda forma costruisce una deque contenente *num* elementi di valore *val*. La terza forma costruisce una deque che contiene gli stessi elementi di *ob*. La quarta forma costruisce una deque che contiene gli elementi dell'intervallo specificato da *start* ed *end*.

Per la classe deque sono definiti i seguenti operatori di confronto:

==, <, <=, !=, >, >=

La classe deque contiene le seguenti funzioni membro:

| MEMBRO | DESCRIZIONE |
|---|---|
| `template <class InIter> void assign(InIter start, InIter end);` | Assegna alla deque la sequenza definita da *start* ed *end*. |
| `void assign(size_type num, const T &val);` | Assegna a *num* elementi della deque il valore *val*. |
| `reference at(size_type i); const_reference at(size_type i) const;` | Restituisce l'indirizzo dell'elemento specificato da *i*. |
| `reference back( ); const_reference back( ) const;` | Restituisce l'indirizzo dell'ultimo elemento della deque. |
| `iterator begin( ); const_iterator begin( ) const;` | Restituisce un iteratore al primo elemento della deque. |
| `void clear( );` | Rimuove tutti gli elementi dalla deque. |
| `bool empty( ) const;` | Restituisce true se la deque chiamante è vuota altrimenti restituisce false. |
| `const_iterator end( ) const; iterator end( );` | Restituisce un iteratore alla fine della deque. |
| `iterator erase(iterator i);` | Rimuove l'elemento puntato da *i*. Restituisce l'iteratore dell'elemento che segue quello rimosso. |
| `iterator erase(iterator start, iterator end);` | Rimuove gli elementi dell'intervallo compreso fra *start* ed *end*. Restituisce l'iteratore dell'elemento che segue l'ultimo elemento rimosso. |
| `reference front( ); const_reference front( ) const;` | Restituisce l'indirizzo del primo elemento della deque. |
| `allocator_type get_allocator( ) const;` | Restituisce l'allocatore della deque. |
| `iterator insert(iterator i, const T &val);` | Inserisce *val* immediatamente prima dell'elemento specificato da *i*. Restituisce l'iteratore dell'elemento. |
| `void insert(iterator i, size_type num, const T &val);` | Inserisce *num* copie di *val* immediatamente prima dell'elemento specificato da *i*. |
| `template <class InIter> void insert(iterator i, InIter start, InIter end);` | Inserisce la sequenza definita da *start* ed *end* immediatamente prima dell'elemento specificato da *i*. |
| `size_type max_size( ) const;` | Restituisce il numero massimo di elementi che la deque può contenere. |
| `reference operator[ ](size_type i); const_reference operator[ ](size_type i) const;` | Restituisce l'indirizzo dell'elemento *i*esimo. |
| `void pop_back( );` | Rimuove l'ultimo elemento della deque. |
| `void pop_front( );` | Rimuove il primo elemento della deque. |

(segue)



(continua)

| MEMBRO | DESCRIZIONE |
|---|---|
| `void push_back(const T &val);` | Aggiunge un elemento con il valore specificato da *val* alla fine della deque. |
| `void push_front(const T &val);` | Aggiunge un elemento con il valore specificato da *val* in fronte alla deque. |
| `reverse_iterator rbegin( ); const_reverse_iterator rbegin( ) const;` | Restituisce un iteratore inverso alla fine della deque. |
| `reverse_iterator rend( ); const_reverse_iterator rend( ) const;` | Restituisce un iteratore inverso all'inizio della deque. |
| `void resize(size_type num, T val = T ( ));` | Cambia le dimensioni della deque secondo quanto specificato da *num*. Se la deque deve essere allungata, allora gli elementi con il valore specificato da *val* vengono aggiunti alla fine. |
| `size_type size( ) const;` | Restituisce il numero di elementi attualmente presenti nella deque. |
| `void swap(deque<T, Allocator> &ob);` | Scambia gli elementi contenuti nella deque chiamante con quelli di *ob*. |

## La classe list

La classe list supporta una lista. La sua specifica template è:

```
template <class T, class Allocator = allocator<T>> class list
```

Qui, T è il tipo dei dati memorizzati nella lista. La classe ha il seguente costruttore:

```
explicit list(const Allocator &a = Allocator( ) );

explicit list(size_type num, const T &val = T ( ),
              const Allocator &a = Allocator( ));

list(const list<T, Allocator> &ob);

template <class InIter>list(InIter start, InIter end,
              const Allocator &a = Allocator( ));
```

La prima forma costruisce una lista vuota. La seconda forma costruisce una lista contenente *num* elementi di valore *val*. La terza forma costruisce una lista

che contiene gli stessi elementi di *ob*. La quarta forma costruisce una lista che contiene gli elementi contenuti nell'intervallo specificato da *start* ed *end*. Per list sono definiti i seguenti operatori di confronto:

==, <, <=, !=, >, >=

La classe list contiene le seguenti funzioni membro.

| MEMBRO | DESCRIZIONE |
|---|---|
| `Template <class InIter> void assign(InIter start, InIter end);` | Assegna alla lista la sequenza definita da *start* ed *end*. |
| `Void assign(size_type num, const T &val);` | Assegna alla lista *num* elementi di valore *val*. |
| `reference back( );`<br>`const_reference back( ) const;` | Restituisce l'indirizzo dell'ultimo elemento contenuto nella lista. |
| `iterator begin( );`<br>`const_iterator begin( ) const;` | Restituisce un iteratore al primo elemento contenuto nella lista. |
| `void clear( );` | Rimuove tutti gli elementi dalla lista. |
| `bool empty( ) const;` | Restituisce true se la lista chiamante è vuota altrimenti restituisce false. |
| `iterator end( );`<br>`const_iterator end( ) const;` | Restituisce un iteratore alla fine della lista. |
| `iterator erase(iterator i);` | Rimuove l'elemento puntato da *i*. Restituisce l'iteratore dell'elemento che segue quello rimosso. |
| `iterator erase(iterator start, iterator end);` | Rimuove gli elementi dell'intervallo compreso fra *start* ed *end*. Restituisce l'iteratore dell'elemento che segue l'ultimo elemento rimosso. |
| `reference front( );`<br>`const_reference front( ) const;` | Restituisce l'indirizzo del primo elemento contenuto nella lista. |
| `allocator_type get_allocator( ) const;` | Restituisce l'allocatore della lista. |
| `iterator insert(iterator i,`<br>`const T &val = T( ));` | Inserisce *val* immediatamente prima dell'elemento specificato da *i*. Restituisce l'iteratore dell'elemento. |
| `void insert(iterator i, size_type num,`<br>`const T & val);` | Inserisce *num* copie di *val* immediatamente prima dell'elemento specificato da *i*. |
| `template <class InIter>`<br>`void insert(iterator i,`<br>`InIter start, InIter end);` | Inserisce la sequenza definita da *start* ed *end* immediatamente prima dell'elemento specificato da *i*. |
| `size_type max_size( ) const;` | Restituisce il numero massimo di elementi che possono essere contenuti nella lista. |

(segue)



(continua)

| MEMBRO | DESCRIZIONE |
|---|---|
| `void merge(list<T, Allocator> &ob);`<br>`template <class Comp>`<br>`void merge(list<T, Allocator> &ob,`<br>`Comp cmpfn);` | Unisce la lista ordinata contenuta in *ob* con la lista ordinata chiamante. Il risultato è ordinato. Dopo l'unione, la lista contenuta in *ob* è vuota. Nella seconda forma, si può specificare una funzione di confronto che determina la posizione relativa degli elementi. |
| `void pop_back( );` | Rimuove l'ultimo elemento contenuto nella lista. |
| `void pop_front( );` | Rimuove il primo elemento contenuto nella lista. |
| `void push_back(const T &val);` | Aggiunge un elemento con il valore specificato da *val* alla fine della lista. |
| `void push_front(const T &val);` | Aggiunge un elemento con il valore specificato da *val* in fronte alla lista. |
| `reverse_iterator rbegin( );`<br>`const_reverse_iterator rbegin( ) const;` | Restituisce un iteratore inverso alla fine della lista. |
| `void remove(const T &val);` | Rimuove dalla lista gli elementi con valore *val*. |
| `template <class nPred>`<br>`void remove_if(UnPred_pr);` | Rimuove gli elementi per i quali il predicato unario *pr* è true.Restituisce un iteratore inverso alla fine della lista. |
| `reverse_iterator rend( );`<br>`const_reverse_iterator rend( ) const;` | Restituisce un iteratore inverso all'inizio della lista. |
| `void resize(size_type num, T val = T ( ));` | Cambia le dimensioni della lista a quelle specificate da *num*. Se la lista deve essere allungata, allora gli elementi con il valore specificato da *val* vengono aggiunti alla fine. |
| `void reverse( );` | Inverte la lista chiamante. |
| `size_type size( ) const;` | Restituisce il numero di elementi attualmente contenuti nella lista. |
| `void sort( );`<br>`template <class Comp>`<br>`void sort(Comp cmpfn);` | Ordina la lista. La seconda forma ordina la lista utilizzando la funzione di confronto *cmpfn* per determinare la posizione relativa degli elementi. |
| `void splice(iterator i,`<br>`list<T, Allocator> &ob);` | Il contenuto di *ob* viene inserito nella lista chiamante alla posizione puntata da *i*. Dopo l'operazione, *ob* è vuota. |
| `void splice(iterator i,`<br>`list<T, Allocator> &ob,`<br>`iterator el);` | L'elemento puntato da *el* viene rimosso dalla lista *ob* e memorizzato nella lista chiamante alla posizione puntata da *i*. |
| `void splice(iterator i,`<br>`list<T, Allocator> &OB,`<br>`ITERATOR start, iterator end);` | L'intervallo definito da *start* ed *end* viene rimosso da *ob* e memorizzato nella lista chiamante a partire dalla posizione puntata da *i*. |
| `void swap(list<T, Allocator> &ob);` | Scambia gli elementi memorizzati nella lista chiamante con quelli contenuti in *ob*. |

(segue)

*(continua)*

| MEMBRO | DESCRIZIONE |
|---|---|
| `void unique( );`<br>`template <class inPred>`<br>`  void unique(BinPred pr);` | Rimuove gli elementi duplicati dalla lista chiamante. La seconda forma determina l'univocità utilizzando *pr*. |

## La classe map

La classe map supporta un container associativo in cui a ogni chiave univoca corrisponde un valore. La sua specifica template è la seguente:

```
template <class Key, class T, class Comp = less<Key>,
          class Allocator = allocator<T>> class map
```

Qui, Key è il tipo delle chiavi, T è il tipo dei valori memorizzati (mappati) e Comp è una funzione che confronta due chiavi. La classe map ha i seguenti costruttori:

```
explicit map(const Comp &cmpfn = Comp( ),
             const Allocator &a = Allocator( ) );

map(const map<Key, T, Comp, Allocator> &ob);

template <class InIter> map(InIter start, InIter end,
          const Comp &cmpfn = Comp( ),
          const Allocator &a = Allocator( ));
```

La prima forma costruisce una mappa vuota. La seconda forma costruisce una mappa che contiene gli stessi elementi di *ob*. La terza forma costruisce una mappa che contiene gli elementi nell'intervallo specificato da *start* ed *end*. L'eventuale ordinamento della mappa è determinato dalla funzione *cmpfn*.

Per la classe map sono definiti i seguenti operatori di confronto:

```
==, <, <=, !=, >, >=
```

Di seguito sono elencate le funzioni membro contenute in map. Nelle descrizioni, key_type è il tipo della chiave e value_type rappresenta pair<Key, T>.



| MEMBRO | DESCRIZIONE |
|---|---|
| `iterator begin( );`<br>`const_iterator begin( ) const;` | Restituisce un iteratore al primo elemento della mappa. |
| `void clear( );` | Rimuove tutti gli elementi dalla mappa. |
| `size_type count(const key_type &k) const;` | Restituisce il numero di volte che *k* è presente nella mappa (1 o 0). |
| `bool empty( ) const;` | Restituisce true se la mappa chiamante è vuota altrimenti restituisce false. |
| `iterator end( );`<br>`const_iterator end( ) const;` | Restituisce un iteratore alla fine della mappa. |
| `pair<iterator, iterator>`<br>`  equal_range(const key_type &k);`<br>`pair<const_iterator, const_iterator>`<br>`  equal_range(const key_type &k) const;` | Restituisce una coppia di iteratori che puntano al primo e all'ultimo elemento della mappa che contengono la chiave specificata. |
| `void erase(iterator i);` | Rimuove l'elemento puntato da *i*. |
| `void erase(iterator start, iterator end);` | Rimuove gli elementi dell'intervallo compreso fra *start* ed *end*. |
| `size_type erase(const key_type &k);` | Rimuove dalla mappa gli elementi la cui chiave è uguale al valore *k*. |
| `iterator find(const key_type &k);`<br>`const_iterator find(const key_type &k)`<br>`  const;` | Restituisce un iteratore alla chiave specificata. Se la chiave non viene trovata, allora restituisce un iteratore alla fine della mappa. |
| `allocator_type get_allocator( ) const;` | Restituisce l'allocatore della mappa. |
| `iterator insert(iterator i,`<br>`  const value_type &val);` | Inserisce *val* nella posizione o subito dopo l'elemento spacificato da *i*. Restituisce l'iteratore dell'elemento. |
| `template <class InIter>`<br>`  void insert(InIter start, InIter end);` | Inserisce un intervallo di elementi. |
| `pair<iterator, bool>`<br>`  insert(const value_type &val);` | Inserisce *val* nella mappa chiamante. Restituisce l'iteratore dell'elemento. L'elemento viene inserito solo se non è presente nella mappa. Se l'elemento viene inserito, la funzione restituisce `pair<iterator, true>`. Altrimenti restituisce `pair<iterator,< false>`. |
| `key_compare key_comp( ) const;` | Restituisce l'oggetto funzione che confronta le chiavi. |
| `iterator lower_bound(const key_type &k);`<br>`const_iterator`<br>`  lower_bound(const key_type &k) const;` | Restituisce un iteratore al primo elemento della mappa la cui chiave è maggiore o uguale a *k*. |
| `size_type max_size( ) const;` | Restituisce il numero massimo di elementi che la mappa può contenere. |
| `reference operator[ ](const key_type &i);` | Restituisce l'indirizzo dell'elemento specificato da *i*. Se questo elemento non esiste, viene inserito. |

*(segue)*

*(continua)*

| MEMBRO | DESCRIZIONE |
|---|---|
| `reverse_iterator rbegin( );`<br>`const_reverse_iterator rbegin( ) const;` | Restituisce un iteratore inverso alla fine della mappa. |
| `reverse_iterator rend( );`<br>`const_reverse_iterator rend( ) const;` | Restituisce un iteratore inverso all'inizio della mappa. |
| `size_type size( ) const;` | Restituisce il numero di elementi attualmente presenti nella mappa. |
| `void swap(map<Key, T, Comp, Allocator> &ob);` | Scambia gli elementi contenuti nella mappa chiamante con quelli di *ob*. |
| `iterator upper_bound(const key_type &k);`<br>`const_iterator upper_bound(const key_type &k) const;` | Restituisce un iteratore al primo elemento della mappa la cui chiave è maggiore o uguale a *k*. |
| `value_compare value_comp( ) const;` | Restituisce l'oggetto funzione che confronta i valori. |

## La classe multimap

La classe multimap supporta un container associativo i cui valori vengono mappati su chiavi non necessariamente univoche. Ecco la sua specifica template:

```
template <class Key, class T, class Comp  less<Key>,
          class Allocator = allocator<T>> class multimap
```

Qui, Key è il tipo delle chiavi, T è il tipo dei valori memorizzati (mappati) e Comp è una funzione che confronta due chiavi. La classe multimap ha i seguenti costruttori:

```
explicit multimap(const Comp &cmpfn = Comp( ),
                  const Allocator &a = Allocator( ) );

multimap(const multimap<Key, T, Comp, Allocator> &ob);

template <class InIter> multimap(InIter start, InIter end,
                  const Comp &cmpfn = Comp( ),
                  const Allocator &a = Allocator( ));
```

La prima forma costruisce una multimap vuota. La seconda forma costruisce una multimap che contiene gli stessi elementi di *ob*. La terza forma costruisce una multimap che contiene gli elementi nell'intervallo specificato da *start* ed *end*.



L'eventuale ordinamento della multimap è determinato dalla funzione *cmpfn*.

Per la classe multimap sono definiti i seguenti operatori di confronto:

==, <, <=, !=, >, >=

Di seguito sono elencate le funzioni membro contenute in multimap. Nelle descrizioni, key_type è il tipo della chiave e value_type rappresenta pair<Key, T>.

| MEMBRO | DESCRIZIONE |
|---|---|
| `iterator begin( );`<br>`const_iterator begin( ) const;` | Restituisce un iteratore al primo elemento della multimap. |
| `void clear( );` | Rimuove tutti gli elementi dalla multimap. |
| `size_type count(const key_type &k) const;` | Restituisce il numero di volte che *k* si presenta nella multimap. |
| `bool empty( ) const;` | Restituisce true se la multimap chiamante è vuota altrimenti restituisce false. |
| `iterator end( );`<br>`const_iterator end( ) const;` | Restituisce un iteratore alla fine della multimap. |
| `pair<iterator, iterator>`<br>`equal_range(const key_type &k);`<br>`pair<const_iterator, const_iterator>`<br>`equal_range(const key_type &k) const;` | Restituisce una coppia di iteratori che puntano al primo e all'ultimo elemento della multimap che contengono la chiave specificata. |
| `void erase(iterator i);` | Rimuove l'elemento puntato da *i*. |
| `void erase(iterator start, iterator end);` | Rimuove gli elementi dell'intervallo compreso fra *start* ed *end*. |
| `size_type erase(const key_type &k);` | Rimuove dalla multimap gli elementi la cui chiave è uguale al valore *k*. |
| `iterator find(const key_type &k);`<br>`const_iterator find(const key_type &k) const;` | Restituisce un iteratore alla chiave specificata. Se la chiave non viene trovata, restituisce un iteratore alla fine della multimap. |
| `allocator_type get_allocator( ) const;` | Restituisce l'allocatore della multimap. |
| `iterator insert(iterator i, const value_type &val);` | Inserisce *val* nella posizione o subito dopo l'elemento specificato da *i*. Restituisce l'iteratore dell'elemento. |
| `template <class InIter>`<br>`void insert(InIter start, InIter end);` | Inserisce un intervallo di elementi. |
| `iterator insert(const value_type &val);` | Inserisce *val* nella multimap chiamante. |
| `key_compare key_comp( ) const;` | Restituisce l'oggetto funzione che confronta le chiavi. |

*(segue)*

*(continua)*

| MEMBRO | DESCRIZIONE |
|---|---|
| `iterator lower_bound(const key_type &k);`<br>`const_iterator lower_bound(const key_type &k) const;` | Restituisce un iteratore al primo elemento della multimap la cui chiave è maggiore o uguale a *k*. |
| `size_type max_size( ) const;` | Restituisce il numero massimo di elementi che possono essere contenuti nella multimap. |
| `reverse_iterator rbegin( );`<br>`const_reverse_iterator rbegin( ) const;` | Restituisce un iteratore inverso alla fine della multimap. |
| `reverse_iterator rend( );`<br>`const_reverse_iterator rend( ) const;` | Restituisce un iteratore inverso all'inizio della multimap. |
| `size_type size( ) const;` | Restituisce il numero di elementi attualmente presenti nella multimap. |
| `void swap(multimap<Key, T, Comp, Allocator> &ob);` | Scambia gli elementi contenuti nella multimap chiamante con quelli di *ob*. |
| `iterator upper_bound(const key_type &k);`<br>`const_iterator upper_bound(const key_type &k) const;` | Restituisce un iteratore al primo elemento della multimap la cui chiave è maggiore o uguale a *k*. |
| `value_compare value_comp( ) const;` | Restituisce l'oggetto funzione che confronta i valori. |

## La classe multiset

La classe multiset supporta un insieme contenente chiavi non univoche. Ecco la sua specifica template:

```
template <class Key, class Comp  less<Key>,
          class Allocator = allocator<Key>> class multiset
```

Qui Key è il tipo delle chiavi e Comp è una funzione che confronta due chiavi. La classe ha i seguenti costruttori:

```
explicit multiset(const Comp &cmpfn = Comp( ),
                  const Allocator &a = Allocator( ) );

multiset(const multiset<Key, Comp, Allocator> &ob);

template <class InIter> multiset(InIter start, InIter end,
                  const Comp &cmpfn = Comp( ),
                  const Allocator &a = Allocator( ));
```



La prima forma costruisce un multiset vuoto. La seconda forma costruisce un multiset che contiene gli stessi elementi di *ob*. La terza forma costruisce un multiset che contiene gli elementi dell'intervallo specificato da *start* ed *end*. L'eventuale ordinamento del multiset è determinato dalla funzione *cmpfn*.

Per la classe multiset sono definiti i seguenti operatori di confronto:

==, <, <=, !=, >, >=

Di seguito sono elencate le funzioni membro contenute in multiset. Nelle descrizioni, sia key_type che value_type sono typedef per Key.

| MEMBRO | DESCRIZIONE |
|---|---|
| `iterator begin( );`<br>`const_iterator begin( ) const;` | Restituisce un iteratore al primo elemento del multiset. |
| `void clear( );` | Rimuove tutti gli elementi dal multiset. |
| `size_type count(const key_type &k) const;` | Restituisce il numero di volte che *k* è presente nel multiset. |
| `bool empty( ) const;` | Restituisce true se il multiset chiamante è vuoto altrimenti restituisce false. |
| `iterator end( );`<br>`const_iterator end( ) const;` | Restituisce un iteratore alla fine del multiset. |
| `pair<iterator, iterator>`<br>`equal_range(const key_type &k) const;` | Restituisce una coppia di iteratori che puntano al primo e all'ultimo elemento del multiset che contengono la chiave specificata. |
| `void erase(iterator i);` | Rimuove l'elemento puntato da *i*. |
| `void erase(iterator start, iterator end);` | Rimuove gli elementi dell'intervallo compreso fra *start* ed *end*. |
| `size_type erase(const key_type &k);` | Rimuove dal multiset gli elementi la cui chiave è uguale al valore *k*. |
| `iterator find(const key_type &k) const;` | Restituisce un iteratore alla chiave specificata. Se la chiave non viene trovata, restituisce un iteratore alla fine del multiset. |
| `allocator_type get_allocator( ) const;` | Restituisce l'allocatore del multiset. |
| `iterator insert(iterator i,`<br>`const value_type &val);` | Inserisce *val* nella posizione o subito dopo l'elemento spacificato da *i*. Restituisce l'iteratore dell'elemento. |
| `template <class InIter>`<br>`void insert(InIter start, InIter end);` | Inserisce un intervallo di elementi. |
| `iterator insert(const value_type &val);` | Inserisce *val* nel multiset chiamante. Restituisce l'iteratore dell'elemento. |
| `key_compare key_comp( ) const;` | Restituisce l'oggetto funzione che confronta le chiavi. |

*(segue)*

(continua)

| MEMBRO | DESCRIZIONE |
|---|---|
| `iterator lower_bound(const key_type &k) const;` | Restituisce un iteratore al primo elemento del multiset la cui chiave è maggiore o uguale a *k*. |
| `size_type max_size( ) const;` | Restituisce il numero massimo di elementi che il multiset può contenere. |
| `reverse_iterator rbegin( );`<br>`const_reverse_iterator rbegin( ) const;` | Restituisce un iteratore inverso alla fine del multiset. |
| `reverse_iterator rend( );`<br>`const_reverse_iterator rend( ) const;` | Restituisce un iteratore inverso all'inizio del multiset. |
| `size_type size( ) const;` | Restituisce il numero di elementi attualmente presenti nel multiset. |
| `void swap(multiset<Key, Comp, Allocator> &ob);` | Scambia gli elementi contenuti nel multiset chiamante con quelli di *ob*. |
| `iterator upper_bound(const key_type &k) const;` | Restituisce un iteratore al primo elemento del multiset la cui chiave è maggiore o uguale a *k*. |
| `value_compare value_comp( ) const;` | Restituisce l'oggetto funzione che confronta i valori. |

## La classe priority_queue

La classe priority_queue supporta una coda semplice a priorità. La sua specifica template è la seguente:

```
template <class T, class Container  vector<T>,
          class Comp = less<Container::value_type>>
          class priority_queue
```

Qui, T è il tipo dei dati da memorizzare. Container è il tipo del container utilizzato per contenere la coda e Comp specifica la funzione di confronto che determina quando un membro della coda ha una priorità inferiore rispetto a un altro. La classe priority_queue ha i seguenti costruttori:

```
explicit priority_queue(const Comp &cmpfn = Comp( ),
                        Container &cnt = Container( ));

template <class InIter> priority_queue(InIter start, InIter end,
                        const Comp &cmpfn = Comp( ),
                        Container &cnt = Container( ));
```



Il primo costruttore priority_queue() crea una coda a priorità vuota. Il secondo crea una coda a priorità che contiene gli elementi specificati dall'intervallo definito da *start* ed *end*. Normalmente priority_queue impiega come container un vector. In alternativa si può usare come container una deque. Il container è contenuto in un oggetto protetto chiamato c di tipo Container. La classe priority_queue contiene le seguenti funzioni membro.

| MEMBRO | DESCRIZIONE |
|---|---|
| `bool empty( ) const;` | Restituisce true se la coda a priorità è vuota altrimenti restituisce false. |
| `void pop( );` | Rimuove il primo elemento della coda a priorità. |
| `void push(const T &val);` | Aggiunge un elemento alla coda a priorità. |
| `size_type size( ) const;` | Restituisce il numero di elementi attualmente presenti nella coda a priorità. |
| `const value_type &top( ) const;` | Restituisce l'indirizzo dell'elemento con la priorità più elevata. L'elemento non viene rimosso. |

## La classe queue

La classe queue supporta una coda semplice. Ecco la sua specifica template:

```
template <class T, class Container  deque<T>> class queue
```

Qui, T è il tipo dei dati memorizzati e Container è il tipo del container utilizzato per la coda. La classe ha il seguente costruttore:

```
explicit queue(const Container &cnt = Container( ));
```

Il costruttore queue() crea una coda vuota. Normalmente queue utilizza come container una deque ma una queue può essere consultata solo in modo FIFO (First In First Out). Come container di una coda si può usare anche una lista. Il container è contenuto in un oggetto protetto chiamato c di tipo Container.

Per queue sono definiti i seguenti operatori di confronto:

```
==, <, <=, !=, >, >=
```

La classe queue contiene le seguenti funzioni membro.

| MEMBRO | DESCRIZIONE |
|---|---|
| `value_type &back( );`<br>`const value_type &back( ) const;` | Restituisce l'indirizzo dell'ultimo elemento della coda. |
| `bool empty( ) const;` | Restituisce true se la coda chiamante è vuota altrimenti restituisce false. |
| `value_type &front( );`<br>`const value_type &front( ) const;` | Restituisce l'indirizzo del primo elemento della coda. |
| `void pop( );` | Rimuove il primo elemento della coda. |
| `void push(const T &val);` | Aggiunge alla fine della coda un elemento con il valore specificato da *val*. |
| `size_type size( ) const;` | Restituisce il numero di elementi attualmente presenti nella coda. |

## La classe set

La classe set supporta un insieme di chiavi univoche. Ecco la sua specifica template:

```
template <class Key, class Comp  less<Key>,
          class Allocator = allocator<Key>> class set
```

Qui Key è il tipo delle chiavi e Comp è una funzione che confronta due chiavi. La classe set ha i seguenti costruttori:

```
explicit set(const Comp &cmpfn = Comp( ),
             const Allocator &a = Allocator( ) );

set(const set<Key, Comp, Allocator> &ob);

template <class InIter> set(InIter start, InIter end,
        const Comp &cmpfn = Comp( ),
        const Allocator &a = Allocator( ));
```

La prima forma costruisce un insieme vuoto. La seconda forma costruisce un insieme che contiene gli stessi elementi di *ob*. La terza forma costruisce un insieme che contiene gli elementi dell'intervallo specificato da *start* ed *end*. L'eventuale ordinamento dell'insieme è determinato dalla presenza della funzione *cmpfn*.

Per la classe set sono definiti i seguenti operatori di confronto:

==, <, <=, !=, >, >=

La classe set contiene le seguenti funzioni membro:



| MEMBRO | DESCRIZIONE |
|---|---|
| `iterator begin( );`<br>`const_iterator begin( ) const;` | Restituisce un iteratore al primo elemento dell'insieme. |
| `void clear( );` | Rimuove tutti gli elementi dall'insieme. |
| `size_type count(const key_type &k) const;` | Restituisce il numero di volte che *k* si presenta nell'insieme. |
| `bool empty( ) const;` | Restituisce true se l'insieme chiamante è vuoto altrimenti restituisce false. |
| `const_iterator end( ) const;`<br>`iterator end( );` | Restituisce un iteratore alla fine dell'insieme. |
| `pair<iterator, iterator>`<br>`equal_range(const key_type &k) const;` | Restituisce una coppia di iteratori che puntano al primo e all'ultimo elemento dell'insieme che contengono la chiave specificata. |
| `void erase(iterator i);` | Rimuove l'elemento puntato da *i*. |
| `void erase(iterator start, iterator end);` | Rimuove gli elementi dell'intervallo compreso fra *start* ed *end*. |
| `size_type erase(const key_type &k);` | Rimuove dall'insieme gli elementi la cui chiave è uguale al valore *k*. La funzione restituisce il numero di elementi rimossi. |
| `iterator find(const key_type &k) const;` | Restituisce un iteratore alla chiave specificata. Se la chiave non viene trovata, restituisce un iteratore alla fine dell'insieme. |
| `allocator_type get_allocator( ) const;` | Restituisce l'allocatore dell'insieme. |
| `iterator insert(iterator i,`<br>`const value_type &val);` | Inserisce *val* nella posizione o subito dopo l'elemento specificato da *i*. Gli elementi duplicati non vengono inseriti. Restituisce l'iteratore dell'elemento. |
| `template <class InIter>`<br>`void insert(InIter start, InIter end);` | Inserisce un intervallo di elementi. Gli elementi duplicati non vengono inseriti. |
| `pair<iterator, bool>`<br>`insert(const value_type &val);` | Inserisce *val* nell'insieme chiamante. Restituisce l'iteratore dell'elemento. L'elemento viene inserito solo se non esiste ancora. Se l'elemento viene inserito, la funzione restituisce `pair<iterator, true>`. Altrimenti restituisce `pair<iterator,< false>`. |
| `iterator lower_bound(const key_type &k)`<br>`const;` | Restituisce un iteratore al primo elemento dell'insieme la cui chiave è maggiore o uguale a *k*. |
| `key_compare key_comp( ) const;` | Restituisce l'oggetto funzione che confronta le chiavi. |
| `size_type max_size( ) const;` | Restituisce il numero massimo di elementi che possono essere contenuti nell'insieme. |
| `reverse_iterator rbegin( );`<br>`const_reverse_iterator rbegin( ) const;` | Restituisce un iteratore inverso alla fine dell'insieme. |

*(segue)*

*(continua)*

| MEMBRO | DESCRIZIONE |
|---|---|
| `reverse_iterator rend( );`<br>`const_reverse_iterator rend( ) const;` | Restituisce un iteratore inverso all'inizio dell'insieme. |
| `size_type size( ) const;` | Restituisce il numero di elementi attualmente presenti nell'insieme. |
| `void swap(set<Key, Comp,Allocator> &ob);` | Scambia gli elementi contenuti nell'insieme chiamante con quelli di *ob*. |
| `iterator upper_bound(const key_type &k) const;` | Restituisce un iteratore al primo elemento dell'insieme la cui chiave è maggiore o uguale a *k*. |
| `value_compare value_comp( ) const;` | Restituisce l'oggetto funzione che confronta i valori. |

## La classe stack

La classe stack supporta uno stack. La sua specifica template è la seguente:

template <class T, class Container deque<T>> class stack

Qui T è il tipo dei dati memorizzati e Container è il tipo del container utilizzato per contenere lo stack. La classe ha il seguente costruttore:

explicit stack(const Container &*cnt* = Container( ));

Il costruttore stack() crea uno stack vuoto. Normalmente come container viene utilizzata una deque ma l'accesso a uno stack può avvenire solo in modo LIFO (Last In First Out). Come container per lo stack si può anche utilizzare un vettore o una lista. Il container è contenuto in un membro protetto chiamato c e di tipo Container.

Per stack sono definiti i seguenti operatori di confronto:

==, <, <=, !=, >, >=

La classe stack contiene le seguenti funzioni membro.



| MEMBRO | DESCRIZIONE |
|---|---|
| `bool empty( ) const;` | Restituisce true se lo stack chiamante è vuoto altrimenti restituisce false. |
| `void pop( );` | Rimuove la cima dello stack, ovvero l'ultimo elemento del container. |
| `void push(const T &val);` | Inserisce (push) un elemento alla fine dello stack. L'ultimo elemento del container rappresenta la cima dello stack. |
| `size_type size( ) const;` | Restituisce il numero di elementi attualmente presenti nello stack. |
| `value_type &top( );`<br>`cont value_type &top( ) const;` | Restituisce l'indirizzo della cima dello stack, ovvero dell'ultimo elemento del container. L'elemento non viene rimosso. |

## La classe vector

La classe vector supporta un array dinamico. Ecco la sua specifica template.

template <class T, class Allocator = allocator<T>> class vector

Qui T è il tipo dei dati memorizzati e Allocator specifica l'allocatore. La classe ha i seguenti costruttori.

explicit vector(const Allocator &*a* = Allocator( ) );

explicit vector(size_type *num*, const T &*val* = T ( ),
const Allocator &*a* = Allocator( ));

vector(const vector<T, llocator> &*ob*);

template <class InIter> vector(InIter *start*, InIter *end*,
const Allocator &a = Allocator( ));

La prima forma costruisce un vettore vuoto. La seconda costruisce un vettore che contiene *num* elementi di valore *val*. La terza forma costruisce un vettore che contiene gli stessi elementi di *ob*. La quarta forma costruisce un vettore che contiene gli elementi dell'intervallo specificato da *start* ed *end*.

Per la classe vector sono definiti i seguenti operatori di confronto.

==, <, <=, !=, >, >=

La classe vector contiene le seguenti funzioni membro.

| MEMBRO | DESCRIZIONE |
|---|---|
| `template <class InIter> void assign(InIter start, InIter end);` | Assegna al vettore la sequenza definita da *start* ed *end*. |
| `void assign(size_type num, const T &val);` | Assegna al vettore *num* elementi di valore *val*. |
| `reference at(size_type i); const_reference at(size_type i) const;` | Restituisce l'indirizzo dell'elemento specificato da *i*. |
| `reference back( ); const_reference back( ) const;` | Restituisce l'indirizzo dell'ultimo elemento del vettore. |
| `iterator begin( ); const_iterator begin( ) const;` | Restituisce un iteratore al primo elemento del vettore. |
| `size_type capacity( ) const;` | Restituisce la capacità attuale del vettore. Questo è il numero di elementi che può contenere prima di dover allocare ulteriore memoria. |
| `void clear( );` | Rimuove tutti gli elementi dal vettore. |
| `bool empty( ) const;` | Restituisce true se il vettore chiamante è vuoto altrimenti restituisce false. |
| `iterator end( ); const_iterator end( ) const;` | Restituisce un iteratore alla fine del vettore. |
| `iterator erase(iterator i);` | Rimuove l'elemento puntato da *i*. Restituisce l'iteratore dell'elemento che segue quello rimosso. |
| `iterator erase(iterator start, iterator end);` | Rimuove gli elementi dell'intervallo compreso fra *start* ed *end*. Restituisce l'iteratore dell'elemento che segue l'ultimo elemento rimosso. |
| `reference front( ); const_reference front( ) const;` | Restituisce l'indirizzo del primo elemento del vettore. |
| `allocator_type get_allocator( ) const;` | Restituisce l'allocatore del vettore. |
| `iterator insert(iterator i, const T &val);` | Inserisce *val* immediatamente prima dell'elemento specificato da *i*. Restituisce l'iteratore dell'elemento. |
| `void insert(iterator i, size_type num, const T & val);` | Inserisce *num* copie di *val* immediatamente prima dell'elemento specificato da *i*. |
| `template <class InIter> void insert(iterator i, InIter start, InIter end);` | Inserisce la sequenza definita da *start* ed *end* immediatamente prima dell'elemento specificato da *i*. |
| `size_type max_size( ) const;` | Restituisce il numero massimo di elementi che il vettore può contenere. |
| `reference operator[ ](size_type i) const; const_reference operator[ ](size_type i) const;` | Restituisce l'indirizzo dell'elemento specificato da *i*. |

*(segue)*



*(continua)*

| MEMBRO | DESCRIZIONE |
|---|---|
| `void pop_back( );` | Rimuove l'ultimo elemento del vettore. |
| `void push_back(const T &val);` | Aggiunge un elemento con il valore specificato da *val* alla fine del vettore. |
| `reverse_iterator rbegin( ); const_reverse_iterator rbegin( ) const;` | Restituisce un iteratore inverso alla fine del vettore. |
| `reverse_iterator rend( ); const_reverse_iterator rend( ) const;` | Restituisce un iteratore inverso all'inizio del vettore. |
| `void reserve(size_type num);` | Imposta la capacità del vettore in modo che sia almeno uguale a *num*. |
| `void resize(size_type num, T val = T ( ));` | Cambia le dimensioni del vettore a quelle specificate da *num*. Se il vettore deve essere allungato, allora gli elementi contenenti il valore specificato da *val* vengono aggiunti alla fine. |
| `size_type size( ) const;` | Restituisce il numero di elementi attualmente presenti nel vettore. |
| `void swap(vector<T, Ilocator> &ob);` | Scambia gli elementi contenuti nel vettore chiamante con quelli di *ob*. |

La libreria STL contiene anche una specializzazione di vector per valori booleani che include tutte le funzionalità di vector ma vi aggiunge due nuovi membri.

| | |
|---|---|
| `void flip( );` | Inverte tutti i bit del vettore. |
| `static void swap(reference i, reference j);` | Scambia i bit specificati da *i* e *j*. |

Capitolo 34

# Gli algoritmi STL

Questo capitolo descrive gli algoritmi definiti dalla libreria STL. Questi algoritmi operano sui container tramite gli iteratori. Tutti gli algoritmi sono funzioni template. Ecco i nomi dei tipi generici utilizzati dagli algoritmi.

| NOME GENERICO | RAPRESENTA |
|---|---|
| `BiIter` | Un iteratore bidirezionale |
| `ForIter` | Un iteratore in avanti |
| `InIter` | Un iteratore di input |
| `OutIter` | Un iteratore di output |
| `RandIter` | Un iteratore di accesso diretto |
| `T` | Un tipo di dati |
| `Size` | Un tipo intero |
| `Func` | Un tipo di funzione |
| `Generator` | Una funzione che genera oggetti |
| `BinPred` | Un predicato binario |
| `UnPred` | Un predicato unario |
| `Comp` | Una funzione di confronto |

## adjacent_find()

```
template <class ForIter>
  ForIter adjacent_find(ForIter start, ForIter end);
template <class ForIter, class BinPred>
  ForIter adjacent_find(ForIter start, ForIter end, BinPred pfn);
```

L'algoritmo adjacent_find() ricerca un elemento adiacente corrispondente nella sequenza specificata da *start* ed *end* e restituisce un iteratore che punta al primo elemento. Se non trova alcuna coppia adiacente, restituisce *end*. La prima versione ricerca elementi equivalenti. La seconda versione consente di specificare un metodo per determinare gli elementi corrispondenti.

## binary_search()

```
template <class ForIter, class T>
  bool binary_search(ForIter start, ForIter end, const T &val);
template <class ForIter, class T, class Comp>
  bool binary_search(ForIter start, ForIter end, const T &val,
Comp cmpfn);
```

L'algoritmo binary_search() esegue una ricerca binaria su una sequenza ordinata che inizia da *start* e termina in *end* per trovare il valore specificato da *val*. L'algoritmo restituisce true se *val* viene trovato e false altrimenti. La prima versione confronta gli elementi contenuti nella sequenza specificata. La seconda versione consente di specificare una funzione di confronto.

## copy()

```
template <class InIter, class OutIter>
  OutIter copy(InIter start, InIter end, OutIter result);
```

L'algoritmo copy() copia la sequenza che inizia da *start* e termina in *end* inserendo il risultato nella sequenza puntata da *result*. L'algoritmo restituisce un iteratore che punta alla fine della sequenza risultante. L'intervallo da copiare non deve sovrapporsi a *result*.

## copy_backward()

```
template <class BiIter1, class BiIter2>
  BiIter2 copy_backward(BiIter1 start, BiIter1 end, BiIter2 result);
```



L'algoritmo copy_backward() è uguale a copy() tranne per il fatto che trasferisce gli elementi partendo dalla fine della sequenza.

## count()

```
template <class InIter, class T>
  size_t count(InIter start, InIter end, const T &val);
```

L'algoritmo count() restituisce il numero di elementi contenuti nella sequenza compresa fra *start* ed *end* che corrispondono al valore *val*.

## count_if()

```
template <class InIter, class nPred>
  size_t count(InIter start, InIter end, UnPred pfn);
```

L'algoritmo count_if() restituisce il numero di elementi della sequenza compresa fra *start* ed *end* per i quali il predicato unario *pfn* restituisce true.

## equal()

```
template <class InIter1, class InIter2>
  bool equal(InIter1 start1, InIter1 end1, InIter2 start2);
template <class InIter1, class InIter2, class BinPred>
  bool equal(InIter1 start1, InIter1 end1, InIter2 start2,
                          BinPred pfn);
```

L'algoritmo equal() determina quando due intervalli sono uguali. L'intervallo determinato da *start1* ed *end1* viene confrontato con la sequenza puntata da *start2*. Se gli intervalli sono uguali, l'algoritmo restituisce true, altrimenti restituisce false.

La seconda forma consente di specificare un predicato binario che determina l'uguaglianza di due elementi.

## equal_range()

```
template <class ForIter, class T>
  pair<ForIter, ForIter> equal_range(ForIter start, ForIter end,
                                     const T &val);
template <class ForIter, class T, class Comp>
  pair<ForIter, ForIter> equal_range(ForIter start, ForIter end,
                                     const T &val, Comp cmpfn);
```

– L'algoritmo equal_range() restituisce un intervallo tale per cui un elemento possa essere inserito in una sequenza senza alterarne l'ordine. La regione in cui deve essere eseguita è specificata dall'intervallo compreso fra *start* ed *end*. Il valore viene passato in *val*. Per specificare i criteri di ricerca basta indicare la funzione di confronto *cmpfn*.

La classe template pair è una classe di servizio che nei suoi membri first e second può contenere una coppia di oggetti.

## fill() e fill_n()

```
template <class ForIter, class T>
  void fill(ForIter start, ForIter end, const T &val);
template <class ForIter, class Size, class T>
  void fill_n(ForIter start, Size num, const T &val);
```

Gli algoritmi fill() e fill_n() riempiono un intervallo con il valore specificato da *val*. Per fill() l'intervallo è specificato da *start* ed *end*. Per fill_n() l'intervallo inizia da *start* e si estende per *num* elementi.

## find()

```
template <class InIter, class T>
  InIter find(InIter start, InIter end, const T &val);
```

L'algoritmo find() ricerca nell'intervallo compreso fra *start* ed *end* il valore specificato da *val*. L'algoritmo restituisce un iteratore che punta alla prima occorrenza dell'elemento oppure *end* se il valore non è contenuto nella sequenza.

## find_end()

```
template <class ForIter1, class ForIter2>
  FwdIter1 find_end(ForIter1 start1, ForIter1 end1,
                    ForIter2 start2, ForIter2 end2);
template <class ForIter1, class ForIter2, class BinPred>
  FwdIter1 find_end(ForIter1 start1, ForIter1 end1,
                    ForIter2 start2, ForIter2 end2, BinPred pfn);
```

L'algoritmo trova l'ultimo iteratore della sequenza definita da *start2* ed *end2* nell'intervallo definito da *start1* ed *end1*. Se la sequenza viene trovata, l'algoritmo restituisce un iteratore che punta all'ultimo elemento della sequenza, altrimenti restituisce l'iteratore *end1*.

La seconda forma consente di specificare un predicato binario che determina la corrispondenza degli elementi.



## find_first_of()

```
template <class ForIter1, class ForIter2>
  FwdIter1 find_first_of(ForIter1 start1, ForIter1 end1,
                         ForIter2 start2, ForIter2 end2);
template <class ForIter1, class ForIter2, class BinPred>
  FwdIter1 find_first_of(ForIter1 start1, ForIter1 end1,
                         ForIter2 start2, ForIter2 end2,
                         BinPred pfn);
```

L'algoritmo find_first_of() trova il primo elemento nella sequenza definita da *start1* ed *end1* che corrisponde a un elemento dell'intervallo compreso fra *start2* ed *end2*. Se non viene trovato un elemento corrispondente, l'algoritmo restituisce l'iteratore *end1*.

La seconda forma consente di specificare un predicato binario che determina quando due elementi corrispondono.

## find_if()

```
template <class InIter, class UnPred>
  InIter find_if(InIter start, InIter end, UnPred pfn);
```

L'algoritmo find_if() ricerca nell'intervallo compreso fra *start* ed *end* un elemento per il quale il predicato unario *pfn* restituisce true. L'algoritmo restituisce un iteratore che punta alla prima occorrenza dell'elemento oppure a *end* se il valore non è contenuto nella sequenza.

## for_each()

```
template<class InIter, class Func>
  Func for_each(InIter start, InIter end, Func fn);
```

L'algoritmo for_each() applica la funzione *fn* a tutti gli elementi dell'intervallo compreso fra *start* ed *end* e restituisce *fn*.

## generate() e generate_n()

```
template <class ForIter, class Generator>
  void generate(ForIter start, ForIter end, Generator fngen);
template <class ForIter, class Size, class Generator>
  void generate_n(OutIter start, Size num, Generator fngen);
```

Gli algoritmi generate() e generate_n() assegnano agli elementi di un intervallo il valore restituito da una funzione. Per generate() l'intervallo in cui eseguire l'assegnamento è specificato da *start* ed *end*. Per generate_n() l'intervallo inizia da *start* e procede per *num* elementi. La funzione generatore (senza parametri) viene passata in *fngen*.

## includes()

```
template <class InIter1, class InIter2>
  bool includes(InIter1 start1, InIter1 end1,
                InIter2 start2, InIter2 end2);
template <class InIter1, class InIter2, class Comp>
  bool includes(InIter1 start1, InIter1 end1,
                InIter2 start2, InIter2 end2, Comp cmpfn);
```

L'algoritmo includes() determina se la sequenza definita da *start1* ed *end1* include tutti gli elementi della sequenza definita da *start2* ed *end2*. Se gli elementi vengono tutti trovati restituisce true, altrimenti restituisce false.

La seconda forma consente di specificare una funzione di confronto che determina quando un elemento è minore di un altro.

## inplace_merge()

```
template <class BiIter>
  void inplace_merge(BiIter start, BiIter mid, BiIter end);
template <class BiIter, class Comp>
  void inplace_merge(BiIter start, BiIter mid, BiIter end, Comp cmpfn);
```

In una singola sequenza, l'algoritmo inplace_merge() unisce l'intervallo definito da *start* e *mid* con quello definito da *mid* ed *end*. Entrambi gli intervalli devono essere ordinati in senso ascendente. Dopo l'esecuzione, la sequenza prodotta sarà ordinata in senso ascendente.

La seconda forma consente di specificare una funzione di confronto che determina quando un elemento è minore di un altro.



## iter_swap()

```
template <class ForIter1, class ForIter2>
  void iter_swap(ForIter1 i, ForIter2 j)
```

L'algoritmo iter_swap() scambia il valore puntato dai due iteratori forniti come argomenti.

## lexicographical_compare()

```
template <class InIter1, class InIter2>
  bool lexicographical_compare(InIter1 start1, InIter1 end1,
                               InIter2 start2, InIter2 end2);
template <class InIter1, class InIter2, class Comp>
  bool lexicographical_compare(InIter1 start1, InIter1 end1,
                               InIter2 start2, InIter2 end2,
                               Comp cmpfn);
```

L'algoritmo lexicographical_compare() confronta alfabeticamente la sequenza definita da *start1* ed *end1* con la sequenza definita da *start2* ed *end2*. Se la prima sequenza è minore in senso lessicale rispetto alla seconda (ovvero se la prima sequenza precede nell'ordine "a dizionario") l'algoritmo restituisce true.

La seconda forma consente di specificare una funzione di confronto che determina quando un elemento è minore di un altro.

## lower_bound()

```
template <class ForIter, class T>
  ForIter lower_bound(ForIter start, ForIter end, const T &val);
template <class ForIter, class T, class Comp>
  ForIter lower_bound(ForIter start, ForIter end, const T &val,
                      Comp cmpfn);
```

L'algoritmo lower_bound() trova il primo punto della sequenza definita da *start* ed *end* che non è minore di *val*.

Quindi restituisce un iteratore a tale punto.

La seconda forma consente di specificare una funzione di confronto che determina quando un elemento è minore di un altro.

## make_heap()

```
template <class RandIter>
  void make_heap(RandIter start, RandIter end);
template <class RandIter, class Comp>
  void make_heap(RandIter start, RandIter end, Comp cmpfn);
```

L'algoritmo make_heap() costruisce uno heap partendo dalla sequenza definita da *start* ed *end*

La seconda forma consente di specificare una funzione di confronto che determina quando un elemento è minore di un altro.

## max()

```
template <class T>
  const T &max(const T &i, const T &j);
template <class T, class Comp>
  const T &max(const T &i, const T &j, Comp cmpfn);
```

L'algoritmo max() restituisce il maggiore di due valori.

La seconda forma consente di specificare una funzione di confronto che determina quando un elemento è minore di un altro.

## max_element()

```
template <class ForIter>
  ForIter max_element(ForIter start, ForIter last);
template <class ForIter, class Comp>
  ForIter max_element(ForIter start, ForIter last, Comp cmpfn);
```

L'algoritmo max_element() restituisce un iteratore all'elemento maggiore dell'intervallo definito da *start* e *last*.

La seconda forma consente di specificare una funzione di confronto che determina quando un elemento è minore di un altro.

## merge()

```
template <class InIter1, class InIter2, class OutIter>
  OutIter merge(InIter1 start1, InIter1 end1,
                InIter2 start2, InIter2 end2,
                OutIter result);
```



```
template <class InIter1, class InIter2, class OutIter, class Comp>
  OutIter merge(InIter1 start1, InIter1 end1,
                InIter2 start2, InIter2 end2,
                OutIter result, Comp cmpfn);
```

L'algoritmo merge() unisce due sequenze ordinate inserendo il risultato in una terza sequenza. Le sequenze da unire sono definite da *start1* ed *end1* e da *start2* ed *end2*.

Il risultato viene inserito nella sequenza puntata da *result*. L'algoritmo restituisce un iteratore che punta alla fine della sequenza risultante.

La seconda forma consente di specificare una funzione di confronto che determina quando un elemento è minore di un altro.

## min()

```
template <class T>
  const T &min(const T &i, const T &j);
template <class T, class Comp>
  const T &min(const T &i, const T &j, Comp cmpfn);
```

L'algoritmo min() restituisce il minore di due valori.

La seconda forma consente di specificare una funzione di confronto che determina quando un elemento è minore di un altro.

## min_element()

```
template <class ForIter>
  ForIter min_element(ForIter start, ForIter last);
template <class ForIter, class Comp>
  ForIter min_element(ForIter start, ForIter last, Comp cmpfn);
```

L'algoritmo max_element() restituisce un iteratore all'elemento minore dell'intervallo definito da *start* e *last*.

La seconda forma consente di specificare una funzione di confronto che determina quando un elemento è minore di un altro.

## mismatch()

```
template <class InIter1, class InIter2>
  pair<InIter1, InIter2> mismatch(InIter1 start1, InIter1 end1,
                                  InIter2 start2);
```

```
template <class InIter1, class InIter2, class BinPred>
  pair<InIter1, InIter2> mismatch(InIter1 start1, InIter1 end1,
                                  InIter2 start2, BinPred pfn);
```

L'algoritmo mismatch() trova la prima differenza fra gli elementi di due sequenze. L'algoritmo restituisce gli iteratori ai due elementi. Se non viene trovata alcuna differenza, vengono restituiti gli iteratori che puntano all'ultimo elemento di entrambe le sequenze.

La seconda forma consente di specificare un predicato unario che determina quando un elemento è uguale a un altro.

La classe template pair contiene due dati membro chiamati *first* e *second* che contengono le coppie di valori.

## next_permutation()

```
template <class BiIter>
  bool next_permutation(BiIter start, BiIter end);
template <class BiIter, class Comp>
  bool next_permutation(BiIter start, BiIter end, Comp cmfn);
```

L'algoritmo next_permutation() costruisce la permutazione successiva in una sequenza. Le permutazioni vengono generate partendo da una sequenza ordinata: la prima permutazione è rappresentata da una sequenza dal più basso al più alto. Se la permutazione successiva non esiste, next_permutation() ordina la sequenza come la sua prima permutazione e restituisce false. Altrimenti restituisce true.

La seconda forma consente di specificare una funzione di confronto che determina quando un elemento è minore di un altro.

## nth_element()

```
template <class RandIter>
  void nth_element(RandIter start, RandIter element, RandIter end);
template <class RandIter, class Comp>
  void nth_element(RandIter start, RandIter element,
                   RandIter end, Comp cmpfn);
```

L'algoritmo nth_element() dispone la sequenza specificata da *start* ed *end* in modo che tutti gli elementi minori di *element* precedano tale elemento e tutti gli elementi maggiori.

La seconda forma consente di specificare una funzione di confronto che determina quando un elemento è minore di un altro.



## partial_sort()

```
template <class RandIter>
  void partial_sort(RandIter start, RandIter mid, RandIter end);
template <class RandIter, class Comp>
  void partial_sort(RandIter start, RandIter mid,
                    RandIter end, Comp cmpfn);
```

L'algoritmo partial_sort() ordina l'intervallo compreso fra *start* ed *end*. Dopo l'esecuzione, saranno in ordine i soli elementi compresi nell'intervallo compreso fra *start* e *mid*.

La seconda forma consente di specificare una funzione di confronto che determina quando un elemento è minore di un altro.

## partial_sort_copy()

```
template <class InIter, class RandIter>
  RandIter partial_sort_copy(InIter start, InIter end,
                             RandIter res_start, RandIter res_end);
template <class InIter, class RandIter, class Comp>
  RandIter partial_sort_copy(InIter start, InIter end,
                             RandIter res_start, RandIter res_end,
                             Comp cmpfn);
```

L'algoritmo partial_sort_copy() ordina l'intervallo definito da *start* ed *end* e poi copia un numero di elementi tali da rientrare nella sequenza di destinazione definita da *res_start* e *res_end*. L'algoritmo restituisce un iteratore che punta all'ultimo elemento copiato della sequenza risultante.

La seconda forma consente di specificare una funzione di confronto che determina quando un elemento è minore di un altro.

## partition()

```
template <class BiIter, class UnPred>
  BiIter partition(BiIter start, BiIter end, UnPred pfn);
```

L'algoritmo partition() dispone la sequenza definita da *start* ed *end* in modo che tutti gli elementi per i quali il predicato specificato da *pfn* restituisce true precedano quelli per i quali il predicato restituisce false. Poi restituisce un iteratore che punta al primo degli elementi per i quali il predicato è false.

## pop_heap()

```
template <class RandIter>
  void pop_heap(RandIter start, RandIter end);
template <class RandIter, class Comp>
  void pop_heap(RandIter start, RandIter end, Comp cmpfn);
```

L'algoritmo pop_heap() scambia gli elementi *first* e *last* - 1 e poi ricostruisce lo heap.

La seconda forma consente di specificare una funzione di confronto che determina quando un elemento è minore di un altro.

## prev_permutation()

```
template <class BiIter>
  bool prev_permutation(BiIter start, BiIter end);
template <class BiIter, class Comp>
  bool prev_permutation(BiIter start, BiIter end, Comp cmpfn);
```

L'algoritmo prev_permutation() costruisce la permutazione precedente in una sequenza. Le permutazioni vengono generate partendo da una sequenza ordinata: la prima permutazione è rappresentata da una sequenza dal più basso al più alto. Se la permutazione precedente non esiste, prev_permutation() ordina la sequenza come la sua ultima permutazione e restituisce false. Altrimenti restituisce true.

La seconda forma consente di specificare una funzione di confronto che determina quando un elemento è minore di un altro.

## push_heap()

```
template <class RandIter>
  void push_heap(RandIter start, RandIter end);
template <class RandIter, class Comp>
  void push_heap(RandIter start, RandIter end, Comp cmpfn);
```

L'algoritmo push_heap() inserisce un elemento alla fine dello heap. L'intervallo specificato da *start* ed *end* deve rappresentare uno heap valido.

La seconda forma consente di specificare una funzione di confronto che determina quando un elemento è minore di un altro.

## random_shuffle()

```
template <class RandIter>
  void random_shuffle(RandIter start, RandIter end);
template <class RandIter, class Generator>
  void random_shuffle(RandIter start, RandIter end, Generator rand_gen);
```

L'algoritmo random_shuffle() rende casuale la sequenza definita da *start* ed *end*.

La seconda forma specifica un generatore personalizzato di numeri casuali. Questa funzione deve avere la seguente forma generale:

rand_gen(*num*);

Tale funzione deve restituire un numero casuale compreso fra 0 e *num*.

## remove(), remove_if(), remove_copy() e remove_copy_if()

```
template <class ForIter, class T>
  ForIter remove(ForIter start, ForIter end, const T &val);
template <class ForIter, class UnPred>
  ForIter remove_if(ForIter start, ForIter end,  UnPred pfn);
template <class ForIter, class OutIter, class T>
  OutIter remove_copy(InIter start, InIter end,
                      OutIter result, const T &val);
template <class ForIter, class OutIter, class UnPred>
  OutIter remove_copy_if(InIter start, InIter end,
                         OutIter result, UnPred pfn);
```

L'algoritmo remove() rimuove dall'intervallo specificato gli elementi uguali a *val*. Restituisce un iteratore che punta alla fine degli elementi rimanenti.

L'algoritmo remove_if() rimuove dall'intervallo gli elementi per i quali il predicato *pfn* è true. Restituisce un iteratore che punta alla fine degli elementi rimanenti.

L'algoritmo remove_copy() copia dall'intervallo specificato gli elementi uguali a *val* e inserisce il risultato nella sequenza puntata da *result*. Restituisce un iteratore che punta alla fine del risultato.

L'algoritmo remove_copy_if() copia dall'intervallo specificato gli elementi per i quali il predicato *pfn* è true e inserisce il risultato nella sequenza puntata da *result*. Restituisce un iteratore che punta alla fine del risultato.

## replace(), replace_if(), replace_copy() e replace_copy_if()

```
template <class ForIter, class T>
  void replace(ForIter start, ForIter end,
               const T &old, const T &new);
template <class ForIter, class UnPred, class T>
  void replace_if(ForIter start, ForIter end,
                  UnPred pfn, const T &new);
template <class ForIter, class OutIter, class T>
  OutIter replace_copy(InIter start, InIter end, OutIter result,
                       const T &old, const T &new);
template <class ForIter, class OutIter, class UnPred, class T>
  OutIter replace_copy_if(InIter start, InIter end, OutIter result,
                          UnPred pfn, const T &new);
```

L'algoritmo replace() sostituisce dall'intervallo specificato gli elementi uguali a *old* con gli elementi uguali a *new*.

L'algoritmo replace_if() sostituisce dall'intervallo specificato gli elementi per i quali il predicato *pfn* è true con gli elementi il cui valore è *new*.

L'algoritmo replace_copy() copia dall'intervallo specificato gli elementi uguali a *old*, li sostituisce con quelli il cui valore è *new* e inserisce il risultato nella sequenza puntata da *result*. L'intervallo originale non viene modificato. Restituisce un iteratore che punta alla fine del risultato.

L'algoritmo replace_copy_if() copia dall'intervallo specificato gli elementi per i quali il predicato *pfn* è true, li sostituisce con quelli il cui valore è *new* e inserisce il risultato nella sequenza puntata da *result*. L'intervallo originale non viene modificato. Restituisce un iteratore che punta alla fine del risultato.

## reverse() e reverse_copy()

```
template <class BiIter>
  void reverse(BiIter start, BiIter end);
template <class BiIter, class OutIter>
  OutIter reverse_copy(BiIter first, BiIter last, OutIter result);
```

L'algoritmo reverse() inverte l'ordine dell'intervallo specificato da *start* ed *end*.

L'algoritmo reverse_copy() copia in ordine inverso l'intervallo specificato da *start* ed *end* e memorizza il risultato in *result*. Restituisce un iteratore che punta alla fine di *result*.



## rotate() e rotate_copy()

```
template <class ForIter>
  void rotate(ForIter start, ForIter mid, ForIter end);
template <class ForIter, class OutIter>
  OutIter rotate_copy(ForIter start, ForIter mid, ForIter end,
OutIter result);
```

L'algoritmo rotate() esegue una rotazione a sinistra degli elementi dell'intervallo specificato da *start* ed *end* in modo che l'elemento *mid* divenga il nuovo primo elemento.

L'algoritmo rotate_copy() copia l'intervallo specificato da *start* ed *end*, memorizzando il risultato in *result*. Nell'eseguire questa operazione, esegue una rotazione a sinistra degli elementi in modo che l'elemento *mid* divenga il primo elemento. Restituisce un iteratore che punta alla fine di *result*.

## search()

```
template <class ForIter1, class ForIter2>
  ForIter1 search(ForIter1 start1, ForIter1 end1,
                  ForIter2 start2, ForIter2 end2);
template <class ForIter1, class ForIter2, class iUnPred>
  ForIter1 search(ForIter1 start1, ForIter1 end1,
                  ForIter2 start2, ForIter2 end2, BinPred pfn);
```

L'algoritmo search() ricerca una sottosequenza in una sequenza. La sequenza in cui eseguire la ricerca è definita da *start1* ed *end1*. La sottosequenza ricercata è specificata da *start2* ed *end2*. Se viene trovata la sottosequenza, l'algoritmo restituisce un iteratore all'inizio di tale sequenza, altrimenti restituisce *end1*.

La seconda forma consente di specificare un predicato binario che determina quando un elemento è uguale a un altro.

## search_n()

```
template <class ForIter, class Size, class T>
  ForIter search_n(ForIter start, ForIter end,
                   Size num, const T &val);
template <class ForIter, class Size, class T, class BinPred>
  ForIter search_n(ForIter start, ForIter end,
                   Size num, const T &val, BinPred pfn);
```

L'algoritmo search_n() ricerca in una sequenza una sottosequenza di *num* elementi il cui valore è *val*. La sequenza in cui eseguire la ricerca è definita da *start1* ed *end1*. Se viene trovata la sottosequenza, l'algoritmo restituisce un iteratore all'inizio di tale sequenza, altrimenti restituisce *end1*.

La seconda forma consente di specificare un predicato binario che determina quando un elemento è uguale a un altro.

## set_difference()

```
template <class InIter1, class InIter2, class OutIter>
  OutIter set_difference(InIter1 start1, InIter1 end1,
          InIter2 start2, InIter2 last2, OutIter result);
template <class InIter1, class InIter2, class OutIter, class Comp>
  OutIter set_difference(InIter1 start1, InIter1 end1,
          InIter2 start2, InIter2 last2,
          OutIter result, Comp cmpfn);
```

L'algoritmo set_difference() produce una sequenza che contiene la differenza fra i due insiemi ordinati definiti da *start1* ed *end1* e *start2* ed *end2*. In pratica sottrae l'insieme definito da *start2* ed *end2* dall'insieme definito da *start1* ed *end1*. Il risultato viene ordinato e inserito in *result*. L'algoritmo restituisce un iteratore che punta alla fine del risultato.

La seconda forma consente di specificare un predicato binario che determina quando un elemento è minore di un altro.

## set_intersection()

```
template <class InIter1, class InIter2, class OutIter>
  OutIter set_iInIntersection(InIter1 start1, InIter1 end1,
          InIter2 start2, InIter2 last2, OutIter result);
template <class InIter1, class InIter2, class OutIter, class Comp>
  OutIter set_iInIntersection(InIter1 start1, InIter1 end1,
          InIter2 start2, InIter2 last2,
          OutIter result, Comp cmpfn);
```

L'algoritmo set_ intersection () produce una sequenza che contiene l'intersezione fra i due insiemi ordinati definiti da *start1* ed *end1* e *start2* ed *end2*. In pratica si tratta degli elementi comuni ai due insiemi. Il risultato viene ordinato e inserito in *result*. L'algoritmo restituisce un iteratore che punta alla fine del risultato.

La seconda forma consente di specificare un predicato binario che determina quando un elemento è minore di un altro.



## set_simmetric_difference()

```
template <class InIter1, class InIter2, class OutIter>
  OutIter set_symmetric_difference(InIter1 start1, InIter1 end1,
          InIter2 start2, InIter2 last2, OutIter result);
template <class InIter1, class InIter2, class OutIter, class Comp>
  OutIter set_symmetric_difference(InIter1 start1, InIter1 end1,
          InIter2 start2, InIter2 last2, OutIter result,
Comp cmpfn);
```

L'algoritmo set_simmetric_difference() produce una sequenza che contiene la differenza simmetrica fra i due insiemi ordinati definiti da *start1* ed *end1* e *start2* ed *end2*. In pratica l'insieme risultante contiene gli elementi che non sono comuni ai due insiemi. Il risultato viene ordinato e inserito in *result*. L'algoritmo restituisce un iteratore che punta alla fine del risultato.

La seconda forma consente di specificare un predicato binario che determina quando un elemento è minore di un altro.

## set_union()

```
template <class InIter1, class InIter2, class OutIter>
  OutIter set_union(InIter1 start1, InIter1 end1,
          InIter2 start2, InIter2 last2, OutIter result);
template <class InIter1, class InIter2, class OutIter, class Comp>
  OutIter set_union(InIter1 start1, InIter1 end1,
          InIter2 start2, InIter2 last2, OutIter result,
Comp cmpfn);
```

L'algoritmo set_union() produce una sequenza che contiene l'unione di due insiemi ordinati definiti da *start1* ed *end1* e *start2* ed *end2*. In pratica l'insieme risultante contiene tutti gli elementi dei due insiemi. Il risultato viene ordinato e inserito in *result*. L'algoritmo restituisce un iteratore che punta alla fine del risultato.

La seconda forma consente di specificare un predicato binario che determina quando un elemento è minore di un altro.

## sort()

```
template <class RandIter>
  void sort(RandIter start, RandIter end);
template <class RandIter, classComp>
  void sort(RandIter start, RandIter end, Comp cmpfn);
```

L'algoritmo sort() ordina l'intervallo specificato da *start* ed *end*.

La seconda forma consente di specificare un predicato binario che determina quando un elemento è minore di un altro.

## sort_heap()

```
template <class RandIter>
  void sort_heap(RandIter start, RandIter end);
template <class RandIter, class Comp>
  void sort_heap(RandIter start, RandIter end, Comp cmpfn);
```

L'algoritmo sort_heap() ordina lo heap nell'intervallo specificato da *start* ed *end*.

La seconda forma consente di specificare un predicato binario che determina quando un elemento è minore di un altro.

## stable_partition()

```
template <class BiIter, class UnPred>
  BiIter stable_partition(BiIter start, BiIter end, UnPred pfn);
```

L'algoritmo stable_partition() dispone la sequenza definita da *start* ed *end* in modo che tutti gli elementi per i quali il predicato specificato da *pfn* restituisce true precedano quelli per i quali il predicato restituisce false. Il partizionamento è stabile, ovvero viene mantenuto l'ordinamento relativo della sequenza. Restituisce un iteratore che punta all'inizio degli elementi per i quali il predicato è false.

## stable_sort()

```
template <class RandIter>
  void stable_sort(RandIter start, RandIter end);
template <class RandIter, class Comp>
  void stable_sort(RandIter start, RandIter end, Comp cmpfn);
```

L'algoritmo stable_sort() ordina l'intervallo specificato da *start* ed *end*. L'ordinamento è stabile, ovvero viene mantenuto l'ordinamento relativo dell'intervallo.

La seconda forma consente di specificare un predicato binario che determina quando un elemento è minore di un altro.



## swap()

```
template <class T>
  void swap(T &i, T &j);
```

L'algoritmo swap() scambia i valori indicati da *i* e *j*.

## swap_ranges()

```
template <class ForIter1, class ForIter2>
  ForIter2 swap_ranges(ForIter1 start1, ForIter1 end1,
ForIter2 start2);
```

L'algoritmo swap_ranges() scambia gli elementi dell'intervallo specificato da *start1* ed *end1* con gli elementi della sequenza che parte da *start2*. Restituisce un iteratore che punta alla fine della sequenza specificata da *start2*.

## transform()

```
template <class InIter, class OutIter, class Func>
  OutIter transform(InIter start, InIter end,
                    OutIter result, Func unaryfunc);
template <class InIter1, class InIter2, class OutIter, class Func>
  OutIter transform(InIter1 start1, InIter1 end1,
                    InIter2 start2, OutIter result,
                    Func binaryfunc);
```

L'algoritmo transform() applica una funzione a un intervallo di elementi e memorizza il risultato in *result*. Nella prima forma l'intervallo è specificato da *start* ed *end*. *unaryfunc* specifica la funzione unaria da applicare. Questa funzione riceve il valore di un elemento nel suo parametro e deve restituire la sua trasformazione.

Nella seconda forma la trasformazione viene applicata impiegando una funzione binaria che riceve nel primo parametro l'elemento della sequenza che deve essere trasformato e nel secondo parametro un elemento della seconda sequenza.

Entrambe le versioni restituiscono un iteratore che punta alla fine della sequenza risultante.

### unique() e unique_copy()

```
template <class ForIter>
  ForIter unique(ForIter start, ForIter end);
template <class ForIter, class BinPred>
  ForIter unique(ForIter start, ForIter end, BinPred pfn);
template <class ForIter, class OutIter>
  OutIter unique_copy(ForIter start, ForIter end, OutIter result);
template <class ForIter, class OutIter, class BinPred>
  OutIter unique_copy(ForIter start, ForIter end, OutIter result,
BinPred pfn);
```

L'algoritmo unique() elimina dall'intervallo specificato gli elementi duplicati. La seconda forma consente di specificare un predicato binario che determina l'uguaglianza degli elementi. L'algoritmo unique () restituisce un iteratore che punta alla fine dell'intervallo.

L'algoritmo unique_copy() copia l'intervallo specificato da *start* ed *end* eliminando gli elementi duplicati. Il risultato viene inserito in *result*. La seconda forma consente di specificare un predicato binario che determina l'uguaglianza degli elementi. L'algoritmo unique_copy() restituisce un iteratore che punta alla fine dell'intervallo.

### upper_bound()

```
template <class ForIter, class T>
  ForIter upper_bound(ForIter start, ForIter end, const T &val);
template <class ForIter, class T, class Comp>
  ForIter upper_bound(ForIter start, ForIter end, const T &val,
                      Comp cmpfn);
```

L'algoritmo upper_bound() trova l'ultimo punto della sequenza definita da *start* ed *end* che non è maggiore di *val*. Quindi restituisce un iteratore a tale punto.

La seconda forma consente di specificare una funzione di confronto che determina quando un elemento è minore di un altro.

Capitolo 35

# Iteratori, allocatori e oggetti funzione STL

35.1 **Gli iteratori**
35.2 **Gli oggetti funzione**
35.3 **Gli allocatori**

Questo capitolo descrive le classi e le funzioni che supportano gli iteratori, gli allocatori e gli oggetti funzione. Questi componenti fanno parte della libreria STL e possono essere utilizzati anche per altri scopi.

## 35.1 Gli iteratori

Mentre i container e gli algoritmi costituiscono le basi della libreria STL, gli iteratori sono come la "colla" che li tiene uniti. Un iteratore è una generalizzazione (o più precisamente un'astrazione) di un puntatore. Gli iteratori vengono gestiti nel programma come se fossero puntatori e implementano gli operatori standard per i puntatori. Essi offrono la possibilità di scorrere il contenuto di un container così come si usa un puntatore per scorrere il contenuto di un array.

Il linguaggio C++ standard definisce un insieme di classi e funzioni che supportano gli iteratori. Tuttavia per la maggior parte delle operazioni di programmazione che impiegano la libreria STL tali classi non vengono utilizzate direttamente. Al contrario si usano gli iteratori forniti dai vari container STL manipolandoli come qualsiasi altro puntatore. Nonostante tutto quello che si è detto finora, è fondamentale conoscere il funzionamento delle classi per iteratori e il loro contenuto. Ad esempio, questo consente di creare degli iteratori specifici per rispondere a particolari situazioni. Inoltre gli sviluppatori di librerie troveranno molto utili le classi per iteratori.

Gli iteratori usano l'header <iterator>.

### I tipi base per gli iteratori

Esistono cinque tipi base per gli iteratori.

| ITERATORE | ACCESSO CONSENTITO |
|---|---|
| Accesso diretto | Memorizza e legge il puntatore. L'accesso agli elementi può avvenire in modo casuale. |
| Bidirezionale | Memorizza e legge il puntatore. Spostamento in avanti e indietro. |
| In avanti | Memorizza e legge il puntatore. Solo spostamento in avanti. |
| Input | Legge ma non memorizza il puntatore. Solo spostamento in avanti. |
| Output | Memorizza ma non legge il puntatore. Solo spostamento in avanti. |

In generale, un iteratore dotato di maggiori capacità di accesso può essere utilizzato al posto di un iteratore dotato di minori capacità. Ad esempio, al posto di un iteratore di input si può utilizzare un iteratore in avanti.

La libreria STL supporta anche l'impiego di *iteratori inversi*. Si tratta di iteratori bidirezionali o ad accesso diretto che scorrono il contenuto di una sequenza in senso inverso. Pertanto se un iteratore inverso punta alla fine di una sequenza, incrementando tale iteratore lo si fa puntare al penultimo elemento.

Esistono anche iteratori basati su stream. Infine vengono fornite delle classi per iteratori di inserimento che semplificano le operazioni di inserimento degli elementi in un container.

Tutti gli iteratori devono supportare le operazioni per puntatori consentite dal loro tipo. Ad esempio, un iteratore di input deve supportare gli operatori ->, ++, *, == e !=. Inoltre per assegnare un valore non si può usare l'operatore *. Al contrario, un iteratore ad accesso diretto deve supportare gli operatori ->, +, ++, -, --, *, <, >, <=, >=, -=, +=, ==, != e [ ]. Inoltre l'operatore * deve consentire l'assegnamento.

## Le classi per iteratori a basso livello

L'header <iterator> definisce numerose classi che forniscono il supporto per l'implementazione degli iteratori. Come si è detto nel Capitolo 24, ognuno dei container STL definisce il proprio tipo di iteratore tramite typedef. Pertanto, quando si usano i container standard STL, in genere non si interagisce direttamente con le classi per iteratori a basso livello. Tuttavia si possono utilizzare queste classi per derivare i propri iteratori.

Molte delle classi per iteratori utilizzano il tipo ptrdiff_t. Questo tipo è in grado di rappresentare la differenza esistente fra due puntatori.

### La classe iterator

La classe iterator è la classe base per gli iteratori e ha il seguente aspetto:

```
template <class Cat, class T, class Dist  ptrdiff_t,
  class Pointer = T *, class Ref = T &>
```



```
struct iterator {
  typedef T value_type;
  typedef Dist difference_type;
  typedef Pointer pointer;
  typedef Ref reference;
  typedef Cat iterator_category;
};
```

Qui difference_type è un tipo che può contenere la differenza fra due indirizzi, value_type è il tipo del valore su cui si opera, pointer è il tipo di un puntatore a un valore, reference è il tipo di un indirizzo per il valore e iterator_category descrive il tipo dell'iteratore.

Per le categorie sono definite le seguenti classi. Il loro nome può essere memorizzato in iterator_category.

```
struct input_iterator_tag {};
struct output_iterator_tag {};
struct forward_iterator_tag: public input_iterator_tag {};
struct bidirectional_iterator_tag: public forward_iterator_tag {};
struct random_access_iterator_tag: public
bidirectional_iterator_tag {};
```

### La classe iterator_traits

La classe iterator_traits fornisce un mezzo condiviso ma molto comodo per esporre i vari tipi definiti da un iteratore. La classe è definita nel seguente modo:

```
template<class Iterator> struct iterator_traits {
  typedef Iterator::difference_type difference_type;
  typedef Iterator::value_type value_type;
  typedef Iterator::pointer pointer;
  typedef Iterator::reference reference;
  typedef Iterator::iterator_category iterator_category;
}
```

## Gli iteratori predefiniti

L'header <iterator> contiene numerosi iteratori predefiniti che possono essere utilizzati direttamente dal programma o possono costituire la base per creare altri iteratori. Questi iteratori sono elencati nella Tabella 35.1. Si noti che quattro di questi iteratori operano su stream. Lo scopo principale degli iteratori per stream è quello di consentire la manipolazione degli stream da parte degli algoritmi. Inoltre si noti la presenza degli iteratori di inserimento. Quando si usano questi iteratori

nelle istruzioni di assegnamento, essi inseriscono gli elementi in una sequenza senza cancellare gli elementi esistenti.

In questa sezione verranno esaminati tutti gli iteratori predefiniti.

### L'iteratore insert_iterator

La classe insert_iterator supporta gli iteratori di output che inseriscono oggetti in un container. Ecco la definizione template della classe:

```
template <class Cont> class insert_iterator:
  public iterator<output_iterator_tag, void, void, void, void>
```

Qui Cont è il tipo del container su cui opera l'iteratore. insert_iterator ha il seguente costruttore:

```
insert_iterator(Cont &cnt, typename Cont::iterator itr);
```

Qui *cnt* è il container su cui si opera e *itr* è un iteratore per il container che verrà utilizzato per inizializzare insert_iterator.

insert_iterator definisce i seguenti operatori: =, *, ++. In una variabile protected chiamata container è contenuto un puntatore al container. L'iteratore del container è contenuto in una variabile protected chiamata iter.

Inoltre definisce la funzione inserter() che crea un iteratore insert_iterator. Ecco l'aspetto di tale funzione:

```
template <class Cont, class Iterator> insert_iterator<Cont>
  inserter(Cont &cnt, Iterator itr);
```

Gli iteratori di inserimento eseguono un inserimento senza cancellare il contenuto di un container. Per comprendere appieno il funzionamento di un iteratore di inserimento, si consideri il seguente programma. Innanzitutto il programma crea un piccolo vettore di interi e poi utilizza un iteratore insert_iterator per inserirvi nuovi elementi senza cancellare gli elementi esistenti.

```
// Illustra l'uso di insert_iterator.
#include <iostream>
#include <iterator>
#include <vector>
using namespace std;
```



```
int main()
{
  vector<int> v;
  vector<int>::iterator itr;
  int i;

  for(i=0; i<5; i++)
    v.push_back(i);

  cout << "Array originale: ";
  itr = v.begin();
  while(itr != v.end())
    cout << *itr++ << " ";
  cout << endl;

  itr = v.begin();
  itr += 2; // punta all'elemento 2

  // crea insert_iterator all'elemento 2
  insert_iterator<vector<int> > i_itr(v, itr);

  // inserisce senza sovrascrivere
  *i_itr++ = 100;
  *i_itr++ = 200;

  cout << "L'array dopo l'inserimento: ";
  itr = v.begin();
  while(itr != v.end())
    cout << *itr++ << " ";

  return 0;
}
```

Ecco l'output prodotto dal programma.

```
Array originale: 0 1 2 3 4
L'array dopo l'inserimento: 0 1 100 200 2 3 4
```

Nel programma, se gli assegnamenti dei valori 100 e 200 fossero stati eseguiti con un iteratore standard, due degli elementi dell'array sarebbero stati cancellati. Gli stessi principi si applicano anche agli iteratori back_insert_iterator e front_insert_iterator.

**Tabella 35.1** Le classi predefinite degli iteratori predefiniti.

| CLASSE | DESCRIZIONE |
|---|---|
| insert_iterator | Un iteratore di output in qualsiasi punto del container. |
| back_insert_iterator | Un iteratore di output per l'inserimento alla fine del container. |
| front_insert_iterator | Un iteratore di output per l'inserimento all'inizio del container. |
| reverse_iterator | Un iteratore inverso, bidirezionale o ad accesso diretto. |
| istream_iterator | Un iteratore di input da stream. |
| istreambuf_iterator | Un iteratore di input da streambuf. |
| ostream_iterator | Un iteratore di output su stream. |
| ostreambuf_iterator | Un iteratore di output su iterator. |

### L'iteratore back_insert_iterator

La classe back_insert_iterator supporta gli iteratori di output che inseriscono oggetti alla fine di un container utilizzando push_back(). Ecco la definizione template della classe:

```
template <class Cont> class back_insert_iterator:
  public iterator<output_iterator_tag, void, void, void, void>
```

Qui Cont è il tipo del container su cui opera l'iteratore. back_insert_iterator ha il seguente costruttore:

```
explicit back_insert_iterator(Cont &cnt);
```

Qui *cnt* è il container su cui si opera. Gli inserimenti verranno eseguiti alla fine del container.

back_insert_iterator definisce i seguenti operatori: =, *, ++. In una variabile protected chiamata container è contenuto un puntatore al container.

Inoltre definisce la funzione back_inserter() che crea un iteratore back_insert_iterator. Ecco l'aspetto di tale funzione:

```
template <class Cont> back_insert_iterator<Cont> back_inserter(Cont
&cnt);
```

### L'iteratore front_insert_iterator

La classe front_insert_iterator supporta gli iteratori di output che inseriscono oggetti all'inizio di un container utilizzando push_front().



Ecco la definizione template della classe:

```
template <class Cont> class front_insert_iterator:
  public iterator<output_iterator_tag, void, void, void, void>
```

Qui Cont è il tipo del container su cui opera l'iteratore. front_insert_iterator ha il seguente costruttore:

```
explicit front_insert_iterator(Cont &cnt);
```

Qui *cnt* è il container su cui si opera. Gli inserimenti verranno eseguiti all'inizio del container.

front_insert_iterator definisce i seguenti operatori: =, *, ++. In una variabile protected chiamata container è contenuto un puntatore al container.

Inoltre definisce la funzione front_inserter() che crea un iteratore front_insert_iterator. Ecco l'aspetto di tale funzione:

```
template <class Cont> front_insert_iterator<Cont> inserter(Cont &cnt);
```

### L'iteratore reverse_iterator

La classe reverse_iterator supporta le operazioni di iterazione inversa. Un iteratore inverso si comporta al contrario di un normale iteratore. Ad esempio l'operatore ++ fa in modo che l'iteratore inverso proceda a ritroso. Ecco la sua definizione template:

```
template <class Iter> class reverse_iterator:
  public iterator<iterator_traits<Iter>::iterator_category,
                  iterator_traits<Iter>::value_type,
                  iterator_traits<Iter>::difference_type,
                  iterator_traits<Iter>::pointer,
                  iterator_traits<Iter>::reference>
```

Qui Iter è un iteratore ad accesso diretto o un iteratore bidirezionale. reverse_iterator ha i seguenti costruttori:

```
reverse_iterator( );
explicit reverse_iterator(Iter itr);
```

Qui *itr* è un iteratore che specifica la posizione di partenza.

Se Iter è un iteratore ad accesso diretto, allora sono disponibili i seguenti operatori: ->, +, ++, -, --, *, <, >, <=, >=, -=, +=, ==, != e [ ]. Se Iter è un iteratore bidirezionale, allora saranno disponibili solamente gli operatori ->, ++, --, *, == e !=.

La classe reverse_iterator definisce un membro protected chiamato current che è un iteratore alla posizione corrente.

reverse_iterator definisce anche la funzione base() il cui prototipo è:

```
Iter base( ) const;
```

La funzione restituisce un iteratore alla posizione corrente.

### L'iteratore istream_iterator

La classe istream_iterator supporta le operazioni di un iteratore di input su uno stream. Ecco la sua definizione template:

```
template <class T, class CharType, class Attr = char_traits<CharType>,
          class Dist = ptrdiff_t> class Distream_iterator:
  public iterator<input_iterator_tag, T, Dist, const T *, const T &>
```

Qui T è il tipo dei dati e CharType è il tipo dei caratteri (char o wchar_t) su cui opera lo stream. Dist è un tipo in grado di contenere la differenza esistente fra due indirizzi. istream_iterator ha i seguenti costruttori:

```
istream_iterator( );
istream_iterator(istream_type &stream);
istream_iterator(const istream_iterator<T, CharType, Attr, Dist> &ob);
```

Il primo costruttore crea un iteratore per uno stream vuoto. Il secondo crea un iteratore allo stream specificato. Il tipo istream_type è un typedef che specifica il tipo dello stream di input. La terza forma crea una copia di un oggetto istream_iterator.

La classe istream_iterator definisce i seguenti operatori: ->, * e ++. Per oggetti di tipo istream_iterator sono definiti anche gli operatori == e !=.

Ecco un breve programma che illustra il funzionamento di istream_iterator. Il programma legge e visualizza i caratteri da cin finché non riceve il carattere punto.

```
// Uso di istream_iterator
#include <iostream>
#include <iterator>
using namespace std;

int main()
{
  istream_iterator<char> in_it(cin);

  do {
    cout << *in_it++;
  } while (*in_it != '.');

  return 0;
}
```



### L'iteratore istreambuf_iterator

La classe istreambuf_iterator supporta le operazioni di un iteratore di input di caratteri su uno stream. Ecco la sua definizione template:

```
template <class CharType, class Attr = char_traits<CharType> >
class istreambuf_iterator:
  public iterator<input_iterator_tag, CharType, typename Attr::off_type,
                  CharType *, CharType &>
```

Qui CharType è il tipo dei caratteri (char o wchar_t) su cui opera lo stream. istreambuf_iterator ha i seguenti costruttori:

```
istreambuf_iterator( ) throw( );
istreambuf_iterator(istream_type &stream) throw( );
istreambuf_iterator(streambuf_type *streambuf) throw( );
```

Il primo costruttore crea un iteratore per uno stream vuoto. Il secondo crea un iteratore allo stream specificato. Il tipo istream_type è un typedef che specifica il tipo dello stream di input. La terza forma crea un iteratore utilizzando il buffer per stream specificato da *streambuf*.

La classe istreambuf_iterator definisce i seguenti operatori: * e ++. Per oggetti di tipo istreambuf_iterator sono definiti anche gli operatori == e !=.

istreambuf_iterator definisce la funzione membro equal() che ha il seguente aspetto:

```
bool equal(istreambuf_iterator<CharType, Attr> &ob);
```

Il suo funzionamento è un po' fuorviante. Restituisce true se l'iteratore chiamante e *ob* puntano entrambi alla fine dello stream. Inoltre restituisce true se entrambi gli iteratori non puntano alla fine dello stream. Non è richiesto che l'oggetto puntato sia lo stesso. Negli altri casi restituisce false. Gli operatori == e != funzionano allo stesso modo.

### L'iteratore ostream_iterator

La classe ostream_iterator supporta le operazioni di un iteratore di output su uno stream. Ecco la sua definizione template:

```
template <class T, class CharType, class Attr = char_traits<CharType> >
class ostream_iterator:
  public iterator<output_iterator_tag, void, void, void, void>
```

Qui T è il tipo dei dati da trasferire e CharType è il tipo dei caratteri (char o wchar_t) su cui opera lo stream. ostream_iterator ha i seguenti costruttori:

```
ostream_iterator(ostream_type &stream);
ostream_iterator(ostream_type &stream, const CharType *delim);
ostream_iterator(const ostream_iterator<T, CharType, Attr> &ob);
```

Il primo costruttore crea un iteratore per lo stream puntato da *stream*. Il tipo ostream_type è un typedef che specifica il tipo dello stream di output. La seconda forma crea un iteratore allo stream specificato da *stream* e usa i delimitatori specificati da *delim*. I delimitatori vengono scritti sullo stream dopo ogni operazione di output. La terza forma crea una copia di un oggetto *ostream_iterator*.

La classe *ostream_iterator* definisce i seguenti operatori: =, * e ++.

Ecco un breve programma che illustra il funzionamento di ostream_iterator.

```
// Uso di ostream_iterator
#include <iostream>
#include <iterator>
using namespace std;

int main()
{
  ostream_iterator<char> out_it(cout);

  *out_it = 'X';
  out_it++;
  *out_it ='Y';
  out_it++;
  *out_it = ' ';

  char str[] = "Gli iteratori C++ sono molto potenti.\n";
  char *p = str;

  while(*p) *out_it++ = *p++;

  ostream_iterator<double> out_double_it(cout);
  *out_double_it = 187.23;
  out_double_it++;
  *out_double_it = -102.7;
```



```
  return 0;
}
```

Ecco l'output del programma

```
XY Gli iteratori C++ sono molto potenti.
187.23-102.7
```

### L'iteratore ostreambuf_iterator

La classe ostreambuf_iterator supporta le operazioni di un iteratore di output di caratteri su uno stream. Ecco la sua definizione template:

```
template <class CharType, class Attr = char_traits<CharType> >
class ostreambuf_iterator:
  public iterator<output_iterator_tag, void, void, void, void>
```

Qui CharType è il tipo dei caratteri (char o wchar_t) su cui opera lo stream. ostreambuf_iterator ha i seguenti costruttori:

```
ostreambuf_iterator(ostream_type &stream) throw( );
ostreambuf_iterator(streambuf_type *streambuf) throw( );
```

Il primo costruttore crea un iteratore per lo stream puntato da *stream*. Il tipo ostream_type è un typedef che specifica il tipo dello stream di output. La seconda forma crea un iteratore allo stream specificato. Il tipo streambuf_type è un typedef che specifica il tipo del buffer dello stream.

La classe ostreambuf_iterator definisce i seguenti operatori: =, * e ++. La funzione membro failed() ha il seguente aspetto:

```
bool failed( ) const throw( );
```

Se non si è verificato alcun problema restituisce false, altrimenti restituisce true.

## Due funzioni per iteratori

Per gli iteratori sono definite due funzioni particolari: advance() e distance(). Ecco il loro aspetto:

```
template <class InIter, class Dist> void advance(InIter &itr, Dist d);
template <class InIter> distance(InIter start, InIter end);
```

La funzione advance() incrementa *itr* della quantità specificata da *d*. La funzione distance() restituisce il numero di elementi compresi fra *start* ed *end*.

Queste due funzioni esistono perché solo gli iteratori ad accesso diretto consentono di sommare o sottrarre un valore da un iteratore. Le funzioni advance() e distance() consentono di superare questa restrizione. Occorre però sapere che alcuni iteratori non sono in grado di implementare queste funzioni in modo efficiente.

## 35.2 Gli oggetti funzione

Gli oggetti funzione sono classi che definiscono operator(). La libreria STL definisce vari oggetti funzione liberamente utilizzabili dai programmi. È anche possibile definire oggetti funzione specifici. Il supporto degli oggetti funzione si trova nell'header <functional>. Inoltre <functional> definisce varie entità di supporto per gli oggetti funzione, ovvero i binder, i negatori e gli adattatori.

NOTA *Per una panoramica sugli oggetti funzione, consultare il Capitolo 24.*

### Cosa sono gli oggetti funzione

Vi sono due tipi di oggetti funzione: oggetti binari e unari. Di seguito sono elencati gli oggetti funzione binari:

| | | | | |
|---|---|---|---|---|
| plus | minus | multiplies | divides | modulus |
| equal_to | not_equal_to | greater | greater_equal | less |
| less_equal | logical_and | logical_or | | |

Ed ecco gli oggetti funzione unari.

logical_not negate

La chiamata a un oggetto funzione ha il seguente aspetto:

*func_ob<type>*( )

Ad esempio:

```
less<int>()
```

richiama less() su operandi di tipo int.



Ecco la classe base di tutti gli oggetti funzione binari, ovvero binary_function:

```
template <class Agument1, class Agument2, class Result>
struct binary_function {
  typedef Argument1 first_argument_type;
  typedef Argument2 second_argument_type;
  typedef Result result_type;
};
```

La classe base per tutte le funzioni unarie è unary_function:

```
template <class Argument, class Result> struct unary_function
  typedef Argument argument_type;
  typedef Result result_type;
};
```

Queste classi template forniscono nomi concreti per i tipi generici utilizzati dagli oggetti funzione. Anche se tecnicamente sono solo una comodità, in realtà vengono utilizzati ogni volta che si creano oggetti funzione.

Le specifiche template di tutti gli oggetti funzione binari sono simili, così come le specifiche template di tutti gli oggetti funzione unari:

```
template <class T> struct plus : binary_function<T, T, T>
{
  T operator() (const T &arg1, const T&arg2) const;
};

template <class T> struct negate : unary_function<T, T>
{
  T operator() (const T &arg) const;
};
```

Ogni funzione operator() restituisce il risultato specificato.

### I binder

I binder collegano un valore a un argomento di un oggetto funzione binario, producendo un oggetto funzione unario. Esistono due binder: bind2nd() e bind1st(). Ecco come sono definiti:

```
template <class BinFunc, class T>
        binder1st<BinFunc> bind1st(const BinFunc &op, const T
```

*&value*);
template <class BinFunc, class T>
binder2nd<BinFunc> bind2nd(const BinFunc *&op*, const T *&value*);

Qui *op* è un oggetto funzione binario, come ad esempio less() o greater(), che fornisce l'operazione desiderata e *value* è il valore da collegare. bind1st() restituisce un oggetto funzione unario in cui l'operando sinistro è collegato a *value*. bind2nd() restituisce un oggetto funzione unario in cui l'operando di destra è collegato a *value*. Il più utilizzato è decisamente bind2nd(). In entrambi i casi, il risultato di un binder è un oggetto funzione unario a cui viene collegato il valore specificato.

Ecco l'aspetto delle classi binder2nd e binder1st.

```
template <class BinFunc> class binder1st:
  public unary_function(typename BinFunc::second_argument_type,
                        typename BinFunc::result_type>
{
protected:
  BinFunc op;
  typename BinFunc::first_argument_type value;
public:
  binder1st(const BinFunc &op,
            const typename BinFunc::first_argument_type &v);
  result_type operator()(const argument_type &v) const;
 };

template <class BinFunc> class binder2nd:
  public unary_function(typename BinFunc::first_argument_type,
                        typename BinFunc::result_type>
{
protected:
  BinFunc op;
  typename BinFunc::second_argument_type value;
public:
  binder2nd(const BinFunc &op,
            const typename BinFunc::second_argument_type &v);
  result_type operator()(const argument_type &v) const;
 };
```

Qui ***BinFunc*** è il tipo di un oggetto funzione binario. Si noti che entrambe le classi ereditano unary_function. Questo è il motivo per cui l'oggetto prodotto da bind2nd() e bind1st() può essere utilizzato ogni volta che è richiesto l'impiego una funzione unaria.



## I negatori

I negatori restituiscono dei predicati che forniscono l'opposto del predicato che modificano. I negatori sono not1() e not2() e sono definiti nel seguente modo:

template <class UnPred> unary_negate<UnPred> not1(const UnPred &pred);
template <class BinPred> binary_negate<BinPred> not2(const BinPred &pred);

Ecco le loro classi:

```
template <class UnPred> class unary_negate:
  public unary_function<typename UnPred::argument_type, bool>
{
public:
  explicit unary_negate(const UnPred &pred);
  bool operator()(const argument_type &v) const;
};

template <class BinPred> class binary_negate:
  public binary_function<typename BinPred::first_argument_type,
                         typename BinPred::second_argument_type,
                         bool>
{
public:
  explicit binary_negate(const BinPred &pred);
  bool operator()(const first_argument_type &v1,
                  const second_argument_type &v2) const;
};
```

In entrambi i casi, operator() restituisce la negazione del predicato specificato da *pred*.

## Gli adattatori

L'header <functional> definisce varie classi chiamate adattatori che consentono di adattare un puntatore a funzione alla forma utilizzabile dalla libreria STL. Ad esempio si può usare un adattatore per utilizzare una funzione come strcmp() come un predicato. Esistono adattatori anche per i puntatori membri.

### Gli adattatori da puntatore a funzione

Ecco gli adattatori da puntatore a funzione.

```
template <class Argument, class Result>
  pointer_to_unary_function<Argument, Result>
    ptr_fun(Result (*func)(Argument));
template <class Argument1, class Argument2, class Result>
  pointer_to_binary_function<Agument1, Argument2, Result>
    ptr_fun(Result (*func)(Argument1, Argument2));
```

Qui ptr_fun() restituisce un oggetto di tipo pointer_to_unary_function oppure pointer_to_binary_function. Ecco le loro classi:

```
template <class Argument, class Result>
class pointer_to_unary_function:
  public unary_function<Argument, Result>
{
public:
  explicit pointer_to_unary_function(Result (*func)(Argument));
  Result operator()(Argument arg) const;
};

template <class Argument1, class Argument2, class Result>
class pointer_to_binary_function:
  public binary_function<Argument1, Argument2, Result>
{
public:
  explicit pointer_to_binary_function(
            Result (*func)(Argument1, Argument2));
  Result operator()(Argument1 arg1, Argument2 arg2) const;
};
```

Per le funzioni unarie, operator() restituisce:

*func*(*arg*).

Per le funzioni binarie, operator() restituisce:

*func*(*arg1*, *arg2*);

Il tipo del risultato dell'operazione è specificata dal tipo generico Result.

### Adattatore da puntatore a funzione membro

Gli adattatori da puntatore a funzione membro sono:

```
template<class Result, class T>
    mem_fun_t<Result, T> mem_fun(Result (T::*func)());
```



```
template<class Result, class T, class Argument>
    mem_fun1_t<Result, T, Argument>
      mem_fun1(Result (T::*func)(Argument));
```

Qui mem_fun() restituisce un oggetto di tipo mem_fun_t e mem_fun1 restituisce un oggetto di tipo mem_fun1_t. Ecco le loro classi:

```
template <class Result, class T> class mem_fun_t:
  public unary_function<T *, Result> {
public:
  explicit mem_fun_t(Result (T::*func)());
  Result operator() (T *func) const;
};

template <class Result, class T,
          class Argument> class mem_fun1_t:
  public binary_function<T *, Argument, Result> {
public:
  explicit mem_fun1_t(Result (T::*func)(Argument));
  Result operator() (T *func, Argument arg) const;
};
```

Qui il costruttore mem_fun_t richiama la funzione membro specificata dal suo parametro. Il costruttore mem_fun1_t richiama la funzione membro specificata nel suo primo parametro passandole come secondo parametro un valore di tipo Argument.

Esistono classi e funzioni parallele per l'utilizzo di indirizzi e membri. Ecco la forma generale delle funzioni:

```
template<class Result, class T>
    mem_fun_t<Result, T> mem_fun_ref(Result (T::*func)());

template<class Result, class T, class Argument>
    mem_fun1_t<Result, T, Argument>
      mem_fun1_ref(Result (T::*func)(Argument));
```

Ecco l'aspetto delle classi mem_fun_ref e mem_fun1_ref.

```
template <class Result, class T> class mem_fun_ref_t:
  public unary_function<T, Result>
{
public:
  explicit mem_fun_ref_t(Result (T::*func)());
```

```
  Result operator()(T &func) const;
};

template <class Result, class T, class Argument>
  class mem_fun1_ref_t:
    public binary_function<T, Result, Argument>
{
public:
  explicit mem_fun1_ref_t(Result (T::*func)(Argument));
  Result operator()(T &func, Argument arg) const;
};
```

## 35.3 Gli allocatori

Un allocatore gestisce le operazioni di allocazione della memoria per un container. Poiché la libreria STL definisce un allocatore standard che viene utilizzato automaticamente dai container, la maggior parte dei programmatori non ha bisogno di conoscere i dettagli del funzionamento degli allocatori o né crearne di nuovi. Tuttavia questi dettagli possono essere utili per creare nuove classi di libreria.

Tutti gli allocatori devono soddisfare vari requisiti. Innanzitutto devono definire i seguenti tipi:

| | |
|---|---|
| const_pointer | Un puntatore const pointer a un oggetto di tipo value_type. |
| const_reference | L'indirizzo const di un oggetto di tipo value_type. |
| difference_type | Può rappresentare la differenza fra due indirizzi. |
| pointer | Un puntatore a un oggetto di tipo value_type. |
| reference | L'indirizzo di un oggetto di tipo value_type. |
| size_type | Può contenere le dimensioni dell'oggetto più esteso che può essere allocato. |
| value_type | Il tipo dell'oggetto da allocare. |

In secondo luogo devono fornire le seguenti funzioni.

| | |
|---|---|
| address | Restituisce un puntatore, dato un indirizzo. |
| allocate | Alloca la memoria. |
| deallocate | Libera la memoria. |
| max_size | Restituisce il numero massimo di oggetti che possono essere allocati. |
| construct | Costruisce un oggetto. |
| destroy | Distrugge un oggetto. |



Devono essere definite anche le operazioni == e !=.

L'allocatore standard è allocator, definito nell'header <memory>. Ecco la sua specifica template:

```
template <class T> class allocator
```

Qui T è il tipo degli oggetti che saranno allocati da allocator. allocator definisce i seguenti costruttori:

```
allocator( ) throw( );
allocator(const allocator<T> &ob) throw( );
```

Il primo crea un nuovo allocatore e il secondo crea una copia di *ob*.

Per allocator sono definiti gli operatori == e !=. La Tabella 35.2 mostra le funzioni membro definite da allocator.

È definita anche una specializzazione di allocator per puntatori void *.

**Tabella 35.2** Le funzioni membro di allocator.

| FUNZIONE | DESCRIZIONE |
|---|---|
| `pointer address(reference ob) const;`<br>`const_pointer address(const_reference ob) const;` | Restituisce l'indirizzo di *ob*. |
| `pointer allocate(size_type num, typename allocator<void>::const_pointer h = 0);` | Restituisce un puntatore alla memoria allocata le cui dimensioni sono sufficienti per contenere *num* oggetti di tipo T. Il valore di *h* (che può essere ignorato) aiuta a soddisfare la richiesta. |
| `void construct(pointer ptr, const_reference val);` | Costruisce in *ptr* un oggetto di tipo T. |
| `void deallocate(pointer ptr, size_type num);` | Dealloca *num* oggetti di tipo T a partire da *ptr*. Il valore di *ptr* deve essere stato ottenuto tramite `allocate()`. |
| `void destroy(pointer ptr);` | Distrugge l'oggetto in *ptr*. Viene automaticamente richiamato il suo distruttore. |
| `size_type max_size( ) const throw( );` | Restituisce il numero massimo di oggetti di tipo T che possono essere allocati. |

Capitolo 36

# La classe string

36.1 **La classe basic_string**
36.2 **La classe char_traits**

Questo capitolo descrive la classe per stringhe del C++ standard. Il linguaggio C++ supporta due tipi di stringhe di caratteri. Il primo è costituito dagli array di caratteri chiusi da un carattere nullo. Questo viene anche chiamato con il nome di "stringa C". Il secondo modo consiste nell'impiegare una classe di tipo basic_string. Esistono due specializzazioni di basic_string: string che supporta stringhe di char e wstring che supporta stringhe di wchar_t (caratteri estesi). In genere si utilizzano oggetti di tipo string.

La classe basic_string è in pratica un container. Questo significa che gli iteratori e gli algoritmi STL possono operare anche sulle stringhe. Tuttavia le stringhe hanno anche alcune funzionalità aggiuntive.

char_traits è una classe utilizzata da basic_string per definire vari attributi dei caratteri che costituiscono la stringa. È importante comprendere che anche se la maggior parte delle stringhe è composta da caratteri char o wchar_t, le stringhe basic_string possono operare su qualsiasi oggetto che possa essere utilizzato per rappresentare un carattere. In questa sezione verranno descritte sia la classe basic_string che la classe char_traits.

NOTA *Per una panoramica sull'uso della classe string, consultare il Capitolo 24.*

## 36.1 La classe basic_string

La specifica template della classe basic_string è la seguente:

```
template <class CharType, class Attr = char_traits<CharType>,
          class Allocator = allocator<T> > class basic_string
```

Qui CharType è il tipo di carattere utilizzato, Attr è la classe che descrive le caratteristiche dei caratteri e Allocator specifica l'allocatore. La classe basic_string ha i seguenti costruttori:

```
explicit basic_string(const Allocator &a = Allocator( ));
basic_string(size_type len, CharType ch ,
        const Allocator &a = Allocator( ));
basic_string(const CharType *str,  const Allocator &a = Allocator( ));
basic_string(const CharType *str, size_type len,
        const Allocator &a = Allocator( ));
basic_string(const basic_string &str, size_type indx = 0,
        size_type len=npos, const Allocator &a = Allocator( ));
template <class nter> basic_string(InIter start, InIter end,
        const Allocator &a = Allocator( ));
```

La prima forma costruisce una stringa vuota. La seconda forma costruisce una stringa contenente *len* caratteri di valore *ch*. La terza forma costruisce una stringa che contiene gli stessi elementi di *str*. La quarta forma costruisce una stringa che contiene una sottostringa di *str* che inizia da 0 e si estende per *len* caratteri. La quinta forma costruisce una stringa a partire da un'altra basic_string utilizzando la sottostringa che inizia da *indx* e si estende per *len* caratteri. La sesta forma costruisce una stringa che contiene gli elementi dell'intervallo specificato da *start* ed *end*.

Per basic_string sono definiti i seguenti operatori di confronto:

```
==, <, <=, ! =, >, >=
```

Inoltre viene definito l'operatore + che fornisce il risultato del concatenamento di una stringa con un'altra e gli operatori di I/O << e >> utilizzabili per eseguire operazioni di input e output di stringhe.

L'operatore + può essere utilizzato per concatenare un oggetto stringa con un altro oggetto stringa oppure un oggetto stringa con una stringa C. Pertanto possono essere utilizzate le seguenti varianti:

```
stringa + stringa
stringa + stringa C
stringa C + stringa
```

L'operatore + può essere utilizzato anche per concatenare un carattere alla fine della stringa. La classe basic_string definisce la costante npos e normalmente



vale -1. Questa costante rappresenta la lunghezza massima che può essere raggiunta da una stringa.

Nelle descrizioni, il tipo generico CharType rappresenta il tipo di carattere contenuto in una stringa. Poiché i nomi dei tipi fittizi di una classe template sono arbitrari, basic_string dichiara versioni typedef di questi tipi. Questo rende più concreti i nomi impiegati per i tipi. Ecco di seguito i tipi definiti da basic_string:

| | |
|---|---|
| size_type | Un tipo intero che in genere equivale a size_t. |
| reference | Un indirizzo per un carattere all'interno di una stringa. |
| const_reference | Un indirizzo const a un carattere all'interno di una stringa. |
| iterator | Un iteratore |
| const_iterator | Un iteratore const |
| reverse_iterator | Un iteratore inverso |
| const_reverse_iterator | Un iteratore inverso const |
| value_type | Il tipo del carattere contenuto nella stringa. |
| allocator_type | Il tipo di allocatore. |
| pointer | Un puntatore a un carattere all'interno di una stringa. |
| const_pointer | Un puntatore const a un carattere all'interno di una stringa. |
| traits_type | Un typedef per char_traits<CharType>. |
| difference_type | Un tipo che può contenere la differenza esistente fra due indirizzi. |

La Tabella 36.1 elenca le funzioni membro definite da basic_string. Poiché la maggior parte dei programmatori impiega stringhe di char e per semplificare la discussione, le tabelle usano il tipo string ma le funzioni possono essere applicate anche a oggetti di tipo wstring (o a qualsiasi altro tipo di basic_string).

**Tabella 36.1** Le funzioni membro di string.

| MEMBRO | DESCRIZIONE |
|---|---|
| `string &append(const string &str);` | Aggiunge *str* alla fine della stringa chiamante. Restituisce `*this`. |
| `string &append(const string &str, size_type indx, size_type len);` | Aggiunge una sottostringa di *str* alla fine della stringa chiamante. La sottostringa da aggiungere inizia da *indx* e si estende per *len* caratteri. Restituisce `*this`. |
| `string &append(const CharType *str);` | Aggiunge *str* alla fine della stringa chiamante. Restituisce `*this`. |

(segue)

**Tabella 36.1** Le funzioni membro di string. *(continua)*

| MEMBRO | DESCRIZIONE |
|---|---|
| `string &append(const CharType *str, size_type num);` | Aggiunge i primi *num* caratteri di *str* alla fine della stringa chiamante. Restituisce `*this`. |
| `string &append(size_type len, CharType ch);` | Aggiunge *len* caratteri specificati da *ch* alla fine della stringa chiamante. Restituisce `*this`. |
| `template<class InIter> string &append(InIter start, InIter end);` | Aggiunge la sequenza specificata da *start* ed *end* alla fine della stringa chiamante. Restituisce `*this`. |
| `string &assign(const string &str);` | Assegna *str* alla stringa chiamante. Restituisce `*this`. |
| `string &assign(const string &str, size_type indx, size_type len);` | Assegna una sottostringa di *str* alla stringa chiamante. La sottostringa da assegnare inizia da *indx* e si estende per *len* caratteri. Restituisce `*this`. |
| `string &assign(const CharType *str);` | Assegna *str* alla stringa chiamante. Restituisce `*this`. |
| `string &assign(const CharType *str, size_type len);` | Assegna i primi *len* caratteri di *str* alla stringa chiamante. Restituisce `*this`. |
| `string &assign(size_type len, CharType ch);` | Assegna *len* caratteri specificati da *ch* alla fine della stringa chiamante. Restituisce `*this`. |
| `template<class InIter> string &assign(InIter start, InIter end);` | Assegna la sequenza specificata da *start* ed *end* alla stringa chiamante. Restituisce `*this`. |
| `reference at(size_type indx); const_reference at(size_type indx) const;` | Restituisce l'indirizzo del carattere specificato da *indx*. |
| `iterator begin( ); const_iterator begin( ) const;` | Restituisce un iteratore al primo elemento della stringa. |
| `const CharType *c_str( ) const;` | Restituisce un puntatore alla versione a stringa C (chiusa dal carattere nullo) della stringa chiamante. |
| `size_type capacity( ) const;` | Restituisce la capacità corrente della stringa. Questo è il numero di caratteri che può contenere prima di dover richiedere ulteriore memoria. |
| `int compare(const string &str) const;` | Confronta *str* con la stringa chiamante. Restituisce uno dei seguenti valori:<br>Minore di zero se `*this` < *str*<br>Zero se `*this` == *str*<br>Maggiore di zero se `*this` > *str* |
| `int compare(size_type indx, size_type len, const string &str) const;` | Confronta *str* con una sottostringa della stringa chiamante. La sottostringa inizia da *indx* e si estende per *len* caratteri. Restituisce uno dei seguenti valori:<br>Minore di zero se `*this` < *str*<br>Zero se `*this` == *str*<br>Maggiore di zero se `*this` > *str* |

*(segue)*



**Tabella 36.1** Le funzioni membro di string. *(continua)*

| MEMBRO | DESCRIZIONE |
|---|---|
| `int compare(size_type indx, size_type len, const string &str, size_type indx2, size_type len2) const;` | Confronta una sottostringa di *str* con una sottostringa della stringa chiamante. La sottostringa in the invoking string inizia da *indx* e si estende per *len* caratteri. La sottostringa in *str* inizia da *indx2* e si estende per *len2* caratteri. Restituisce uno dei seguenti valori:<br>Minore di zero se `*this` < *str*<br>Zero se `*this` == *str*<br>Maggiore di zero se `*this` > *str* |
| `int compare(const CharType *str) const;` | Confronta *str* con la stringa chiamante. Restituisce uno dei seguenti valori:<br>Minore di zero se `*this` < *str*<br>Zero se `*this` == *str*<br>Maggiore di zero se `*this` > *str* |
| `int compare(size_type indx, size_type len, const CharType *str, size_type len2 = npos) const;` | Confronta una sottostringa di *str* con una sottostringa della stringa chiamante. La sottostringa della stringa chiamante inizia da *indx* e si estende per *len* caratteri. La sottostringa di *str* inizia da zero e si estende per *len2* caratteri. Restituisce uno dei seguenti valori:<br>Minore di zero se `*this` < *str*<br>Zero se `*this` == *str*<br>Maggiore di zero se `*this` > *str* |
| `size_type copy(CharType *str, size_type len, size_type indx = 0) const;` | A partire da *indx*, copia *len* caratteri della stringa chiamante nell'array di caratteri puntato da *str*. Restituisce il numero di caratteri copiati. |
| `const CharType *data( ) const;` | Restituisce un puntatore al primo carattere della stringa chiamante. |
| `bool empty( ) const;` | Restituisce `true` se la stringa chiamante è vuota e `false` altrimenti. |
| `iterator end( ); const_iterator end( ) const;` | Restituisce un iteratore che punta alla fine un iteratore. |
| `iterator erase(iterator i);` | Rimuove il carattere puntato da *i*. Restituisce un iteratore al carattere che segue quello rimosso. |
| `iterator erase(iterator start, iterator end);` | Rimuove i caratteri nell'intervallo compreso fra *start* ed *end*. Restituisce un iteratore al carattere che segue l'ultimo carattere rimosso. |
| `string &erase(size_type indx = 0, size_type len = npos);` | A partire da *indx*, rimuove *len* caratteri della stringa chiamante. Restituisce `*this`. |
| `size_type find(const string &str, size_type indx = 0) const;` | Restituisce l'indice della prima occorrenza di *str* nella stringa chiamante. La ricerca inizia dall'indice *indx*. Se non viene trovata alcuna corrispondenza viene restituito npos. |
| `size_type find(const CharType *str, size_type indx = 0) const;` | Restituisce l'indice della prima occorrenza di *str* nella stringa chiamante. La ricerca inizia dall'indice *indx*. Se non viene trovata alcuna corrispondenza viene restituito npos. |

*(segue)*

**Tabella 36.1** Le funzioni membro di string. *(continua)*

| MEMBRO | DESCRIZIONE |
|---|---|
| `size_type find(const CharType *str, size_type indx, size_type len) const;` | Restituisce l'indice della prima occorrenza dei primi *len* caratteri di *str* nella stringa chiamante. La ricerca inizia dall'indice *indx*. Se non viene trovata alcuna corrispondenza viene restituito npos. |
| `size_type find(CharType ch, size_type indx = 0) const;` | Restituisce l'indice della prima occorrenza di *ch* nella stringa chiamante. Se non viene trovata alcuna corrispondenza viene restituito npos. |
| `size_type find_first_of(const string &str, size_type indx = 0) const;` | Restituisce l'indice del primo carattere della stringa chiamante che corrisponde con un carattere di *str*. La ricerca inizia dall'indice *indx*. Se non viene trovata alcuna corrispondenza viene restituito npos. |
| `size_type find_first_of(const CharType *str, size_type indx = 0) const;` | Restituisce l'indice del primo carattere della stringa chiamante che corrisponde con un carattere di *str*. La ricerca inizia dall'indice *indx*. Se non viene trovata alcuna corrispondenza viene restituito npos. |
| `size_type find_first_of(const CharType *str, size_type indx, size_type len) const;` | Restituisce l'indice del primo carattere della stringa chiamante che corrisponde a un carattere dei primi *len* caratteri of *str*. La ricerca inizia dall'indice *indx*. Se non viene trovata alcuna corrispondenza viene restituito npos. |
| `size_type find_first_of(CharType ch, size_type indx = 0) const;` | Restituisce l'indice della prima occorrenza di *ch* nella stringa chiamante. La ricerca inizia dall'indice *indx*. Se non viene trovata alcuna corrispondenza viene restituito npos. |
| `size_type find_first_not_of( const string &str, size_type indx = 0) const;` | Restituisce l'indice del primo carattere della stringa chiamante che non corrisponde ad alcun carattere di *str*. La ricerca inizia dall'indice *indx*. Se non viene trovata alcuna corrispondenza viene restituito npos. |
| `size_type find_first_not_of( const CharType *str, size_type indx = 0) const;` | Restituisce l'indice del primo carattere della stringa chiamante che non corrisponde ad alcun carattere di *str*. La ricerca inizia dall'indice *indx*. Se non viene trovata alcuna corrispondenza viene restituito npos. |
| `size_type find_first_not_of( const CharType *str, size_type indx, size_type len) const;` | Restituisce l'indice del primo carattere della stringa chiamante che non corrisponde ad alcun carattere dei primi *len* caratteri of *str*. La ricerca inizia dall'indice *indx*. Se non viene trovata alcuna corrispondenza viene restituito npos. |
| `size_type find_first_not_of( CharType ch, size_type indx = 0) const;` | Restituisce l'indice del primo carattere della stringa chiamante che non corrisponde a *ch*. La ricerca inizia dall'indice *indx*. Se non viene trovata alcuna corrispondenza viene restituito npos. |
| `size_type find_last_of(const string &str, size_type indx = npos) const;` | Restituisce l'indice dell'ultimo carattere della stringa chiamante che corrisponde con un carattere di *str*. La ricerca inizia dall'indice *indx*. Se non viene trovata alcuna corrispondenza viene restituito npos. |

*(segue)*



**Tabella 36.1** Le funzioni membro di string. *(continua)*

| MEMBRO | DESCRIZIONE |
|---|---|
| `size_type find_last_of(const CharType *str, size_type indx = npos) const;` | Restituisce l'indice dell'ultimo carattere della stringa chiamante che corrisponde con un carattere di *str*. La ricerca inizia dall'indice *indx*. Se non viene trovata alcuna corrispondenza viene restituito npos. |
| `size_type find_last_of(const CharType *str, size_type indx, size_type len) const;` | Restituisce l'indice dell'ultimo carattere della stringa chiamante che corrisponde con un carattere dei primi *len* caratteri di *str*. La ricerca inizia dall'indice *indx*. Se non viene trovata alcuna corrispondenza viene restituito npos. |
| `size_type find_last_of(CharType ch, size_type indx = npos) const;` | Restituisce l'indice dell'ultima occorrenza di *ch* della stringa chiamante. La ricerca inizia dall'indice *indx*. Se non viene trovata alcuna corrispondenza viene restituito npos. |
| `size_type find_last_not_of( const string &str, size_type indx = npos) const;` | Restituisce l'indice dell'ultimo carattere della stringa chiamante che non corrisponde ad alcun carattere di *str*. La ricerca inizia dall'indice *indx*. Se non viene trovata alcuna corrispondenza viene restituito npos. |
| `size_type find_last_not_of( const CharType *str, size_type indx = npos) const;` | Restituisce l'indice dell'ultimo carattere della stringa chiamante che non corrisponde ad alcun carattere di *str*. La ricerca inizia dall'indice *indx*. Se non viene trovata alcuna corrispondenza viene restituito npos. |
| `size_type find_last_not_of( const CharType *str, size_type indx, size_type len) const;` | Restituisce l'indice dell'ultimo carattere della stringa chiamante che non corrisponde ad alcun carattere dei primi *len* caratteri di *str*. La ricerca inizia dall'indice *indx*. Se non viene trovata alcuna corrispondenza viene restituito npos. |
| `size_type find_last_not_of(CharType ch, size_type indx = npos) const;` | Restituisce l'indice dell'ultimo carattere della stringa chiamante che non corrisponde a *ch*. La ricerca inizia dall'indice *indx*. Se non viene trovata alcuna corrispondenza viene restituito npos. |
| `allocator_type get_allocator( ) const;` | Restituisce l'allocatore della stringa. |
| `iterator insert(iterator i, const CharType &ch );` | Inserisce *ch* appena prima del carattere specificato da *indx*. Viene restituito un iteratore al carattere |
| `string &insert(size_type indx, const string &str);` | Inserisce *str* nella stringa chiamante all'indice specificato da *indx*. Restituisce `*this`. |
| `string &insert(size_type indx1, const string &str, size_type indx2, size_type len);` | Inserisce una sottostringa di *str* nella stringa chiamante all'indice specificato da *indx1*. La sottostringa inizia da *indx2* e si estende per *len* caratteri. Restituisce `*this`. |
| `string &insert(size_type indx, const CharType *str);` | Inserisce *str* nella stringa chiamante all'indice specificato da *indx*. Restituisce `*this`. |
| `string &insert(size_type indx, const CharType *str, size_type len);` | Inserisce i primi *len* caratteri di *str* nella stringa chiamante all'indice specificato da *indx*. Restituisce `*this`. |

*(segue)*

**Tabella 36.1** Le funzioni membro di string. *(continua)*

| MEMBRO | DESCRIZIONE |
|---|---|
| `string &insert(size_type indx, size_type len, CharType ch);` | Inserisce *len* caratteri di valore *ch* nella stringa chiamante all'indice specificato da *indx*. Restituisce `*this`. |
| `void insert(iterator i, size_type len, const CharType &ch)` | Inserisce *len* copie di *ch* appena prima dell'elemento specificato da *i*. |
| `template <class InIter> void insert(iterator i, InIter start, InIter end);` | Inserisce la sequenza definita da *start* ed *end* appena prima dell'elemento specificato da *i*. |
| `size_type length( ) const;` | Restituisce il numero di caratteri contenuti nella stringa. |
| `size_type max_size( ) const;` | Restituisce il numero massimo di caratteri che la stringa può contenere. |
| `reference operator[ ](size_type indx) const;` `const_reference operator[ ](size_type indx) const;` | Restituisce l'indirizzo del carattere specificato da *indx*. |
| `string &operator=(const string &str);` `string &operator=(const CharType *str);` `string &operator=(CharType ch);` | Assegna alla stringa chiamante la stringa o il carattere specificato. Restituisce `*this`. |
| `string &operator+=(const string &str);` `string &operator+=(const CharType *str);` `string &operator+=(CharType ch);` | Aggiunge la stringa o il carattere specificato alla fine della stringa chiamante. Restituisce `*this`. |
| `reverse_iterator rbegin( );` `const_reverse_iterator rbegin( ) const;` | Restituisce un iteratore inverso alla fine della stringa. |
| `reverse_iterator rend( );` `const_reverse_iterator rend( ) const;` | Restituisce un iteratore inverso all'inizio della stringa. |
| `string &replace(size_type indx, size_type len, const string &str);` | Sostituisce fino a *len* caratteri della stringa chiamante, a partire da *indx* con la stringa contenuta in *str*. Restituisce `*this`. |
| `string &replace(size_type indx1, size_type len1, const string &str, size_type indx2, size_type len2);` | Sostituisce fino a *len1* caratteri della stringa chiamante a partire da *indx1* con i *len2* caratteri della stringa contenuta in *str* che inizia in *indx2*. Restituisce `*this`. |
| `string &replace(size_type indx, size_type len, const CharType *str);` | Sostituisce fino a *len* caratteri della stringa chiamante, a partire da *indx* con la stringa contenuta in *str*. Restituisce `*this`. |
| `string &replace(size_type indx1, size_type len1, const CharType *str, size_type len2);` | Sostituisce fino a *len1* caratteri della stringa chiamante a partire da *indx1* con i *len2* caratteri della stringa contenuta in *str* che inizia da *indx2*. Restituisce `*this`. |

*(segue)*



**Tabella 36.1** Le funzioni membro di string. *(continua)*

| MEMBRO | DESCRIZIONE |
|---|---|
| `string &replace(size_type indx, size_type len1, size_type len2, CharType ch);` | Sostituisce fino a *len1* caratteri della stringa chiamante a partire da *indx* con i *len2* caratteri specificati da *ch*. Restituisce `*this`. |
| `string &replace(iterator start, iterator start, const string &str);` | Sostituisce l'intervallo specificato da *start* ed *end* con *str*. Restituisce `*this`. |
| `string &replace(iterator start, iterator start, const CharType *str);` | Sostituisce l'intervallo specificato da *start* ed *end* con *str*. Restituisce `*this`. |
| `string &replace(iterator start, iterator end, const CharType *str, size_type len);` | Sostituisce l'intervallo specificato da *start* ed *end* con i primi *len* caratteri di *str*. Restituisce `*this`. |
| `string &replace(iterator start, iterator end, size_type len, CharType ch);` | Sostituisce l'intervallo specificato da *start* ed *end* con i *len* caratteri specificati da *ch*. Restituisce `*this`. |
| `template <class InIter> string &replace(iterator start1, iterator end1, InIter start2, InIter end2);` | Sostituisce l'intervallo specificato da *start1* ed *end1* con i caratteri specificati da *start2* ed *end2*. Restituisce `*this`. |
| `void reserve(size_type num = 0);` | Imposta la capacità della stringa a un valore uguale ad almeno *num*. |
| `void resize(size_type num)` `void resize(size_type num, CharType ch);` | Cambia le dimensioni della stringa a quelle specificate da *num*. Se la stringa deve essere allungata, alla fine vengono aggiunti elementi con il valore specificato da *ch*. |
| `size_type rfind(const string &str, size_type indx = npos) const;` | Restituisce l'indice dell'ultima occorrenza di *str* nella stringa chiamante. La ricerca inizia dall'indice *indx*. Se non viene trovata alcuna corrispondenza restituisce npos. |
| `size_type rfind(const CharType *str, size_type indx = npos) const;` | Restituisce l'indice dell'ultima occorrenza di *str* nella stringa chiamante. La ricerca inizia dall'indice *indx*. Se non viene trovata alcuna corrispondenza restituisce npos. |
| `size_type rfind(const CharType *str, size_type indx, size_type len) const;` | Restituisce l'indice dell'ultima occorrenza dei primi *len* caratteri di *str* nella stringa chiamante. La ricerca inizia dall'indice *indx*. Se non viene trovata alcuna corrispondenza restituisce npos. |
| `size_type rfind(CharType ch, size_type indx = npos) const;` | Restituisce l'indice dell'ultima occorrenza di *ch* nella stringa chiamante. La ricerca inizia dall'indice *indx*. Se non viene trovata alcuna corrispondenza restituisce npos. |
| `size_type size( ) const;` | Restituisce il numero di caratteri attualmente contenuti nella stringa. |

*(segue)*

**Tabella 36.1** Le funzioni membro di string. *(continua)*

| MEMBRO | DESCRIZIONE |
|---|---|
| `string substr(size_type indx = 0, size_type len = npos) const;` | Restituisce una sottostringa di *len* caratteri a partire da *indx* nella stringa chiamante. |
| `void swap(string &str)` | Scambia i caratteri contenuti nella stringa chiamante con quelli di *ob*. |

## 36.2 La classe char_traits

La classe char_traits descrive vari attributi associati a un carattere. Ecco la sua specifica template:

template<class CharType> struct char_traits

Qui CharType specifica il tipo del carattere.

La libreria C++ fornisce due specializzazioni di char_traits: una per caratteri char e una per caratteri wchar_t. La classe char_traits definisce i cinque tipi seguenti.

| | |
|---|---|
| char_type | Il tipo del carattere. Si tratta di un typedef per CharType. |
| int_type | Un tipo intero che può contenere un carattere di tipo char_type o il carattere EOF. |
| off_type | Un tipo intero che può rappresentare un offset in una stringa. |
| pos_type | Un tipo intero che può rappresentare una posizione in uno stream. |
| state_type | Un oggetto che contiene lo stato di una conversione (si applica a caratteri multibyte). |

La Tabella 36.2 elenca le funzioni membro della classe char_traits.

**Tabella 36.2** Le funzioni membro della classe char_traits.

| MEMBRO | DESCRIZIONE |
|---|---|
| `static void assign(char_type &ch1, const char_type &ch2);` | Assegna *ch2* a *ch1*. |
| `static char_type *assign(char_type *str, size_t num, char_type ch2);` | Assegna *ch2* ai primi *num* caratteri di *str*. Restituisce *str*. |

*(segue)*



**Tabella 36.2** Le funzioni membro della classe char_traits. *(continua)*

| MEMBRO | DESCRIZIONE |
|---|---|
| `static int compare(const char_type *str1, const char_type *str2, size_t num);` | Confronta *num* caratteri di *str1* con quelli di *str2*. Se le stringhe sono uguali restituisce zero. Altrimenti, restituisce un valore minore di zero se *str1* è minore di *str2* o maggiore di zero se *str1* è maggiore di *str2*. |
| `static char_type *copy(char_type *to, const char_type *from, size_t num);` | Copia *num* caratteri di *from* in *to*. Restituisce *to*. |
| `static int_type eof( );` | Restituisce il codice di fine file. |
| `static bool eq(const char_type &ch1, const char_type &ch2);` | Confronta *ch1* con *ch2* e restituisce `true` se i caratteri sono uguali e `false` altrimenti. |
| `static bool eq_int_type(const int_type &ch1, const int_type &ch2);` | Restituisce `true` se *ch1* è uguale a *ch2* e `false` altrimenti. |
| `static const char_type *find(const char_type *str, size_t num, const char_type *ch);` | Restituisce un puntatore alla prima occorrenza di *ch* in *str*. Vengono esaminati solo i primi *num* caratteri. In caso di fallimento restituisce un puntatore nullo. |
| `static size_t length(const char_type *str);` | Restituisce la lunghezza di *str*. |
| `static bool lt(const char_type &ch1, const char_type &ch2);` | Restituisce `true` se *ch1* è minore di *ch2* e `false` altrimenti. |
| `static char_type *move(char_type *to, const char_type *from, size_t num);` | Copia *num* caratteri di *from* in *to*. Restituisce *to*. |
| `static int_type not_eof(const int_type &ch);` | Se *ch* non è il carattere EOF, restituisce *ch*. Altrimenti, restituisce il carattere EOF. |
| `static state_type get_state(pos_type pos);` | Restituisce lo stato della conversione. |
| `static char_type to_char_type(const int_type &ch);` | Converte *ch* in un `char_type` e restituisce il risultato. |
| `static int_type to_int_type(const char_type &ch);` | Converte *ch* in un int_type e restituisce il risultato. |

Capitolo 37

# Le classi per numeri

37.1 **La classe complex**
37.2 **La classe valarray**
37.3 **Gli algoritmi numerici**

Una delle funzionalità aggiunte durante la standardizzazione del linguaggio C++ è la libreria di classi numeriche. Queste classi aiutano nello sviluppo di programmi numerici. Molte delle funzioni membro di queste classi corrispondono alle funzioni ereditate dalla libreria C. La differenza consiste nel fatto che molte delle funzioni numeriche descritte in questo capitolo operano su oggetti di tipo valarray che in pratica sono array di valori oppure su oggetti di tipo complex che rappresentano un numero complesso. Grazie all'introduzione delle classi numeriche, il C++ standard ha esteso le possibilità di programmazione in campo numerico.

## 37.1 La classe complex

La classe complex, che rappresenta numeri complessi, è definita dall'header <complex>. Tale file definisce anche una serie di funzioni e operatori applicabili a oggetti di tipo complex.

Ecco la specifica template di complex:

```
template <class T> class complex
```

Qui T specifica il tipo utilizzato per memorizzare i componenti di un numero complesso. Vi sono tre specializzazioni predefinite di complex:

```
class complex<float>
class complex<double>
class complex<long double>
```

La classe complex ha i seguenti costruttori:

```
complex(const T &real = T( ), const T &imaginary = T( ));
complex(const complex &ob);
template <class T1> complex(const complex<T1> &ob);
```

Il primo costruisce un oggetto complex con un componente reale e un componente immaginario. Se non vengono specificati questi valori assumono il valore standard 0 (zero). il secondo crea una copia di *ob*. Il terzo crea un oggetto complex a partire da *ob*.

Per gli oggetti complex sono definite le seguenti operazioni:

```
+       -       *       /
-=      +=      /=      *=
=       ==      !=
```

Gli operatori che non eseguono assegnamenti sono modificati tramite overloading in tre modi. Uno per operazioni riguardanti un oggetto complex a sinistra un oggetto scalare a destra, un altro per operazioni che riguardano un oggetto scalare a sinistra e un oggetto complex a destra e infine uno per operazioni che riguardano due oggetti complex. Ad esempio, sono consentite le seguenti operazioni:

```
complex_ob + scalar
scalar + complex_ob
complex_ob + complex_ob
```

Le operazioni che riguardano quantità scalari riguardano solo il componente reale. La classe complex definisce due funzioni membro: real() e imag() rappresentate di seguito:

```
T real( ) const;
T imag( ) const;
```

La funzione real() restituisce il componente reale dell'oggetto chiamante e ovviamente la funzione imag() restituisce il componente immaginario. Per gli oggetti complex sono definite le funzioni elencate nella Tabella 37.1.

Ecco un semplice programma che illustra l'uso della classe complex:

```
// Illustra l'uso della classe complex.
#include <iostream>
#include <complex>
using namespace std;

int main()
{
  complex<double> cmpx1(1, 0);
  complex<double> cmpx2(1, 1);

  cout << cmpx1 << " " << cmpx2 << endl;

  complex<double> cmpx3 = cmpx1 + cmpx2;
  cout << cmpx3 << endl;

  cmpx3 += 10;
  cout << cmpx3 << endl;

  return 0;
}
```

Ecco l'output del programma:

```
(1,0) (1,1)
(2,1)
(12,1)
```



**Tabella 37.1** Le funzioni definite dalla classe complex.

| FUNZIONE | DESCRIZIONE |
|---|---|
| `template <class T> T abs(const complex<T> &ob);` | Restituisce il valore assoluto di *ob*. |
| `template <class T> T arg(const complex<T> &ob);` | Restituisce l'angolo di fase di *ob*. |
| `template <class T> complex<T> conj(const complex<T> &ob);` | Restituisce il coniugato di *ob*. |
| `template <class T> complex<T> cos(const complex<T> &ob);` | Restituisce il coseno di *ob*. |
| `template <class T> complex<T> cosh(const complex<T> &ob);` | Restituisce il coseno iperbolico di *ob*. |
| `template <class T> complex<T> exp(const complex<T> &ob);` | Restituisce $e^{ob}$. |

(segue

**Tabella 37.1** Le funzioni definite dalla classe complex. *(continua)*

| FUNZIONE | DESCRIZIONE |
|---|---|
| `template <class T> T imag(const complex<T> &ob);` | Restituisce il componente immaginario di *ob*. |
| `template <class T> complex<T> log(const complex<T> &ob);` | Restituisce il logaritmo naturale di *ob*. |
| `template <class T> complex<T> log10(const complex<T> &ob);` | Restituisce il logaritmo in base 10 di *ob*. |
| `template <class T> T norm(const complex<T> &ob);` | Restituisce la magnitudine di *ob* elevato al quadrato. |
| `template <class T> complex<T> polar(const T &v, const T &theta=0);` | Restituisce un numero complesso con la magnitudine specificata da *v* e l'angolo di fase *theta*. |
| `template <class T> complex<T> pow(const complex<T> &b, int e);` | Restituisce $b^e$. |
| `template <class T> complex<T> pow(const complex<T> &b, const T &e);` | Restituisce $b^e$. |
| `template <class T> complex<T> pow(const complex<T> &b, const complex<T> &e);` | Restituisce $b^e$. |
| `template <class T> complex<T> pow(const T &b, const complex<T> &e);` | Restituisce $b^e$. |
| `template <class T> T real(const complex<T> &ob);` | Restituisce il componente reale di *ob*. |
| `template <class T> complex<T> sin(const complex<T> &ob);` | Restituisce il seno di *ob*. |
| `template <class T> complex<T> sinh(const complex<T> &ob);` | Restituisce il seno iperbolico di *ob*. |
| `template <class T> complex<T> sqrt(const complex<T> &ob);` | Restituisce la radice quadrata di *ob*. |
| `template <class T> complex<T> tan(const complex<T> &ob);` | Restituisce la tangente di *ob*. |

*(segue)*



**Tabella 37.1** Le funzioni definite dalla classe complex. *(continua)*

| FUNZIONE | DESCRIZIONE |
|---|---|
| `template <class T> complex<T> tanh(const complex<T> &ob);` | Restituisce la tangente iperbolica di *ob*. |

## 37.2 La classe valarray

L'header <valarray> definisce una serie di classi per il supporto degli array di numeri. La classe principale è valarray che definisce un array monodimensionale di valori. In tale classe è definita un'ampia varietà di operatori e funzioni membro e anche un gran numero di funzioni non membro. Anche se la descrizione di valarray che verrà fornita in questa sezione dovrebbe essere sufficiente per la maggior parte delle situazioni, coloro che fossero particolarmente interessati all'elaborazione numerica dovranno studiare valarray in maggior dettaglio. Un'ultima annotazione: anche se valarray è molto estesa, la maggior parte delle sue operazioni è piuttosto intuitiva.

La classe valarray ha la seguente specifica template:

template <class T> class valarray

La classe definisce i seguenti costruttori:

```
valarray( );
explicit valarray (size_t num);
valarray(const T &v, size_t num);
valarray(const T *ptr, size_t num);
valarray(const valarray<T> &ob);
valarray(const slice_array<T> &ob);
valarray(const gslice_array<T> &ob);
valarray(const mask_array<T> &ob);
valarray(const indirect_array<T> &ob);
```

Il primo costruttore crea un oggetto vuoto. Il secondo crea un valarray di lunghezza *num*. Il terzo crea un valarray di lunghezza *num* e inizializzato a *v*. Il quarto crea un valarray di lunghezza *num* e lo inizializza con gli elementi puntati da *ptr*. Il quinto crea una copia di *ob*. I quattro costruttori successivi creano un valarray a partire da una delle classi helper di valarray. Questi costruttori non vengono utilizzati direttamente dal programma ma vengono richiamati automaticamente quando si svolgono le operazioni definite da valarray.

Per valarray sono definiti i seguenti operatori:

| | | | |
|---|---|---|---|
| + | - | * | / |
| -= | += | /= | *= |
| = | == | != | << |
| ^= | % | %= | ~ |
| ! | \| | \|= | & |
| &= | [ ] | | |

Questi operatori hanno varie forme modificate tramite overloading che verranno descritte nelle relative tabelle.

La Tabella 32.3 elenca le funzioni e gli operatori membro definiti in valarray. La Tabella 37.3 elenca le funzioni operatore non membro definite da valarray. La Tabella 37.4 elenca le funzioni trascendentali definite per valarray.

**Tabella 37.2** Le funzioni membro di valarray.

| FUNZIONE | DESCRIZIONE |
|---|---|
| `valarray<T> apply(T func(T)) const;`<br>`valarray<T> apply(T func(const T &ob)) const;` | Applica *func( )* all'array chiamante e restituisce un array contenente il risultato. |
| `valarray<T> cshift(int num) const;` | Esegue una rotazione a sinistra di *num* posizioni dell'array chiamante (ovvero esegue uno scorrimento circolare a sinistra). Restituisce un array contenente il risultato. |
| `T max( ) const;` | Restituisce il valore più elevato contenuto nell'array chiamante. |
| `T min( ) const` | Restituisce il valore più basso contenuto nell'array chiamante. |
| `valarray<T> &operator=(const valarray<T> &ob);` | Assegna gli elementi di *ob* agli elementi corrispondenti nell'array chiamante. Restituisce l'indirizzo dell'array chiamante. |
| `valarray<T> &operator=(const T &v);` | Assegna a ciascun elemento dell'array chiamante il valore *v*. Restituisce l'indirizzo dell'array chiamante. |
| `valarray<T> &operator=(const slice_array<T> &ob);` | Assegna un sottoinsieme. Restituisce l'indirizzo dell'array chiamante. |
| `valarray<T> &operator=(const gslice_array<T> &ob);` | Assegna un sottoinsieme. Restituisce l'indirizzo dell'array chiamante. |
| `valarray<T> &operator=(const mask_array<T> &ob);` | Assegna un sottoinsieme. Restituisce l'indirizzo dell'array chiamante. |
| `valarray<T> &operator=(const indirect_array<T> &ob);` | Assegna un sottoinsieme. Restituisce l'indirizzo dell'array chiamante. |

(segue)



**Tabella 37.2** Le funzioni membro di valarray. *(continua)*

| FUNZIONE | DESCRIZIONE |
|---|---|
| `valarray<T> operator+( ) const;` | Più unario applicato a ciascun elemento dell'array chiamante. Restituisce l'array risultante. |
| `valarray<T> operator-( ) const;` | Meno unario applicato a ciascun elemento dell'array chiamante. Restituisce l'array risultante. |
| `valarray<T> operator~( ) const;` | NOT unario bit-a-bit applicato a ciascun elemento dell'array chiamante. Restituisce l'array risultante. |
| `valarray<T> operator!( ) const;` | NOT unario logico applicato a ciascun elemento dell'array chiamante. Restituisce l'array risultante. |
| `valarray<T> &operator+=(const T &v) const;` | Somma *v* a ciascun elemento dell'array chiamante. Restituisce l'indirizzo dell'array chiamante. |
| `valarray<T> &operator-=(const T &v) const;` | Sottrae *v* da ciascun elemento dell'array chiamante. Restituisce l'indirizzo dell'array chiamante. |
| `valarray<T> &operator/=(const T &v) const;` | Divide ciascun elemento dell'array chiamante per *v*. Restituisce l'indirizzo dell'array chiamante. |
| `valarray<T> &operator*=(const T &v) const;` | Moltiplica ciascun elemento dell'array chiamante per *v*. Restituisce l'indirizzo dell'array chiamante. |
| `valarray<T> &operator%=(const T &v) const;` | Assegna a ciascun elemento dell'array chiamante il resto della divisione per *v*. Restituisce l'indirizzo dell'array chiamante. |
| `valarray<T> &operator^=(const T &v) const;` | Esegue uno XOR di *v* con ciascun elemento dell'array chiamante. Restituisce l'indirizzo dell'array chiamante. |
| `valarray<T> &operator&=(const T &v) const;` | Esegue un AND di *v* con ciascun elemento dell'array chiamante. Restituisce l'indirizzo dell'array chiamante. |
| `valarray<T> &operator\|=(const T &v) const;` | Esegue un OR di *v* con ciascun elemento dell'array chiamante. Restituisce l'indirizzo dell'array chiamante. |
| `valarray<T> &operator<<(const T &v) const;` | Esegue uno scorrimento a sinistra di *v* posizioni di ciascun elemento dell'array chiamante. Restituisce l'indirizzo dell'array chiamante. |
| `valarray<T> &operator>>=(const T &v) const;` | Esegue uno scorrimento a destra di *v* posizioni di ciascun elemento dell'array chiamante. Restituisce l'indirizzo dell'array chiamante. |
| `valarray<T> &operator+=(const valarray<T> &ob) const;` | Somma gli elementi corrispondenti dell'array chiamante e di *ob*. Restituisce l'indirizzo dell'array chiamante. |
| `valarray<T> &operator-=(const valarray<T> &ob) const;` | Sottrae gli elementi di *ob* dagli elementi corrispondenti dell'array chiamante. Restituisce l'indirizzo dell'array chiamante. |

(segue)

**Tabella 37.2** Le funzioni membro di valarray. *(continua)*

| FUNZIONE | DESCRIZIONE |
|---|---|
| `valarray<T>& operator/=(const valarray<T> &ob) const;` | Divide gli elementi dell'array chiamante per gli elementi corrispondenti di *ob*. Restituisce l'indirizzo dell'array chiamante. |
| `valarray<T>& operator*=(const valarray<T> &ob) const;` | Moltiplica gli elementi corrispondenti dell'array chiamante e di *ob*. Restituisce l'indirizzo dell'array chiamante. |
| `valarray<T> operator%=(const valarray<T> &ob) const;` | Divide gli elementi dell'array chiamante per gli elementi corrispondenti di *ob* e memorizza il resto. Restituisce l'indirizzo dell'array chiamante. |
| `valarray<T> operator^=(const valarray<T> &ob) const;` | Applica l'operatore XOR agli elementi corrispondenti di *ob* e dell'array chiamante. Restituisce l'indirizzo dell'array chiamante. |
| `valarray<T> operator&=(const valarray<T> &ob) const;` | Applica l'operatore AND agli elementi corrispondenti di *ob* e dell'array chiamante. Restituisce l'indirizzo dell'array chiamante. |
| `valarray<T> operator\|=(const valarray<T> &ob) const;` | Applica l'operatore OR agli elementi corrispondenti di *ob* e dell'array chiamante. Restituisce l'indirizzo dell'array chiamante. |
| `valarray<T> operator<<=(const valarray<T> &ob) const;` | Gli elementi dell'array chiamante vengono fatti scorrere a sinistra del numero di posizioni specificate dagli elementi corrispondenti di *ob*. Restituisce l'indirizzo dell'array chiamante. |
| `valarray<T> operator>>=(const valarray<T> &ob) const;` | Gli elementi dell'array chiamante vengono fatti scorrere a destra del numero di posizioni specificate dagli elementi corrispondenti di *ob*. Restituisce l'indirizzo dell'array chiamante. |
| `T &operator[ ] (size_t indx) ;` | Restituisce l'indirizzo dell'elemento che si trova all'indice specificato. |
| `T operator[ ] (size_t indx) const;` | Restituisce il valore che si trova all'indice specificato. |
| `slice_array<T> operator[ ](slice ob);` | Restituisce il sottoinsieme specificato. |
| `valarray<T> operator[ ](slice ob) const;` | Restituisce il sottoinsieme specificato. |
| `gslice_array<T> operator[ ](const gslice &ob);` | Restituisce il sottoinsieme specificato. |
| `valarray<T> operator[ ](const gslice &ob) const;` | Restituisce il sottoinsieme specificato. |
| `mask_array<T> operator[ ](valarray<bool> &ob);` | Restituisce il sottoinsieme specificato. |
| `valarray<T> operator[ ](valarray<bool> &ob) const;` | Restituisce il sottoinsieme specificato. |

*(segue)*



**Tabella 37.2** Le funzioni membro di valarray. *(continua)*

| FUNZIONE | DESCRIZIONE |
|---|---|
| `indirect_array<T> operator[ ](const valarray<size_t> &ob);` | Restituisce il sottoinsieme specificato. |
| `valarray<T> operator[ ](const valarray<size_t> &ob) const;` | Restituisce il sottoinsieme specificato. |
| `void resize(size_t num, T v = T( ));` | Ridimensiona l'array chiamante. Se devono essere aggiunti degli elementi, gli viene assegnato il valore specificato da *v*. |
| `size_t size( ) const;` | Restituisce le dimensioni (in termini di numero di elementi) dell'array chiamante. |
| `valarray<T> shift(int num) const;` | Fa scorrere l'array chiamante verso sinistra di *num* posizioni. Restituisce un array contenente il risultato. |
| `T sum( ) const;` | Restituisce la somma dei valori contenuti nell'array chiamante. |

**Tabella 37.3** Le funzioni non membro definite per valarray.

| FUNZIONE | DESCRIZIONE |
|---|---|
| `template <class T> valarray<T> operator+(const valarray<T> ob, const T &v);` | Somma *v* a ciascun elemento di *ob*. Restituisce un array contenente il risultato. |
| `template <class T> valarray<T> operator+(const T &v, const valarray<T> ob);` | Somma *v* a ciascun elemento di *ob*. Restituisce un array contenente il risultato. |
| `template <class T> valarray<T> operator+(const valarray<T> ob1, const valarray<T> &ob2);` | Somma ciascun elemento di *ob1* con l'elemento corrispondente di *ob2*. Restituisce un array contenente il risultato. |
| `template <class T> valarray<T> operator-(const valarray<T> ob, const T &v);` | Sottrae *v* da ciascun elemento di *ob*. Restituisce un array contenente il risultato. |
| `template <class T> valarray<T> operator-(const T &v, const valarray<T> ob);` | Sottrae ciascun elemento di *ob* da *v*. Restituisce un array contenente il risultato. |
| `template <class T> valarray<T> operator-(const valarray<T> ob1, const valarray<T> &ob2);` | Sottrae ciascun elemento di *ob2* dall'elemento corrispondente di *ob1*. Restituisce un array contenente il risultato. |
| `template <class T> valarray<T> operator*(const valarray<T> ob, const T &v);` | Moltiplica ciascun elemento di *ob* per *v*. Restituisce un array contenente il risultato. |

*(segue)*

**Tabella 37.3** Le funzioni non membro definite per valarray. *(continua)*

| FUNZIONE | DESCRIZIONE |
|---|---|
| `template <class T> valarray<T> operator*(const T &v, const valarray<T> ob);` | Moltiplica ciascun elemento di *ob* per *v*. Restituisce un array contenente il risultato. |
| `template <class T> valarray<T> operator*(const valarray<T> ob1, const valarray<T> &ob2);` | Moltiplica gli elementi corrispondenti di *ob1* e *ob2*. Restituisce un array contenente il risultato. |
| `template <class T> valarray<T> operator/(const valarray<T> ob, const T &v);` | Divide ciascun elemento di *ob* per *v*. Restituisce un array contenente il risultato. |
| `template <class T> valarray<T> operator/(const T &v, const valarray<T> ob);` | Divide *v* per ciascun elemento di *ob*. Restituisce un array contenente il risultato. |
| `template <class T> valarray<T> operator/(const valarray<T> ob1, const valarray<T> &ob2);` | Divide ciascun elemento di *ob1* per l'elemento corrispondente di *ob2*. Restituisce un array contenente il risultato. |
| `template <class T> valarray<T> operator%(const valarray<T> ob, const T &v);` | Ottiene il resto della divisione di ciascun elemento di *ob* per *v*. Restituisce un array contenente il risultato. |
| `template <class T> valarray<T> operator%(const T &v, const valarray<T> ob);` | Ottiene il resto della divisione di *v* per ciascun elemento di *ob*. Restituisce un array contenente il risultato. |
| `template <class T> valarray<T> operator%(const valarray<T> ob1, const valarray<T> &ob2);` | Ottiene il resto della divisione di ciascun elemento di *ob1* per l'elemento corrispondente di *ob2*. Restituisce un array contenente il risultato. |
| `template <class T> valarray<T> operator^(const valarray<T> ob, const T &v);` | Applica l'operatore XOR fra ciascun elemento di *ob* e *v*. Restituisce un array contenente il risultato. |
| `template <class T> valarray<T> operator^(const T &v, const valarray<T> ob);` | Applica l'operatore XOR fra ciascun elemento di *ob* e *v*. Restituisce un array contenente il risultato. |
| `template <class T> valarray<T> operator^(const valarray<T> ob1, const valarray<T> &ob2);` | Applica l'operatore XOR fra ciascun elemento di *ob1* e il corrispondente elemento di *ob2*. Restituisce un array contenente il risultato. |
| `template <class T> valarray<T> operator&(const valarray<T> ob, const T &v);` | Applica l'operatore AND fra ciascun elemento di *ob* e *v*. Restituisce un array contenente il risultato. |
| `template <class T> valarray<T> operator&(const T &v, const valarray<T> ob);` | Applica l'operatore AND fra ciascun elemento di *ob* e *v*. Restituisce un array contenente il risultato. |

*(segue)*



**Tabella 37.3** Le funzioni non membro definite per valarray. *(continua)*

| FUNZIONE | DESCRIZIONE |
|---|---|
| `template <class T> valarray<T> operator&(const valarray<T> ob1, const valarray<T> &ob2);` | Applica l'operatore AND fra ciascun elemento di *ob1* e l'elemento corrispondente di *ob2*. Restituisce un array contenente il risultato. |
| `template <class T> valarray<T> operator\|(const valarray<T> ob, const T &v);` | Applica l'operatore OR fra ciascun elemento di *ob* e *v*. Restituisce un array contenente il risultato. |
| `template <class T> valarray<T> operator\|(const T &v, const valarray<T> ob);` | Applica l'operatore OR fra ciascun elemento di *ob* e *v*. Restituisce un array contenente il risultato. |
| `template <class T> valarray<T> operator\|(const valarray<T> ob1, const valarray<T> &ob2);` | Applica l'operatore OR fra ciascun elemento di *ob1* e l'elemento corrispondente di *ob2*. Restituisce un array contenente il risultato. |
| `template <class T> valarray<T> operator<<(const valarray<T> ob, const T &v);` | Fa scorrere a sinistra ciascun elemento di *ob* per il numero di posizioni specificate da *v*. Restituisce un array contenente il risultato. |
| `template <class T> valarray<T> operator<<(const T &v, const valarray<T> ob);` | Fa scorrere a sinistra *v* per il numero di posizioni specificate dagli elementi di *ob*. Restituisce un array contenente il risultato. |
| `template <class T> valarray<T> operator<<(const valarray<T> ob1, const valarray<T> &ob2);` | Fa scorrere a sinistra ciascun elemento di *ob1* per il numero di posizioni specificate dai corrispondenti elementi di *ob2*. Restituisce un array contenente il risultato. |
| `template <class T> valarray<T> operator>>(const valarray<T> ob, const T &v);` | Fa scorrere a destra ciascun elemento di *ob* per il numero di posizioni specificate da *v*. Restituisce un array contenente il risultato. |
| `template <class T> valarray<T> operator>>(const T &v, const valarray<T> ob);` | Fa scorrere a destra *v* per il numero di posizioni specificate dagli elementi di *ob*. Restituisce un array contenente il risultato. |
| `template <class T> valarray<T> operator>>(const valarray<T> ob1, const valarray<T> &ob2);` | Fa scorrere a destra ciascun elemento di *ob1* per il numero di posizioni specificate dai corrispondenti elementi di *ob2*. Restituisce un array contenente il risultato. |
| `template <class T> valarray<bool> operator==(const valarray<T> ob, const T &v);` | Per ogni i, esegue *ob*[i] == *v*. Restituisce un array booleano contenente il risultato. |
| `template <class T> valarray<bool> operator==(const T &v, const valarray<T> ob);` | Per ogni i, esegue *v* == *ob*[i]. Restituisce un array booleano contenente il risultato. |
| `template <class T> valarray<bool> operator==(const valarray<T> ob1, const valarray<T> &ob2);` | Per ogni i, esegue *ob1*[i] == *ob2*[i]. Restituisce un array booleano contenente il risultato. |

*(segue)*

**Tabella 37.3** Le funzioni non membro definite per valarray. *(continua)*

| FUNZIONE | DESCRIZIONE |
|---|---|
| `template <class T> valarray<bool> operator!=(const valarray<T> ob, const T &v);` | Per ogni i, esegue *ob*[i] != *v*. Restituisce un array booleano contenente il risultato. |
| `template <class T> valarray<bool> operator!=(const T &v, const valarray<T> ob);` | Per ogni i, esegue *v* != *ob*[i]. Restituisce un array booleano contenente il risultato. |
| `template <class T> valarray<bool> operator!=(const valarray<T> ob1, const valarray<T> &ob2);` | Per ogni i, esegue *ob1*[i] != *ob2*[i]. Restituisce un array booleano contenente il risultato. |
| `template <class T> valarray<bool> operator<(const valarray<T> ob, const T &v);` | Per ogni i, esegue *ob*[i] < *v*. Restituisce un array booleano contenente il risultato. |
| `template <class T> valarray<bool> operator<(const T &v, const valarray<T> ob);` | Per ogni i, esegue *v* < *ob*[i]. Restituisce un array booleano contenente il risultato. |
| `template <class T> valarray<bool> operator<(const valarray<T> ob1, const valarray<T> &ob2);` | Per ogni i, esegue *ob1*[i] < *ob2*[i]. Restituisce un array booleano contenente il risultato. |
| `template <class T> valarray<bool> operator<(const valarray<T> ob, const T &v);` | Per ogni i, esegue *ob*[i] <= *v*. Restituisce un array booleano contenente il risultato. |
| `template <class T> valarray<bool> operator<(const T &v, const valarray<T> ob);` | Per ogni i, esegue *v* <= *ob*[i]. Restituisce un array booleano contenente il risultato. |
| `template <class T> valarray<bool> operator<(const valarray<T> ob1, const valarray<T> &ob2);` | Per ogni i, esegue *ob1*[i] <= *ob2*[i]. Restituisce un array booleano contenente il risultato. |
| `template <class T> valarray<bool> operator>(const valarray<T> ob, const T &v);` | Per ogni i, esegue *ob*[i] > *v*. Restituisce un array booleano contenente il risultato. |
| `template <class T> valarray<bool> operator>(const T &v, const valarray<T> ob);` | Per ogni i, esegue *v* > *ob*[i]. Restituisce un array booleano contenente il risultato. |
| `template <class T> valarray<bool> operator>(const valarray<T> ob1, const valarray<T> &ob2);` | Per ogni i, esegue *ob1*[i] > *ob2*[i]. Restituisce un array booleano contenente il risultato. |
| `template <class T> valarray<bool> operator>=(const valarray<T> ob, const T &v);` | Per ogni i, esegue *ob*[i] >= *v*. Restituisce un array booleano contenente il risultato. |

*(segue)*

**Tabella 37.3** Le funzioni non membro definite per valarray. *(continua)*

| FUNZIONE | DESCRIZIONE |
|---|---|
| `template <class T> valarray<bool> operator>=(const T &v, const valarray<T> ob);` | Per ogni i, esegue *v* >= *ob*[i]. Restituisce un array booleano contenente il risultato. |
| `template <class T> valarray<bool> operator>=(const valarray<T> ob1, const valarray<T> &ob2);` | Per ogni i, esegue *ob1*[i] >= *ob2*[i]. Restituisce un array booleano contenente il risultato. |
| `template <class T> valarray<bool> operator&&(const valarray<T> ob, const T &v);` | Per ogni i, esegue *ob*[i] && *v*. Restituisce un array booleano contenente il risultato. |
| `template <class T> valarray<bool> operator&&(const T &v, const valarray<T> ob);` | Per ogni i, esegue *v* && *ob*[i]. Restituisce un array booleano contenente il risultato. |
| `template <class T> valarray<bool> operator&&(const valarray<T> ob1, const valarray<T> &ob2);` | Per ogni i, esegue *ob1*[i] && *ob2*[i]. Restituisce un array booleano contenente il risultato. |
| `template <class T> valarray<bool> operator\|\|(const valarray<T> ob, const T &v);` | Per ogni i, esegue *ob*[i] \|\| *v*. Restituisce un array booleano contenente il risultato. |
| `template <class T> valarray<bool> operator\|\|(const T &v, const valarray<T> ob);` | Per ogni i, esegue *v* \|\| *ob*[i]. Restituisce un array booleano contenente il risultato. |
| `template <class T> valarray<bool> operator\|\|(const valarray<T> ob1, const valarray<T> &ob2);` | Per ogni i, esegue *ob1*[i] \|\| *ob2*[i]. Restituisce un array booleano contenente il risultato. |

**Tabella 37.4** Le funzioni trascendentali definite per valarray.

| FUNZIONE | DESCRIZIONE |
|---|---|
| `template<class T> valarray<T> abs(const valarray<T> &ob);` | Ottiene il valore assoluto di ciascun elemento di *ob* e restituisce un array contenente il risultato. |
| `template<class T> valarray<T> acos(const valarray<T> &ob);` | Ottiene l'arcocoseno di ciascun elemento di *ob* e restituisce un array contenente il risultato. |
| `template<class T> valarray<T> asin(const valarray<T> &ob);` | Ottiene l'arcoseno di ciascun elemento di *ob* e restituisce un array contenente il risultato. |
| `template<class T> valarray<T> atan(const valarray<T> &ob);` | Ottiene l'arcotantente di ciascun elemento di *ob* e restituisce un array contenente il risultato. |
| `template<class T> valarray<T> atan2(const valarray<T> &ob1, const valarray<T> &ob2);` | Per ogni i, ottiene l'arcotancente di *ob1*[i] / *ob2*[i] e restituisce un array contenente il risultato. |

*(segue)*

**Tabella 37.4** Le funzioni trascendentali definite per valarray. *(continua)*

| FUNZIONE | DESCRIZIONE |
|---|---|
| `template<class T> valarray<T> atan2(const T &v, const valarray<T> &ob);` | Per ogni i, ottiene l'arcotangente di *v* / *ob1*[i] e restituisce un array contenente il risultato. |
| `template<class T> valarray<T> atan2(const valarray<T> &ob, const T &v);` | Per ogni i, ottiene l'arcotangente di *ob1*[i] / *v* e restituisce un array contenente il risultato. |
| `template<class T> valarray<T> cos(const valarray<T> &ob);` | Ottiene il coseno di ciascun elemento di *ob* e restituisce un array contenente il risultato. |
| `template<class T> valarray<T> cosh(const valarray<T> &ob);` | Ottiene il coseno iperbolico di ciascun elemento di *ob* e restituisce un array contenente il risultato. |
| `template<class T> valarray<T> exp(const valarray<T> &ob);` | Calcola la funzione esponenziale per ciascun elemento di *ob* e restituisce un array contenente il risultato. |
| `template<class T> valarray<T> log(const valarray<T> &ob);` | Ottiene il logaritmo naturale di ciascun elemento di *ob* e restituisce un array contenente il risultato. |
| `template<class T> valarray<T> log10(const valarray<T> &ob);` | Ottiene il logaritmo comune di ciascun elemento di *ob* e restituisce un array contenente il risultato. |
| `template<class T> valarray<T> pow(const valarray<T> &ob1, const valarray<T> &ob2);` | Per ogni i, calcola $ob1[i]^{ob2[i]}$ e restituisce un array contenente il risultato. |
| `template<class T> valarray<T> pow(const T &v, const valarray<T> &ob);` | Per ogni i, calcola $v^{ob[i]}$ e restituisce un array contenente il risultato. |
| `template<class T> valarray<T> pow(const valarray<T> &ob, const T &v);` | Per ogni i, calcola $ob1[i]^{v}$ e restituisce un array contenente il risultato. |
| `template<class T> valarray<T> sin(const valarray<T> &ob);` | Ottiene il seno di ciascun elemento di *ob* e restituisce un array contenente il risultato. |
| `template<class T> valarray<T> sinh(const valarray<T> &ob);` | Ottiene the il seno iperbolico di ciascun elemento di *ob* e restituisce un array contenente il risultato. |
| `template<class T> valarray<T> sqrt(const valarray<T> &ob);` | Ottiene la radice quadrata di ciascun elemento di *ob* e restituisce un array contenente il risultato. |
| `template<class T> valarray<T> tan(const valarray<T> &ob);` | Ottiene la tangente di ciascun elemento di *ob* e restituisce un array contenente il risultato. |
| `template<class T> valarray<T> tanh(const valarray<T> &ob);` | Ottiene la tangente iperbolica di ciascun elemento di *ob* e restituisce un array contenente il risultato. |

Il seguente programma illustra alcune delle molte funzionalità offerte da valarray.

```
// Illustra l'uso della classe valarray
#include <iostream>
#include <valarray>
#include <cmath>
using namespace std;

int main()
{
  valarray<int> v(10);
  int i;

  for(i=0; i<10; i++) v[i]  i;

  cout << "Contenuto originale: ";
  for(i=0; i<10; i++)
    cout << v[i] << " ";
  cout << endl;

  v = v.cshift(3);

  cout << "Contenuto dopo lo scorrimento: ";
  for(i=0; i<10; i++)
    cout << v[i] << " ";
  cout << endl;

  valarray<bool> vb = v < 5;
  cout << "Elementi minori di 5: ";
  for(i=0; i<10; i++)
    cout << vb[i] << " ";
  cout << endl << endl;

  valarray<double> fv(5);
  for(i=0; i<5; i++) fv[i]  (double) i;

  cout << "Contenuto originale: ";
  for(i=0; i<5; i++)
    cout << fv[i] << " ";
  cout << endl;

  fv = sqrt(fv);

  cout << "Radici quadrate: ";
  for(i=0; i<5; i++)
    cout << fv[i] << " ";
  cout << endl;

  fv = fv + fv;
  cout << "Doppio delle radici quadrate: ";
```

```
  for(i=0; i<5; i++)
    cout << fv[i] << " ";
  cout << endl;

  fv = fv - 10.0;
  cout << "Dopo aver sottratto 10 da ciascun elemento:";
  for(i=0; i<5; i++)
    cout << fv[i] << " ";
  cout << endl;

  return 0;
}
```

Ecco l'output prodotto dal programma:

```
Contenuto originale: 0 1 2 3 4 5 6 7 8 9
Contenuto dopo lo scorrimento: 3 4 5 6 7 8 9 0 1 2
Elementi minori di 5: 1 1 0 0 0 0 0 1 1 1

Contenuto originale: 0 1 2 3 4
Radici quadrate: 0 1 1.41421 1.73205 2
Doppio delle radici quadrate: 0 2 2.82843 3.4641 4
Dopo aver sottratto 10 da ciascun elemento:
-10 -8 -7.17157 -6.5359 -6
```

## Le classi slice e gslice

L'header <valarray> definisce due classi di servizio chiamate slice e gslice che incapsulano una porzione di un array. Queste classi sono utilizzate con le forme per sottoinsiemi dell'operatore [ ] di valarray.

Ecco l'aspetto della classe slice.

```
class slice {
public:
  slice();
  slice(size_t start, size_t len, size_t interval);
  size_t start() const;
  size_t size() const;
  size_t stride();
};
```

Il primo costruttore crea una slice vuota. Il secondo costruttore crea una slice che specifica l'elemento di partenza, il numero di elementi e l'intervallo fra gli elementi (*stride*). Questi sono i valori restituiti dalle funzioni membro.



Ecco un programma che illustra l'uso della classe slice.

```
// Illustra l'uso della classe slice
#include <iostream>
#include <valarray>
using namespace std;

int main()
{
  valarray<int> v(10), result;
  int i;

  for(i=0; i<10; i++) v[i]  i;

  cout << "Contenuto di v: ";
  for(i=0; i<10; i++)
    cout << v[i] << " ";
  cout << endl;

  result = v[slice(0,5,2)];

  cout << "Contenuto di result: ";
  for(i=0; i<result.size(); i++)
    cout << result[i] << " ";

  return 0;
}
```

Ecco l'output del programma:

```
Contenuto di v: 0 1 2 3 4 5 6 7 8 9
Contenuto di result: 0 2 4 6 8
```

Come si può vedere, l'array prodotto è costituito da cinque elementi di v che iniziano da 0 e procedono di due elementi alla volta.

Questa è l'aspetto della classe gslice:

```
class gslice {
public:
  gslice();
  gslice()(size_t start, const valarray<size_t> &lens,
           const valarray<size_t> &intervals);
  size_t start()-const;
```

```
  valarray<size_t> size() const;
  valarray<size_t> stride() const;
};
```

Il primo costruttore crea una gslice vuota. Il secondo costruttore crea una gslice che specifica l'elemento iniziale, un array che specifica il numero di elementi e un array che specifica l'intervallo fra gli elementi (*stride*). Le quantità delle lunghezze e degli interalli devono coincidere. Le funzioni membro restituiscono questi parametri. Questa classe è utilizzata per creare array multidimensionali a partire da un valarray (che è sempre monodimensionale).

Il seguente programma illustra l'uso di gslice.

```
// Illustra il funzionamento di gslice()
#include <iostream>
#include <valarray>
using namespace std;

int main()
{
  valarray<int> v(12), result;
  valarray<size_t> len(2), interval(2);
  int i;

  for(i=0; i<12; i++) v[i]  i;

  len[0] = 3; len[1] = 3;
  interval[0] = 2; interval[1] = 3;

  cout << "Contenuto di v: ";
  for(i=0; i<12; i++)
    cout << v[i] << " ";
  cout << endl;

  result = v[gslice(0,len,interval)];

  cout << "Contenuto di result: ";
  for(i=0; i<result.size(); i++)
    cout << result[i] << " ";

  return 0;
}
```

Ecco l'output del programma:



```
Contenuto di v: 0 1 2 3 4 5 6 7 8 9 10 11
Contenuto di result: 0 3 6 2 5 8 4 7 10
```

### Le classi helper

Le classi numeriche utilizzano alcune classi di supporto che non vengono mai istanziate direttamente dal programma: slice_array, gslice_array, indirect_array e mask_array.

## 37.3 Gli algoritmi numerici

L'header <numeric> definisce quattro algoritmi numerici utilizzabili nell'elaborazione del contenuto dei container.

### L'algoritmo accumulate()

L'algoritmo accumulate() calcola la somma di tutti gli elementi che si trovano all'interno dell'intervallo specificato e restituisce il risultato. Ecco i suoi prototipi:

```
template <class InIter, class T> T accumulate(InIter start, InIter end, T v);
template <class InIter, class T, class BinFunc>
        T accumulate(InIter start, InIter end, T v, BinFunc func);
```

Qui, T è il tipo dei valori su cui si opera. La prima versione calcola la somma di tutti gli elementi contenuti nell'intervallo compreso fra *start* ed *end*. La seconda versione applica *func* al totale (in pratica *func* specifica il modo in cui deve essere eseguita la somma). Il valore di *v* fornisce un valore iniziale a cui deve essere sommato il totale.

Ecco un esempio d'uso dell'algoritmo accumulate().

```
// Illustra l'uso di accumulate()
#include <iostream>
#include <vector>
#include <numeric>
using namespace std;

int main()
{
  vector<int> v(5);
  int i, total;
```

```
  for(i=0; i<5; i++) v[i]  i;

  total = accumulate(v.begin(), v.end(), 0);

  cout << "Somma di v: " << total;

  return 0;
}
```

Il programma produce il seguente output:

```
Somma di v: 10
```

## L'algoritmo adjacent_difference()

L'algoritmo adjacent_difference() produce una nuova sequenza in cui ciascun elemento è la differenza fra elementi adiacenti della sequenza originale (il primo elemento del risultato è uguale al primo elemento originale). Ecco i prototipi per adjacent_difference():

```
template <class InIter, class OutIter>
   outIter adjacent_difference(InIter start, InIter end, OutIter result);
template <class InIter, class OutIter, class BinFunc>
   outIter adjacent_difference(InIter start, InIter end, OutIter result,
                               BinFunc func);
```

Qui *start* ed *end* sono iteratori che puntano all'inizio e alla fine della sequenza originale. La sequenza risultante viene memorizzata nella sequenza puntata da *result*. Nella prima forma gli elementi adiacenti vengono sottratti, ovvero l'elemento nella posizione *n* viene sottratto dall'elemento che si trova alla posizione *n* + 1. Nella seconda agli elementi adiacenti si applica la funzione binaria *func*. Viene restituito un iteratore che punta alla fine di *result*.

Ecco un esempio che utilizza adjacent_difference().

```
// Illustra l'uso di adjacent_difference()
#include <iostream>
#include <vector>
#include <numeric>
using namespace std;

int main()
{
  vector<int> v(10), r(10);
  int i;

  for(i=0; i<10; i++) v[i]  i*2;
  cout << "Sequenza originale: ";
  for(i=0; i<10; i++)
    cout << v[i] << " ";
  cout << endl;

  adjacent_difference(v.begin(), v.end(), r.begin());

  cout << "Sequenza risultante: ";
  for(i=0; i<10; i++)
    cout << r[i] << " ";

  return 0;
}
```



Il programma produce il seguente output:

```
Sequenza originale: 0 2 4 6 8 10 12 14 16 18
Sequenza risultante: 0 2 2 2 2 2 2 2 2 2
```

Come si può vedere, la sequenza risultante contiene la differenza fra il valore degli elementi adiacenti.

## L'algoritmo inner_product()

L'algoritmo inner_product() produce la somma del prodotto degli elementi corrispondenti in due sequenze e restituisce il risultato. Ecco i suoi prototipi:

```
template <class InIter1, class InIter2, class T>
   T inner_product(InIter1 start1, InIter1 end1, InIter2 start2, T v);
template <class InIter1, class InIter2, class T, class BinFunc1, class
BinFunc2>
   T inner_product(InIter1 start1, InIter1 end1, InIter2 start2, T v,
                   BinFunc1 func2, BinFunc2 func2);
```

Qui *start1* ed *end1* sono iteratori che puntano all'inizio e alla fine della prima sequenza. L'iteratore *start2* punta all'inizio della seconda sequenza. Il valore *v* fornisce il valore iniziale cui deve essere sommato il totale. Nella seconda forma *func1* specifica una funzione binaria che determina il modo in cui viene calcolato

il totale e *func2* specifica una funzione binaria che determina il modo in cui vengono moltiplicate due sequenze.

Ecco un programma che illustra l'uso di inner_product().

```
// Illustra l'uso di inner_product()
#include <iostream>
#include <vector>
#include <numeric>
using namespace std;

int main()
{
  vector<int> v1(5), v2(5);
  int i, total;

  for(i=0; i<5; i++) v1[i]  i;
  for(i=0; i<5; i++) v2[i]  i+2;

  total = inner_product(v1.begin(), v1.end(),
                        v2.begin(), 0);

  cout << "Il prodotto interno è: " << total;

  return 0;
}
```

Ecco l'output prodotto:

```
Il prodotto interno è: 50
```

## L'algoritmo partial_sum()

L'algoritmo partial_sum() somma una sequenza di valori inserendo il totale corrente nei successivi elementi di una nuova sequenza (ovvero crea una sequenza costituita dai totali della sequenza originaria). Il primo elemento del risultato sarà uguale al primo elemento della sequenza originaria. Ecco i prototipi di partial_sum():

```
template <class InIter, class OutIter>
    OutIter partial_sum(InIter start, InIter end, OutIter result);
template <class InIter, class OutIter, class BinFunc>
    OutIter partial_sum(InIter start, InIter end, OutIter result,
                        BinFunc func);
```



Qui *start1* ed *end1* sono iteratori che puntano all'inizio e alla fine della sequenza originaria. L'iteratore *result* punta all'inizio della sequenza risultante. Nella seconda forma, *func* specifica una funzione binaria che determina il modo in cui viene calcolato il totale. Viene restituito un iteratore che punta alla fine di *result*.

Ecco un esempio di uso di partial_sum().

```
// Illustra l'uso di partial_sum()
#include <iostream>
#include <vector>
#include <numeric>
using namespace std;

int main()
{
  vector<int> v(5), r(5);
  int i;

  for(i=0; i<10; i++) v[i]  i;
  cout << "Sequenza originale: ";
  for(i=0; i<5; i++)


    cout << v[i] << " ";
  cout << endl;

  partial_sum(v.begin(), v.end(), r.begin());

  cout << "Sequenza risultante: ";
  for(i=0; i<5; i++)
    cout << r[i] << " ";

  return 0;
}
```

Ecco dall'output prodotto:

```
Sequenza originale: 0 1 2 3 4
Sequenza risultante: 0 1 3 6 10
```

Capitolo 38

# Le classi per la gestione delle eccezioni


38.1 **Le eccezioni**
38.2 **La classe auto_ptr**
38.3 **La classe pair**
38.4 **La localizzazione**
38.5 **Altre classi interessanti**


Questo capitolo descrive le classi per la gestione delle eccezioni. Inoltre descrive le classi auto_ptr e pair e fornisce una rapida introduzione alla libreria di localizzazione.

## 38.1 Le eccezioni

La libreria standard C++ definisce due header strettamente correlati con le eccezioni: <exception> e <stdexcept>. Le eccezioni sono utilizzate per rilevare una condizione di errore. Questa sezione esamina entrambi questi header.

### L'header <exception>

L'header <exception> definisce le classi, i tipi e le funzioni che sono in relazione con la gestione delle eccezioni. Ecco le classi definite da <exception>.

```
class exception {
public:
  exception() throw();
  exception(const bad_exception &ob) throw();
  virtual ~exception() throw();

  exception &operator=(const exception &ob) throw();
  virtual const char *what(() const throw();
};

class bad_exception: public exception {
```

prietà dell'oggetto. Il vantaggio principale di questo approccio è il fatto che gli oggetti allocati dinamicamente possono essere distrutti quando viene gestita un'eccezione.

Ecco la specifica template della classe auto_ptr:

```
template <class T> class auto_ptr
```

Qui T specifica il tipo del puntatore contenuto in auto_ptr.
Ecco l'aspetto dei costruttori di auto_ptr:

```
explicit auto_ptr(T *ptr = 0) throw( );

auto_ptr(const auto_ptr &ob) throw( );

template <class T2> auto_ptr(const auto_ptr<T2> &ob) throw( );
```

Il primo costruttore crea un auto_ptr che punta all'oggetto specificato da *ptr*. Il secondo costruttore crea una copia del puntatore auto_ptr specificato da *ob* e trasferisce la proprietà al nuovo oggetto. Il terzo converte (se possibile) *ob* nel tipo T e trasferisce la proprietà.

La classe auto_ptr definisce gli operatori =, * e -> e le seguenti funzioni membro:

```
T *get( ) const throw( );

T *release( ) const throw( );
```

La funzione get() restituisce un puntatore all'oggetto memorizzato. La funzione release() rimuove la proprietà dell'oggetto memorizzato dall'auto_ptr chiamante e restituisce un puntatore all'oggetto. Dopo una chiamata a release(), l'oggetto puntato non verrà più distrutto automaticamente quando l'oggetto auto_ptr uscirà dal campo di visibilità.

Ecco un breve programma che illustra l'uso di auto_ptr.

```
// Illustra l'uso di auto_ptr.
#include <iostream>
#include <memory>
using namespace std;

class X {
public:
  X() { cout << "costruzione\n"; }
  ~X() { cout << "distruzione\n"; }
  void f() { cout << "All'interno di f()\n"; }
};

int main()
{
  auto_ptr<X> p1(new X), p2;

  p2 = p1; // trasferisce la proprietà
  p2->f();

  // assegnamento a un normale puntatore
  X *ptr = p2.get();
  ptr->f();

  return 0;
}
```



Ecco l'output prodotto dal programma:

```
costruzione
All'interno di f()
All'interno di f()
distruzione
```

Si noti che la funzione f() membro di X può essere richiamata tramite auto_ptr o tramite il normale puntatore restituito da get().

## 38.3 La classe pair

La classe pair viene utilizzata per memorizzare coppie di oggetti, come se si trattasse di un container associativo. La classe ha la seguente specifica template:

```
template <class Ktype, class Vtype> struct pair {
  typedef Ktype first_type;
  typedef Vtype second_type;
  Ktype first;
  Vtype second;

  // costruttori
  pair();
```

```
  pair(const Ktype &k, const Vtype &v);
  template<class A, class B> pair(const<A, B> &ob);
}
```

In genere il valore di first contiene una chiave e il valore di second contiene il valore associato a tale chiave. Per pair sono definiti i seguenti operatori: ==, !=, <=, >= e >.

Si può costruire una coppia pair utilizzando uno dei suoi costruttori oppure la funzione make_pair() che costruisce un oggetto pair sulla base di tipi utilizzati come parametri. make_pair() è una funzione generica che ha il seguente prototipo:

```
template <class <Ktype, class Vtype>
  pair<Ktype, Vtype>> make_pair(const Ktype &k, const Vtype &v);
```

Come si può vedere restituisce un oggetto pair costituito dai valori dei tipi specificati da Ktype e Vtype. make_pair() ha il vantaggio che i tipi degli oggetti memorizzati vengono determinati automaticamente dal compilatore e non devono essere specificati esplicitamente.

La classe pair e la funzione make_pair() richiedono l'uso dell'header <utility>.

## 38.4 La localizzazione

Il C++ standard fornisce un'estesa libreria di classi di localizzazione. Queste classi consentono a un'applicazione di impostare oppure ottenere informazioni relative all'ambiente geopolitico in cui verrà eseguito il programma. Pertanto le classi definiscono ad esempio il formato della valuta, della data e dell'ora e l'ordine utilizzato per questi valori. Inoltre forniscono la classificazione dei caratteri. La libreria di localizzazione utilizza l'header <locale> e opera tramite una serie di classi che definiscono varie informazioni associate a locale. Tutti questi elementi derivano dalla classe facet che è contenuta nella classe locale.

Onestamente la libreria di localizzazione è straordinariamente estesa e complessa e la descrizione delle sue funzionalità non rientra negli scopi di questo volume.



## 38.5 Altre classi interessanti

Ecco alcune classe molto interessanti definite dalla libreria standard C++.

| CLASSE | DESCRIZIONE |
|---|---|
| type_info | Utilizzata insieme all'operatore typeid e descritta in modo approfondito nel Capitolo 22. Usa l'header <typeinfo>. |
| numeric_limts | Incapsula vari limiti numerici. Usa l'header <limits>. |
| raw_storage_iterator | Incapsula l'allocazione di memoria non inizializzata. Usa l'header <memory>. |

Parte quinta

# APPLICAZIONI C++

La Parte quinta di questo volume presenta due applicazioni C++ d'esempio. Lo scopo di questa sezione è duplice. Innanzitutto gli esempi aiutano a illustrare i vantaggi della programmazione a oggetti. In secondo luogo mostrano come il linguaggio C++ possa essere applicato alla soluzione di due problemi di programmazione molto differenti.

Capitolo 39

# Integrazione delle nuove classi: una classe personalizzata per le stringhe


39.1 **La classe StrType**
39.2 **Le funzioni costruttore e distruttore**
39.3 **Operazioni di I/O di stringhe**
39.4 **Le funzioni di assegnamento**
39.5 **Il concatenamento**
39.6 **Sottrazione di sottostringhe**
39.7 **Gli operatori relazionali**
39.8 **Funzioni varie**
39.9 **L'intera classe StrType**
39.10 **Uso della classe StrType**
39.11 **Creazione e integrazione di nuovi tipi**
39.12 **Un esercizio**


Questo capitolo descrive e implementa una piccola classe per stringhe. Come si sa, il linguaggio C++ fornisce già una classe per stringhe molto potente chiamata basic_string. Lo scopo di questo capitolo non è quello di sviluppare un'alternativa a tale classe ma piuttosto quello di dare al lettore un esempio del modo in cui si può aggiungere e integrare nell'ambiente C++ un nuovo tipo di dati. La creazione di una classe per stringhe è un esempio classico in questo senso. Nel passato, molti programmatori hanno affinato le proprie capacità sviluppando una classe per stringhe realizzata con tecniche a oggetti. In questo capitolo si proverà a eseguire la stessa operazione.

Anche se la classe per stringhe sviluppata in questo capitolo è molto più semplice di quella fornita dal C++ standard, presenta un grande vantaggio: offre un controllo completo sul modo in cui le stringhe vengono implementate e manipolate. Si troverà che questa classe, oltre a essere un esercizio divertente, può risultare utile in varie situazioni.

## 39.1 La classe StrType

La classe StrType prende a modello quella fornita dalla libreria standard. Naturalmente non sarà così estesa o sofisticata. La classe per stringhe definita in questo capitolo avrà i seguenti requisiti.

- L'assegnamento di stringhe avverrà tramite l'operatore di assegnamento.
- A un oggetto stringa sarà possibile assegnare altri oggetti stringa o stringhe quotate.
- Il concatenamento di due oggetti stringa sarà ottenuto con l'operatore +.
- La cancellazione di sottostringhe verrà eseguita tramite l'operatore -.
- Il confronto fra stringhe verrà eseguito tramite operatori relazionali.
- Gli oggetti stringa saranno inizializzati con una stringa quotata o un oggetto stringa.
- Le stringhe potranno avere lunghezza arbitraria e variabile. Questo significa che la memoria delle stringhe sarà allocata in modo dinamico.
- Verrà fornito un metodo per convertire gli oggetti stringa in stringhe chiuse dal carattere nullo.

Anche se la classe per stringhe che verrà sviluppata sarà meno potente della classe standard, includerà una funzionalità che non è definita da basic_string: la cancellazione di una sottostringa con l'operatore -.

La classe per la gestione di stringhe si chiamerà StrType. Ecco la sua dichiarazione:

```
class StrType {
  char *p;
  int size;
public:
  StrType();
  StrType(char *str);
  StrType(const StrType &o); // costruttore di copie

  ~StrType() { delete [] p; }

  friend ostream &operator<<(ostream &stream, trType &o);
  friend istream &operator>>(istream &stream, StrType &o);

  StrType operator=(StrType &o); // assegna un oggetto StrType
  StrType operator=(char *s); // assegna una stringa quotata
```



```
  StrType operator+(StrType &o); // concatena un oggetto StrType
  StrType operator+(char *s); // concatena una stringa quotata
  friend StrType operator+(char *s, StrType &o); /* concatena
              una stringa quotata e un oggetto StrType */

  StrType operator-(StrType &o); // sottrae una sottostringa
  StrType operator-(char *s); // sottrae una sottostringa quotata

  // operazioni relazionali fra oggetti StrType
  int operator==(StrType &o) { return !strcmp(p, o.p); }
  int operator!=(StrType &o) { return strcmp(p, o.p); }
  int operator<(trType &o) { return strcmp(p, o.p) < 0; }
  int operator>(StrType &o) { return strcmp(p, o.p) > 0; }
  int operator<=(trType &o) { return strcmp(p, o.p) < 0; }
  int operator>=(StrType &o) { return strcmp(p, o.p) >= 0; }

  // operazioni fra oggetti StrType e stringhe quotate
  int operator==(char *s) { return !strcmp(p, s); }
  int operator!=(char *s) { return strcmp(p, s); }
  int operator<(char *s) { return strcmp(p, s) < 0; }
  int operator>(char *s) { return strcmp(p, s) > 0; }
  int operator<=(char *s) { return strcmp(p, s) < 0; }
  int operator>=(char *s) { return strcmp(p, s) >= 0; }

  int strsize() { return strlen(p); } // restituisce le dimensioni della
stringa
  void makestr(char *s) { strcpy(s, p); } // crea una stringa quotata

  operator char *() { return p; } // conversione in char *
};
```

La parte privata di StrType contiene due soli elementi: p e size. Quando viene creato un oggetto StrType, la memoria che conterrà la stringa viene riallocata dinamicamente con new e in p viene inserito un puntatore a tale area di memoria. La stringa puntata da p sarà un normale array di caratteri chiuso dal carattere nullo. Anche se tecnicamente questo non è necessario, le dimensioni della stringa sono contenute in size. Poiché la stringa puntata da p è una normale stringa chiusa dal carattere nullo, sarebbe possibile calcolarne le dimensioni senza problemi. Ma come si vedrà, questo valore viene utilizzato così frequentemente dalle funzioni membro di StrType da non giustificare un continuo ricorso alla funzione strlen().

Le prossime sezioni descrivono il funzionamento della classe StrType.

## 39.2 Le funzioni costruttore e distruttore

Un oggetto StrType può essere dichiarato in tre diversi modi: senza inizializzazione, tramite una stringa quotata oppure con un oggetto StrType. Ecco i costruttori che supportano queste tre operazioni:

```
// Nessuna inizializzazione esplicita.
StrType::StrType() {
  size = 1; // crea spazio per il terminatore nullo
  try {
    p = new char[size];
  } catch (bad_alloc xa) {
    cout << "Errore di allocazione\n";
    exit(1);
  }
  strcpy(p, "");
}

// Inizializza con una stringa quotata.
StrType::StrType(char *str) {
  size = strlen(str) + 1; // crea spazio per il terminatore nullo
  try {
    p = new char[size];
  } catch (bad_alloc xa) {
    cout << "Errore di allocazione\n";
    exit(1);
  }
  strcpy(p, str);
}

// Inizializza con un oggetto StrType.
StrType::StrType(const StrType &o) {
  size = o.size;
  try {
    p = new char[size];
  } catch (bad_alloc xa) {
    cout << "Errore di allocazione\n";
    exit(1);
  }
  strcpy(p, o.p);
}
```

Quando un oggetto StrType viene creato senza inizializzatori, gli viene assegnata una stringa nulla. Anche se la stringa sarebbe potuta rimanere non definita, il fatto di sapere che tutti gli oggetti StrType contengono una stringa chiusa dal carattere nullo, semplifica le altre funzioni membro.



Quando un oggetto StrType viene inizializzato con una stringa quotata, innanzitutto si determinano le dimensioni della stringa. Questo valore viene memorizzato in size. Poi tramite new viene allocata un'area di memoria sufficiente e la stringa di inizializzazione viene copiata nella memoria puntata da p.

Quando l'inizializzazione dell'oggetto StrType avviene tramite un altro oggetto StrType, le operazioni sono simili a quelle svolte nel caso della stringa quotata. L'unica differenza è il fatto che le dimensioni della stringa sono note e dunque non devono essere calcolate. Questa versione del costruttore StrType è anche il costruttore di copie della classe. Questo costruttore viene richiamato ogni volta che si inizializza un oggetto StrType tramite un altro oggetto StrType. Questo significa che verrà richiamato quando dovranno essere creati oggetti temporanei e quando alle funzioni verranno passati oggetti di tipo StrType. Per una descrizione dei costruttori di copie, consultare il Capitolo 14.

I costruttori appena realizzati consentono dunque di utilizzare le seguenti dichiarazioni:

```
StrType x("stringa"); // usa una stringa quotata
StrType y(x); // usa un altro oggetto
StrType z; // nessuna inizializzazione esplicita
```

Il distruttore di StrType non fa altro che liberare la memoria puntata da p.

## 39.3 Operazioni di I/O di stringhe

Poiché le operazioni di input e output di stringhe sono molto comuni, la classe StrType non può fare a meno di eseguire l'overloading degli operatori << e >>:

```
// Produce in output una stringa.
ostream &operator<<(ostream &stream, trType &o)
{
  stream << o.p;
  return stream;
}

// Legge una stringa.
istream &operator>>(istream &stream, StrType &o)
{
  char t[255]; // dimensioni arbitrarie - modificabile a piacere
  int len;
```

```
  stream.getline(t, 255);
  len = strlen(t) + 1;

  if(len > o.size)
    delete [] o.p;
    try {
      o.p = new char[len];
    } catch (bad_alloc xa)
      cout << "Errore di allocazione\n";
      exit(1);
    }
    o.size = len;
  }
  strcpy(o.p, t);
  return stream;
}
```

Come si può vedere, l'output è molto semplice. Si deve solo notare che il parametro o viene passato per indirizzo. Poiché gli oggetti StrType possono essere piuttosto estesi, il passaggio per indirizzo può essere molto più efficiente rispetto al passaggio per valore. Per questo motivo, tutti i parametri StrType vengono passati per indirizzo (così come avviene per ogni funzione che accetti parametri StrType).

L'input di una stringa è un po' più difficoltoso. Innanzitutto si deve leggere la stringa utilizzando la funzione getline(). La lunghezza massima della stringa è limitata a 254 caratteri più il terminatore nullo. Come indicato dai commenti, questo valore può essere modificato a piacere. I caratteri vengono letti fino al codice di fine riga. Dopo aver letto la stringa, se le sue dimensioni superano il valore attualmente contenuto in o, tale area di memoria viene cancellata e viene allocata un'area di maggiori dimensioni. Quindi vi viene copiata la nuova stringa.

## 39.4 Le funzioni di assegnamento

L'assegnamento di un oggetto StrType può avvenire in due modi: si può assegnare un altro oggetto di tipo StrType oppure si può assegnare una stringa quotata. Ecco l'overloading delle funzioni operator=() che eseguono queste operazioni:

```
// Assegna un oggetto StrType a un oggetto StrType.
StrType StrType::operator=(StrType &o)
{
  StrType temp(o.p);
```



```
  if(o.size > size) {
    delete [] p; // libera la vecchia memoria
    try {
      p = new char[o.size];
    } catch (bad_alloc xa) {
      cout << "Errore di allocazione\n";
      exit(1);
    }
    size = o.size;
  }

  strcpy(p, o.p);
  strcpy(temp.p, o.p);

  return temp;
}

// Assegna una stringa quotata a un oggetto StrType.
StrType StrType::operator=(char *s)
{
  int len = strlen(s) + 1;
  if(size < len) {
    delete [] p;
    try {
      p = new char[len];
    } catch (bad_alloc xa) {
      cout << "Errore di allocazione\n";
      exit(1);
    }
    size = len;
  }
  strcpy(p, s);
  return *this;
}
```

Queste due funzioni controllano innanzitutto che la memoria attualmente puntata da p (dell'oggetto StrType di destinazione) sia sufficiente per contenere ciò che vi verrà copiato. In caso negativo, la vecchia memoria viene rilasciata e viene allocata nuova memoria. Poi la stringa viene copiata nell'oggetto e viene restituito il risultato. Queste funzioni consentono i seguenti assegnamenti:

```
StrType x("test"), y;

y = x; // Da oggetto StrType a oggetto StrType
```

```
x = "nuova stringa per x"; // da stringa quotata a oggetto StrType
```

Ogni funzione di assegnamento deve restituire il valore assegnato (ovvero il valore destro) in modo da poter eseguire assegnamenti multipli:

```
StrType x, y, z;

x = y = z = "test";
```

## 39.5 Il concatenamento

Il concatenamento di due stringhe viene eseguito impiegando l'operatore +. La classe StrType deve rispondere a tre diverse situazioni di concatenamento.

- Concatenamento di un oggetto StrType e di un altro oggetto StrType.
- Concatenamento di un oggetto StrType e di una stringa quotata.
- Concatenamento di una stringa quotata e di un oggetto StrType.

Quando è utilizzato in queste situazioni, l'operatore + produce come risultato un oggetto StrType formato dal concatenamento dei suoi due operandi. I due operandi non vengono modificati in alcun modo.

Ecco l'aspetto dell'overloading delle funzioni operator+():

```
// Concatena due oggetti StrType.
StrType StrType::operator+(StrType &o)
{
  int len;
  StrType temp;

  delete [] temp.p;
  len = strlen(o.p) + strlen(p) + 1;
  temp.size = len;
  try {
    temp.p = new char[len];
  } catch (bad_alloc xa) {
    cout << "Errore di allocazione\n";
    exit(1);
  }
  strcpy(temp.p, p);

  strcat(temp.p, o.p);

  return temp;
}
```

```
// Concatena un oggetto StrType e una stringa quotata.
StrType StrType::operator+(char *s)
{
  int len;
  StrType temp;

  delete [] temp.p;

  len = strlen(s) + strlen(p) + 1;
  temp.size = len;
  try {
    temp.p = new char[len];
  } catch (bad_alloc xa) {
    cout << "Errore di allocazione\n";
    exit(1);
  }
  strcpy(temp.p, p);

  strcat(temp.p, s);

  return temp;
}

// Concatena una stringa quotata e un oggetto StrType.
StrType operator+(char *s, StrType &o)
{
  int len;
  StrType temp;

  delete [] temp.p;

  len = strlen(s) + strlen(o.p) + 1;
  temp.size = len;
  try {
    temp.p = new char[len];
  } catch (bad_alloc xa) {
    cout << "Errore di allocazione\n";
    exit(1);
  }
  strcpy(temp.p, s);

  strcat(temp.p, o.p);

  return temp;
}
```

Queste tre funzioni si comportano fondamentalmente allo stesso modo. Innanzitutto creano un oggetto temporaneo di tipo StrType chiamato temp. Questo oggetto conterrà il risultato del concatenamento e sarà l'oggetto restituito dalle funzioni. Quindi la memoria puntata da temp.p viene liberata. Il motivo di ciò è che quando si crea temp, viene allocato un solo byte di memoria in quanto non vi è un'inizializzazione esplicita. Successivamente viene allocata un'area di memoria sufficiente per contenere il concatenamento delle due stringhe. Infine le due stringhe vengono copiate nella memoria puntata da temp.p e viene restituito il valore di temp.

## 39.6 Sottrazione di sottostringhe

Una funzione molto utile ma assente in basic_string è la sottrazione di sottostringhe. Così come viene implementata dalla classe StrType, la sottrazione di sottostringhe rimuove da una stringa tutte le occorrenze della sottostringa specificata. La sottrazione di sottostringhe è ottenuta tramite l'operatore -.

La classe StrType supporta due casi di sottrazione. Una consente la sottrazione di un oggetto StrType da un altro oggetto StrType e l'altra consente di sottrarre da un oggetto StrType una stringa quotata. Ecco l'aspetto delle due funzioni operator-():

```
// Sottrae una sottostringa da una stringa utilizzando oggetti StrType.
StrType StrType::operator-(StrType &substr)
{
  StrType temp(p);
  char *s1;
  int i, j;

  s1 = p; _
  for(i=0; *s1; i++) {
    if(*s1!=*substr.p) { // se non è la prima lettera della sottostringa
      temp.p[i] = *s1;  // la copia in temp
      s1++;
    }
    else {
      for(j=0; substr.p[j]==s1[j] && substr.p[j]; j++) ;
      if(!substr.p[j]) { // è una sottostringa, la cancella
        s1 += j;
        i--;
      }
      else { // non è una sottostringa, continua a copiare
        temp.p[i] = *s1;
```



```
        s1++;
      }
    }
  }
  temp.p[i] = '\0';
  return temp;
}

// Sottrae la stringa quotata da un oggetto StrType.
StrType StrType::operator-(char *substr)
{
  StrType temp(p);
  char *s1;
  int i, j;

  s1 = p;
  for(i=0; *s1; i++) {
    if(*s1!=*substr) { // se non è la prima lettera della sottostringa
      temp.p[i] = *s1; // la copia in temp
      s1++;
    }
    else {
      for(j=0; substr[j]==s1[j] && substr[j]; j++) ;
      if(!substr[j]) { // è una sottostringa, la cancella
        s1 += j;
        i--;
      }
      else { // non è una sottostringa, continua a copiare
        temp.p[i] = *s1;
        s1++;
      }
    }
  }
  temp.p[i] = '\0';
  return temp;
}
```

Queste due funzioni copiano il contenuto dell'operando sinistro in temp rimuovendo tutte le occorrenze della sottostringa specificata dall'operando di destra. Quindi restituiscono l'oggetto StrType risultante. Nessuno degli operandi viene modificato.

La classe StrType consente dunque l'esecuzione di sottrazioni come le seguenti:

```
StrType x("Mi piace il C++"), y("piace");
StrType z;
```

```
z = x - y; // z conterrà "Mi il C++"

z = x - "C++"; // z conterrà "Mi il "

// rimozione dei duplicati
z = "ABCDABCD";
x = z -"A"; // x conterrà "BCDBCD"
```

## 39.7 Gli operatori relazionali

La classe StrType supporta tutti gli operatori relazionali. Tali operatori sono definiti nella dichiarazione della classe ma verranno ripetuti qui per facilitare la consultazione:

```
// operazioni relazionali fra oggetti StrType
int operator==(StrType &o) { return !strcmp(p, o.p); }
int operator!=(StrType &o) { return strcmp(p, o.p); }
int operator<(trType &o) { return strcmp(p, o.p) < 0; }
int operator>(StrType &o) { return strcmp(p, o.p) > 0; }
int operator<=(trType &o) { return strcmp(p, o.p) <= 0; }
int operator>=(StrType &o) { return strcmp(p, o.p) >= 0; }

// operazioni fra oggetti StrType e stringhe quotate
int operator==(char *s) { return !strcmp(p, s); }
int operator!=(char *s) { return strcmp(p, s); }
int operator<(char *s) { return strcmp(p, s) < 0; }
int operator>(char *s) { return strcmp(p, s) > 0; }
int operator<=(char *s) { return strcmp(p, s) <= 0; }
int operator>=(char *s) { return strcmp(p, s) >= 0; }
```

Gli operatori relazionali sono molto semplici; non dovrebbe essere difficile capire come sono stati implementati. Tuttavia si deve tenere in considerazione che la classe StrType implementa il confronto fra due oggetti StrType oppure confronti che usano un oggetto StrType come operando sinistro e una stringa quotata come operando destro. Per poter eseguire confronti ponendo la stringa quotata a sinistra e l'oggetto StrType a destra sarà dunque necessario aggiungere un nuovo gruppo di funzioni relazionali.

Date le funzioni per operatori relazionali definite da StrType, è possibile eseguire i seguenti confronti:

```
StrType x("uno"), y("due"), z("tre");

if(x < y) cout << "x è minore di y";

if(z=="tre")  cout << "z è uguale a tre";

y = "u";
z = "no";
if(x==(y+z)) cout << "x è uguale a y+z";
```



## 39.8 Funzioni varie

La classe StrType definisce tre funzioni che consentono di integrare meglio gli oggetti StrType nell'ambiente di programmazione C++. Si tratta delle funzioni strsize() e makestr() e della funzione di conversione operator char *(). Queste funzioni sono definite nella dichiarazione di StrType e hanno il seguente aspetto:

```
int strsize() { return strlen(p); } // restituisce le dimensioni della stringa
void makestr(char *s) { strcpy(s, p); } // crea una stringa quotata
operator char *() { return p; } // conversione in char *
```

Le prime due funzioni sono molto semplici. Come si può vedere, la funzione strsize() restituisce la lunghezza della stringa puntata da p. Poiché l'effettiva lunghezza della stringa può essere differente rispetto al valore contenuto nella variabile size (ad esempio a causa dell'assegnamento di una stringa più breve) la lunghezza viene calcolata richiamando strlen(). La funzione makestr() copia in un array di caratteri la stringa puntata da p. Questa funzione è utile quando si vuole ottenere da un oggetto StrType una stringa chiusa dal carattere NULL.

La funzione di conversione operator char *() restituisce p che ovviamente è un puntatore alla stringa contenuta nell'oggetto. Questa funzione consente di utilizzare gli oggetti StrType in tutte le situazione in cui è necessario impiegare una stringa chiusa dal carattere nullo. Ad esempio l'oggetto StrType potrà essere utilizzato nel seguente modo:

```
StrType x("Salve");
char s[20];

// copia un oggetto stringa con la funzione strcpy()
strcpy(s, x); // conversione automatica in char *
```

Si ricordi che la funzione di conversione viene eseguita automaticamente quando un oggetto StrType viene impiegato in un'espressione per la quale è definita la

conversione. In questo caso, poiché il prototipo della funzione strcpy() dice al compilatore che il suo secondo argomento deve essere di tipo char *, viene automaticamente eseguita una conversione da StrType a char * e questo provoca la restituzione di un puntatore alla stringa contenuta in x. Questo puntatore verrà quindi utilizzato da strcpy() per copiare la stringa in s. Grazie alla funzione di conversione, si potrà utilizzare un oggetto StrType al posto della stringa chiusa dal carattere nullo come argomento per qualsiasi funzione che prenda un argomento di tipo char *.

**NOTA** *La conversione a char * inganna l'incapsulazione poiché una volta che una funzione ottiene un puntatore alla stringa contenuta nell'oggetto, tale funzione potrà modificare direttamente la stringa, scavalcando le funzioni membro di StrType e all'insaputa dell'oggetto puntato. Per questo motivo la conversione in char * deve essere utilizzata con cautela. La perdita di incapsulazione in questo caso è controbilanciata dalla maggiore utilità e integrazione con le altre funzioni della libreria. Tuttavia non sempre è consigliabile seguire questo approccio.*

## 39.9 L'intera classe StrType

Ecco il listato completo della classe StrType e una breve funzione main() che illustra le sue funzionalità:

```
#include <iostream>
#include <new>
#include <cstring>
#include <cstdlib>
using namespace std;

class StrType {
  char *p;
  int size;
public:
  StrType();
  StrType(char *str);
  StrType(const StrType &o); // costruttore di copie

  ~StrType() { delete [] p; }

  friend ostream &operator<<(ostream &stream, StrType &o);
  friend istream &operator>>(istream &stream, StrType &o);
```



```
  StrType operator=(StrType &o); // assegna un oggetto StrType
  StrType operator=(char *s); // assegna una stringa quotata

  StrType operator+(StrType &o); // concatena un oggetto StrType
  StrType operator+(char *s); // concatena una stringa quotata
  friend StrType operator+(char *s, StrType &o); /* concatena
                una stringa quotata e un oggetto StrType */

  StrType operator-(StrType &o); // sottrae una sottostringa
  StrType operator-(char *s); // sottrae una sottostringa quotata

  // operazioni relazionali fra oggetti StrType
  int operator==(StrType &o) { return !strcmp(p, o.p); }
  int operator!=(StrType &o) { return strcmp(p, o.p); }
  int operator<(StrType &o) { return strcmp(p, o.p) < 0; }
  int operator>(StrType &o) { return strcmp(p, o.p) > 0; }
  int operator<=(StrType &o) { return strcmp(p, o.p) <= 0; }
  int operator>=(StrType &o) { return strcmp(p, o.p) >= 0; }

  // operazioni fra oggetti StrType e stringhe quotate
  int operator==(char *s) { return !strcmp(p, s); }
  int operator!=(char *s) { return strcmp(p, s); }
  int operator<(char *s) { return strcmp(p, s) < 0; }
  int operator>(char *s) { return strcmp(p, s) > 0; }
  int operator<=(char *s) { return strcmp(p, s) <= 0; }
  int operator>=(char *s) { return strcmp(p, s) >= 0; }

  int strsize() { return strlen(p); } // restituisce le dimensioni
                                          della stringa
  void makestr(char *s) { strcpy(s, p); } // stringa chiusa
                                             dal carattere nullo
  operator char *() { return p; } // conversione in char *
};

// Nessuna inizializzazione esplicita.
StrType::StrType() {
  size = 1; // crea spazio per il terminatore nullo
  try {
    p = new char[size];
  } catch (bad_alloc xa) {
    cout << "Errore di allocazione\n";
    exit(1);
  }
  strcpy(p, "");
}
```

```
// Inizializza con una stringa quotata.
StrType::StrType(char *str) {
  size = strlen(str) + 1; // crea spazio per il terminatore nullo
  try {
    p = new char[size];
  } catch (bad_alloc xa) {
    cout << "Errore di allocazione\n";
    exit(1);
  }
  strcpy(p, str);
}

// Inizializza con un oggetto StrType.
StrType::StrType(const StrType &o) {
  size = o.size;
  try {
    p = new char[size];
  } catch (bad_alloc xa) {
    cout << "Errore di allocazione\n";
    exit(1);
  }
  strcpy(p, o.p);
}

// Produce in output una stringa.
ostream &operator<<(ostream &stream, StrType &o)
{
  stream << o.p;
  return stream;
}

// Legge una stringa.
istream &operator>>(istream &stream, StrType &o)
{
  char t[255]; // dimensioni arbitrarie - modificabile a piacere
  int len;

  stream.getline(t, 255);
  len = strlen(t) + 1;

  if(len > o.size) {
    delete [] o.p;
    try {
      o.p = new char[len];
    } catch (bad_alloc xa) {
      cout << "Errore di allocazione\n";
```



```
      exit(1);
    }
    o.size = len;
  }
  strcpy(o.p, t);
  return stream;
}

// Assegna un oggetto StrType a un oggetto StrType.
StrType StrType::operator=(StrType &o)
{
  StrType temp(o.p);

  if(o.size > size) {
    delete [] p; // libera la vecchia memoria
    try {
      p = new char[o.size];
    } catch (bad_alloc xa) {
      cout << "Errore di allocazione\n";
      exit(1);
    }
    size = o.size;
  }

  strcpy(p, o.p);
  strcpy(temp.p, o.p);

  return temp;
}

// Assegna una stringa quotata a un oggetto StrType.
StrType StrType::operator=(char *s)
{
  int len = strlen(s) + 1;
  if(size < len) {
    delete [] p;
    try {
      p = new char[len];
    } catch (bad_alloc xa) {
      cout << "Errore di allocazione\n";
      exit(1);
    }
    size = len;
  }
  strcpy(p, s);
```

```
  return *this;
}

// Concatena due oggetti StrType.
StrType StrType::operator+(StrType &o)
{
  int len;
  StrType temp;

  delete [] temp.p;
  len = strlen(o.p) + strlen(p) + 1;
  temp.size = len;
  try {
    temp.p = new char[len];
  } catch (bad_alloc xa) {
    cout << "Errore di allocazione\n";
    exit(1);
  }
  strcpy(temp.p, p);

  strcat(temp.p, o.p);

  return temp;
}

// Concatena un oggetto StrType e una stringa quotata.
StrType StrType::operator+(char *s)
{
  int len;
  StrType temp;

  delete [] temp.p;

  len = strlen(s) + strlen(p) + 1;
  temp.size = len;
  try {
    temp.p = new char[len];
  { catch (bad_alloc xa) {
    cout << "Errore di allocazione\n";
    exit(1);
  }
  strcpy(temp.p, p);

  strcat(temp.p, s);
```



```
  return temp;
}

// Concatena una stringa quotata e un oggetto StrType.
StrType operator+(char *s, StrType &o)
{
  int len;
  StrType temp;

  delete [] temp.p;

  len = strlen(s) + strlen(o.p) + 1;
  temp.size = len;
  try {
    temp.p = new char[len];
  } catch (bad_alloc xa) {
    cout << "Errore di allocazione\n";
    exit(1);
  }
  strcpy(temp.p, s);

  strcat(temp.p, o.p);

  return temp;
}

// Sottrae una sottostringa da una stringa utilizzando oggetti StrType.
StrType StrType::operator-(StrType &substr)
{
  StrType temp(p);
  char *s1;
  int i, j;

  s1 = p;
  for(i=0; *s1; i++) {
    if(*s1!=*substr.p) { // se non è la prima lettera della sottostringa
      temp.p[i] = *s1;  // la copia in temp
      s1++;
    }
    else {
      for(j=0; substr.p[j]==s1[j] && substr.p[j]; j++) ;
      if(!substr.p[j]) // è una sottostringa, la cancella
        s1 += j;
        i--;
      }
```

```
      else { // non è una sottostringa, continua a copiare
        temp.p[i] = *s1;
        s1++;
      }
    }
  }
  temp.p[i] = '\0';
  return temp;
}

// Sottrae la stringa quotata da un oggetto StrType.
StrType StrType::operator-(char *substr)
{
  StrType temp(p);
  char *s1;
  int i, j;

  s1 = p;
  for(i=0; *s1; i++) {
    if(*s1!=*substr) { // se non è la prima lettera della sottostringa
      temp.p[i] = *s1; // la copia in temp
      s1++;
    }
    else {
      for(j=0; substr[j]==s1[j] && substr[j]; j++) ;
      if(!substr[j]) { // è una sottostringa, la cancella
        s1 += j;
        i--;
      }
      else { // non è una sottostringa, continua a copiare
        temp.p[i] = *s1;
        s1++;
      }
    }
  }
  temp.p[i] = '\0';
  return temp;
}

int main()
{
  StrType s1("Una sessione d'esempio sull'uso di oggetti stringa.\n");
  StrType s2(s1);
  StrType s3;
  char s[80];
```



```
  cout << s1 << s2;

  s3 = s1;
  cout << s1;

  s3.makestr(s);
  cout << "Converte in stringa: " << s;

  s2 = "Questa è una nuova stringa.";
  cout << s2 << endl;

  StrType s4(" Anche questa.");
  s1 = s2+s4;
  cout << s1 << endl;

  if(s2==s3) cout << "Stringhe uguali.\n";
  if(s2!=s3) cout << "Stringhe differenti.\n";
  if(s1<s4) cout << "s1 è minore di s4\n";
  if(s1>s4) cout << "s1 è maggiore di s4\n";
  if(s1<=s4) cout << "s1 è minore o uguale a s4\n";
  if(s1>=s4) cout << "s1 è maggiore o uguale a s4\n";

  if(s2 > "ABC") cout << "s2 è maggiore di ABC\n\n";

  s1 = "uno due tre uno due tre\n";
  s2 = "due";
  cout << "Stringa iniziale: " << s1;
  cout << "Stringa dopo aver sottratto due: ";
  s3 = s1 - s2;
  cout << s3;

  cout << endl;
  s4 = "Salve!";
  s3 = s4 + " Le stringhe C++ sono divertenti\n";
  cout << s3;
  s3 = s3 - "Salve!";
  s3 = "Non siete d'accordo che" + s3;
  cout << s3;

  s1 = s3 - "Non ";
  cout << s1;
  s3 = s1;

  cout << "Introdurre una stringa: ";
  cin >> s1;
  cout << s1 << endl;
```

```
  cout << "s1 è lunga " << s1.strsize() << " caratteri.\n";

  puts(s1); // converte in to char *

  s1 = s2 = s3;
  cout << s1 << s2 << s3;

  s1 = s2 = s3 = "Arrivederci ";
  cout << s1 << s2 << s3;

  return 0;
}
```

Il programma produce il seguente output:

```
Una sessione d'esempio sull'uso di oggetti stringa.
Una sessione d'esempio sull'uso di oggetti stringa.
Una sessione d'esempio sull'uso di oggetti stringa.
Converte in stringa: Una sessione d'esempio sull'uso di oggetti stringa.
Questa è una nuova stringa.
Questa è una nuova stringa. Anche questa.
Stringhe differenti.
s1 è maggiore di s4
s1 è maggiore o uguale a s4
s2 è maggiore di ABC

Stringa iniziale: uno due tre uno due tre
Stringa dopo aver sottratto due: uno tre uno tre

Salve! Le stringhe C++ sono divertenti
Non siete d'accordo che Le stringhe C++ sono divertenti
siete d'accordo che Le stringhe C++ sono divertenti
Introdurre una stringa: Ma che bello il C++
s1 è lunga 19 caratteri.
Ma che bello il C++
siete d'accordo che Le stringhe C++ sono divertenti
siete d'accordo che Le stringhe C++ sono divertenti
siete d'accordo che Le stringhe C++ sono divertenti
Arrivederci Arrivederci Arrivederci
```

Qui si suppone che l'utente immetta la stringa "Ma che bello il C++".

Per accedere con facilità alla classe StrType si deve rimuovere la funzione main() e inserire il listato nel file STR.H. Poi basterà includere questo file header in qualsiasi programma nel quale si voglia utilizzare StrType.



## 39.10 Uso della classe StrType

Per concludere questo capitolo verranno presentati due brevi esempi che illustrano l'uso della classe StrType. Come si vedrà, grazie agli operatori definiti per la classe e grazie alla funzione di conversione in char *, la classe StrType è ben integrata nell'ambiente di programmazione C++, ovvero può essere utilizzata come qualsiasi altro tipo definito dal C++ standard.

Il primo esempio usa oggetti StrType per creare un semplice dizionario dei sinonimi. Innanzitutto crea un array bidimensionale di oggetti StrType. All'interno di ogni coppia di stringhe, la prima contiene la parola chiave che può essere ricercata e la seconda contiene una serie di parole alternative o correlate. Il programma chiede una parola e, se la parola è contenuta nel suo elenco, visualizza le alternative. Questo programma è molto semplice ma si noti la pulizia e la chiarezza con la quale vengono gestite le stringhe grazie alla classe StrType e ai suoi operatori. Si ricordi che il file header deve contenere la classe StrType.

```
#include "str.h"
#include <iostream>
using namespace std;

StrType thesaurus[][2] = {
  "libro", "volume, tomo",
  "negozio", "mercante, bottega, spaccio",
  "pistola", "rivoltella, arma, mano armata",
  "corsa", "gara, trotto, competizione",
  "pensare", "ispirarsi, contemplare, riflettere",
  "calcolare", "analizzare, determinare, risolvere"
  "", ""
};

int main()
{
  StrType x;

  cout << "Introdurre una parola: ";
  cin >> x;

  int i;
  for(i=0; thesaurus[i][0]!=""; i++)
    if(thesaurus[i][0]==x) cout << thesaurus[i][1];

  return 0;
}
```

Il prossimo esempio usa un oggetto StrType per controllare se esiste una versione eseguibile di un programma. Per utilizzare il programma si deve specificare nella riga di comando il nome del file senza estensione. Il programma cerca un file eseguibile avente tale nome aggiungendo via via una nuova estensione, poi tenta di aprire tale file e mostra il risultato. Se il file non esiste, ovviamente non può essere aperto. Dopo aver tentato un'estensione, questa viene sottratta dal nome del file, al quale viene aggiunta una nuova estensione. Anche in questo caso, l'uso della classe StrType e dei suoi operatori semplifica notevolmente la manipolazione delle stringhe.

```
#include "str.h"
#include <iostream>
#include <fstream>
using namespace std;

// estensioni dei file eseguibili
char ext[3][4] = {
  "EXE",
  "COM",
  "BAT"
};

int main(int argc, char *argv[])
{
  StrType fname;
  int i;

  if(argc!=2) {
    cout << "esempio: nomefile\n";
    return 1;
  }

  fname = argv[1];

  fname = fname + "."; // aggiunge il punto
  for(i=0; i<3; i++) {
    fname = fname + ext[i]; // aggiunge l'estensione
    cout << "Ricerca di " << fname << " ";
    ifstream f(fname);
    if(f) {
      cout << "- Esiste\n";
      f.close();
    }
    else cout << "- Non esiste\n";
    fname = fname - ext[i]; // sottrae l'estensione
  }

  return 0;
}
```



Ad esempio, se il programma si chiama ISEXEC e supponendo che esista TEST.EXE, l'esecuzione del comando ISEXEC TEST produrrà il seguente output:

```
Ricerca di TEST.EXE - Esiste
Ricerca di TEST.COM - Non esiste
Ricerca di TEST.BAT - Non esiste
```

Un'annotazione riguardante questo programma: il costruttore ifstream utilizza un oggetto StrType. Questa tecnica funziona grazie alla funzione di conversione char *() che viene richiamata automaticamente. Come illustrato da questa situazione, un'attenta applicazione delle caratteristiche del linguaggio C++ consente di raggiungere una notevole integrazione tra i tipi standard del C++ e i tipi creati dal programmatore.

## 39.11 Creazione e integrazione di nuovi tipi

Come dimostra l'esempio della classe StrType, è piuttosto facile creare e integrare un nuovo tipo nell'ambiente C++. In generale si deve seguire questa procedura.

1. Eseguire l'overloading degli operatori appropriati, inclusi gli operatori di I/O.
2. Definire tutte le funzioni di conversione appropriate.
3. Fornire i costruttori che consentono di creare gli oggetti in varie situazioni.

L'estendibilità è un elemento fondamentale della potenza del linguaggio C++. Dunque è importante saperla sfruttare in modo appropriato.

## 39.12 Un esercizio

Ecco un esercizio interessante. Si può provare a implementare la classe StrType utilizzando la libreria STL. Per contenere i caratteri che formano la stringa si potrà usare un container. Per operare sulla stringa si potranno usare gli iteratori e per eseguire la manipolazione della stringa si potranno usare gli algoritmi.

Capitolo 40

# Un analizzatore di espressioni realizzato con tecniche a oggetti


40.1 **Le espressioni**

40.2 **L'elaborazione delle espressioni: il problema**

40.3 **Analisi di un'espressione**

40.4 **La classe parser**

40.5 **Sezionamento di un'espressione**

40.6 **Un semplice parser di espressioni**

40.7 **Aggiunta delle variabili**

40.8 **Controllo della sintassi in un parser a discesa ricorsiva**

40.9 **Realizzazione di un parser generico**

40.10 **Alcune estensioni da provare**


Anche se il C++ standard è piuttosto esteso, esistono anche elementi che il linguaggio non fornisce. In questo capitolo si parlerà di uno di essi: il *parser di espressioni*. Un parser di espressioni può essere utilizzato per valutare un'espressione algebrica, come (10 - 8) * 3. I parser sono piuttosto utili e possono essere impiegati in un'ampia varietà di applicazioni. Sono anche una delle entità più elusive della programmazione. Per vari motivi, le procedure utilizzate per creare un parser non sono state molto sviluppate e diffuse. Inoltre molti programmatori abili in altri campi vengono intimiditi dal processo di analisi delle espressioni.

L'analisi delle espressioni è in realtà piuttosto semplice, talvolta molto più semplice di altri campi della programmazione. Ciò è dovuto al fatto che l'analisi di espressione è un argomento ben definito che deve seguire le rigide regole dell'algebra. Questo capitolo svilupperà un parser a discesa ricorsiva e tutte le routine di supporto necessarie per valutare espressioni numeriche complesse. Verranno create tre versioni del parser. Le prime due sono versioni non-generiche e l'ultima è generica e può essere applicata a qualsiasi tipo numerico. Ma prima di poter sviluppare un parser, è necessaria una breve introduzione alle espressioni e alla loro analisi.

## 40.1 Le espressioni

Poiché un parser valuta espressioni algebriche, è importante comprendere quali sono gli elementi che costituiscono le espressioni. Anche se esistono vari tipi di espressioni, questo capitolo si concentra sulle sole espressioni numeriche. Per quanto riguarda questo capitolo, le espressioni numeriche sono costituite dai seguenti elementi:

- i numeri
- gli operatori +, -, *, /, ^, % e =
- le parentesi
- le variabili

In questo parser, l'operatore ^ indica un elevamento a potenza (e non l'operatore XOR come in C++) e = è l'operatore di assegnamento. Questi elementi possono essere combinati per creare espressioni secondo le regole classiche dell'algebra. Ecco alcuni esempi:

```
10 – 8
(100 – 5) * 14/6
a + b – c
10^5
a = 10 – b
```

Per gli operatori verranno considerate le seguenti regole di precedenza:

| | |
|---|---|
| Alta | + - (unari) |
| | ^ |
| | * / % |
| | + - |
| Bassa | = |

Gli operatori aventi lo stesso livello di precedenza si valutano da sinistra a destra.

Negli esempi contenuti in questo capitolo, tutte le variabili sono costituite da un'unica lettera (ovvero è possibile impiegare fino a 26 variabili, da A a Z). Per le variabili non viene fatta alcuna distinzione fra lettere maiuscole e minuscole (a e A vengono trattate come la stessa variabile). Per la prima versione del parser, tutti i valori numerici vengono elevati a double anche se è possibile scrivere con facilità delle routine in grado di gestire altri tipi di valori. Infine, per mantenere la chiarezza e la semplicità della logica del parser, verrà incluso un livello molto limitato di verifica degli errori.



## 40.2 L'elaborazione delle espressioni: il problema

Chi non abbia mai riflettuto sul problema dell'analisi delle espressioni, potrebbe pensare che si tratti di un'operazione piuttosto semplice. Per comprendere meglio il problema, si provi a valutare questa semplice espressione:

```
10 - 2 * 3
```

Come si sa, il risultato di questa espressione è 4. Sebbene sia possibile generare con facilità un programma che calcoli questa *specifica* espressione, il problema è creare un programma che dia la risposta corretta per qualsiasi espressione. Inizialmente si potrebbe pensare a una routine come la seguente:

```
a = leggi il primo operando
while(ci sono altri operandi) {
    op = leggi l'operatore
    b = leggi il secondo operando
    a = a op b
}
```

Questa routine legge il primo operando, l'operatore e il secondo operando e quindi esegue la prima operazione, poi legge l'operatore e l'operando successivi, esegue l'operazione seguente e così via. Tuttavia, se si impiega questo semplice approccio, l'espressione 10 - 2 * 3 fornirà il valore 24 (ovvero 8 * 3) invece di 4, in quanto questa procedura non tiene conto della precedenza degli operatori. Non si possono considerare gli operandi e gli operatori semplicemente da sinistra a destra poiché le regole dell'algebra stabiliscono che la moltiplicazione debba essere eseguita prima della sottrazione. Qualcuno potrebbe pensare che questo problema possa essere risolto con facilità e talvolta, in casi molto limitati, questo è vero. Ma il problema peggiora se si aggiungono le parentesi, l'elevamento a potenza, le variabili, gli operatori unari e così via.

Anche se vi sono vari modi per scrivere una routine di valutazione delle espressioni, quella sviluppata in questo capitolo ha il pregio di essere scritta con facilità e infatti è molto comune. Il metodo utilizzato è quello del parser a discesa ricorsiva e nel corso del capitolo si scopriranno i motivi di questo nome. Altri metodi utilizzati per realizzare parser impiegano tabelle complesse che devono essere generate da un programma distinto. In questi casi si parla di parser a tabelle.

## 40.3 Analisi di un'espressione

Vi sono vari modi per analizzare e valutare un'espressione. Se si usa un parser a discesa ricorsiva, le espressioni possono essere considerate come strutture dati ricorsive, ovvero come espressioni definite in termini di altre espressioni. Se, per il momento, si immagina che le espressioni possano utilizzare solo gli operatori +, -, *, / e le parentesi, tutte le espressioni possono essere definite tramite le seguenti regole:

espressione –> termine [+ termine] [– termine]
termine –> fattore [* fattore] [/ fattore]
fattore –> variabile, numero o (espressione)

Le parentesi quadre indicano un elemento opzionale mentre la freccia -> si pronuncia "*produce*". Per questo motivo tali regole sono normalmente chiamate regole di produzione delle espressioni. Pertanto, la definizione di *termine* si può leggere nel seguente modo: "termine produce fattore per fattore oppure fattore diviso fattore ". Si noti che la precedenza degli operatori è implicita nel modo in cui sono definite le espressioni.

L'espressione:

10 + 5 * B

è costituita da due termini: 10 e 5 * B. Il secondo termine contiene due fattori: 5 e B. Questi fattori sono un numero e una variabile.

Al contrario l'espressione:

14 * (7 – C)

contiene due fattori: 14 e (7 - C). I fattori sono un numero e un'espressione racchiusa fra parentesi. L'espressione fra parentesi contiene due termini: un numero e una variabile.

Questo processo costituisce la base per un parser a discesa ricorsiva che è un insieme di funzioni ricorsive operanti in modo concatenato che implementano le regole di produzione. Ad ogni passo, il parser esegue le operazioni specificate in una sequenza corretta dal punto di vista algebrico. Per vedere il funzionamento di queste regole di produzione, si proverà a esaminare la seguente espressione:

9/3 – (100 + 56)

Ecco la sequenza che verrà eseguita.



1. Leggere il primo termine, 9 / 3.
2. Leggere ciascun fattore e dividere gli interi. Il valore risultante è 3.
3. Leggere il secondo termine, (100 + 56). A questo punto iniziare ad analizzare ricorsivamente la seconda subespressione.
4. Leggere i termini e sommarli. Il risultato è 156.
5. Uscire dalla chiamata ricorsiva e sottrarre 156 da 3. La risposta è -153.

A questo punto le cose potrebbero risultare un po' confuse. In effetti il concetto è abbastanza complesso e richiede un po' di adattamento. Occorre ricordare un paio di cose sull'analisi ricorsiva delle espressioni. Innanzitutto la precedenza degli operatori è implicita nel modo in cui sono definite le regole di produzione. In secondo luogo, questo metodo di analisi e valutazione delle espressioni è molto simile al modo in cui un essere umano valuta le espressioni matematiche.

La parte rimanente di questo capitolo sviluppa tre parser. Il primo analizza e valuta espressioni in virgola mobile di tipo double costituite unicamente da valori costanti. Successivamente questo parser verrà esteso per supportare l'uso di variabili. Infine la terza versione del parser è implementata sotto forma di classe template e può essere utilizzata per analizzare espressioni di qualsiasi tipo.

## 40.4 La classe parser

Il parser di espressioni si basa sulla classe parser. Ecco l'aspetto della prima versione di questa classe. Le versioni successive estenderanno questa prima versione.

```
class parser {
  char *exp_ptr;  // punta all'espressione
  char token[80]; // contiene il token corrente
  char tok_type;  // contiene il tipo del token

  void eval_exp2(double &result);
  void eval_exp3(double &result);
  void eval_exp4(double &result);
  void eval_exp5(double &result);
  void eval_exp6(double &result);
  void atom(double &result);
  void get_token();
  void serror(int error);
  int isdelim(char c);
public:
  parser();
  double eval_exp(char *exp);
};
```

La classe parser contiene tre variabili membro private. L'espressione da valutare è contenuta in una stringa chiusa dal carattere nullo puntata da exp_ptr. Pertanto il parser valuta espressioni contenute in una normale stringa C. Ad esempio, le seguenti stringhe contengono espressioni valutabili dal parser:

"10 – 5"
"2 * 3.3 / (3.1416 * 3.3)"

Quando il parser inizia l'esecuzione, exp_ptr deve puntare al primo carattere della stringa contenente l'espressione. Durante l'esecuzione il parser avanza nella stringa fino a trovare il codice nullo di chiusura della stringa.

Il significato delle altre due variabili membro, token e tok_type, verrà descritto nella prossima sezione.

Il punto di ingresso nel parser è rappresentato da eval_exp() che deve essere richiamata con un puntatore all'espressione da analizzare. Le funzioni da eval_exp2() a eval_exp6(), insieme ad atom() costituiscono il parser a discesa ricorsiva. Tali funzioni implementano una versione estesa delle regole di produzione di cui si è parlato in precedenza. Nelle prossime versioni del parser, verrà aggiunta anche una funzione eval_exp1().

La funzione serror() gestisce gli errori di sintassi dell'espressione. Le funzioni get_token() e isdelim() suddividono l'espressione nei suoi componenti come descritto nella prossima sezione.

## 40.5 Sezionamento di un'espressione

Per poter valutare le espressioni, occorre essere in grado di spezzare un'espressione nei suoi componenti. Poiché questa è un'operazione fondamentale per l'analisi, se ne parlerà prima di descrivere il parser vero e proprio.

Ogni componente di un'espressione è un token. Ad esempio, l'espressione:

A * B – (W + 10)

contiene i token A, *, B, -, (, W, +, 10 e ). Ogni token rappresenta un'unità indivisibile dell'espressione. In generale occorre una funzione che restituisca di volta in volta i singoli token di un'espressione. La funzione deve anche essere in grado di saltare gli spazi e i caratteri di tabulazione e rilevare la fine dell'espressione. La funzione che verrà utilizzata per eseguire questo compito si chiama get_token(), una funzione membro della classe parser.

Oltre a individuare il token, occorre determinare il tipo del token individuato. Per il parser sviluppato in questo capitolo, esistono solo tre tipi di token: VARIABLE, NUMBER e DELIMITER (DELIMITER è utilizzato sia per gli operatori che per le parentesi).



Di seguito si può vedere l'aspetto della funzione get_token(). Tale funzione ottiene dall'espressione puntata da exp_ptr il token successivo e lo inserisce nella variabile membro token. Poi inserisce il tipo del token nella variabile membro tok_type.

```
// Ottiene il token successivo.
void parser::get_token()
{
  register char *temp;

  tok_type = 0;
  temp = token;
  *temp = '\0';

  if(!*exp_ptr) return; // alla fine dell'espressione

  while(isspace(*exp_ptr)) ++exp_ptr; // salta gli spazi vuoti

  if(strchr("+-*/%^=()", *exp_ptr)){
    tok_type = DELIMITER;
    // avanza fino al carattere successivo
    *temp++ = *exp_ptr++;
  }
  else if(isalpha(*exp_ptr)) {
    while(!isdelim(*exp_ptr)) *temp++ = *exp_ptr++;
    tok_type = VARIABLE;
  }
  else if(isdigit(*exp_ptr)) {
    while(!isdelim(*exp_ptr)) *temp++ = *exp_ptr++;
    tok_type = NUMBER;
  }

  *temp = '\0';
}

// Restituisce true se c è un delimitatore.
int parser::isdelim(char c)
{
  if(strchr(" +-/*%^=()", c) || c==9 || c=='\r' || c==0)
    return 1;
  return 0;
}
```

Osservando attentamente le funzioni precedenti, si nota che dopo le prime inizializzazioni, get_token() controlla se è stato trovato il codice nullo che chiude l'espressione. A tale scopo controlla il carattere puntato da exp_ptr. Poiché exp_ptr punta all'espressione da analizzare, se punta a null significa che è stata raggiunta la fine dell'espressione. Se l'espressione contiene altri token, innanzitutto get_token() salta gli spazi iniziali. Dopo aver saltato tali spazi, get_token() punterà a un numero, una variabile, un operatore o, se l'espressione termina con una sequenza di spazi, al carattere null. Se il carattere successivo è un operatore, verrà restituito in token sotto forma di stringa ed in tok_type verrà inserito DELIMITER. Se invece il carattere successivo è una lettera, si presume che si tratti di una variabile. Anch'essa verrà restituita come stringa in token e a tok_type verrà assegnato il valore VARIABLE. Se il carattere successivo è una cifra, verrà letto l'intero numero che verrà inserito nella stringa token con il tipo NUMBER. Infine, se il carattere successivo non è nessuno di quelli elencati, si presume che sia stata raggiunta la fine dell'espressione. In questo caso, token sarà null per segnalare il raggiungimento della fine dell'espressione.

Come si è detto in precedenza, per non complicare il codice di questa funzione, è stata omessa una buona parte degli elementi di verifica degli errori e sono state fatte alcune supposizioni. Ad esempio, l'espressione sarà chiusa da qualsiasi carattere non riconosciuto. Inoltre in questa versione le variabili possono avere un nome di qualsiasi lunghezza ma solo la prima lettera sarà significativa. Naturalmente sarà possibile aggiungere tutte le verifiche di errore e i dettagli richiesti dall'applicazione che si deve sviluppare.

Per comprendere meglio il processo di tokenizzazione, si studi ciò che viene restituito per ciascun token analizzando la seguente espressione:

A + 100 – (B * C) /2

| TOKEN | TIPO DEL TOKEN |
|---|---|
| A | VARIABLE |
| + | DELIMITER |
| 100 | NUMBER |
| – | DELIMITER |
| ( | DELIMITER |
| B | VARIABLE |
| * | DELIMITER |

*(segue)*



*(continua)*

| TOKEN | TIPO DEL TOKEN |
|---|---|
| C | VARIABLE |
| ) | DELIMITER |
| / | DELIMITER |
| 2 | NUMBER |
| null | null |

Si ricordi che token contiene sempre una stringa chiusa dal carattere nullo, anche se essa è costituita da un solo carattere.

## 40.6 Un semplice parser di espressioni

Ecco finalmente la prima versione del parser. Il parser può valutare espressioni costituite unicamente da costanti, operatori e parentesi. Pertanto non accetterà espressioni contenenti variabili.

```
/* Questo modulo contiene il parser a discesa ricorsiva
   che non contiene variabili.
*/

#include <iostream>
#include <cstdlib>
#include <cctype>
#include <cstring>
using namespace std;

enum types { DELIMITER = 1, VARIABLE, NUMBER};

class parser {
  char *exp_ptr;  // punta all'espressione
  char token[80]; // contiene il token corrente
  char tok_type;  // contiene il tipo del token

  void eval_exp2(double &result);
  void eval_exp3(double &result);
  void eval_exp4(double &result);
  void eval_exp5(double &result);
  void eval_exp6(double &result);
```

```
  void atom(double &result);
  void get_token();
  void serror(int error);
  int isdelim(char c);
public:
  parser();
  double eval_exp(char *exp);
};

// costruttore di parser
parser::parser()
{
  exp_ptr = NULL;
}

// Punto di ingresso nel parser.
double parser::eval_exp(char *exp)
{
  double result;

  exp_ptr = exp;

  get_token();
  if(!*token) {
    serror(2); // non contiene alcuna espressione
    return 0.0;
  }
  eval_exp2(result);
  if(*token) serror(0); // l'ultimo token deve essere null
  return result;
}

// Somma o sottrae due termini.
void parser::eval_exp2(double &result)
{
  register char op;
  double temp;

  eval_exp3(result);
  while((op = *token) == '+' || op == '-') {
    get_token();
    eval_exp3(temp);
    switch(op) {
      case '-':
        result = result - temp;
        break;
```



```
      case '+':
        result = result + temp;
        break;
    }
  }
}

// Moltiplica o divide due fattori.
void parser::eval_exp3(double &result)
{
  register char op;
  double temp;

  eval_exp4(result);
  while((op = *token) == '*' || op == '/' || op == '%') {
    get_token();
    eval_exp4(temp);
    switch(op) {
      case '*':
        result = result * temp;
        break;
      case '/':
        result = result / temp;
        break;
      case '%':
        result = (int) result % (int) temp;
        break;
    }
  }
}

// Elabora un elevamento a potenza
void parser::eval_exp4(double &result)
{
  double temp, ex;
  register int t;

  eval_exp5(result);
  if(*token== '^') {
    get_token();
    eval_exp4(temp);
    ex = result;
    if(temp==0.0) {
      result = 1.0;
      return;
    }
```

```
    for(t=(int)temp-1; t>0; --t) result = result * (double)ex;
  }
}

// Valuta un + o un - unario.
void parser::eval_exp5(double &result)
{
  register char  op;

  op = 0;
  if((tok_type == DELIMITER) && *token=='+' || *token == '-') {
    op = *token;
    get_token();
  }
  eval_exp6(result);
  if(op=='-') result = -result;
}

// Elabora un'espressione fra parentesi.
void parser::eval_exp6(double &result)
{
  if((*token == '(')) {
    get_token();
    eval_exp2(result);
    if(*token != ')')
      serror(1);
    get_token();
  }
  else atom(result);
}

// Legge il valore di un numero.
void parser::atom(double &result)
{
  switch(tok_type) {
    case NUMBER:
      result = atof(token);
      get_token();
      return;
    default:
      serror(0);
  }
}

// Visualizza un errore di sintassi.
void parser::serror(int error)
```



```
{
  static char *e[]= {
      "Errore di sintassi",
      "Parentesi non bilanciate",
      "Non contiene alcuna espressione"
  };
  cout << e[error] << endl;
}

// Ottiene il token successivo.
void parser::get_token()
{
  register char *temp;

  tok_type = 0;
  temp = token;
  *temp = '\0';

  if(!*exp_ptr) return; // alla fine dell'espressione

  while(isspace(*exp_ptr)) ++exp_ptr; // salta gli spazi vuoti

  if(strchr("+-*/%^=()", *exp_ptr)) {
    tok_type = DELIMITER;
    // avanza fino al carattere successivo
    *temp++ = *exp_ptr++;
  }
  else if(isalpha(*exp_ptr)) {
    while(!isdelim(*exp_ptr)) *temp++ = *exp_ptr++;
    tok_type = VARIABLE;
  }
  else if(isdigit(*exp_ptr)) {
    while(!isdelim(*exp_ptr)) *temp++ = *exp_ptr++;
    tok_type = NUMBER;
  }

  *temp = '\0';
}

// Restituisce true se c è un delimitatore.
int parser::isdelim(char c)
{
  if(strchr(" +-/*%^=()", c) || c==9 || c=='\r' || c==0)
    return 1;
  return 0;
}
```

La prima versione del parser può gestire i seguenti operatori: +, -, *, / e %. Inoltre può gestire l'elevamento a potenza intera e il meno unario. Infine il parser gestisce correttamente le parentesi. La valutazione delle espressioni si svolge nelle funzioni ricorsive e mutuamente esclusive che vanno da eval_exp2() a eval_exp6() e nella funzione atom() che restituisce il valore di un numero. I commenti posti all'inizio di ciascuna funzione descrivono il ruolo che tali funzioni giocano nell'analisi dell'espressione.

La seguente funzione main() illustra l'uso di questo parser.

```
int main()
{
  char expstr[80];

  cout << "Per uscire digitare il punto.\n";

  parser ob; // istanzia un parser

  for(;;) {
    cout << "Introdurre un'espressione: ";
    cin.getline(expstr, 79);
    if(*expstr=='.') break;
    cout << "Il risultato è: " << ob.eval_exp(expstr) << "\n\n";
  };

  return 0;
}
```

Ecco un esempio d'uso.

```
Per uscire digitare il punto.
Introdurre un'espressione: 10-2*3
Il risultato è: 4

Introdurre un'espressione: (10-2)*3
Il risultato è: 24

Introdurre un'espressione: 10/3
Il risultato è: 3.33333

Introdurre un'espressione: .
```



## Il funzionamento del parser

Per comprendere esattamente il modo in cui il parser valuta un'espressione, si provi a seguire cosa accade nell'analisi della seguente espressione (si supponga che exp_ptr punti all'inizio dell'espressione).

```
10 - 3 * 2
```

Quando viene richiamata eval_exp(), il punto di ingresso nel parser, questa ottiene il primo token. Se il primo token è nullo, la funzione visualizza un messaggio e quindi termina. Ma in questo caso il token contiene il numero 10. Poiché il primo token non è nullo, viene richiamata eval_exp2(). Come risultato, eval_exp2() richiama eval_exp3() ed eval_exp3() richiama eval_exp4() che a sua volta richiama eval_exp5(). La funzione eval_exp5() controlla se il token è un operatore + o - unario; in questo caso non è così e dunque viene richiamata eval_exp6(). A questo punto, eval_exp6() può richiamare ricorsivamente eval_exp2() (nel caso di un'espressione fra parentesi) oppure atom() per conoscere il valore del numero. Poiché il token non è una parentesi aperta, viene eseguito atom() e a result viene assegnato il valore 10. Quindi viene letto un altro token e la funzione inizia a uscire dalla catena di chiamate a funzione. Poiché il token ora è un operatore, la funzione esce fino a eval_exp2().

Ciò che accade ora è molto importante. Poiché il token è -, questo viene salvato in op. Quindi il parser legge il token successivo, il numero 3, e ricomincia la discesa nella catena delle funzioni. Come prima si entra in atom(). In valore 3 viene restituito in result e viene letto il token *. Questo provoca una risalita nella catena delle funzioni fino a eval_exp3() dove viene letto il token finale 2. A questo punto inizia la prima operazione aritmetica, la moltiplicazione fra 2 e 3. Il risultato viene restituito a eval_exp2() che esegue la sottrazione. La sottrazione fornisce la risposta 4. Anche se a prima vista questa procedura può sembrare complessa, si possono trovare altri esempi per verificare che questo metodo funziona correttamente per ogni espressione.

Questo parser può essere adatto per realizzare una semplice calcolatrice. Ma prima di poter essere utilizzata in un linguaggio per computer, in un database o in una calcolatrice più sofisticata, occorre dargli la possibilità di gestire le variabili. Questo è l'argomento della prossima sezione.

## 40.7 Aggiunta delle variabili

Le variabili sono utilizzate da tutti i linguaggi di programmazione, da molte calcolatrici e dai fogli elettronici per contenere valori che possono essere utili in

seguito. Se il parser deve essere utilizzato per tali applicazioni, occorre espanderlo per includere le variabili. Questa aggiunta richiede varie modifiche al parser. Innanzitutto occorre aggiungere le variabili. Come si è detto in precedenza, per le variabili verranno utilizzate le lettere a A a Z. Le variabili sono contenute in un array all'interno della classe parser. Ogni variabile utilizza una posizione di un array di 26 elementi double. Pertanto alla classe parser si dovrà aggiungere tale array:

```
double vars[NUMVARS]; // contiene il valore delle variabili
```

Inoltre si dovrà modificare il costruttore della classe parser.

```
// costruttore di parser
parser::parser()
{
  int i;

  exp_ptr = NULL;

  for(i=0; i<NUMVARS; i++) vars[i]  0.0;
}
```

Come si può vedere per comodità le variabili vengono tutte inizializzate a 0.

Ora è necessario pensare a una funzione in grado di accedere al valore di ogni singola variabile. Poiché le variabili hanno un nome che va da A sua Z, il loro nome può essere utilizzato con facilità per gestire l'indice dell'array sottraendo al codice ASCII della lettera della variabile il codice ASCII della lettera A. L'operazione viene eseguita dalla funzione membro find_var():

```
// Restituisce il valore di una variabile.
double parser::find_var(char *s)
{
  if(!isalpha(*s)){
    serror(1);
    return 0.0;
  }
  return vars[toupper(*token)-'A'];
}
```

Questa funzione accetta variabili dal nome molto lungo ma solo la prima letterà sarà significativa. Naturalmente è possibile modificare questa funzione per adattarla alle proprie esigenze.



Inoltre si deve modificare la funzione atom() in modo da gestire sia numeri che variabili. Ecco la nuova versione:

```
// Legge il valore di un numero o di una variabile.
void parser::atom(double &result)
{
  switch(tok_type) {
    case VARIABLE:
      result = find_var(token);
      get_token();
      return;
    case NUMBER:
      result = atof(token);
      get_token();
      return;
    default:
      serror(0);
  }
}
```

Tecnicamente queste aggiunte sono tutto ciò che è necessario per consentire al parser di utilizzare le variabili. Tuttavia non è stato ancora previsto alcuno strumento per assegnare un valore a queste variabili. Spesso l'operazione viene eseguita all'esterno del parser ma si può trattare il segno di uguaglianza come un operatore di assegnamento (così come avviene in C++) in modo che tale operatore entri a far parte del parser. Vi sono vari modi per eseguire questa operazione. Un metodo consiste nell'aggiungere alla classe parser una nuova funzione, chiamata eval_exp1(). Questa funzione inizia la catena delle discese ricorsive. Questo significa che eval_exp() deve richiamare eval_exp1() e non eval_exp2() per iniziare l'analisi dell'espressione. Ecco l'aspetto della funzione eval_exp1():

```
// Elabora un assegnamento.
void parser::eval_exp1(double &result)
{
  int slot;
  char ttok_type;
  char temp_token[80];

  if(tok_type==VARIABLE) {
    // salva il vecchio token
    strcpy(temp_token, token);
    ttok_type = tok_type;
```

```
    // calcola l'indice della variabile
    slot = toupper(*token) - 'A';

    get_token();
    if(*token != '=') {
      putback(); // restituisce il token corrente
      // ripristina il vecchio token - non è un assegnamento
      strcpy(token, temp_token);
      tok_type = ttok_type;
    }
    else
      get_token(); // legge la parte successiva dell'espressione
      eval_exp2(result);
      vars[slot] = result;
      return;
    }
  }

  eval_exp2(result);
}
```

Come si può vedere, la funzione deve avanzare per vedere se si sta eseguendo un assegnamento. Questo è dovuto al fatto che il nome di una variabile precede sempre un assegnamento ma il solo nome della variabile non garantisce che l'espressione successiva sia un assegnamento. Questo significa che il parser accetterà l'espressione A = 100, ovvero un assegnamento ma sarà in grado di scoprire che A / 10 non è un assegnamento. Per fare questo, eval_exp1() legge il token successivo dallo stream di input. Se non si tratta del segno di uguaglianza, il token viene restituito allo stream con putback() per essere riutilizzato in seguito. Pertanto occorre includere nella classe parser anche la funzione putback():

```
// Restituisce un token allo stream di input.
void parser::putback()
{
  char *t;

  t = token;
  for(; *t; t++) exp_ptr--;
}
```

Dopo aver apportato tutte le modifiche necessarie, il parser avrà il seguente aspetto.



```
/* Questo modulo contiene il parser a discesa ricorsiva
   che riconosce le variabili.
*/

#include <iostream>
#include <cstdlib>
#include <cctype>
#include <cstring>
using namespace std;

enum types { DELIMITER = 1, VARIABLE, NUMBER};

const int NUMVARS = 26;

class parser {
  char *exp_ptr;  // punta all'espressione
  char token[80]; // contiene il token corrente
  char tok_type;  // contiene il tipo del token
  double vars[NUMVARS]; // contiene il valore delle variabili

  void eval_exp1(double &result);
  void eval_exp2(double &result);
  void eval_exp3(double &result);
  void eval_exp4(double &result);
  void eval_exp5(double &result);
  void eval_exp6(double &result);
  void atom(double &result);
  void get_token();
  void putback();
  void serror(int error);
  double find_var(char *s);
  int isdelim(char c);
public:
  parser();
  double eval_exp(char *exp);
};

// costruttore di parser
parser::parser()
{
  int i;

  exp_ptr = NULL;

  for(i=0; i<NUMVARS; i++) vars[i]  0.0;
}
```

```
// Punto di ingresso nel parser.
double parser::eval_exp(char *exp)
{
  double result;

  exp_ptr = exp;

  get_token();
  if(!*token) {
    serror(2); // non contiene alcuna espressione
    return 0.0;
  }
  eval_exp1(result);
  if(*token) serror(0); // l'ultimo token deve essere null
  return result;
}

// Elabora un assegnamento.
void parser::eval_exp1(double &result)
{
  int slot;
  char ttok_type;
  char temp_token[80];

  if(tok_type==VARIABLE) {
    // salva il vecchio token
    strcpy(temp_token, token);
    ttok_type = tok_type;

    // calcola l'indice della variabile
    slot = toupper(*token) - 'A';

    get_token();
    if(*token != '=') {
      putback(); // restituisce il token corrente
      // ripristina il vecchio token - non è un assegnamento
      strcpy(token, temp_token);
      tok_type = ttok_type;
    }
    else {
      get_token(); // legge la parte successiva dell'espressione
      eval_exp2(result);
      vars[slot] = result;
      return;
    }
  }
```



```
  eval_exp2(result);
}

// Somma o sottrae due termini.
void parser::eval_exp2(double &result)
{
  register char op;
  double temp;

  eval_exp3(result);
  while((op = *token) == '+' || op == '-') {
    get_token();
    eval_exp3(temp);
    switch(op) {
      case '-':
        result = result - temp;
        break;
      case '+':
        result = result + temp;
        break;
    }
  }
}

// Moltiplica o divide due fattori.
void parser::eval_exp3(double &result)
{
  register char op;
  double temp;

  eval_exp4(result);
  while((op = *token) == '*' || op == '/' || op == '%') {
    get_token();
    eval_exp4(temp);
    switch(op) {
      case '*':
        result = result * temp;
        break;
      case '/':
        result = result / temp;
        break;
      case '%':
        result = (int) result % (int) temp;
        break;
    }
  }
```

```
}

// Elabora un elevamento a potenza
void parser::eval_exp4(double &result)
{
  double temp, ex;
  register int t;

  eval_exp5(result);
  if(*token== '^') {
    get_token();
    eval_exp4(temp);
    ex = result;
    if(temp==0.0) {
      result = 1.0;
      return;
    }
    for(t=(int)temp-1; t>0; --t) result = result * (double)ex;
  }
}

// Valuta un + o un - unario.
void parser::eval_exp5(double &result)
{
  register char  op;

  op = 0;
  if((tok_type == DELIMITER) && *token=='+' || *token == '-') {
    op = *token;
    get_token();

  eval_exp6(result);
  if(op=='-') result = -result;
}

// Elabora un'espressione fra parentesi.
void parser::eval_exp6(double &result)
{
  if((*token == '(')) {
    get_token();
    eval_exp2(result);
    if(*token != ')')
      serror(1);
    get_token();
  }
  else atom(result);
```



```
}

// Legge il valore di un numero o di una variabile.
void parser::atom(double &result)
{
  switch(tok_type) {
    case VARIABLE:
      result = find_var(token);
      get_token();
      return;
    case NUMBER:
      result = atof(token);
      get_token();
      return;
    default:
      serror(0);
  }
}

// Restituisce un token allo stream di input.
void parser::putback()
{
  char *t;

  t = token;
  for(; *t; t++) exp_ptr--;
}

// Visualizza un errore di sintassi.
void parser::serror(int error)
{
  static char *e[]= {
      "Errore di sintassi",
      "Parentesi non bilanciate",
      "Non contiene alcuna espressione"
  };
  cout << e[error] << endl;
}

// Ottiene il token successivo.
void parser::get_token()
{
  register char *temp;

  tok_type = 0;
  temp = token;
```

```
  *temp = '\0';

  if(!*exp_ptr) return; // alla fine dell'espressione

  while(isspace(*exp_ptr)) ++exp_ptr; // salta gli spazi vuoti

  if(strchr("+-*/%^=()", *exp_ptr)) {
    tok_type = DELIMITER;
    // avanza fino al carattere successivo
    *temp++ = *exp_ptr++;
  }
  else if(isalpha(*exp_ptr)) {
    while(!isdelim(*exp_ptr)) *temp++ = *exp_ptr++;
    tok_type = VARIABLE;
  }
  else if(isdigit(*exp_ptr)) {
    while(!isdelim(*exp_ptr)) *temp++ = *exp_ptr++;
    tok_type = NUMBER;
  }

  *temp = '\0';
}

// Restituisce true se c è un delimitatore.
int parser::isdelim(char c)
{
  if(strchr(" +-/*%^=()", c) || c==9 || c=='\r' || c==0)
    return 1;
  return 0;
}

// Restituisce il valore di una variabile.
double parser::find_var(char *s)
{
  if(!isalpha(*s)) {
    serror(1);
    return 0.0;
  }
  return vars[toupper(*token)-'A'];
}
```

Per provare questa nuova versione del parser, si può impiegare la stessa funzione main() utilizzata per il primo parser. La seconda versione del parser accetta anche le seguenti espressioni:



```
A = 10/4
A - B
C = A * (F - 21)
```

## 40.8 Controllo della sintassi in un parser a discesa ricorsiva

Prima di parlare della versione template del parser, è opportuno esaminare brevemente la verifica della sintassi. Nell'analisi delle espressioni, un errore di sintassi è semplicemente una situazione in cui l'espressione non segue le rigide regole richieste dal parser. La maggior parte delle volte si tratterà di un errore umano, in genere di un errore di digitazione. Ad esempio le seguenti espressioni non verranno accettate dai parser creati in questo capitolo:

```
10 ** 8
(10 - 5) * 9)
/8
```

La prima espressione contiene due operatori in sequenza, la seconda contiene un numero errato di parentesi e l'ultima contiene il segno di divisione all'inizio dell'espressione. Nessuna di queste condizioni può essere consentita dai parser. Poiché gli errori di sintassi possono fare in modo che il parser fornisca risultanti errati, è opportuno trovare il modo per risolvere queste situazioni.

Dell'analisi del codice dei parser, si sarà probabilmente notata la presenza della funzione serror() che viene richiamata in alcune situazioni. A differenza di molti altri tipi di parser, i parser a discesa ricorsiva semplificano la verifica della sintassi poiché essa si verifica principalmente in atom(), find_var() ed eval_exp6(), dove vengono controllate le parentesi. L'unico problema del controllo della sintassi è che il parser non viene chiuso in caso di errore. Questo porta a una successione di messaggi di errore. Il modo migliore per implementare la funzione serror() consiste nell'eseguire una qualche forma di reset. Ad esempio, i compilatori C++ forniscono una coppia di funzioni di salto chiamate setjmp() e longjmp(). Queste due funzioni consentono a un programma di saltare in un'altra funzione. Pertanto serror() potrebbe eseguire un salto con longjmp() in un punto sicuro del programma, all'esterno del parser.

A seconda dell'uso che si fa del parser, si potrebbe anche decidere che il meccanismo di gestione delle eccezioni fornito dal linguaggio C++ (implementato tramite try, catch e throw) può essere comodo per la gestione degli errori.

Se si lascia il codice così come è, il parser può produrre più messaggi di errore. Questo può essere un problema in alcune situazioni e vantaggioso in altre,

dove è importante rilevare tutti gli errori. Ma generalmente sarà opportuno estendere la sezione di controllo della sintassi prima di utilizzare un parser di questo tipo in un programma commerciale.

## 40.9 Realizzazione di un parser generico

I due parser precedenti operavano su espressioni numeriche in cui tutti i valori dovevano essere di tipo double. Anche se questo può andare bene per applicazioni che utilizzano valori double, è certamente eccessivo per le applicazioni che utilizzano solo valori interi. Il fatto di definire nel codice il tipo dei valori utilizzati riduce senza motivo le possibilità di applicazione del parser. Fortunatamente, grazie all'impiego di un template per classi, non è difficile creare una versione generica del parser che funzioni con ogni tipo di dati per i quali siano definite le espressioni algebriche. In questo modo il parser potrà essere utilizzato sia con i tipi standard che con i tipi numerici creati dal programmatore. Ecco la versione generica del parser di espressioni.

```
// Un parser generico.

#include <iostream>
#include <cstdlib>
#include <cctype>
#include <cstring>
using namespace std;

enum types { DELIMITER = 1, VARIABLE, NUMBER};

const int NUMVARS = 26;

template <class PType> class parser {
  char *exp_ptr;  // punta all'espressione
  char token[80]; // contiene il token corrente
  char tok_type;  // contiene il tipo del token
  PType vars[NUMVARS]; // contiene il valore delle variabili

  void eval_exp1(PType &result);
  void eval_exp2(PType &result);
  void eval_exp3(PType &result);
  void eval_exp4(PType &result);
  void eval_exp5(PType &result);
  void eval_exp6(PType &result);
  void atom(PType &result);
```



```
  void get_token(), putback();
  void serror(int error);
  PType find_var(char *s);
  int isdelim(char c);
public:
  parser();
  PType eval_exp(char *exp);
};

// costruttore di parser
template <class PType> parser<PType>::parser()
{
  int i;

  exp_ptr = NULL;

  for(i=0; i<NUMVARS; i++) vars[i]  (PType) 0;
}

// Punto di ingresso nel parser.
template <class PType> PType parser<PType>::eval_exp(char *exp)
{
  PType result;

  exp_ptr = exp;

  get_token();
  if(!*token) {
    serror(2); // non contiene alcuna espressione
    return (PType) 0;
  }
  eval_exp1(result);
  if(*token) serror(0); // l'ultimo token deve essere null
  return result;
}

// Elabora un assegnamento.
template <class PType> void parser<PType>::eval_exp1(PType &result)
{
  int slot;
  char ttok_type;
  char temp_token[80];

  if(tok_type==VARIABLE) {
    // salva il vecchio token
    strcpy(temp_token, token);
```

```
    ttok_type = tok_type;

    // calcola l'indice della variabile
    slot = toupper(*token) - 'A';

    get_token();
    if(*token != '=') {
      putback(); // restituisce il token corrente
      // ripristina il vecchio token - non è un assegnamento
      strcpy(token, temp_token);
      tok_type = ttok_type;
    }
    else {
      get_token(); // legge la parte successiva dell'espressione
      eval_exp2(result);
      vars[slot] = result;
      return;
    }
  }

  eval_exp2(result);
}

// Somma o sottrae due termini.
template <class PType> void parser<PType>::eval_exp2(PType &result)
{
  register char op;
  PType temp;

  eval_exp3(result);
  while((op = *token) == '+' || op == '-') {
    get_token();
    eval_exp3(temp);
    switch(op) {
      case '-':
        result = result - temp;
        break;
      case '+':
        result = result + temp;
        break;
    }
  }
}

// Moltiplica o divide due fattori.
template <class PType> void parser<PType>::eval_exp3(PType &result)
```



```
{
  register char op;
  PType temp;

  eval_exp4(result);
  while((op = *token) == '*' || op == '/' || op == '%') {
    get_token();
    eval_exp4(temp);
    switch(op) {
      case '*':
        result = result * temp;
        break;
      case '/':
        result = result / temp;
        break;
      case '%':
        result = (int) result % (int) temp;
        break;
    }
  }
}

// Elabora un elevamento a potenza
template <class PType> void parser<PType>::eval_exp4(PType &result)
{
  PType temp, ex;
  register int t;

  eval_exp5(result);
  if(*token== '^') {
    get_token();
    eval_exp4(temp);
    ex = result;
    if(temp==0.0) {
      result = (PType) 1;
      return;
    }
    for(t=(int)temp-1; t>0; --t) result = result * ex;
  }
}

// Valuta un + o un - unario.
template <class PType> void parser<PType>::eval_exp5(PType &result)
{
  register char  op;
```

```
  op = 0;
  if((tok_type == DELIMITER) && *token=='+' || *token == '-') {
    op = *token;
    get_token();
  }
  eval_exp6(result);
  if(op=='-') result = -result;
}

// Elabora un'espressione fra parentesi.
template <class PType> void parser<PType>::eval_exp6(PType &result)
{
  if((*token == '(')) {
    get_token();
    eval_exp2(result);
    if(*token != ')')
      serror(1);
    get_token();
  }
  else atom(result);
}

// Legge il valore di un numero o di una variabile.
template <class PType> void parser<PType>::atom(PType &result)
{
  switch(tok_type) {
    case VARIABLE:
      result = find_var(token);
      get_token();
      return;
    case NUMBER:
      result = (PType) atof(token);
      get_token();
      return;
    default:
      serror(0);
  }
}

// Restituisce un token allo stream di input.
template <class PType> void parser<PType>::putback()
{
  char *t;

  t = token;
  for(; *t; t++) exp_ptr--;
```



```
}

// Visualizza un errore di sintassi.
template <class PType> void parser<PType>::serror(int error)
{
  static char *e[]= {
      "Errore di sintassi",
      "Parentesi non bilanciate",
      "Non contiene alcuna espressione"
  };
  cout << e[error] << endl;
}

// Ottiene il token successivo.
template <class PType> void parser<PType>::get_token()
{
  register char *temp;

  tok_type = 0;
  temp = token;
  *temp = '\0';

  if(!*exp_ptr) return; // alla fine dell'espressione

  while(isspace(*exp_ptr)) ++exp_ptr; // salta gli spazi vuoti

  if(strchr("+-*/%^=()", *exp_ptr)) {
    tok_type = DELIMITER;
    // avanza fino al carattere successivo
    *temp++ = *exp_ptr++;
  }
  else if(isalpha(*exp_ptr)) {
    while(!isdelim(*exp_ptr)) *temp++ = *exp_ptr++;
    tok_type = VARIABLE;
  }
  else if(isdigit(*exp_ptr)) {
    while(!isdelim(*exp_ptr)) *temp++ = *exp_ptr++;
    tok_type = NUMBER;
  }

  *temp = '\0';
}

// Restituisce true se c è un delimitatore.
template <class PType> int parser<PType>::isdelim(char c)
{
```

```
  if(strchr(" +-/*%^=()", c) || c==9 || c=='\r' || c==0)
    return 1;
  return 0;
}

// Restituisce il valore di una variabile.
template <class PType> PType parser<PType>::find_var(char *s)
{
  if(!isalpha(*s)){
    serror(1);
    return (PType) 0;
  }
  return vars[toupper(*token)-'A'];
}
```

Come si può vedere, il tipo di dati su cui opera il parser è specificato dal tipo generico PType. La seguente funzione illustra l'uso del parser generico:

```
int main()
{
  char expstr[80];

  // Illustra l'uso del parser per numeri in virgola mobile.
  parser<double> ob;

  cout << "Parser per numeri in virgola mobile.   ";
  cout << "Per uscire digitare il punto\n";
  for(;;) {
    cout << "Introdurre un'espressione: ";
    cin.getline(expstr, 79);
    if(*expstr=='.') break;
    cout << "Il risultato è: " << ob.eval_exp(expstr) << "\n\n";
  }
  cout << endl;

  // Illustra l'uso del parser per numeri interi.
  parser<int> Iob;

  cout << "Parser per numeri interi.   ";
  cout << "Per uscire digitare il punto\n";
  for(;;) {
    cout << "Introdurre un'espressione: ";
    cin.getline(expstr, 79);
    if(*expstr=='.') break;
    cout << "Il risultato è: " << ob.eval_exp(expstr) << "\n\n";
```



```
  }

  return 0;
}
```

Ecco un esempio d'uso.

```
Parser per numeri in virgola mobile.  Per uscire digitare il punto
Introdurre un'espressione: a=10.1
Il risultato è: 10.1

Introdurre un'espressione: b=3.2
Il risultato è: 3.2

Introdurre un'espressione: a/b
Il risultato è: 3.15625

Introdurre un'espressione: .

Parser per numeri interi.  Per uscire digitare il punto
Introdurre un'espressione: a=10
Il risultato è: 10

Introdurre un'espressione: b=3
Il risultato è: 3

Introdurre un'espressione: a/b
Il risultato è: 3

Introdurre un'espressione: .
```

Come si può vedere, il parser in virgola mobile usa valori in virgola mobile e il parser per interi usa valori interi.

## 40.10 Alcune estensioni da provare

Come si è detto in precedenza in questo capitolo, questi parser svolgono solo un livello minimo di verifica degli errori. Dunque si può pensare di aggiungere un controllo degli errori più dettagliato. Ad esempio si potrebbero evidenziare i punti dell'espressione in cui è stato rilevato l'errore. Questo consentirà di trovare e correggere l'errore di sintassi.

Così come è il parser può valutare solo espressioni numeriche. Tuttavia, con poche aggiunte, è possibile consentire al parser di valutare altri tipi di espressioni,

ad esempio stringhe, coordinate spaziali o numeri complessi. Ad esempio, per consentire al parser di valutare oggetti stringa si possono apportare le seguenti modifiche.

1. Definire un nuovo tipo di token chiamato STRING.
2. Estendere get_token() in modo da consentire il riconoscimento delle stringhe.
3. Aggiungere ad atom() un nuovo caso per la gestione di token di tipo STRING.

Dopo aver implementato questi passi, il parser potrà gestire espressioni come le seguenti:

```
a = "uno"
b = "due"
c = a + b
```

Il risultato in c sarà il concatenamento di a e b per produrre la frase "unodue".

Ecco un suggerimento applicativo per il parser. Creare un una piccola calcolatrice che accetti un'espressione introdotta dall'utente e ne visualizzi il risultato. Questa potrebbe essere un'interessante aggiunta a un qualsiasi programma commerciale. Se il programma viene realizzato per Windows, l'operazione sarà particolarmente facile.

# Indice analitico

## A


abort(), 753
abs(), 754
accumulate(), 893
acos(), 733
adattatori, 857
  da puntatore a funzione, 857
  da puntatore a funzione membro, 858
adjacent_difference(), 894
adjacent_find(), 824
advance(), 853
algoritmi, 632, 664
  accumulate(), 893
  adjacent_difference(), 894
  inner_product(), 895
  numerici, 893
  partial_sum(), 896
algoritmi STL, 823
allineamento
  dell'output, 209
allocatori, 860
allocazione
  degli array, 363
  della memoria, 362
  di oggetti, 364
allocazione dinamica, 136
ambiguità, 390
ANSI (American National Standards Institute), 3
applicazioni
  C++, 907
argc, 150
argomenti
  standard, 388, 390
argv, 150
array, 95, 337
  allocazione, 363
  bidimensionali, 102
  compattamento, 480
  creazione, 340
  dinamici
    uso, 626
  generare un puntatore, 97
  inizializzati, 340
  inizializzazione, 111
  di input, 623
  limiti, 97
  monodimensionale, passaggio alla funzione, 98
  monodimensionali, 95, 97
  multidimensionali, 108, 112
  non dimensionati, inizializzazione, 113
  non inizializzati, 340
  di oggetti, 337
  operazioni di I/O, 621
  di puntatori, 128
  relazione con i puntatori, 127
  di stringhe, 106
  nelle strutture, 181
asctime(), 742
asin(), 734
assegnamenti multipli, 40
assert(), 754
assign()
  prototipo, 688
atan(), 734
atan2(), 734
atexit(), 754
atof(), 755
atoi(), 755
atol(), 755
auto, 28


## B


bad(), 785
BASIC, 6, 8
binary_search(), 824
binder, 855
  uso dei, 681
binding
  anticipato, 467
  ritardato, 467
blocchi, 61, 93
bool, 16
break, 71
bsearch(), 756
Bubblesort, 478
bufferizzazione della riga, 198
bug, 6
byte, 555


## C


C
  abbreviazioni, 60
  aspetto di un programma, 9
  blocco di codice, 7
  compilazione separata, 12
  funzione, 7
  linguaggio per programmatori, 7
  linguaggio strutturato, 5
  origini del linguaggio, 3
  parole chiave, 9
  peculiarità del, 5
C ANSI
  elenco parole chiave, 10
  tipi di dati, 17

C++
  descrizione del, 1
  elementi di base, 268
  forma generale, 297
  introduzione alle classi, 279
  linguaggio, 261
  manipolatori, 532
  nuovi header, 277
  origini del, 263
  parole riservate, 297
  programma di esempio, 268
  stream, 520
  tipo di dati bool, 275
  trasformazione automatica in int, 274
  vecchio stile e moderno, 275
calloc(), 749
campi bit, 182
caratteri, 15, 555, 721
  di controllo, 721
  stampabili, 721
ceil(), 735
char, 15
chiamata
  a funzione e array, 148
  per indirizzo, 145
  per indirizzo, creazione, 146
  per valore, 145
cicli
  do-while, 84
  for, 74
  for senza corpo, 81
  for, varianti, 76
  infiniti, 80
  while, 81
classe
  per stringhe, 863
classe di memorizzazione
  specificatori, 27
classi, 299, 303, 305
  accesso alle, 445
  astratte, 464
  auto_ptr, 901
  back_insert_iterator, 848
  bad_exception, 515
  basate su array, 621
  base, 429
    controllo dell'accesso, 429
  base virtuali, 448
  basic_string, 863, 865, 866, 867, 868, 869, 870, 871, 872
  bitset, 801, 802, 803
  per caratteri, 779
  per caratteri estesi, 779
  char_traits, 872, 873
  complex, 875, 877, 878, 879
container, 635
container STL, 799
definizione di funzioni inline, 316
deque, 803, 804, 805
derivate, 429
eccezioni, 503
elenco oggetti, 300
ereditate, 429
exception, 515, 900
failure, 784
per i file, 549
friend, 312
front_insert_iterator, 848
generiche, 469, 482
  argomenti non-tipi, 488
  specializzazioni esplicite, 492
gerarchia di, 778
gslice, 890, 891
helper, 893
di I/O, 777
insert_iterator, 846
istream_iterator, 850
istreambuf_iterator, 851
iterator, 844
iterator_traits, 845
per iteratori a basso livello, 844
list, 805, 806, 807, 808
locali, 330
di localizzazione, 904
map, 808, 809, 810
membri static, 320
multimap, 810, 811, 812
multiset, 812, 813, 814
nidificazione, 330
ostream_iterator, 851
ostreambuf_iterator, 853
pair, 903
parser, 939
priority_queue, 814
queue, 815
reverse_iterator, 849
set, 816, 817, 818
slice, 890, 891
specificatori di accesso, 300
stack, 818, 819
per stream C++, 520
string, 683
StrType, 910, 922
  uso di, 931
template, 779
  applicazione di typeid, 585
  argomenti standard, 490
  creazione per classi generiche per array, 486
  tipi di dati generici, 485
  uso di dynamic_cast, 593
  valarray, 879, 880, 881, 882, 883, 884, 885, 886, 887, 888
  vector, 819, 820, 821
clear(), 785
  prototipo, 573
clearerr(), 698
clock(), 742
COBOL, 8
codici speciali, 37
commenti, 259
compilatori, 8
concatenamento, 916
const, 25
container, 632
  generici
    vector, 637
conteggio, 664
conversione di tipo
  automatica, 39
conversioni
  cast, 58
conversioni di tipo, 40
copy(), 824
copy_backward(), 824
cos(), 735
cosh(), 735
costanti, 35, 56
  carattere speciali, 37
  esadecimali, 36
  intere, 35
  ottali, 36
  stringa, 36
  in virgola mobile, 35
costrutti
  if-else-if, 65
costruttori, 293, 437, 912
  di copie, 377
  esecuzione, 327
  espliciti, 599, 617
  parametrizzati, 317
  passaggio di parametri, 441
  con un solo parametro, 319
count(), 825
count_if(), 825
ctime(), 742

## D

dati
  membri, 302
  membri static, 320
dati principali, 15
definizione circolare, 160
dichiarazione di variabili, 85
dichiarazioni
  forward, 310
  stile delle, 359
differenze
  fra C e C++, riepilogo, 628
difftime(), 743
direttive per il preprocessore, 245
distance(), 853
distruttori, 293, 437, 912
  esecuzione, 327
div(), 757
do-while, 61
double, 15, 35

## E

eccezioni, 899
  raccolta, 507
  restrizioni, 510
  rilancio, 512
editor, 8
EDOM, 733
elementi
  rimozione e sostituzione, 669
enumerazione, 169
enumerazioni, 188
eof(), 785
equal(), 825
equal_range(), 825
ERANGE, 733
ereditarietà, 267, 288, 429, 437
  di una classe base come protected, 435
  da più classi base, 436
  dei membri protected, 432
espressioni, 15, 56, 61, 92, 936
  analisi delle, 938
  condizionali, 70
  conversioni di tipo, 57
  elaborazione, 937
  sezionamento, 940
  spaziatura e parentesi, 59
estrattori
  creazione, 542
etichette, 61
exceptions(), 785
exit(), 757
exp(), 735
extern, 28

## F

fabs(), 736
fail(), 786



fclose(), 698
  prototipo, 225
  uso di, 227
feof(), 698
  prototipo, 229
  uso di, 228
ferror(), 699
  prototipo, 232
fflush(), 699
fgetc(), 699
fgetpos(), 700
fgets(), 230, 700
  prototipo, 230
file, 220
  accesso diretto, 566
  apertura, 223, 550
  cancellazione, 234
  chiusura, 225, 550
  indicatore di posizione, 221
  modalità di apertura, 550
  personalizzazione delle operazioni di I/O, 573
  posizione corrente, 570
  puntatore, 223
  rilevamento della fine, 563
  scrittura di un carattere, 226
  system
    funzionamento, 222
    funzioni di tipo, ANSI, 222
  di testo
    lettura e scrittura, 553
file header
  <cerrno>, 733
  <climits>, 16
  <cstdio>, 196
  <cstdlib>, 753
  <cstring>, 721
  <cwctype>, 768
  <errno.h>, 733
  <exception>, 899
  <fstream>, 549
  <functional>, 634, 857
  <iterator>, 843
  <stdexcept>, 901
  <stdlib.h>, 753
  <string.h>, 100
  <utility>, 634
  <wchar.h>, 767
  <wctype.h>, 767
file header.
  <typeinfo>, 577
fill(), 786, 826
fill_n(), 826
find(), 826
  prototipo, 691
find_end(), 826
find_first_of(), 827
find_if(), 827
flag
  di formattazione, 780
flag di formattazione
  attivazione, 529
  cancellazione, 525
  impostazione, 524
  valore dei, 528
flags(), 786
float, 15, 35
floor(), 736
flush(), 787
  prototipo, 566
fmod(), 736
fopen(), 701
  prototipo, 223
  uso di, 227
  valori consentiti, 701
for, 61
for_each(), 827
forme prefisse e postfisse
  degli operatori, 401
FORTRAN, 265
fprint()
  prototipo, 239
fprintf(), 702
fputc(), 703
fputs(), 230, 703
  prototipo, 230
fread()
  prototipo, 235
  uso di, 236
free(), 750
  prototipo, 138
freopen(), 704
  prototipo, 243
frexp(), 737
fscanf(), 704
  prototipo, 239
fseek(), 705
  macro, 705
  prototipo, 237
fsetpos(), 705
fstream(), 787
ftell(), 706
funzione
  uso degli oggetti, 674
funzioni, 143
  di allocazione dinamica della memoria, 749
  argomenti, 145
  argomenti standard, 383
  per array di caratteri estesi, 773
  di assegnamento, 914
  di classificazione per caratteri este, 768
  di conversione, 599
    creazione, 611

di conversione per stringhe di caratt, 773
costruttore, esecuzione, 438
costruttore, overloading, 373
distruttore, esecuzione, 438
efficienza, 167
elementi implementativi, 166
end(), 651
exit(), 89
fill(), 530
flush(), 566
forma generale, 143
formato rilocabile, 11
friend, 307, 407
generate, 471
generiche, 469
per il compattamento di un array, 480
ordinamento, 478
restrizioni, 477
uso delle, 478
get(), 556
getline(), 561
di I/O, 697
di I/O di utilizzo generale, 784
di I/O per caratteri estesi, 770
ignore(), 565
inline, 313
main(), 159
di manipolazione
creazione, 544
matematiche, 733
membro, 302, 599, 614
const, 614
definite da vector, 639
non-const, 614
utilizzate di list, 648
volatile, 617
membro static, 325
numero variabile
di parametri, 165
overloading, 284, 371
parametri formali, 145
passaggio di oggetti, 331
peek(), 566
per date, ore e localizzazione, 741
precision(), 530
prototipi, 162
push_back(), 652
push_front(), 652
put(), 556
putback(), 566
read(), 558
regole di visibilità, 144
restituzione dei valori, 155
restituzione di puntatori, 157
di servizio, 753
per stringhe, 721
per stringhe di caratteri estesi, 772
template, 471
parametri standard, 476
terminate(), 513
di tipo void, 159
uncaught_exception(), 515
unexpected(), 513
uscita dalle, 154
di utilizzo generale, 166
variabili locali, 144
varie, 921
virtuali, 453, 459
attributo virtual, 458
chiamata tramite indirizzo, 456
pure, 462
uso, 464
width(), 530
write(), 558
funzioni di allocazione dinamica del C, 136
funzioni di conversione fra caratteri estesi e mul
di conversione fra caratteri estesi e mul, 774
fwrite(), 706
prototipo, 235
uso di, 236


## G


gcount(), 788
geline()
prototipo, 561
generate(), 828
generate_n(), 828
gestione
dei dati, 210
gestione delle eccezioni, 497, 899
applicazioni, 516
per classi derivate, 506
opzioni, 507
principi di, 497
get(), 788
altre forme di, 561
prototipo, 556
getc(), 707
uso di, 227
getchar(), 707
alternative a, 198
problemi, 198
prototipo, 197
getenv(), 757
getline(), 789
gets(), 707
gmtime(), 743
good(), 789
gruppo di scansione, 215


## H


header
di I/O, 780
HUGE_VAL, 733


## I


I/O
binario
uso con stream basati su array, 628
da console
lettura e scrittura di stringhe, 199
operazioni, 202
formattato, operazioni, 522
funzionamento degli operatori, 271
manipolatori per la formattazione, 532
operazioni ad accesso diretto, 237
operazioni binarie, 555
operazioni C e C++, 219
operazioni C++ vecchie e nuove, 520
operazioni da console, 195, 242
operazioni non formattate, 555
operazioni su array, 621
personalizzazione delle operazioni, 573
sistemi di, 519
stato delle operazioni, 571
identificatori, 18
identificazione run-time dei tipi (RTTI), 577
una semplice applicazione, 583
if, 61
ifstream(), 787
ignore(), 790
prototipo, 565
incapsulamento, 266
includes(), 828





indirizzi, 337, 351
indipendenti, 357
come parametri, 351
restituzione di, 356
restrizioni, 358
di tipo derivato, 358
inner_product(), 895
inplace_merge(), 828
input
soppressione, 218
inseritori
creazione, 536
int, 15
interpreti, 8
intervalli di valori, 16
isalnum(), 721
isalpha(), 722
iscntrl(), 722
isdigit(), 722
isgraph(), 723
islower(), 723
isprint(), 723
ispunct(), 723
isspace(), 724
istringstream(), 795
istruzioni
di blocco, 93
break, 88
catch, 505
di ciclo, 61
composte, 61, 93
continue, 91
goto, 87
if, 62
if nidificati, 64
di iterazione, 61, 74
return, 87
di salto, 61, 86
di selezione, 61, 62
switch, 70
switch nidificate, 73
using, 603
isupper(), 724
isxdigit(), 724
iter_swap(), 829
iteratori, 633, 843
accesso ai vettori, 641
predefiniti, 845
tipi di base, 843


## L


labs(), 758
ldexp(), 737
ldiv(), 758
lettura
di un indirizzo, 215
di interi senza segno, 213
di numeri, 212
di singoli caratteri con scanf(), 213
di stringhe, 214
lexicographical_compare(), 829
libreria, 10
STL (Standard Template Library), 11
libreria di classi, 775
libreria standard
prototipi di funzione, 164
libreria STL
algoritmi, 632, 664, 665, 666, 667, 823
classi container, 635, 800
componenti standard, 634
container, 632, 799
elementi della, 634
file header, 634
funzionamento generale, 636
introduzione alla, 632
iteratori, 633, 843
oggetti funzione, 675
linguaggi
ad alto livello, 4
di medio livello, 4
strutturati e non strutturati, 6
linker, 10
linking, 11
liste, 647
memorizzazione di oggetti, 656
ordinamento, 653
unione, 654
localeconv(), 743
localtime(), 745
log(), 737
log10(), 737
long, 17
longjmp(), 758
lower_bound(), 829


## M


macro, 753
che operano come funzioni, 248
predefinite, 258
main(), 150
make_heap(), 830
malloc(), 750
prototipo, 137
manipolatori
di I/O, 780
mappe, 658
costruttori, 659
memorizzazione di oggetti, 662
max(), 830
max_element(), 830
mblen(), 759
mbstowcs(), 759
mbtowc(), 760
memchr(), 724
memcmp(), 725
memcpy(), 725
memmove(), 725
memoria
allocata, inizializzazione, 362
memset(), 726
merge(), 830
min(), 831
min_element(), 831
mismatch(), 831
mktime(), 745
modelli, 469
funzioni generiche, 469
overloading esplicito, 473
tipo di dati generico, 472
modf(), 738
modificatori
* e #, 210
di accesso, 25
di formato, 207, 217


## N


namespace, 278, 599
negatori, 857
next_permutation(), 832
nomi esterni, 18
nth_element(), 832
numeri
interi, 15
in virgola mobile, 15
in virgola mobile doppi, 15


## O


ofstream(), 787
oggetti
allocazione, 364
assegnamento, 335
creazione, 375
inizializzati, 375
non inizializzati, 375
passaggio di indirizzi, 355
restituzione di, 334

oggetti funzione, 675, 854
  binari, 676
  creazione, 679
  definizione, 854
  unari, 676
  uso di quelli forniti, 676
open(), 790
operatori, 38, 56
  ( ) e [ ], 55
  ?, 67
  di allocazione dinamica, 337, 360
  aritmetici, 38, 41
  bit-a-bit, 38, 46
  const_cast, 595
  di conversione cast, 587
  di decremento, 401
  di incremento, 401
  dynamic_cast, 587
  di incremento e decremento, 42
  logici, 38
  ordine di precedenza, 56
  overloading, 287
  overloading delle forme abbreviate, 402
  per i puntatori, 51
  punto, 173
  punto e freccia, 55
  reinterpret_cast, 597
  relazionali, 38, 920
  relazionali e logici, 43
  restrizioni all'overloading degli, 402
  di risoluzione del campo d'azione, 281, 329
  di scorrimento, 50
  sizeof, 53
  static_cast, 596
  string, 686
  ternario ?, 51
  virgola, 54
operatoro
  di assegnamento, 38
operazioni
  di I/O di stringhe, 913
  di I/O binarie e non formattate, 555
ordinamento
  generico, 478
ordine di valutazione, 57
ostringstream(), 795
ottimizzatori prestazionali, 8
Overloading
  ambiguità, 390
overloading, 143, 388
  di un costruttore, 373
  creazione di operator membro, 396
  dell'operatore (), 422
  dell'operatore ->, 424
  dell'operatore <, 784
  dell'operatore <<, 535
  dell'operatore >, 784
  dell'operatore >>, 535
  dell'operatore [], 418
  dell'operatore virgola, 425
  esplicito di una funzione, 473
  flessibilità, 373
  delle forme abbreviate degli operatori, 402
  funzione modificata tramite, 381
  delle funzioni, 284
  di funzioni, 371, 390
  delle funzioni costruttore, 373
  di new e delete, 409
  uso con array, 414
  degli operatori, 287, 395
  di operatori particolari, 418
  di operatori tramite funzioni friend, 403
  restrizioni degli operatori, 402
  di un template di funzioni, 475
  uso di friend per gli operatoi ++ e--, 405
  della versione nothrow di new e delete, 417

P

parametri
  dichiarazione di, 165
  di utilizzo generale, 166
parametri indirizzo, 143
parole riservate
  asm, 619
  export, 493
  namespace, 600
  overload, 383
  protected, 432
  typedef, 193
  typename, 493
  using, 603
parser
  aggiunta delle variabili, 949
  controllo della sintassi, 959
  di espressioni, 935, 943
  estensioni, 967
  funzionamento, 949
  generico, realizzazione, 960
partial_sort(), 833
partial_sort_copy(), 833
partial_sum(), 896
partition(), 833
pcount()
  prototipo, 623
peek(), 791
  prototipo, 566
perror(), 708
polimorfismo, 266, 453
pop_heap(), 834
posizionamento, 221
pow(), 738
precision(), 791
predicati, 634
  binari, 634
  unari, 634
preprocessore, 245
  direttive, 245
    #define, 246
    #elif, 251
    #else, 251
    #endif, 251
    #error, 249
    #if, 251
    #ifdef, 253
    #ifndef, 253
    #include, 250
    #line, 256
    #pragma, 256
    per compilazini condizionali, 250
    uso di defined, 255
  operatori # e ##, 257
prev_permutation(), 834
printf(), 708, 710
  prototipo, 203
programmazione a oggetti
  identificazione run-time dei tipi, 577
  operatori cast, 577
programmazione a oggetti (OOP), 264, 265
  ereditarietà, 267
  incapsulamento, 266
  polimorfismo, 266
programmi
  Countdown, 152
  DTOS: Legge il contenuto di un file, 228
  KTOD: Un programma di scrittura su file, 227
  numero magico - Versione 1, 63
  numero magico - Versione 2, 63
  numero magico - Versione 3, 65
  numero magico - Versione 4, 66
  un semplice dizionario, 201
  tris (tic-tac-toe), 114
programmi di servizio per la gestione di file, 8
programmi per la gestione di eventi in tempo reale, 8
puntatori, 95, 119, 337
  aritmetica, 123
  array di, 128
  assegnamento, 123
  C++ verifiche di tipo, 343
  confronti, 125
  definzione, 119
  espressioni, 122
  a funzioni, 133
  indicizzazione, 109
  indirizzamento multilivello, 129
  inizializzazione, 131
  ai membri di una classe, 348
  non inizializzati, 139
  a oggetti, 341
  operatori per i, 121
  problemi, 138
  a puntatori, 129
  relazione con gli array, 127
  a struttura dichiarazione, 178
  uso, 178
  a strutture, 177
  this, 343
  a tipi derivati, 345
  variabili, 120
push_heap(), 834
put(), 792
  prototipo, 556
putback(), 792
  prototipo, 566
putc(), 711
  prototipo, 226
  uso di, 227
putchar(), 711
  prototipo, 197
puts(), 712
  prototipo, 200

Q

qsort(), 760





R

raise(), 761
rand(), 761
random_shuffle(), 835
rdstate(), 792
read(), 793
  prototipo, 558
readsome(), 793
realloc(), 750
register, 28, 33
remove(), 712, 835
  prototipo, 234
remove_copy(), 835
remove_copy_if(), 835
remove_if(), 835
rename(), 712
replace(), 836
replace_copy(), 836
  prototipo, 669
replace_copy_if(), 836
replace_if(), 836
return, 154
reverse(), 836
  prototipo, 671
reverse_copy(), 836
rewind(), 231, 712
  prototipo, 231
rfind()
  prototipo, 691
ricorsione, 160
rotate(), 837
rotate_copy(), 837

S

scanf(), 713, 714, 715
  lettura di caratteri, 213
  parametri, 217
  prototipo, 211
  specificatori di formato, 713
search(), 837
search_n(), 837
seekg(), 793
  prototipo, 566, 570
seekp(), 793
  prototipo, 566, 570
sequenza istruzioni, 71
set di caratteri ASCII, 16
set_difference(), 838
set_intersection(), 838
set_simmetric_difference(), 839
set_union(), 839
setbuf(), 716
setf(), 794
  modificata tramite overloading, 526
setjmp(), 761
setlocale(), 745
setstate(), 795
setvbuf(), 716
short, 17
signal(), 762
signed, 16
sin(), 738
sinh(), 739
sistemi operativi, 8
sizeof
  uso di per la trasportabilità del codice, 191
sort(), 839
sort_heap(), 840
sottostringhe
  sottrazione di, 918
spazi
  caratteri diversi da, 217
  eliminazione, 216
spazi dei nomi, 599
  opzioni, 607
  senza nome, 605
  std, 609
spazi di nomi
  basi, 600
specificatori, 28
  %n, 206, 215
  di formato, 212
  di minima ampiezza di campo, 207
  di precisione, 208
specifiche di linking, 620
sprintf(), 717
sqrt(), 739
srand(), 762
sscanf(), 717
stable_partition(), 840
stable_sort(), 840
static, 28, 30
str(), 795
strcat(), 726
strchr(), 726
strcmp(), 727
strcoll(), 727
strcpy(), 728
strcspn(), 728
stream, 220, 520
  classi per C++, 520
  di input/output su array, 625
  di output su array, 621
  predefiniti per C++, 522
  standard, 240
  svuotamento, 235
  di testo, 220
strerror(), 728

strftime(), 746
stringhe, 99, 683
  di caratteri, 99
  chiuse dal carattere nullo, 95, 693
  confronto, 692
  container, 694
  di controllo, 217
  lettura e scrittura, 199
  manipolazione di base, 688
  operazioni sulle, 688
  ricerca di, 691
stringstream(), 795
strlen(), 728
  prototipo, 27
strncat(), 729
strncmp(), 729
strncpy(), 729
strpbrk(), 730
strrchr(), 730
strspn(), 730
strstr(), 731
strtod(), 763
strtok(), 731
strtol(), 763
strtoul(), 764
strutture, 169, 170, 303
  accesso ai membri, 173
  all'interno di strutture, 181
  array, 174
  assegnamenti, 173
  dichiarazione, 170
  membri, 170
  passaggio alle funzioni, 175, 176
  puntatori a, 177
strxfrm(), 731
swap(), 841
swap_ranges(), 841
switch, 61
sync_with_stdio(), 797
system(), 764

T

tan(), 739
tanh(), 740
tellg(), 797
tellp(), 797
template, 469
  funzione con due tipi generici, 472
  di funzioni overloading, 475
  funzioni generiche, 469
  overloading esplicito, 473
  potenza dei, 494
terminate()
  prototipo, 513
time(), 747
tipi
  creazione e integrazione, 933
  iostate, 783
  off_type, 783
  openmode, 783
  pos_type, 783
  seekdir, 783
  del sistema di I/O, 782
  streamoff, 782
  streampos, 782
  streamsize, 782
  wstreampos, 782
tmpfile(), 718
tmpnam(), 718
token, 942
tolower(), 732
toupper(), 732
towctrans(), 769
transform(), 841
  prototipo, 672
typeid, 578, 583
  applicazioni a classi template, 585
  sostituzione con dynamic_cast, 591

U

uncaught_exception()
  prototipo, 515
unexpected()
  prototipo, 513
ungetc(), 718
unioni, 169, 185, 305
  anonime, 306
unique(), 842
unique_copy(), 842
unsetf(), 797
unsigned, 16
upper_bound(), 842
utilizzo di un vecchio compilatore, 279

V

va_arg(), 765
va_end(), 765
va_start(), 765
variabili, 19, 56
  automatiche, 19
  dichiarazione, 19
  globali, 23
  globali, static, 31
  inizializzazione, 34
  locali, 19, 144
    dichiarazione, 272
  locali, memorizzazione, 22
  locali, static, 30
  membro, 302
  parametri formali, 23
  puntatore, 51
vettori, 637
  accesso tramite iteratori, 641
  elementi
    inserimento e cancellazione, 643
  memorizzazione di oggetti, 645
vfprintf(), 719
visualizzazione
  dei caratteri, 203
  indirizzo di memoria, 205
  dei numeri, 203
void, 15, 16
volatile, 27
vprintf(), 719
vsprintf(), 719

W

wchar_t, 16
wcstombs(), 765
wctomb(), 766
wctrans(), 769
while, 61
width(), 798
write(), 798
  prototipo, 558