USERS

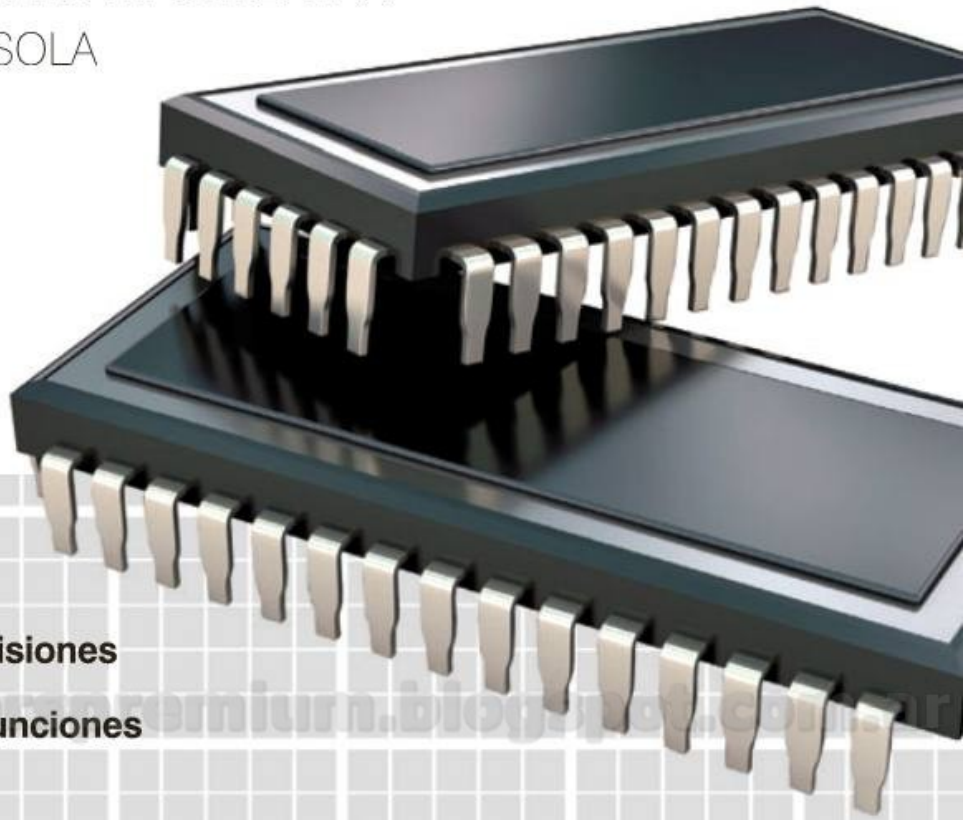
TÉCNICO en
ELECTRÓNICA

CONCEPTOS FUNDAMENTALES Y PRÁCTICA PROFESIONAL

1

PROGRAMACIÓN EN C

CONCEPTOS BÁSICOS DE PROGRAMACIÓN EN C Y HERRAMIENTAS PARA IMPLEMENTAR APLICACIONES DE CONSOLA



Entorno de programación

Comparación y toma de decisiones

Declaración de variables y funciones

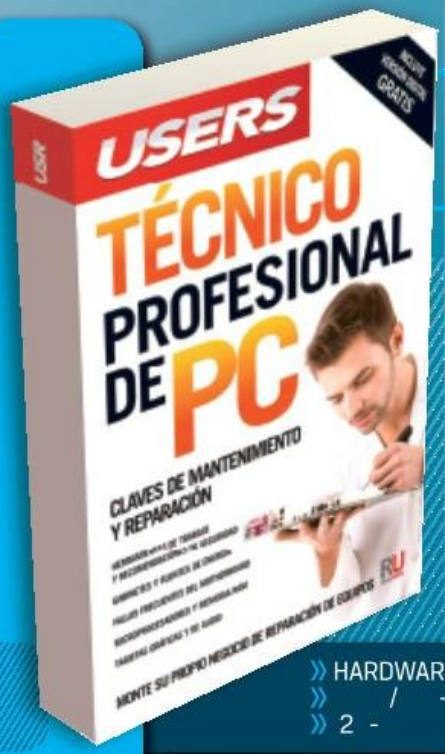
Sentencias y operadores

Uso de arreglos y punteros

CONÉCTESE
CON LOS MEJORES LIBROS DE COMPUTACIÓN

LLEGAMOS A TODO EL MUNDO
VÍA OCA* Y DHL**
usershop.redusers.com
usershop@redusers.com
+54 (011) 4110-8700

* SÓLO VÁLIDO EN LA REPÚBLICA ARGENTINA / ** VÁLIDO EN TODO EL MUNDO EXCEPTO ARGENTINA

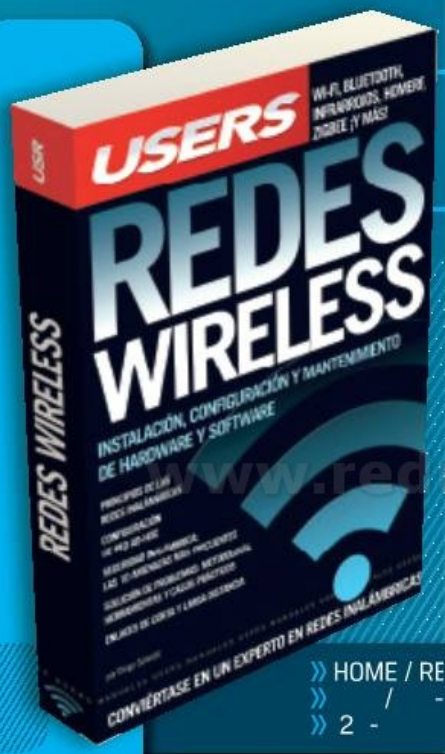


MONTE SU PROPIO NEGOCIO DE REPARACIÓN DE EQUIPOS

» HARDWARE
» / - 2
» 2 -

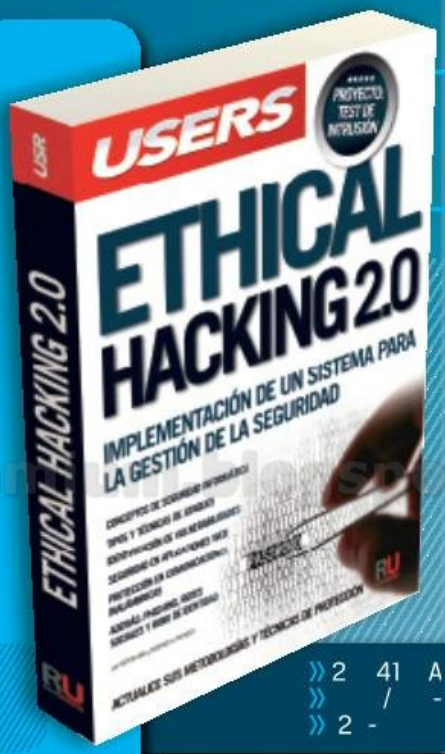
CAPACÍTESE PARA OBTENER UNA MEJOR SALIDA LABORAL

» HARDWAR .
» / - 2
» 2 -



CONVIÉRTASE EN UN EXPERTO EN REDES INALÁMBRICAS

» HOME / RE 2
» / - 2
» 2 -



ACTUALICE SUS METODOLOGÍAS Y TÉCNICAS DE PROTECCIÓN

» 2 41 A -31- 3
» / - 2
» 2 -

TÍTULO: Programación en C
AUTOR: Guillermo Brolin
COLECCIÓN: Pocket Users
FORMATO: 19 x 13.5 cm
PÁGINAS: 96

Primera impresión realizada en julio de MMXIII.

Sevagraf, Costa Rica 5226, Grand Bourg, Malvinas Argentinas, Pcia. De Buenos Aires.

ISBN 978-987-1949-13-7

Brolin, Guillermo
Programación en C / Guillermo Brolin; coordinado por Paula Budris. - 1a ed. - Buenos Aires: Fox Andina; Dalaga, 2013.
96 p.; 19x13 cm. - (Pocket Users; 33)

ISBN 978-987-1949-13-7

1. Informática. I. Budris, Paula, coord. II. Título
CDD 005.3

VISITE NUESTRA WEB

EN NUESTRO SITIO PUEDE OBTENER, DE FORMA GRATUITA, UN CAPÍTULO DE CADA UNO DE LOS LIBROS EN VERSIÓN PDF Y PREVIEW DIGITAL. ADEMÁS, PODRÁ ACCEDER AL SUMARIO COMPLETO, LIBRO DE UN VISTAZO, IMÁGENES AMPLIADAS DE TAPA Y CONTRATAPA Y MATERIAL ADICIONAL.

RedUSERS
COMUNIDAD DE TECNOLOGIA

redusers.com

Nuestros libros incluyen guías visuales, explicaciones paso a paso, recuadros complementarios, ejercicios, glosarios, atajos de teclado y todos los elementos necesarios para asegurar un aprendizaje exitoso y estar conectado con el mundo de la tecnología.

LLEGAMOS A TODO EL MUNDO VÍA OCA* Y DHL**

* SÓLO VÁLIDO EN LA REPÚBLICA ARGENTINA // ** VÁLIDO EN TODO EL MUNDO EXCEPTO ARGENTINA

usershop.redusers.com usershop@redusers.com + 54 (011) 4110-8700

Prólogo al contenido

Los orígenes del lenguaje C se remontan a la década de 1970, cuando Dennis M. Ritchie pensó en obtener un lenguaje destinado a desarrollar sistemas operativos. En este contexto surgió C, desde la necesidad de codificación del sistema UNIX.

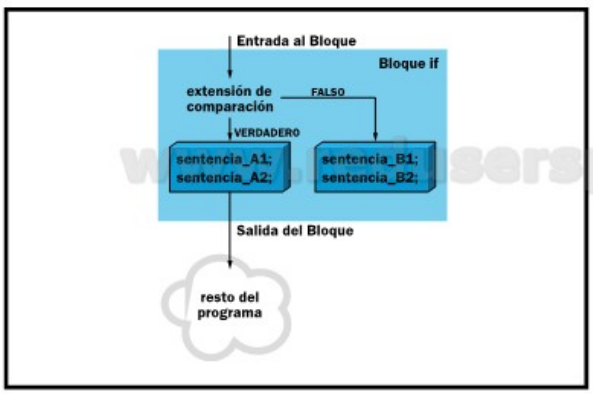
Debemos considerar que la versión 5 de este sistema operativo fue considerada el estándar del lenguaje durante mucho tiempo y fue documentado por Brian W. Kernighan y Dennis M. Ritchie en la primera edición de su obra The C Programming Language en el año 1978.

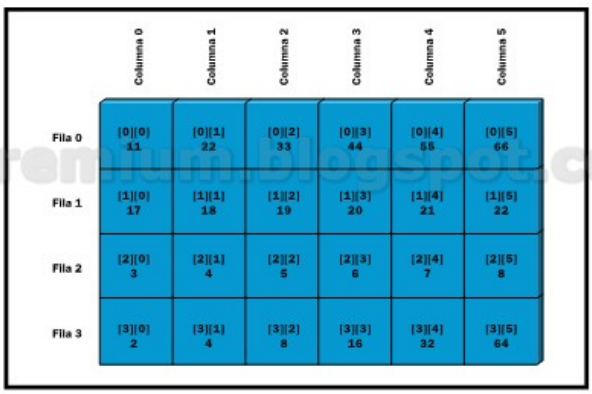
Pero la aparición de C obedece a ciertos hechos que nos permiten trazar sus orígenes en forma precisa. Por ejemplo, podemos mencionar la creación de un lenguaje llamado BCPL, el cual fue desarrollado por Martin Richards. Se trató de un lenguaje que basó su funcionamiento en la influencia de lenguaje llamado B desarrollado por Ken Thompson. Todo esto sirvió como base para que el desarrollo de C se llevara a cabo durante el año 1971.

En el año 1983, la American National Standard Institute (ANSI) estableció la creación de un comité destinado exclusivamente a definir el estándar del lenguaje con dos objetivos fundamentales: el lenguaje debía ser independiente de la plataforma en donde esté implementado y no debía poseer ambigüedades. Esto porque la necesidad de una actualización del lenguaje se hizo cada vez más evidente. Lo que se logró en esta actualización fue documentado en la segunda edición de la obra The C Programming Language.

Teniendo presente los orígenes de este lenguaje ya podemos darnos cuenta de los alcances y la importancia del mismo, por esta razón necesitamos introducirnos en su sintaxis. Este libro se encargará de acompañarnos en los primeros pasos para entender los conceptos básicos de la programación en C y nos dejará herramientas para que podamos implementar aplicaciones de consola, sirviéndonos además como base para el aprendizaje de lenguajes de mayor nivel, como C++.

Contenido del libro





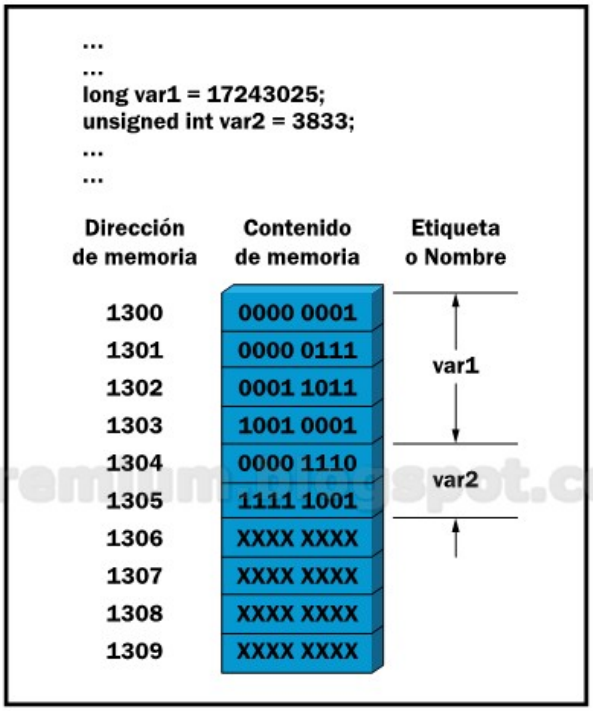
*04 Punteros

*05 Estructuración de programas

* Servicios al lector

ON WEB

*06 Estructuras de datos



*01

Primeros pasos en C

Surgido en sus inicios como un lenguaje de programación de sistemas operativos, C fue desarrollado como una herramienta para facilitar la codificación del sistema operativo UNIX. Vamos a experimentar los primeros pasos en la programación de este lenguaje que, seguramente, nos resultará fascinante por su sencillez y capacidad.

↗↗↗

El entorno de programación

Cualquiera sea el lenguaje de programación que estemos utilizando, existen, básicamente, cuatro pasos fundamentales para la creación de un programa. Estos son: **edición**, **compilación**, **vinculación** y **ejecución**.

La edición es el proceso mediante el cual el usuario escribe el código fuente. La escritura en lenguaje C es sensible al uso de mayúsculas y minúsculas, por lo que debemos prestar atención durante la edición. No es lo mismo, como ya veremos, escribir **Printf()** que **printf()**. Para este proceso, existen diversos **entornos de desarrollo** o **IDE** (por sus siglas en inglés: **I**ntegrated **D**evelopment **E**nvironment), que nos ofrecen herramientas que facilitan la correcta escritura del código fuente porque resaltan con colores las palabras reservadas del lenguaje. Un programa fuente en C se reconoce como tal cuando la extensión del archivo termina en **.c**, por ejemplo, **archivo1.c**. Esta extensión de archivo es la convención usada para los programas fuente, y es la que emplean la mayoría de los compiladores para procesar el código.

La compilación es el proceso por el cual el compilador analiza el código fuente: filtra toda la información que no forma parte de él (por ejemplo, los comentarios de programa), evalúa su sintaxis para detectar y reportar errores en la escritura, y lo convierte a un lenguaje que la computadora pueda interpretar. El producto final del compilador es un archivo objeto, el cual se identifica con la extensión **.obj** y lleva el mismo nombre que el programa fuente; como ejemplo podemos mencionar **archivo1.obj**.

Durante la vinculación se combinan todos los archivos que hayan sido generados por el compilador y se reemplazan todos los nombres de funciones del código por sus respectivos módulos de código, extraídos de las librerías de programa. El producto final de esta etapa es un archivo ejecutable, que en un entorno de Microsoft Windows tiene extensión **.exe**, por ejemplo, **archivo1.exe**.

La ejecución es aquella etapa en donde efectivamente corremos el programa para analizar su funcionamiento. En este punto, debemos aclarar que, previamente a la ejecución del programa, es muy importante saber qué debemos esperar cuando lo corremos.

El lenguaje C, al ser tan simple en su estructura, resulta una herramienta muy poderosa: va a hacer exactamente lo que le digamos que haga a través de la programación. Es decir, como producto final de esta etapa, podemos obtener diversos resultados que van desde la correcta ejecución del programa hasta el bloqueo completo de la computadora, o más aun, si programamos indiscriminadamente esperando que en algún momento el lenguaje vaya a advertirnos sobre manejos indebidos de la memoria o el hardware, podemos llegar a producir daños irreparables sobre estos.

Manos a la obra

Como la mejor manera de aprender a programar es programando, veremos aquí la forma de realizar nuestras primeras líneas de código, para obtener un mensaje en la pantalla de la computadora. Así, nos familiarizaremos con la dinámica de escritura de código. Pero antes de empezar a escribir, debemos tener algún entorno de programación IDE.

En esta obra, utilizamos el entorno que ofrece **Turbo C**, de Borland Internacional Inc., el cual posee las capacidades de edición,

LIBRERÍAS DE PROGRAMA

Son las encargadas de extender el lenguaje más allá de las librerías nativas suministradas como parte del lenguaje C. Por ejemplo, pueden contener código para realizar operaciones trigonométricas, operaciones con cadenas de texto, manejo de fecha y hora, etcétera.

compilación, vinculación y ejecución. No obstante, para la edición, es factible emplear cualquier editor de texto, tal como el **Bloc de Notas** de Windows o el editor **Microsoft Word**. Si este es el caso, debemos recordar guardar el programa fuente en formato texto con la extensión **.c**, para que luego el compilador pueda procesar el código.

Ejecutamos el editor y, sin preocuparnos mucho por entender el significado de cada una de las líneas de código, introducimos el siguiente programa, sin omitir nada de lo que está escrito:

```
/* EJEMPLO 1.1 – Mi primer programa en C */

#include <stdio.h>

void main()
{
        printf("Hola mundo\n");
}
```

Una vez ingresado el código, grabamos en una carpeta el archivo con el nombre **hola.c**, y ya estamos listos para compilar el programa. Vamos al menú **Compile** y seleccionamos **Compile to OBJ**, para generar el archivo objeto. Si todo va bien, el compilador nos indicará que la compilación ha tenido éxito y que el archivo **hola.obj** fue creado satisfactoriamente. Por último, solamente resta vincular el archivo objeto con las librerías de programa, para generar el archivo ejecutable. Vamos al menú **Compile** y seleccionamos **Link EXE file**. Finalmente, si no hay ningún inconveniente, el compilador nos indicará que la vinculación ha tenido éxito y que el archivo **hola.exe** fue creado satisfactoriamente.

Ahora, tenemos que comprobar el resultado de nuestro programa. Vamos al menú **Run** y seleccionamos **Run**. Una vez hecho esto,

el programa se ejecuta y termina inmediatamente, pues su única función es imprimir un mensaje en la pantalla del usuario. Tras la finalización del programa, el control se devuelve al entorno IDE. El mensaje se muestra en la pantalla del usuario, y para visualizarlo vamos al menú **Run** y elegimos **User screen**. Allí, se presenta nuestro mensaje:

```
Hola mundo
```

¿Qué hace el programa?

Hasta aquí, hemos conseguido realizar exitosamente una aplicación en C cuya única tarea es la de mostrar un mensaje en pantalla, pero ¿qué es lo que hace que esto sea así?, ¿cómo sabe la aplicación que el mensaje lo debe mostrar en pantalla y no, por ejemplo, enviarlo por uno de los puertos de comunicación de la computadora?

Pues bien, analicemos en detalle cada una de las líneas de programa para entender su funcionamiento. Veamos la primera:

```
/* EJEMPLO 1.1 – Mi primer programa en C */
```

Como podemos observar, esta línea no forma parte del código, en el sentido de que no aporta instrucciones a la computadora para hacer algo, sino que simplemente está allí para transmitir información a quien interpreta el programa. Es, simplemente, un comentario. Observamos que el comentario está encerrado entre los símbolos **/*** y ***/**. Usualmente, cuando el compilador se encuentra con ellos, descarta por completo cualquier cosa que esté escrita entre ellos, sin importar cuántas líneas tenga (inclusive, si hay líneas de código no las tiene en cuenta), y continúa buscando más código real para procesar al final del comentario.

En la segunda línea escribimos lo siguiente:

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

Aquí, nos encontramos con la directiva de pre-proceso **#include**, la cual no forma parte del código ejecutable, pero sí es absolutamente indispensable para el correcto funcionamiento del programa. De hecho, si esta línea no estuviera, el compilador nos arrojaría un error. El símbolo # indica que es una directiva de pre-proceso y le está diciendo al compilador que debe hacer algo antes de compilar el código fuente; en este caso, que tiene que **incluir** en nuestro programa el archivo **stdio.h**, que es la librería que contiene las declaraciones de las funciones de entrada/salida estándar. Este archivo se llama **header** o **cabecera** y se lo identifica con la extensión **.h**. De esta manera, cuando el compilador, en su proceso de codificación, vaya encontrando las funciones a las que el código fuente hace referencia, sabrá interpretarlas y qué hacer con ellas.

Desde la tercera línea de código en adelante, se declara una función:

```
void main()
{
        printf("Hola mundo\n");
}
```

SOBRE FUNCIONES

Cuando programamos en C, es muy importante tener siempre presente y no perder de vista el hecho de que estamos programando funciones, cada una de las cuales puede recibir ninguno, uno, o más argumentos y puede retornar o no valores a quien llamó la función.

En este caso, el nombre de la función es **main()** y no es arbitrario ni podría ser distinto. La razón de esto es que todo programa C contiene una función con este nombre y es el punto de partida, es por donde el programa comienza su ejecución. La función está precedida por la palabra reservada **void**, que indica el tipo de datos que retorna la función. Aquí, particularmente, **void** nos indica que la función no retorna valores a quien la llama (en este caso, el sistema operativo), pero hay veces en donde vamos a querer devolver datos al sistema operativo, por lo que la palabra reservada será diferente.

De la misma manera, vemos que la función contiene dos paréntesis que se abren y se cierran sin encerrar argumentos. Estos paréntesis encierran una definición de los tipos de datos que se van a pasar a la función, pero cuando están vacíos significa que ninguna información es necesaria para su funcionamiento.

Finalmente, podemos observar que, seguido de la definición de la función, encontramos una estructura de código encerrado entre dos llaves **{...}** que abren y cierran. Lo que encierran es el **cuerpo de la función**, la cual se encarga de definir categóricamente lo que ella está programada para hacer.

En este ejemplo, la función **main()** está programada para emitir un mensaje a través de la pantalla de la computadora, y de esto se encarga la función **printf()**, cuya definición está incluida en la librería estándar **stdio.h**, que viene con el lenguaje C. Consideremos que la información en los paréntesis que siguen inmediatamente a

PALABRAS RESERVADAS

En C existen palabras reservadas, que son aquellas que tienen un significado especial para el lenguaje. Por ello, no pueden ser utilizadas para otros propósitos tales como, por ejemplo, definición de variables o constantes, ya que el compilador les va a dar otro significado y arrojará un error.

la función se denomina **argumento** y define el tipo de datos que se va a pasar a ella. En este caso, son del tipo **string** o **cadena**. Esta función es la que canaliza la información que recibe como argumento hacia el área de memoria de la computadora encargada de mostrar dicha información en pantalla. Notemos aquí que la cadena de texto que se pasa a la función **printf()** termina en **\n** y no forma parte del resultado final que se obtiene en pantalla.

Por último, debemos notar algo muy importante, que es la presencia del punto y coma (**;**) al final de la sentencia **printf()**. Esta es la forma que tiene el lenguaje para diferenciar entre el final de una y el inicio de la siguiente.

Errores comunes

Durante la escritura de un programa, es factible que cometamos errores. Cuando introducimos código que luego el compilador deberá traducir en instrucciones para que una computadora las lleve adelante, debemos atenernos a todas las estrictas reglas del lenguaje, de lo contrario, el resultado que obtendremos será un mensaje de error u otro totalmente inesperado.

Por ejemplo, si nos olvidamos de colocar un punto y coma entre dos sentencias **printf()**, el compilador interpretará la segunda sentencia como parte de la primera, y esto podrá ocasionar un mensaje de error. Generalmente, para el usuario que recién se inicia en el lenguaje, e inclusive para programadores que llevan varios años de oficio, el olvido del punto y coma es un error bastante frecuente, por lo que debemos, desde un principio, tomar como buena práctica el hecho de colocarlo siempre al final de cada línea de código que escribamos.

Usualmente, los errores de tipografía son más frecuentes de lo que podemos imaginarnos. En el mejor de los casos, serán detectados durante el proceso de compilación y/o vinculación y nos darán la oportunidad de revisar nuestro código y corregirlo. Puede darse el caso de que el olvido de una coma donde se espera que vaya,

o un punto y coma de más donde no debería estar, produzca un código perfectamente ejecutable, pero sin dudas distinto del que nosotros esperamos, causando comportamientos erráticos durante la ejecución del programa o realizando tareas que no estaban dentro de nuestra lógica de programación.

Sin embargo, los problemas descriptos anteriormente son detectables a través de un análisis detallado de la sintaxis del código. Pero, por supuesto, estos inconvenientes no son los únicos que pueden ocasionar errores. Más grave aun puede resultar el hecho de equivocarnos en la lógica del programa, en las instrucciones encargadas de la toma de decisiones. Esto suele ser particularmente probable en aquellos programas complicados, con muchas decisiones y/o comparaciones, las cuales pueden ser perfectamente correctas desde el punto de vista de la programación del lenguaje, pero incorrectas desde el de la lógica. Es de suponer que el resultado final sea completamente distinto del que esperamos. En general, estos problemas son los más difíciles de detectar, sobre todo si los pasos de programación no están lo suficientemente comentados.

El siguiente paso

Hasta aquí, nos encontramos con las herramientas básicas para crear programas en C capaces de mostrar información estática en la pantalla de la computadora. Pudimos realizar distintas variaciones al programa analizado e, inclusive, experimentar con los caracteres de control para entender su funcionamiento. Pero, las aplicaciones más versátiles son aquellas en donde el usuario puede interactuar con el programa, por ejemplo, a través del ingreso de datos desde el teclado, desde un puerto de comunicación de la PC o desde algún otro dispositivo de entrada de datos para que, de esta forma, el programa pueda tomar decisiones en función de la información que ha sido suministrada.

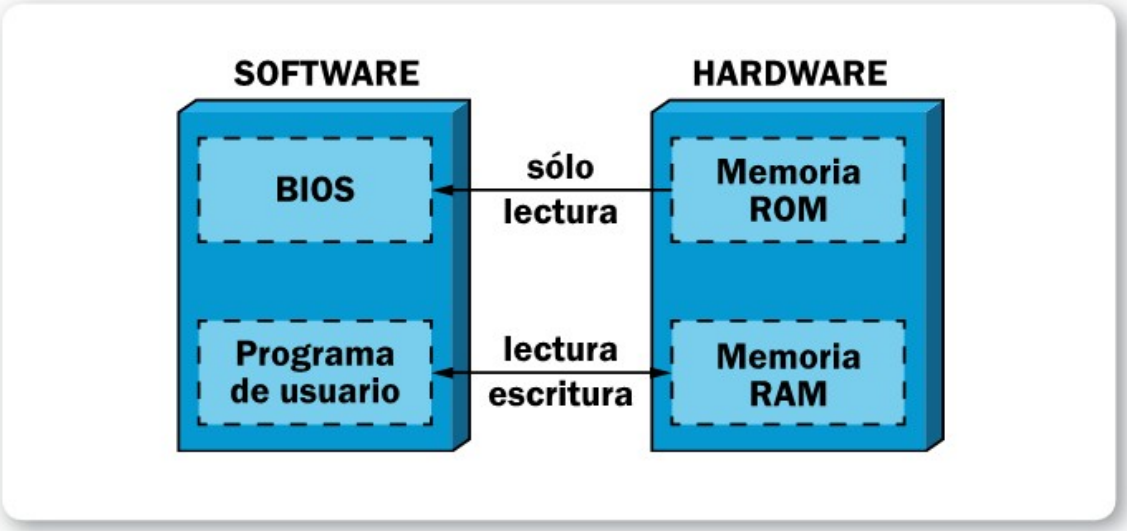


Figura 1. Accesos permitidos del software hacia la memoria ROM y RAM.

El programa va a tener que manejar estos datos, que serán distintos cada vez que se ejecute. Por lo tanto, vamos a necesitar una nueva herramienta que permita hacerlo. Esta herramienta se denomina **variable** y, como veremos, las hay de distintos tipos, pero todas tienen la capacidad de recordar la información que reciben y de entregarla cada vez que es solicitada por el programa.

Para poder entender cómo maneja las variables de la computadora, primero debemos comprender cómo está organizada su memoria.

Existen, básicamente, dos grandes tipos de memoria. La información que maneja un programa cuando está en ejecución la almacena en algún lado y luego puede recuperarla. Esto significa que el programa puede leer y escribir a la memoria de la computadora. Este tipo de memoria se denomina **RAM** o **memoria de acceso aleatorio** (por sus siglas en inglés, **R**andom **A**ccess **M**emory). Comúnmente, se la llama **memoria principal** o de programa, ya que es donde reside el programa cuando se ejecuta.

El otro tipo de memoria, más específica, se denomina **ROM** o **memoria de solo lectura** (por sus siglas en inglés, **R**ead **O**nly **M**emory). Tal como su nombre lo indica, la computadora solamente puede leer datos de la memoria, que ya viene de fábrica con información previamente almacenada. Normalmente, lo que se almacena

en estas memorias se denomina **BIOS** o **B**asic **I**nput-**O**utput **S**ystem. Estos BIOS son los que cargan y manejan el sistema operativo básico de una computadora o los dispositivos conectados a ella.

BITs, BYTEs y palabras

Pongamos las cosas un poco más en claro y hagamos una analogía. Resulta conveniente pensar la memoria RAM como una secuencia de cajas ordenadas, en donde únicamente es posible tenerlas en uno de dos estados: **llenas** o **vacías**. Al estado de caja llena le asociamos el número binario **1** y la condición **verdadero**, mientras que al estado de caja vacía le asociamos el número binario **0** y la condición **falso**. De esta forma, decimos que cualquiera de las cajas representa un **BIT** (acrónimo de **B**inary **d**igit), que es la menor unidad en el sistema binario.

Estas cajas o BITs pueden estar agrupadas en determinadas cantidades. Cuando los BITs están en grupos de 8, forman un **BYTE** y a este último lo etiquetamos con un número que lo identifica, empezando desde el 0, continuando con el 1 y así hasta el último BYTE que tengamos en la memoria de la computadora. Esta etiqueta que le asignamos a cada BYTE se conoce como su **dirección**, así como la dirección de nuestras casas nos identifica, por ejemplo, al momento de recibir una correspondencia.

Cuando los BITs están en grupos de 16, tenemos 2 BYTEs, y al grupo de dos BYTEs lo denominamos **palabra** o **word**.

APRENDER A AHORRAR

Si bien es posible tener tantas variables como se quiera en nuestros programas, no debemos perder de vista el hecho de que estamos trabajando con máquinas que tienen una capacidad de memoria limitada.

En esta obra, utilizaremos el vocablo castellano para referirnos al grupo de dos BYTEs. La dirección de una palabra está identificada por la dirección del primer BYTE que la conforma.

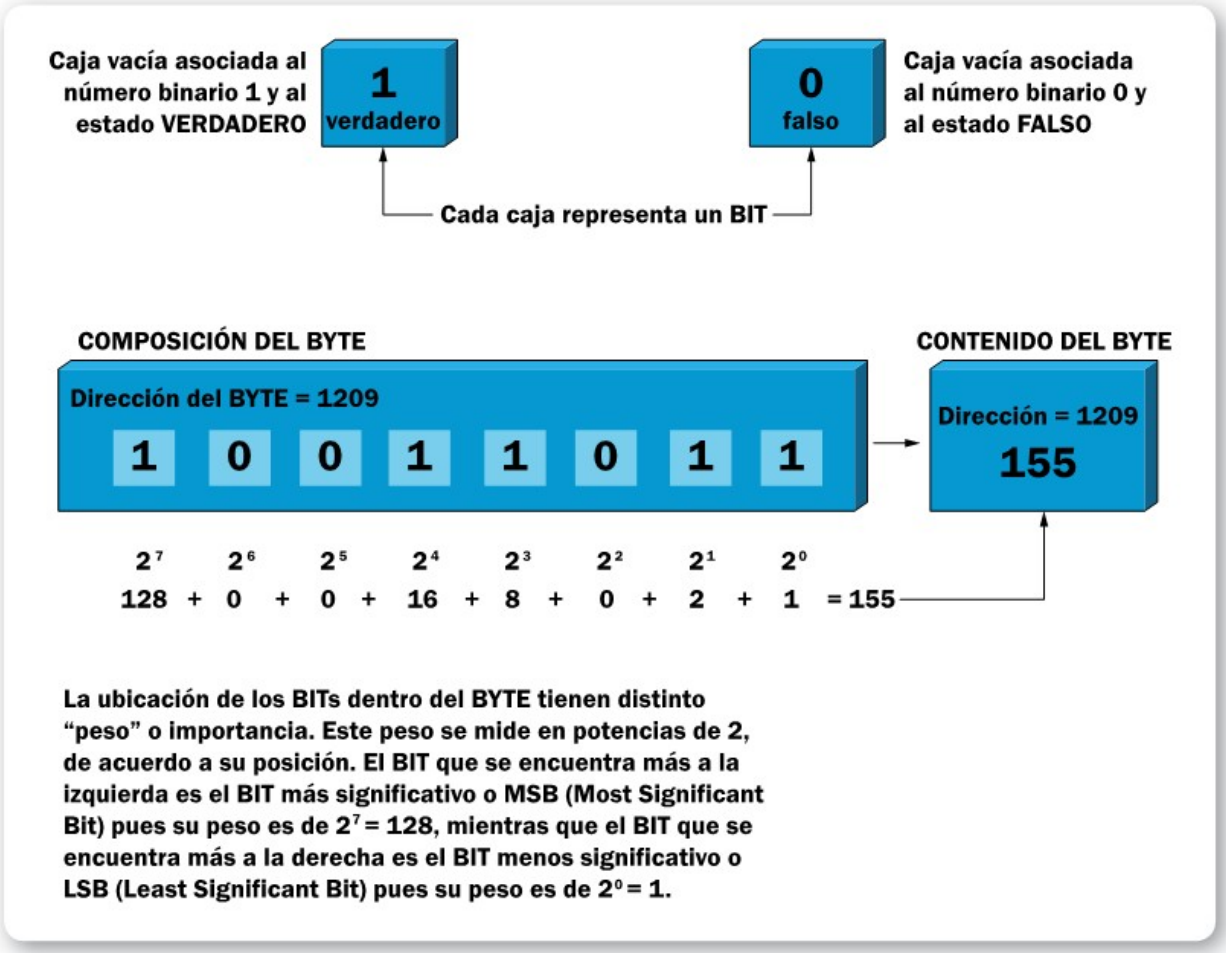


Figura 2. Estructura de BITs y BYTEs en la memoria de la computadora. Podemos observar la forma en que se organizan los BITs para formar los BYTEs y cómo se obtiene el valor decimal del contenido de un BYTE de acuerdo con el peso de cada BIT.

¿Qué es una variable?

Una **variable** no es ni más ni menos que una porción de memoria en la computadora, que puede ocupar uno o varios BYTEs continuos. Todas las variables tienen un nombre y es posible almacenar u obtener información de una de ellas a través de él. Esto es

similar, por ejemplo, a las direcciones de Internet y a las direcciones IP. Está claro que no vamos a acordarnos la dirección IP pública de los sitios por los que navegamos, lo más conveniente es que exista alguna asociación entre la dirección IP real del sitio y algún nombre de fantasía o **alias** que sea más fácil de recordar para nosotros. Por ejemplo, es mucho más fácil acordarnos y escribir en nuestro navegador de Internet la dirección **www.microsoft.com**, que **http://207.43.196.254**.

Si bien el resultado que obtendremos es el mismo, claramente lo conveniente es utilizar los alias. Es más, estas direcciones IP pueden cambiar eventualmente, pero por lo general el alias se mantiene, de manera que la modificación pasa desapercibida para el usuario.

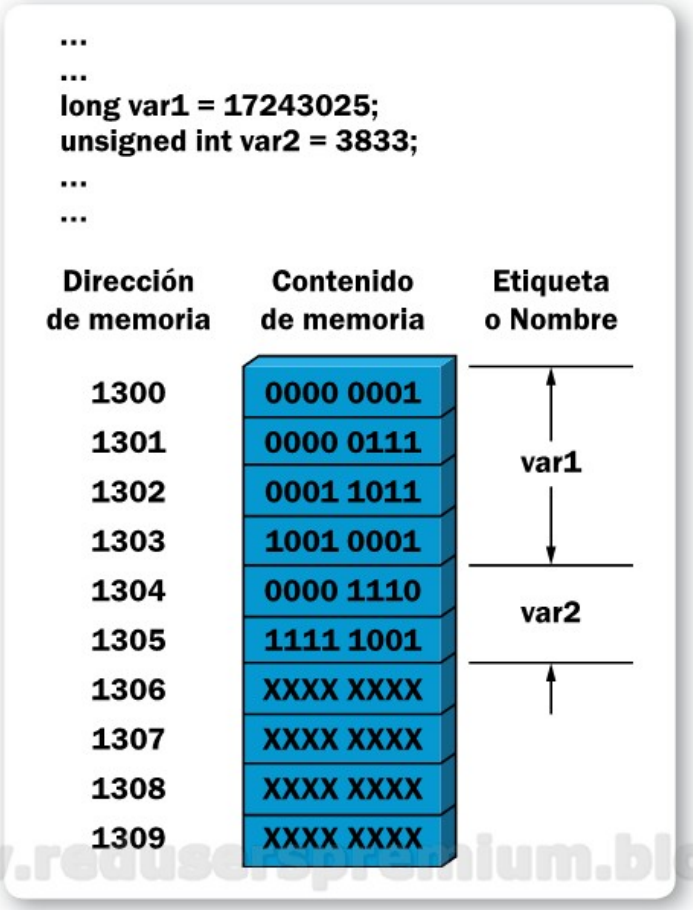


Figura 3. La asignación de nombres a las variables nos evita tener que recordar las direcciones en donde estas son creadas. Más aun cuando son invocadas, por lo que su posición de memoria puede cambiar constantemente.

La función **printf()** convierte, aplica formato, e imprime sus argumentos en la salida estándar, de acuerdo con lo que establece **formato**, y retorna como resultado de esta función la cantidad de caracteres impresos. La cadena **formato** consta de dos tipos de objetos:

a) Caracteres comunes, que son pasados directamente y sin tratamiento al flujo de salida.
b) Especificaciones de conversión, que convierten e imprimen los datos que son pasados como argumentos, desde el primero hasta el último, en orden sucesivo (**arg1**, **arg2**, etcétera). El comienzo de las especificaciones de conversión se señaliza con el símbolo **%** y terminan con un carácter de conversión, como vemos en la **Tabla 1**.

CARACTERES BÁSICOS DE CONVERSIÓN

▼ CARÁCTER	▼ TIPO DE DATO	▼ DESCRIPCIÓN
d, i	int	Número decimal (con signo).
U	int	Número decimal (sin signo).
x, X	int	Número hexadecimal (sin signo).
C	int	Carácter simple.
S	char *	Imprime caracteres de la cadena de texto dada como argumento, desde el principio hasta que se encuentra el final de cadena o carácter nulo (\0).
%	—	El argumento se convierte. Se imprime el símbolo %.

Tabla 1. Caracteres de conversión más utilizados en cadenas de texto. Todos deben ir precedidos del símbolo **%**.

Tipos de variables

Existen diferentes tipos de variables, y cada una de ellas se utiliza en función de la naturaleza del dato que maneja.

Variables de números enteros

Se emplean para poder manejar números enteros. Un número entero es aquel que no posee decimales.

Existen diferentes tipos, dependiendo básicamente del rango de datos que cada una de ellas puede manejar. El tipo de variable **int** ocupa, normalmente, 2 BYTEs (16 BITs) de memoria.

Esta cantidad, en realidad, la define el tipo de computadora y el compilador C que estemos utilizando pero, por lo general, en una PC ocupa 2 BYTEs. Entonces, el rango de valores con signo va desde el número -32768 hasta el 32767 y, sin signo, desde el 0 hasta el 65535, es decir, el tipo **int** puede manejar 216 = 65536 valores posibles.

Para poder usar una variable, primero tiene que estar declarada, y su declaración puede estar hecha dentro o fuera de las estructuras de funciones de nuestro programa.

Esto, como veremos más adelante, tiene que ver con su visibilidad, pero por ahora solo es necesario que nos concentremos en las declaraciones. De esta forma tengamos en cuenta que una variable tipo **int** se declara de la siguiente manera:

CARÁCTER NULO

El carácter nulo se indica con **\0** y señaliza la posición del final en una cadena de texto. Este carácter se agrega automáticamente al término de todas las cadenas cuando son creadas. Los caracteres totales de una cadena de texto exceden en uno a su longitud aparente.

```
int numero;                    /*VARIABLE CON SIGNO*/
unsigned int precio;    /*VARIABLE SIN SIGNO*/
```

De esta forma, el compilador está reservando dos posiciones de memoria (2 BYTEs), a las que les pone la etiqueta asignada por nosotros (**numero** o **precio**). En este caso, como la variable solamente está declarada pero no inicializada, cuando el compilador asigne el espacio en memoria se inicializará con un valor que dependerá de lo que hasta el momento haya tenido ese espacio de memoria.

Ahora bien, si tenemos la intención de asignar un valor de inicialización, lo podemos hacer en la misma sentencia:

```
int numero = 10;
unsigned int precio = 10000;
```

Además, tenemos la posibilidad de declarar más de una variable en una misma sentencia:

```
int numero1, numero2 = 179, numero3 = 17281;
```

Aquí, declaramos tres variables del tipo **INT**, de las cuales dos están inicializadas y una sin inicializar. Cada una de ellas ocupará 2 BYTEs.

MODIFICADOR UNSIGNED

Este modificador se utiliza anteponiendo la palabra reservada **UNSIGNED** (sin signo) antes del tipo de variable para especificar que se desea cambiar el rango de valores apropiados para la variable.

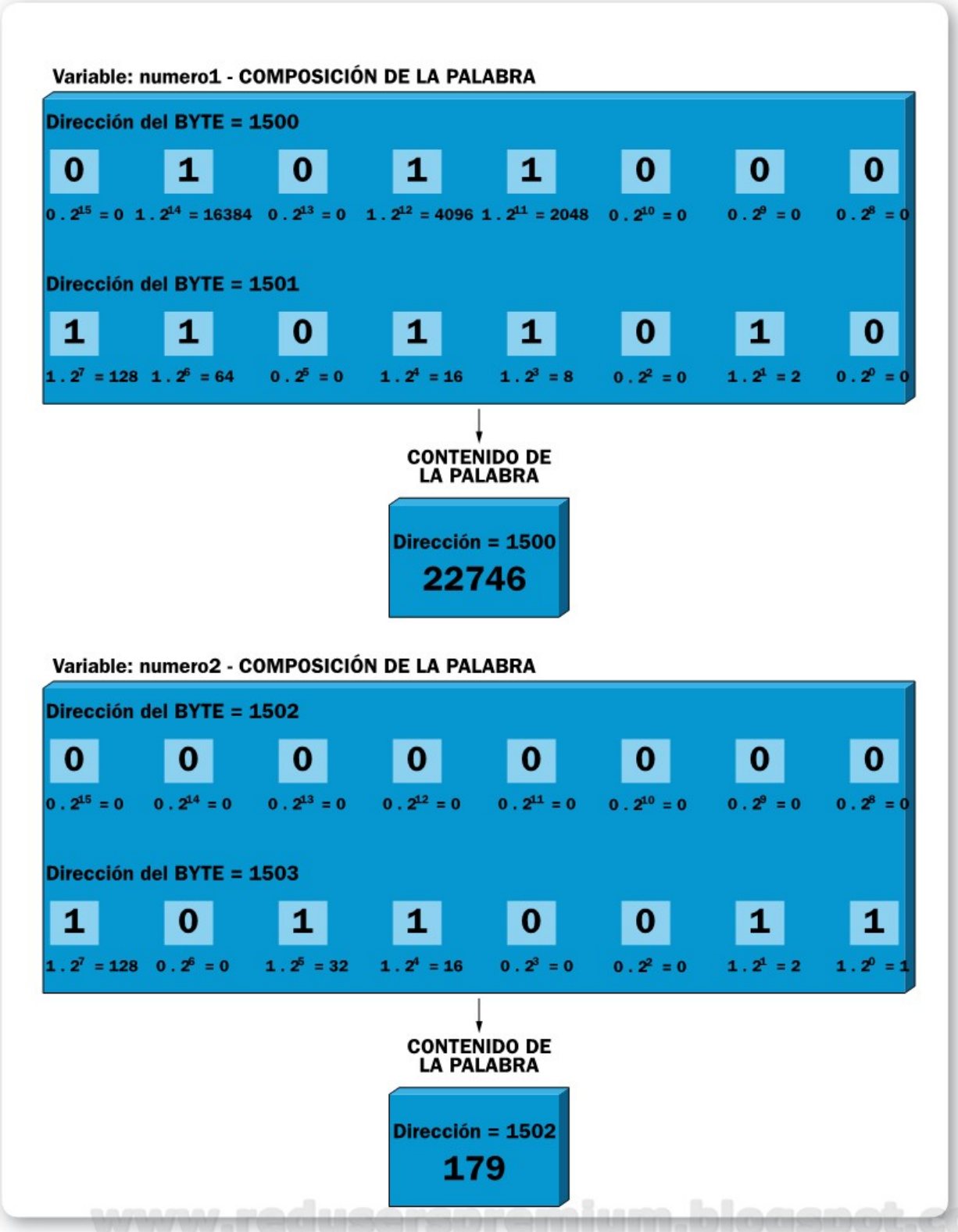


Figura 4. Organización de la memoria de datos durante la declaración e inicialización de variables. La dirección es arbitraria, solamente la variable **numero2** fue inicializada. La **numero1** toma un valor residual de memoria.

Existen otros tipos de variables para enteros, que difieren únicamente en la cantidad de memoria que se reserva para el almacenamiento de datos: **short** y **long**. El tipo de variable **short** reserva 2 BYTEs de memoria, mientras que el tipo **long**, 4 BYTEs.

Variables de caracteres

El tipo de variable **char** permite almacenar caracteres, los cuales se traducen a números de acuerdo con el código ASCII estándar. El rango de valores posibles está comprendido entre -128 y 127 para valores con signo, y entre 0 y 255 para valores sin signo. El espacio de memoria que ocupa es 1 BYTE (8 BITs). La declaración e inicialización de una variable tipo **char** es como vemos a continuación:

```
char inicial = 'A';          /*USA COMILLAS SIMPLES*/
```

Como el carácter finalmente se termina traduciendo al código ASCII correspondiente, también hubiera sido correcto hacer la asignación directa del número, es decir:

```
char inicial = 65;           /*ASIGNACION DIRECTA*/
```

En el primer caso, declaramos la variable y la inicializamos con el carácter **A** en forma explícita, únicamente con comillas simples para encerrarlo (las dobles se utilizan para cadenas de texto). En el segundo caso, declaramos e inicializamos la variable con el número ASCII correspondiente al carácter **A** en forma directa. Ambos modos son correctos, pero debemos saber que el primero resulta más fácil de entender.

Un uso frecuente de este tipo de variable (con o sin signo) es el almacenamiento de números pequeños dentro del rango admitido.

Esto nos permite ahorrar memoria en el sentido de que utiliza un solo BYTE para los datos.

```
unsigned char contador = 0;        /*CUENTA EVENTOS*/
```

En este ejemplo, se declara la variable **contador** como **char** sin signo, para llevar el registro de la ocurrencia de eventos. Podremos contar hasta 256 eventos.

Variables de punto flotante

Este tipo de variable almacena números en **punto flotante**, que son una representación conveniente empleada para números enteros muy grandes o fraccionarios muy pequeños. Existen tres tipos diferentes de variables en punto flotante.

TIPOS DE VARIABLE EN PUNTO FLOTANTE

TIPO	MEMORIA QUE OCUPA	RANGO DE VALORES VÁLIDOS	PRECISIÓN
Float	4 BYTEs	± 3.4E38	6 dígitos
double	8 BYTEs	± 1.7E308	15 dígitos
long double	10 BYTEs	± 1.2E4932	19 dígitos

Tabla 2. Espacios de memoria y rango de validez de los tipos de variable en punto flotante.

A continuación, se detallan algunos claros ejemplos de declaraciones de variable en punto flotante:

```
float radio = 3.4f;                          /*FLOAT*/
double habitantes = 2.7E17;                  /*DOUBLE*/
long double enorme = 1234567.89L;            /*LONG DOUBLE*/
```

Visibilidad de variables

Una variable existe solamente dentro del entorno en donde fue creada. Hasta aquí, las que hemos empleado han sido creadas o declaradas dentro del bloque funcional **main()**, que constituye su entorno. Esto quiere decir que tienen validez únicamente dentro del bloque **main()**, siendo imposible cualquier referencia a ellas fuera de él.

El cuerpo de cada función se denomina **bloque**, y las variables que son creadas cuando son declaradas y destruidas al finalizar un bloque se llaman **automáticas**. La extensión dentro de un programa de una variable se conoce como su **alcance**. Estas variables son **locales** al bloque y no tienen existencia en ningún otro lado.

Variables externas

Debemos tener en cuenta que cuando trabajamos en un programa compuesto por módulos o archivos, puede ocurrir que queramos acceder a una variable global que está definida en otro lado. Por ejemplo, si tenemos variables globales definidas en otro archivo con las siguientes sentencias:

```
int numero = 25;
long array[10] = { 0L };
```


Entonces, en la función desde donde queramos acceder especificamos que son externas. Para esto empleamos **extern**:

```
extern int numero;
extern long array[10];
```

Estas sentencias no crean las variables, sino que las identifican y especifican que están definidas en algún otro lado.

Variables estáticas

Si queremos que la variable perdure más allá de su ámbito de validez, entonces debe ser declarada con el modificador **static**. ¿Cuál es el objetivo de querer que una variable no pierda su validez? En ciertas ocasiones, puede ser necesario que debamos mantener un contador dentro de una función, pero que no tengamos que inicializarlo cada vez que esta se ejecuta. Sería conveniente que este contador conservara su valor a lo largo de sucesivas llamadas a la función. Pues bien, el lenguaje nos permite hacer esto cuando la variable está declarada como **static**.

Constantes

Es muy frecuente que en un programa tengamos que utilizar repetidamente un mismo valor que no cambia en el tiempo o entre sucesivas ejecuciones del programa. Podríamos definir la variable y

PROFESOR EN LÍNEA

Si tiene alguna consulta técnica relacionada con el contenido, puede contactarse con nuestros expertos: profesor@redusers.com

asignarle un valor determinado, pero lo más correcto sería definir una **constante**. Una constante es, entonces, una variable que no cambia en el tiempo. Existen dos formas de definirlas en C: con la directiva **#define**, o con la palabra reservada **const**.

Cuando usamos **#define**, la sintaxis es la siguiente:

```
#define nombre_constante                valor_constante
```

Podemos observar que en las directivas de pre-proceso no se utiliza el punto y coma. Cuando usamos la palabra reservada **const**, la sintaxis es la siguiente:

```
const tipo nombre_constante = valor_constante;
```

En este último caso, sí es necesario el uso de punto y coma. Por otro lado, debemos tener en cuenta que **const** puede figurar antes o después del tipo de constante.

RESUMEN

A lo largo de este capítulo, tuvimos nuestro primer contacto con el lenguaje y vimos una de las sentencias más populares: printf(). Hemos abarcado los distintos tipos de variables que utiliza el lenguaje y la forma en que estas se organizan en la memoria de la computadora. Ahora conocemos cómo se estructuran los programas en C y su particular sintaxis de escritura, a la cual debemos prestar mucha atención.

*02

Toma de decisiones

Luego de haber tenido un primer contacto con la estructura de programación en C, en este capítulo avanzaremos sobre el proceso de toma de decisiones, que nos brindará mucho más dinamismo en nuestros programas.

Tomar decisiones en C

El proceso de toma de decisiones nos brinda una gran flexibilidad a la hora de programar, ya que nos permite, de acuerdo con el resultado de una comparación de variables o constantes, seleccionar qué bloque de código se ejecutará a continuación. Pero antes de tomar decisiones debemos establecer algún mecanismo que nos posibilite comparar, que es justamente lo que veremos aquí.

Como las variables que podemos usar involucran generalmente números, resulta conveniente introducir algunos nuevos operadores que nos permitan comparar valores numéricos.

OPERADORES DE COMPARACIÓN ARITMÉTICA

▼ OPERADOR	▼ DESCRIPCIÓN
<	Es menor que
==	Es igual a
>	Es mayor que
>=	Es mayor o igual que
<=	Es menor o igual que
!=	Es distinto a

Tabla 1. Operadores básicos de comparación.

Debemos tener presente que, en C, es muy frecuente cometer errores tipográficos, los cuales no solamente generarán errores a la hora de ejecución del programa, sino que producirán resultados inesperados.

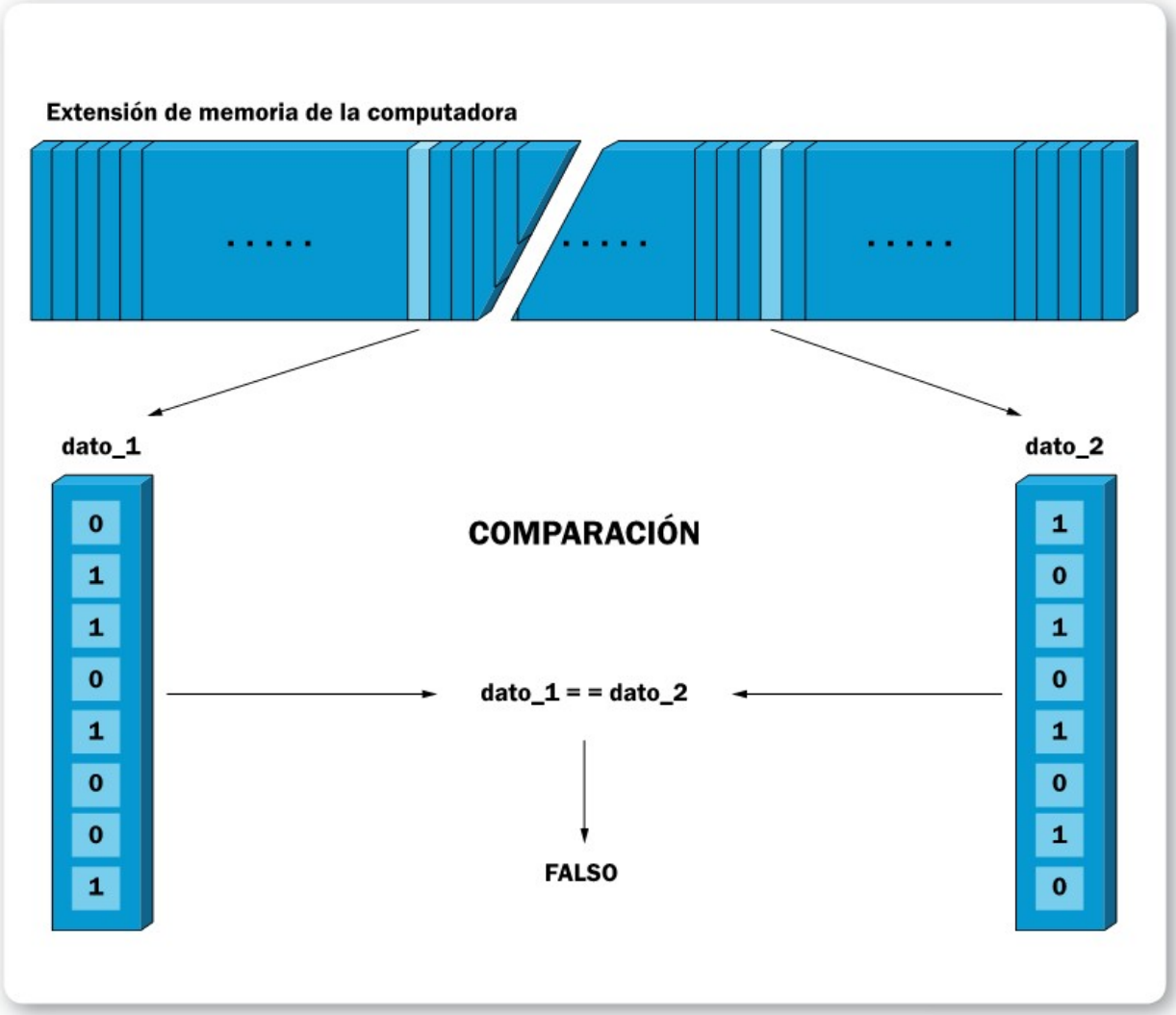


Figura 1. Proceso de comparación. Los datos son obtenidos de la memoria y comparados entre sí. En caso de que no sean iguales, el resultado de la comparación será falso.

OPERADOR ==

El operador de comparación "es igual a" se escribe con dos signos igual (==). Es un error frecuente escribirlo, en nuestros programas, con uno solo. Si no tenemos en claro el significado del doble signo, nos generará confusión. En C, el doble signo se utiliza para comparación, mientras que el simple, para asignación.

La sentencia if

Una vez presentados los operadores de comparación, necesitamos conocer, ahora, cómo está compuesta la estructura de comparación del lenguaje. La forma más simple de hacer una comparación en C es a través de la sentencia **if**, cuya forma estructural es la siguiente:

```
...
if (expresión_de_comparación)
{
        sentencia_A1;
        sentencia_A2;
        ...
}
else
{
        sentencia_B1;
        sentencia_B2;
        ...
}
...
```

El bloque de sentencias encerrado entre llaves, que sigue inmediatamente a la sentencia **if**, se ejecuta si el resultado de **expresión_de_comparación** es verdadero, mientras que si es falso, se ejecuta el bloque de sentencias encerrado entre llaves, que sigue inmediatamente a **else**. Como podemos observar, la sentencia **if (expresión_de_comparación)** no requiere el signo punto y coma (;) al final. Ahora podríamos preguntarnos si en cada una

de las sentencias que ejecuta el bloque **if**, podrían colocarse otros bloques **if**. La respuesta es **sí**. Esto es factible porque cada bloque de comparación es independiente, en cuanto a su funcionamiento, respecto de los otros. Tendremos así, **bloques de comparación anidados**.

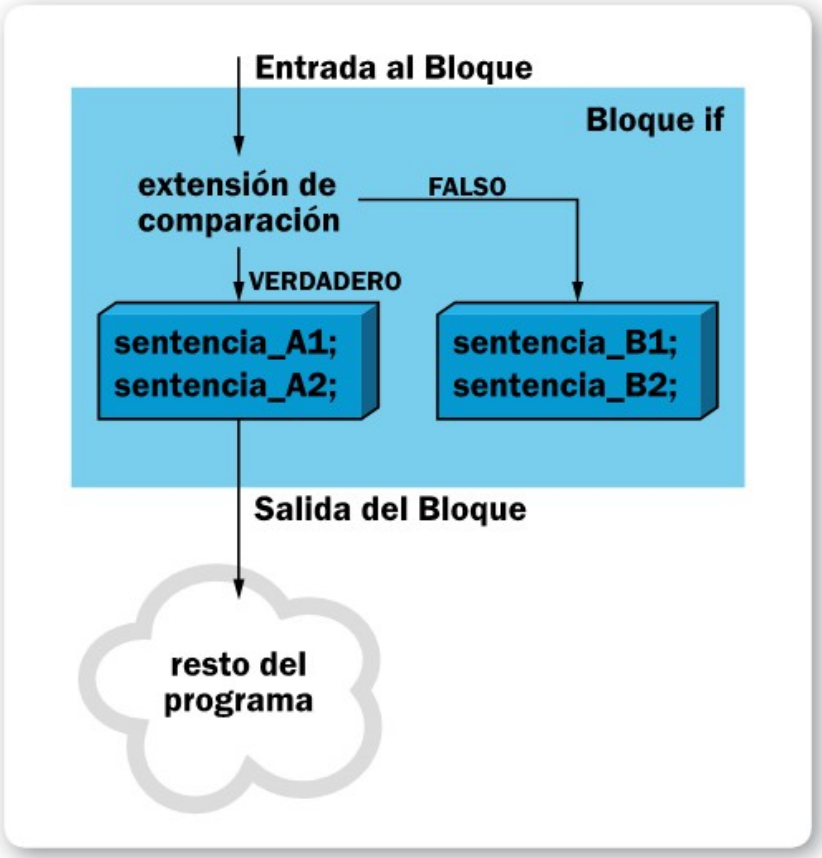


Figura 2. La forma más simple de control en la toma de decisiones en C la conforma el bloque de sentencia `if`.

Veamos un ejemplo de cómo funciona la comparación. Supongamos que tenemos un programa en donde manejamos una variable **edad** del tipo **int** que guarda la edad de una persona, y queremos que decida si está comprendida entre 30 y 40 años inclusive. Particularmente, deberá informarnos con un mensaje si la edad es de 38 años. La toma de decisiones, para este ejemplo, debe hacerse con sentencias **if** anidadas.

A continuación, veremos otros tipos de operadores que nos ayudarán a ampliar el espectro de posibilidades de comparación.

Operadores lógicos

Los **operadores lógicos** nos permiten combinar, en una sola expresión de comparación, varias expresiones individuales. Estas combinaciones pueden ser empleadas dentro de las expresiones de comparación de los bloques **if**.

El operador Y lógico: &&

Este operador nos permite combinar dos expresiones lógicas, es decir, expresiones que tienen un valor **verdadero** o **falso**. El resultado de la operación es verdadero si, y solo si, ambas expresiones lógicas son verdaderas. La forma lógica de la operación es:

```
dato_1 && dato_2
```

El operador NO lógico: !

El signo de admiración de cierre es el que utiliza el lenguaje para representar la forma lógica **no**. La función de este operador es invertir el resultado lógico: lo verdadero se vuelve falso, y lo falso se vuelve verdadero. Esto puede resultar un poco confuso al principio, pero su empleo en programas es más común de lo que

OPERADOR LÓGICO ||

Este operador nos permite verificar si alguno de los datos que resulta de la combinación lógica es verdadero. Debemos tener en cuenta que si es así, el resultado que obtendremos con la operación será verdadero.

podemos imaginar y, al mismo tiempo, es muy útil, ya que a veces puede ocurrir que resulte más sencillo verificar una comparación por verdadero y, luego, aplicando el operador **!**, obtener el resultado complementario (falso).

A continuación vemos un ejemplo de su uso:

```
if !(edad >= 30 && edad <= 40)
{
        /*Aquí la edad es menor de 30 años o
  mayor que 40 años */
        ...
}
else
{

        /*Aquí la edad esta comprendida entre 30 y
          40 años inclusive*/

        ...
}
```

El operador condicional

Existe otra sentencia que podemos usar para verificar datos, pero que no altera la secuencia de ejecución dependiendo de un resultado de comparación. En lugar de ello, simplemente produce un valor diferente. La forma general de representación es la siguiente:


```
condición ? resultado_1 : resultado_2
```

Operadores de BIT

Son parecidos a los operadores lógicos, pero su funcionamiento es diferente pues operan a nivel de BITs en números enteros. Aquí, enunciamos los operadores de BIT que posee el lenguaje.

Operador Y de BIT: &

Su forma de aplicación general es la siguiente:

```
dato_1 & dato_2
```

Operador O de BIT: |

Su forma de aplicación general es la siguiente:

```
dato_1 | dato_2
```

Operador O-exclusivo de BIT: ^

Su forma de aplicación general es la siguiente:

```
dato_1 ^ dato_2
```


Para entender de mejor forma el funcionamiento de este operador, mencionamos la tabla lógica o tabla de verdad que se utiliza para este operador:

OPERADOR ^		
▼ DATO_1	▼ DATO_2	▼ RESULTADO DE |
0	0	0
0	1	1
1	0	1
1	1	0

Tabla 2. Lógica de verdad O-exclusivo a nivel de BITs.

En la tabla de verdad de este operador observamos que cuando los datos tienen el mismo valor en BITs, arroja un 0, mientras que cuando los datos tienen valores diferentes, arroja un 1. Esto es útil cuando, por ejemplo, queremos verificar si dos datos son idénticos a nivel de BITs. En caso de que tengan idéntica composición de BITs, el resultado del operador O-exclusivo será 0.

Operador NO de BIT: ~

Este operador invierte los BITs que componen un BYTE y se lo utiliza para calcular el **complemento a 1**. El complemento a 1 de un número entero positivo n, escrito en notación binaria, se obtiene al invertir todos los BITs en forma individual, es decir, los 0 se reemplazan por 1 y los 1 se reemplazan por 0. Esta notación resulta de especial aplicación en computación, ya que nos permite la representación binaria de números negativos. La forma de aplicación general de este operador es la siguiente:


```
~dato_1
```

Trabaja de acuerdo con la tabla lógica siguiente:

OPERADOR ~	
▼ DATO_1	▼ RESULTADO DE ~
0	1
1	0

Tabla 3. Lógica de verdad NO a nivel de BITs.

Veamos un ejemplo:

```
char dato_1 = 107;
char resultado = ~dato_1;
```

Operador de desplazamiento a la derecha: >>

Su forma de aplicación general es la siguiente:

```
dato_1 >> cantidad_de_posiciones
```

Este operador produce el desplazamiento hacia la derecha de los BITs que componen la variable **dato_1** tantas posiciones como lo especifica **cantidad_de_posiciones**.

Debemos tener en cuenta que los espacios que van quedando a la izquierda del BIT más significativo se van completando con 0, mientras que el BIT menos significativo se pierde en cada desplazamiento. Veamos lo que acabamos de mencionar en el ejemplo que presentamos a continuación:

```
int dato_1 = 128;
int cantidad = 3;
int resultado = dato_1 >> cantidad;
```

La operación que se lleva a cabo a nivel de BITs, en sucesivos desplazamientos, es la siguiente:

OPERADOR →→			
▼ VARIABLE	▼ Nº DESPLAZAMIENTO	▼ CONTENIDO	
dato_1	0 (sin desplazar)	1000 0000	128
	1	0100 0000	64
	2	0010 0000	32
	3	0001 0000	16
Resultado	—	0001 0000	16

Tabla 4. Efecto de la aplicación del operador >> sobre una variable.

Cuando consultamos el valor de **resultado** luego del desplazamiento, obtenemos el valor **16** en decimal.

Operador de desplazamiento a la izquierda: <<

Este operador produce el desplazamiento hacia la izquierda de los BITs que componen la variable **dato_1** tantas posiciones como lo especifica **cantidad_de_posiciones**. Los espacios que van quedando a la derecha del BIT menos significativo se van completando con 0,

mientras que el BIT más significativo se pierde en cada desplazamiento. Veamos el siguiente ejemplo:

```
int dato_1 = 8;
int cantidad = 3;
int resultado = dato_1 << cantidad;
```

La operación que se lleva a cabo a nivel de BITs, en sucesivos desplazamientos, es la siguiente:

OPERADOR ←←			
▼ VARIABLE	▼ Nº DESPLAZAMIENTO	▼ CONTENIDO	
dato_1	0 (sin desplazar)	0000 1000	8
	1	0001 0000	16
	2	0010 0000	32
	3	0100 0000	64
Resultado	—	0000 0000	64

Tabla 5. Efecto de la aplicación del operador << sobre una variable.

La sentencia switch

Si quisiéramos manejar situaciones donde debiéramos verificar gran cantidad de condiciones diferentes, resultaría engorroso tener que escribir largos bloques de sentencias **if** y más.

Para estos casos, existe otro bloque de sentencias que permite tomar decisiones en base a los valores que adquiere una variable: **switch**. En él, se programa una determinada cantidad de opciones y una sola es seleccionada en base a una condición. La forma general del bloque de sentencia **switch** es la siguiente:

```
...
switch (valor_entero)
{
        case valor_constante_1:
                sentencia_1;
                sentencia_2;
                ...
                break;
        ...
        case valor_constante_n:
                sentencia_1;
                sentencia_2;
                ...
                break;
        default:
                sentencia_1;
                sentencia_2;
                ...
}
...
```


La comparación se realiza en torno de la variable **valor_entero**. Si este valor se corresponde con alguno de los **case** indicados, definido por su asociado **valor_constante_n**, entonces se ejecutan las sen-

tencias que le siguen inmediatamente, en forma secuencial, hasta encontrar la sentencia **break**, que indica final de ejecución del bloque **switch** y ocasiona que la ejecución del programa continúe con el código que sigue inmediatamente al cierre de llave del bloque.

Tengamos en cuenta que de no hallarse correspondencia entre **valor_entero** y los **case**, entonces se ejecutan las sentencias que siguen inmediatamente a **default**. Esta última puede obviarse y, en este caso, si no hay correspondencia, simplemente no pasa nada, es decir, nada se ejecuta.

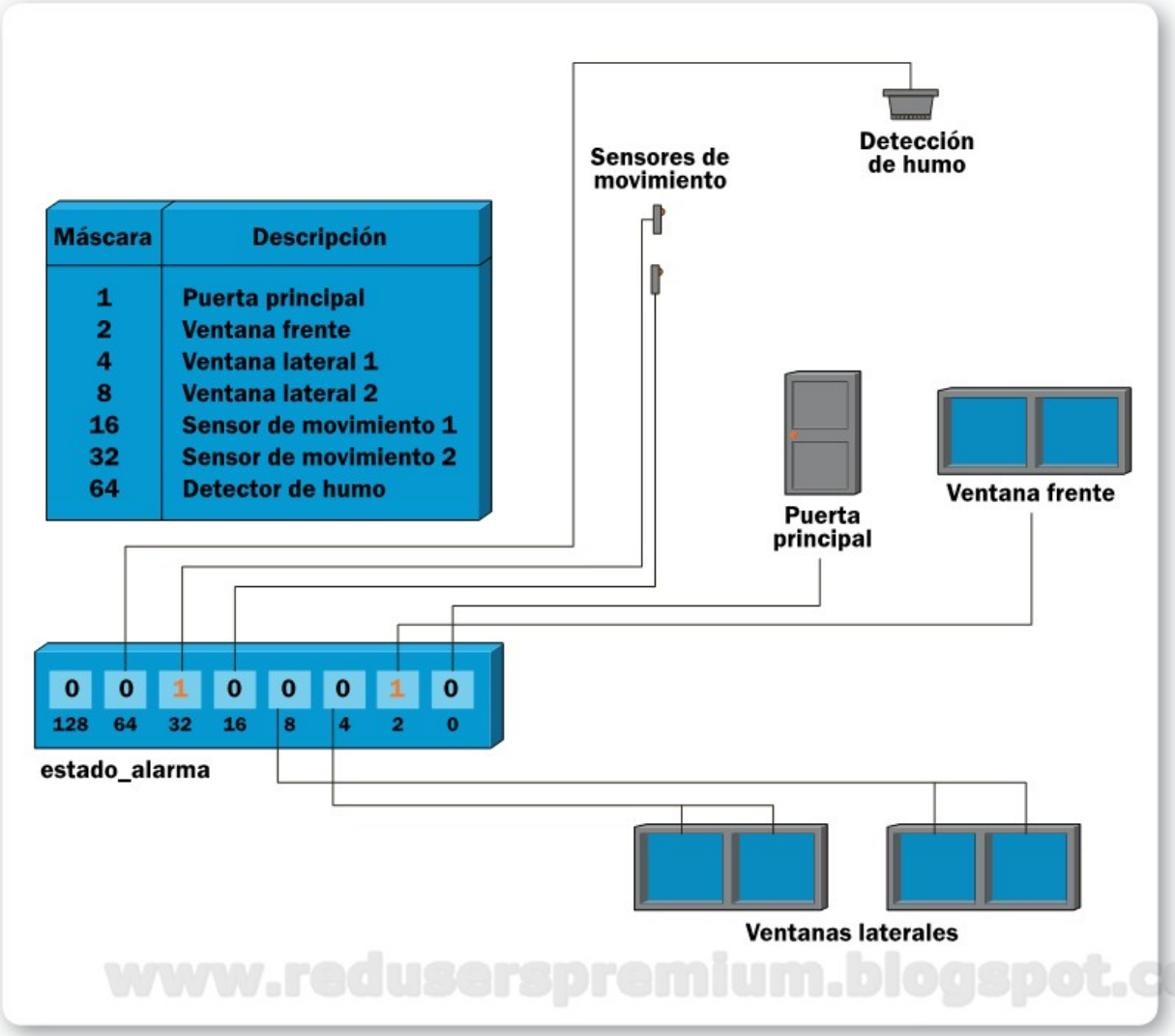


Figura 3. Esquema de bloques de la sentencia **`switch`**, que simplifica la utilización de la sentencia **`if`** cuando es necesario comparar grandes cantidades de información.

La sentencia goto

Supongamos que vamos manejando por las calles de nuestra ciudad y nos encontramos, de repente, con que el camino adelante está cerrado al paso por reparaciones que se están llevando a cabo. Es muy probable que hallemos un cartel con la inscripción **desvío**, que nos obligue a tomar una ruta alternativa para poder continuar. Análogamente a esta situación, cuando un programa se encuentra con esta sentencia, hace justamente eso, desvía el flujo de ejecución normal del programa hacia donde sea que le indique la etiqueta que sigue inmediatamente a la sentencia denominada **goto**.

Consideremos que lo hace sin realizar ninguna verificación de datos y sin preguntar al usuario si es eso lo que realmente se deseaba hacer, por lo tanto es una sentencia de cuidado.

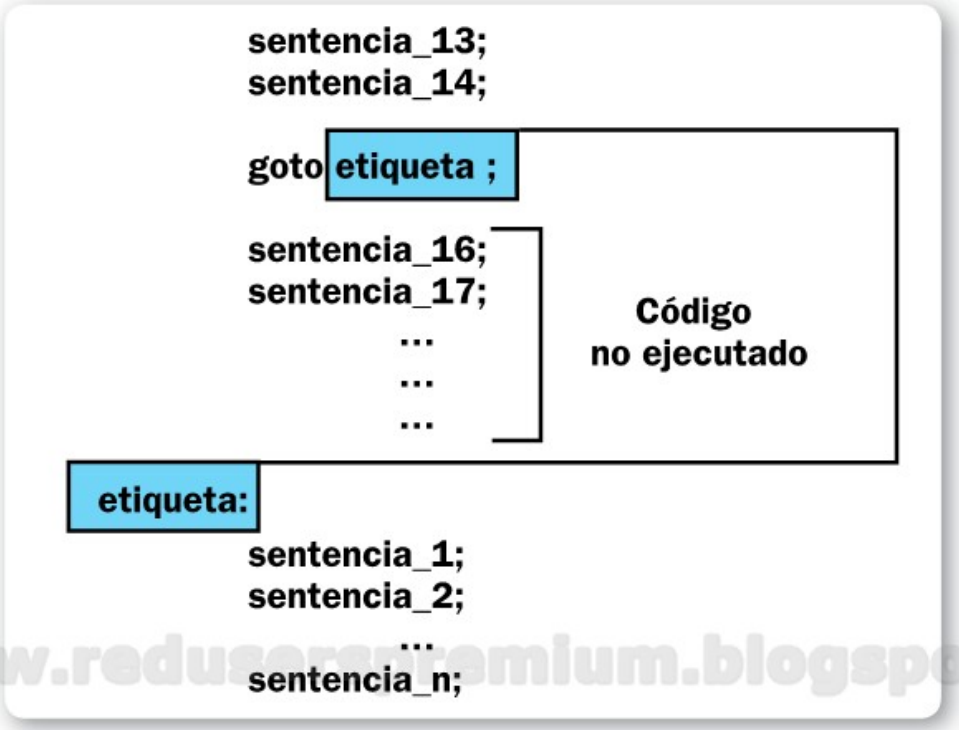


Figura 4. Estructura de la sentencia **goto**. Puede ocurrir que haya ciertos bloques de código que nunca sean ejecutados ante la inclusión de una sentencia de este tipo.

Observemos que si inmediatamente después de la sentencia **goto** hay código escrito, este **nunca será ejecutado**, a menos que esté identificado con otra etiqueta y esta sea referenciada a través de otra sentencia **goto** en algún otro lugar del programa. Esto es debido a que el programa **siempre** va a desviar la ejecución del código hacia donde lo especifique la etiqueta, cada vez que se encuentre con esta sentencia.

A pesar de que el lenguaje lo soporta, debemos tener en cuenta que la utilización de esta sentencia debe llevarse a cabo con mucho cuidado, ya que a primera vista podríamos tentarnos y emplearla repetidamente en nuestros programas. Sin embargo, su uso indiscriminado conduce a que nuestro código se torne ininteligible y desordenado.

La sentencia break

La sentencia **break** se utiliza, como hemos visto, dentro de la sentencia **switch**, para indicar el final de la ejecución de las opciones **case**. Esta sentencia también se emplea dentro de las estructuras de lazo (**bucles**), como instrumento para finalizar abruptamente su ejecución. Los bucles los veremos más adelante.

Operador sizeof

Cuando hablamos de los tipos de variables, mencionamos por ejemplo, que el tipo **int** puede ocupar 2 ó 4 BYTEs, de acuerdo con la plataforma en donde estemos ejecutando nuestro programa y/o del compilador que estemos utilizando. Pues bien, ¿cómo podemos conocer de forma precisa cuánto ocupa?

El lenguaje nos provee de una herramienta que se encarga de ello, y es el operador **sizeof**. Los siguientes ejemplos nos ilustran acerca de la forma adecuada de usarlo:

```
/*Operador sizeof        */

#include <stdio.h>
void main()
{
        printf("Las variables del tipo long double ");
printf("ocupan %i bytes\n",
sizeof(long double));
}
```

Operadores incrementales y decrementales

Estos operadores se utilizan para aumentar o disminuir en una unidad el valor de la variable entera a la cual se aplica. Se los puede pensar como **más uno** y **menos uno** para cada caso.

El operador incremental reconocido por el lenguaje se indica con el doble signo de suma (**++**), mientras que el operador decremental se señala con el doble signo de resta (**--**).

Consideremos que se los coloca inmediatamente antes de la variable a la cual serán aplicados.

PROFESOR EN LÍNEA

Si tiene alguna consulta técnica relacionada con el contenido, puede contactarse con nuestros expertos: profesor@redusers.com

Por ejemplo, si tenemos una variable llamada **numero**:

```
++numero;        /*Incrementa la variable en 1*/
--numero;        /*Decrementa la variable en 1*/
```

A medida que avancemos en la programación en C, nos daremos cuenta de que estos operadores resultan muy prácticos, especialmente cuando usamos bucles.

RESUMEN

En este capítulo, nos encontramos con muchas nuevas herramientas que hacen al soporte de la programación básica en C. Los bloques if-else y switch-case nos aportan gran flexibilidad y control a la hora de tomar decisiones, mientras que los operadores lógicos amplían las posibilidades de comparación entre variables. Ya empezamos a vislumbrar la potencialidad que nos ofrece el lenguaje, a través de su sólida estructura de programación.

*03

Bucles, arreglos y cadenas

Este capítulo nos introducirá en estructuras de programación que nos permitirán realizar iteraciones basadas en determinadas condiciones. Estas iteraciones, también llamadas bucles, son de amplia utilización en programación.

Tipos de bucle

El proceso por el cual una serie de instrucciones se ejecuta repetidamente una determinada cantidad de veces o hasta que una determinada condición se cumpla, se denomina **bucle**. Los bucles constituyen otro bloque fundamental de programación y pueden ser combinados junto con la capacidad de comparación. Esto nos habilita a repetir determinadas acciones hasta que una condición se dé.

Por ejemplo, podemos continuar ejecutando una acción en particular hasta que dos variables sean iguales. Una vez que son iguales, continuamos con otra acción diferente.

El bucle for

En su forma más simple, este bucle nos permite ejecutar un bloque de sentencias una determinada cantidad de veces, dependiendo de las expresiones que son pasadas a la sentencia.

La forma estructural del bucle **for** es la siguiente:

```
for (expresión_1; expresión_2; expresión_3)
{
        sentencia_1;
        sentencia_2;
        ...
}
```


Generalmente, encontraremos que **expresión_1** y **expresión_3** pueden ser asignaciones o llamadas a una función, mientras que **expresión_2** es una comparación.

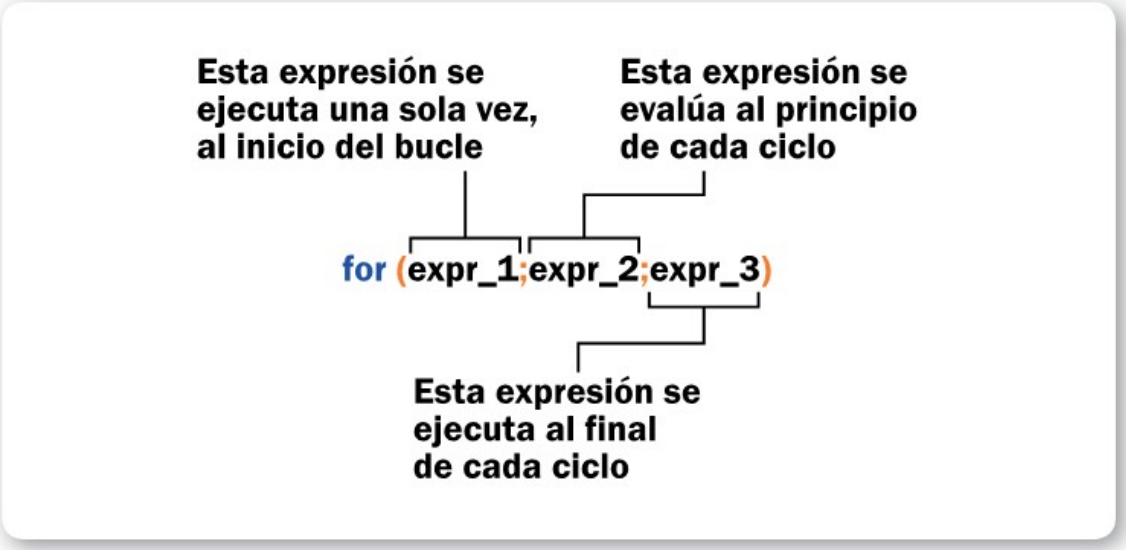


Figura 1. Forma estructural de la sentencia **for**. Permite la implementación de bucles simples.

Veremos, a continuación, algunos ejemplos que ilustran el funcionamiento de la sentencia **for**.

```
/* Sentencia for                    */

#include <stdio.h>

void main()
{
```

BUCLE INFINITO

El hecho de tener un bucle de ejecución infinito significa que es el programador quien debe proveer una salida del bucle a través de código en el bloque de sentencias. Recordemos que podemos forzar la salida de un bucle por medio de la sentencia **break** o **return**.

```
        int a,b;
        printf("TABLAS DE MULTIPLICAR\n\n");
        for(a=1;a<=10;a++)                    /*Primer bucle     */
        {
                for(b=1;b<=10;b++)            /*Segundo bucle  */
                {
                        /*Tabla de multiplicar de a*b
*/
                        printf("%ix%i=%i ",a,b,a*b);
                }
                /*Siguiente línea  */
                printf("\n");
        }
   }
```

El resultado de la ejecución de este ejemplo es la tabla de multiplicar desde el 1 hasta el 10:

El bucle while

Este bucle, al igual que **for**, nos permite ejecutar un bloque de sentencias en forma indefinida, mientras una determinada expresión sea distinta de **0** (**verdadera**). Cuando la expresión es igual a **0** (**falsa**), termina la ejecución del bucle.

La forma estructural del bucle **while** es la siguiente:


```
while (expresión)
        {
                sentencia_1;
```

```
            sentencia_2;
            ...
    }
```

Si bien el uso de **while** o **for** es meramente una cuestión de gusto personal, podríamos decir que el empleo de este último es más claro desde el punto de vista de la programación, cuando es necesario inicializar variables previo al inicio del bucle y cuando es necesario realizar incremento de variables que se utilizan dentro del bucle. Esto es debido a que la sentencia **for** mantiene la variable de control y el incremento visibles en su estructura.

El bucle do – while

Este bucle, al igual que **while**, nos permite ejecutar un bloque de sentencias en forma indefinida, mientras una determinada expresión sea **verdadera**. Cuando la expresión es **falsa**, termina la ejecución del bucle. Pero, a diferencia del **while**, primero se ejecuta el bloque de sentencias y luego se evalúa la expresión. Es decir que el bloque de sentencias se ejecuta, al menos, una vez. La forma estructural del bucle **do – while** es la siguiente:

```
{
    sentencia_1;
    sentencia_2
    ...
} while (expresión);
```

Arreglos

Un arreglo no es ni más ni menos que una colección de datos del mismo tipo, representados por una única variable a la cual se le asigna un nombre. A cada uno de los datos que constituyen el arreglo se lo denomina **elemento** y se lo puede diferenciar unívocamente a través de su posición, identificándolo con un número o un índice.La declaración general de un arreglo es la siguiente:

```
tipo nombre_variable[cantidad_datos];
```

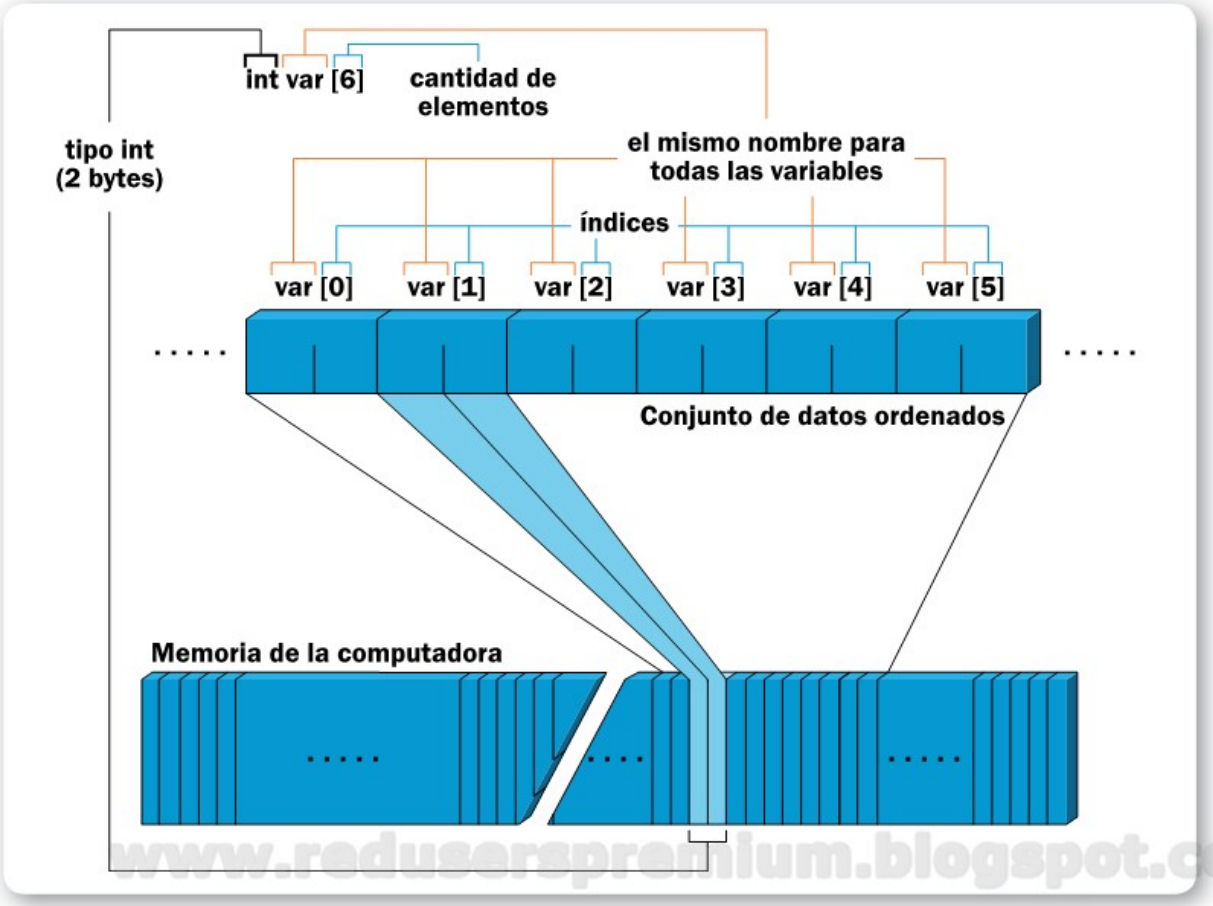


Figura 2. Declaración de un arreglo. Distribución en memoria y cantidad de elementos. Vemos que su máximo índice es igual a la cantidad de elementos menos uno.

Como un arreglo es un conjunto ordenado de variables del mismo tipo, la cantidad de memoria que se reserva cuando se lo declara está dada por la longitud, en BYTEs, del tipo de variable que lo define, multiplicada por el parámetro **cantidad_datos**.

Representación hexadecimal

La representación **hexadecimal** (o **hexa**) es una forma conveniente de representar números binarios de 4 dígitos (también llamados **nibbles**) o valores decimales en el rango 0 a 15, que es el mismo rango para un solo dígito hexadecimal, de 0 a F. De esta forma, si un dígito hexadecimal representa 4 dígitos binarios, 2 dígitos hexadecimales conforman un BYTE. Esta manera de representación es ampliamente usada en programación.

Representación de caracteres

¿Existe alguna manera a través de la cual podamos almacenar caracteres y otros símbolos utilizados en los alfabetos? La respuesta es no. No obstante, tenemos la posibilidad de almacenar números a los cuales se les asigna un carácter por convención y que, al mismo tiempo, puede ser almacenado en un simple BYTE de memoria. Es así que se ha establecido el código estándar

SINTAXIS DE SENTENCIAS

Algo para tener muy presente cuando programamos es la sintaxis de las sentencias que estamos utilizando. Mientras que las sentencias **for** y **while** no llevan punto y coma al final, el bucle **do – while** sí lo requiere. Es necesario considerar la forma estructural de las sentencias ya que, de lo contrario, el lenguaje interpretará de manera errónea las sentencias.

americano para el intercambio de información (por sus siglas en inglés, **A**merican **S**tandard **C**ode for **I**nformation **I**nterchange), más ampliamente conocido como código **ASCII**. De acuerdo con este código, la letra **A** se representa con el número 65 decimal o 41 hexa, mientras que la letra **G** se señala con el número 71 decimal o 47 hexa, y el símbolo ~ se indica con el número 126 decimal o 7E hexa.

Arreglos y direcciones

Cuando introdujimos el concepto de arreglos, vimos un ejemplo de programa que nos mostraba en pantalla la cantidad de memoria que se reserva en la declaración del arreglo y la dirección que le fue asignada cuando fue creado. Esta dirección de memoria, a la que llamaremos **dirección de memoria base del arreglo**, corresponde al primer elemento del arreglo. Si guardamos un dato en ella, estamos definiendo el valor de su primer elemento. Pero, ¿cómo hacemos si queremos conocer la dirección de memoria del n - ésimo elemento del arreglo para guardar un dato en él? Pues bien, a partir de la dirección de memoria base del elemento, debemos sumarle un offset que resulta de multiplicar el índice del n - ésimo elemento por la cantidad de BYTEs requerida.

Inicialización de arreglos

De la misma forma que inicializamos variables cuando las declaramos, muy a menudo resulta necesario inicializar los arreglos declarados con valores iniciales para cada uno de los elementos que los constituyen. La forma de inicializarlos es, simplemente, encerrar entre llaves toda la lista de los valores iniciales separados por comas.


```
int valores[6] = {17,7,71,24,12,76}
```

Hemos declarado un arreglo del tipo **int** con 6 elementos a los cuales se los ha inicializado con la lista de valores encerrada entre llaves. Si se especifican menos valores de inicialización que la cantidad de elementos que constituyen el arreglo, entonces a los elementos que no han sido inicializados se les da el valor **0**.

Pero debemos tener en cuenta que, por el contrario, si en la lista existen más valores de inicialización que elementos en el arreglo correspondiente, entonces el compilador nos dará un mensaje de error al compilar el programa.

Arreglos multidimensionales

Hasta aquí, hemos visto arreglos en los cuales especificamos la cantidad de elementos encerrados entre corchetes. Estrictamente, este tipo de arreglo se denomina **unidimensional** y es un caso especial de los arreglos **multidimensionales**.

Consideremos que un arreglo multidimensional es conocido, generalmente, como una **matriz de elementos**, y podemos representarla si ordenamos los elementos en filas y columnas. Así, la declaración de un arreglo multidimensional se realiza de la siguiente manera:

```
tipo nombre_arreglo [filas][columnas];
```

ATENCIÓN CON LA MEMORIA

Si bien los arreglos son convenientes para el manejo de gran cantidad de información con pocas variables, también son grandes consumidores de memoria, por lo que debemos prestar especial cuidado cuando programamos y utilizar solamente los que resultan indispensables.

Cadenas

Una cadena es una secuencia ordenada de caracteres o símbolos encerrados entre comillas. Ya las hemos usado cuando empleamos la función **printf()**. Las cadenas de texto terminan con el **carácter nulo** (**\0**), que se agrega en forma automática para señalizar su final.

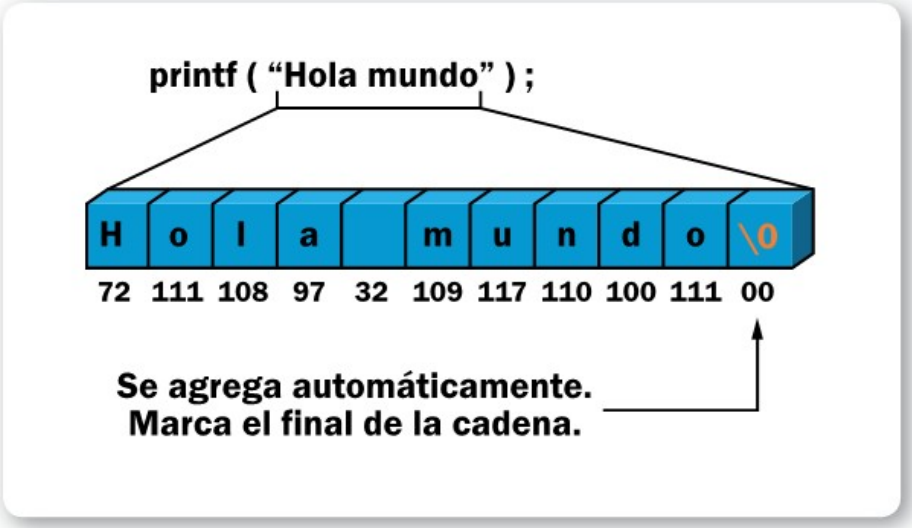


Figura 3. Forma de representación de una cadena de texto en memoria. El final de cadena lo marca el carácter nulo (**\0**), que se agrega automáticamente.

Métodos de manejo de cadenas y texto

Declaramos una cadena de texto a través de un arreglo del tipo **char**, de la siguiente forma:

```
char cadena[] = "Hola mundo";
```


El compilador reserva la memoria necesaria para poder almacenar la cadena indicada. Aquí, se encarga de reservar once posiciones de memoria para el arreglo.

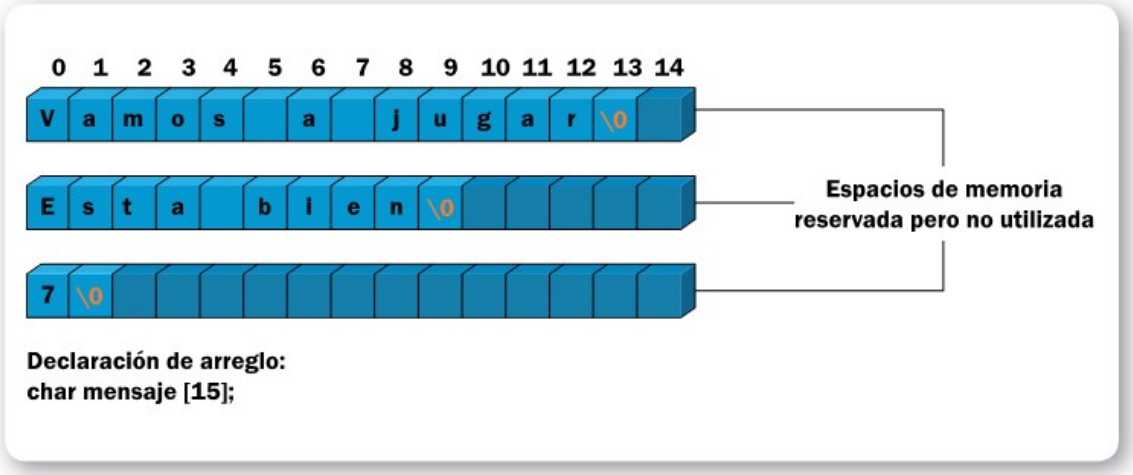


Figura 4. Con cadenas de longitud variable, arreglos del tipo **char** resultan ineficientes en el uso de memoria.

Librería de funciones de cadena

Hemos visto cómo llevar a cabo distintos tipos de operaciones con cadenas y con arreglos de cadenas, y hemos realizado ejercicios demostrando cómo manejar cadenas.

Los programas presentados tienen cierta complejidad para el usuario que recién se inicia en la programación, pero de todas formas su funcionamiento es fácil de comprender si logramos concentrarnos en lo que el programa está haciendo.

Copia de cadenas

La librería **string.h** incluye una función que realiza la copia de una cadena de texto en otra con una sintaxis muy clara:

```
strcpy (cadena1, cadena2);
```

Los argumentos **cadena1** y **cadena2** son arreglos de cadena del tipo **char**. Lo que hace esta función es, simplemente, copiar **cadena2** en **cadena1**, incluyendo el marcador de final de cadena (**\0**). La función no realiza un control previo para verificar que **cadena1** tenga suficiente espacio para alojar a **cadena2**, por lo que es nuestra responsabilidad efectuarlo. Tampoco lo hace sobre los tamaños de los arreglos.

Unión de cadenas

La librería de cadenas incluye una función que realiza esta misma función, y la manera de utilizarla es la siguiente:

```
strcat (cadena1, cadena2);
```

La función copia **cadena2** al final de **cadena1**, y devuelve **cadena1** como resultado, conteniendo la unión de ambas cadenas.

RESUMEN

En este capítulo, nos hemos encontrado con muchas herramientas que nos permiten implementar programas que realicen iteraciones, trabajar con arreglos y matrices, y manejar texto sin mayores inconvenientes. Asimismo, hemos introducido nuevas librerías que aportan considerable funcionalidad y nos ahorran mucho tiempo cuando debemos trabajar con cadenas de texto. El capítulo siguiente presenta una herramienta muy poderosa del lenguaje, los punteros, que nos brindarán aun más flexibilidad a la hora de trabajar con cadenas.

*04

Punteros

En la programación en C los punteros constituyen una de las herramientas más poderosas. Al mismo tiempo, puede ser también una de las más confusas. Entender la noción de punteros implica tener una sólida base en la idea de variables. En este capítulo, introduciremos las pautas iniciales para lograr su comprensión, que involucra muchos nuevos conceptos.

Introducción a los punteros

Cuando el programa es compilado y vinculado, el nombre de la variable se reemplaza directamente por la posición de memoria correspondiente. Por ejemplo, observemos la siguiente sentencia:

```
int numero = 10;
```

Ya sabemos que, al hacer esto, estamos reservando dos BYTEs en memoria y que, en la dirección de memoria que le fue asignada a la variable cuando fue creada, almacenamos el valor **10** en decimal. En C, existe un tipo de variable especial, que está diseñada para almacenar direcciones de memoria. Estas variables especiales se denominan **punteros**, y las direcciones de memoria que almacenan son, generalmente, las de otras variables. Un aspecto importante a tener en cuenta es que debemos conocer el tipo de variable a la cual se destina el puntero.

Un puntero será asociado a un determinado tipo de variable, y puede ser utilizado solamente para apuntar a variables de ese tipo. Punteros del tipo **int** apuntan a variables del tipo **int**, punteros del tipo **float** apuntan a variables del tipo **float**.

¿A QUÉ TIPO DE DATO APUNTA?

Es necesario conocer el tipo de dato al que apunta. Si este es desconocido, entonces es prácticamente imposible manejar el contenido de memoria de lo que apunta. Por ejemplo, un puntero que apunta a un valor del tipo **char**, lo hace a un valor que ocupa 1 BYTE, mientras que uno que apunta a un valor del tipo **long**, lo hace a un valor que ocupa 4 BYTEs.

Declaración de punteros

Veremos, aquí, la forma de declaración de punteros, pero para hacerlo más práctico y entendible, lo haremos a través de un ejemplo:

```
/* Declaración de punteros        */

#include <stdio.h>

void main()
{
        int var=0;          /*Variable tipo int*/
        int *p=NULL;        /*Variable tipo puntero a int*/

        var=17;                         /*Asignación de variable*/
        printf("\nDirección de variable: %p",&var);
        printf("\nValor de la variable: %i",var);

        p=&var; /*Guardo la dirección de la variable*/
        printf("\n\nValor del puntero: %p",p);
        printf("\nValor al que apunta: %i",*p);
}
```

Las dos primeras líneas de código dentro de la estructura **main** declaran variables. La primera de ellas declara una variable llamada **var**, del tipo **int**. La segunda línea declara una variable ***p**. Utilizamos el asterisco delante del nombre de la variable para indicar al compilador que lo que estamos declarando es un puntero que apunta a un tipo de variable del tipo **int**.

Como podemos observar, en la declaración del puntero empleamos la macro **NULL**, que está incluida en el lenguaje y que es el equivalente al carácter nulo, pero que se aplica solamente a punteros. Simplemente, especifica que el puntero no apunta a nada.

Ahora, ¿cómo se logra que el puntero apunte a la variable? Como los punteros guardan direcciones de memoria, hacemos uso del operador **address** o **&**. La sentencia:

```
p=&var;
```

logra lo que estamos buscando. El puntero **p** apunta a la dirección (**&**) de la variable **var**. ¿Y cómo logramos acceder al contenido de memoria al que apunta un puntero? Observemos la última sentencia del ejemplo:

```
printf("\nValor al que apunta: %i",*p);
```

A través del **operador de indirección** u **operador de referencia** (*), logramos alcanzar el contenido de lo que apunta el puntero **p**. Esto es exactamente igual que lo que hicimos unas líneas arriba en el ejemplo:


```
printf("\nValor de la variable: %i",var);
```

El resultado es el mismo, por un lado, invocando explícitamente el nombre de la variable (forma directa) y, por otro, usando un puntero (forma indirecta).

Utilización de punteros

Ya tenemos un puntero que apunta a una variable, pero nos podríamos preguntar si esta es una forma complicada de usar el contenido de una variable. Mucho más simple resulta emplear las variables por su nombre. Entonces, ¿cuál es realmente la ventaja de utilizar punteros? En el ejemplo anterior, si es posible acceder al contenido de la variable **var** a través del puntero **p**, entonces:

```
*p += 10;
```

incrementa el contenido de la variable **var** en 10 unidades. Ahora, como **p** puede apuntar a cualquier variable del tipo **int**, nada nos impide hacer:

```
p=&var2;                    /*Otra variable del tipo int */
```

Es decir, **p** deja de apuntar a **var** y pasa a hacerlo a una nueva variable del tipo **int**, llamada **var2**. Si usamos la misma sentencia que empleamos con **var**:


```
*p+=10;
```

Hemos presentado, hasta aquí, muchos conceptos relacionados con los punteros y la forma en que pueden ser usados.

Por esta razón es recomendable plantear un ejemplo antes de continuar, de esta forma podremos fijar mejor los nuevos conocimientos que hemos adquirido.

```
/* Uso de punteros 1     */

#include <stdio.h>

void main()
{
  /*Inicialización de variables     */
  int a=0;                    /*Contador de bucle*/
  int var[10]={2,6,4,8,3,12,32,54,25,77}
  int *p=NULL;                /*Puntero inicializado a cero*/

  do
  {
    p=&var[a]; /*Trabajo con la a-esima
                                        ariable*/          v
    printf("\nVariable var[%i]=%i\tDirección:");
    printf(" %p",a,var[a],&var[a]);
    printf("\nPuntero p=%p\t\tDirección: %p\t");
    printf("\tContenido al que apunta: %i");
    printf("\n",p,&p,*p);
    a++;          /*Incremento el contador*/
  }while(a<10);
}
```

Las primeras líneas de código dentro del bloque **main** se encargan de declarar e inicializar una variable, un arreglo, y el puntero que vamos a utilizar. Ya dentro del bucle **do**, lo primero que encontramos es la asignación del puntero a **int p**, con la dirección del primer elemento del arreglo. Luego, imprimimos en pantalla el contenido del a-ésimo elemento del arreglo **var** y su dirección. Observemos que las direcciones de memoria reservada para el arreglo son contiguas y ocupan 2 BYTEs. Esto es así porque el arreglo es del tipo **int**. Esta es la forma directa de trabajar con las variables. La forma indirecta es a través del uso del puntero **p**. En efecto, en las líneas subsiguientes vemos cómo **p** se va asignando con las direcciones de memoria donde están alojados cada uno de los elementos del arreglo y cómo obtener su contenido.

```
/* Uso de punteros con scanf()   */

#include <stdio.h>

void main()

{
        /*Inicialización de variables          */
```

INTERCAMBIABILIDAD

Los punteros pueden asumir el aspecto de cualquier variable del mismo tipo. Esto nos habilita a utilizar un solo puntero para cambiar el valor de distintas variables. La única condición es que todas estas variables deben ser del mismo tipo.

```
        int a=0;              /*Contador de bucle*/
        int *p=NULL;          /*Puntero inicializado a cero*/

        p=&a;     /*El puntero apunta a la variable a*/

        printf("\nIngrese un número entero: ");
        scanf("%i",p);        /*Guardo el valor ingresado
                                        en  orma  ndirecta*/

        printf("\n\nEl valor ingresado es: %i",a);
}
```

La constante NULL

La inicialización del puntero en los ejemplos anteriores es:

```
int *p=NULL;
```

Aquí, **NULL** es una constante especial en C y es el equivalente al carácter nulo utilizado para los números ordinarios, pero aplicado a los punteros. Su definición está incluida en el archivo de cabecera **stdio.h**. Su significado se interpreta simplemente como que el puntero inicialmente no apunta a nada. Se garantiza que su contenido será distinto de cualquier valor de dirección real que pueda ocurrir.

Nomenclatura

A medida que nuestros programas se hacen cada vez más extensos, resultará cada vez más difícil recordar qué variables son comunes y cuáles son punteros. En este sentido, se considera buena

práctica para la programación diferenciar a los punteros de las demás variables, anteponiendo un prefijo tal como **p_**. Por ejemplo, podemos tener una variable **numero** y un puntero a ella **p_numero**.

Si empleamos esta nomenclatura, no tendremos dudas entre cuál es una variable común y cuál es un puntero.

Arreglos y punteros

Podemos utilizar un solo puntero para acceder a diferentes variables, siempre y cuando estas sean del mismo tipo. Los arreglos y punteros son muy semejantes, a tal punto que en ocasiones pueden ser intercambiados. Como vimos, una cadena de texto es un arreglo del tipo **char**. Si lo que buscamos es ingresar un único carácter en una variable con **scanf()**, simplemente hacemos uso del operador **address** o **&**:

```
char caracter;
scanf("%s",&caracter);
```

Pero si lo que buscamos es ingresar una cadena de texto, lo hacemos con un arreglo:

```
char cadena[20];
scanf("%s",cadena);
```


Observemos que, en esta ocasión, no hemos usado el operador **&**. Es decir, estamos empleando el nombre del arreglo **como si fuera** un puntero. Utilizando el nombre de esta forma, sin indi-

cación de un valor de índice, entonces se refiere a la dirección de memoria del primer elemento en el arreglo.

Veremos, aquí, algunos ejemplos de los conceptos presentados. Ingresemos el siguiente programa en nuestro editor y luego procedamos a ejecutarlo:

```
/* Arreglos y Punteros 1          */

#include <stdio.h>

void main()
{
        char arreglo[]="El lenguaje C es apasionante";

        printf("\nUsando el operador address of: %p"
                ,&arreglo[0]);
        printf("\nSin usar el operador: %p",arreglo);
}
```

Podemos concluir que **&arreglo[0]=arreglo**. Ahora bien, ¿qué pasaría si usáramos el segundo índice en lugar de la primera posición?

¿DÓNDE ESTÁ LA DIFERENCIA?

Si bien expusimos que podemos usar un arreglo **como si fuera** un puntero dentro de la función **scanf()**, debemos tener siempre presente que un arreglo **no es** un puntero. Existe una gran diferencia: podemos cambiar la dirección guardada en un puntero, como vimos en los ejemplos de este capítulo, pero no podemos cambiar la dirección a la que hace referencia el nombre de un arreglo.

```
/* Arreglos y Punteros 2          */

#include <stdio.h>

void main()
{
        char arreglo[]="El lenguaje C es apasionante";

        printf("\nValor del segundo elemento : %c"
                ,arreglo[1]);
        printf("\nValor del arreglo sumando 1: %c\n"
                ,*(arreglo+1));
}
```

Observamos, aquí, que **arreglo[1]=*(arreglo+1)**. Demos un paso más adelante y veamos el siguiente ejemplo:

```
/* Arreglos y Punteros 3          */

#include <stdio.h>

void main()
{
        long arreglo[]={16,32,64};
```



```
        printf("\nDirección del primer elemento : %p"
                "\tValor: %d\n",arreglo,*arreglo);
        printf("\nDirección del segundo elemento: %p"
```

```
                "\tValor: %d\n",arreglo+1,*(arreglo+1));
        printf("\nDirección del tercer elemento : %p"
                "\tValor: %d\n",arreglo+2,*(arreglo+2));
}
```

El resultado puede parecernos equivocado en un primer momento. La dirección del primer elemento del arreglo es **FFD0** (este valor depende del sistema donde se esté ejecutando y del compilador usado). Cuando sumamos 1 unidad al arreglo para pasar al siguiente elemento, obtenemos **FFD4**, y si sumamos otra unidad más obtenemos **FFD8**. Sin embargo, el resultado es correcto ya que, en verdad, el compilador entiende que lo que queremos hacer es acceder al siguiente elemento del arreglo, y el salto de memoria entre cada uno de los elementos del arreglo es de 4 BYTEs en correspondencia con el tipo de datos (**long** = 4 BYTEs). Este es uno de los motivos por el cual, cuando se declara un puntero, es absolutamente necesario especificar el tipo de variable a la que apunta.

Arreglos multidimensionales y punteros

Veremos, aquí, cómo trabajar con arreglos multidimensionales. En primer lugar, desarrollaremos, a través de un ejemplo, cómo el compilador maneja las direcciones de memoria relacionadas con el arreglo:


```
/* Arreglos multidimensionales 1*/
```

```
#include <stdio.h>

void main()
{
        char tabla[3][3]={
                                                {'1','2','3'},
                                                {'4','5','6'},
                                                {'7','8','9'}
                                        };

        printf("\nDirección de tabla:\t\t%p\n",tabla);
        printf("Dirección de tabla[0][0]:\t%p\n",
                &tabla[0][0]);
        printf("Contenido de tabla[0]:\t\t%p\n",
                tabla[0]);
}
```

Los tres valores resultantes son idénticos.

```
tabla
tabla[0]
&tabla[0][0]
```

PROFESOR EN LÍNEA

Si tiene alguna consulta técnica relacionada con el contenido, puede contactarse con nuestros expertos: profesor@redusers.com

Es decir que, producen el mismo resultado. Por lo tanto, son expresiones equivalentes. Ahora, ¿qué conclusión podemos sacar de esto? Cuando declaramos un arreglo unidimensional y colocamos el índice a continuación del nombre del arreglo, estamos diciendo al compilador que es un arreglo del tamaño indicado por el índice. Si a este arreglo le agregamos otro índice a continuación del primero, le estamos diciendo al compilador que queremos hacer un arreglo del tamaño indicado por el primer índice y en donde cada elemento es un arreglo del tamaño indicado por el segundo índice. Es decir, cuando declaramos un arreglo multidimensional estamos creando un arreglo de sub-arreglos.

Por lo tanto, cuando accedemos al arreglo multidimensional utilizando solamente el primer índice (**tabla[0]**, por ejemplo), en realidad estamos haciendo referencia a la dirección del primer sub-arreglo. Cuando usamos el nombre del arreglo por sí solo (por ejemplo, **tabla**), estamos haciendo referencia a la dirección del comienzo del arreglo de arreglos, que coincide con la dirección del principio del primer sub-arreglo. Ahora, ¿cómo accedemos al contenido del arreglo de arreglos empleando notación de punteros? Debemos recurrir al operador de **indirección**.

RESUMEN

Este ha sido un capítulo en donde hemos puesto de manifiesto muchos conceptos fuertes del lenguaje. Los punteros son una base fundamental para poder escribir código eficiente. Por eso, resulta necesario haber entendido muy bien los conceptos expuestos para poder continuar. Si este no es el caso, sugerimos volver atrás y retomar cada uno de los temas que han sido abordados, con mayor detenimiento.

*05

Estructuración de programas

A medida que avanzamos cada vez más en la programación en C y nuestros programas comienzan a tener una longitud razonablemente larga, es hora de pensar en fraccionarlo y de tener módulos de código que desarrollen tareas específicas.

Estructura de programa

Todos los programas en C están constituidos por una o más funciones que desarrollan distintas tareas. La función más importante y que nunca puede faltar, ya que es por donde comienza a ejecutarse el programa, es **main()**. También vimos cómo se utilizan otras funciones tales como **printf()**, dentro de nuestros programas. Es decir, funciones que llaman a otras funciones. Cada una de estas funciones se desempeña como una sola unidad. Cuando alguna de ellas es invocada, su código asociado se ejecuta. Luego, el control se retorna hacia el punto en donde la función ha sido llamada.

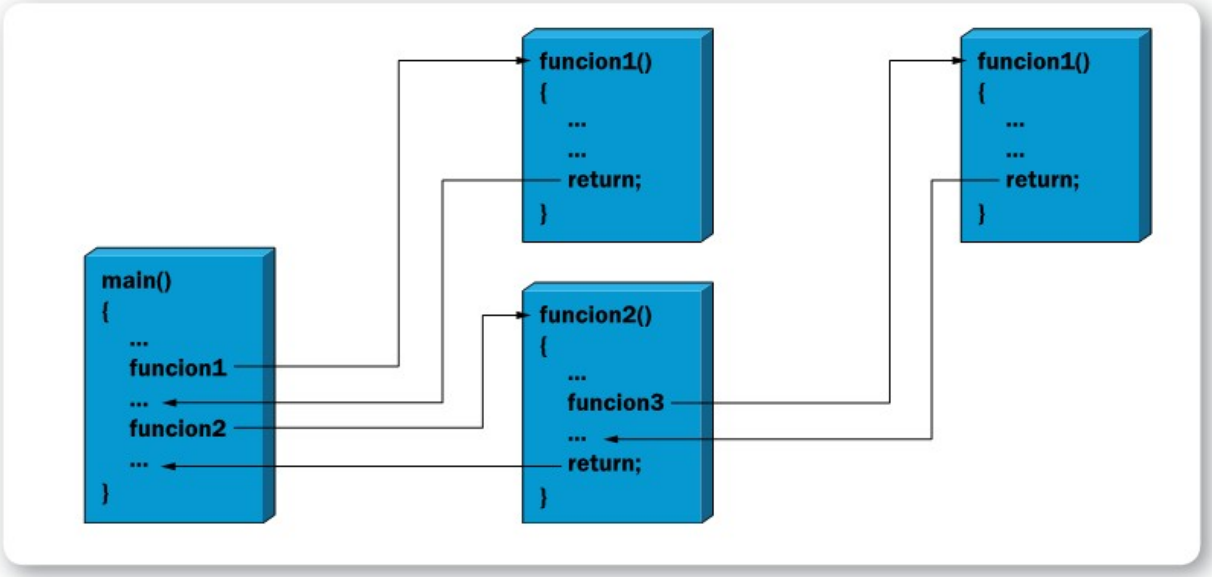


Figura 1. Secuencia de ejecución de funciones en un programa C. Observamos que, luego de la ejecución de la función, el control se devuelve al punto desde donde esta ha sido llamada.

Alcance de variables

Debemos tener en cuenta que la extensión dentro de un programa, donde una determinada variable existe, se conoce como

su **alcance**. De esta forma, las variables automáticas que han sido declaradas dentro de una función son locales de ella y no pueden existir en otra parte. Esto significa que las variables que son declaradas dentro de una función son independientes de aquellas declaradas dentro de otras funciones, lo cual nos habilita a utilizar el mismo nombre en variables de distintas funciones, siempre y cuando esto no nos confunda.

Definición de funciones

Para definir una función, en primer lugar debemos especificar su **encabezado**. A continuación, se coloca su **cuerpo**, encerrado entre corchetes. El encabezado define el nombre que se le dará, sus parámetros (si hay más de uno, estos estarán separados por comas) y el tipo de variable que retornará. El cuerpo contiene las operaciones que serán llevadas a cabo, con los parámetros que le son pasados en su encabezado. La forma general de definición de una función es la siguiente:

```
tipo_devuelto Nombre_funcion (parametros)
{
        ...
        sentencias;
        ...
}
```


Todas las funciones están programadas para hacer algo, y ese algo depende de las sentencias que figuran dentro su cuerpo. Sin embargo, no es imprescindible que el cuerpo de la función

contenga sentencias. Puede estar vacío. En este caso, la función simplemente no hará nada, y tampoco retornará un valor. Por este motivo, **tipo_devuelto** debe ser **void**.

La forma general de llamada a una función es la siguiente:

```
Nombre_funcion (parametros);
```

Los parámetros, si hay más de uno, deben estar separados por comas. La llamada a una función debe finalizar con un punto y coma.

Nomenclatura

El nombre de la función que se especifica en el encabezado durante su definición puede ser cualquiera, excepto las palabras reservadas y los nombres de aquellas funciones que se definen en las librerías de programa. Los nombres, preferentemente, deben hacer alusión a la tarea que desarrollan. Por ejemplo, a una función que calcula la raíz cúbica de un número sería conveniente llamarla con el nombre **raiz_cubica** o **r_cubica**.

Parámetros

Los **parámetros** que se pasan a las funciones se especifican en el encabezado, a continuación del nombre de la función, encerrados entre paréntesis. No son otra cosa que una lista de variables con su respectivo tipo, separadas por comas, y el medio a través del cual las funciones obtienen datos desde el programa cuando son llamadas. Los valores que se le suministran a la función mediante los parámetros se denominan **argumentos**. Estos serán posteriormente utilizados por las sentencias escritas

en el cuerpo de la función, es decir, trabajarán con ellos para desarrollar la tarea para la cual está programada y la función, finalmente, podrá o no devolver un valor al programa cuando finalice su ejecución.

Cuando pasamos parámetros a una función, automáticamente se realiza una copia en una zona de memoria determinada y la función trabaja con esta copia.

Por lo tanto, no se corre el riesgo de perder los valores originales con los que se la ha llamado. Por ejemplo, un encabezado de una función puede ser el siguiente:

```
int NombreJugador (char* jugador, int puntaje);
```

La función tiene el nombre **NombreJugador** y se le pasan dos parámetros: el primero, del tipo puntero, a **char** y, el segundo, del tipo **int**. Finalmente, la función devuelve un valor del tipo **int**.

El llamado **NombreJugador** se realiza haciendo referencia a su nombre en algún lugar del programa, por esta razón debemos tener en cuenta que es necesario especificar los argumentos, que deben coincidir exactamente en tipo, cantidad y secuencia. Debemos tener presente que cuando se lleva a cabo el llamado a la función, estos argumentos se sustituyen en el orden y secuencia especificada en su definición.

CUERPO DE UNA FUNCIÓN

Las sentencias que figuran dentro del cuerpo de una función pueden contener bloques de sentencias anidadas. Sin embargo, no es posible definir funciones dentro del cuerpo de otra.

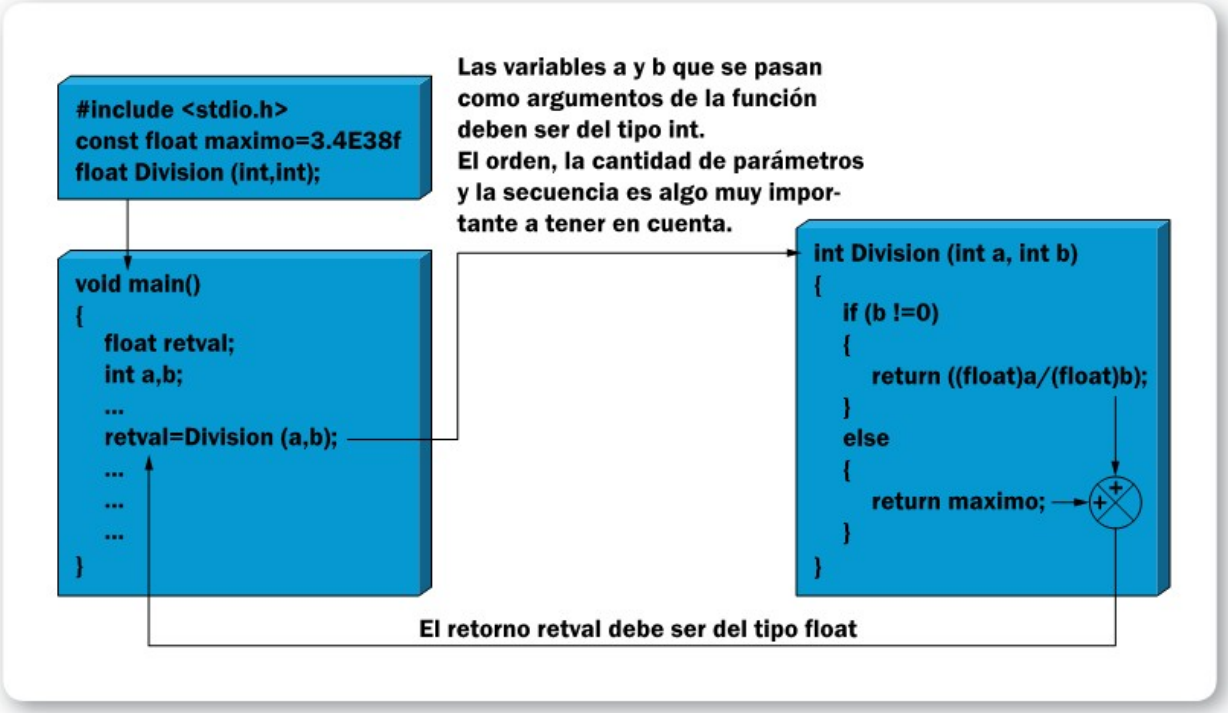


Figura 2. Proceso de llamada a una función, pasando argumentos en orden, cantidad y secuencia esperada por esta. Los tipos de variable que se pasan también deben ser los esperados por la función.

Especificación de los tipos de retorno

Cuando estudiamos la definición de funciones vimos que especifica que esta puede retornar un valor. Si la función se utiliza dentro de una expresión o está ubicada a la derecha del signo igual en una asignación, entonces se evalúa y su valor de retorno se reemplaza por la función en esa posición. El tipo de dato que retorna puede ser cualquiera de los que emplea el lenguaje (**int**, **short**, **long**, **double**, etcétera). También puede devolver un **void**, que significa que la función no devuelve valores. En este último caso, no tiene la posibilidad de ser usada dentro de una expresión o asignación.

Además, puede devolver un puntero a **void**, que es un puntero a un valor sin especificar su tipo. Ya hemos visto este tipo de devoluciones cuando describimos la función **malloc()**, que devuelve un puntero a **void**, de manera que podamos asignarle el tipo de variable a través de un ajuste de variable (**casting**).

La manera en que una función devuelve valores al programa que la ha llamado es a través de la palabra reservada **return**. La forma general de utilización de esta sentencia es:

```
return expresion;
```

Cuando **expresion** está presente, el tipo de valor retornado debe coincidir con el que fue declarado en el prototipo de la función. Puede darse el caso de omitir **expresion** y dejar solamente la sentencia **return**. Esto sucede en funciones que no devuelven valores y que han sido definidas con **void**.

En el párrafo anterior hablamos del prototipo de una función. Este simplemente indica al compilador que, en nuestro programa, estaremos definiendo una función que usará un determinado tipo de variable para sus parámetros y para el dato que devuelve. No es necesario, en esta instancia, especificar el nombre de los parámetros que utilizará la función. Declaramos el prototipo de la función **Division()** como sigue:

```
float Division (int, int);
```

SENTENCIA RETURN

Si tratamos de compilar un programa que contiene una función que quiere devolver un valor, obtendremos un error cuando esta ha sido declarada con devolución tipo **void**. De la misma forma, obtendremos un error si utilizamos la sentencia **return** sola, sin especificar la expresión devuelta, cuando la función ha sido declarada con una devolución distinta de **void**.

En esta instancia, simplemente declaramos el nombre de la función, los tipos de variable que se pasarán como parámetros a esta y el tipo de variable que devuelve. Su única tarea es dar las especificaciones esenciales de la función.

Declaración de funciones

La declaración de funciones define las características esenciales de una función. Cuando declaramos funciones, estamos especificando su prototipo. Este indica al compilador la forma en que la función es utilizada y los parámetros que espera (tipo de datos y secuencia).

Cuando, al inicio de un programa, incluimos los archivos cabecera (**.h**), estos agregan los prototipos de función especificados en las librerías de programa. Por ejemplo, hemos visto que el archivo **stdio.h** contiene los prototipos de las funciones **printf()** y **scanf()**, entre otras. Cuando lo incluimos en nuestros programas, no necesitamos declarar las funciones contenidas en la librería, pues en ella ya están incorporadas las declaraciones de prototipos y nos permite utilizar, así, las funciones directamente.

DECLARACIÓN DE PROTOTIPOS

Cuando las funciones están ubicadas dentro de un programa antes de la función **main()**, no es necesario realizar la declaración de su prototipo, pues el programa, al ejecutarse, va encontrando todas las funciones antes de llegar a **main()** y ya conoce todo sobre ellas. Ahora, si están ubicadas después de **main()**, entonces sí es necesario declarar el prototipo. Sin embargo, es buena práctica llevar a cabo la declaración del prototipo sin importar su ubicación dentro del programa.

Punteros y funciones

Si queremos pasar una o más variables a una función para que esta pueda eventualmente modificarla, la única manera de llevarlo a cabo es a través de punteros. Lo que debemos hacer es pasar, como parámetro de la función, la dirección que fue asignada a la variable cuando esta fue creada. Luego, la función podrá acceder a la variable mediante su dirección y operar sobre ella. Finalmente, podrá guardar un nuevo valor en esta dirección.

Veamos un ejemplo de cómo pasar punteros a funciones:

```
/* Punteros a funciones 1          */

#include <stdio.h>
void promedio(float*, int,int);      /*Prototipo*/

void main()
{
        int a,b;
        float result=0;      /*Resultado del promedio*/
        float* pRes=&result;         /*Dirección*/

        printf("\nIngrese un número entero: ");
        scanf("%i",a);
        printf("\Ingrese otro número entero: ");
        scanf("%i",b);
        promedio(pRes,a,b);
        printf("\n\nEl promedio entre %i y %i es %f\n"
                ,a,b,result);
}
```

```
void promedio(float* r, int a, int b)
{
        /*Calculo el promedio y el resultado lo
        guardo donde apunta el puntero r                    */
        *r = (float)((a+b)/2);
        return;
}
```

Retorno de punteros de una función

Nos podríamos plantear si es posible que una función retorne un puntero en lugar de un dato numérico. La respuesta es afirmativa y aquí veremos cómo es el proceso. Simplemente, cuando declaramos el prototipo de la función, especificamos que el valor devuelto por esta es un puntero. Por ejemplo:

```
char *Nombre (int orden);
```

El prototipo de la función anterior devuelve un puntero a **char** donde puede estar guardado un texto con una lista de nombres que se corresponden con un número de **orden** especificado.

PARA TENER EN CUENTA

En una de las exposiciones iniciales de este libro dijimos que el lenguaje C hace exactamente lo que nosotros le digamos que haga desde la programación. Por ejemplo, aunque es posible hacerlo, no resulta correcto devolver la dirección de una variable local de una función.

Declaración de un puntero a una función

Hasta aquí, hemos visto cómo pasar o devolver información de datos a funciones que manejan estos datos. Pero también podríamos usar punteros para manejar funciones. De hecho, es la única forma de hacerlo. Como una función tiene una dirección en memoria donde inicia su ejecución, será entonces esta dirección la que guardará el puntero. Pero, si pensamos en lo que hemos visto aquí sobre funciones, nos daremos cuenta de que no es suficiente con saber su dirección para invocarla. Necesitamos, además, conocer la cantidad y el tipo de parámetros que debemos pasarle, y el tipo de datos que retornará la función. Veamos un ejemplo de declaración de un puntero a una función:

```
char (*pMiFuncion) (int);
```

Parece un poco extraño y quizás hasta confuso. El nombre de la función es **pMiFuncion** y sirve solamente para apuntar a funciones que tienen un solo argumento del tipo **int** y que retornan un valor tipo **char** al programa que la llama. Solamente podemos usar este puntero para apuntar a funciones que han sido definidas con estas características. Si, por ejemplo, quisiéramos un puntero a funciones que devuelven datos tipo **float**, entonces necesitaríamos definir otro puntero con las nuevas características.

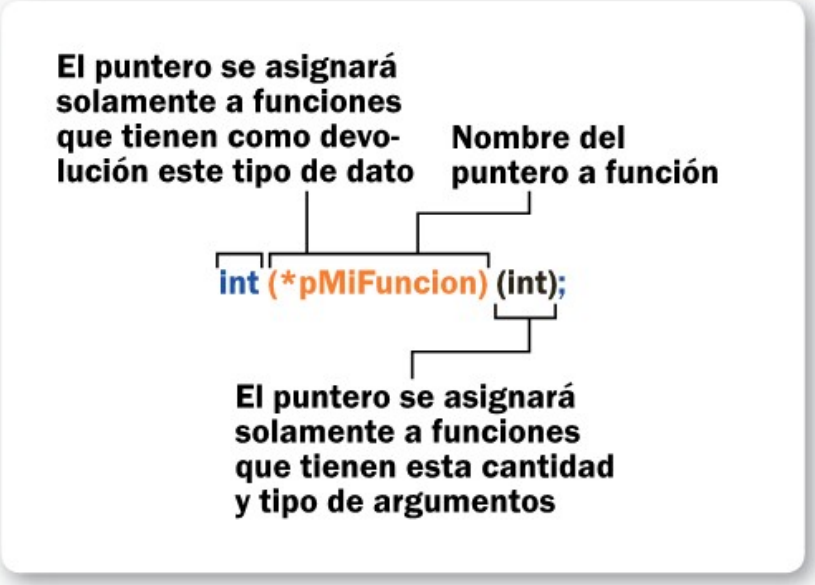


Figura 3. Declaración de un puntero a una función. Es importante cuidar muy bien la sintaxis de escritura de la declaración. Los paréntesis son necesarios, ya que de lo contrario la declaración es totalmente diferente de lo buscado.

En la declaración de punteros a función, se debe encerrar el operador de indirección y el nombre de la función entre paréntesis. El segundo juego de paréntesis encierra la lista de parámetros que se le pasa a la función. Veamos un ejemplo de cómo aplicar punteros a funciones:

```
/* Punteros a función                */

#include <stdio.h>
/*Declaración de los prototipos de función*/
int Mult(int,int);
int Div(int,int);

void main()
{
```

```
        int a=20,b=5;
        int res=0;              /*Variable de resultado*/
        int (*pFunc) (int,int);         /*Puntero a función*/

        pFunc=Mult;                     /*Apunto a función Mult*/
        res=pFunc(a,b);    /*Ejecuto función*/
        printf("\nProducto de %i y %i es %i",a,b,res);
        pFunc=Div;                      /*Apunto a función Div*/
        res=pFunc(a,b);    /*Ejecuto función*/
        printf("\nDivisión de %i y %i es %i",a,b,res);
}

int     Mult(int a, int b)
{
        return (a*b);
}

int     Div(int a, int b)
{
        return (a/b);
}
```

Podemos observar la forma en que se asigna la dirección de memoria de cada una de las funciones al puntero. Consideremos que cuando hacemos esto, estamos tomando la dirección de memoria del inicio de la función y posteriormente nos encargamos de asignarla al puntero que corresponde. Luego, simplemente procedemos a invocar a la función de forma indirecta a través del puntero mencionado, para ello pasamos como argumentos los parámetros esperados por esta.

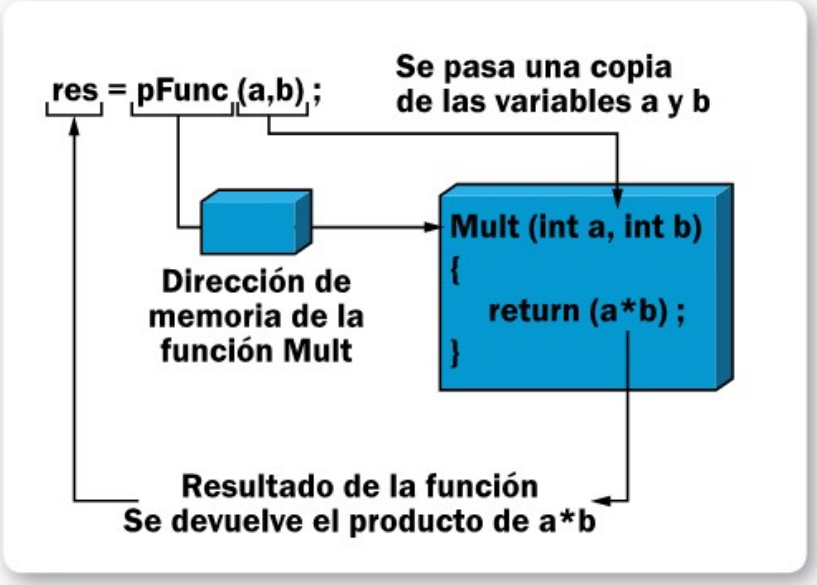


Figura 4. Forma de funcionamiento de un puntero a función. El puntero toma la dirección de memoria de inicio de la función para poder acceder a esta.

Como los punteros a funciones son variables como cualquier otra, entonces podemos pasar punteros a funciones como argumentos de otras funciones. Por ejemplo:

```
/* Punteros a función                    */

#include <stdio.h>
/*Declaración de los prototipos de función*/
int Mult(int,int);
int Div(int,int);
int F(int(*pF)(int,int),int,int); /*Función con
        un puntero a función como parámetro*/

void main()
{
        int a=20,b=5;
```

```
        int res=0;          /*Variable de resultado*/

        res=F(Mult,a,b);    /*Paso puntero a función*/
        printf("\nProducto de %i y %i es %i",a,b,res);
        res=F(Div,a,b);     /*Paso puntero a función*/
        printf("\nDivisión de %i y %i es %i",a,b,res);
}

int F(int(*pFunc)(int,int),int a, int b)
{
        return pFunc(a,b);
}

int     Mult(int a, int b)
{
        return (a*b);
}

int     Div(int a, int b)
{
        return (a/b);
}
```

En este caso, podemos darnos cuenta de que nos encargamos de declarar una función **F**, cuyo primer parámetro es un puntero a función, y los dos parámetros siguientes son los valores con los cuales operan las funciones **Mult** y **Div**.

Consideremos que la función **F** se encarga de recibir estos parámetros y posteriormente nos devuelve el valor de las funciones invocadas a través del puntero a función.

Variables en funciones

La forma de estructurar nuestros programas en funciones simplifica el seguimiento y la búsqueda de errores durante la programación, y permite tener más claridad a la hora de interpretar el código. El uso de variables dentro de los programas nos brinda flexibilidad y amplía la potencialidad del lenguaje a través de sus propiedades.

Compartir variables entre funciones

De la misma forma en que declaramos constantes al inicio de un programa, podemos declarar variables fuera de las funciones que lo componen. Estas variables se denominan **variables globales** y se declaran de la misma forma que cualquier otra. Aquí, es la posición dentro del programa lo que importa: las variables globales pueden ser accedidas desde cualquier función. De esta manera, pueden reemplazar la necesidad de declarar funciones que reciban parámetros. No obstante, cuanto menos variables globales tengamos, mejor claridad y seguimiento se podrá hacer sobre el programa. Es muy fácil modificar una variable global y luego perder de vista su significado o, peor aun, perder de vista las consecuencias que puede acarrear su mal uso. Estas dificultades crecen a medida que el programa se va haciendo más extenso. De esta forma, nuestros programas se vuelven cada vez más proclives a errores.

Recursividad

Es una técnica de programación en donde una función se llama a sí misma. Si bien no es una práctica muy común, puede resultar muy efectiva en ciertos casos. Uno de los mejores ejemplos para ilustrar el proceso de la recursividad es a través del cálculo del factorial de un número entero. Para entender qué hace el programa,

debemos conocer qué es el factorial de un número: es, simplemente, el resultado de multiplicar todos los números enteros desde el 1 hasta el número en sí mismo. Por ejemplo, el factorial de **5** es **120**, que resulta de multiplicar **1x2x3x4x5**. Veamos, aquí, el programa:

```
/* Recursividad: Factorial de un n°*/
#include <stdio.h>

long fact(long);          /*Prototipo de función*/

void main()
{
        long n=0;
        printf("\nIngrese un número entero: ");
        scanf("%ld",&n);
        printf("\n\nEl factorial de %ld es: %ld\n",
                n,fact(n));
}

long fact(long n)
{
        /*Cálculo del factorial con recursividad*/
```

PROFESOR EN LÍNEA

Si tiene alguna consulta técnica relacionada con el contenido, puede contactarse con nuestros expertos: profesor@redusers.com

```
        return n<2 ? n : n*fact(n-1);
}
```

Podemos observar la forma en que la función **fact** se llama a sí misma, pero con un argumento que va disminuyendo sucesivamente llamada tras llamada, hasta que se cumpla la condición **n<2**. Cuando se cumple, se van completando hacia atrás con los valores de la función, hasta alcanzar nuevamente el nivel de **main()** en donde se imprime el resultado. Para un determinado número entero **n**, se realizan **n** llamadas a la función **fact**, y en cada llamada se guarda una copia del argumento de la función y la dirección de memoria donde se debe devolver el valor. Como podemos intuir, para muchos niveles de recursividad, el proceso consume mucha memoria. Esto es particularmente crítico en sistemas con memoria muy limitada, como son los sistemas basados en microcontroladores o los sistemas embebidos.

RESUMEN

En este capítulo, estudiamos las bases para estructurar programas en C y nos acercamos a un mejor conocimiento sobre cómo el lenguaje interpreta y maneja las funciones. Hemos aprendido nuevas y poderosas herramientas de manejo de funciones con punteros, que nos habilitan a desarrollar aplicaciones más estructuradas y con un mejor aprovechamiento de recursos escasos como la memoria. Debemos tener presente que las estructuras funcionales de código son inherentes al lenguaje y este se maneja muy bien de esta forma. Como hemos visto, el lenguaje es ágil, potente y muy permisivo.

*06

Estructuras de datos

En este capítulo, conoceremos las herramientas más flexibles para manejar información y nos daremos cuenta por qué dijimos al principio de esta obra que el lenguaje C es muy simple y, a la vez, muy potente.

Estructuras

Supongamos que tenemos que realizar una aplicación que maneje información acerca de los alumnos de un curso de grado. Por cada uno de ellos, necesitamos conocer, entre otros datos, su nombre y apellido, el género (masculino o femenino), su número de documento, su edad y su dirección. Esta información involucra datos numéricos y alfanuméricos. Con lo que hemos visto hasta ahora, podríamos hacer un arreglo para cada uno de los datos en cuestión y tendríamos tantos arreglos como datos necesitáramos. Sin embargo, esta metodología de trabajo tiene sus limitaciones, ya que no nos permite asociar la información de un arreglo con otra de manera fácil. Se podría hacer por el lado difícil, pero lo mejor sería mantenerlo simple para no complicar la programación. Afortunadamente, el lenguaje nos provee de otra herramienta que nos permite manejar información de distinto tipo, de forma simple y precisa: las **estructuras** y **uniones**.

Para definir una estructura en C, se utiliza la palabra reservada **struct**. Una estructura es, simplemente, una colección de datos que puede o no ser de distinto tipo, y se hace referencia a ellos a través de un solo nombre. La forma general de declaración de una estructura es como presentamos a continuación:

```
struct tipo_estructura
{
        campo_de_estructura_1;
        campo_de_estructura_2;
        ...
        campo_de_estructura_n;
} nombre_estructura = { init_campo_estructura_1,
                                                ampo:estructura_2,  init_c
                                                                    ...
```

```
                                                    ampo_estructura_n   init_c
                                              };
    ;
```

De esta forma, estamos declarando un tipo de datos personalizado, un tipo de datos cuyo nombre está dado por **tipo_estructura** y que se maneja de la misma manera que cualquier otro tipo de datos, tales como **int**, **long**, **float**, etcétera.

La estructura encierra, entre llaves, los **campos_de_estructura_n**, que son las variables con sus respectivos tipos que constituyen la estructura. Es posible tener tantos tipos y cantidad de variables como se quiera, siendo la única limitación la cantidad de memoria disponible en el sistema.

En la forma general de la declaración hemos colocado **nombre_estructura** y los valores de inicialización encerrados entre llaves. Esta parte es opcional; de todos modos, la sentencia debe estar terminada con un punto y coma. Cuando se omite **nombre_estructura**, lo que hacemos simplemente es informar al compilador que a partir de esta parte del código existe un nuevo tipo de datos definido por la estructura, pero no estamos reservando espacio en memoria ya que ninguna variable de este tipo ha sido declarada aun. Sin embargo, si decidimos declarar una estructura al mismo tiempo que

ESTRUCTURAS

Cuando realizamos la declaración de una estructura, estamos definiendo un nuevo tipo de datos, de la misma manera que ya teníamos los tipos de datos **int**, **double**, etcétera. El nuevo estándar ANSI permite copiar estructuras, realizar asignaciones, pasar a funciones y que puedan retornar estructuras de la misma manera que lo hacen con variables ordinarias.

la definimos, como es de suponer, **nombre_estructura** es el nombre que se le va a dar a la estructura que será del tipo **tipo_estructura**. Los valores de inicialización indicados como **init_campo_estructura_n** tendrán los valores iniciales que se asignan a cada campo de la estructura. A modo de ejemplo, veamos cómo construir una estructura para guardar información relacionada con los alumnos del curso que mencionamos al principio de este capítulo.

```
...
/*Definición y declaración de la estructura          */
struct curso
{
        char nombre[12];
        char apellido[15];
        char genero;
        long documento;
        char edad;
        char direccion[30];
} alumno = {"Juan","Perez",0,12345678,38,
                    "Av.Libertador 1771 – 1A"};
...
```

REFERENCIAS

La pérdida de referencia en una lista vinculada se produce cuando se rompe la unión entre cualquiera de sus nodos. Esta situación puede producir la pérdida de datos e incluso de memoria ya que, como los bloques de memoria se reservan dinámicamente con la función **malloc()**, si se pierde su referencia, no será posible liberarla cuando no se esté usando.

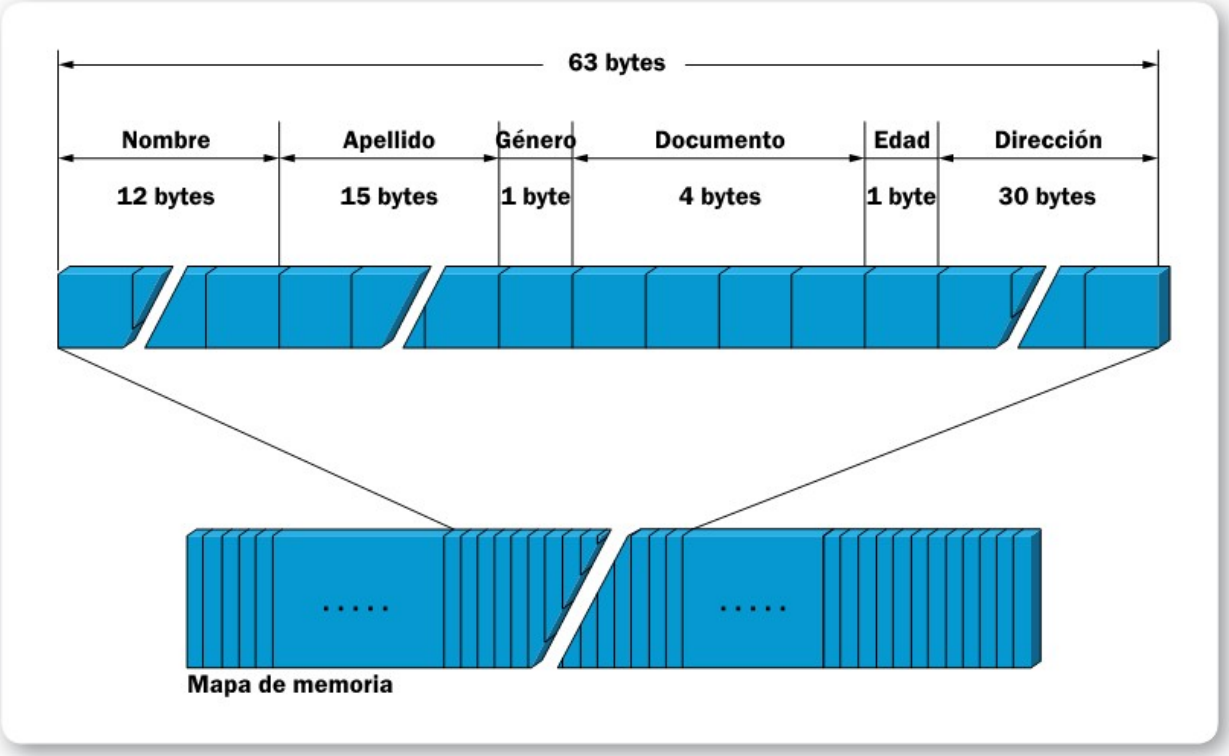


Figura 1. Definición de una estructura en memoria.

Aquí, estamos definiendo y declarando una variable llamada **alumno**, que es del tipo **curso**, y estamos inicializando sus campos constitutivos con valores iniciales dados por **{"Juan","Perez",0,12345678,38, "Av. Libertador 1771 – 1A"}**. Como podemos observar, la inicialización no se hace dentro de las llaves donde se especifican los campos de la estructura, sino afuera, luego de declarar el nombre de la estructura. Esto es porque cuando damos el nombre a los campos de la estructura, estamos definiendo sus miembros y todavía la estructura no está completamente terminada. Recién cuando cerramos la llave de los campos de estructura, esta queda totalmente definida y podemos darle un nombre y asignarle valores iniciales.

Si queremos conocer la cantidad de memoria utilizada por la estructura **alumno**, podemos aplicar el operador **siezeof**:


```
printf("\nTamaño de la estructura: %i bytes",
sizeof(alumno));
```

En este caso, la sentencia anterior arroja el siguiente resultado:

```
Tamaño de la estructura: 63 bytes
```

Analicemos esta respuesta: la estructura está compuesta por tres arreglos tipo **char** de 12, 15 y 30 BYTEs de longitud cada uno, dos variables tipo **char** de 1 BYTE cada una y una variable tipo **long** de 4 BYTEs. Todo esto suma 63 BYTEs. Como expusimos precedentemente, **nombre_estructura** puede ser omitida al momento de la definición de la estructura, y su declaración puede estar hecha en otra parte del programa. Para el ejemplo presentado tendríamos, entonces:

```
...
/*Definición de la estructura                */
struct curso
{
        char nombre[12];
        char apellido[15];
        char genero;
        long documento;
        char edad;
        char direccion[30];
};
...
/*Declaración de la variable                  */
struct curso alumno = {"Juan", "Perez", 0,
        12345678, 38, "Av.Libertador 1771 – 1A"};
...
```

Acceso a estructuras

Hasta aquí, hemos podido definir y declarar estructuras, pero ¿cómo es el procedimiento para acceder a sus campos? Para referirnos a ellos debemos usar el operador **punto** (.), que sirve de nexo entre el nombre de la estructura y el nombre del miembro al cual queremos acceder.

Por ejemplo, en el caso anterior, podremos acceder a cada uno de los campos de la siguiente manera:

```
alumno.nombre = "Juan";
alumno.apellido = "Perez";
alumno.genero = 0;
alumno.documento = 12345678;
alumno.edad = 38;
alumno.direccion = "Av.Libertador 1771 – 1A";
```

También es posible tener estructuras anidadas, es decir, estructuras que contengan otras estructuras como miembro.

Solamente existe una salvedad, y es que debemos tener en cuenta que una estructura no puede contenerse a sí misma.

Veamos un ejemplo donde definimos una estructura que se encarga de agrupar personas por su característica:

```
struct caract
{
        int edad;
        char genero[10];
};
```

Ahora, procedemos a definir otra estructura que se encarga de agrupar personas por sus datos:

```
struct datos
{
        struct caract pnal;
        char nombre[20];
        char apellido[20];
};
```

Y, finalmente, definimos una estructura que agrupa personas por su nacionalidad:

```
struct pais
{
        struct datos pers;
        char nacionalidad[20];
char provincia[20];
char localidad[20];
};
```

Para poder hacer uso de estas estructuras, debemos crear una instancia de cada una de ellas:


```
struct pais indiv;
```

Para poder acceder a cada uno de los datos miembro de la estructura para asignar los valores iniciales, debemos hacer:

```
indiv.pers.pnal.edad = 30;
strcpy(indiv.pers.pnal.genero,"Masculino");
strcpy(indiv.pers.nombre,"Jorge");
strcpy(indiv.pers.apellido,"Gomez");
strcpy(indiv.nacionalidad,"Argentino");
strcpy(indiv.provincia,"Buenos Aires");
strcpy(indiv.localidad,"Suipacha");
```

Observamos que la asignación de variables numéricas se realiza de forma directa, mientras que la asignación de cadenas de texto se lleva a cabo, por ejemplo, a través de la función de cadena **strcpy**. Es decir, esta función se emplea para copiar una cadena de texto dentro de la variable miembro de la estructura. Sin embargo, la utilización de los datos de la estructura es directa, como podemos ver en las siguientes líneas de código. El acceso a los campos y sub-campos de la estructura se lleva a cabo a través del operador **punto**.

```
printf("\n%s %s %s es de nacionalidad %s.",
strcmp(indiv.pers.pnal.genero,"Masculino")?
        "La señora":"El señor",indiv.pers.nombre,
        indiv.pers.apellido,indiv.nacionalidad);
printf("\nTiene %i años y es de la localidad"
        " de %s, %s.",indiv.pers.pnal.edad,
        indiv.localidad,indiv.provincia);
```


Usamos el operador condicional y la función de comparación de cadenas de texto **strcmp** para imprimir uno u otro texto, dependiendo del género de la persona. Si el género es Masculino, imprime **"El señor"**, mientras que de otra forma imprime **"La señora"**.

Cuando vimos la forma general de declaración de una estructura dijimos que **tipo_estructura** especifica el tipo de datos que estamos creando. Sin embargo, **tipo_estructura** puede ser omitida de la definición cuando realizamos la definición y declaración de la estructura en una sola sentencia, es decir, cuando se crea, al menos, una instancia de la estructura cuyo nombre está dado por **nombre_estructura**. Ahora, posteriormente no podremos declarar más variables del tipo de la estructura.

En un ejemplo anterior, definimos una estructura del tipo **curso** y declaramos en forma simultánea una variable llamada **alumno** de este tipo. En esa oportunidad, también hubiera sido factible hacer:

```
struct
{
        char nombre[12];
        char apellido[15];
        char genero;
        long documento;
        char edad;
        char direccion[30];
} alumno;
```

En esta instancia, nos encargamos de realizar la definición de una estructura sin nombre y declarando una variable cuyo tipo está especificado por la estructura. Posteriormente, no será posible crear más variables de esta clase. En el caso que necesitemos hacerlo, lo llevaremos a cabo a continuación de **alumno**; para ello tendremos que colocar los nombres de cada una de las variables deseadas, separados por comas.

Arreglos de estructuras

De la misma forma que podemos crear arreglos con variables ordinarias, podemos crear arreglos con estructuras. Para ello, procedemos de forma idéntica a cuando declaramos arreglos con otro tipo de variables. Por ejemplo, vimos anteriormente cómo declaramos una estructura llamada **indiv**, que guardaba información acerca de un determinado individuo. Esta variable, que es del tipo **pais**, tratada tal cual está, nos sirve para guardar información de un solo individuo. Podríamos, entonces, declarar un arreglo para guardar la información de varios individuos:

```
struct pais indiv[1000];
```

De esta forma, tenemos la posibilidad de guardar la información de un grupo de, por ejemplo, 1000 individuos. Nuevamente, la cantidad de información que podremos almacenar está limitada solamente por la cantidad de memoria disponible que tengamos en nuestro sistema.

Ahora bien, ¿qué tipo de operaciones podemos llevar a cabo con estructuras? Pues bien, las únicas permitidas son la asignación y la aplicación del operador **&** para obtener su dirección. En el ejemplo anterior, si hacemos lo siguiente:

OPERACIONES CON ESTRUCTURAS

Las únicas operaciones permitidas con estructuras son la asignación y la aplicación del operador **&** para obtener su dirección. No es posible sumar o comparar con una estructura completa. Deberíamos, en todo caso, realizar estas operaciones con cada uno de los campos en forma individual.

```
indiv[70] = indiv[231];
```

Estamos asignando a cada uno de los campos de **indiv[70]** los valores que actualmente tiene **indiv[231]**.

Dijimos que las estructuras no eran otra cosa que una definición de un tipo personalizado de datos que puede ser asignado a una variable. Además, a las estructuras podemos aplicarles el operador **&** para conocer su dirección. De lo anterior, surge prácticamente de forma natural que también es posible tener **punteros a estructuras**. Por ejemplo:

```
struct pais *pIndiv
;
```

Ahora, tenemos un puntero a estructura **pIndiv** del tipo **pais**. Para apuntar puntualmente a uno de los elementos del arreglo, cualquiera de ellos, hacemos:

```
pIndiv = &indiv[9];
```

De esta forma, el puntero apunta al inicio del décimo elemento del arreglo. Para acceder a los miembros de la estructura, por ejemplo el nombre del individuo, hacemos:


```
printf("\nNombre: ",(*pIndiv).pers.nombre);
```

Veamos la sintaxis: los paréntesis que encierran el nombre del puntero son esenciales, no obstante, si esta nomenclatura nos puede

resultar engorrosa o no muy clara, el lenguaje nos provee de una manera más sencilla de acceder a los miembros de una estructura a través de un puntero, y es la siguiente:

```
printf("\nNombre: ",pIndiv->nombre);
```

En las dos últimas sentencias, la forma de acceder a los miembros de una estructura es idéntica. Sin embargo, la notación utilizada en la segunda sentencia es mucho más clara y de ella se desprende inmediatamente hacia dónde apunta el puntero de una forma mucho más intuitiva, al mismo tiempo que permite una lectura mucho más fácil y limpia del código. El operador **->** está compuesto simplemente por el signo **menos** (**-**) seguido inmediatamente del signo **mayor que** (**>**).

Si avanzamos un poco más, podemos decir que un puntero pue de ser miembro de una estructura. Incluso, podemos tener punteros a estructuras como miembros. En nuestro ejemplo, la estructura del tipo **pais** puede contener un puntero a una estructura del tipo **pais**. Ahora, ¿puede tener alguna utilidad tener una estructura que contenga como miembro un puntero que apunta a otra estructura del mismo tipo? Definitivamente, sí. A continuación, veremos por qué. En primer lugar, debemos incluir un puntero en la definición de la estructura **pais**:

```
struct pais
{
        struct datos pers;
        char nacionalidad[20];
        char provincia[20];
        char localidad[20];
```

```
        struct pais *siguiente; /*Puntero a siguiente
                                  esturctura*/
};
```

Esto nos permite guardar, en el puntero, la dirección de otra estructura, de manera que podremos concatenar distintas estructuras entre sí a través de este puntero.

El objetivo de esto es lograr armar una cadena de estructuras enlazadas entre sí a través de un **puntero de enlace**, que es miembro de cada estructura y que guarda la dirección de memoria de la siguiente estructura en la cadena. Esto se conoce generalmente como **listas vinculadas** y cada una de las estructuras que forman parte de la lista vinculada se denomina **nodo**.

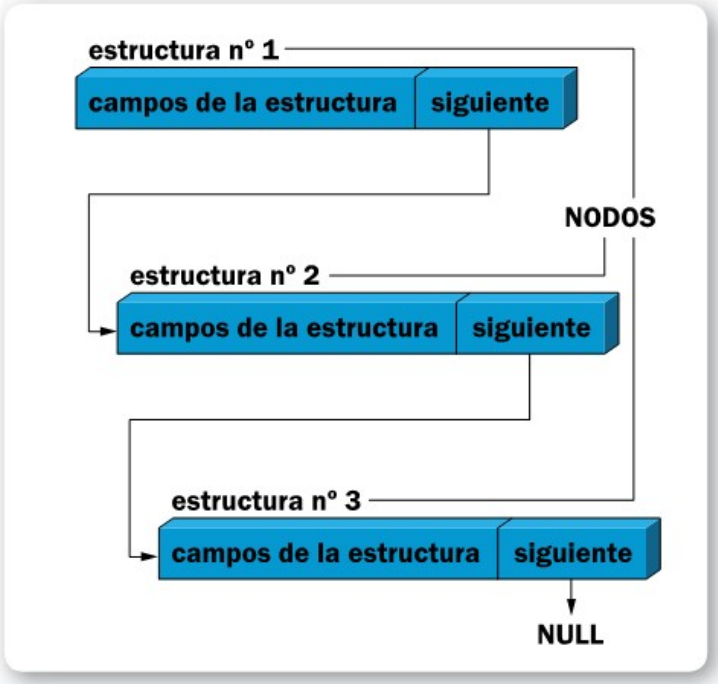


Figura 2. El vínculo se realiza a través de un puntero que es miembro de la lista y que guarda la dirección de memoria del siguiente nodo en la lista.

Imaginemos que, al principio, la lista está vacía. Entonces, en nuestro programa, tendremos algo como esto:

```
void main()
{
        struct pais *lista=NULL,*aux=NULL,*nuevo=NULL;
        ...
}
```

Es decir, especificamos explícitamente que la lista está vacía y asignamos un **puntero cabecera** al que llamamos **lista**, que apunta a **NULL**. Al mismo tiempo, declaramos dos punteros auxiliares llamados **aux** y **nuevo**, del mismo tipo, que nos ayudarán en el proceso de concatenación de datos entre las estructuras de la lista. Ahora, para agregar la primera estructura a la lista, debemos reservar el espacio de memoria suficiente a través de la función **malloc()**:

```
lista = malloc(sizeof(struct pais));
```

Con esto, logramos que el puntero cabecera **lista** apunte al inicio de la lista vinculada. Pero los miembros de la primera estructura están vacíos, por lo que debemos completarlos:

```
lista->pers.pnal.edad = 30;
strcpy(lista->pers.pnal.genero,"Masculino");
strcpy(lista->pers.nombre,"Jorge");
strcpy(lista->pers.apellido,"Gomez");
strcpy(lista->nacionalidad,"Argentino");
strcpy(lista->provincia,"Buenos Aires");
strcpy(lista->localidad,"Suipacha");
```


Finalmente, hacemos que el puntero de enlace apunte a **NULL**, pues esta estructura es la última de la cadena.

```
lista->siguiente = NULL;
```

Ahora, vamos a agregar otro nodo a la lista. Para ello, tenemos dos opciones: podemos agregarlo al **principio** o al **final**. Si queremos hacerlo al principio, debemos hacer que el puntero auxiliar tome la dirección del primer nodo, luego que el puntero cabecera apunte al nuevo nodo y, finalmente, guardar en el puntero de enlace del nodo agregado el contenido del puntero auxiliar:

```
aux = lista; /*Guardo direccion del nodo actual*/
lista=malloc(sizeof(struct pais)); /*Nuevo nodo*/
/*Cargo los datos en el nuevo nodo*/
lista->pers.pnal.edad = 43;
strcpy(lista->pers.pnal.genero,"Masculino");
strcpy(lista->pers.nombre,"Juan");
strcpy(lista->pers.apellido,"Paredes");
strcpy(lista->nacionalidad,"Argentino");
strcpy(lista->provincia,"Córdoba");
strcpy(lista->localidad,"Córdoba");
/*Vinculo los nodos de la lista*/
lista->siguiente = aux;
```

La primera línea del bloque de código anterior es de vital importancia, pues evita la **pérdida de referencia**. Cuando esta se produce, tenemos una fuga de memoria porque, al romperse un vínculo de la lista, ya no tendremos forma de conocer la dirección de memoria del nodo desvinculado y, por lo tanto, este espacio de memoria no podrá ser recuperado.

Si queremos que los nodos se vayan agregando al final de la lista, la situación cambia y la solución ya no es tan evidente.

Ahora, necesitaremos un puntero más que nos ayude en esta tarea, por lo que recurriremos al puntero auxiliar **nuevo**, que hemos declarado anteriormente. Debemos buscar el último elemento de la lista y haremos referencia a él a través del puntero auxiliar **aux**. Para ello, implementamos un bucle que recorra los nodos hasta encontrar aquel cuyo puntero de enlace tenga el valor **NULL**, es decir, el final de lista.

Una vez que encontramos el último nodo, creamos uno nuevo y usamos el segundo puntero auxiliar **nuevo** para apuntar a él. Completamos los datos del nuevo nodo y hacemos que su puntero de enlace apunte a **NULL** (que será el último nodo de la lista una vez que quede vinculado). A continuación, en el puntero de enlace del nodo apuntado por **aux**, guardamos el contenido del puntero **nuevo** (que tiene la dirección de memoria del nuevo nodo). Veamos cómo queda todo esto en código:

```
aux=lista;
while(aux->siguiente != NULL)    /*Recorro todos los
        aux=aux->siguiente;                    punteros de enlace*/
nuevo=malloc(sizeof(struct pais)); /*Nuevo nodo*/
/*Cargo los datos en el nuevo nodo*/
lista->pers.pnal.edad = 50;
strcpy(lista->pers.pnal.genero,"Masculino");
strcpy(lista->pers.nombre,"Germán");
strcpy(lista->pers.apellido,"Sosa");
strcpy(lista->nacionalidad,"Argentino");
strcpy(lista->provincia,"Misiones");
strcpy(lista->localidad,"Oberá");
nuevo->siguiente=NULL;          /*Ultimo nodo apunta
                                                ULL*/      a      N
/*Vinculo los nodos de la lista*/
aux->siguiente = nuevo;
```

Campos de BITs en estructuras

Los **campos de BITs** en estructuras nos proveen un mecanismo que nos permite definir variables en donde cada una de ellas representa uno o más BITs dentro de un tipo de dato entero, pudiendo hacer referencia a cada una de estas variables explícitamente a través de su nombre. A estas variables se las conoce como **banderas** o **flags**. Los campos de BITs tienen amplia aplicación en donde la memoria es un bien muy preciado y el costo de desperdiciarla en declarar variables que guardan datos muy chicos, comparados con el tamaño de memoria que ocupan, es muy alto. Por ejemplo, usar un **int** para declarar una variable que guarda un dato que puede tomar dos valores: **verdadero** o **falso**. En este caso, resultaría muy conveniente poder emplear solamente 1 BIT (**1**=verdadero, **0**=falso) de los 16 que nos proporciona el tipo **int**. Los sistemas electrónicos basados en microcontroladores son el ejemplo perfecto en donde podemos aplicar esta técnica. Normalmente, la memoria de la que disponemos en estos sistemas es muy escasa, por lo que las aplicaciones que ejecutemos en estas plataformas deben ser de código muy reducido y eficiente, y el uso de memoria debe mantenerse al límite, ahorrando cada BIT que sea posible.

Veamos un ejemplo de definición de estructura de campos de BITs. Representaremos, con un BIT, los estados de sensores en un

LISTAS VINCULADAS

Una desventaja que presentan las listas vinculadas es que solamente podemos recorrerlas en un solo sentido: desde el principio hasta el final. Esto es debido a que cada nodo en la lista contiene información del siguiente. En cambio, si en cada estructura, además de tener un puntero que apunta al siguiente nodo, tuviéramos un puntero que apuntara al nodo anterior, sería posible recorrer la lista en ambos sentidos. Esto da origen a la **lista doblemente vinculada**.

sistema de alarma doméstica y tendremos un contador de 2 BITs que llevará la cuenta de la cantidad de intentos al ingresar una contraseña por teclado. Sabemos que no debe superar los tres intentos. Entonces, nos alcanza con disponer 2 BITs para esta operación:

```
struct
{
        unsigned int pta_ppal : 1;/*Puerta principal*/
        unsigned int pta_tras : 1;/*Puerta trasera*/
        unsigned int ventana1 : 1;/*Ventana 1*/
        unsigned int ventana2 : 1;/*Ventana 2*/
        unsigned int ventana3 : 1;/*Ventana 3*/
        unsigned int s_humo1  : 1;/*Sensor de humo*/
        unsigned int c_acceso : 2;/*Cont.de accesos*/
} estado;
```

Vemos que, en la declaración de la estructura, no le hemos dado un nombre al tipo de dato que estamos creando, ya que en la misma definición estamos declarando una variable llamada **estado**, y no será necesario en el resto del programa declarar otra variable con

CAMPOS DE BITS

Si bien el uso de campos de BITs nos aporta beneficios a la hora de ahorrar memoria, contar con ellos implica pagar un costo. El hecho de usar campos de BITs en nuestros programas hace que su ejecución sea más lenta en comparación con el uso de variables ordinarias. Debemos, entonces, poner en la balanza cuál es nuestra prioridad y decidir si el ahorro de memoria puede pagar el costo de la velocidad de ejecución del programa.

este tipo de datos. En una sola variable (**estado**), llevamos el control del estado de 6 sensores de nuestro sistema de alarma doméstica y tenemos un contador que lleva el registro de los intentos de acceso por teclado. Como vemos, los beneficios que nos brinda esta técnica desde el punto de vista del ahorro de memoria son excelentes. Al mismo tiempo, la técnica facilita su uso, al no tener que recordar qué BIT está asignado a cada sensor. Simplemente, nos referimos a cada BIT de estado por su nombre y automáticamente accedemos a su contenido, sin tener la necesidad de conocer exactamente qué BITs fueron asignados a las banderas de estado.

El sentido de definir la estructura **estado** es, en este caso, tener un medio en donde el programa se puede enterar del estado de cada sensor y poder actuar en consecuencia. Pero, ¿quién modifica realmente las banderas de estado? Pues bien, en nuestro programa tendremos una función que se encarga de recibir la información que provee cada sensor y de reflejarla en las banderas de estado. Una posibilidad sería la siguiente:

```
...
estado.pta_ppal = 1;      /*Puerta ppal. activada*/
estado.pta_tras = 0; /*Puerta trasera normal*/
estado.ventana1 = 0; /*Ventana1 normal*/
estado.ventana2 = 0; /*Ventana1 normal*/
estado.ventana3 = 0; /*Ventana1 normal*/
estado.s_humo   = 0; /*Sensor de humo normal*/
...
estado.conta++;                  /*Incremento contador*/
if(estado.conta>3)
{
      /*Hubo más de tres intentos de acceso*/
```

```
        ...
}
else
{
        /*Ingreso por teclado normal*/
        ...
}
```

Estructuras y funciones

El uso de estructuras en C es muy útil para el lenguaje, y su empleo con funciones es muy importante. Aquí, veremos cómo pasar estructuras a funciones y cómo estas pueden devolver estructuras.

Estructuras como argumento de funciones

No hay nada misterioso en pasar una estructura a una función. Recordemos que las estructuras se manejan como una variable normal, por lo tanto, es de esperar que la forma de pasarlas a funciones sea similar a lo que ya vimos.

Consideremos la siguiente estructura:

```
struct cliente
{
        char razon_social[20];
        char direccion[20];
        long telefono;
        char localidad[15];
```

```
        char contacto[15];
        char email[30];
}
```

En nuestro programa, podemos tener una función que se encargue de verificar si la razón social de un determinado cliente ya existe:

```
int verifica_cliente(struct cliente miembro1,
        struct cliente miembro2)
{
        if(strcmp(miembro1.razon_social,
                miembro2.razon_social)==0)
                return 1; /*El cliente ya existe*/
        else
                return 0; /*El cliente no existe*/
}
```

La función **verifica_cliente** tiene como argumento dos variables del tipo **cliente** y verifica, a través de la función **stcmp()**, si los campos **razon_social** de cada uno de ellos son idénticos, lo cual ocurre cuando **strcmp()** arroja un **0** como resultado.

Punteros a estructuras como argumento de funciones

Cuando vimos la manera que utiliza el lenguaje cuando se llama a una función, dijimos que los argumentos originales se mantienen a salvo y lo que en verdad se pasa es una copia de estos. Lo mismo ocurre cuando pasamos estructuras a funciones. Es decir, lo que en realidad se pasa no es la estructura original, sino una copia de

cada una de las estructuras que se pase como argumento. Como las estructuras que se pasan tienen muchos campos, y más aun en el caso de que tengamos que pasar muchos argumentos, esto implica que la copia de los argumentos que se genera en forma automática ocupará mucha memoria. Para mantener el uso de memoria al mínimo posible, lo mejor es pasar como argumento de la función un puntero a la estructura a la que haremos referencia. Esta práctica reduce el uso de memoria en el proceso, y también implica una disminución en el tiempo de copia. Es mayor el tiempo que lleva copiar una estructura que el que toma copiar un puntero. La función podrá acceder a la estructura original en forma directa y no a través de una copia, lo cual hace que el proceso sea más eficiente.

En el ejemplo anterior, si usáramos punteros para pasar información de la estructura a la función, tendríamos:

```
int verifica_cliente(struct cliente *miembro1,
        struct cliente *miembro2)
{
        if(strcmp(miembro1->razon_social,
                miembro2->razon_social)==0)
                return 1; /*El cliente ya existe*/
        else
                return 0; /*El cliente no existe*/
}
```

Estructuras como valor de retorno de funciones

Nuevamente, encontraremos que tampoco hay nada misterioso en devolver una estructura desde una función. Solamente debemos indicar, en el prototipo de la función, que el tipo de dato devuelto por esta será el que corresponde a la estructura.

```
struct cliente Correo (void);                    /*Prototipo*/
```

Sin embargo, a pesar de que es posible devolver una estructura desde una función, generalmente es mucho más conveniente devolver un puntero a la estructura a la que hacemos referencia.

Uniones

Vimos que es posible realizar un ahorro significativo de memoria cuando hacemos uso de técnicas tales como campos de BITs. Adicionalmente a ello, el lenguaje nos provee de otra forma que nos permite colocar distintas variables dentro de un mismo espacio de memoria, solapadas unas con otras.

Esta técnica, aplicada en conjunto con campos de BITs, conforma una herramienta sin igual en el ahorro de espacio de memoria donde esta es muy limitada o escasa. El único inconveniente es que, como la información está almacenada en un espacio confinado de memoria y las variables están superpuestas unas con otras, solamente una variable del conjunto puede ser accedida en un determinado momento.

La herramienta que tiene el lenguaje que permite compartir una misma porción de memoria con un número diferente de variables se denomina **unión**. La forma general de declaración de una unión es muy similar a la de una estructura:

```
union nombre_tipo_union
{
        campo_union_1;
        campo_union_2;
        ...
```

```
        campo_union_n;
} nombre_union;
```

Esta sentencia define una unión del tipo **nombre_tipo_union**, que comparte memoria con los campos **campo_union_n**, y declara una instancia de la unión a través de **nombre_union**.

Para acceder a cada uno de los miembros de la unión, debemos proceder de la misma manera que lo hacíamos con las estructuras.

```
nombre_union.campo_union_1 = valor;
...
nombre_union.campo_union_n = valor;
```

A continuación, veremos un ejemplo en donde declaramos una estructura y una unión que tienen los mismos tipos y cantidad de campos, a fin de mostrar el solapamiento de memoria que ocurre en el caso de las uniones.

```
/* Estructuras y Uniones */
#include <stdio.h>
/*Definición de la estructura*/
strcut
{
        int a;
        int b;
        float c;
        char d[20];
        char e[40];
```

```
} el;

union
{
        int a;
        int b;
        float c;
        char d[20];
        char e[40];

} ul;

void main()
{

        printf("\nTamaño de la estructura el: %i "
                "bytes",sizeof(el));

        printf("\nTamaño de la unión ul:          %i "

                "bytes",sizeof(ul));
}
```

PROFESOR EN LÍNEA

Si tiene alguna consulta técnica relacionada con el contenido, puede contactarse con nuestros expertos: profesor@redusers.com

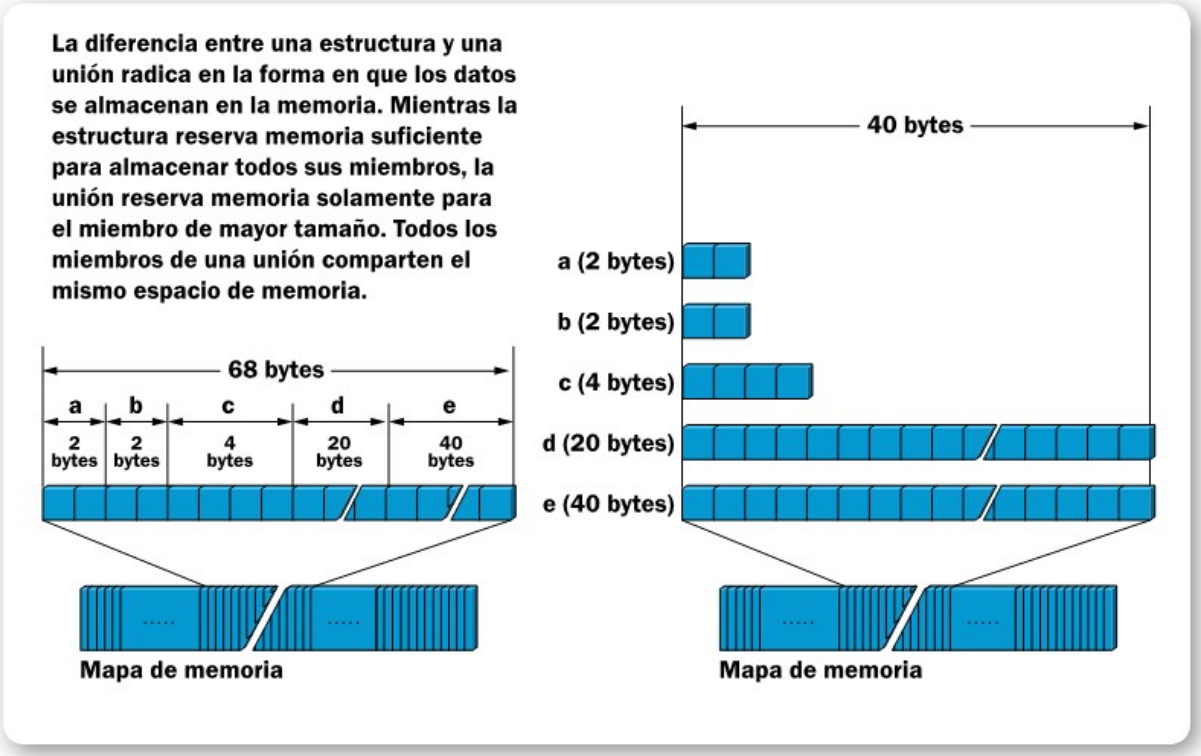


Figura 3. Diferencia entre una estructura y una unión. Las estructuras reservan espacio de memoria para todos sus miembros, mientras que las uniones reservan espacio para el mayor de sus miembros y este espacio se comparte con los demás.

Definición de tipos de datos propios

¿Acaso cuando vimos estructuras y uniones no estuvimos declarando tipos de datos propios? Bueno, desde un cierto punto de vista esto es así pero, de todas maneras, debemos usar la palabra reservada **struct** o **union** en las declaraciones. Sin embargo, el lenguaje nos provee de un método para resolver esto y permitir a las estructuras o uniones tener una sintaxis similar a los tipos nativos del lenguaje. Esto se lleva a cabo a través de la palabra reservada **typedef**.

Veamos cómo aplicarla a una estructura:

```
struct el
{
        int a;
        int b;
        int c;
};
```

Ahora, podemos definir otro nombre para declarar la estructura anterior usando **typedef**:

```
typedef struct el coord;
```

Entonces, cuando queramos declarar nuevas instancias de la estructura **Estructural** del tipo **el**, hacemos:

```
coord inicio;
coord final;
```

De esta forma, declaramos dos variables: **inicio** y **final**, que son del tipo **coord**. A su vez, **coord** hace referencia a la estructura tipo **el**.

RESUMEN

Las estructuras de datos en C nos aportan nuevas y poderosas herramientas que nos permiten utilizar el lenguaje de forma ágil y efectiva. A lo largo de estos capítulos, hemos cubierto los tópicos clave para la programación en C, que nos habilitan a crear aplicaciones profesionales: tipos de datos, operadores, punteros, arreglos, cadenas, funciones, estructuras y uniones.

Servicios al lector

Índice temático

F

I

L

M

N

O

P

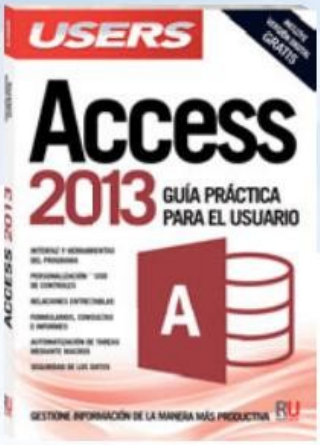
R

S

T

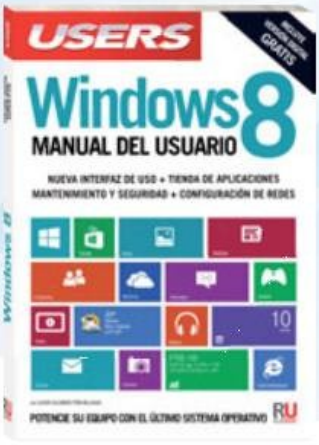
U

V



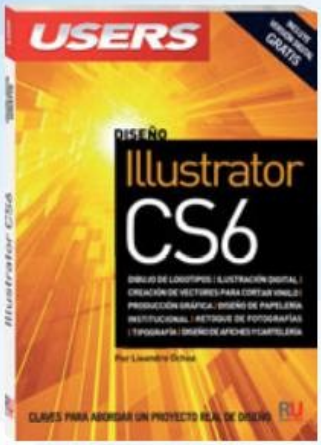
Simplifique tareas cotidianas de la manera más productiva y obtenga información clave para la toma de decisiones.

→ 320 páginas / ISBN 978-987-1949-17-5



Acceda a consejos indispensables y aproveche al máximo el potencial de la última versión del sistema operativo más utilizado.

→ 320 páginas / ISBN 978-987-1949-09-0



La mejor guía a la hora de generar piezas de comunicación gráfica, ya sean para web, dispositivos electrónicos o impresión.

→ 320 páginas / ISBN 978-987-1949-04-5

Aprenda a simplificar su trabajo, convirtiendo sus datos en información necesaria para solucionar diversos problemas cotidianos.

→ 320 páginas / ISBN 978-987-1949-08-3

Un libro ideal para ampliar la funcionalidad de las planillas de Microsoft Excel, desarrollando macros y aplicaciones VBA.

→ 320 páginas / ISBN 978-987-1949-02-1

El libro indicado para enfrentar los desafíos del mundo laboral actual de la mano de un gran sistema administrativo-contable.

→ 352 páginas / ISBN 978-987-1949-01-4

Un libro ideal para ampliar la funcionalidad de las planillas de Microsoft Excel, desarrollando macros y aplicaciones VBA.

→ 320 páginas / ISBN 978-987-1857-99-9

Un libro para maestros que busquen dinamizar su tarea educativa integrando los diferentes recursos que ofrecen las TICs.

→ 320 páginas / ISBN 978-987-1857-95-1

Libro ideal para introducirse en el mundo de la maquetación, aprendiendo técnicas para crear verdaderos diseños profesionales.

→ 352 páginas / ISBN 978-987-1857-74-6

CURSOS INTENSIVOS CON SALIDA LABORAL

Los temas más importantes del universo de la tecnología, desarrollados con la mayor profundidad y con un despliegue visual de alto impacto: explicaciones teóricas, procedimientos paso a paso, videotutoriales, infografías y muchos recursos más.

- 25 Fascículos
- 600 Páginas
- 2 DVDs / 2 Libros

Curso para dominar las principales herramientas del paquete Adobe CS3 y conocer los mejores secretos para diseñar de manera profesional. Ideal para quienes se desempeñan en diseño, publicidad, productos gráficos o sitios web.

Obra teórica y práctica que brinda las habilidades necesarias para convertirse en un profesional en composición, animación y VFX (efectos especiales).

- 25 Fascículos
- 600 Páginas
- 2 CDs / 1 DVD / 1 Libro

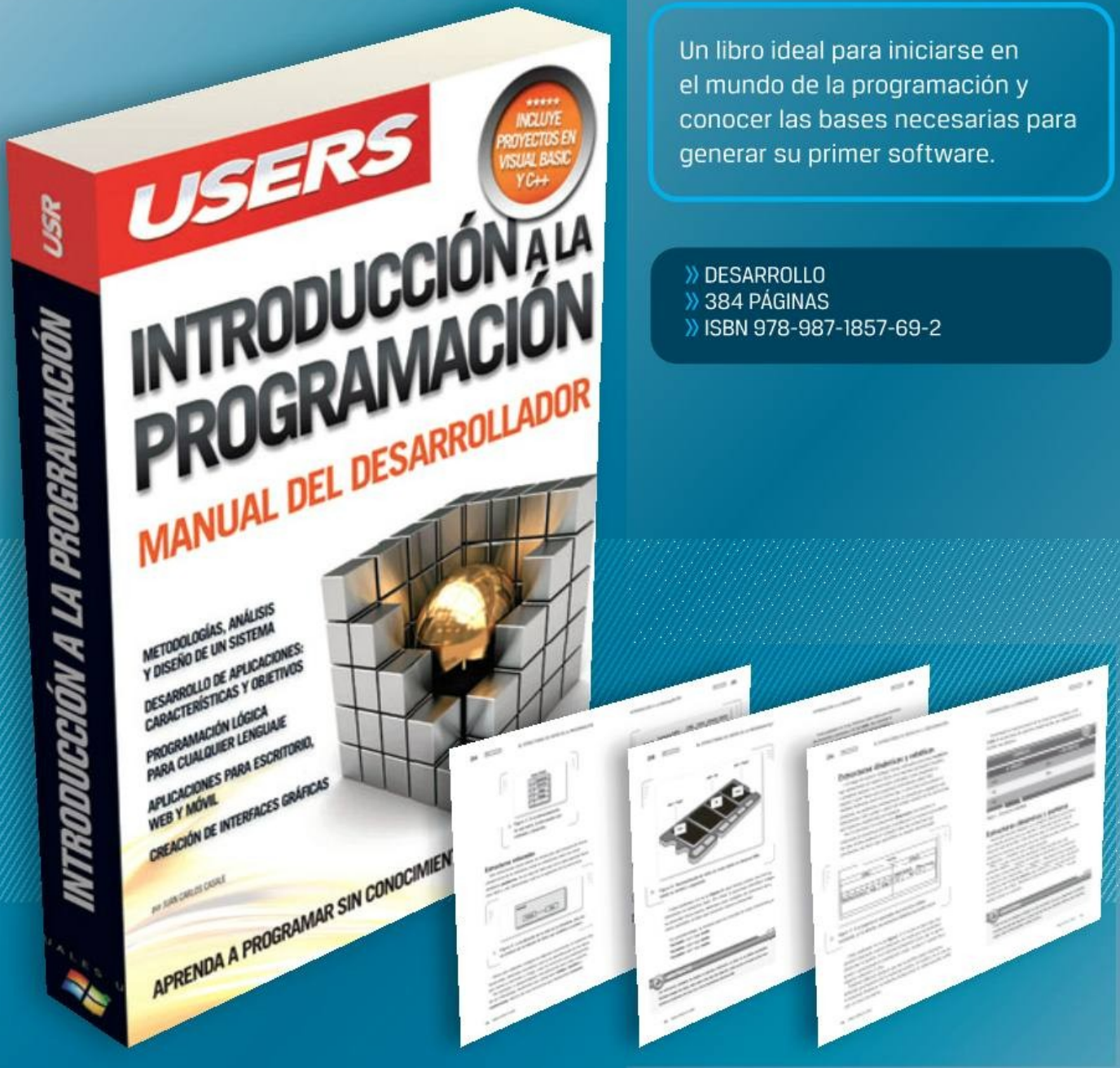
- 25 Fascículos
- 600 Páginas
- 4 CDs

Obra ideal para ingresar en el apasionante universo del diseño web y utilizar Internet para una profesión rentable. Elaborada por los máximos referentes en el área, con infografías y explicaciones muy didácticas.

Brinda las habilidades necesarias para planificar, instalar y administrar redes de computadoras de forma profesional. Basada principalmente en tecnologías Cisco, busca cubrir la creciente necesidad de profesionales.

- 25 Fascículos
- 600 Páginas
- 3 CDs / 1 Libros

APRENDA A PROGAMAR SIN CONOCIMIENTOS PREVIOS



Un libro ideal para iniciarse en el mundo de la programación y conocer las bases necesarias para generar su primer software.

» DESARROLLO
» 384 PÁGINAS
» ISBN 978-987-1857-69-2

USERS

TÉCNICO en
ELECTRÓNICA

PROGRAMACIÓN EN C

El lenguaje C tuvo sus inicios en la programación de sistemas operativos y fue utilizado para realizar la codificación del sistema operativo UNIX. Es un lenguaje de bajo nivel ya que permite acceder en forma muy cercana a instancias de hardware, y por esta razón es indispensable para los técnicos en electrónica.

En esta obra recorreremos los primeros pasos en el aprendizaje de este lenguaje que resulta fascinante, tanto por su facilidad de uso como por su sencillez y capacidad. Conoceremos diferentes funciones, procedimientos, conectores lógicos y estructuras, entre otros conceptos. Los temas se desarrollan en forma didáctica, con explicaciones claras y ejemplos de programación y servirán de base para el aprendizaje de lenguajes de mayor nivel.

CONTENIDO

EXCLUSIVO PARA LECTORES

Profesores en línea:
profesor@redusers.com

TÉCNICO en ELECTRÓNICA
9 789871 949137