AULA POLITÈCNICA / ETSETB

Marco A. Peña Basurto
José M. Cela Espín

Introducción a la programación en C

EDICIONS UPC

Primera edición: septiembre de 2000

Diseño de la cubierta: Manuel Andreu

© Edicions UPC, 2000
Edicions de la Universitat Politècnica de Catalunya, SL
Jordi Girona Salgado 31, 08034 Barcelona
Tel.: 934 016 883 Fax: 934 015 885
Edicions Virtuals: www.edicionsupc.es
E-mail: edicions-upc@upc.es

Producción: CPET (Centre de Publicacions del Campus Nord)
La Cup. Gran Capità s/n, 08034 Barcelona

Depósito legal: B-32.449-2000
ISBN: 84-8301-429-7

Introducción a la programación en C

Marco A. Peña
José M. Cela
Departament d'Arquitectura de Computadors
Universitat Politècnica de Catalunya
08034 Barcelona, España
marcoa@ac.upc.es
cela@ac.upc.es

19 de junio de 2000

Índice General

Índice de Figuras

Índice de Tablas

Prólogo

Este libro surge a partir de la experiencia docente de los autores en la asignatura *Introducción a los ordenadores* de la *Escola Tècnica Superior d'Enginyeria de Telecomunicació de Barcelona*, de la *Universitat Politècnica de Catalunya*. Como su título indica, se trata de un texto de introdución a la programación en lenguaje C. El libro pretende ceñirse a los aspectos fundamentales del estándar ANSI C actual. Aunque hoy en día existen otros lenguajes de programación muy populares como C++ o JAVA, la comprensión de estos lenguajes exige un sólido conocimiento de las bases de programación en C.

El texto está concebido como un curso completo y por lo tanto debe ser leído de forma secuencial. Al final de cada capítulo hay un conjunto de ejercicios propuestos. El lector debe tratar de resolver el mayor número posible de estos ejercicos. De igual forma es una buena práctica el programar los ejemplos resueltos en el texto. El lector debe recordar que la programación es una técnica aplicada, igual que tocar un instrumento musical, y por lo tanto requiere muchas horas de ensayo para ser dominada. Los ejercicios propuestos son lo suficientemente simples como para no requerir conocimientos adicionales de otras materias (matemáticas, física, contabilidad, etc).

Uno de los puntos más importantes para quien empieza a programar es adoptar desde el principio un buen estilo de programación. Esto es, escribir las construcciones del lenguaje de una forma clara y consistente. En este sentido, los ejemplos resueltos en el texto muestran un buen estilo básico de programación, por lo que se recomienda al lector imitar dicho estilo cuando realice sus programas.

Los apéndices A y B son de lectura obligada antes de comenzar a desarrollar programas de complejidad media. En dichos apéndices el lector se familiarizará con el uso del preprocesador y de la librería estándar. Ambas son herramientas fundamentales para desarrollar programas.

El apéndice C incluye una descripción de los formatos de datos en el computador. Este tema no es propiamente de programción, pero la comprensión de dichos formatos ayuda al programador a entender mejor conceptos básicos, como las operaciones de conversión de tipos. El lector puede leer este apéndice en cualquier momento, aunque recomendamos leerlo antes del capítulo 6.

En las referencias bibliográficas se indican algunas direcciones web donde el lector podrá encontrar preguntas y respuestas comunes de quienes se inician en el lenguaje C. Así mismo, el lector podrá encontrar materiales adicionales sobre programación, historia del lenguaje, etc.

Capítulo 1

Conceptos básicos de programación

1.1 Ordenador y periféricos

Un ordenador sólo es capaz de ejecutar órdenes y operaciones muy básicas, tales como:

- Aritmética entera: sumar, restar, multiplicar, etc.
- Comparar valores numéricos o alfanuméricos
- Almacenar o recuperar información

Con la combinación de estas operaciones básicas, y gracias a su gran potencia de cálculo, el ordenador puede llevar a cabo procesos muy complejos. Sin embargo, en cualquier caso existe una estrecha dependencia del ordenador con el programador. Es el programador quien indica a la máquina cómo y qué debe hacer, mediante la lógica y el razonamiento previo, expresado en forma de un programa.

En definitiva, el ordenador sólo es capaz de aceptar datos de entrada, procesarlos y facilitar otros datos o resultados de salida. Los datos se introducen u obtienen del ordenador mediante los periféricos de entrada y salida. Éstos son los encargados de facilitar la relación entre el corazón del ordenador y el mundo exterior, y en particular los usuarios de ordenadores. Dependiendo de su función particular, los periféricos pueden clasificarse en:

Periféricos de entrada: cuya función es facilitar la introducción de datos y órdenes al ordenador: teclado, ratón, lápiz óptico, lector de código de barras, escáner, tableta digitalizadora, etc.

Periféricos de salida: cuya función es mostrar al exterior información almacenada en memoria o los resultados de las operaciones realizadas por el ordenador: pantalla, impresora, plotter, etc.

Periféricos de entrada y salida: capaces tanto de introducir como de extraer información del ordenador: discos y cintas magnéticos, discos ópticos, etc.

Periféricos de comunicación: encargados de establecer y facilitar el intercambio de información entre dos ordenadores: módem, tarjetas de red (*Ethernet*, *Token Ring*, *RDSI*, ...), etc.

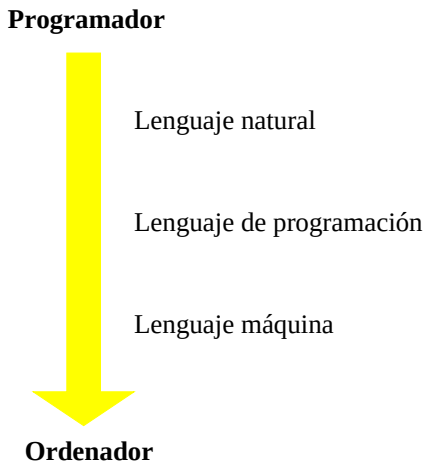


Figura 1.1: Niveles de abstracción en los lenguajes de programación

1.2 Bits, bytes y palabras

La unidad de memoria más pequeña en un ordenador se denomina *bit* (del inglés **bi***nary digi***t**). Puede tomar únicamente dos posibles valores: 0 o 1. En ocasiones, debido a la relación intrínseca con los valores en las señales eléctricas en circuitos digitales, se dice que un bit está *bajo* o *alto*, o bien *desconectado* o *conectado*. Como puede verse, no es posible almacenar mucha información en un solo bit. Sin embargo, un ordenador posee cantidades ingentes de ellos, por lo que podría decirse que los bits son los bloques básicos con los que se construye la memoria del ordenador.

El *byte*, compuesto por ocho bits (algunos autores se refieren a esta unidad como *octeto*), es una unidad de memoria más útil. Puesto que cada bit puede tomar el valor 0 o 1, en un byte pueden representarse hasta $2^8 = 256$ combinaciones de ceros y unos (256 *códigos binarios*). Con estas combinaciones pueden representarse, por ejemplo, los enteros entre 0 y 255 ($0 \ldots 2^8 - 1$), un conjunto de caracteres, etc.

La unidad natural de memoria para un ordenador es la *palabra*. Los ordenadores de sobremesa actuales, por ejemplo, suelen trabajar con palabras de 32 o 64 bits. En grandes ordenadores, el tamaño de la palabra puede ser mucho mayor, pero siempre formada por un número de bits, potencia de 2. En cualquier caso, los ordenadores encadenan dos o más palabras de memoria con el fin de poder almacenar datos complejos y, en general, de mayor tamaño.

1.3 Lenguajes de programación

Un lenguaje de programación podría definirse como una notación o conjunto de símbolos y caracteres que se combinan entre sí siguiendo las reglas de una sintaxis predefinida, con el fin de posibilitar la transmisión de instrucciones a un ordenador. Dichos símbolos y caracteres son traducidos internamente a un conjunto de señales eléctricas representadas en sistema binario, es decir, sólo dos valores: 0 y 1. Esta traducción es necesaria porque el procesador sólo entiende ese lenguaje, al cual nos referiremos como *lenguaje máquina*.

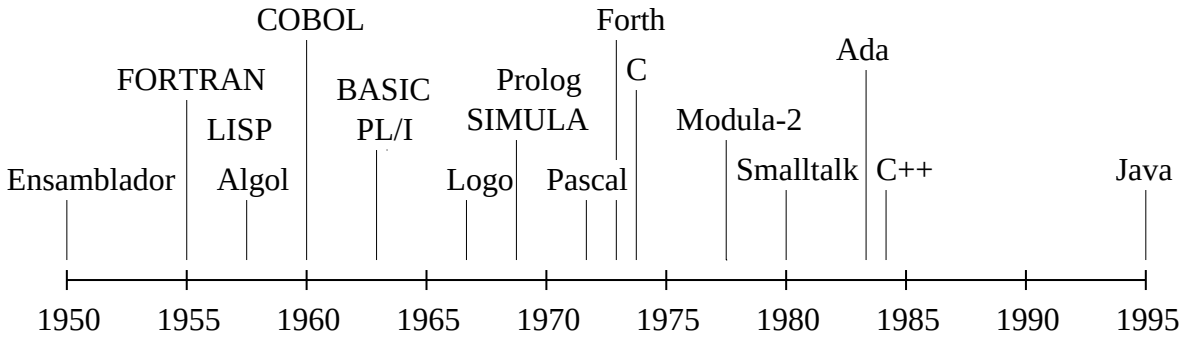


Figura 1.2: Cronología en el desarrollo de algunos lenguajes de programación

1.3.1 Lenguajes de bajo nivel

Se incluyen en esta categoría aquellos lenguajes que por sus características se encuentran más próximos a la arquitectura del ordenador, como el *lenguaje máquina* y el *lenguaje ensamblador*.

Lenguaje máquina

Cualquier problema que deseemos resolver se plantea en primer lugar en nuestro lenguaje natural. Sin embargo, para que la secuencia de pasos que resuelven el problema pueda ser entendida por un ordenador, debe traducirse a un lenguaje muy básico denominado *lenguaje máquina*.

El lenguaje máquina se caracteriza por ser el único que es directamente inteligible por el ordenador, puesto que se basa en la combinación de dos únicos símbolos (0 y 1) denominados bits. Además cada procesador posee su propio lenguaje máquina, por lo que un programa escrito en lenguaje máquina de un procesador X no podrá, en principio, ejecutarse en un procesador Y.

Lenguaje ensamblador

Constituye una evolución del lenguaje máquina. Se basa en la utilización de mnemotécnicos, esto es, abreviaturas de palabras que indican nombres de instrucciones. Para programar en lenguaje ensamblador es necesario conocer en profundidad la estructura y funcionamiento interno del ordenador, así como dominar el uso de diferentes sistemas de numeración, como el binario, hexadecimal, octal, etc.

En general, los programas escritos en ensamblador requieren mucho menos espacio de memoria y se ejecutan más rápidamente que si se hubiesen desarrollado en un lenguaje de alto nivel, puesto que están optimizados para una arquitectura específica. Sin embargo, esto último es un inconveniente, pues causa que los programas no sean portables de un ordenador a otro con un procesador distinto.

1.3.2 Lenguajes de alto nivel

Se engloban aquí todos los lenguajes de programación que por sus características se asemejan más al lenguaje natural del programador. Algunos de los más conocidos son: FORTRAN, BASIC, Pascal, Modula, C, Ada, Java, etc. (ver Fig. 1.2).

La característica más importante de estos lenguajes es que son independientes de la arquitectura del ordenador, por lo que un programa escrito en un lenguaje de alto nivel puede ejecutarse sin problemas

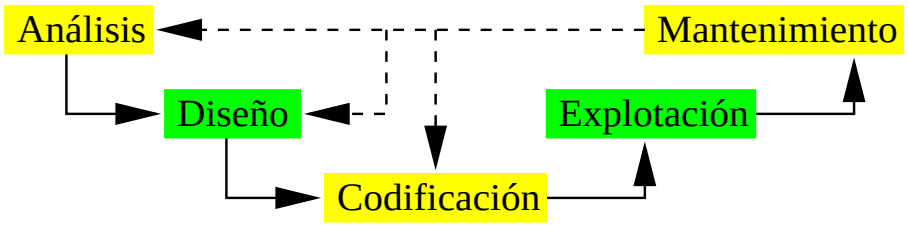


Figura 1.3: Ciclo de vida de un programa

en otros ordenadores con procesadores distintos. Por ello, el programador no necesita conocer a fondo el funcionamiento del ordenador en el que programa, sino que el lenguaje le permite abstraerse de los detalles de bajo nivel. Esta abstracción de la arquitectura de la máquina implica que todo programa escrito en un lenguaje de alto nivel deberá traducirse a lenguaje máquina, de forma que pueda ser entendido y ejecutado por el ordenador. Para ello cada tipo de ordenador deberá disponer de unos programas especiales que realicen dicha traducción (ver Sec. 1.5).

1.4 Elaboración de un programa

El desarrollo de un programa para solucionar un determinado problema informáticamente puede resumirse en el ya clásico concepto de *ciclo de vida*. Éste puede desglosarse en los siguientes pasos a seguir secuencialmente: análisis, diseño, codificación, explotación y mantenimiento (ver Fig. 1.3).

Análisis

En la fase de análisis se estudia cuál es el problema a resolver y se especifican a muy alto nivel los procesos y estructuras de datos necesarios, de acuerdo con las necesidades del cliente. Para realizar un buen análisis será necesario interaccionar con el cliente y conocer a fondo sus necesidades. Antes de proceder al diseño es muy importante haber comprendido correctamente los requerimientos del problema.

Diseño

Una vez bien definido el problema y las líneas generales para solucionarlo, se requiere una solución adecuada a un conjunto de recursos determinado. Tanto físicos: en qué ordenador va a funcionar la aplicación, de qué tipo de periféricos se dispone . . . , como lógicos: qué sistema operativo se usará, qué herramientas de desarrollo, qué bases de datos . . . Finalmente se diseñará un conjunto de algoritmos que resuelvan los distintos subproblemas en que se haya dividido el desarrollo.

Codificación

Consiste en la traducción de los algoritmos diseñados previamente, utilizando el lenguaje y entorno de desarrollo escogidos en la fase anterior. Será necesario realizar pruebas que garanticen al máximo la calidad de los programas desarrollados. Entre otras cosas, que estén libres de errores.

La documentación generada en esta fase junto con la de las fases anteriores será muy útil en el futuro para las eventuales actuaciones de mantenimiento.

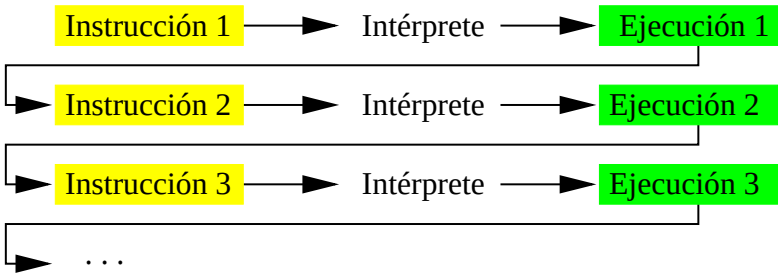


Figura 1.4: Fases en la interpretación de un programa

Explotación

Los diferentes programas desarrollados en la fase anterior se instalan en el entorno final de trabajo. Si es necesario se instalarán también otras herramientas de utilidad, necesarias para el correcto funcionamiento del sistema. Se debe proporcionar documentación, manuales de usuario, formación, etc.

Mantenimiento

En esta fase se realizarán correcciones al sistema desarrollado, bien para solventar errores no depurados, bien para cambiar o añadir nuevas funcionalidades requeridas por el cliente. Dependiendo de la importancia del caso, será necesario retomar el ciclo de vida a nivel de codificación, diseño o incluso análisis (ver Fig. 1.3).

Cuanto mejor se haya documentado el desarrollo en las primeras fases del ciclo de vida, menor será el tiempo necesario para llevar a cabo los distintos tipos de mantenimiento.

1.5 Traductores

Como ya se ha comentado, el único lenguaje directamente inteligible por el ordenador es el lenguaje máquina. Por ello, si se programa usando lenguajes de alto nivel será necesario algún programa traductor. Éste, a su vez, será el encargado de comprobar que los programas estén escritos correctamente, de acuerdo con la definición del lenguaje de programación empleado. Pueden distinguirse varios tipos de traductores:

1.5.1 Ensambladores

Los programas ensambladores son los encargados de traducir a lenguaje máquina los programas escritos en lenguaje ensamblador. La correspondencia entre ambos lenguajes es muy directa, por lo que los ensambladores suelen ser programas relativamente sencillos.

1.5.2 Intérpretes

El objetivo de un intérprete es procesar una a una las instrucciones de un programa escrito en un lenguaje de alto nivel. Para cada instrucción se verifica la sintaxis, se traduce a código máquina y finalmente se ejecuta. Es decir, que la traducción y la ejecución se realizan como una sola operación (ver Fig. 1.4).

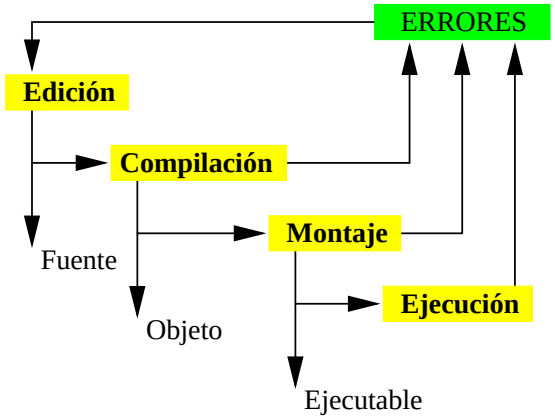


Figura 1.5: Fases en la compilación de un programa

La principal desventaja de los intérpretes es su lentitud para ejecutar los programas, pues es necesario verificar la sintaxis y realizar la traducción en cada ejecución.

1.5.3 Compiladores

La función de un compilador consiste en traducir un programa fuente escrito en un lenguaje de alto nivel a su equivalente en código máquina (también llamado *código objeto*).

Mientras que un intérprete traduce y ejecuta al mismo tiempo cada una de las instrucciones, un compilador analiza, traduce y posteriormente ejecuta todo el programa en fases completamente separadas (ver Fig. 1.5). Así pues, una vez se ha compilado un programa, no es necesario volverlo a compilar cada vez. Esto hace que la ejecución de un programa compilado sea mucho más rápida que la de uno interpretado.

El proceso de compilación

Edición Consiste en escribir el programa fuente usando el lenguaje de programación seleccionado y su grabación en un fichero. Para ello es necesario usar un programa editor, que puede o no formar parte del entorno de desarrollo.

Compilación En esta fase se verifica la sintaxis del programa fuente y se traduce el programa a código máquina (objeto). Si se producen errores, el compilador muestra información del tipo de error y dónde se ha producido.

Montaje Consistente en la combinación de los diferentes módulos objeto y librerías del lenguaje para crear un programa ejecutable. Esta fase se conoce también como *linkado*.

Ejecución En esta fase se invoca al programa de la manera adecuada dependiendo del sistema operativo sobre el que vaya a funcionar.

Como nota final, cabe decir que todo lenguaje de programación puede ser tanto interpretado como compilado. Sin embargo, dependiendo del las características del lenguaje y del uso mayoritario a que esté destinado, es normal asociar a cada lenguaje una forma de traducción particular. Por ejemplo, el lenguaje BASIC es mayoritariamente interpretado, mientras que C es compilado.

Capítulo 2

Primer contacto con C

2.1 Un poco de historia

El lenguaje de programación C fue desarrollado por Dennis Ritchie en los Laboratorios Bell de la empresa de comunicaciones AT&T, en 1972. C fue creado inicialmente con un propósito muy concreto: el diseño del sistema operativo UNIX. Sin embargo, pronto se reveló como un lenguaje muy potente y flexible, lo que provocó que su uso se extendiese rápidamente, incluso fuera de los Laboratorios Bell. De esta forma, programadores de todo el mundo empezaron a usar el lenguaje C para escribir programas de todo tipo.

Durante años, el estándar *de facto* del lenguaje C fue el definido en el libro *El lenguaje de programación C*, escrito por Brian Kernighan y Dennis Ritchie en 1978. Sin embargo, con el tiempo proliferaron distintas versiones de C, que con sutiles diferencias entre ellas, empezaron a causar problemas de incompatibilidad a los programadores. Con el fin de cambiar esta situación, el Instituto Nacional de Estándares Americano (más conocido como ANSI) creó un comité en 1983 para establecer una definición estándar de C que fuese no ambigua e independiente de la arquitectura interna de cualquier ordenador. Finalmente, en 1989 se estableció el estándar ANSI C. Actualmente, cualquier compilador moderno soporta ANSI C. Sin embargo, probablemente debido a razones comerciales, la opción por defecto en muchos compiladores de C es una versión propia del desarrollador del compilador. Dicha version suele ser ligeramente diferente e incompatible con el estándar ANSI C.

El lenguaje C debe su nombre a su predecesor, el lenguaje B desarrollado por Ken Thompson, también en los Laboratorios Bell.

2.2 Características del lenguaje

Actualmente existe gran variedad de lenguajes de programación de alto nivel entre los que elegir, como BASIC, Pascal, C, C++, Java, etc. Todos ellos pueden usarse para resolver la mayoría de proyectos de programación. Sin embargo, existen algunas razones que hacen de C el preferido de muchos programadores:

- Potencia y flexibilidad. Se ha usado en contextos tan dispares como el desarrollo de sistemas operativos, procesadores de texto, gráficos, bases de datos, compiladores de otros lenguajes, etc.

- Popularidad. Existe una gran variedad de compiladores, librerías, herramientas de apoyo a la programación, etc. Es el lenguaje predominante en el entorno UNIX.
- Portabilidad. El mismo programa escrito en C puede compilarse y ejecutarse sin prácticamente ningún cambio en diferentes ordenadores. Esto se debe en gran parte al estándar ANSI C.
- Sencillez. C utiliza pocas palabras clave, por lo que puede aprenderse fácilmente.
- Estructura y modularidad. Los programas en C pueden escribirse agrupando el código en funciones que a su vez se agrupan en distintos módulos. De esta forma, el código puede reutilizarse.

De acuerdo con esto, C representa una buena elección como lenguaje de programación. Sin embargo, seguro que el lector ha oído hablar de los lenguajes C++, Java y de la programación orientada a objetos, además de preguntarse sobre las diferencias entre C y C++. Pues bien, C++ puede verse como un superconjunto de C, lo que significa que casi cualquier aspecto de C es perfectamente válido en C++ (pero no al revés). Java por su parte, al igual que C++, también se basa en la sintaxis de C.

Finalmente, diremos que aunque C es considerado como un lenguaje de alto nivel, mantiene muchas características de los lenguajes de bajo nivel, por lo que podría clasificarse como de nivel bajo-medio.

2.3 Creación de un programa

La creación de un programa de ordenador consta generalmente de una serie de pasos claramente diferenciados.

Edición

El primer paso consiste en usar un editor de textos y crear un fichero que contenga el código del programa en C. Este código, normalmente llamado *código fuente*, servirá para dar instrucciones precisas al ordenador. Por ejemplo, la siguiente linea de código fuente en C indica al ordenador que debe mostrar el mensaje entre comillas en la pantalla:

```
printf( "Esto es un mensaje" );
```

El formato del texto admitido por la mayoría de compiladores se basa en el Código Estándar Americano para el Intercambio de Información (ASCII). La mayor parte de los procesadores de texto utilizan códigos especiales para dar formato a los documentos, por lo que normalmente no pueden ser usados como editores de programas.

Hoy en día, la mayoría de los entornos de programación incluyen un editor, sin embargo, otros no. En estos casos pueden usarse otros programas genéricos de edición de textos ASCII proporcionados por el sistema. Por ejemplo, en UNIX pueden usarse editores como `ed`, `ex`, `edit`, `vi`, `emacs`, o `nedit`, entre otros. En MS-Windows puede usarse el *Bloc de Notas*. En MS-DOS puede usarse `edit`. En OS/2, pueden usarse *E* y *EPM*.

El fichero fuente de un programa debe grabarse con un nombre. Normalmente, el nombre del fichero debe permitir intuir qué hace el programa. Adicionalmente, los ficheros fuente en C suelen tener la extensión `.c` para identificarlos fácilmente.

Compilación

Puesto que el ordenador es incapaz de entender directamente un lenguaje de alto nivel como C, antes de que un programa pueda ejecutarse en el ordenador debe traducirse a lenguaje máquina. Esta traducción la realiza un programa llamado compilador que, dado un fichero fuente, produce un fichero con las instrucciones de lenguaje máquina correspondientes al programa fuente original. El nuevo fichero recibe el nombre de fichero objeto.

El fichero objeto suele tener el mismo nombre que el fichero fuente, pero con la extensión `.OBJ` (o `.o` en UNIX).

Montaje

En el tercer paso, las diferentes partes del código compilado se combinan para crear el programa ejecutable.

Parte del lenguaje C consiste en una librería de funciones precompiladas que contiene código objeto. Las funciones en esta librería realizan operaciones de uso frecuente, como mostrar datos en pantalla o leer datos de un fichero. La función `printf` del ejemplo anterior es una función de dicha librería. Así pues, el fichero objeto producido al compilar el fichero fuente debe combinarse con el código objeto de la librería para crear el fichero del programa ejecutable.

Cabe destacar que en la mayoría de compiladores actuales, como los que funcionan en MS-DOS o MS-Windows, compilación y montaje se realizan como si fuesen una sola acción.

2.4 Primeros pasos con C

A continuación se muestra el programa más sencillo posible en C:

```
void main()
{
}
```

Todo programa en C debe tener una y sólo una función `main()`. Esta función deberá constar de una serie de sentencias (en este caso vacía) delimitada por los símbolos `{ }`. Dichas sentencias especifican la secuencia de acciones que el programa deberá llevar a cabo.

En C pueden ponerse comentarios en cualquier lugar del programa, utilizando los símbolos `/* */`. El compilador de C ignora todo el texto entre el inicio del comentario (`/*`) y el final del mismo (`*/`). Añadir comentarios a un programa en C no incrementa el tamaño de los ficheros objeto ni ejecutable, ni tampoco ralentiza la ejecución del programa. Veamos un ejemplo de programa con comentarios:

```
/* Mi primer programa en C */
void main()
{
   /* Otro comentario */
}
/* ...  y otro comentario */
```

Sin embargo, no es posible poner un comentario dentro de otro. Por ejemplo sería ilegal:

```
/* Mi primer programa en C */
void main()
{
   /* Otro comentario /* Comentario ilegal */ */
}
/* ...  y otro comentario */
```

Pero veamos un programa no tan simple. Por ejemplo, el siguiente programa usa la función `printf`, predefinida en la librería estándar `stdio.h`, para mostrar un mensaje en la pantalla.

```
/* Mi primer programa en C */
#include <stdio.h>
void main()
{
   printf( "Mi primer mensaje en pantalla \n" );
}
```

2.5 El modelo de compilación de C

A continuación se describe el modelo de compilación de C y los distintos procesos implicados: preprocesador, compilador y montador (ver Fig. 2.1).

Preprocesador

Aunque en el apéndice A se verá en detalle esta parte del proceso de compilación, seguidamente se describen algunos aspectos básicos.

El preprocesador toma como entrada el código fuente y es el responsable de eliminar los comentarios (ya que en realidad no representan ninguna instrucción) y de interpretar las directivas especiales del preprocesador, denotadas por el símbolo #. Por el momento destacaremos sólamente dos de las directivas más utilizadas:

- `#include`, que incluye un fichero externo dentro del fichero fuente. Se usarán los símbolos `< >` para indicar que el fichero se encuentra en un directorio del entorno de compilación, diferente del directorio de trabajo actual. Por el contrario, se usarán los símbolos `" "` para indicar fichero locales. Por ejemplo:
  - `#include <math.h>` incluye el fichero con las definiciones de las funciones matemáticas de la librería estándar.
  - `#include <stdio.h>` incluye el fichero con las definiciones de las funciones de entrada y salida de la librería estándar.
  - `#include "funciones.h"` incluye el fichero `funciones.h` del directorio actual.
- `#define`, que define un nombre simbólico. Cuando el preprocesador encuentra un nombre simbólico en el programa lo substituye por el valor que se le haya asociado con la directiva `#define`.
  - `#define NUM_ELEMENTOS 100` define la constante `NUM_ELEMENTOS` con valor 100.
  - `#define PI 3.1416` define la constante `PI`.

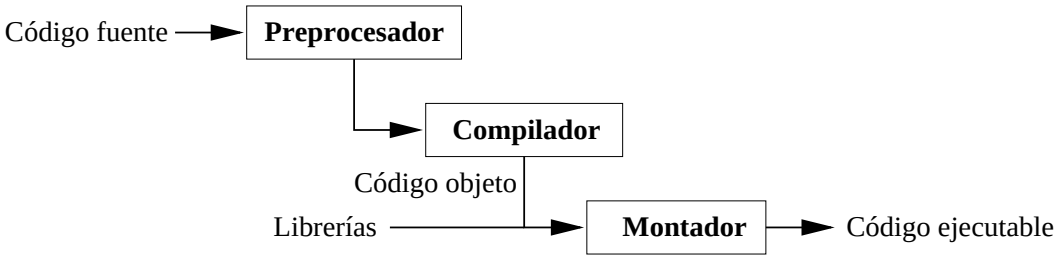


Figura 2.1: Modelo de compilación de C

Compilador

El compilador de C recibe el código fuente producido por el preprocesador y lo traduce a *código objeto* (ficheros con extensión `.OBJ` en MS-Windows, o extensión `.o` en UNIX).

Montador

Si un fichero fuente hace referencia a funciones de una librería (como la librería estándar) o a funciones definidas en otros ficheros fuente, el montador se encarga de:

- combinar todos los ficheros objeto correspondientes,
- verificar que sólo uno de ellos contenga la función principal `main()` y
- crear el fichero finalmente ejecutable.

Capítulo 3

Empezando a programar

3.1 Identificadores

Un *identificador* en un lenguaje de programación es un nombre utilizado para referir un valor constante, una variable, una estructura de datos compleja, o una función, dentro de un programa. Todo identificador está formado por una secuencia de letras, números y caracteres de subrayado, con la restricción de que siempre debe comenzar por una letra o un subrayado y que no puede contener espacios en blanco. Cada compilador fija un máximo para la longitud de los identificadores, siendo habitual un máximo de 32 caracteres.

C diferencia entre mayúsculas y minúsculas, según lo cual C considerará los identificadores `contador`, `Contador` y `CONTADOR`, por ejemplo, como diferentes.

En cualquier caso, nunca pueden utilizarse las *palabras reservadas* del lenguaje para la construcción de identificadores. De acuerdo con el estándar ANSI, C consta únicamente de 32 palabras reservadas (ver Tab. 3.1).

Tabla 3.1: Palabras reservadas de C

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

3.2 Estructura de un programa

Todo programa en C consta básicamente de los siguientes elementos:

- Directivas del preprocesador
- Definiciones de tipos de datos
- Declaraciones de funciones

Por el momento se tratará únicamente la declaración de la función `main()` que corresponde al programa principal. La ejecución de un programa escrito en C siempre comienza por dicha función. Por esta razón, en un programa sólo puede haber una función con dicho nombre. La definición de tipos de datos se deja para el capítulo 8.

Toda función en C, y en particular la función `main()`, tiene la siguiente estructura:

```
tipo_datos nombre_función ( parámetros )
{
   variables_locales;

   secuencia_de_sentencias;
}
```

No entraremos aquí en las particularidades de las funciones como el paso de parámetros y la devolución de resultados de un tipo de datos determinado (ver Cap. 10). Comentaremos simplemente que tanto la devolución de resultados como los parámetros son opcionales, y que en la mayoría de programas sencillos no se usan en la definición del programa principal.

A continuación se muestra, como ejemplo, un programa para evaluar la expresión $3 * 5 - 32/4$:

```
/* Evaluando una expresión */
#include <stdio.h>
void main()
{
   int a, b, c = 5;
   a = 3 * c;
   b = 32 / 4;
   c = a - b;
   printf( "El valor de la expresión es:  %d\n", c );
}
```

El *cuerpo* del programa principal lo constituyen todas las líneas de programa comprendidas entre los símbolos { y }. En cada una de dichas líneas puede haber una o más sentencias. Una sentencia es una orden completa para el ordenador. Toda sentencia debe acabar con un punto y coma (`;`).

3.3 Variables y constantes

Los programas de ordenador utilizan diferentes tipos de datos, por lo que requieren de algún mecanismo para almacenar el conjunto de valores usado. C ofrece dos posibilidades: *variables* y *constantes*. Una variable es un objeto donde se guarda un valor, el cual puede ser consultado y modificado durante la ejecución del programa. Por el contrario, una constante tiene un valor fijo que no puede ser modificado.

3.3.1 Variables

Toda variable debe declararse antes de ser usada por primera vez en el programa. Las sentencias de declaración de variables indican al compilador que debe reservar cierto espacio en la memoria del ordenador con el fin de almacenar un dato de tipo elemental o estructurado. Por ejemplo, la siguiente

declaración de variables indica al compilador que debe reservar espacio en la memoria para tres variables de tipo entero, a las que nos referiremos con los nombres `a`, `b` y `c`:

```
int a, b, c;
```

La declaración consiste en dar un nombre significativo a la variable e indicar el tipo de datos a que corresponden los valores que almacenará. A continuación se muestra la sintaxis más sencilla de una sentencia de declaración para una sola variable.

```
tipo_datos nombre_variable;
```

Además, en una sola sentencia pueden declararse varias variables de un mismo tipo de datos, separando los nombres de las variables mediante comas:

```
tipo_datos nombre_variable1, ..., nombre_variableN;
```

Opcionalmente, es posible asignar un valor inicial a las variables en la propia declaración.

```
tipo_datos nombre_variable = valor_inicial;
```

3.3.2 Constantes

C admite dos tipos diferentes de constantes: literales y simbólicas.

Constantes literales

Todo valor que aparece directamente en el código fuente cada vez que es necesario para una operación constituye una *constante literal*. En el siguiente ejemplo, los valores `20` y `3` son constantes literales del tipo de datos entero:

```
int cont = 20;
cont = cont + 3;
```

Si una constante numérica contiene un punto decimal, el compilador considera dicha constante como un valor real de coma flotante. Este tipo de constantes puede escribirse también utilizando alguna de las notaciones científicas comúnmente aceptadas (ver Sec. 6.3).

Por el contrario, el resto de constantes numéricas son consideradas por el compilador, como valores enteros. Pueden usarse tres formatos alternativos:

- Toda constante que comience por un dígito distinto de `0` es interpretada como un entero decimal (esto es, en base 10). Se especifican mediante los dígitos del 0 al 9 y el signo positivo o negativo.
- Si una constante comienza con el dígito `0`, se interpreta como un entero octal (base 8). Se especifican mediante los dígitos del 0 al 7 y el signo positivo o negativo.
- Finalmente, las constantes que comienzan por `0x` o `0X` se interpretan como enteros en base hexadecimal (base 16). Se especifican mediante los dígitos del 0 al 9, las letras de la A a la F, y el signo positivo o negativo.

Para saber más sobre los distintos sistemas de numeración, ver el apéndice C.

Constantes simbólicas

Una *constante simbólica* es una constante representada mediante un nombre (símbolo) en el programa. Al igual que las constantes literales, no pueden cambiar su valor. Sin embargo para usar el valor constante, se utiliza su nombre simbólico, de la misma forma que lo haríamos con una variable. Una constante simbólica se declara una sola vez, indicando el nombre y el valor que representa.

Las constantes simbólicas tienen dos ventajas claras respecto a las literales. Supongamos el siguiente código para calcular el perímetro de una circunferencia y el área del círculo que define:

```
perimetro = 2 * 3.14 * radio;
area = 3.14 * radio * radio;
```

Si por el contrario se hubiese definido una constante simbólica de nombre `PI` y valor `3.14`, podríamos escribir un código mucho más claro:

```
perimetro = 2 * PI * radio;
area = PI * radio * radio;
```

Es más, imaginemos ahora que para incrementar la precisión del cálculo se desea usar un valor más preciso de la constante π, como `3.14159`. En el primer caso debería substituirse uno a uno el valor `3.14` en todo el programa. En el segundo caso, bastaría cambiar la definición de la constante `PI` con el nuevo valor.

El método más habitual para definir constantes en C es la directiva del preprocesador `#define`. Por ejemplo, en el caso anterior podríamos haber escrito:

```
#define PI 3.14159
```

Es decir, el nombre simbólico y a continuación el valor constante que representa.

3.3.3 Entrada y salida de valores

Aunque en el apéndice B se verán con más detalle las funciones de la librería estándar `printf` y `scanf`, se introducen en este punto con el fin de poder realizar algunos programas sencillos.

C utiliza operaciones de entrada y salida con formato. Por ejemplo, la función `printf` usa como carácter especial de formato el símbolo de porcentaje (`%`). El carácter que sigue a este símbolo define el formato de un valor (constante, variable o expresión). Por ejemplo, `%c` para valores de tipo carácter o `%d` para valores de tipo entero. El siguiente ejemplo muestra por pantalla el contenido de una variable de tipo carácter (`ch`), y una variable entera (`num`).

```
char ch;
int num;
.  .  .
printf( "Esto es un carácter:  %c\n", ch );
printf( "Y esto un entero:  %d\n", num );
```

El formato es todo aquello especificado entre las comillas, al que le sigue una lista de variables, constantes o expresiones separadas por comas. Es responsabilidad del programador asegurar la perfecta

Tabla 3.2: Operadores aritméticos en C

Unarios	**Signo negativo**	$-$
	Incremento	$++$
	Decremento	$--$
Binarios	**Suma**	$+$
	Resta	$-$
	Multiplicación	$*$
	División	$/$
	Módulo	$\%$

correspondencia entre el formato y la lista de valores, tanto en número como en el tipo de los mismos. Finalmente, la secuencia especial `\n` indica un salto de línea.

Por su parte, `scanf` es una función para la entrada de valores a una estructura de datos, y en particular a una variable. Su formato es similar al de `printf`. Por ejemplo:

```
char ch;
int num;
.  .  .
scanf( "%c%d", &ch, &num );
```

Permite introducir desde el teclado un carácter en la variable `ch` y seguidamente un valor entero en la variable `num`. Nótese que en el caso de `scanf` se antepone el símbolo `&` a las variables. Por el momento, no debemos olvidar utilizarlo, y tengamos en mente que el uso de `&` tiene que ver con direcciones de memoria y punteros (ver Cap. 9).

3.4 Expresiones

Una expresión es una fórmula matemática cuya evaluación especifica un valor. Los elementos que constituyen una expresión son: constantes, variables y operadores.

3.4.1 Operador de asignación

El operador de asignación permite asignar valores a las variables. Su símbolo es un signo igual `=`. Este operador asigna a la variable que está a la izquierda del operador el valor que está a la derecha. Un ejemplo de expresiones válidas con el operador de asignación son: `x = 1; z = 1.35;`.

3.4.2 Operadores aritméticos

Además de los operadores aritméticos tradicionales, C proporciona algunos operadores adicionales (ver Tab. 3.2).

La expresión, `x++;` equivale a `x = x+1;`, y `x--;` equivale a `x = x-1;`. Aunque en el pasado algunos compiladores generaban código más eficiente si se usaba los operadores `++`, `--` en lugar de sus expresiones equivalentes, esto ya no es cierto en los compiladores modernos. Los operadores `++` y `--` pueden usarse tanto de manera postfija (más habitual) como prefija, indicando en cada

caso si el valor de la variable se modifica después o antes de la evaluación de la expresión en la que aparece. Por ejemplo, la siguiente línea de código:

```
x = ((++z) - (w--)) % 100;
```

es equivalente al siguiente grupo de sentencias:

```
z = z + 1;
x = (z - w) % 100;
w = w - 1;
```

Nótese que en C no existe ningún operador especial para la división entera, de forma que cuando los dos operandos de la división son enteros, el cociente que se obtiene es el correspondiente a la división entera (el cociente no se redondea, sino que se trunca). Si alguno de los operandos es un valor real, el resultado de la división será también real. Por ejemplo, `x = 3/2;` asigna el valor `1` a la variable `x` (que debe ser entera), mientras que `x = 3.0/2;` o `x = 3/2.0;` asigna el valor `1.5` a la variable `x` (que debe ser real). Finalmente, el operador de módulo (`%`) permite obtener el resto de una división entera, por lo que sus operandos deben ser también enteros. Por ejemplo, `x = 8 % 5;` asigna el valor `3` a la variable entera `x`.

Existe además una manera abreviada de expresar ciertos cálculos en C. Es muy común tener expresiones del estilo de `i = i + 5;` o `x = x * (y + 2);`. Este tipo de expresiones puede escribirse en C de forma compacta como:

```
expresión1  op =  expresión2
```

que es equivalente a:

```
expresión1 = expresión1  op  expresión2
```

Según esto, la asignación `i = i + 5;` puede reescribirse como `i += 5;` y la asignación `x = x * (y + 2);` como `x *= y + 2;`. Nótese que esta última expresión no significa en ningún caso `x = (x * y) + 2;`.

Como puede verse, un uso abusivo de los operadores abreviados de C puede hacer difícil de leer un programa. C permite escribir expresiones muy compactas, pero que pueden generar confusión al ser leídas. Hay que recordar siempre que la legibilidad (mantenimiento) de un programa es tan importante como su correcto funcionamiento.

3.4.3 Operadores relacionales

Los operadores relacionales se utilizan principalmente para elaborar condiciones en las sentencias condicionales e iterativas (ver Cap. 4 y Cap. 5).

La tabla 3.3 resume los distintos operadores de relación en C. Al relacionar (comparar) dos expresiones mediante uno de estos operadores se obtiene un resultado lógico, es decir: 'CIERTO' o 'FALSO'. Por ejemplo, la expresión `4 > 8` da como resultado el valor falso, la expresión `num == num` da como resultado cierto, la expresión `8 <= 4` da como resultado falso, etc.

Es interesante destacar que a diferencia de otros lenguajes, C no dispone de un tipo de datos específico para los valores lógicos o booleanos. En su lugar, C representa un resultado 'FALSO' como

Tabla 3.3: Operadores relacionales (izquierda) y lógicos (derecha) en C

Menor que	$<$
Mayor que	$>$
Menor o igual que	$<=$
Mayor o igual que	$>=$
Igual que	$==$
Distinto que	$!=$

Conjunción ó Y lógico	**&&**
Disyunción u O lógico	**||**
Negación ó NO lógico	**!**

Tabla 3.4: Tabla de verdad de los operadores lógicos en C

A	B	! A	A && B	A || B
Cierto	**Cierto**	**Falso**	**Cierto**	**Cierto**
Cierto	**Falso**	**Falso**	**Falso**	**Cierto**
Falso	**Cierto**	**Cierto**	**Falso**	**Cierto**
Falso	**Falso**	**Cierto**	**Falso**	**Falso**

el valor numérico entero cero, y un resultado 'CIERTO' como cualquier valor entero diferente de cero. Es muy importante recordar este hecho de ahora en adelante.

Un error habitual es confundir el operador relacional de igualdad `==` con el operador de asignación `=`. Por ejemplo, la sentencia `x = 3` asigna el valor `3` a la variable `x`, mientras que `x == 3` compara el valor de `x` con la constante `3`.

3.4.4 Operadores lógicos

Los operadores lógicos (ver Tab. 3.3) se utilizan principalmente en conjunción con los relacionales para elaborar condiciones complejas en las sentencias condicionales e iterativas (ver Cap. 4 y Cap. 5).

Es importante no confundir los operadores lógicos `&&` y `||` con los operadores de manejo de bits `&` y `|` que veremos en la sección 8.5.

La tabla 3.4 muestra la *tabla de verdad* para los operadores lógicos. De acuerdo con dicha tabla, las expresiones `4 && 0`, `!(4 > 1)` y `5 <= 0` dan como resultado `0` (falso), mientras que las expresiones `4 || 9`, `(8 == 4*2) && (5 > 2)` y `2 && (4 < 9)` dan como resultado `1` (cierto).

3.4.5 Prioridad de operadores

La tabla 3.5 muestra los operadores vistos anteriormente, así como la prioridad entre ellos. La prioridad desciende al descender en la tabla. También se muestra la asociatividad para operadores con el mismo nivel de prioridad.

Por ejemplo, según dicha tabla, la expresión `(a < 10 && 2 * b < c)` se interpretará como `(a < 10) && ((2 * b) < c)`.

Tabla 3.5: Prioridad y asociatividad de los operadores en C

Operador	Símbolo	Asociatividad
Paréntesis	()	**Izquierda a derecha**
NO lógico	**!**	**Derecha a izquierda**
Signo negativo	−	
Incremento	++	
Decremento	−−	
Multiplicación	*	**Izquierda a derecha**
División	/	
Módulo	**%**	
Suma	+	**Izquierda a derecha**
Resta	−	
Menor que	<	**Izquierda a derecha**
Menor o igual que	<=	
Mayor que	>	
Mayor o igual que	>=	
Igual que	==	**Izquierda a derecha**
Distinto que	!=	
Y lógico	**&&**	**Izquierda a derecha**
O lógico	||	**Izquierda a derecha**
Asignaciones	= += −=	**Derecha a izquierda**
	*= /= **%**=	

3.5 Ejercicios

Escribir un programa para cada uno de los siguientes ejercicios:

1. Pedir la base y la altura de un rectángulo, calcular su área y su perímetro, y mostrar los resultados por pantalla.

2. Pedir una cantidad de segundos y mostrar por pantalla a cuántas horas, minutos y segundos corresponden.

3. Suponiendo que previamente se ha realizado la declaración `int x = 7, y;` , calcular el valor de la variable `y` tras evaluar cada una de las siguientes sentencias de asignación:

   (a) `y = -2 + --x;`

   (b) `y += 2;`

   (c) `y = (y == x);`

   (d) `y = y++ - x;`

4. Evaluar las siguientes expresiones:

   (a) `5 / 2 + 20 % 6`

   (b) `4 * 6 / 2 - 15 / 2`

(c) `5 * 15 / 2 / (4 - 2)`

(d) `8 == 16 || 7 != 4 && 4 < 1`

(e) `(4 * 3 < 6 || 3 > 5 - 2) && 3 + 2 < 12`

Capítulo 4

Construcciones condicionales

Una de las construcciones importantes que pueden especificarse en un programa es el hecho de realizar diferentes tareas en función de ciertas condiciones. Esto es, ejecutar una parte del código u otra, condicionalmente. Para ello será necesario especificar dichas condiciones (ver Sec. 3.4) y disponer de un mecanismo para indicar qué acciones tomar dependiendo de cómo se evalúe una determinada condición en un momento dado de la ejecución del programa.

Antes de empezar, un recordatorio. Como ya de comentó en la sección 3.4.3, C no dispone de valores booleanos o lógicos, que podrían usarse en la evaluación de condiciones. En su defecto, C "simula" los valores falso y cierto, como el valor numérico cero, y cualquier valor no cero (incluyendo negativos), respectivamente.

Así pues, en este capítulo veremos las distintas maneras que C ofrece para controlar el flujo de ejecución de un programa de forma condicional, que son:

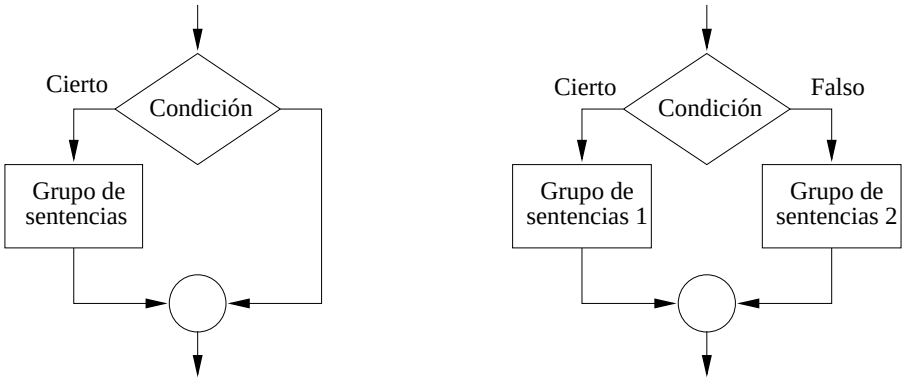
- la construcción `if`,
- el operador condicional `?`, y
- la construcción `switch`.

4.1 Construcción `if`

La construcción `if` es similar a la existente en otros lenguajes de programación, aunque en C posee ciertas peculiaridades. El formato general de esta construcción para decidir si una determinada sentencia debe ejecutarse o no (*alternativa simple*) es el siguiente:

```
if (condición)
   sentencia;
```

La construcción `if` puede escribirse también de forma más general para controlar la ejecución de un grupo de sentencias, de la siguiente manera:



(a) (b)

Figura 4.1: Esquema de funcionamiento de `if` *y de* `if-else`

```
if (condición)
{
   sentencia_1;
   sentencia_2;
   .  .  .
   sentencia_N;
}
```

El funcionamiento de la construcción `if` es muy simple. En primer lugar se evalúa la condición, que no es otra cosa que una expresión de tipo entero. A continuación, si la expresión se ha evaluado como cierta, se ejecuta la sentencia o grupo de sentencias. En caso contrario la ejecución del programa continúa por la siguiente sentencia en orden secuencial (ver Fig. 4.1 (a)).

El siguiente ejemplo muestra el uso de la construcción `if`. El programa lee un número entero y lo transforma en el impar inmediatamente mayor, si es que no era ya impar.

```
#include <stdio.h>
void main()
{
   int a;

   scanf("%d", &a);
   if (a % 2 == 0) /* Comprobar si a es par.  */
      a = a + 1;
   printf( "Ahora es impar:  %d\n", a );
}
```

Nótese que después de la condición no se escribe '`;`'. Escribir '`;`' detrás de la condición equivaldría a que la construcción `if` ejectutase un conjunto vacío de sentencias, lo cual no tiene ningún sentido. Nótese, sin embargo, que tal hecho es válido sintácticamente (no produce ningún error de compilación), por lo que deberá tenerse cuidado al escribir esta construcción. Algo similar ocurre con los bucles `for` y `while` (ver Cap. 5).

4.1.1 Variante `if-else`

Existe otra forma más general, denominada *alternativa doble*, que ofrece dos alternativas de ejecución, en función de si la condición se evalúa cierta o falsa.

```
if (condición)
   sentencia_1;
else
   sentencia_2;
```

Y para un grupo de sentencias:

```
if (condición)
{
   grupo_de_sentencias_1;
}
else
{
   grupo_de_sentencias_2;
}
```

Así pues, si la condición es cierta se ejecutará la primera sentencia (el primer grupo de sentencias), y si es falsa se ejecutará la segunda sentencia (el segundo grupo). Ver figura 4.1 (b).

El siguiente programa muestra el uso de esta construcción. El programa calcula el máximo de dos números enteros:

```
#include <stdio.h>
void main()
{
   int a, b, max;

   scanf( "%d %d", &a, &b );
   if (a > b)
      max = a;
   else
      max = b;
   prinf( "El máximo es:  %d\n", max );
}
```

Es importante destacar que la sentencia en la construcción `else` es opcional, es decir, puede ser nula. Veámoslo en el siguiente ejemplo que determina si un número es par:

```
#include <stdio.h>
void main()
{
   int x;
```

```
    scanf( "%d", &x );
    if (x % 2 == 0)
       printf( "Es par.\n" );
    else ;
}
```

El hecho de que la construcción `else` sea opcional puede causar problemas de ambigüedad al compilador cuando se utilizan construcciones `if` o `if-else` anidadas. Para solventar el problema se ha establecido una regla muy sencilla que todo compilador de C tiene en cuenta. La regla consiste en que una sentencia `else` se asocia con el `if` precedente más cercano siempre y cuando éste no tenga ya asociada otra sentencia `else`.

A continuación se muestran dos porciones de programa prácticamente iguales, pero con comportamientos completamente diferentes. Se deja para el lector el análisis de ambos casos.

```
.  .  .
if (n > 0)
   if (a > b)
      z = a;
   else
      z = b;
.  .  .
```

```
.  .  .
if (n > 0)
{
   if (a > b)
      z = a;
}
else
   z = b;
.  .  .
```

4.1.2 Variante `if-else-if`

Existe finalmente otra construcción alternativa muy común, conocida como `if-else-if` o simplemente `else-if`. Su construcción, donde las condiciones se plantean de forma escalonada, se muestra a continuación:

```
if (condición_1)
{
   grupo_de_sentencias_1;
}
else if (condición_2)
{
   grupo_de_sentencias_2;
}
.  .  .
else if (condición_N)
{
   grupo_de_sentencias_N;
}
else
{
   grupo_de_sentencias_por_defecto;
}
```

Las condiciones se evalúan secuencialmente de arriba hacia abajo hasta encontrar una que dé como resultado cierto. En ese punto, se ejecuta el grupo de sentencias correspondiente a dicha condición. El resto de condiciones y sentencias asociadas se ignoran. En caso de que ninguna de las condiciones se evalúe cierta, se ejecutaría el grupo de sentencias por defecto. Como en todos los casos anteriores, el último `else` es opcional.

A continuación se muestra un ejemplo del uso de esta construcción:

```
#include <stdio.h>
void main()
{
   int hora;

   scanf( "%d", &hora );
   if ((hora >= 0) && (hora < 12))
      printf( "Buenos días" );
   else if ((hora >= 12) && (hora < 18))
      printf( "Buenas tardes" );
   else if ((hora >= 18) && (hora < 24))
      printf( "Buenas noches" );
   else
      printf( "Hora no válida" );
}
```

4.2 El operador condicional `?`

El operador condicional `?` es el único operador ternario de C. La forma general de las expresiones construidas con este operador es la siguiente:

```
expresión_1 ?  expresión_2 :  expresión_3;
```

De manera que si la primera expresión se evalúa cierta, toda la expresión toma el valor de la segunda expresión. En cambio, si la primera expresión se evalúa falsa, toda la expresión toma el valor de la tercera expresión.

Un ejemplo típico del uso de este operador es el cálculo del máximo de dos valores. En la siguiente sentencia, `c` toma el valor del máximo entre la variable `a` y `b`.

```
c = (a > b) ?  a :  b;
```

Esto mismo podría haberse escrito usando la construcción `if-else` como:

```
if (a > b)
   c = a;
else
   c = b;
```

De esta manera, algunas sentencias `if-else` sencillas pueden escribirse de manera muy compacta mediante el operador `?`.

Finalmente, el operador condicional, por ser en realidad un operador para expresiones, puede usarse en lugares donde no puede usarse un `if-else`, como se muestra a continuación:

```
printf("El mínimo es %d \n", ((x < y) ?  x :  y) );
```

4.3 Construcción `switch`

Esta construcción permite especificar múltiples sentencias al estilo `if-else-if`, pero de manera más compacta, legible y elegante. Su forma general es la siguiente:

```
switch ( expresión )
{
   case constante_1 :
      grupo_de_sentencias_1;
      break;
   case constante_2 :
      grupo_de_sentencias_2;
      break;
   .  .  .
   case constante_N :
      grupo_de_sentencias_N;
      break;
   default :
      grupo_de_sentencias_por_defecto;
      break;
}
```

donde la expresión debe ser de tipo entero o carácter, al igual que todas las constantes asociadas a cada etiqueta `case`. Es importante resaltar que no pueden usarse variables o expresiones en los distintos `case`, sino sólo constantes.

El funcionamiento de la construcción `switch` es como sigue. En primer lugar se evalúa la expresión. Seguidamente su valor es comparado secuencialmente con el de las diferentes constantes en los `case`. Si el valor de la expresión coincide con alguna de ellas, se ejecuta el grupo de sentencias correspondiente y `switch` concluye gracias a la sentencia `break`. En caso contrario, y si existe el caso `default` (que es opcional), se ejecutaría el grupo de sentencias por defecto (ver Fig. 4.2).

Cabe mencionar de forma especial, la sentencia `break` que volveremos a ver en capítulos sucesivos. En general, `break` se utiliza para finalizar de forma forzada la ejecución dentro de un bloque de código, de manera que la siguiente sentencia a ejecutar será la primera sentencia justo después de dicho bloque. En la construcción `switch`, `break` es necesario para concluir la ejecución del grupo de sentencias asociado al caso cuya constante coincide con el valor de la expresión. Así pues, la sentencia a ejecutar después de `break` en un `switch`, será la primera sentencia posterior a la llave } que cierra el `switch`.

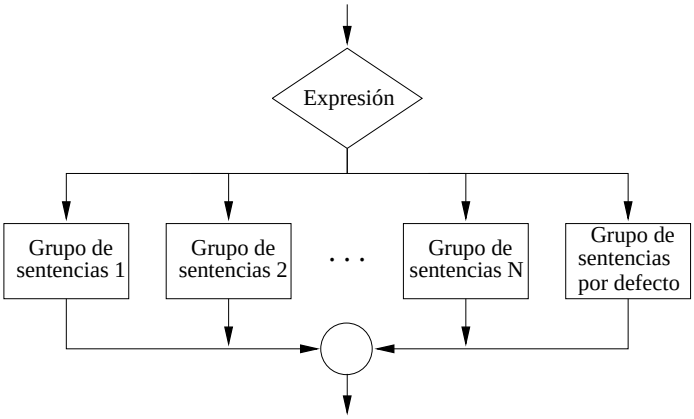


Figura 4.2: Esquema de funcionamiento de `switch`

La construcción `switch` también podría escribirse de forma equivalente mediante sentencias del tipo `if-else-if`, de la siguiente forma:

```
if (expresión == constante_1)
{
   grupo_de_sentencias_1;
}
else if (expresión == constante_2)
{
   grupo_de_sentencias_2;
} .  .  .
else if (expresión == constante_N)
{
   grupo_de_sentencias_N;
}
else
{
   grupo_de_sentencias_por_defecto;
}
```

que, como puede verse, es mucho más ineficiente en tiempo de ejecución, puesto que la expresión debe evaluarse repetidas veces, una para cada condición.

El siguiente ejemplo muestra un programa que hace uso de `switch` para traducir a caracteres un dígito entre 1 y 5.

```
#include <stdio.h>
void main()
{
   int num;

   scanf( "%d", &num );
   switch ( num )
```

```
    {
       case 1 :
          printf( "Uno.\n" );
          break;

       case 2 :
          printf( "Dos.\n" );
          break;
       .  .  .
       case 5 :
          printf( "Cinco.\n" );
          break;
       default :
          printf( "El dígito está fuera de rango.\n" );
          break;
    }
}
```

Finalmente, cabe decir que el grupo de sentencias asociado a un `case` puede ser vacío. Este caso particular tiene su utilidad cuando se desea que varias etiquetas `case` ejecuten un mismo grupo de sentencias. Por ejemplo:

```
#include <stdio.h>
void main()
{
   int num;

   scanf( "%d", &num );
   switch ( num )
   {
      case 1:
      case 3:
      case 7:
         printf( "Es un uno, un tres o un siete.\n" );
         break;
      case 4:
      case 8:
         printf( "Es un cuatro, o un ocho.\n" );
         break;
      default:
         printf( "Dígito no controlado.\n" );
         break;
   }
}
```

4.4 Ejercicios

1. Escribir un programa que lea tres valores enteros y muestre por pantalla el máximo y el mínimo de ellos.

2. Dado el siguiente programa, realizar un seguimiento de la ejecución en los siguientes supuestos:

   (a) `a = 0, b = 0, c = 5, d = 3`

   (b) `a = 2, b = 1, c = 5, d = 3`

   (c) `a = 2, b = 1, c = 2, d = 2`

   (d) `a = 2, b = 1, c = 0, d = 0`

```
#include <stdio.h>
void main()
{
   int a, b, c, d;

   scanf( "%d %d %d %d", &a, &b, &c, &d );
   if ( ((a > 0) || (b > a)) && (c != d) )
   {
      a = c;
      b = 0;
   }
   else
   {
      c += d;
      c = (c == 0) ?  (c + b) :  (c - a);
      b = a + c + d;
   }
   printf( "%d %d %d %d\n", a, b, c, d );
}
```

3. Escribir un programa que lea un valor entero y determine si es múltiplo de 2 y de 5.

4. Escribir un programa que muestre por pantalla la ecuación de una recta en un plano, $Ax + By + C = 0$, leyendo previamente las coordenadas de dos de sus puntos (x_1, y_1) y (x_2, y_2). Recordar que:

$$A = y_2 - y_1 \quad y \quad B = y_1 * (x_2 - x_1) - x_1 * (y_2 - y_1)$$

Capítulo 5

Construcciones iterativas

Hasta ahora hemos visto algunos aspectos básicos del control del flujo de ejecución de un programa en C. Este capítulo presenta los mecanismos que C ofrece para la ejecución repetida de sentencias, bien un número prefijado de veces, bien dependiendo de cierta condición. Es decir, mecanismos para la creación de *bucles* de ejecución.

C proporciona las siguientes construcciones iterativas:

- la construcción `while`,
- la construcción `do-while`, y
- la construcción `for`.

5.1 Construcción `while`

La construcción `while` es similar a la existente en otros lenguajes de programación. Sin embargo, debido a que en C toda sentencia puede considerarse como una expresión, la construcción `while` de C ofrece cierta potencia añadida.

La forma más general para la ejecución repetida de una sola sentencia es:

```
while (condición)
   sentencia;
```

O para la ejecución repetida de un grupo de sentencias:

```
while (condición)
{
   grupo_de_sentencias;
}
```

El funcionamiento de esta construcción es bastante simple. El *cuerpo del bucle*, es decir, la sentencia o grupo de sentencias dentro del bucle, se ejecuta mientras el valor de la expresión que actúa de condición sea cierto. En el momento en que la condición sea falsa, la ejecución del programa continúa secuencialmente con la siguiente instrucción tras el bucle (ver Fig. 5.1).

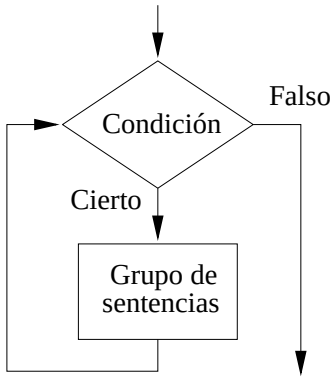


Figura 5.1: Esquema de funcionamiento de `while`

El siguiente ejemplo calcula la media de una secuencia de números enteros leídos por teclado acabada en -1:

```
#include <stdio.h>
void main()
{
   int num, cont, suma;

   cont = 0;
   suma = 0;
   scanf( "%d", &num );
   while (num != -1)
   {
      cont++;
      suma = suma + num;
      scanf( "%d", &num );
   }
   if (cont != 0)
      printf( "La media es %d\n", sum/cont );
   else
      printf( "La secuencia es vacía.\n" );
}
```

En la construcción `while` la condición se evalúa al principio del bucle. Por ello, si cuando se alcanza el bucle por primera vez, la condición es falsa, el cuerpo del bucle no se ejecuta nunca (imagínese el caso, en el ejemplo anterior, en que el primer número de la secuencia sea -1). Como consecuencia, el cuerpo de un bucle `while` puede ejecutarse entre 0 y N veces, donde N depende de la condición.

Nótese también que si la condición permanece cierta, el bucle puede no terminar nunca (imagínese qué ocurriría si se elimina la sentencia `scanf` del cuerpo del bucle del ejemplo anterior). Por ello, habitualmente las sentencias del cuerpo del bucle modifican las variables que aparecen en la condición, de forma que ésta sea falsa en algún momento.

Por otra parte, la condición del bucle (y esto es extensible a las diferentes construcciones repetitivas)

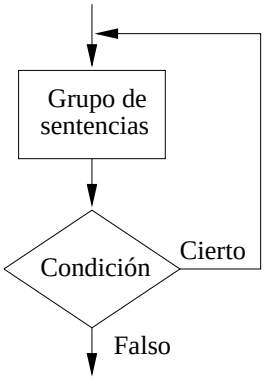


Figura 5.2: Esquema de funcionamiento de `do-while`

no tiene por qué ser simplemente una expresión lógica, sino que puede ser cualquier expresión. Por ejemplo, los siguiente bucles

```
while (x--) { ... }

while (x = x+1);
```

son perfectamente válidos. En ambos casos el cuerpo del bucle se repetirá mientras el valor de `x` sea distinto de 0. Nótese que en el segundo caso el cuerpo del bucle es nulo, lo cual también es posible.

5.2 Construcción `do-while`

La forma general de la construcción `do-while` es la siguiente:

```
do
{
   sentencia;   o   grupo_de_sentencias;
} while (condición);
```

Nótese que tanto para ejecutar una sola sentencia como para ejecutar un grupo de ellas, las llaves { } son igualmente necesarias.

Esta construcción funciona de manera muy similar a la construcción `while`. Sin embargo, al contrario que ésta, `do-while` ejecuta primero el cuerpo del bucle y después evalúa la condición. Por lo cual, el cuerpo del bucle se ejecuta como mínimo 1 vez (ver Fig. 5.2).

El siguiente ejemplo cuenta el número de veces que aparece el número 3 en una secuencia de números enteros acabada en -1:

```
#include <stdio.h>
void main()
{
   int num, cont;
```

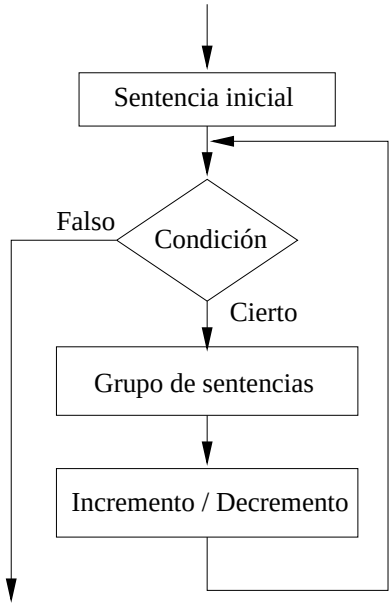


Figura 5.3: Esquema de funcionamiento de `for`

```
    cont = 0;

    do
    {
       scanf( "%d", &num );
       if (num == 3)
          cont++;
    } while (num != -1);
    printf( "El 3 ha aparecido %d veces\n", cont );
}
```

Es importante destacar el uso de '`;`' después de la condición, a diferencia de en la construcción `while` , donde no se utiliza.

Finalmente, cabe decir que tradicionalmente, tanto la construcción `while` como la construcción `do-while` se utilizan en bucles donde se desconoce *a priori* el número exacto de iteraciones.

5.3 Construcción `for`

Esta construcción iterativa no presenta un formato fijo estricto, sino que admite numerosas variantes, lo que la dota de gran potencia y flexibilidad.

Su forma más general para la ejecución repetida de una sola sentencia es:

```
for (sentencia_inicial ; condición ; incremento/decremento)
   sentencia;
```

o para la ejecución repetida de un grupo de sentencias:

```
for (sentencia_inicial ; condición ; incremento/decremento)
{
   grupo_de_sentencias;
}
```

La primera parte de la construcción `for` acostumbra a ser una sentencia de asignación donde se inicializa alguna variable que controla el número de veces que debe ejecutarse el cuerpo del bucle. Esta sentencia se ejecuta una sola ocasión, antes de entrar por primera vez al cuerpo del bucle.

La segunda parte corresponde a la condición que indica cuándo finaliza el bucle, de la misma forma que en las construcciones iterativas anteriores. En este caso, la condición se evalúa antes de ejecutar el cuerpo del bucle, por lo que al igual que en la construcción `while`, el cuerpo puede ejecutarse entre 0 y N veces, donde N depende de la condición.

La tercera parte corresponde normalmente a una sentencia de incremento o decremento sobre la variable de control del bucle. Esta sentencia se ejecuta siempre después de la ejecución del cuerpo del bucle.

La figura 5.3 muestra esquemáticamente el funcionamiento del bucle `for`.

El programa del siguiente ejemplo utiliza la construcción `for` para calcular el sumatorio $\sum_{i=1}^{10} i^3$:

```
#include <stdio.h>
void main()
{
   int i, cubo, suma;

   suma = 0;
   for (i = 0 ; i <= 10 ; i++)
   {
      cubo = i * i * i;
      suma += cubo;
   }     printf( "El sumatorio es %d\n", suma );
}
```

Las tres partes de la construcción `for` son opcionales, por lo que es posible omitir alguna o todas ellas. En cualquier caso, los punto y coma (`;`) separadores son siempre necesarios. Un ejemplo clásico de este tipo de bucle `for` es el bucle infinito (nunca concluye la ejecución):

```
for ( ; 1 ; )
{
   /* Grupo de sentencias */
}
```

Tradicionalmente la construcción `for` se utiliza en bucles donde el número exacto de iteraciones es conocido *a priori*, y puede controlarse mediante una variable que actúa como contador.

5.3.1 El operador coma (,)

C permite la utilización de más de una sentencia en la primera y tercera partes de la construcción `for`, así como de más de una condición en la segunda parte. Por ejemplo, el siguiente bucle es válido en C:

```
for (i = 0, j = 10 ; i < 10, j > 0 ; i++, j-=2)
{
   /* Grupo de sentencias */
}
```

Así pues, las variables `i` y `j` se inicializan a 0 y 10, respectivamente, antes de comenzar la ejecución del bucle. En la segunda parte de la construcción, aparecen dos condiciones, `i < 10` y `j > 0`. Si alguna de ellas es falsa, la ejecución del bucle se detiene. Finalmente, tras ejecutarse el cuerpo del bucle, `i` se incrementa en 1 y `j` se decrementa en 2, tras lo cual vuelven a comprobarse las condiciones, y así sucesivamente.

5.3.2 Equivalencia `for-while`

Como se ha podido ver, C trata el bucle `for` de manera muy similar al bucle `while`, usando una condición para decidir cuándo concluye la ejecución. De hecho, todo bucle `for` puede escribirse de forma equivalente mediante un bucle `while` de la siguiente forma:

```
sentencia_inicial;
while (condición)
{
   sentencia;
   incremento/decremento;
}
```

5.4 Las sentencias `break` y `continue`

C proporciona dos mecanismos para alterar la ejecución de las construcciones iterativas: las sentencias `break` y `continue`.

`break`

Esta sentencia tiene una doble finalidad. Por un lado, indica el final de un `case` en la construcción `switch`, como ya se vió en la sección 4.3. Y por otro, para forzar la terminación inmediata de la ejecución de un bucle. De esta forma, se permite salir de la construcción repetitiva ignorando la evaluación de la condición. Si bien su uso está reconocido como no muy elegante, permite en ocasiones escribir programas más legibles y compactos.

`continue`

Esta sentencia se utiliza únicamente en las construcciones repetitivas. Su función es la de evitar que se ejecute todo el código a continuación de ella y hasta el final del cuerpo del bucle, durante una iteración determinada.

El siguiente ejemplo pretende ilustrar el uso de estas sentencias:

```
do
{
   scanf("%d", &num);
   if (num < 0)
   {
      printf( "Valor ilegal\n" );
      break; /* Abandonar el bucle.  */
   }
   if (num > 100)
   {
      printf( "Valor demasiado grande\n" );
      continue; /* No ejecutar el resto de sentencias
                   e ir al final del bucle.  */
   }

   /* Procesar el valor leído */
   .  .  .
} while (num != 0 );
.  .  .
```

5.5 Ejercicios

Se recomienda realizar los siguientes ejercicios utilizando las diferentes construcciones iterativas presentadas.

1. Escribir un programa que calcule la suma de los 20 primeros números múltiplos de 5 o de 7.
2. Escribir un programa que calcule la potencia de un número entero, dado su valor y el del exponente.
3. Escribir un programa que lea N números enteros y muestre el mayor y el menor de todos ellos.
4. Escribir un programa que escriba la tabla de multiplicar de un número leído por teclado.
5. Escribir un programa que muestre la serie de Fibonacci hasta un límite dado. Recordar que la serie de Fibonacci se define como

$$F_0 = 1 \qquad F_1 = 1 \qquad F_i = F_{i-1} + F_{i-2}$$

6. Escribir un programa que convierta un número entero positivo a cualquier base de numeración dada, igual o inferior a 10.
7. Escribir un programa que determine si un número entero dado es primo o no.
8. Escribir un programa que calcule el factorial de un número entero leído por teclado.
9. Escribir un programa que calcule la suma de todos los números múltiplos de 5 comprendidos entre dos enteros leídos por teclado.
10. Escribir un programa que muestre los 15 primeros números de la serie de Fibonacci.

Capítulo 6

Tipos de datos elementales

Hasta el momento se ha usado implícitamente el tipo de datos entero en todos los ejemplos presentados. En este capítulo entraremos en los detalles de este tipo de datos, así como de otros tipos de datos predefinidos en C.

C proporciona los siguientes *tipos de datos elementales*:

- números enteros,
- caracteres, y
- números de coma flotante (reales).

6.1 Números enteros

En el apéndice C se describe el formato usado para almacenar números enteros en memoria. Es importante que el lector se familiarice con este formato para obtener una mejor compresión de esta sección.

La palabra clave utilizada para especificar el tipo de datos entero es `int`. Dependiendo del ordenador, del sistema operativo y el compilador utilizados, el tamaño de un entero en memoria varía. Sin embargo, hoy en día la mayoría de los computadores almacenan las variables del tipo `int` en 32 bits (4 bytes), por lo que el rango representable de números enteros va desde -2147483648 hasta 2147483647 (esto es, desde -2^{31} hasta $2^{31}-1$). Por otro lado, en el entorno IBM-PC aún existen compiladores en donde un entero sólo ocupa 16 bits (2 bytes), con un rango entre -32768 y 32767 (desde -2^{15} hasta $2^{15}-1$). Por ejemplo, Turbo C 2.0, Visual C++ 1.5, etc.

Aunque ya se ha visto previamente, el formato para declarar variables enteras es:

```
int  lista_de_variables;
```

Donde se especifica una lista de nombres de variables separados por comas.

Los números enteros en C se expresan mediante una serie de dígitos, precedidos o no por un signo $+$ o $-$. Si el número es positivo, el signo $+$ es opcional. Habitualmente se utiliza la notación decimal, aunque también es posible usar notación octal y hexadecimal. En los números expresados en octal, se utiliza un `0` como primer dígito, mientras que en los números expresados en hexadecimal, el número es precedido por un `0` y una equis mayúscula o minúscula (`0x` o `0X`). La tabla 6.1 muestra algunos ejemplos de representación de constantes enteras en los distintos formatos.

Tabla 6.1: Representación de enteros en decimal, octal y hexadecimal

Decimal	Octal	Hexadecimal
27	033	0X1B
4026	07672	0xFBA
-149	-0225	-0x95

6.1.1 Modificadores

C define una serie de modificadores para el tipo de datos entero.

El modificador `short`

Este modificador se emplea para representar números enteros de 16 bits, independientemente del procesador, por lo que su rango es $[-32768, 32767]$. Así pues, hay entornos de programación donde el tamaño y rango de una variable `short int` coincide con el de `int`. Mientras que en otros entornos, una variable de tipo `short int` ocupa la mitad de espacio y tiene un rango mucho menor. La declaración de variables de este tipo tiene la forma:

```
short int  lista_de_variables;
```

O simplemente:

```
short  lista_de_variables;
```

No existe ninguna manera de especificar explícitamente constantes de tipo `short int`.

El modificador `long`

Este modificador se emplea para representar números enteros con un rango mayor al permitido por `int`, por lo que también ocupan más espacio en memoria. Por lo tanto las variables del tipo `long int` pueden ocupar 32 o 64 bits según el entorno. Habitualmente son 64 bits, lo que da un rango de representacion de $[-9223372036854775808, 9223372036854775807]$, esto es $[-2^{63}, -2^{63} - 1]$. La declaración de variables es como sigue:

```
long int  lista_de_variables;
```

O simplemente:

```
long  lista_de_variables;
```

Para especificar explícitamente que una constante es de tipo `long int`, debe escribirse una letra ele (mayúscula o minúscula), justo detrás del valor constante. Cabe decir, que esto sólo es necesario en caso que el valor de la constante que se desea especificar esté dentro del rango de `int`. Es recomendable el uso de '`L`', pues '`l`' puede confundirse con el dígito 1. A continuación se muestra un ejemplo:

```
long x;
.  .  .
x = -554L;
x = 187387632;
```

El modificador `signed`

Es el modificador usado por defecto para todo dato de tipo `int`, por lo que no suele utilizarse de forma explícita. En cualquier caso, las declaraciones tiene la forma:

```
signed int  lista_de_variables;
```

O simplemente:

```
int  lista_de_variables;
```

El modificador `unsigned`

Este modificador permite especificar números enteros sin signo. Como consecuencia de eliminar el signo de la representación, el rango de valores positivos se amplia hasta $[0, 65535]$ si se emplean 16 bits, o $[0, 4294967295]$ si se emplean 32 bits. La declaración de variables se realiza como:

```
unsigned int  lista_de_variables;
```

O simplemente:

```
unsigned  lista_de_variables;
```

Pueden especificarse constantes de tipo `unsigned`, escribiendo una letra u mayúscula justo detrás del valor constante, como en:

```
unsigned x;
.  .  .
x = 45728;
x = 345U;
```

Finalmente, cabe comentar que el modificador `unsigned` puede combinarse con `short` y `long`. Por ejemplo, los datos de tipo `unsigned long` tienen un rango válido de $[0, 2^{64} - 1]$, es decir $[0, 18446744073709551615]$.

6.1.2 Resumen

La tabla 6.2 resume los distintos tipos de datos enteros que se han presentado, mostrando su rango, ocupación de memoria en bits y el modificador de formato para `printf` y `scanf`.

Tabla 6.2: Resumen de tipos de datos enteros

Tipo	Rango	Tamaño	Formato
`int`	$[-2147483648, 2147483647]$	**32 bits**	`%d`
`unsigned int`	$[0, 4294967295]$	**32 bits**	`%u`
`short int`	$[-32768, 32767]$	**16 bits**	`%d`
`unsigned short int`	$[0, 65535]$	**16 bits**	`%u`
`long int`	$[-2^{63}, 2^{63} - 1]$	**64 bits**	`%ld`
`unsigned long int`	$[0, 2^{64} - 1]$	**64 bits**	`%lu`

6.2 Caracteres

Este tipo de datos se identifica con la palabra reservada `char`. Comprende un conjunto ordenado y finito de caracteres, representados mediante códigos numéricos. La codificación más extendida es el estándar ASCII que utiliza 8 bits. Así pues, el tipo de datos `char` es internamente tratado como un entero, aunque puede manipularse como un carácter propiamente. El apéndice D muestra la tabla completa de códigos ASCII y los caracteres asociados. La declaración de variables se realiza como:

```
char  lista_de_variables;
```

La forma habitual de expresar una constante de tipo `char` es mediante el carácter correspondiente entre comillas:

```
char x, y;
.  .  .
x = 'f';
y = '?';
x = '5';
```

Debido a la representación interna de los caracteres mediante números enteros, pueden realizarse operaciones aritméticas como: `'F'+'5'` , o lo que es lo mismo, $70 + 53 = 123$, que equivale al carácter `'{'`; o como `'F'+5`, esto es $70 + 5 = 75$ que equivale al carácter `'K'`.

Por la misma razón también se establece un orden entre los caracteres, lo que permite usar los operadores relacionales habituales. Así pues, la comparación `'a' <= 'h'` da como resultado el valor cierto, o la comparación `'K' > 'V'` da como resultado el valor falso.

6.2.1 Caracteres especiales

Algunos de los caracteres definidos por el estándar ASCII no son directamente imprimibles, por lo que para utilizarlos en un programa es necesario un mecanismo especial. En realidad, C ofrece dos:

- Puede usarse el código ASCII asociado al carácter, que puede representarse tanto en decimal, octal o hexadecimal. Por ejemplo, el carácter `'\9'` representa un tabulador mediante su código ASCII en decimal; el carácter `'\040'` corresponde al espacio en blanco mediante su código ASCII en octal, etc.

- O bien, pueden usarse *secuencias de escape* especiales, como `'\n'` , para forzar un salto de línea, que hemos utilizado en ejemplos anteriores. La siguiente tabla muestra los más utilizados:

Tabla 6.3: Caracteres interpretados como enteros

Tipo	Rango	Tamaño	Formato
`char`	$[-128, 127]$	**1 byte**	`%c`
`signed char`	$[-128, 127]$	**1 byte**	`%d`
`unsigned char`	$[0, 255]$	**1 byte**	`%u`

Código	Significado
`\a`	**pitido**
`\n`	**salto de línea**
`\f`	**salto de página**
`\r`	**principio de la misma línea**
`\t`	**tabulador**
`\'`	**apóstrofe**
`\"`	**comillas**
`\0`	**carácter nulo**

6.2.2 Enteros y el tipo `char`

Puesto que el tipo `char` está internamente representado mediante códigos numéricos de 8 bits, una variable de este tipo puede interpretarse también como una variable entera. Así pues, pueden utilizarse también los modificadores `signed` y `unsigned`. Los rangos de valores se muestran en la tabla 6.3.

6.2.3 Conversiones de tipos

En general, en C puede convertirse el tipo de una variable o constante a otro tipo cualquiera. Para ello, basta con escribir delante de dicha variable el nuevo tipo entre paréntesis. Aunque no profundizaremos en la conversión de tipos en C, veremos su utilidad en este caso, para obtener el código ASCII de un carácter y viceversa.

En el ejemplo de la izquierda, se asigna a la variable `cod` el código ASCII del carácter `'A'` almacenado en la variable `c`, que es 65. En el ejemplo de la derecha, se asigna a la variable `c` el carácter correspondiente al código ASCII 98 almacenado en la variable `cod`, que es el caracter `'b'`.

```
int cod;
char c;
.  .  .
c = 'A';
cod = (int) c;
.  .  .
```

```
int cod;
char c;
.  .  .
cod = 98;
c = (char) cod;
.  .  .
```

Para concluir, un par de aspectos prácticos:

- Si la variable `c` almacena algún carácter del rango `'0'...'9'`, puede obtenerse su valor numérico equivalente (no confundir con el código ASCII) mediante la sentencia:

  ```
  i = (int)c - (int)'0';
  ```

Tabla 6.4: Resumen de tipos de datos reales

Tipo	Rango	Tamaño	Formato
`float`	$\pm 1.1754945E-38$ $\pm 3.4028232E+38$	**4 bytes**	`%f`, `%e`, `%g`
`double`	$\pm 2.225073858507202E-308$ $\pm 1.797693134862312E+308$	**8 bytes**	`%f`, `%e`, `%g`
`long double`	$\pm 8.4 \ldots E-4932$ $\pm 5.9 \ldots E+4932$	**16 bytes**	`%f`, `%e`, `%g`

- Si `i` es una variable entera con un valor en el rango $[0, 9]$, puede obtenerse el carácter correspondiente de la siguiente forma:

```
c = (char)((int)'0' + i);
```

6.3 Números reales

Nuevamente, en el apéndice C se describe el formato de coma flotante usado para almacenar números reales. Es importante que el lector se familiarice con este formato para obtener una mejor comprensión de esta sección.

En C existen básicamente dos tipos de números reales, los de precisión simple (`float`) y los de precisión doble (`double`). Un valor de tipo `float` ocupa 4 bytes (32 bits) y permite representar números reales con 8 dígitos de precisión. Un valor de tipo `double` ocupa 8 bytes (64 bits) y permite representar números reales con 16 dígitos de precisión (ver Tab. 6.4).

Las constantes reales pueden representarse mediante dos notaciones: la decimal, constituida por una serie de dígitos donde la parte decimal se sitúa detrás de un punto, por ejemplo `-5.034` o `443.43`; y la notación científica o exponencial, formada por una parte en notación decimal que representa la mantisa, seguida del carácter `'E'` o `'e'` y un número entero que representa el exponente, como por ejemplo: `-3.56E67` o `7.9e-15`.

Por defecto, las constantes reales se consideran de tipo `double`. Para especificar explícitamente constantes de tipo `float`, debe escribirse una letra efe mayúscula tras la constante, como en el siguiente ejemplo:

```
double x;
float y;
.  .  .
x = 34E23;
y = 2.3E12F;
```

Algunos compiladores admiten el uso del modificador `long` para el tipo `double`. Este tipo de variables se almacenan en 16 bytes y proporcionan precisión cuádruple (32 dígitos).

Para concluir, veamos un ejemplo de uso de números reales en un programa que calcula el sumatorio $\sum_{i=1}^{\infty} \frac{1}{i}$ con un error inferior a un valor ϵ, mientras que $i \leq 1000$. Expresando el error

matemáticamente tenemos:

$$\mid \sum_{i=1}^{k} \frac{1}{i} - \sum_{i=1}^{k-1} \frac{1}{i} \mid < \epsilon$$

A continuación se muestra un programa que realiza dicho cálculo:

```
#include <stdio.h>
#define LIMITE 1000
void main ()
{
   int i, fin;
   float suma, t, epsilon;

   suma = 0.0F;
   fin = 0;
   i = 1;
   scanf( "%f", &epsilon );
   while (!fin)
   {
      t = 1.0F / (float)i;
      suma = suma + t;
      i++;
      fin = ((t < epsilon) || (i > LIMITE));
   }
   printf( "La suma es %f\n", suma );
}
```

Obsérvese el uso del modificador de formato `%f` para la entrada y salida de valores de coma flotante, y el uso de la variable `fin` para controlar la terminación del bucle.

En el fichero de cabeceras `math.h` (`#include <math.h>`), perteneciente a la librería estándar (ver Ap. B), se definen una serie de funciones matemáticas para operar con valores reales, como: `sqrt` para la raíz cuadrada, `sin` y `cos` para senos y cosenos, etc.

6.4 Ejercicios

1. Escribir un programa que cuente el número de veces que aparece una letra en una secuencia de caracteres acabada en `'.'` .

2. Escribir un programa que lea un carácter de teclado e informe de si es alfabético, numérico, blanco o un signo de puntuación.

3. Escribir un programa que convierta una secuencia de dígitos entrados por teclado al número entero correspondiente. Supóngase que el primer dígito leído es el de mayor peso. Obsérvese también que el peso efectivo de cada dígito leído es desconocido hasta haber concluido la introducción de la secuencia.

4. Sean las variables enteras `i` y `j` con valores 5 y 7, respectivamente. Y las variables de coma flotante `f` y `g` con valores 5.5 y -3.25, respectivamente. ¿Cuál será el resultado de las siguientes asignaciones?

   (a) `i = i % 5;`

   (b) `f = (f - g) / 2;`

   (c) `j = j * (i - 3);`

   (d) `f = f % g;`

   (e) `i = i / (j - 2);`

5. Escribir un programa que calcule y muestre por pantalla las raíces de una ecuación de segundo grado, leyendo previamente los coeficientes A, B y C de la misma: $Ax^2 + Bx + C = 0$.

6. Escribir un programa que calcule el perímetro de una circunferencia y que lo muestre por pantalla con cuatro decimales de precisión. Si el radio introducido por el usuario es negativo, el perímetro resultante será 0.

7. Escribir un programa para calcular de forma aproximada el número e. Recordar que $e = \sum_{i=0}^{\infty} \frac{1}{i!}$.

Capítulo 7

Tipos de datos estructurados: Tablas

En este capítulo veremos algunos de los mecanismos que C ofrece para la creación de tipos de datos complejos. Éstos se construyen, en un principio, a partir de los tipos de datos elementales predefinidos por el lenguaje (ver Cap. 6).

Comenzaremos hablando del tipo abstracto de datos *tabla*, tanto de una (*vectores*), dos (*matrices*) o múltiples dimensiones. En C existe un tratamiento especial para los vectores de caracteres, por lo que dedicaremos una parte de este capítulo a su estudio. Se deja para el capítulo 8 el estudio de otros tipos de datos estructurados, como las *estructuras*, las *uniones*, y los tipos de datos *enumerados*.

Las tablas en C son similares a las que podemos encontrar en otros lenguajes de programación. Sin embargo, se definen de forma diferente y poseen algunas peculiaridades derivadas de la estrecha relación con los punteros. Volveremos a esto más adelante en el capítulo 9.

7.1 Vectores

Los *vectores*, también llamados *tablas unidimensionales*, son estructuras de datos caracterizadas por:

- Una colección de datos del mismo tipo.
- Referenciados mediante un mismo nombre.
- Almacenados en posiciones de memoria físicamente contiguas, de forma que, la dirección de memoria más baja corresponde a la del primer elemento, y la dirección de memoria más alta corresponde a la del último elemento.

El formato general para la declaración de una variable de tipo vector es el siguiente:

```
tipo_de_datos nombre_tabla [tamaño];
```

donde:

- `tipo_de_datos` indica el tipo de los datos almacenados por el vector. Recordemos que todos los elementos del vector son forzosamente del mismo tipo. Debe aparecer necesariamente en la declaración, puesto que de ella depende el espacio de memoria que se reservará para almacenar el vector.

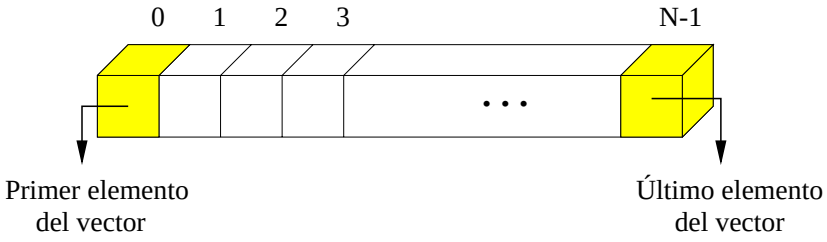


Figura 7.1: Representación gráfica de un vector de N elementos

- `nombre_tabla` es un identificador que usaremos para referiremos tanto al vector como un todo, como a cada uno de sus elementos.
- `tamaño` es una expresión entera constante que indica el número de elementos que contendrá el vector. El espacio ocupado por un vector en memoria se obtiene como el producto del número de elementos que lo componen y el tamaño de cada uno de éstos.

7.1.1 Consulta

El acceso a un elemento de un vector se realiza mediante el nombre de éste y un índice entre corchetes (`[ ]`). El índice representa la posición relativa que ocupa dicho elemento dentro del vector y se especifica mediante una expresión entera (normalmente una constante o una variable). Formalmente:

```
nombre_vector[índice];
```

A continuación se muestran algunas formas válidas de acceso a los elementos de un vector:

```
int contador[10];
int i, j, x;
.  .  .
x = contador[1];
x = contador[i];
x = contador[i * 2 + j];
.  .  .
```

Como muestra la figura 7.1, el primer elemento de un vector en C se sitúa en la posición 0, mientras que el último lo hace en la posición $N - 1$ (N indica el número de elementos del vector). Por esta razón, el índice para acceder a los elementos del vector debe permanecer entre estos dos valores. Es responsabilidad del programador garantizar este hecho, para no acceder a posiciones de memoria fuera del vector, lo cual produciría errores imprevisibles en el programa.

7.1.2 Asignación

La asignación de valores a los elementos de un vector se realiza de forma similar a como se consultan. Veámoslo en un ejemplo:

```
int contador[3];
.  .  .
contador[0] = 24;
contador[1] = 12;
contador[2] = 6;
```

En muchas ocasiones, antes de usar un vector (una tabla en general) por primera vez, es necesario dar a sus elementos un valor inicial. La manera más habitual de inicializar un vector en tiempo de ejecución consiste en recorrer secuencialmente todos sus elementos y darles el valor inicial que les corresponda. Veámoslo en el siguiente ejemplo, donde todos los elementos de un vector de números enteros toman el valor 0:

```
#define TAM 100
void main()
{
   int vector[TAM], i;

   for (i= 0; i< TAM; i++)
      vector[i] = 0;
}
```

Nótese que el bucle recorre los elementos del vector empezando por el elemento 0 (`i=0`) y hasta el elemento `TAM-1` (condición `i<TAM`).

Existe también un mecanismo que permite asignar un valor a todos los elementos de un tabla con una sola sentencia. Concretamente en la propia declaración de la tabla. La forma general de inicializar una tabla de cualquier número de dimensiones es la siguiente:

```
tipo_de_datos nombre_tabla [tam1]...[tamN] = { lista_valores };
```

La lista de valores no deberá contener nunca más valores de los que puedan almacenarse en la tabla. Veamos algunos ejemplos:

```
int contador[3] = {24, 12, 6}; /* Correcto */
char vocales[5] = {'a', 'e', 'i', 'o', 'u'}; /* Correcto */
int v[4] = {2, 6, 8, 9, 10, 38}; /* Incorrecto */
```

Finalmente, cabe destacar que no está permitido en ningún caso comparar dos vectores (en general dos tablas de cualquier número de dimensiones) utilizando los operadores relacionales que vimos en la sección 3.4.3. Tampoco está permitida la copia de toda una tabla en otra con una simple asignación. Si se desea comparar o copiar toda la información almacenada en dos tablas, deberá hacerse elemento a elemento.

Los mecanismos de acceso descritos en esta sección son idénticos para las matrices y las tablas multidimensionales. La única diferencia es que será necesario especificar tantos índices como dimensiones posea la tabla a la que se desea acceder. Esto lo veremos en las siguientes secciones.

7.1.3 Ejemplos

A continuación se muestra un programa que cuenta el número de apariciones de los números 0, 1, 2 y 3 en una secuencia de enteros acabada en -1.

```
#include <stdio.h>
void main ()
{
   int num, c0=0, c1=0, c2=0, c3=0;

   scanf("%d", &num);
   while (num != -1)
   {
      if (num == 0) c0++;
      if (num == 1) c1++;
      if (num == 2) c2++;
      if (num == 3) c3++;
      scanf( "%d", &num );
   }
   printf( "Contadores:%d, %d, %d, %d\n", c0, c1, c2, c3 );
}
```

¿Qué ocurriría si tuviésemos que contar las apariciones de los cien primeros números enteros? ¿Deberíamos declarar cien contadores y escribir cien construcciones `if` para cada caso? La respuesta, como era de esperar, se halla en el uso de vectores. Veámoslo en el siguiente programa:

```
#include <stdio.h>
#define MAX 100
void main ()
{
   int i, num, cont[MAX];

   for (i= 0; i< MAX; i++)
      cont[i] = 0;
   scanf("%d", &num);
   while (num != -1) {
      if ((num >= 0) && (num <= MAX))
         cont[num]++;
      scanf( "%d", &num );
   }
   for (i= 0; i< MAX; i++)
      printf( "Contador [%d] = %d\n", i, cont[i] );
}
```

Veamos finalmente otro ejemplo en el que se muestra cómo normalizar un vector de números reales. La normalización consiste en dividir todos los elementos del vector por la raíz cuadrada de la suma de sus cuadrados. Destaca el uso de la constante `MAX` para definir el número de elementos del vector y de la función `sqrt` para el cálculo de raíces cuadradas.

```
#include <math.h>
#include <stdio.h>
#define MAX 10
void main()
{
   float vector[MAX], modulo;
   int i;

   /* Lectura del vector.  */
   for (i= 0; i< MAX; i++)
      scanf("%f", &vector[i]);

   /* Calcular módulo.  */
   modulo = 0.0;
   for (i= 0; i< MAX; i++)
      modulo = modulo + (vector[i] * vector[i]);
   modulo = sqrt(modulo);

   /* Normalizar */
   for (i= 0; i< MAX; i++)
      vector[i] /= modulo;

   /* Escritura del vector.  */
   for (i= 0; i< MAX; i++ )
      printf( "%f ", vector[i] );
}
```

7.2 Matrices

Las *matrices*, también llamadas *tablas bidimensionales*, no son otra cosa que vectores con dos dimensiones. Por lo que los conceptos de acceso, inicialización, etc. son similares.

La declaración de una variable matriz tiene la forma siguiente:

```
tipo_de_datos nombre_tabla [tamaño_dim1][tamaño_dim2];
```

Donde `tamaño_dim1` y `tamaño_dim2` indican, respectivamente, el número de filas y de columnas que componen la matriz. La figura 7.2 muestra la representación gráfica habitual de una matriz de datos.

Otro hecho importante es que las matrices en C se almacenan “por filas”. Es decir, que los elementos de cada fila se sitúan en memoria de forma contigua. Así pues, en la matriz de la figura anterior, el primer elemento almacenado en memoria es el `(0,0)`, el segundo el `(0,1)`, el tercero el `(0,2)`,..., `(0,M-1)`, después `(1,0)`, y así sucesivamente hasta el último elemento, es decir `(N-1,M-1)`.

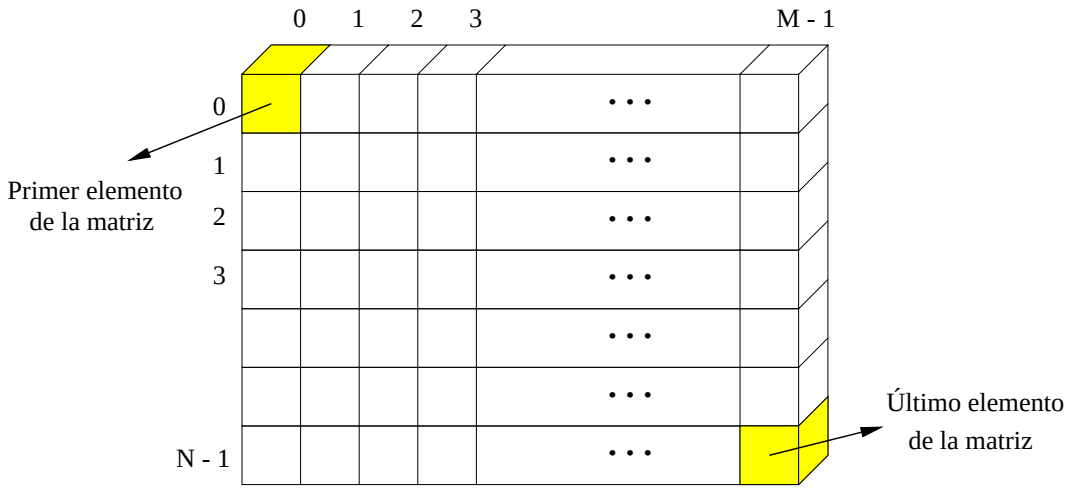


Figura 7.2: Representación gráfica de una matriz de N filas y M columnas

7.2.1 Consulta

El acceso a un elemento de una matriz se realiza mediante el nombre de ésta y dos índices (uno para cada dimensión) entre corchetes. El primer índice representa la fila y el segundo la columna en que se encuentra dicho elemento. Tal como muestra la figura 7.2, el índice de las filas tomará valores entre 0 y el número de filas menos 1, mientras que el índice de las columnas tomará valores entre 0 y el número de columnas menos 1. Es responsabilidad del programador garantizar este hecho.

```
nombre_matriz[índice_1][índice_2];
```

7.2.2 Asignación

Comentaremos únicamente la inicialización de una matriz en la propia declaración. El siguiente ejemplo declara una matriz de tres filas y cuatro columnas y la inicializa. Por claridad, se ha usado identación en los datos, aunque hubiesen podido escribirse todos en una sola línea.

```
int mat[3][4] = { 24, 12, 6, 17,
                  15, 28, 78, 32,
                  0, 44, 3200 , -34
                };
```

La inicialización de matrices, y en general de tablas multidimensionales, puede expresarse de forma más clara agrupando los valores mediante llaves ({ }), siguiendo la estructura de la matriz. Así pues, el ejemplo anterior también podría escribirse como:

```
int mat[3][4] = { { 24, 12, 6, 17 },
                  { 15, 28, 78, 32 },
                  { 0, 44, 3200 , -34 }
                };
```

7.2.3 Ejemplo

El siguiente ejemplo calcula la matriz traspuesta de una matriz de enteros. La matriz tendrá unas dimensiones máximas según la constante `MAX`.

```
#include <stdio.h>
#define MAX 20
void main()
{
   int filas, columnas, i, j;
   int matriz[MAX][MAX], matrizT[MAX][MAX];

   /* Lectura matriz */
   printf( "Num.  filas, Num.  columnas:  " );
   scanf( "%d%d", &filas, &columnas );
   printf ("Introducir matriz por filas:" );
   for (i= 0; i< filas; i++)
      for (j= 0; j< columnas; j++)
      {
         printf( "\nmatriz[%d][%d] = ", i, j );
         scanf( "%d", &matriz[i][j] );
      }

   /* Traspuesta */
   for (i= 0; i< filas; i++)
      for (j= 0; j< columnas; j++)
         matrizT[j][i] = matriz[i][j];

   /* Escritura del resultado */
   for (i= 0; i< filas; i++)
      for (j= 0; j< columnas; j++)
         printf( "\nmatrizT[%d][%d] = %d ",
               i, j, matrizT[i][j] );
}
```

Obsérvese que para recorrer todos los elementos de una matriz es necesario el empleo de dos bucles anidados que controlen los índices de filas y columnas (siempre entre 0 y el número de filas o columnas menos 1). En este ejemplo todos los recorridos se realizan “por filas”, es decir, que primero se visitan todos los elementos de una fila, luego los de la siguiente, etc. Finalmente, cabe comentar que para el recorrido de tablas multidimensionales será necesario el empleo de tantos bucles como dimensiones tenga la tabla.

7.3 Tablas multidimensionales

Este tipo de tablas se caracteriza por tener tres o más dimensiones. Al igual que vectores y matrices, todos los elementos almacenados en ellas son del mismo tipo de datos.

La declaración de una tabla multidimensional tiene la forma siguiente:

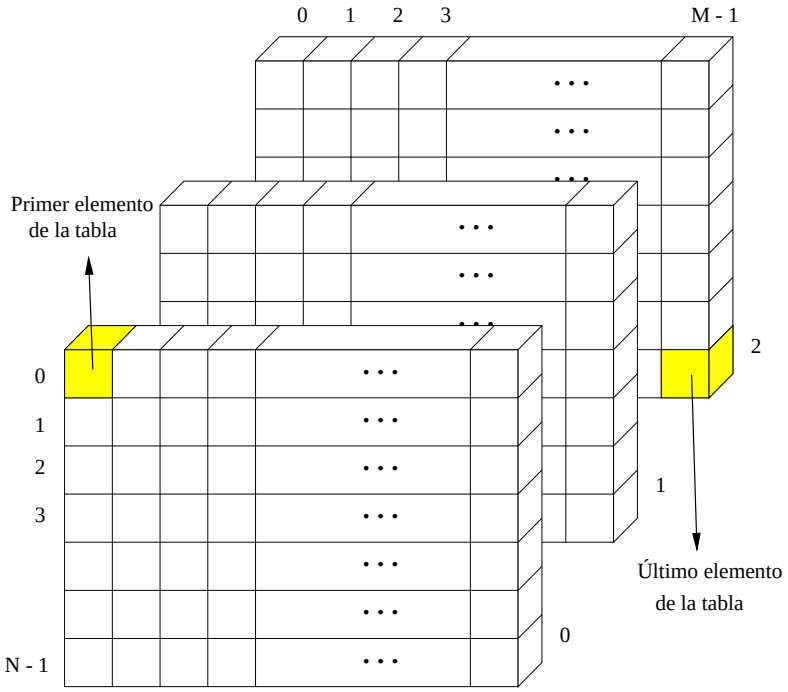


Figura 7.3: Representación gráfica de una tabla de tres dimensiones: $N \times M \times 3$

```
tipo_de_datos nombre_tabla [tamaño_dim1] ...[tamaño_dimN];
```

Para acceder a un elemento en particular será necesario usar tantos índices como dimensiones:

```
nombre_vector[índice_1] ...[índice_N];
```

Aunque pueden definirse tablas de más de tres dimensiones, no es muy habitual hacerlo. La figura 7.3 muestra como ejemplo la representación gráfica habitual de una tabla de tres dimensiones.

7.3.1 Ejemplo

El siguiente ejemplo muestra el empleo de una tabla multidimensional. Concretamente, el programa utiliza una tabla de 3 dimensiones para almacenar 1000 números aleatorios. Para generar números aleatorios se usa la función `rand` de la librería estándar `stdlib.h`. También se ha usado la función `getchar` (`stdio.h`), que interrumpe el programa y espera a que el usuario presione una tecla.

```
#include <stdio.h>
#include <stdlib.h>
#define DIM 10
void main()
{
   int tabla_random [DIM][DIM][DIM], a, b, c;

   for (a= 0; a< DIM; a++)
```

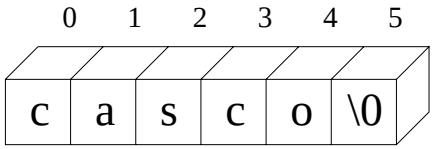
```
        for (b= 0; b< DIM; b++)
           for (c= 0; c< DIM; c++)
              tabla_random[a][b][c] = rand();

     /* Muestra series de DIM en DIM elementos.  */
     for (a= 0; a< DIM; a++)
        for (b= 0; b < DIM; b++)
        {
           for (c= 0; c < DIM; c++)
           {
              printf( "\n tabla[%d][%d][%d] = ", a, b, c );
              printf( "%d", tabla_random[a][b][c] );
           }
           printf( "\nPulse ENTER para seguir" );
           getchar();
        }
}
```

7.4 Cadenas de caracteres

Las cadenas de caracteres son vectores de tipo carácter (`char`) que reciben un tratamiento especial para simular el tipo de datos "*string*", presente en otros lenguajes de programación.

Para que un vector de caracteres pueda ser considerado como una cadena de caracteres, el último de los elementos útiles del vector debe ser el carácter *nulo* (código ASCII 0). Según esto, si se quiere declarar una cadena formada por N caracteres, deberá declararse un vector con $N + 1$ elementos de tipo carácter. Por ejemplo, la declaración `char cadena[6];` reserva suficiente espacio en memoria para almacenar una cadena de 5 caracteres, como la palabra `"casco"`:



En C pueden definirse constantes correspondientes a cadenas de caracteres. Se usan comillas dobles para delimitar el principio y el final de la cadena, a diferencia de las comillas simples empleadas con las constantes de tipo carácter. Por ejemplo, la cadena constante `"H"` tiene muy poco que ver con el carácter constante `'H'`, si observamos la representación interna de ambos:

7.4.1 Asignación

Mientras que la consulta de elementos de una cadena de caracteres se realiza de la misma forma que con los vectores, las asignaciones tienen ciertas peculiaridades.

Como en toda tabla, puede asignarse cada carácter de la cadena individualmente. No deberá olvidarse en ningún caso que el último carácter válido de la misma debe ser el carácter nulo (`'\0'`). El siguiente ejemplo inicializa la cadena de caracteres `cadena` con la palabra `"casco"`. Nótese que las tres últimas posiciones del vector no se han usado. Es más, aunque se les hubiese asignado algún carácter, su contenido sería ignorado. Esto es, el contenido del vector en las posiciones posteriores al carácter nulo es ignorado.

```
char cadena[10];

.  .  .
cadena[0] = 'c';
cadena[1] = 'a';
cadena[2] = 's';
cadena[3] = 'c';
cadena[4] = 'o';
cadena[5] = '\0';
```

La inicialización de una cadena de caracteres durante la declaración puede hacerse del mismo modo que en los vectores, aunque no debe olvidarse incluir el carácter nulo al final de la cadena. Sin embargo, existe un método de inicialización propio de las cadena de caracteres, cuyo formato general es:

```
char nombre [tamaño] = "cadena";
```

Usando este tipo de inicialización, el carácter nulo es añadido automáticamente al final de la cadena. Así pues, una inicialización típica de vectores como la siguiente:

```
char nombre[10] = { 'N', 'U', 'R', 'I', 'A', '\0' };
```

puede hacerse también de forma equivalente como:

```
char nombre[10] = "NURIA";
```

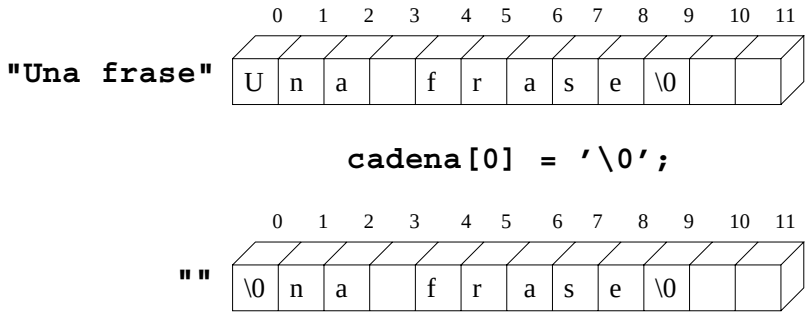
Finalmente, la inicialización anterior puede hacerse sin necesidad de especificar el tamaño del vector correspondiente. En este caso, el compilador se encarga de calcularlo automáticamente, reservando espacio de memoria suficiente para almacenar la cadena, incluyendo el carácter nulo al final. Así pues, la siguiente declaración reserva memoria para almacenar 6 caracteres y los inicializa adecuadamente con las letras de la palabra `NURIA`:

```
char nombre[] = "NURIA";
```

La cadena vacía

Otra curiosidad de las cadenas de caracteres se refiere a la cadena vacía, `""`, que consta únicamente del carácter nulo. Puesto que los caracteres posteriores al carácter nulo son ignorados, convertir una cadena con cualquier valor almacenado a la cadena vacía es tan simple como asignar el carácter nulo a la posición 0 de dicha cadena. He aquí un ejemplo:

```
char cadena [12] = "Una frase";
.  .  .
cadena[0] = '\0'; /* Ahora es una cadena vacía */
```



7.4.2 Manejo de cadenas de caracteres

Aunque C no incorpora en su definición operadores para el manejo de cadenas de caracteres, todo compilador de C proporciona una librería estándar (`string.h`) con funciones para facilitar su utilización. Destacar algunas de ellas:

- `strlen` para obtener la longitud de la cadena, sin contar el carácter nulo,
- `strcpy` para copiar una cadena en otra,
- `strcat` para concatenar dos cadenas,
- `strcmp` para comparar dos cadenas, etc.

Para más información sobre estas y otras funciones, consultar el apéndice B.

Entrada y salida

En cuanto a la entrada y salida de cadenas de caracteres, existe un formato especial `%s` que puede utilizarse en las funciones `scanf` y `printf`. Por ejemplo, la siguiente sentencia leerá una cadena de caracteres en la variable `cad`. Sólo se asignarán caracteres mientras no sean caracteres blancos, tabuladores o saltos de línea. Por lo tanto, el empleo de `%s` sólo tendrá sentido para la lectura de palabras. Además del formato `%s` existen los formatos `%[^abc]` y `%[abc]`, que permiten leer respectivamente una cadena de caracteres hasta encontrar algún carácter del conjunto {a, b, c}, o bien hasta no encontrar un caracter del conjunto {a, b, c}. En cualquier caso el carácter del conjunto {a, b, c} no es leído. Ver el apéndice B para más información sobre el empleo de `scanf` y la lectura de cadenas de caracteres.

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

Nótese que, en el ejemplo, no se ha antepuesto el símbolo `&` a la variable `cad`. Por el momento, tengámoslo en mente y esperemos hasta el capítulo 9 para comprender a qué se debe este hecho.

La librería estándar de entrada y salida (`stdio.h`) proporciona además las funciones `gets` y `puts`, que permiten leer de teclado y mostrar por pantalla una cadena de caracteres completa, respectivamente (ver el apéndice B para más detalles).

7.4.3 Ejemplos

Para finalizar, veamos un par de ejemplos de manejo de cadenas de caracteres.

El siguiente programa cuenta el número de veces que se repite una palabra en una frase. El programa emplea la función de comparación de cadenas `strcmp`. Dicha función devuelve 0 en caso de que las cadenas comparadas sean iguales (ver Sec. B.1).

```
#include <stdio.h>
#include <string.h>
#define MAXLIN 100
void main()
{
   char pal[MAXLIN];       /* La que buscamos.  */
   char palfrase[MAXLIN];  /* Una palabra de la frase.  */
   char c;
   int total = 0;

   printf( "\nPALABRA:" );
   scanf( "%s", pal );
   printf( "\nFRASE:" );
   c = ' ';
   while (c != '\n')
   {
      scanf( "%s%c", palfrase, &c );
      if (strcmp(pal, palfrase) == 0)
         total++;
   }
   printf( "\nLa palabra %s aparece %d veces.", pal, total );
}
```

A continuación se muestra otro ejemplo que hace uso de las cadenas de caracteres. El programa lee dos cadenas de caracteres, las concatena, convierte las letras minúsculas en mayúsculas y viceversa, y finalmente escribe la cadena resultante.

```
#include <stdio.h>
#include <string.h>
void main()
{
   char cad1[80], cad2[80], cad3[160];
   int  i, delta;

   printf( "Introduzca la primera cadena:\n" );
   gets(cad1);
   printf( "Introduzca la segunda cadena:\n" );
   gets( cad2 );
   /* cad3 = cad1 + cad2 */
   strcpy( cad3, cad1 );
```

```
    strcat( cad3, cad2 );
    i = 0;
    delta = 'a' - 'A';
    while (cad3[i] != '\0')
    {
       if ((cad3[i] >= 'a') && (cad3[i] <= 'z'))
          cad3[i] -= delta; /* Convierte a mayúscula */
       else if ((cad3[i] >= 'A') && (cad3[i] <= 'Z'))
          cad3[i] += delta; /* Convierte a minúscula */
       i++;
    }
    printf( "La cadena resultante es:  %s \n", cad3 );
}
```

7.5 Ejercicios

1. ¿Dónde está el error en el siguiente programa?

```
void main()
{
   int vector [10];
   int x = 1;

   for (x= 1; x<= 10; x++)
      vector[x] = 23;
}
```

2. Escribir un programa que lea del teclado un vector de 10 números enteros, lo invierta y finalmente lo muestre de nuevo.

3. Escribir un programa que cuente el número de palabras de más de cuatro caracteres en una frase. Ésta se almacena en forma de vector cuyo último elemento es el carácter `'.'`.

4. Escribir un programa que lea del teclado dos números enteros de hasta 20 dígitos y los sume. Usar vectores para almacenar los dígitos de cada número.

5. Escribir un programa que decida si una palabra es palíndroma o no. La palabra se almacena en un vector de caracteres acabado en `'.'`.

6. Escribir un programa para calcular la *moda* de un conjunto de números enteros. La moda es el valor que se repite más veces.

7. ¿Dónde está el error en el siguiente programa?

```
void main()
{
   int matriz [10][3], x, y;
```

```
    for (x= 0; x< 3; x++)
       for (y= 0; y< 10; y++)
          matrix[x][y] = 0;
}
```

8. Escribir un programa que inicialice cada elemento de una matriz de enteros con el valor de la suma del número de fila y columna en que está situado.

9. Escribir un programa que calcule la suma de dos matrices de enteros.

10. Escribir un programa que calcule los *puntos de silla* de una matriz de enteros. Un elemento de una matriz es un punto de silla si es el mínimo de su fila y el máximo de su columna.

11. Escribir un programa que determine si una matriz es simétrica.

12. Escribir un programa que multiplique dos matrices.

13. Escribir un programa que lea una frase del teclado y cuente los espacios en blanco.

14. Escribir un programa que, dada una cadena de caracteres y un entero correspondiente a una posición válida dentro de ella, genere una nueva cadena de caracteres que contenga todos los caracteres a la izquierda de dicha posición, pero en orden inverso.

15. Escribir un programa que, dada una cadena de caracteres, la limpie de caracteres blancos. Por ejemplo, la cadena `"Esto es una frase"` deberá transformarse en `"Estoesunafrase"`. Escribir dos versiones, una utilizando una cadena auxiliar y otra versión que realice los cambios sobre la misma cadena.

16. Escribir un programa que lea dos cadenas de caracteres, las compare e informe de si son iguales o diferentes. No usar la función de la librería estándar `strcmp`.

Capítulo 8

Otros tipos de datos

En este capítulo veremos los restantes mecanismos que C ofrece para la creación y manejo de tipo de datos complejos. Concretamente las *estructuras* y las *uniones*. Por otra parte, se incluye también un apartado que estudia los tipos de datos *enumerados* y otro donde se trata la definición de nuevos tipos de datos por parte del programador.

8.1 Estructuras

En el capítulo 7 hemos estudiado el tipo abstracto de datos *tabla*, formado por un conjunto de elementos todos ellos del mismo tipo de datos. En una *estructura*, sin embargo, los elementos que la componen pueden ser de distintos tipos. Así pues, una estructura puede agrupar datos de tipo carácter, enteros, cadenas de caracteres, matrices de números ..., e incluso otras estructuras. En cualquier caso, es habitual que todos los elementos agrupados en una estructura tengan alguna relación entre ellos. En adelante nos referiremos a los elementos que componen una estructura como *campos*.

La definición de una estructura requiere especificar un nombre y el tipo de datos de todos los campos que la componen, así como un nombre mediante el cual pueda identificarse toda la agrupación. Para ello se utiliza la palabra reservada `struct` de la forma siguiente:

```
struct nombre_estructura
{
   tipo_campo_1   nombre_campo_1;
   tipo_campo_2   nombre_campo_2;
   .   .   .
   tipo_campo_N   nombre_campo_N;
};
```

El siguiente ejemplo define una estructura denominada `cliente` en la que puede almacenarse la ficha bancaria de una persona. El significado de los diferentes campos es obvio:

```
struct cliente
{
   char        nombre[100];
   long int    dni;
```

```
    char        domicilio[200];
    long int    no_cuenta;
    float       saldo;
}
```

Puede decirse que la definición de una estructura corresponde a la definición de una "plantilla" genérica que se utilizará posteriormente para declarar variables. Por tanto la definición de una estructura no representa la reserva de ningún espacio de memoria.

8.1.1 Declaración de variables

La declaración de variables del tipo definido por una estructura puede hacerse de dos maneras diferentes. Bien en la misma definición de la estructura, bien posteriormente. La forma genérica para el primer caso es la siguiente:

```
struct nombre_estructura
{
   tipo_campo_1   nombre_campo_1;
   tipo_campo_2   nombre_campo_2;
   .   .   .
   tipo_campo_N   nombre_campo_N;
} lista_de_variables;
```

Nótese que al declararse las variables al mismo tiempo que se define la estructura, el nombre de ésta junto a la palabra reservada `struct` se hace innecesario y puede omitirse.

Por otra parte, suponiendo que la estructura `nombre_estructura` se haya definido previamente, la declaración de variables en el segundo caso sería:

```
struct nombre_estructura lista_de_variables;
```

Siguiendo con el ejemplo anterior, según la primera variante de declaración, podríamos escribir:

```
struct
{
   char        nombre[100];
   long int    dni;
   char        domicilio[200];
   long int    no_cuenta;
   float       saldo;
} antiguo_cliente, nuevo_cliente;
```

O bien, de acuerdo con la segunda variante donde se asume que la estructura `cliente` se ha definido previamente:

```
struct cliente antiguo_cliente, nuevo_cliente;
```

8.1.2 Acceso a los campos

Los campos de una estructura de manejan habitualmente de forma individual. El mecanismo que C proporciona para ello es el operador "punto" (`.`). La forma general para acceder a un campo en una variable de tipo estructura es el siguiente:

```
variable.nombre_campo
```

Así pues, podríamos acceder a los campos de la variable `nuevo_cliente` como el `nombre`, número de `dni` o el `saldo` de la siguiente forma:

```
nuevo_cliente.nombre  nuevo_cliente.dni  nuevo_cliente.saldo
```

8.1.3 Asignación

La asignación de valores a los campos de una variable estructura puede realizarse de dos formas diferentes. Por un lado, accediendo campo a campo, y por otro, asignando valores para todos los campos en el momento de la declaración de la variable.

A continuación se muestran algunas posibles maneras de asignar datos a los campos individuales de la variable `nuevo_cliente` declarada previamente. Como puede verse, cada campo es tratado como si de una variable se tratase, con lo que pueden usarse funciones como `strcpy` para copiar una cadena de caracteres sobre un campo de ese tipo, o `gets` para leerla del teclado, etc.

```
strcpy( nuevo_cliente.nombre, "Federico Sancho Buch" );
nuevo_cliente.dni = 23347098;
gets( nuevo_cliente.domicilio );
scanf( "%ld",&nuevo_cliente.no_cuenta );
nuevo_cliente.saldo += 10000.0;
```

Por otra parte, el siguiente ejemplo muestra la inicialización completa de todos los campos de una variable de tipo estructura en el momento de su declaración:

```
struct cliente nuevo_cliente = {
                                  "Federico Sancho Buch",
                                  23347098,
                                  "Rue del Percebe 13 - Madrid",
                                  7897894,
                                  1023459.34
                                };
```

Finalmente, también es posible copiar todos los datos de una variable estructura a otra variable estructura del mismo tipo mediante una simple asignación:

```
nuevo_cliente = antiguo_cliente;
```

No está permitido en ningún caso, comparar dos estructuras utilizando los operadores relacionales que vimos en la sección 3.4.3.

8.1.4 Ejemplo

El siguiente programa define las estructuras de datos para gestionar un conjunto de fichas personales. Nótese el uso de estructuras anidadas, así como el uso de tablas de estructuras. Nótese también el uso de la función `gets` para leer cadenas de caracteres. Se deja como ejercicio para el lector escribir este mismo programa usando `scanf` en lugar de `gets` para leer dichas cadenas.

```
#include <stdio.h>
struct datos
{
   char nombre[20];
   char apellido[20];
   long int dni;
};
struct tablapersonas
{
   int numpersonas;
   struct datos persona[100];
};
void main()
{
   int i;
   struct tablapersonas tabla;

   printf( "Número de personas:  " );
   scanf( "%d", &tabla.numpersonas );
   for (i= 0; i< tabla.numpersonas; i++)
   {
      printf( "\nNombre:  " );
      gets( tabla.persona[i].nombre );
      printf( "\nApellido:  " );
      gets( tabla.persona[i].apellido );
      printf( "\nDNI: " );
      scanf( "%ld", &tabla.persona[i].dni );
   }
}
```

8.2 Uniones

Al igual que las estructuras, las *uniones* también pueden contener múltiples campos de diferentes tipos de datos. Sin embargo, mientras que cada campo en una estructura posee su propia área de almacenamiento, en una unión, todos los campos que la componen se hallan almacenados en la misma área de memoria. El espacio de memoria reservado por el compilador corresponde al del campo de la unión que requiere mayor espacio de almacenamiento. El resto de campos comparten dicho espacio.

Así pues, los campos de una unión están solapados en memoria, por lo que, en un momento dado de la ejecución del programa, sólo podrá haber almacenado un dato en uno de los campos. Es responsabili-

dad del programador hacer que el programa mantenga control sobre qué campo contiene la información almacenada en la unión en cada momento. Intentar acceder al tipo de información equivocado puede producir resultados sin sentido.

La definición de una unión es muy similar a la de una estructura, excepto por el uso de la palabra reservada `union`:

```
union nombre_union
{
   tipo_campo_1 nombre_campo_1;
   tipo_campo_2 nombre_campo_2;
   .  .  .
   tipo_campo_N nombre_campo_N;
};
```

Tanto la declaración de variables de tipo unión como el acceso a los campos se expresa de forma similar a como se mostró en las estructuras.

Finalmente, diremos que una estructura puede ser el campo de una unión y viceversa. Igualmente, pueden definirse tablas de uniones, etc.

8.2.1 Ejemplo

El siguiente ejemplo define tres estructuras para almacenar la información asociada a tres tipos diferentes de máquinas voladoras (`jet`, `helicoptero` y `carguero`). Finalmente define una estructura genérica que puede contener, alternativamente, los datos de cualquiera de ellos (`un_aeroplano`). Para controlar de qué tipo es el objeto almacenado en la unión `datos`, se utiliza la variable `tipo`.

```
struct jet
{
   int num_pasajeros;
   .  .  .
};
struct helicoptero
{
   int capacidad_elevador;
   .  .  .
};
struct carguero
{
   int carga_maxima;
   .  .  .
};
union aeroplano
{
   struct jet jet_u;
   struct helicoptero helicopter_u;
   struct carguero carguero_u;
};
```

```
struct un_aeroplano
{
   /* 0:jet, 1:helicoptero, 2:carguero */
   int tipo;
   union aeroplano datos;
};
```

8.3 Tipos de datos enumerados

Un objeto enumerado consiste en un conjunto ordenado de constantes enteras cuyos nombres indican todos los posibles valores que se le pueden asignar a dicho objeto.

La definición de una "plantilla" de un objeto enumerado requiere especificar un nombre mediante el cual pueda identificarse, así como la lista de constantes de los posibles valores que puede tomar. Para ello se utiliza la palabra reservada `enum` de la forma siguiente:

```
enum nombre_enumeración { constante_1,
                          constante_2,
                          .  .  .
                          constante_N;
                        };
```

El siguiente ejemplo define una enumeración denominada `dia_semana`, cuyo significado es obvio.

```
enum dia_semana { LUNES, MARTES, MIERCOLES, JUEVES, VIERNES,
                  SABADO, DOMINGO
                };
```

Las constantes que definen los posibles valores de la enumeración son representadas internamente como constantes enteras. El compilador asigna a cada una de ellas un valor en función del orden (empezando por 0) en que aparecen en la definición. Así pues, en el ejemplo anterior tenemos que `LUNES` es 0, `MARTES` es 1, etc. Sin embargo, podría ser de interés modificar dicha asignación por defecto, asignando a las constantes otro valor numérico. Ver el siguiente ejemplo:

```
enum dia_semana { LUNES=2, MARTES=3, MIERCOLES=4,
                  JUEVES=5, VIERNES=6,
                  SABADO=7, DOMINGO=1
                };
```

También puede asignarse el mismo valor numérico a diferentes constantes, así como dejar que el compilador numere algunas por defecto. Ver el siguiente ejemplo:

```
enum dia_semana { LUNES=1, MARTES, MIERCOLES, JUEVES, VIERNES,
                  SABADO, DOMINGO=10, FESTIVO=10
                };
```

donde los valores asociados a las constantes son, respectivamente: 1, 2, 3, 4, 5, 6, 10 y 10.

La declaración de variables puede hacerse de dos formas diferentes. Bien en la misma definición de la enumeración, bien posteriormente. Por su similitud con la declaración de variables en el caso de estructuras y uniones, veremos simplemente un ejemplo del segundo caso. El mismo código muestra algunos ejemplos de utilización de las constantes definidas en un tipo enumerado.

```
enum dia_semana { LUNES=1, MARTES, MIERCOLES, JUEVES, VIERNES,
                  SABADO, DOMINGO=10, FESTIVO=10
                };

void main()
{
   enum dia_semana dia;

   .  .  .
   if (dia <= VIERNES)
      printf( "Es laborable" );
   .  .  .
   dia = MARTES;
   /* Muestra el valor de dia */
   printf( "Hoy es:  %d", dia );
   .  .  .
}
```

8.4 Definición de nuevos tipos de datos

C ofrece al programador la posibilidad de definir nuevos tipos de datos mediante la palabra reservada `typedef`. Los nuevos tipos definidos pueden utilizarse de la misma forma que los tipos de datos predefinidos por el lenguaje. Es importante destacar que `typedef` tiene especial utilidad en la definición de nuevos tipos de datos estructurados, basados en tablas, estructuras, uniones o enumeraciones.

La sintaxis general para definir un nuevo tipo de datos es la siguiente:

```
typedef tipo_de_datos nombre_nuevo_tipo;
```

De esta forma se ha definido un nuevo tipo de datos de nombre `nombre_nuevo_tipo` cuya estructura interna viene dada por `tipo_de_datos`.

A continuación se muestran algunos ejemplos de definición de nuevos tipos de datos, así como su utilización en un programa:

```
typedef float Testatura;

typedef char Tstring [30];

typedef enum Tsexo = { HOMBRE, MUJER };
```

```
typedef struct
{
   Tstring nombre, apellido;
   Testatura alt;
   Tsexo sex;
} Tpersona;

void main()
{
   Tpersona paciente;

   .  .  .
   scanf( "%f", &paciente.alt );
   gets( paciente.nombre );
   printf( "%s", paciente.apellido );
   paciente.sex = MUJER;
}
```

8.5 Tiras de bits

C es un lenguaje muy versátil que permite programar con un alto nivel de abstraccion. Sin embarco, C también permite programar a muy bajo nivel. Esta característica es especialmente útil, por ejemplo, para programas que manipulan dispositivos *hardware*, como el programa que controla el funcionamiento de un *modem*, etc. Este tipo de programas manipulan tiras de bits.

En C, una tira de bits se debe almacenar como una variable entera con o sin signo. Es decir, como una variable de tipo `char`, `short`, `int`, `long`, `unsigned`, etc. Seguidamente veremos las operaciones que C ofrece para la manipulación tiras de bits.

8.5.1 Operador de negación

Este operador también recibe el nombre de operador de *complemento a 1*, y se representa mediante el símbolo $\sim$. La funcion de este operador consiste en cambiar los 1 por 0 y viceversa. Por ejemplo, el siguiente programa:

```
#include <stdio.h>
void main()
{
   unsigned short a;

   a= 0x5b3c;      /* a = 0101 1011 0011 1100 */
   b= ~a;          /* b = 1010 0100 1100 0011 */
   printf( " a= %x     b= %x\n", a, b );
   printf( " a= %u     b= %u\n", a, b );
   printf( " a= %d     b= %d\n", a, b );
}
```

Tabla 8.1: Tabla de verdad de los operadores lógicos

x	y	**AND** (&)	**OR** (|)	**XOR** (^)
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

mostraría en pantalla los siguientes mensajes:

```
a= 0x5b3c     b= 0xa4c3
a= 23356      b= 42179
a= 23356      b= -23357
```

como resultado de intercambiar por sus complementarios los bits de la variable `a`.

8.5.2 Operadores lógicos

C permite realizar las operaciones lógicas *AND* (&), *OR* (|) y *XOR* (^), también llamadas respectivamente "Y", "O" y "O exclusivo". La tabla 8.1 muestra la tabla de verdad de estas operaciones, donde `x` e `y` son bits.

Cuando aplicamos estos operadores a dos variables la operación lógica se realiza bit a bit. Veámoslo en el siguiente ejemplo:

```
a= 0x5b3c;  /* a = 1101 1011 0001 1101 */
b= 0xa4c3;  /* b = 1010 0101 1100 1011 */
c= a & b;   /* c = 1000 0001 0000 1001 */
d= a | b;   /* d = 1111 1111 1101 1111 */
e= a ^ b;   /* e = 0111 1110 1101 0110 */
```

8.5.3 Operadores de desplazamiento de bits

Existen dos operadores de desplazamiento que permiten desplazar a derecha o izquierda una tira de bits. El operador de desplazamiento a la dercha se denota como `>>` y el operador de desplazamiento a la izquierda se denota como `<<`. Su uso es como sigue:

```
var1 << var2
var1 >> var2
```

donde `var1` es la tira de bits desplazada y `var2` es el número de bits desplazados. En el desplazamiento a la izquierda se pierden los bits de más a la izquierda de `var1`, y se introducen ceros por la derecha. De forma similar, en el desplazamiento a la derecha se pierden los bits de más a la derecha de `var1`, y por la izquierda se introducen, o bien ceros (si la variable es de tipo `unsigned`, o bien se repite el último bit (si la variable es de tipo `signed`).

El siguiente ejemplo muestra el resultado de aplicar los operadores de desplazamiento de bits:

```
unsigned short a, d, e;
short b, c, f, g;

a= 28;      /* a = 0000 0000 0001 1100 */
b= -28;     /* b = 1111 1111 1110 0100 */
c= 3;       /* c = 0000 0000 0000 0011 */
d= a << c;  /* d = 0000 0000 1110 0000 = 224 */
e= a >> c;  /* e = 0000 0000 0000 0011 = 3 */
f= b << c;  /* f = 1111 1111 0010 0000 = -224 */
g= b >> c;  /* g = 1111 1111 1111 1100 = -3 */
```

Es importante señalar que los operadores de desplazamiento tienen un significado aritmético en base 10. El desplazamiento a la derecha de 1 bit es equivalente a dividir por 2, obteniéndose el cociente de la division entera. El desplazamiento a la izquierda de 1 bit es equivalente a multiplicar por 2. Por lo tanto, cuando trabajemos con variables enteras:
`var << n` equivale a `var * ` 2^n , y `var >> n` equivale a `var / ` 2^n.

8.6 Ejercicios

1. Escribir un programa que, dadas dos fechas, indique cuál de las dos es anterior a la otra. Para ello, el programa deberá definir una estructura para el tipo de datos fecha (día, mes y año).

2. Definir una estructura de datos para representar polinomios de hasta grado 10. Escribir un programa capaz de sumar dos polinomios expresados con dicha estructura.

3. Dada la siguiente definición de tipos de datos:

   ```
   typedef char Tstring [50];

   typedef struct
   {
      Tstring nombre;
      Tstring area;
      long int ventas[4];
   } Tagente;
   ```

   Escribir un programa que defina un vector de N agentes, permita la introducción de los datos por teclado y, finalmente, muestre los datos del agente con la máxima media de ventas.

4. Dada la siguiente definición de tipo de datos:

   ```
   typedef struct
   {
      Tstring pais;
      int temp_max;
      int temp_min;
   } Ttemp;
   ```

escribir un programa que defina un vector con los datos de N países y permita introducirlos por teclado. Finalmente, el programa deberá mostrar el nombre de los países cuya temperatura mínima sea inferior a la media de las temperaturas mínimas.

5. Dada la siguiente definición de tipos de datos:

```
typedef char Tstring [50];

typedef struct
{
   long int num;
   char letra;
} Tnif;

typedef struct
{
   Tnif nif;
   Tstring nombre;
   Tstring direc;
   long int telf;
} Tempresa;
```

escribir un programa que defina un vector con los datos de N empresas y permita introducirlos por teclado. Dado el NIF de una empresa, el programa deberá permitir mostrar los datos de la misma, así como eliminarla del vector, reorganizándolo para no dejar espacios vacíos.

Capítulo 9

Punteros

Un puntero es una variable que contiene como valor una dirección de memoria. Dicha dirección corresponde habitualmente a la dirección que ocupa otra variable en memoria. Se dice entonces que el puntero *apunta a dicha variable*. La variable apuntada puede ser de cualquier tipo elemental, estructurado o incluso otro puntero.

Los punteros constituyen una parte fundamental del lenguaje C y son básicos para comprender toda la potencia y flexibilidad que ofrece el lenguaje. Son especialmente importantes en la programación a bajo nivel, donde se manipula directamente la memoria del ordenador. Algunas de las ventajas que aporta el uso de punteros en C son:

- Constituyen la única forma de expresar algunas operaciones.
- Su uso produce código compacto y eficiente.
- Son imprescindibles para el paso de parámetros por referencia a funciones.
- Tienen una fuerte relación con el manejo eficiente de tablas y estructuras.
- Permiten realizar operaciones de asignación dinámica de memoria y manipular estructuras de datos dinámicas.

Finalmente, cabe advertir al lector que los punteros son tradicionalmente la parte de C más difícil de comprender. Además deben usarse con gran precaución, puesto que al permitir manipular directamente la memoria del ordenador, pueden provocar fallos en el programa. Estos fallos suelen ser bastante difíciles de localizar y de solucionar.

9.1 Declaración y asignación de direcciones

En la declaración de punteros y la posterior asignación de direcciones de memoria a los mismos, se utilizan respectivamente los operadores unarios `*` y `&`. El operador `&` permite obtener la *dirección que ocupa una variable en memoria*. Por su parte, el operador de *indirección* `*` permite obtener el *contenido de un objeto apuntado por un puntero*.

9.1.1 Declaración

En la declaración de variables puntero se usa también el operador `*` , que se aplica directamente a la variable a la cual precede. El formato para la declaración de variables puntero es el siguiente:

```
tipo_de_datos * nombre_variable_puntero;
```

Nótese que un puntero debe estar asociado a un tipo de datos determinado. Es decir, no puede asignarse la dirección de un `short int` a un puntero a `long int`. Por ejemplo:

```
int *ip;
```

declara una variable de nombre `ip` que es un puntero a un objeto de tipo `int`. Es decir, `ip` contendrá direcciones de memoria donde se almacenaran valores enteros.

No debe cometerse el error de declarar varios punteros utilizando un solo `*`. Por ejemplo:

```
int *ip, x;
```

declara la variable `ip` como un puntero a entero y la variable `x` como un entero (no un puntero a un entero).

El tipo de datos utilizado en la declaración de un puntero debe ser del mismo tipo de datos que las posibles variables a las que dicho puntero puede apuntar. Si el tipo de datos es `void`, se define un puntero genérico de forma que su tipo de datos implícito será el de la variable cuya dirección se le asigne. Por ejemplo, en el siguiente código, `ip` es un puntero genérico que a lo largo del programa apunta a objetos de tipos distintos, primero a un entero y posteriormente a un carácter.

```
void *ip;
int x;
char car;
.  .  .
ip = &x;    /* ip apunta a un entero */
ip = &car;  /* ip apunta a un carácter */
```

Al igual que cualquier variable, al declarar un puntero, su valor no está definido, pues es el correspondiente al contenido aleatorio de la memoria en el momento de la declaración. Para evitar el uso indebido de un puntero cuando aún no se le ha asignado una dirección, es conveniente inicializarlo con el valor nulo `NULL`, definido en el fichero de la librería estándar `stdio.h`. El siguiente ejemplo muestra dicha inicialización:

```
int *ip = NULL;
```

De esta forma, si se intentase acceder al valor apuntado por el puntero `ip` antes de asignarle una dirección válida, se produciría un error de ejecución que interrumpiría el programa.

9.1.2 Asignación de direcciones

El operador `&` permite obtener la dirección que ocupa una variable en memoria. Para que un puntero apunte a una variable es necesario asignar la dirección de dicha variable al puntero. El tipo de datos de puntero debe coincidir con el tipo de datos de la variable apuntada (excepto en el caso de un puntero de tipo `void`). Para visualizar la dirección de memoria contenida en un puntero, puede usarse el modificador de formato `%p` con la función `printf`:

```
double num;
double *pnum = NULL;
.  .  .
pnum = &num;
printf( "La dirección contenida en 'pnum' es:  %p", pnum );
```

9.2 Indirección

Llamaremos *indirección* a la forma de referenciar el valor de una variable a través de un puntero que apunta a dicha variable. En general usaremos el término indirección para referirnos al hecho de referenciar el valor contenido en la posición de memoria apuntada por un puntero. La indirección se realiza mediante el operador `*`, que precediendo al nombre de un puntero indica el valor de la variable cuya dirección está contenida en dicho puntero. A continuación se muestra un ejemplo del uso de la indirección:

```
int x, y;
int *p;     /* Se usa * para declarar un puntero */
.  .  .
x = 20;
p = &x;
*p = 5498;  /* Se usa * para indicar el valor de la
               variable apuntada */
y = *p;     /* Se usa * para indicar el valor de la
               variable apuntada */
```

Después de ejecutar la sentencia `*p = 5498;` la variable `x`, apuntada por `p`, toma por valor `5498`. Finalmente, después de ejecutar la sentencia `y = *p;` la variable `y` toma por valor el de la variable `x`, apuntada por `p`.

Para concluir, existe también la indirección múltiple, en que un puntero contiene la dirección de otro puntero que a su vez apunta a una variable. El formato de la declaración es el siguiente:

```
tipo_de_datos ** nombre_variable_puntero;
```

En el siguiente ejemplo, `pnum` apunta a `num`, mientras que `ppnum` apunta a `pnum`. Así pues, la sentencia `**ppnum = 24;` asigna `24` a la variable `num`.

```
int num;
int *pnum;
int **ppnum;
.  .  .
pnum = &num;
ppnum = &pnum;
**ppnum = 24;
printf( "%d", num );
```

La indirección múltiple puede extenderse a más de dos niveles, aunque no es recomendable por la dificultad que supone en la legibilidad del código resultante.

La utilización de punteros debe hacerse con gran precaución, puesto que permiten manipular directamente la memoria. Este hecho puede provocar fallos inesperados en el programa, que normalmente son difíciles de localizar. Un error frecuente consiste en no asignar una dirección válida a un puntero antes de usarlo. Veamos el siguiente ejemplo:

```
int *x;
.  .  .
*x = 100;
```

El puntero `x` no ha sido inicializado por el programador, por lo tanto contiene una dirección de memoria aleatoria, con lo que es posible escribir involuntariamente en zonas de memoria que contengan código o datos del programa. Este tipo de error no provoca ningún error de compilación, por lo que puede ser difícil de detectar. Para corregirlo es necesario que `x` apunte a una posición controlada de memoria, por ejemplo:

```
int y;
int *x;
.  .  .
x = &y;
*x = 100;
```

9.3 Operaciones con punteros

En C pueden manipularse punteros mediante diversas operaciones como asignación, comparación y operaciones aritméticas.

Asignación de punteros

Es posible asignar un puntero a otro puntero, siempre y cuando ambos apunten a un mismo tipo de datos. Después de una asignación de punteros, ambos apuntan a la misma variable, pues contienen la misma dirección de memoria. En el siguiente ejemplo, el puntero `p2` toma por valor la dirección de memoria contenida en `p1`. Así pues, tanto `y` como `z` toman el mismo valor, que corresponde al valor de la variable `x`, apuntada tanto por `p1` como por `p2`.

```
int x, y, z;
int *p1, *p2;
.  .  .
x = 4;
p1 = &x;
p2 = p1;
y = *p1;
z = *p2;
```

Comparación de punteros

Normalmente se utiliza la comparación de punteros para conocer las posiciones relativas que ocupan en memoria las variables apuntadas por los punteros. Por ejemplo, dadas las siguientes declaraciones,

```
int *p1, *p2, precio, cantidad;
*p1 = &precio;
*p2 = &cantidad;
```

la comparación `p1 > p2` permite saber cuál de las dos variables (`precio` o `cantidad`) ocupa una posición de memoria mayor. Es importante no confundir esta comparación con la de los valores de las variables apuntadas, es decir, `*p1 > *p2`.

Aritmética de punteros

Si se suma o resta un número entero a un puntero, lo que se produce implícitamente es un incremento o decremento de la dirección de memoria contenida por dicho puntero. El número de posiciones de memoria incrementadas o decrementadas depende, no sólo del número sumado o restado, sino también del tamaño del tipo de datos apuntado por el puntero. Es decir, una sentencia como:

```
nombre_puntero = nombre_puntero + N;
```

se interpreta internamente como:

```
nombre_puntero = dirección + N * tamaño_tipo_de_datos;
```

Por ejemplo, teniendo en cuenta que el tamaño de un `float` es de 4 bytes, y que la variable `num` se sitúa en la dirección de memoria 2088, ¿cuál es el valor de `pnum` al final del siguiente código?

```
float num, *punt, *pnum;
.  .  .
punt = &num;
pnum = punt + 3;
```

La respuesta es 2100. Es decir, $2088 + 3 * 4$.

Es importante advertir que aunque C permite utilizar aritmética de punteros, esto constituye una práctica *no recomemdable*. Las expresiones aritméticas que manejan punteros son difíciles de entender y generan confusión, por ello son una fuente inagotable de errores en los programas. Como veremos en la siguiente sección, no es necesario usar expresiones aritméticas con punteros, pues C proporciona una notación alternativa mucho más clara.

9.4 Punteros y tablas

En el lenguaje C existe una fuerte relación entre los punteros y las estructuras de datos de tipo tabla (vectores, matrices, etc.).

En C, el nombre de una tabla se trata internamente como un puntero que contiene la dirección del primer elemento de dicha tabla. De hecho, el nombre de la tabla es una constante de tipo puntero, por

lo que el compilador no permitirá que las instrucciones del programa modifiquen la dirección contenida en dicho nombre. Así pues, dada una declaración como `char tab[15]`, el empleo de `tab` es equivalente al empleo de `&tab[0]`. Por otro lado, la operación `tab = tab + 1` generará un error de compilación, pues representa un intento de modificar la dirección del primer elemento de la tabla.

C permite el uso de punteros que contengan direcciones de los elementos de una tabla para acceder a ellos. En el siguiente ejemplo, se usa el puntero `ptab` para asignar a `car` el tercer elemento de `tab`, leer de teclado el quinto elemento de `tab` y escribir por pantalla el décimo elemento del vector `tab`.

```
char car;
char tab[15];
char *ptab;
.  .  .
ptab = &tab;
ptab = ptab + 3;
car = *(ptab);    /* Equivale a car = tab[3]; */
scanf( "%c", ptab+5 );
printf( "%c", ptab+10 );
```

Pero la relación entre punteros y tablas en C va aún más allá. Una vez declarado un puntero que apunta a los elementos de una tabla, pueden usarse los corchetes para indexar dichos elementos, como si de una tabla se tratase. Así, siguiendo con el ejemplo anterior, sería correcto escribir:

```
scanf( "%c", ptab[0] );/* ptab[0] equivale a tab[0] */
ptab[7] = 'R';         /* ptab[7] equivale a *(ptab +7) */
```

Por lo tanto, vemos que no es necesario usar expresiones aritméticas con punteros; en su lugar usaremos la sintaxis de acceso a una tabla. Es importante subrayar que este tipo de indexaciones sólo son válidas si el puntero utilizado apunta a los elementos de una tabla. Además, no existe ningún tipo de verificación al respecto, por lo que es responsabilidad del programador saber en todo momento si está accediendo a una posición de memoria dentro de una tabla o ha ido a parar fuera de ella y está sobreescribiendo otras posiciones de memoria.

El siguiente ejemplo muestra el uso de los punteros a tablas para determinar cuál de entre dos vectores de enteros es más *fuerte*. La *fuerza* de un vector se calcula como la suma de todos sus elementos.

```
#include <stdio.h>
#define DIM 10
void main()
{
   int v1[DIM], v2[DIM];
   int i, fuerza1, fuerza2;
   int *fuerte;

   /* Lectura de los vectores.  */
   for (i= 0; i< DIM; i++)
      scanf( "%d ", &v1[i] );
```

```
    for (i= 0; i< DIM; i++)
       scanf( "%d ", &v2[i] );

    /* Cálculo de la fuerza de los vectores.  */
    fuerza1 = 0;
    fuerza2 = 0;
    for (i= 0; i< DIM; i++)
    {
       fuerza1 = fuerza1 + v1[i];
       fuerza2 = fuerza2 + v2[i];
    }

    if (fuerza1 > fuerza2)
       fuerte = v1;
    else
       fuerte = v2;

    /* Escritura del vector más fuerte.  */
    for (i= 0; i< DIM; i++)
       printf( "%d ", fuerte[i] );
}
```

En el caso de usar punteros para manipular tablas multidimensionales, es necesario usar fórmulas de transformación para el acceso a los elementos. Por ejemplo, en el caso de una matriz de n filas y m columnas, el elemento que ocupa la fila i y la columna j se referencia por medio de un puntero como `puntero[i*m+j]`. En el siguiente ejemplo se muestra el acceso a una matriz mediante un puntero.

```
float mat[3][5];
float *pt;

pt = mat;
pt[i*5+j] = 4.6; /* Equivale a mat[i][j]=4.6 */
```

Cuando usamos el puntero para acceder a la matriz, la expresión `pt[k]` significa acceder al elemento de tipo `float` que está `k` elementos por debajo del elemento apuntado por `pt`. Dado que en C las matrices se almacenan por filas, para acceder al elemento de la fila i columna j deberemos contar cuantos elementos hay entre el primer elemento de la matriz y el elemento $[i][j]$. Como la numeración comienza en cero, antes de la fila i hay exactamente i filas, y cada una tiene m columnas. Por lo tanto hasta el primer elemento de la fila i tenemos $i \times m$ elementos. Dentro de la fila i, por delante del elemento j, hay j elementos. Por lo tanto tenemos que entre `mat[0][0]` y `mat[i][j]` hay $i \times m \times j$ elementos. La figura 9.1 muestra como está dispuesta en memoria la matriz de este ejemplo y una explicación gráfica del cálculo descrito.

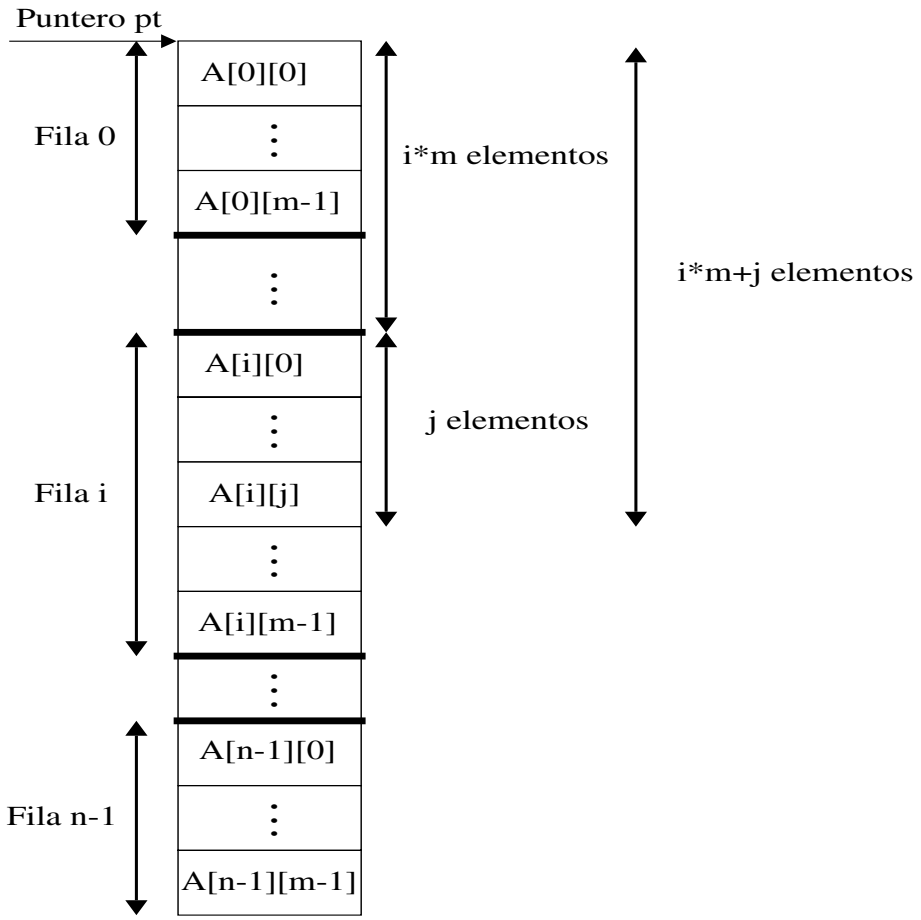


Figura 9.1: Acceso a una matriz mediante un puntero

Punteros y cadenas de caracteres

Una cadena de caracteres puede declararse e inicializarse sin necesidad de especificar explícitamente su longitud. La siguiente declaración es un ejemplo:

```
char mensaje[] = "Reiniciando sistema";
```

La cadena de caracteres `mensaje` ocupa en memoria 20 bytes (19 más el carácter nulo `'\0'`). El nombre `mensaje` corresponde a la dirección del primer carácter de la cadena. En esta declaración, `mensaje` es un puntero constante, por lo que no puede modificarse para que apunte a otro carácter distinto del primer carácter de la cadena. Sin embargo, puede usarse `mensaje` para acceder a caracteres individuales dentro de la cadena, mediante sentencias como `mensaje[13]='S'`, etc.

La misma declaración puede hacerse usando un puntero:

```
char *pmens = "Reiniciando sistema";
```

En esta declaración, `pmens` es una variable puntero inicializada para apuntar a una cadena constante, que como tal no puede modificarse. Sin embargo, el puntero `pmens` puede volver a utilizarse para apuntar a otra cadena.

Puede decirse que en general a una tabla puede accederse tanto con índices como con punteros (usando la aritmética de punteros). Habitualmente es más cómodo el uso de índices, sin embargo en el paso de parámetros a una función (ver Cap. 10) donde se recibe la dirección de una tabla, es necesario utilizar punteros. Como veremos, dichos punteros podrán usarse como tales o usando la indexación típica de las tablas.

9.5 Punteros y estructuras

Los punteros a estructuras funcionan de forma similar a los punteros a variables de tipos elementales, con la salvedad de que en las estructuras se emplea un operador específico para el acceso a los campos.

Puede accederse a la dirección de comienzo de una variable estructura en memoria mediante el empleo del operador de dirección `&`. Así pues, si `var` es una variable estructura, entonces `&var` representa la dirección de comienzo de dicha variable en memoria. Del mismo modo, puede declararse una variable puntero a una estructura como:

```
tipo_estructura *pvar;
```

donde `tipo_estructura` es el tipo de datos de las estructuras a las que `pvar` puede apuntar. Así pues, puede asignarse la dirección de comienzo de una variable estructura al puntero de la forma habitual:

```
pvar = &var;
```

Veamos un ejemplo. A continuación se define, entre otros, el tipo de datos `Tnif` como:

```
typedef char Tstring [50];

typedef struct
{
   long int num;
   char letra;
} Tnif;

typedef struct
{
   Tnif nif;
   Tstring nombre;
   Tstring direc;
   long int telf;
} Tempresa;
```

De forma que en el programa puede declararse una variable `cliente` de tipo `Tnif` y un puntero `pc` a dicho tipo, así como asignar a `pc` la dirección de inicio de la variable estructura `cliente`:

```
Tnif cliente;
Tnif *pc;
.  .  .
pc = &cliente;
```

Para acceder a un campo individual en una estructura apuntada por un puntero puede usarse el operador de indirección `*` junto con el operador punto (`.`) habitual de las estructuras. Vemos un ejemplo:

```
(*pc).letra = 'Z';
scanf( "%ld", &(*pc).num );
```

Los paréntesis son necesarios ya que el operador punto (`.`) tiene más prioridad que el operador `*`. Nótese que el operador de dirección `&` en la llamada a `scanf` se refiere al campo `num` y no al puntero `pc`.

El hecho de que sea obligatorio usar paréntesis en esta notación hace que se generen errores debido al olvido de los paréntesis adecuados. Para evitar estos errores, C posee otra notación para acceder a los campos de una estructura apuntada por un puntero.

El acceso a campos individuales puede hacerse también mediante el operador especial de los punteros a estructuras `->` (guión seguido del símbolo de mayor que). Así pues, puede escribirse el mismo ejemplo anterior como:

```
pc->letra = 'Z';
scanf( "%ld", &pc->num );
```

El operador `->` tiene la misma prioridad que el operador punto (`.`). *Es recomendable usar el operador* `->` *al manejar* `structs` *apuntados por punteros.*

El acceso a los campos de una estructura anidada a partir de un puntero puede hacerse combinando los operadores de punteros con el punto (`.`). Veamos el siguiente ejemplo:

```
Tempresa emp;
Tempresa *pe;
char inicial;
.  .  .
pe = &emp;
pe->nif.letra = 'Z';
scanf( "%ld", &pe->nif.num );
gets( pe->nombre );
inicial = pe->nombre[0];
.  .  .
```

9.6 Ejercicios

1. Escribir un programa que pase a mayúsculas la inicial de todas las palabras en una cadena de caracteres. Usar un puntero a dicha cadena para acceder a los elementos de la misma.

2. Escribir un programa que calcule el máximo de un vector de números enteros, utilizando un puntero para acceder a los elementos del vector.

3. Escribir un programa que, usando punteros, copie en orden inverso una cadena de caracteres en otra.

4. Un programa en C contiene las siguientes instrucciones:

```
char u, v;
char *pu, *pv;
.  .  .
v = 'A';
pv = &v;
*pv = v + 1;
u = *pv + 1;
pu = &u;
```

Si cada carácter ocupa 1 byte de memoria y la variable `u` se sitúa en la dirección `FC8` (hexadecimal), responder a las siguientes preguntas:

(a) ¿Qué valor representa `&v` ?

(b) ¿Qué valor se asigna a `pv` ?

(c) ¿Qué valor representa `*pv` ?

(d) ¿Qué valor se asigna a `u` ?

(e) ¿Qué valor representa `&u` ?

(f) ¿Qué valor se asigna a `pu` ?

(g) ¿Qué valor representa `*pu` ?

5. Un programa en C contiene las siguientes instrucciones:

```
float a = 0.1, b = 0.2;
float c, *pa, *pb;
.  .  .
pa = &a;
*pa = 2 * a;
pb = &b;
c = 5 * (*pb - *pa);
```

Si cada variable de tipo `float` ocupa 4 bytes de memoria y la variable `a` se sitúa en la dirección `1230` (hexadecimal), responder a las siguientes preguntas:

(a) ¿Qué valor representa `&a` ?

(b) ¿Qué valor representa `&c` ?

(c) ¿Qué valor representa `&pb` ?

(d) ¿Qué valor se asigna a `pa` ?

(e) ¿Qué valor representa `*pa` ?

(f) ¿Qué valor representa `&(*pa)` ?

(g) ¿Qué valor se asigna a `pb` ?

(h) ¿Qué valor representa `*pb` ?

(i) ¿Qué valor se asigna a `c` ?

6. Un programa de C contiene las siguientes sentencias:

```
int i, j = 25;
int *pi, *pj = &j;
.  .  .
*pj = j + 5;
i = *pj + 5;
pi = pj;
*pi = i + j;
```

Suponiendo que cada variable entera ocupa 2 bytes de memoria. Si la variable `i` se sitúa en la dirección 1000 y la variable `j` en la dirección 1002, entonces:

(a) ¿Qué valor representan `&i` y por `&j` ?

(b) ¿Qué valor se asigna a `pj` , `*pj` y a `i` ?

(c) ¿Qué valor representa `pi` ?

(d) ¿Qué valor se asigna a `*pi` ?

(e) ¿Qué valor representa `pi + 2` ?

(f) ¿Qué valor representa la expresión `(*pi + 2)` ?

(g) ¿Qué valor representa la expresión `*(pi + 2)` ?

Capítulo 10

Funciones

Hasta ahora hemos visto como el programa principal (`main( )`) utiliza funciones de la librería estándar de C para realizar algunas tareas comunes (`printf( )`, `scanf( )`, ...). C permite también la definición de funciones por parte del programador. Como veremos, al usar funciones definidas por el programador, los programas pueden estructurarse en partes más pequeñas y sencillas. Cada una de estas partes debe responder a un propósito único e identificable, pudiendo además utilizarse en distintos lugares del programa. La distribución del código de un programa usando funciones se conoce como *modularización*.

El diseño modular de programas ofrece diversas ventajas. Por ejemplo, muchos programas requieren la ejecución de un mismo grupo de instrucciones en distintas partes del programa. Este grupo de instrucciones puede incluirse dentro de una sola función, a la que se puede *llamar* cuando sea necesario. Además, puede proporcionarse un conjunto de datos (*parámetros*) diferente a la función cada vez que se la llama.

Es importante también la claridad en la lógica del programa resultante de la descomposición del mismo en partes bien definidas. Un programa concebido de esta forma es mucho más fácil de escribir y de depurar, no sólo por el propio programador, sino también (y más importante) por otros programadores que posteriormente deban mantener el programa. Este hecho es especialmente cierto en programas grandes donde participan muchos programadores.

La utilización de funciones permite también la construcción *a medida* de librerías de funciones de uso frecuente. Por ejemplo, un programador especializado en el desarrollo de programas matemáticos podría crear una librería con funciones para el manejo de matrices, números complejos, etc. De esta forma se evita la reescritura del mismo código repetidas veces para distintos programas.

10.1 Generalidades

Una función es una porción de programa, identificable mediante un nombre, que realiza determinadas tareas bien definidas por un grupo de sentencias sobre un conjunto de datos. Las operaciones que realiza la función son siempre las mismas, pero los datos pueden variar cada vez que se llame a la función.

Todo programa en C consta de una o más funciones, una (y sólo una) de la cuales debe llamarse `main`. La ejecución del programa comienza siempre en dicha función, desde donde puede llamarse a otras funciones de la librería estándar o definidas por el programador. Al llamar a una función se

ejecutan las sentencias que la componen y, una vez completada la ejecución de la función, la ejecución del programa continúa desde el punto en que se hizo la llamada a la función.

Generalmente, al llamar a una función se le proporciona un conjunto de datos (parámetros) que se procesan ejecutando las sentencias que la componen. La función devuelve un solo valor mediante la sentencia `return`. Algunas funciones reciben parámetros, pero no devuelven nada (como la función `printf`), mientras que otras no reciben parámetros pero sí devuelven un valor (como la función `rand`).

El programa del siguiente ejemplo calcula el número combinatorio $\begin{pmatrix} m \\ n \end{pmatrix} = \frac{m!}{n!\,(m-n)!}$, donde es necesario calcular el factorial de tres número diferentes. Una posible realización de este programa, si no se tuviese en cuenta el uso de funciones, sería la siguiente:

```
#include <stdio.h>
void main()
{
   long int m, n, fm = 1, fn = 1, fdif = 1;
   float res;
   int i;

   printf( "Introduzca m y n:  " );
   scanf( "%d %d", &m, &n );
   for (i= 2; i<= m; i++)
      fm = fm * i;
   for (i= 2; i<= n; i++)
      fn = fn * i;
   for (i= 2; i<= m-n; i++)
      fdif = fdif * i;
   res = (float) fm / ((float)fn* (float)fdif);
   printf( "m sobre n = %f\n", res );
}
```

Como puede verse, el código para el cálculo del factorial se halla triplicado. Una solución más clara y elegante puede obtenerse usando funciones:

```
#include <stdio.h>
long int fact ( int x )
{
   int i;
   long int f = 1;

   for (i= 2; i<= x; i++)
      f = f * i;
   return(f);
}

void main()
```

```
{
   long int m, n;
   float res;

   printf( "Introduzca m y n:  " );
   scanf( "%d %d", &m, &n );
   res = (float) fact(m) / ((float)fact(n)*(float)fact(m-n));
   printf( "m sobre n = %f\n", res );
}
```

En el ejemplo se ha definido la función `fact`, que recibe como parámetro un valor de tipo `int`, al que se ha llamado `x`, y devuelve un resultado de tipo `long int`. El código en el interior de la función calcula el factorial de `x` acumulándolo sobre la variable local `f`. Finalmente la función devuelve el resultado del cálculo mediante la sentencia `return`. Obsérvese que la definición de una función se asemeja a la del programa principal. De hecho, el programa principal `main` es una función.

Seguidamente veremos más formalmente cómo definir funciones, cómo llamarlas, las distintas variantes del paso de parámetros, etc.

10.2 Definición y llamada

10.2.1 Definición

La *definición* de una función se hace de forma similar a la de la función `main`. Su forma más genérica consta básicamente de dos partes: un línea llamada *cabecera* donde se especifica el nombre de la función, el tipo del resultado que devuelve y los parámetros que recibe; y un conjunto de sentencias encerrado entre llaves formando el *cuerpo*.

```
tipo nombre_función(tipo1 param1, ..., tipoN paramN)
{
   cuerpo
}
```

- `tipo`: es el tipo de datos del valor que devuelve la función. Si no se especifica ninguno, C asume que la función devuelve un valor de tipo entero.
- `nombre_función`: identificador que se usará posteriormente para llamar a la función.
- `tipoi parami`: tipo y nombre de cada uno de los parámetros que recibe la función. Se especifican entre paréntesis y separados por comas. Algunas funciones pueden no tener parámetros. Los parámetros de la declaración se denominan *parámetros formales*, ya que representan los nombres con que referirse dentro de la función a los datos que se transfieren a ésta desde la parte del programa que hace la llamada.
- `cuerpo`: conjunto de declaración de variables y sentencias de ejecución (incluyendo llamadas a funciones) necesarias para la realización de la tarea especificada por la función. Debe incluir una o más sentencias `return` para devolver un valor al punto de llamada.

10.2.2 Prototipos

Si en el punto del programa donde se va a realizar una llamada a una función, dicha función ya ha sido definida previamente, entonces ya se conocen sus características (tipo del resultado, número y tipo de los parámetros, etc.), por lo que la llamada puede realizarse sin problemas. Sin embargo, si la función que se va a llamar se halla definida posteriormente al punto desde donde se realiza la llamada, entonces debe crearse un *prototipo* de la función a la cual se desea llamar. Dicho prototipo deberá colocarse antes del punto donde se haga la primera llamada a la función, y consta únicamente de la cabecera de dicha función.

El prototipo de una función puede interpretarse como un aviso al compilador, para que cuando encuentre una llamada a dicha función pueda conocer el tipo del resultado que devuelve y la información sobre los parámetros que recibe.

A continuación se muestra el formato general de un prototipo, que corresponde a la cabecera de la función seguida de un punto y coma:

```
tipo nombre_función(tipo1 param1, ..., tipoN paramN);
```

De acuerdo con esto, el programa del ejemplo anterior podría haberse escrito de otro modo, definiendo la función `fact` con posterioridad a `main` y usando un prototipo:

```
#include <stdio.h>
long int fact ( int x ); /* Prototipo */

void main()
{
   long int m, n;
   float res;

   printf( "Introduzca m y n:  " );
   scanf( "%d %d", &m, &n );
   res = (float) fact(m) / ((float)fact(n)*(float)fact(m-n));
   printf( "m sobre n = %f\n", res );
}

long int fact ( int x )
{
   int i;
   long int f = 1;

   for (i= 2; i<= x; i++)
      f = f * i;
   return(f);
}
```

La utilización de prototipos de funciones no es obligatorio en C. Sin embargo, es aconsejable, ya que facilitan la comprobación y detección de errores entre las llamadas a funciones y las definiciones correspondientes.

10.2.3 Llamada

Finalmente, la *llamada* a una función se realiza con el nombre de la misma y una lista de parámetros (si es que los requiere) entre paréntesis. El número y tipo de los parámetros empleados en la llamada a la función debe coincidir con el número y tipo de los parámetros formales de la definición o prototipo. Adicionalmente, si la función devuelve algún valor (es decir, no es de tipo `void`) la llamada a la función debe estar incluida en una expresión que recoja el valor devuelto. Siguiendo con el ejemplo del factorial:

```
fm = fact(m);
prod = fact(n)*fact(m-n);
```

Los datos empleados en la llamada a una función reciben el nombre de *parámetros reales*, ya que se refieren a la información que se transfiere a la función para que ésta se ejecute. Como veremos más adelante, los identificadores de los parámetros formales son *locales* a la función, en el sentido de que no son reconocidos desde fuera de ésta. Por tanto, los nombres de los parámetros formales no tienen por qué coincidir con los nombres de la variables usadas como parámetros reales en el momento de la llamada.

10.3 Variables y parámetros

Las variables de un programa pueden clasificarse en función del *ámbito* en el cual son conocidas y por tanto accesibles. El ámbito de las variables, así como su *tiempo de vida*, depende del lugar donde se hallen declaradas dentro del programa. Así pues, se distinguen los siguientes tipos: variables locales, variables globales y parámetros formales.

10.3.1 Variables locales

Una variable local se halla declarada al comienzo del cuerpo de una función (esto incluye a la función `main`). Su ámbito se circunscribe al bloque de sentencias que componen el cuerpo de la función, por lo que sólo son conocidas dentro de él. Por otra parte, su tiempo de vida va desde que se entra en la función hasta que se sale de ella, por lo que las variables locales se crean al comenzar la ejecución de la función y se destruyen al concluir dicha ejecución.

En el ejemplo del cálculo de $\binom{m}{n}$, las variables `i` y `f` son locales a la función `fact`, por lo que no son accesibles fuera de ella. De forma similar, las variables `m`, `n` y `res` son locales a la función `main`.

10.3.2 Variables globales

Una variable global se halla declarada fuera de toda función del programa al principio del fichero principal. Su ámbito se extiende a lo largo de todas las funciones del programa. Su tiempo de vida está limitado por el tiempo que dura la ejecución del programa, por lo que las variables globales se crean al comenzar a ejecutar el programa y se destruyen cuando éste concluye la ejecución.

Si todas las funciones del programa están incluidas en un mismo fichero, basta con escribir una sola vez al principio del fichero la declaración de las variables globales. Sin embargo, si tenemos las

funciones repertidas en diferentes ficheros, deberemos incluir en uno de los ficheros la declaración y repetir en los demás ficheros la declaración precedida de la palabra `extern`. También es posible definir variables globales únicamente dentro de un fichero. Para ello antepondremos en la declaración la palabra `static`.

El siguiente ejemplo muestra las declaraciones de tres variables globales. La variable `A` es accesible en todo el programa y está declarada en este fichero. La variable `B` es accesible en todo el programa, pero está declarada en otro fichero, es decir, hay otras funciones además de las de este fichero que pueden acceder a ella. Finalmente, la variable `C` sólo es accesible por las funciones de este fichero.

```
int A;
extern int B;
static int C;
void main ()
{
   .  .  .
}
int func1()
{
   .  .  .
}
int func2()
{
   .  .  .
}
```

El uso de variables globales debe hacerse con precaución, puesto que al poderse modificar desde cualquier función del programa donde sean accesibles, pueden producirse *efectos laterales* difíciles de detectar y corregir. *Es una buena práctica de programación no emplear variables globales salvo en casos muy excepcionales*. En general, el uso de una variable global puede substituirse por una variable local al programa principal y el adecuado paso de parámetros a todas las funciones que requieran acceder a dicha variable.

10.3.3 Parámetros formales

El ámbito y el tiempo de vida de un parámetro formal en una función son los mismos que los de una variable local a dicha función. Es decir, que el ámbito es toda la función y que la variable se crea al entrar en la función y se destruye al salir de la misma.

Como hemos comentado, el ámbito de las variables locales y los parámetros de una función se circunscribe únicamente al interior de la misma. Es decir, que ni las variables ni los parámetros formales de una función son accesibles desde fuera de ella. Incluso en el caso de que el programa use el mismo nombre para variables de distintas funciones, el compilador es capaz de diferenciarlas. Para demostrar ésto usaremos el operador de dirección `&` en un ejemplo muy sencillo:

```
void func( int par )
{
   int loc = 10;
   printf( "En func(), loc=%d y &loc=%p\n", loc, &loc );
   printf( "En func(), par=%d y &par=%p\n", par, &par );
}
void main()
{
   int loc = 24, par = 5;
   printf( "En main(), loc=%d y &loc=%p\n", loc, &loc );
   printf( "En main(), par=%d y &par=%p\n", par, &par );
   func(loc);
}
```

Al compilar y ejecutar este ejemplo, observaremos que las variables `loc` y `par` se sitúan en direcciones de memoria diferentes (y por tanto, son variables diferentes) si estamos dentro de la función o en el programa principal.

10.4 Devolución de resultados

Cuando una función termina de realizar la tarea para la que fue diseñada, devuelve el control de la ejecución a la parte del programa desde donde se hizo la llamada. Para concluir la ejecución de una función se utiliza la sentencia `return`, que fuerza la salida de la función en el punto donde se ha especificado dicha sentencia. Si no existe ninguna sentencia `return`, la función termina con la llave que cierra el cuerpo.

El siguiente ejemplo muestra dos maneras de escribir la misma función para el cálculo del máximo de dos números enteros:

```
void maximo( int x, int y )
{
   int max;

   if (x > y)
      max = x;
   else
      max = y;
   printf( "MAX=%d", max );
}
```

```
void maximo( int x, int y )
{
   if (x > y)
   {
      printf( "MAX=%d", x );
      return;
   }
   printf( "MAX=%d", y );
}
```

Si la función no es de tipo `void`, la sentencia `return`, además de especificar la terminación de la función, puede especificar la devolución de un valor para que pueda ser utilizado en el punto donde se hizo la llamada. El valor devuelto, y por ende el tipo de la función, puede ser cualquiera de los tipos elementales, estructurados o definidos por el programador, excepto tablas. Aunque en C es legal que una función retorne cualquier tipo de estructura, por cuestiones de eficiencia en la ejecución del programa no es conveniente retornar estructuras muy grandes en tamaño. Por ejemplo, estructuras en cuyo interior haya tablas. Cuando se desea que una función *devuelva* tipos de datos complejos, se utiliza el paso de parámetros por referencia que veremos en la sección 10.5.

En el siguiente programa se ha rescrito la función `maximo` para que devuelva el máximo de dos enteros, en lugar de mostrarlo por pantalla.

```
int maximo( int x, int y )
{
   if (x > y)
      return(x);
   else
      return(y);
}
```

En la parte del programa que hace la llamada puede usarse el valor devuelto por la función dentro de cualquier expresión válida (en particular una sentencia de escritura):

```
printf( "MAX(%d,%d)=%d", a, b, maximo(a,b) );
```

10.5 Paso de parámetros

Como ya hemos comentado, los parámetros de una función no son más que variables que actúan de enlace entre la parte del programa donde se realiza la llamada y el cuerpo de la función. Así pues, los parámetros formales de una función son variables locales a ésta. Como tales, se crean y reciben sus valores al entrar en la función, y se destruyen al salir de la misma. El paso de parámetros puede realizarse de dos formas: por valor o por referencia.

10.5.1 Paso de parámetros por valor

Al entrar a la función, los parámetros formales reciben una copia del valor de los parámetros reales. Por tanto las modificaciones sobre los parámetros formales son locales y no afectan a los parámetros reales de la parte del programa que hace la llamada a la función. Veamos un ejemplo:

```
#include <stdio.h>
void cuadrado ( int x )
{
   x = x * x;
   printf( "Dentro x = %d\n", x );
}
void main()
{
   int x = 5;

   printf( "Antes x = %d\n", x );
   cuadrado( x );
   printf( "Después x = %d\n", x );
}
```

El resultado de ejecutarlo será:

```
Antes x = 5
Dentro x = 25
Después x = 5
```

Como puede verse, la modificación sobre el parámetro formal no afecta a la variable del programa principal.

En el paso de parámetros por valor, la transferencia de información es sólo en un sentido, es decir, desde la parte del programa donde se hace la llamada hacia el interior de la función, pero no al revés. Gracias a ello, es posible utilizar expresiones como parámetros reales, en lugar de necesariamente variables. Esto es así puesto que el valor asignado al parámetro formal es el resultado de la evaluación de dicha expresión. Es más, si el parámetro real es una variable, su valor es protegido de posibles modificaciones por parte de la función.

10.5.2 Paso de parámetros por referencia

Hasta este punto hemos visto cómo proporcionar datos a una función (paso de parámetros por valor) o cómo hacer que la función devuelva resultados con la sentencia `return`. Sin embargo, ¿cómo podríamos hacer que una función devolviese más de un valor? O bien, ¿cómo podríamos conseguir que las modificaciones sobre los parámetros formales afectasen también a los parámetros reales? Es decir, ¿cómo podríamos modificar variables del ámbito en el que se realizó la llamada desde el interior de una función?

La respuesta a estas cuestiones se halla en la utilización del paso de parámetros por referencia. Este tipo de paso de parámetros se conoce también en C como *paso por dirección* o *paso por puntero*.

En este caso los parámetros reales son una *referencia* (puntero) a las variables de la parte del programa que realiza la llamada y no las variables en sí. La referencia se copia en los parámetros formales, de forma que dentro de la función puede usarse dicha referencia para modificar una variable que, de otra forma, no sería accesible desde el interior de la función (modificar el valor referenciado).

El paso de parametros por referencia implica el uso de los operadores de dirección (`&`) y puntero (`*`) de C:

- `&` , que antepuesto a una variable permite obtener la dirección de memoria en que se halla ubicada. Se usará en los parámetros reales de la llamada a una función para pasarle por referencia dicha variable. En otras palabras, el parámetro que se pasa a la función es un puntero a una variable. Ya hemos utilizado esto anteriormente en las llamadas a la función `scanf`, donde las variables en que se almacenan los valores leídos del teclado se pasan por referencia.

- `*` , que se utiliza tanto en la declaración de los parámetros formales de la función como en el cuerpo de la misma. Aparecerá precediendo al nombre de un parámetro formal en la cabecera para indicar que dicho parámetro será pasado por referencia (será un puntero). Aparecerá en el cuerpo de la función, antepuesto al nombre de un parámetro formal, para acceder al valor de la variable externa a la función y referenciada por el parámetro formal.

En el siguiente ejemplo se muestra la función `swap` que intercambia el valor de sus dos parámetros. Para ello los parámetros formales `x` e `y` se han declarado de paso por referencia usando `*` en la cabecera. De forma complementaria, en la llamada a la función se ha usado `&` para pasar a la función una referencia a las variables `a` y `b` del programa principal. Dentro de la función usamos nuevamente

`*` para referenciar el valor del parámetro real. Por ejemplo, la sentencia `aux = *x;` asigna a la variable local `aux` el valor de la variable `a` del programa principal, puesto que el parámetro formal `x` contiene una referencia a la variable `a`.

```
#include <stdio.h>
void swap(int *x, int *y)
{
   int aux;

   /* Se asigna a aux el valor referenciado por x */
   aux = *x;
   /* Se asigna el valor referenciado por y
      al valor referenciado por x */
   *x = *y;
   /* El valor referenciado por y pasa a ser el valor de aux */
   *y = aux;
}
void main()
{
   int a, b;

   scanf( "%d %d", &a, &b );
   swap( &a, &b );
   printf( "Los nuevos valores son a=%d y b=%d\n", a, b );
}
```

Como puede verse, el paso de parámetros por referencia tiene una estrecha relación con el uso de punteros y direcciones de memoria que vimos en el capítulo 9. De hecho un purista del lenguaje diría que en C no existe el paso por referencia y que todo paso de parámetros se hace por valor. El paso por referencia se "simula" mediante el paso (por valor) de punteros a las variables externas a la función y que se desean modificar desde el interior de la misma.

10.5.3 Las tablas y las funciones

Los elementos individuales de una tabla (vector, matriz, etc.) se pasan a las funciones como si de variables individuales se tratase, tanto por valor como por referencia.

El siguiente ejemplo busca el elemento mayor en un vector de números enteros. Para ello utiliza la función `maximo` que vimos anteriormente. Obsérvese el paso por referencia del elemento i-ésimo de un vector, a la función `scanf` y, por valor, a la función `maximo`.

```
#include <stdio.h>
void main()
{
   int max, v[20], i;

   printf( "Introducir elementos del vector:\n" );
   for (i= 0; i< 20; i++)
```

```
      scanf( "%d", &v[i] );
   max = v[0];
   for (i= 1; i< 20; i++)
      max = maximo( max, v[i] );
   printf( "El elemento mayor es:  %d\n", max );
}
```

Algo muy diferente es el paso de una tabla a una función. La única manera que C ofrece para ello es el paso por referencia de toda la tabla en cuestión. Sin embargo, al contrario que en el paso por referencia habitual, no se usan los símbolos `&` y `*`. En su lugar se utiliza directamente el nombre de la tabla, que constituye de por sí una referencia al primer elemento de la tabla, tal como vimos en el capítulo 9. Por lo tanto, el parámetro formal de la función debe ser un puntero para poder recoger la dirección de inicio de la tabla.

El siguiente ejemplo define una función para el cálculo de la media de los elementos de un vector de números de coma flotante:

```
#include <stdio.h>
#define DIM 20
float media( float vec[], int n )
{
   int j;
   float sum;

   sum = 0.0;
   for (j= 0; j< n; j++)
      sum = sum + vec[j];
   return (sum/(float)n);
}
void main()
{
   float med, v[DIM];
   int i;

   printf( "Introducir elementos del vector:\n" );
   for (i= 0; i< DIM; i++)
      scanf( "%f", &v[i] );
   med = media( v, DIM );
   printf( "La media es:  %f\n", med );
}
```

La cabecera de la función `media` hubiese podido escribirse también como:

```
float media( float *vec, int n )
```

Es importante notar que la definición del parámetro `vec` en la función `media` es una declaración de un *puntero*. Es decir, *no existe diferencia alguna entre* `float *vec` *y* `float vec[]`. El motivo de usar esta nueva notación es dar mayor claridad al programa. Por lo tanto, cuando un parámetro

sea un *puntero a una tabla* usaremos la notación `tipo nombrePuntero[]`, mientras que cuando tengamos un puntero a cualquier otro tipo de datos (tanto tipos elementales como estructurados) usaremos la notación `tipo *nombrePuntero`.

Finalmente, la cabecera de la función `media` también pudiera haberse escrito como:

```
float media( float vec[DIM], int n )
```

En este caso de nuevo el parámetro `vec` es un puntero a una tabla, pero ahora además indicamos el tamaño de la tabla, lo que clarifica aún más el programa. Notar que esto sólo es posible hacerlo si las dimensiónes de la tabla son constantes. Es decir, una expresión como

```
float media( float vec[n], int n )
```

sería incorrecta.

Vemos ahora un ejemplo con una matriz. Deseamos hacer una función que multiplique una matriz por un vector. La solución propuesta es la siguiente:

```
#include <stdio.h>
#define MAXFIL 3
#define MAXCOL MAXFIL

void matXvec( int nfil, int ncol,
              float A[], float x[], float y[] )
{  /*Calcula y = A*x */
   int i, j;

   for (i= 0; i< nfil; i++)
   {
      y[i] = 0.0;
      for (i= 0; i< ncol; i++)
         y[i] = y[i] + A[i*MAXCOL+j] * x[j];
   }
}
void main()
{
   int nfil, ncol;
   float v1[MAXCOL], v2[MAXFIL], M[MAXFIL][MAXCOL];

   ...   /* Leer los nfil, ncol, A, x */
   matXvec( nfil, ncol, M, v1, v2 );
   ...   /* Mostrar y */
}
```

Nótese que los parámetros formales `A`, `x` e `y` son todos punteros. Se accede a los elementos de la matriz a través de un puntero que señala al primer elemento (`[0][0]`). En la sección 9.4 vimos la forma de realizar un acceso de este tipo. Nótese también que en la fórmula que da el número de elementos entre el primero y el elemento `[i][j]`, se debe usar `MAXCOL` y no `ncol`. Es decir,

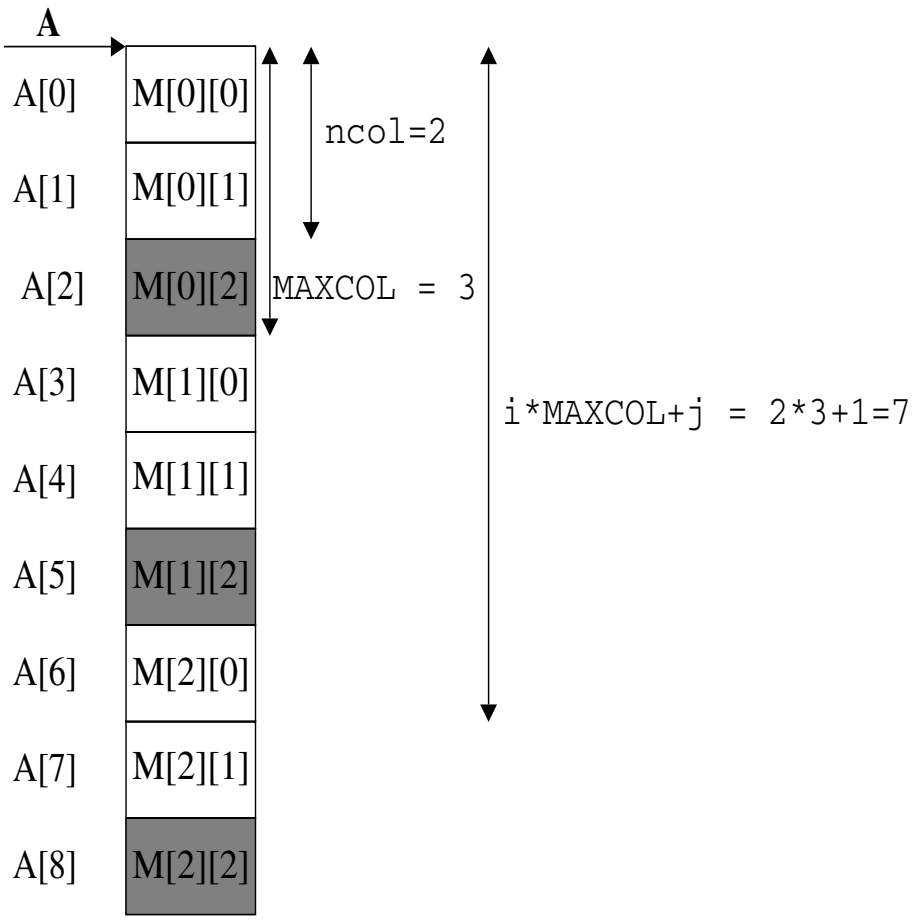


Figura 10.1: Acceso a una matriz mediante un puntero

debemos contar *todos los elementos que hay en memoria* entre el primero y el elemento `[i][j]`, incluyendo los que no son usados por el algoritmo. La figura 10.1 muestra este ejemplo gráficamente suponiendo que `ncol` vale 2, `MAXCOL` vale 3, y que queremos acceder el elemento `[2][1]` de la matriz.

10.5.4 Parámetros en la función `main`

Hasta el momento hemos empleado la función `main` sin parámetros. Sin embargo, es posible pasar parámetros a la función principal del programa, desde la línea de órdenes del sistema operativo. Los parámetros de la función `main` son dos; se conocen tradicionalmente como `argc` y `argv`, aunque pueden tomar cualquier nombre.

El parámetro `argc` es un valor entero que contiene el número de parámetros dados al programa al ser ejecutado desde la línea de órdenes del sistema operativo. El nombre del programa se considera como el primer parámetro, por lo que el valor mínimo de `argc` es 1.

El parámetro `argv` es un vector de punteros a cadenas de caracteres. Estas cadenas toman el valor de cada uno de los parámetros dados al programa al ejecutarlo. Cada parámetro de la línea de órdenes debe estar separado por un espacio o un tabulador.

Así pues, el formato de la función `main` con parámetros es el siguiente:

```
void main( int argc, char *argv[] )
{
   /* Cuerpo de la función.  */
}
```

El siguiente programa de ejemplo permite introducir el nombre del usuario del programa al ejecutarlo y mostrar un mensaje que lo incluya.

```
#include <stdio.h>
void main( int argc, char *argv[] )
{
   if (argc > 2)
   {
      printf( "Demasiados parámetros\n" );
   }
   else if (argc < 2)
   {
      printf( "Faltan parámetros\n" );
   }
   else
   {
      printf( "Yo te saludo %s\n", argv[1] );
   }
}
```

Así pues, si hemos llamado al programa ejecutable `saludo` y se escribe en la línea de órdenes del sistema operativo `saludo Pepe`, la salida del programa será:

```
Yo te saludo Pepe
```

Si alguno de los parámetros que se pasan al programa contiene espacios en blanco o tabuladores, deben usarse comillas en la ejecución del programa.

Línea de órdenes del sistema operativo: `saludo "Pepe Pérez"`
Salida del programa: `Yo te saludo Pepe Pérez`

10.6 Recursividad

Se llama *recursividad* a un proceso en el que una función se llama a sí misma repetidamente hasta que se satisface una cierta condición. Este proceso se emplea para cálculos repetitivos en los que el resultado de cada iteración se determina a partir del resultado de alguna iteración anterior.

Frecuentemente un mismo problema puede expresarse tanto de forma recursiva como de forma iterativa. En estos casos, debido a que la ejecución recursiva requiere de numerosas llamadas a funciones, es preferible utilizar una solución no recursiva. Sin embargo, en otros casos, la escritura de una solución no recursiva puede resultar extraordinariamente compleja. Es entonces cuando es apropiada una solución recursiva.

A continuación se presentan dos funciones recursivas para el cálculo del factorial y el cálculo de valores de la serie de Fibonacci:

```
int fact( int x )
{
   if (x <= 1)
      return (1);
   return (x * fact(x-1));
}

int fibo( int n )
{
   if ((n==0) || (n==1))
      return (1);
   return (fibo(n-1)+fibo(n-2));
}
```

Otro ejemplo de solución recursiva es este programa para invertir los elementos de un vector usando la función `swap`:

```
#define N 10
void invert( int v[], int i, int j )
{
   swap( &v[i], &v[j] );
   i++;
   j--;
   if (i < j)
      invert( v, i, j );
}

void main()
{
   int i, vector[N];

   for(i= 0; i< N; i++)
      scanf( "%d", &vector[i] );
   invert( v, 0, N-1 );
   for(i= 0; i< N; i++)
      printf( "%d\n", vector[i] );
}
```

10.7 Ejercicios

1. El cálculo de e^x puede aproximarse mediante el cálculo de $\sum_{i=0}^{n} \frac{x^i}{i!}$ para n suficientemente grande. Escribir una función `pot` que permita calcular la potencia i-ésima de un número x, donde i y x son parámetros enteros de la función. Usando esta función y la función `fact` del

principio del capítulo, escribir un programa que calcule e^x de forma aproximada para un valor de n dado.

2. Escribir un programa para calcular $sen(x) = \sum_{i=0}^{\infty} \frac{x^{2i+1}}{(2i+1)!} (-1)^i$. Utilizar las funciones `pot` y `frac` del problema anterior.

3. Escribir un programa para calcular $cos(x) = \sum_{i=0}^{\infty} \frac{x^{2i}}{(2i)!} (-1)^i$. Utilizar las funciones `pot` y `frac`.

4. Se dispone de las dos funciones siguientes:

```
void f1( int x, int *y, int a, int b )
{
   x = x + 1;
   *y = *y + 1;
   x = x + a;
   *y = *y + b;
   printf( "%d %d\n", x, *y );
}
void f2( int a, int *b )
{
   a = a + 1;
   *b = *b + 1;
   a = a + a;
   *b = *b + *b;
   printf( "%d %d\n", a, *b );
}
```

   Y del programa principal:

```
#include <stdio.h>
void main()
{
   int a = 0, b = 0;

   llamada
   printf( "%d %d\n", a, b );
}
```

   Indicar el resultado de ejecutar este programa en caso de que `llamada` se substituya por:

   - `f1( a, &b, a, b );` o bien por
   - `f2( a, &b );`

5. A continuación se muestra el esqueleto de un programa en C:

```
int f1( char a, char b )
{
   a = 'P';
   b = 'Q';
   return ((a<b)?(int)a:(int)b);
}
int f2( char *c1, char *c2 )
{
   *c1 = 'R';
   *c2 = 'S';
   return ((*c1==*c2)?(int)*c1:(int)*c2);
}
void main()
{
   char a, b;
   int i, j;
   .  .  .
   a = 'X';
   b = 'Y';
   i = f1( a, b );
   printf( "a=%c,b=%c\n", a, b );
   .  .  .
   j = f2( &a, &b );
   printf( "a=%c,b=%c\n", a, b );
}
```

(a) ¿Qué valores se asignan a `i` y `j` en `main`?

(b) ¿Qué valores escribe la primera llamada a `printf`?

(c) ¿Qué valores escribe la segunda llamada a `printf`?

6. ¿Qué valor calcula el siguiente programa?

```
void func( int p[] )
{
   int i, sum = 0;

   for(i= 3; i< 7; ++i)
      sum = sum + p[i];
   printf( "suma = %d", sum );
}
void main()
{
   int v[10] = {1,2,3,4,5,6,7,8,9,10};
   func( &v[2] );
}
```

7. Determinar si un número no negativo es *perfecto* o tiene algún *amigo*. Dos números son amigos cuando la suma de los divisores de uno de ellos es igual al otro número. Por ejemplo: 220 y 284 son amigos. Por otra parte, un número es perfecto cuando la suma de sus divisores es él mismo. Por ejemplo $6 = 3 + 2 + 1$ es perfecto.

8. Dado el siguiente tipo de datos:

```
#include <stdio.h>
typedef struct
{
   char a[10];
   char b[10];
   char c[10];
} Tcolores;
```

   describir la salida generada por cada uno de los siguientes programas:

(a)
```
void func( Tcolores X )
{
   X.a = "cian";
   X.b = "magenta";
   X.c = "amarillo";
   printf( "%s%s%s\n", X.a, X.b ,X.c );
   return();
}
void main()
{
   Tcolores col = { "rojo", "verde", "azul" };
   printf( "%s%s%s\n", col.a, col.b, col.c );
   func( col );
   printf( "%s%s%s\n", col.a, col.b, col.c );
}
```

(b)
```
void func(Tcolores *X)
{
   X->a = "cian";
   X->b = "magenta";
   X->c = "amarillo";
   printf( "%s%s%s\n", X->a, X->b, X->c );
   return();
}
void main()
{
   Tcolores col = { "rojo", "verde", "azul" };
   printf( "%s%s%s\n", col.a, col.b, col.c );
   func( & col );
   printf( "%s%s%s\n", col.a, col.b, col.c );
}
```

Capítulo 11

Ficheros

Es preciso algún mecanismo que permita almacenar de forma permamente ciertos datos de los programas. Por ejemplo, un programa que gestione la contabilidad de una empresa necesita una serie de informaciones iniciales (balance hasta ese momento, lista de compras, lista de ventas, etc). De igual forma, genera una serie de informaciones que deben ser almacenadas cuando el programa finaliza.

Desde el punto de vista del *hardware*, hay diferentes dispositivos para almacenar información de forma permanente: discos duros, unidades de cinta, CDs, disquetes, etc. Para un programador, el dispositivo físico que se use carece de importancia. Los programas deben funcionar tanto si la información está en un disco duro como en un CD como en una cinta. Por lo tanto, es preciso un conjunto de funciones (una librería) que permita realizar almacenamiento permamente de información, pero omitiendo los detalles específicos de cada dispositivo *hardware*. Esta librería de funciones la proporciona el sistema operativo.

Para ocultarle al programador los detalles específicos de cada dispositivo *hardware*, se usa el concepto de *fichero*. Un fichero es un objeto abstracto sobre el cual se puede leer y escribir información.

Existen dos tipos fundamentales de ficheros: *ficheros de texto* y *ficheros binarios*. En los ficheros de texto la información se almacena usando caracteres (códigos ASCII). Por ejemplo, una variable de tipo `int` se almacena en la memoria como una secuencia de bits que debe ser interpretada como un código *complemento a dos*. Sin embargo, cuando escribimos dicha variable en un fichero de texto, lo que almacenamos es un conjunto de caracteres que representan el valor de la variable en base 10. Una variable tipo de `int` que en memoria se almacenase como $1\overbrace{0\cdots0}^{31}$ en un fichero de texto se escribiría como -2147483648. En los ficheros binarios la información se almacena de igual forma que en la memoria, mediante la misma secuencia de unos y ceros.

Usar ficheros de texto tiene la ventaja de que la información almacenada puede ser visualizada y comprendida por un ser humano. Pero tiene el inconveniente de ocupar aproximadamente tres veces más espacio que los ficheros binarios. Por ejemplo, la variable de tipo `int` mostrada anteriormente, ocupa 4 bytes en un fichero binario (lo mismo que ocupa en memoria), y ocupa 11 bytes (signo y 10 cifras) en un fichero de texto.

Existen diferentes funciones para trabajar con ficheros de texto y con ficheros binarios. En este libro nos ocuparemos únicamente de los ficheros de texto.

Un fichero de texto almacena la información como una secuencia de códigos ASCII. La figura 11.1 muestra un ejemplo de almacenamiento de un fichero de texto. En esta figura se muestra el aspecto que

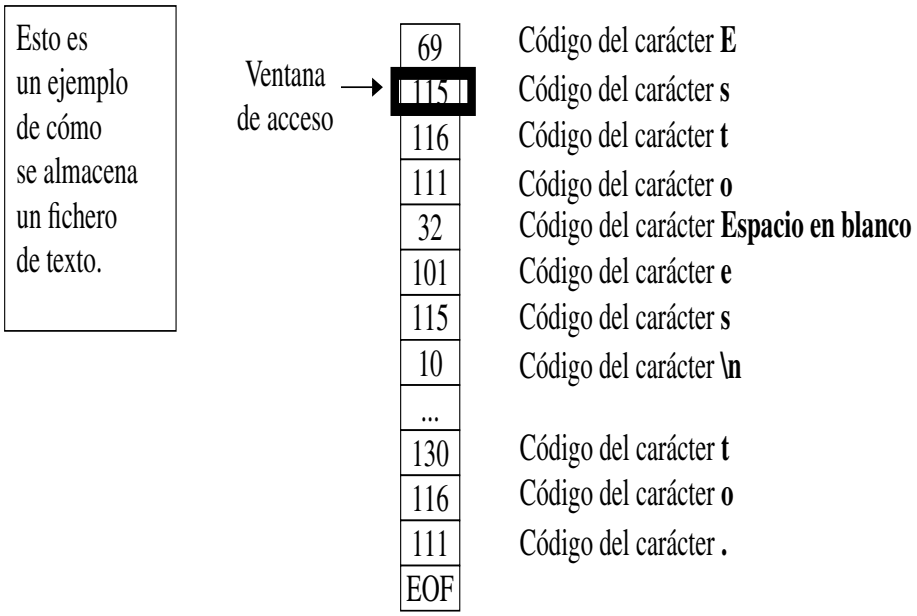


Figura 11.1: Almacenamiento de un fichero de texto

tendría un fichero de texto al mostrarse en pantalla, así como la secuencia de bytes que se almacenarían en disco. Todos los ficheros de texto finalizan com un carácter especial que indica el final del fichero. Este carácter lo representamos mediante la constante `EOF` (End Of File) que se halla definida en el fichero `stdio.h`.

Cuando queramos leer o escribir información en un fichero de texto deberemos hacerlo de forma secuencial. Por ejemplo, si queremos leer el fichero de la figura 11.1, deberemos leer en primer lugar el primer carácter y así sucesivamente. Esto se debe a que existe una *ventana* asociada al fichero que sólo puede avanzar secuencialmente, nunca a saltos.

El sistema operativo usa variables del tipo `FILE` para manejar los dispositivos *hardware* asociados a ficheros. La definición del tipo `FILE` se encuentra en el fichero `stdio.h`. Todos los programas que manejen ficheros deberán incluir `stdio.h`. Una variable del tipo `FILE` es una estructura cuyo contenido sólo puede ser entendido y manejado por funciones del sistema operativo. Dicha estructura contiene información, por ejemplo, de la pista y el sector del disco donde comienza el fichero, etc. Dado que las variables de tipo `FILE` pertenecen al sistema operativo, *nunca* tendremos una variable de este tipo en nuestros programas. Lo único que necesitamos son punteros a dichas variables. Esto es, si queremos manejar un fichero dentro de un programa, deberemos tener una declaración como la siguiente:

```
FILE *fp;
```

El puntero `fp` debe ser inicializado mediante la función `fopen`, de forma que apunte a una variable del sistema operativo que contenga los datos del fichero concreto que usemos. Las funciones de lectura y escritura en el fichero únicamente necesitan conocer dicho puntero para saber la variable que deben usar.

Para utilizar un fichero se debe realizar una secuencia fija de acciones:

1. Abrir el fichero. Esto significa decirle al sistema operativo que inicialice una variable de tipo `FILE`, de forma que a partir de ese momento, las acciones de lectura/escritura que utilicen dicha variable se realicen realmente en el dispositivo *hardware* correspondiente. Esto se hace llamando a una función `fopen`.

2. Leer o Escribir en el fichero. Para ello usaremos las funciones `fscanf` y `fprintf`. Estas funciones sólo necesitan dos datos: la variable de tipo `FILE` asociada al fichero y la información que queremos leer o escribir en dicho fichero.

3. Cerrar el fichero. Esto significa indicar al sistema operativo que ya no necesitamos la variable tipo `FILE` asociada al fichero. Para ello se usa la función `fclose`.

11.1 Abrir y cerrar ficheros

La cabecera de la función `fopen` es la siguiente:

```
FILE * fopen( char nombreFichero[], char modoAcceso[] )
```

donde `nombreFichero` es una cadena de caracteres con el nombre del fichero físico que se quiere usar, y `modoAcceso` es una cadena de caracteres que indica la acción que realizaremos sobre el fichero.

Existen tres modos básicos para abrir ficheros:

- `"r"`: Abrir un fichero ya existente para lectura.
- `"w"`: Abrir un fichero nuevo para escritura. Si el fichero ya existía, será destruido y creado de nuevo.
- `"a"`: Abrir un fichero ya existente para añadir información; esto es, escribir al final del mismo. Si el fichero no existía se creará uno nuevo.

Además de estos modos existen otros de uso menos frecuente:

- `"r+"`: Abrir un fichero ya existente tanto para lectura como escritura.
- `"w+"`: Abrir un fichero nuevo tanto para lectura como escritura. Si el fichero ya existía será destruido y creado de nuevo.
- `"a+"`: Abrir un fichero ya existente para leer y añadir. Si el fichero no existía se creará uno nuevo.

La función `fopen` retorna la constante `NULL` si no ha podido abrir el fichero. Esta condición de error debe ser comprobada *siempre* que se use la función `fopen`. La constante `NULL` está definida en `stdio.h`.

Cuando ya hemos acabado de utilizar el fichero, debemos indicárselo al sistema operativo mediante la función `fclose`, que libera la variable de tipo `FILE` asociada al fichero. La cabecera de la función `fclose` es la siguiente:

```
int fclose( FILE *fp )
```

Si no se produce ningún error al cerrar el fichero `fclose` retorna 0. En caso contrario retorna la constante `EOF` (recordemos que esta constante está definida en `stdio.h`). Veamos algunos ejemplos del uso de `fopen` y `fclose`.

En el siguiente ejemplo abrimos un fichero llamado `miFichero.txt` para leer la información contenida en él. En caso de error al abrir el fichero mostramos un mensaje en pantalla y finalizamos la ejecución del programa mediante la función `exit`. Finalmente cerramos el fichero.

```
#include <stdio.h>
void main( )
{
   FILE *fp;

   fp = fopen( "miFichero.txt", "r" );
   if (fp == NULL)
   {
      printf( "Error abriendo miFichero.txt\n" );
      exit(0);
   }
   .  .  .
   /* Aquí podemos leer datos del fichero */
   .  .  .
   fclose( fp );
}
```

A continuación se muestra otro ejemplo para el caso en que abrimos un fichero para escritura. El nombre del fichero es introducido por el usuario a través del teclado. Si el fichero ya existe, será destruido y creado de nuevo. Esta acción la realiza de forma automática la función `fopen`. Finalmente cerramos el fichero.

```
#include <stdio.h>
#define N 256
void main( )
{
   FILE *fp;
   char nombreFichero[N];

   printf( " Nombre del fichero (< %d caracteres):  ", N );
   scanf( "%s%*c", nombreFichero );
   fp = fopen( nombreFichero, "w" );
   if (fp == NULL)
   {
      printf( "Error abriendo %s\n", nombreFichero );
      exit(0);
   }
   .  .  .
```

```
    /* Aquí podemos escribir datos en el fichero */
    .  .  .
    fclose( fp );
}
```

11.2 Leer y escribir en ficheros

Una vez abierto un fichero, podemos leer y escribir en él mediante las funciones `fscanf` y `fprint`. Las cabeceras de estas funciones son:

```
int fscanf( FILE *fp, char formato[], <lista variables> )
int fprintf( FILE *fp, char formato[], <lista variables> )
```

La función `fscanf` permite leer del fichero apuntado por `fp`, mientras que la función `fprintf` permite escribir en el fichero apuntado por `fp`. El uso de estas funciones es análogo al de `scanf` y `printf`, que permiten leer variables desde el teclado y escribir variables en pantalla, respectivamente. Por tanto, `formato` es una cadena de caracteres que describe el formato de las variables a leer/escribir. Por su parte, `<lista variables>` contiene las direcciones de memoria de todas las variables en el caso de `fscanf` y las variables propiamente dichas en el caso de `fprintf`. Los operadores de formato se hallan descritos en el apéndice B.

La función `fprintf` retorna el número de bytes (caracteres) escritos en el fichero, o un número negativo en caso de que ocurra algún error en la escritura. La función `fscanf` retorna el número de variables correctamente leídas, o la constante `EOF` en caso de error.

Veamos algunos ejemplo de uso de `fscanf` y `fprintf`.

En el siguiente ejemplo leemos un vector de enteros de un fichero de entrada. El fichero contiene en la primera línea el número de elementos del vector. El resto de líneas del fichero contienen un elemento del vector en cada línea. Finalmente, el programa escribe el vector en un fichero de salida usando el mismo formato que en el fichero de entrada.

```
#include <stdio.h>
#define N 256
#define MAXELE 100
void main( )
{
   FILE *fp;
   char nombreFichero[N];
   int lon = 0;
   int vec[MAXELE];

   printf( "Fichero de entrada(< %d caracteres):  ", N );
   scanf( "%s%*c", nombreFichero );
   fp = fopen( nombreFichero, "r" );
   if (fp == NULL)
   {
      printf( "Error abriendo %s\n", nombreFichero );
```

```
      exit(0);
   }
   fscanf( fp, "%d", &lon );
   if (lon < MAXELE)
   {
      for (i= 0; i< lon; i= i+1)
         fscanf( fp, "%d", &vec[i] );
   }
   else
      printf( "El vector tiene demasiados elementos\n" );
   fclose( fp );
   .  .  .
   /* Aquí podemos modificar vec */
   .  .  .
   printf( "Fichero de salida(< %d caracteres):  ", N );
   scanf( "%s%*c", nombreFichero );
   fp = fopen( nombreFichero, "w" );
   if (fp == NULL)
   {
      printf( "Error abriendo %s\n", nombreFichero );
      exit(0);
   }
   fprintf( fp, "%d\n", lon );
   for (i= 0; i< lon; i= i+1)
      fprintf( fp, "%d\n", vec[i] );
   fclose( fp );
}
```

En el ejemplo anterior, si el nombre del fichero de salida es el mismo que el nombre del fichero de entrada, los datos iniciales se perderán, ya que al abrir el fichero en modo `"w"`, el fichero que ya existía es destruido y creado de nuevo. En este ejemplo, sin embargo, leemos de un fichero y el resultado del programa es añadido al final del mismo fichero.

```
#include <stdio.h>
#define N 256
#define MAXELE 100
void main( )
{
   FILE *fp;
   char nombreFichero[N];
   int lon;
   int vec[MAXELE];

   printf( "Nombre del fichero(< %d caracteres):  ", N );
   scanf( "%s%*c", nombreFichero );
   fp = fopen( nombreFichero, "r" );
   if (fp == NULL)
```

```
    {
       printf( "Error abriendo %s\n", nombreFichero );
       exit(0);
    }
    fscanf( fp, "%d", &lon );
    if (lon < MAXELE)
    {
       for (i= 0; i< lon; i= i+1)
          fscanf( fp, "%d", &vec[i] );
    }
    else
       printf( "El vector tiene demasiados elementos\n" );
    fclose( fp );
    .  .  .
    /* Aquí trabajamos con vec */
    .  .  .
    fp = fopen( nombreFichero, "a" );
    if (fp == NULL)
    {
       printf( "Error abriendo %s\n", nombreFichero );
       exit(0);
    }
    fprintf( fp, "%d\n", lon );
    for (i= 0; i< lon; i= i+1)
       fprintf( fp, "%d\n", vec[i] );
    fclose( fp );
}
```

11.3 Otras funciones para el manejo de ficheros

11.3.1 `feof`

En la mayoría de ocasiones debemos leer un fichero sin saber su tamaño a priori y, por lo tanto, sin saber la cantidad de datos que debemos leer. En esta situación se hace necesaria una función que nos indique cuándo se alcanza el final de fichero. Esta función es `feof`, cuya cabecera es la siguiente:

```
int feof( FILE *fp )
```

La función `feof` retorna un número diferente de 0 (*cierto*) cuando el carácter especial `EOF` ha sido alcanzado en una lectura *previa* del fichero señalado por `fp`. En caso contrario, retorna 0 (*falso*). Es muy importante notar que la función `feof` sólo indica ”fin de fichero” si previamente hemos realizado una lectura mediante `fscanf` que no ha podido leer nada (que ha alcanzado el carácter `EOF`). Veamos algunos ejemplos del uso de `feof`.

El programa del siguiente ejemplo lee de un fichero los elementos de un vector de enteros. El fichero contiene un elemento del vector en cada línea. Nótese que en el bucle `while` se controlan dos condiciones: alcanzar el fin del fichero y llenar completamente el vector.

```
#include <stdio.h>
#define N 256
#define MAXELE 100
void main( )
{
   FILE *fp;
   char nombreFichero[N];
   int lon;
   int vec[MAXELE];

   printf( " Nombre del fichero(< %d caracteres):  ", N );
   scanf( "%s%*c", nombreFichero );
   fp = fopen( nombreFichero, "r" );
   if (fp == NULL)
   {
      printf( "Error abriendo %s\n", nombreFichero );
      exit(0);
   }
   lon = 0;
   while (!feof(fp) && (lon < MAXELE))
   {
      kk = fscanf( fp, "%d", &vec[lon] );
      if (kk == 1)
         lon++;
      if (!feof(fp) && (lon == MAXELE))
         printf( "Todo el contenido del fichero no
                  cabe en el vector\n" );
   }
   fclose( fp );
   .  .  .
}
```

Supongamos que el fichero contiene sólo tres líneas como las siguientes:

```
123
254
-35
```

Al ejecutar el programa, el bucle `while` realizará cuatro iteraciones. En la tercera iteración se leerá del fichero el número `-35` y se almacenará en `vec[2]`. Sin embargo, la función `feof` aún no indicará el final del fichero, es decir, retornará 0. En la cuarta iteración, la función `fscanf` detectará el carácter `EOF` y por lo tanto no podrá leer ningún valor válido. Así pues, en `vec[3]` no almacenamos nada (se queda como estaba, con un valor aleatorio). Podremos saber que esta situación ha ocurrido consultando el valor retornado por la función `fscanf`. En este ejemplo, como sólo leemos una variable, `fscanf` debe retornar 1 si ha podido realizar una lectura correcta. Nótese que el programa sólo incrementa el valor de `lon` si la lectura ha sido correcta. Después de la cuarta iteración, `feof` retornará un valor diferente de 0 (*cierto*).

11.3.2 ferror

La cabecera de esta función es la siguiente:

```
int ferror( FILE *fp )
```

La función ferror retorna un valor diferente de 0 si ha ocurrido algún error en una lectura/escritura previa en el fichero señalado por fp. En caso contrario retorna 0.

11.3.3 fflush

La cabecera de esta función es la siguiente:

```
int fflush( FILE *fp )
```

Cuando escribimos en un fichero, en realidad la escritura no se produce en el mismo momento de ejecutar la función fprintf. Sin embargo, esta función deja la información a escribir en un *buffer* temporal del sistema operativo. Más tarde, cuando el sistema operativo lo decida (esté libre de otras tareas, por ejemplo), se vuelca el contenido de dicho *buffer* sobre el fichero físico. De esta forma en un computador con varios usuarios se puede organizar de forma más eficiente el acceso a las unidades de almacenamiento de información. La función fflush puede utilizarse para forzar en el instante deseado el volcado del *buffer* sobre el fichero. Si no se produce ningún error, la función fflush retorna 0, en caso contrario retorna EOF.

Un ejemplo típico del uso de fflush es la depuración de programas. Supongamos que tenemos un error en un programa y para encontrarlo ponemos diversos fprintf, que muestran valores de algunas variables. Si no ponemos después de cada fprintf una llamada a fflush, no veremos el valor que queremos en el momento en que realmente se produce, lo que nos puede llevar a conclusiones erróneas sobre el comportamiento del programa.

11.4 Ficheros estándar: stdin, stdout, stderr

En C existen tres constantes del tipo FILE *, definidas en stdio.h, llamadas stdin, stdout y stderr. El puntero stdin apuntan a un fichero abierto sólo para lectura. Los punteros stdout y stderr apuntan a ficheros abiertos sólo para escritura.

Estos punteros están inicializados por el sistema operativo de forma que una lectura de stdin sea en realidad una lectura del teclado. Es decir, que una llamada como

```
fscanf( stdin, "%d", &i )
```

es equivalente a

```
scanf( "%d", &i ) .
```

De igual forma, los ficheros asignados a stdout y stderr están inicialmente redirigidos a la pantalla, de forma que

```
fprintf( stdout, "Hola\n" ) o fprintf( stderr, "Hola\n" )
```

tienen el mismo efecto que

```
printf( "Hola\n" ) .
```

Las constantes stdin, stdout y stderr pueden ser usadas para inicializar varibles del tipo FILE * de forma que la entrada/salida sea a través de teclado/pantalla.

El siguiente ejemplo muestra un programa para multiplicar una matriz por un vector. Los datos de entrada se leen de `stdin`, es decir, del teclado. Por otra parte, los datos de salida se escriben en `stdout` (pantalla), y los mensajes de error en `stderr` (también la pantalla). Nótese que `stdin`, `stdout` y `stderr` no están declaradas en el programa, puesto que ya lo están en `stdio.h`. Cuando el programa funciona usando teclado/pantalla, muestra una serie de mensajes en pantalla explicando al usuario los datos que debe introducir. Sin embargo, cuando se usan ficheros, estos mensajes no tienen sentido, por lo que no son mostrados.

```
#include <stdio.h>
#define MAXFILAS 10
#define MAXCOLUMNAS MAXFILAS
void main( )
{
   int i, j, k, Nfilas, Ncolumnas;
   double x[MAXCOLUMNAS], y[MAXFILAS];
   double A[MAXFILAS][MAXCOLUMNAS];
   char car;
   FILE *fi = stdin;
   FILE *fo = stdout;
   FILE *fe = stderr;

   printf( "Entrada/Salida por ficheros?  (s/n)" );
   scanf( "%c", &car );
   if (car == 's'|| car == 'S')
   {
      fi = fopen( "Entrada.txt", "r" );
      if (fi == NULL)
      {
         printf( "Error abriendo Entrada.txt\n" );
         exit(0);
      }
      fo = fopen( "Salida.txt", "w" );
      if (fo == NULL)
      {
         printf( "Error abriendo Salida.txt\n" );
         exit(0);
      }
      fe = fopen( "Errores.txt", "w" );
      if (fe == NULL)
      {
         printf( "Error abriendo Errores.txt\n" );
         exit(0);
      }
   }
   if (fo == stdout)
      fprintf( fo, " N. filas = " );
```

```
fscanf( fi, "%d", &Nfilas );
if (Nfilas > MAXFILAS)
{
   fprintf( fe, "Demasiadas filas\n" );
   exit (0);
}

if (fo == stdout)
   fprintf( fo, " N. columnas = " );
fscanf( fi, "%d", &Ncolumnas );
if (Ncolumnas > MAXCOLUMNAS)
{
   fprintf( fe, "Demasiadas columnas\n" );
   exit (0);
}

for (i= 0; i< Nfilas; i++)
   for (j= 0; j< Ncolumnas; j++)
   {
      if (fo == stdout)
         fprintf( fo, "A[%d][%d] = ", i, j );
      k = fscanf( fi, "%lf", &A[i][j] );
      if (k != 1)
      {
         fprintf( fe, "Error leyendo la matriz\n" );
         exit (0);
      }
   }

for (i= 0; i< Nfilas; i++)
{
   if (fo == stdout)
      fprintf( fo, "x[%d] = ", i );
   k = fscanf( fi, "%lf", &x[i] );
   if (k != 1)
   {
      fprintf( fe, "Error leyendo el vector\n" );
      exit (0);
   }
}

for (i= 0; i< Nfilas; i++)
{
   y[i] = 0.0;
   for (j= 0; j< Ncolumnas; j++)
      y[i] = y[i] + A[i][j] * x[j];
```

```
    }

    for (i= 0; i< Nfilas; i++)
       fprintf( fo, "y[%d] = %lf\n", i, y[i] );

    if (fi != stdin)
       fclose( fi );
    if (fo != stdout)
       fclose( fo );
    if (fe != stderr)
       fclose( fe );
}
```

11.5 Ejercicios

1. Se dispone de dos ficheros con números enteros ordenados de menor a mayor. Escribir los siguientes programas de forma que el fichero resultante contenga los números ordenados de mayor a menor.

   - Un programa que construya un fichero con todos los números que están en ambos ficheros simultáneamente (AND de ficheros).
   - Un programa que construya un fichero con todos los números que están en cualquiera de los dos ficheros (OR de ficheros). En el fichero resultante no debe haber números repetidos.
   - Un programa que construya un fichero con los números que están en cualquiera de los dos ficheros, pero no en los dos simultáneamente (XOR de ficheros).

2. Se dispone de un fichero que contiene texto y se pretende realizar una compactación del mismo. Para ello se substituyen las secuencias de cuatro o más caracteres blancos por la secuencia `#n#` , donde `n` indica el número de caracteres blancos que se han substituido. Para evitar confución, el carácter `#` se sustituye por `##`. Diseñar una función que lea de un fichero un texto no compactado y lo compacte según los criterios expuestos. Diseñar también otra función que lea un fichero de texto resultado de una compactación previa y lo descompacte.

3. Se dispone de un fichero que contiene un número no determinado de etiquetas. Cada etiqueta es un conjunto de datos sobre un determinado individuo (nombre, dirección, teléfono, etc). La etiqueta está formada por 3 líneas consecutivas de texto. Cada línea de texto tiene 15 caracteres como máximo. Las etiquetas están separadas por una línea que contiene únicamente el carácter `*`. Se desea diseñar un programa que permita construir un nuevo fichero que contenga las etiquetas del fichero original pero organizadas en columnas de 3 etiquetas (que empezarán respectivamente en las columnas 0, 20 y 40). Supondremos que las líneas de un fichero pueden tener un máximo de 80 caracteres. Las filas de etiquetas deben estar separadas por una línea en blanco. Por ejemplo:

   Fichero de Entrada

   ```
   Juan Pérez
   c/ Aragón
   Tlf.  932 491 134
   ```

```
*
Pedro López
Avd.  Europa
Tlf.  931 113 456
*
Juan García
c/ Gracia
Lérida
*
Andrés Villa
Tlf.  931 113 457
Badalona
*
Pedro Cubero
Tlf.  971 456 789
Mallorca
*
```

Fichero de Salida

```
Juan Pérez          Pedro López         Juan García
c/ Aragón           Avd.  Europa        c/ Gracia
Tlf.  932 491 134   Tlf.  931 113 456   Lérida

Andrés Villa        Pedro Cubero
Tlf.  931 113 457   Tlf.  971 456 789
Badalona            Mallorca
```

4. Se dispone de un fichero compuesto únicamente por letras mayúsculas, espacios en blanco, comas y puntos. El contenido de este fichero tiene las siguientes características:

   - Entre palabra y palabra puede haber cualquier número de espacios en blanco.
   - Entre una palabra y un signo de puntuación puede haber cualquier número de espacios en blanco.
   - Entre un signo de puntuación y una palabra puede haber cualquier número de espacios en blanco.
   - El primer y último carácter del texto de entrada es una letra.

   Debemos realizar un algoritmo que escriba en un fichero de caracteres el contenido del fichero de entrada formateado de tal manera que en el texto resultante se cumplan los siguientes requisitos:

   - Todas las palabras estarán escritas con letras minúsculas excepto la primera letra después de un punto y la primera letra del texto.
   - Entre palabra y palabra sólo puede haber un blanco.
   - Entre la última letra de una palabra y un signo de puntuación no debe haber ningún blanco.

- Entre un signo de puntuación y la primera letra de una palabra debe haber un espacio en blanco.
- El último carácter debe ser un punto.

Apéndice A

El preprocesador

El preprocesador es una herramienta muy útil para el programador. Las *directivas* del preprocesador son en realidad simples comandos de edición, es decir, comandos que modifican el fichero con código fuente del programa, de igual forma que lo haríamos mediante un editor de textos. El fichero modificado por el preprocesador sigue siendo un fichero de texto.

Las directivas del preprocesador se distinguen de las líneas de código C porque empiezan con el símbolo `#` en la primera columna. Es importante hacer notar que es obligatorio que el símbolo `#` esté en la primera columna, ya que en caso contrario se generá un error de compilación.

En este apéndice veremos las directivas más importantes del preprocesador.

A.1 Directiva `include`

La directiva `include` permite incluir en el fichero de código fuente otros ficheros de texto. Esta directiva puede usarse dos formas distintas:

```
#include <fichero.h>
#include "miFichero.h"
```

Cuando el fichero incluido pertenece al sistema operativo se usan los símbolos `< >` para delimitar el nombre del fichero. Si el fichero no forma parte del sistema operativo se usan los símbolos `" "`. En cualquier caso, el efecto de un `include` es el mismo: se sustituye la línea donde aparece la directiva por el contenido del fichero indicado.

A.2 Directivas `define` y `undef`

Como su nombre indica, la directiva `define` permite definir símbolos. Por su parte, la directiva `undef` permite eliminar símbolos previamente definidos. El uso de estas directivas es el siguente:

```
#define nombreSímbolo valorSímbolo
#undef nombreSímbolo
```

donde `nombreSímbolo` es el nombre del símbolo que definimos/eliminamos y `valorSímbolo` es el valor que damos a dicho símbolo. Dar valor al símbolo es optativo.

El principal uso de la directiva `define` es substituir un texto por otro texto. Por ejemplo:

```
#define N 100
```

significa que el preprocesador sustituirá el símbolo `N` por el texto `100` dentro del programa. A continuación se muestra un fragmento de código antes y después de ser tratado por el preprocesador:

Antes del preprocesador

```
...
for (i= 0; i< N; i++)
   Numeros[i] = i;
...
```

Después del preprocesador

```
...
for (i= 0; i< 100; i++)
   Numeros[i] = i;
...
```

Nótese que la palabra `Numeros` no ha sido substituida por `100umeros`. Sólo se ha substituido el texto `N` allí donde las reglas sintácticas de C indican que dicho texto es el nombre de un símbolo.

Esto se aprovecha para la definición de constantes. Normalmente estas constantes son las dimensiones máximas de tablas del programa. De esta forma, cuando deseemos modificar estas dimensiones, bastará modificar la línea de código que contiene el `define`, sin tener que buscar por el programa todas las apariciones de las constantes.

La directiva `define` tiene otros usos importantes que veremos en las secciones A.3 y A.4.

A.3 Directivas `ifdef` y `ifndef`

En ocasiones es preciso que determinados fragmentos de código fuente sólo se compilen si se cumplen ciertas condiciones. A este hecho se le denomina *compilación condicional*. Las directivas `ifdef` y `ifndef` sirven para realizar dicho tipo de compilación. El uso de estas directivas es el siguiente:

```
#ifdef nombre
   código1
#else
   código2
#endif

#ifndef nombre
   código1
#else
   código2
#endif
```

donde `nombre` es un símbolo definido mediante la directiva `define`. Los textos indicados por `código1` y `código2` representan fragmentos de código fuente en C. En la directiva `ifdef`, si

existe un `define` que defina el símbolo `nombre` el código que finalmente se complia corresponde al fragmento indicado por `código1`. En caso contrario, el código compilado corresponde a `código2`. Por otra parte, en la directiva `ifndef`, si *no* existe un `define` para el símbolo `nombre`, el código compilado es el correspondiente a `código1`. En caso contrario, el código compilado es el correspondiente a `código2`. En ambas directivas el uso de `else` es optativo. Veamos algunos ejemplos.

Supongamos que un programa debe mostrar ciertos valores en pantalla para estar seguros de su funcionamiento. Pero esto sólo es necesario hacerlo mientras el programa está en la fase de desarrollo. Una vez finalizada esta fase, no es necesario que muestre toda esa información. Una solución consistirá en borrar manualmente las líneas de código pertinentes, pero si el programa es grande (miles o millones de líneas de código) podemos cometer errores fácilmente al eliminar dichas líneas. En el siguiente código, el símbolo `DEBUG` controla si se compila o no el `printf`. Nótese que la directiva `#define DEBUG` no le asigna valor a la constante `DEBUG`, símplemente la define como símbolo.

```
#define DEBUG
...
for (i= 0; i< Nfilas; i++)
{
   y[i] = 0.0;
   for (j= 0; j< Ncolumnas; j++)
   {
#ifdef DEBUG
      printf( "y[%d]= %lf, x[%d]= %lf, A[%d][%d]= %lf\n",
            i, y[i], j, x[j], i, j, A[i][j] );
#endif
      y[i] = y[i] + A[i][j] * x[j];
   }
}
```

Supongamos ahora que necesitamos mostrar en pantalla los recursos que usa un programa (memoria, tiempo de ejecución, etc). Para ello debemos llamar a una función del sistema operativo. Pero en cada sistema operativo el nombre de dicha función puede ser diferente, o puede que incluso no exista dicha función. El siguiente código muestra una solución para que, en cualquier caso, el programa se pueda compilar sin problemas:

```
...
printf( "Recursos usados por el programa\n" );
#ifdef WINDOWS
   printf( "Funcion no disponible en sistema WINDOWS\n" );
#else
   getrusage( RUSAGE_SELF, &rusage );
   ...
#endif
...
```

A.4 Macros

La directiva `define` también permite definir *macros*. La sintaxis de una macro es la siguiente:

```
#define nombreMacro( param1, param2, ...  )  código
```

donde `código` es un conjunto válido de sentencias en C, y `param1`, etc. son símbolos que aparecen en `código`. Cuando el preprocesador se ejecuta, substituye cada llamada a la macro por el texto escrito en `código`, substituyendo dentro de `código` los símbolos `param1`, etc. por los valores que tengan en la llamada a la macro.

Veamos un ejemplo:

```
#define SWAP( p1, p2, p3 )     p3=p1; p1=p2; p2=p3;
```

En el código anterior hemos definido la macro `SWAP`. Esta macro tiene tres parámetros `p1`, `p2` y `p3`. Donde esta macro sea invocada, el preprocesador susbtituirá la macro por el código indicado. Esta macro sirve para intercambiar los valores de los parámetros `p1` y `p2`, usando el parámetro `p3` como una variable temporal. Veamos a continuación qué haría el preprocesador con dos llamas a la macro `SWAP` en el siguente programa:

Antes del preprocesador

```
double x, y, z;
int a, b, c;
...
SWAP ( x, y, z );
SWAP ( a, b, c );
...
```

Después del preprocesador

```
double x, y, z;
int a, b, c;
...
z=x; x=y; y=z;
c=a; a=b; b=c;
...
```

Una macro siempre se puede substituir por una función. Pero si las sentencias de la macro son muy simples, podemos gastar más tiempo llamando y retornando de la función que ejecutando su código. Además, en un ejemplo como el anterior vemos que la misma macro sirve para valores enteros, reales, caracteres, estructuras, etc. Sin embargo necesitaríamos una función diferente para cada tipo de datos distinto.

Apéndice B

La librería estándar

Como se ha visto, una de las características de C es que su sintaxis se basa en un conjunto muy reducido de palabras reservadas (ver Tab. 3.1). Por esta razón, las operaciones de entrada y salida, el manejo de cadenas de caracteres, las funciones matemáticas, etc. no forman parte propiamente del lenguaje C. Todas estas funcionalidades, y muchas otras más, se hallan implementadas en una librería de funciones y tipos de datos especiales que se conoce como la *librería estándar*.

El programador puede acceder a la librería estándar mediante un conjunto de ficheros de cabeceras (con extensión '`.h`'). Así pues, si un programa utiliza alguna de las funciones de la librería, dicho programa deberá incluir el fichero de cabeceras correspondiente donde se halle definida dicha función. Por ejemplo, si el programa utiliza una función de entrada y salida como `printf`, deberá incluir el fichero `stdio.h` de la siguiente forma: `#include <stdio.h>`.

Este apéndice resume algunas de las funciones disponibles en dicha librería agrupándolas según el fichero de cabeceras donde se hallan definidas. Para cada función, se proporciona su nombre, parámetros y resultado devuelto, así como una breve descripción. Cabe decir que alguno estos datos puede diferir de un sistema a otro, por lo que es recomendable consultar los manuales correspondientes para mayor seguridad.

B.1 Manipulación de cadenas de caracteres

Las siguientes funciones se hallan definidas en el fichero de cabeceras `string.h`:

- `int strcasecmp( char s1[], char s2[] )` Compara las dos cadenas `s1` y `s2`, ignorando mayúsculas y minúsculas. Devuelve un entero menor, igual o mayor que 0, si `s1` es menor, igual o mayor lexicográficamente que `s2`, respectivamente.
- `char *strcat( char dest[], char src[] )` Concatena la cadena `src` al final de la cadena `dest`. La cadena `dest` debe tener suficiente espacio para albergar la cadena resultante.
- `char *strchr( char s[], int c )` Devuelve un puntero a la posición de la primera ocurrencia del carácter `c` en la cadena `s`.
- `int strcmp( char s1[], char s2[] )` Compara las dos cadenas `s1` y `s2`. Devuelve un entero menor, igual o mayor que 0, si `s1` es menor, igual o mayor lexicográficamente

que `s2`, respectivamente.

- `char *strcpy( char dest[], char src[] )` Copia la cadena `scr` en la cadena `dest`. La cadena `dest` debe tener suficiente espacio para albergar la cadena `src`.
- `char *strdup( char s[] )` Devuelve un puntero a una nueva cadena que es un duplicado de la cadena `s`.
- `int strlen( char s[] )` Devuelve la longitud de la cadena `s`, sin contar el carácter `'\0'`.
- `char *strncat( char dest[], char src[], int n )` Similar a `strcat`, a excepción de que sólo se concatenan al final de `dest`, los `n` primeros caracteres de la cadena `src`.
- `int strncmp( char s1[], char s2[], int n )` Similar a `strcmp`, a excepción de que sólo se comparan los `n` primeros caracteres de ambas cadenas.
- `char *strncpy( char dest[], char src[], int n )` Similar a `strcpy`, a excepción de que sólo se copian en `dest` los `n` primeros caracteres de la cadena `src`.
- `int strncasecmp( char s1[], char s2[], int n )` Similar a `strcasecmp`, a excepción de que sólo se comparan los `n` primeros caracteres de ambas cadenas.
- `char *strrchr( char s[], int c )` Devuelve un puntero a la posición de la última ocurrencia del carácter `c` en la cadena `s`.
- `char *strstr( char s1[], char s2[] )` Devuelve un puntero a la posición de la primera ocurrencia de la cadena `s2` en la cadena `s1`.

B.2 Entrada y salida

Las siguientes funciones se hallan definidas en el fichero de cabeceras `stdio.h`.

B.2.1 Entrada y salida básica

Hay varias funciones que proporcionan entrada y salida básica. Probablemente las más conocidas sean:

- `int getchar()` Lee un carácter del teclado.
- `char *gets( char string[] )` Lee una cadena de caracteres del teclado.
- `int putchar( char ch )` Escribe un carácter por pantalla. Devuelve el carácter escrito.
- `int puts( char string[] )` Escribe una cadena de caracteres por pantalla. Devuelve el número de caracteres escritos.

B.2.2 Entrada y salida con formato

Ya hemos visto el uso de la entrada y salida con formato mediante las funciones `printf` y `scanf`. Veámoslas ahora con más detalle:

```
int printf( char format[], ...  )
```

Escribe en pantalla la lista de argumentos de acuerdo con el formato especificado para cada uno de ellos, y devuelve el número de caracteres escritos. El formato consta de caracteres ordinarios (que se escriben directamente en pantalla) y de especificadores de formato denotados por el carácter %. Debe haber tantos especificadores de formato como argumentos. La forma general de uno de estos especificadores es la siguiente:

```
%[-|+][ancho][.prec][h|l|L]tipo
```

donde:

- `tipo` especifica el tipo de datos del argumento según la tabla:

`tipo`	**Argumento**	**Formato de salida**
c	char	carácter
i, d	int	entero decimal: dígitos 0, . . . , 9
o	”	entero octal: dígitos 0, . . . , 7
x, X	”	entero hexadecimal: dígitos 0, . . . , 9, A, . . . , F
u	”	entero decimal sin signo
s	char *	cadena de caracteres hasta ’\0’
f	double/float	[-]dddd.dddd
e, E	”	notación científica: [-]d.dddd[e/E][+/-]ddd
g, G	”	la forma más compacta entre %f y %e
p	puntero	dirección de memoria
%	ninguno	carácter %

- `[h|l|L]` como modificador del tipo de datos básico. Se usa `h` para `short int`, `l` para `long int` y `double`, y `L` para `long double`.
- `[.prec]` número de decimales al escribir un número de coma flotante, o número de caracteres al escribir una cadena de caracteres.
- `[ancho]` número de espacios empleados para la escritura. Si es inferior al necesario se ignora. `ancho` puede tomar dos valores:
  - `n` Se emplean `n` espacios rellenando con blancos el espacio sobrante.
  - `0n` Se emplean `n` espacios rellenando con 0s el espacio sobrante.
- `[-|+]` Se usa `-` para justificar a la izquierda rellenando con blancos, y `+` para forzar la escritura del signo en los números.

Veamos algunos ejemplos ilustrativos:

```
printf( "%030.5f", 1.5236558 );     000000000000000000000001.52326
printf( "%+30.5f", 1.5236558 );                            +1.52326
printf( "%+-030.5f", 1.5236558 );  +1.52326000000000000000000000
printf( "%8.3s", "hola" );                 hol
printf( "%-08.3s", "hola" );        hol00000
```

`int scanf( char format[], ... )`

Lee datos del teclado (carácter a carácter) y los coloca en las direcciones de memoria especificadas en la lista de argumentos de acuerdo con el formato. Devuelve el número de argumentos leídos. El formato consta de caracteres ordinarios (que se espera se tecleen) y de especificadores de formato denotados por el carácter %. Debe haber tantos especificadores de formato como direcciones de argumentos donde almacenar los datos leídos. La forma general de uno de estos especificadores es la siguiente:

`%[*][ancho][h|l|L]tipo`

donde:

- `tipo` especifica el tipo de datos del argumento según la tabla:

`tipo`	**Argumento**	**Entrada esperada**
c	char *	carácter
i	int *	entero decimal, octal o hexadecimal
d	”	entero decimal
o	”	entero octal
x	”	entero hexadecimal
u	”	entero decimal sin signo
I	long int *	entero decimal, octal o hexadecimal
D	”	entero decimal
O	”	entero octal
X	”	entero hexadecimal
U	”	entero decimal sin signo
s	char []	cadena de caracteres hasta blanco, tabulador o salto de línea
f	double/float	número en coma flotante
e, E	”	”
g, G	”	”
p	puntero	dirección de memoria hexadecimal: YYYY:ZZZZ o ZZZZ
%	ninguno	carácter %

- `*` no asigna el argumento leído a ninguna variable de los argumentos.

El resto de campos del modificador son idénticos al caso de `printf`. Sin embargo existen un par de convenciones especiales que merece la pena destacar:

- `%[set]` que permite leer una cadena de caracteres hasta encontrar un carácter que no pertenezca al conjunto `set` especificado. Dicho carácter no se lee.
- `%[^set]` que permite leer una cadena de caracteres hasta encontrar un carácter que pertenezca al conjunto `set` especificado. Dicho carácter no se lee.

B.2.3 Ficheros

- `int fclose( FILE *fich )` Cierra el fichero `fich` y devuelve un código de error.
- `int feof( FILE *fich )` Comprueba si se ha llegado al final del fichero `fich`.

- `int ferror( FILE *fich )` Comprueba si se ha producido algún error en alguna operación sobre el fichero `fich`.
- `int fflush( FILE *fich )` Fuerza la escritura en disco de las escrituras diferidas realizadas sobre `fich`.
- `int fgetc( FILE *fich )` Lee un carácter de `fich`.
- `char *fgets( char string[], int max, FILE *fich )` Lee una cadena de hasta `max` caracteres de `fich`.
- `FILE *fopen( char nombre[], char modo[] )` Abre el fichero con el `nombre` y `modo` de apertura especificados. `modo` puede ser: `"r"` para lectura, `"w"` para escritura y `"a"` para añadir información al final del fichero.
- `int fprintf( FILE *fich, char formato[], ... )` Escritura con formato en `fich`. Ver `printf`.
- `int fputc( int c, FILE *fich )` Escribe el carácter `c` en `fich`.
- `int fputs( char string[], FILE *fich )` Escribe una cadena de caracteres en `fich`.
- `int fscanf( FILE *fich, char formato[], ... )` Lectura con formato de `fich`. Ver `scanf`.
- `int getc( FILE *fich )` Lee un carácter de `fich`.
- `int putc( int c, FILE *fich )` Escribe el carácter `c` en `fich`.
- `void rewind( FILE *fich )` Sitúa el *cursor* para lecturas/escrituras de `fich` al principio del mismo.
- `int sprintf( char string[], char formato[], ... )` Escritura con formato en una cadena caracteres. Ver `printf`.
- `int sscanf( char buffer[], char formato[], ... )` Lectura con formato de una cadena de caracteres. Ver `scanf`.
- `int ungetc( int c, FILE *fich )` Devuelve el carácter `c`, leído previamente, al fichero `fich` de donde fue leído.

B.3 Funciones matemáticas

Las siguientes funciones se hallan definidas en el fichero de cabeceras `math.h`:

- `double acos( double x )` Calcula el arco coseno de `x`.
- `double asin( double x )` Calcula el arco seno de `x`.
- `double atan( double x )` Calcula el arco tangente de `x`.

- `double atan2( double y, double x )` Calcula el arco tangente de `y/x`.
- `double ceil( double x )` Calcula el entero más pequeño que es mayor que `x`.
- `double cos( double x )` Calcula el coseno de `x` en radianes.
- `double cosh( double x )` Calcula el coseno hiperbólico de `x`.
- `double exp( double x )` Calcula $e^{\mathtt{x}}$.
- `double fabs( double x )` Calcula el valor absoluto de `x`.
- `double floor( double x )` Calcula el entero más grande que es menor que `x`.
- `labs( long n )` Calcula el valor absoluto de `n`.
- `double log( double x )` Calcula el logaritmo natural de `x`.
- `double log10( double x )` Calcula el logaritmo en base 10 de `x`.
- `double pow( double x, double y )` Calcula $\mathtt{x}^{\mathtt{y}}$.
- `double sin( double x )` Calcula el seno de `x` en radianes.
- `double sinh( double x )` Calcula el seno hiperbólico de `x`.
- `double sqrt( double x )` Calcula la raíz cuadrada de `x`.
- `void srand( unsigned seed )` Fija un nuevo *germen* para el generador de números aleatorios (`rand`).
- `double tan( double x )` Calcula la tangente de `x` en radianes.
- `double tanh( double x )` Calcula la tangente hiperbólica de `x`.

B.4 Clasificación y manipulación de caracteres

Las siguientes funciones se hallan definidas en el fichero de cabeceras `ctype.h`:

- `int isalnum( int c )` Devuelve cierto si `c` es un carácter alfanumérico.
- `int isalpha( int c )` Devuelve cierto si `c` es una letra.
- `int isascii( int c )` Devuelve cierto si `c` corresponde a un código ASCII.
- `int iscntrl( int c )` Devuelve cierto si `c` es un carácter de control.
- `int isdigit( int c )` Devuelve cierto si `c` es un dígito decimal.
- `int isgraph( int c )` Devuelve cierto si `c` es un carácter gráfico.
- `int islower( int c )` Devuelve cierto si `c` es una letra minúscula.
- `int isprint( int c )` Devuelve cierto si `c` es un carácter imprimible.

- `int ispunct( int c )` Devuelve cierto si `c` es un símbolo de puntuación.
- `int isspace( int c )` Devuelve cierto si `c` es un carácter de espaciado.
- `int isupper( int c )` Devuelve cierto si `c` es una letra mayúscula.
- `int isxdigit( int c )` Devuelve cierto si `c` es un dígito hexadecimal.
- `int toascii( int c )` Obtiene el código ASCII de `c`.
- `tolower( int c )` Convierte `c` a minúscula.
- `int toupper( int c )` Convierte `c` a mayúscula.

B.5 Conversión de datos

Las siguientes funciones se hallan definidas en el fichero de cabeceras `stdlib.h`:

- `double atof( char string[] )` Convierte una cadena de caracteres en un número de coma flotante.
- `int atoi( char string[] )` Convierte una cadena de caracteres en un número entero.
- `int atol( char string[] )` Convierte una cadena de caracteres en un número entero de doble precisión.

B.6 Manipulación de directorios

Las siguientes funciones se hallan definidas en el fichero de cabeceras `dir.h`:

- `int chdir( char path[] )` Cambia el directorio de trabajo actual de acuerdo con el `path` especificado.
- `char *getcwd( char path[], int numchars )` Devuelve el nombre del directorio de trabajo actual.
- `int mkdir( char path[] )` Crea un nuevo directorio con el nombre especificado en `path`.
- `int rmdir( char path[] )` Borra el directorio con el nombre especificado en `path`.

B.7 Memoria dinámica

Una de las características más importantes de un programa es la cantidad de memoria que necesita para ejecutarse. Es importante que un programa no desperdicie memoria. Esto plantea un serio problema cuando declaramos las variables, esencialmente las tablas, ya que deberemos dimensionar el espacio de memoria para el peor caso posible.

Para evitar este problema existen funciones que permiten una gestión *dinámica* de la memoria, es decir, permiten que un programa adquiera memoria según la necesite, y la vaya liberándola cuando deje

de necesitarla. C dispone de las siguiente funciones para gestionar de forma dinámica la memoria, todas ellas están definidas en el fichero `stdlib.h`:

- `void *malloc( size_t num_bytes )`
  Reserva un bloque de memoria de `num_bytes` bytes. Devuelve un puntero al primer byte del bloque de memoria reservado, o `NULL` si no hay suficiente memoria disponible.

- `void *calloc( size_t num_elems, size_t tam_elem )`
  Reserva un bloque de memoria capaz de almacenar `num_elems` de `tam_elem` bytes cada uno. Este espacio de memoria es inicializado con ceros. Devuelve un puntero al primer byte del bloque de memoria reservado, o `NULL` si no hay suficiente memoria disponible.

- `void *realloc( void *ptr, size_t num_bytes )`
  Cambia el tamaño del bloque de memoria apuntado por `ptr` para que tenga `num_bytes` bytes. Devuelve un puntero al primer byte del nuevo bloque de memoria reservado, o `NULL` si no hay suficiente memoria disponible.

- `void free( void *ptr )`
  Libera el bloque de memoria apuntado por `ptr`. Dicho bloque debe haber sido previamente obtenido mediante `malloc`, `calloc` o `realloc`. Si `ptr` es `NULL`, no se hace nada.

El tipo de datos `size_t` es un número natural (sin signo). Cuando llamamos a estas funciones y les pasamos como parámetros varibles de tipo entero (`short`, `int` o `long`), se realiza una conversión de tipo de forma automática.

Junto con estas funciones usaremos el operador de C `sizeof(tipo_de_datos)`. Este operador retorna el número de bytes que ocupa una variable del tipo `tipo_de_datos`, tanto si este tipo está predefinido en C (`int`, `float`, etc.) como si es un tipo definido por el programador.

Veamos algunos ejemplos que ilustran el empleo de memoria dinámica.

El siguiente ejemplo gestiona de forma dinámica dos vectores.

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

typedef struct
      {
         long DNI;
         char nom[256];
      } Tpersona;

void main( )
{
   int i, lon;
   double *nota;
   Tpersona *alumno;
```

```
    do
    {
       printf( "Cuántos alumnos hay?\n" );
       scanf( "%d", &lon );
    } while( lon < 0 );

    nota = malloc( lon*sizeof(double) );
    alumno = malloc( lon*sizeof(Tpersona) );

    if ((nota == NULL) || (alumno == NULL))
    {
       printf( " No hay memoria suficiente\n" );
       exit(0);
    }

    ...
    /* Introducción de datos y notas de cada alumno.  */
    ...

    for (i= 0; i< lon; i++)
       printf( "Alumno:%d nombre:%s DNI:%ld nota:%lf\n",
                i, alumno[i].nom, alumno[i].DNI, nota[i] );

    free( alumno );
    free( nota );
}
```

Hay varios puntos a destacar:

- Notar que en la declaración de variables no declaramos ningún vector. En su lugar declaramos *punteros*. En este caso un puntero al tipo `double` y otro al tipo `Tpersona`. En estos punteros almacenamos las direcciones de memoria que devuelve `malloc`.
- Para indicarle a la función `malloc` qué cantidad de bytes de memoria necesitamos, hemos usado el operador `sizeof`. Los bytes requeridos son el número de elementos multiplicado por el tamaño en bytes de cada elemento. Cabe notar que `sizeof` se puede usar también con tipos definidos por el programador como el tipo `Tpersona`.
- Después de llamar a `malloc` comprobamos que los punteros no sean `NULL`. Si `lon` es muy grande, puede ocurrir que el computador no tenga memoria suficiente. *Esta comprobación de error siempre debe hacerse.*
- Cuando tenemos los bloques de memoria ya reservados, podemos acceder a ellos a través de los punteros. En este caso, la notación para acceder a través de los punteros es idéntica a la que se usa con vectores. Pero no debemos olvidar que `alumno` y `nota` son punteros.
- En el momento en que los bloques de memoria ya no son necesarios, debemos liberarlos. Para ello usamos la función `free`. Cualquier intento de acceder a los bloques de memoria después de la llamada a `free` generaría un error de ejecución.

En este ejemplo multiplicamos la matriz `A` por el vector `x` dejando el resultado en el vector `y`.

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

void main( )
{
   int i, j, nfil, ncol;
   double *x, *y, *A;

   do
   {
      printf( " Número de filas?\n" );
      scanf( "%d", &nfil );
   } while( nfil < 0 );

   do
   {
      printf( " Número de columnas?\n" );
      scanf( "%d", &ncol );
   } while( ncol < 0 );

   A = malloc( nfil*ncol*sizeof(double) );
   x = malloc( ncol*sizeof(double) );
   y = calloc( nfil, sizeof(double) );

   if ((x == NULL) || (y == NULL) || (A == NULL))
   {
      printf( " No hay memoria suficiente\n" );
      exit(0);
   }

   ...
   /* Introducción del vector x y la matrix A */
   ...

   for (i= 0; i< nfil; i++)
   {
      for (j= 0; j< ncol; j++)
         y[i] = y[i] + A[i*ncol+j] * x[j];
   }

   for (i= 0; i< nfil; i++)
      printf( "y[%d] = %lf\n", i, y[i] );

   free( A );
```

```
    free( x );
    free( y );
}
```

Los puntos más destacables son los siguientes:

- Para reservar memoria para el vector `y` utilizamos la función `calloc`. De esta forma el bloque de memoria queda inicializado a *cero* y no es necesario inicializar cada componente de `y` en el algoritmo de multiplicación.
- La notación para acceder a los bloques de memoria `x` e `y` a través de los punteros coincide con la que usaríamos si fuesen vectores declarados de forma estática. Pero no pasa lo mismo con la matriz.

Finalmente, en este ejemplo modificamos el tamaño de un bloque de memoria, que previamnete había sido reservado mediante la función `malloc`.

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

void main( )
{
   int lon1, lon2;
   double *vec;

   do
   {
      printf( "Longitud del vector?\n" );
      scanf( "%d", &lon1 );
   } while( lon1 < 0 );

   vec = malloc( lon1*sizeof(double) );

   if (vec == NULL)
   {
      printf( " No hay memoria suficiente\n" );
      exit(0);
   }

   ...
   /* Aquí trabajamos con vec */
   ...

   do
   {
      printf( " Nueva longitud del vector?\n" );
      scanf( "%d", &lon2 );
```

```
    } while( lon2 < 0 );

    vec = realloc( vec, lon2*sizeof(double) );

    if (vec == NULL)
    {
       printf( " No hay memoria suficiente\n" );
       exit(0);
    }
    ...
    /* Aquí trabajamos con vec */
    ...

    free( vec );
}
```

La función `realloc` nos permite modificar el tamaño del bloque de memoria reservado, pero no modifica los datos almacenados en dicho bloque. Es decir:

- Si `lon2 < lon1`, tendremos un bloque de memoria más pequeño, pero los `lon2` valores almacenados seguirán siendo los mismos.
- Si `lon2 > lon1`, tendremos un bloque de memoria más grande. Los primeros `lon1` valores serán los mismos que había antes de la llamada a `realloc`, mientras que los `lon2 - lon1` valores finales serán aleatorios (no estarán inicializados).

Apéndice C

Sistemas de numeración

Un computador usa el sistema de numeración en base 2 debido a cómo funcionan los dispositivos electrónicos básicos (los transistores) que lo forman. En el sistema de numeración en base dos sólo existen 2 cifras el 0 y el 1. A las cifras de un número en base 2 se las denomina *bits*. A un grupo de 8 bits se le denomina *byte*.

C.1 Naturales

Los números naturales se representan mediante un código llamado *binario natural*, que consiste simplemente en expresar el número en base 2. Si disponemos n bits para representar números naturales tendremos las 2^n combinaciones que se muestran en la tabla C.1.

C.2 Enteros

Los números enteros se representan mediante un código llamado *complemento a 2*. Este código se usa porque simplifica notablemente la construcción de los circuitos electrónicos necesarios para realizar operaciones aritméticas, fundamentalmente sumas y restas.

El complemento a 2 de un número se calcula de la siguiente forma: si el número es positivo su complemento a 2 coincide con su expresión en binario natural; si el número es negativo se debe escribir la representación en binario natural de su módulo (valor absoluto), complementar a 1 dicha expresión

Tabla C.1: Representación de números naturales en binario natural

Valor Decimal	Binario Natural
0	`0.....0`
1	`0....01`
2	`0...010`
3	`0...010`
4	`0...100`
...	...
$2^n - 1$	`1.....1`

Tabla C.2: Representación de números enteros en complemento a 2

Valor Decimal	Complemento a 2
-2^{n-1}	`10....0`
...	...
-1	`1.....1`
0	`0.....0`
1	`0....01`
...	...
$+2^{n-1}-1$	`01....1`

(cambiar los 0 por 1 y viceversa), y finalmente sumarle 1. Por ejemplo si disponemos de 4 bits para representar números enteros y deseamos representar el número -3, tendríamos que:

```
 3 = 0011
     1100        Complemento a 1
     +  1
-3 = 1101        Complemento a 2
```

En general, si disponemos de n bits, podemos representrar los números en complemento a 2 que se muestran en la tabla C.2.

La codificación en complemento a 2 tiene algunas propiedades interesantes como las siguientes:

- El cero tiene una única representacion: `0...0 = 1...1 + 1`
- Todos los números del mismo signo empiezan por el mismo bit: 1 para los negativos, 0 para los positivos
- La suma/resta de numeros en complemento a 2 se puede realizar mediante la suma binaria bit a bit, despreciando el último acarreo. Por ejemplo:

```
-2 - 1 = -3        1....10  +  1...1  =  1...10
 2 - 1 =  1        0...010  +  1...1  =  0...01
```

C.3 Reales

Los números reales se representan mediante un código llamado *coma flotante*. Este código es simplemente la representación en notación científica normalizada y en base 2.

Recordemos que un numero en notación científica se representa mediante la siguiente expresión: $\pm mantisa \times base^{exponente}$. Donde la $mantisa$ es un número real con la coma decimal colocada a la derecha o a la izquierda de la primera cifra significativa (normalización por la derecha o por la izquierda); la $base$ es la misma que la base de numeración usada para representar mantisa y exponente; y el $exponente$ es un número entero. Los siguientes ejemplos muestran números representados en notación científica en base 10, normalizados por la derecha y por la izquierda:

Tabla C.3: Representación de números enteros en $exceso\ 2^{e-1}$

Valor Decimal	**Complemento a 2**	$\boldsymbol{exceso\ 2^{e-1}}$
-2^{e-1}	`10....0`	`0.....0`
...	...	...
-1	`1.....1`	`01....1`
0	`0.....0`	`10....0`
1	`0....01`	`10...01`
...	...	...
$+2^{e-1}-1$	`01....1`	`11....1`

$-3,141592 \cdot 10^{0} = -0,3141592 \cdot 10^{1}$
$2,53547 \cdot 10^{-3} = 0,253547 \cdot 10^{-2}$

La notación científica usada por los computadores tiene algunos detalles especiales debidos a que usa la base 2.

1. En la memoria unicamente se almacena:

   - una secuencia de m bits que representa la mantisa,
   - una secuencia de e bits que representa el exponente,
   - el signo de la mantisa se almacena usando 1 bit (0 significa positivo y 1 significa negativo).

   La base no es necesario almacenarla, ya que siempre es 2. Si pudiésemos ver un número real almacenado en la memoria del computador veríamos una secuencia de bits como la siguiente:

   $$\underbrace{\overbrace{1}^{1\ bit}}_{signo}\ \underbrace{\overbrace{101\ldots010}^{m\ bits}}_{mantisa}\ \underbrace{\overbrace{10010}^{e\ bits}}_{exponente}$$

2. En base 2 sólo existen 2 cifras, el 0 y el 1. Así pues, el primer bit significativo siempre será un 1, por lo que este primer bit no se almacena en la memoria. Los circuitos electrónicos que operan con datos en coma flotante ya tienen en cuenta que delante de todos los bits de la mantisa siempre hay un 1. A este bit que no se almacena se le denomina *bit oculto* o *implícito*.

3. El exponente se almacena usando un código llamado $exceso\ 2^{e-1}$, donde e es el número de bits usados para almacenar el exponente. Este código proviene de rotar de forma cíclica 2^{e-1} posiciones la tabla del complemento a 2 (ver Tab. C.3).

   Para calcular un código en exceso podemos usar la siguiente fórmula:

   $$\textit{Valor decimal} = \textit{Valor en binario natural} - 2^{e-1}$$

   Notar que en el código en exceso todos los números negativos comienzan por 0 y todos los números positivos comienzan por 1. Además, a valores crecientes les corresponden códigos que leídos en binario natural también son crecientes. Es decir: $-2 < 1 \quad \Leftrightarrow \quad 01\ldots10 < 10\ldots01$. De esta forma podemos comparar números de un código en exceso usando un simple comparador de números en binario natural. Sin embargo, los circuitos para sumar/restar números codificados en exceso, ya no serán un simple sumador binario. Por razones históricas se decidió usar el código en exceso en vez del código en complemento a 2 para representar los exponentes.

Tabla C.4: Representación de números reales

Valor Decimal	Signo	Mantisa	Exponente
± 0	x	$0 \ldots 0$	$0 \ldots 0$
NAN	x	$x \ldots x$	$0 \ldots 0$
$\pm Mínimo$	x	$0 \ldots 0$	$0 \ldots 1$
$\ldots$	$\ldots$	$\ldots$	$\ldots$
$\pm Máximo$	x	$1 \ldots 1$	$1 \ldots 1$

4. Debido al uso del bit oculto, el número cero no se puede representar, ya que sea cual sea la mantisa, ésta nunca será cero. Dado que el número cero es un número importante, que es necesario poder representar, se hace una excepción a la regla de representación. De forma arbitraria se decide que el número cero se representa mediante la combinación en que todos los bits de la mantisa y el exponente son ceros.

5. Además, las diferentes combinaciones de mantisa con el exponente cuyo código son todo ceros tampoco se usan como números. Estos códigos se reservan como códigos de error, generados por los circuitos aritméticos cuando se producen condiciones de error como: una división en que el divisor es cero, la raíz cuadrada de un número negativo, etc. A estos códigos de error se les denomina *NAN* (del ingles, *Not A Number*). Por lo tanto:

   - El exponente mínimo es $-(2^{e-1}-1)$ y se representa mediante la combinación $0 \ldots 01$.
   - El exponente máximo es $+(2^{e-1}-1)$, y se representa mediante la combinación $1 \ldots 1$.
   - El número representable más cercano a cero es $\pm 1.0 \times 2^{Exp.Mínimo}$. Este número tiene toda su mantisa con ceros y el exponente es la combinación $0 \ldots 01$.
   - El número representable más grande es $\pm 2(1-2^{-(m+1)}) \times 2^{Exp.Máximo}$. Este número tiene toda su mantisa con unos y el exponente es la combinación $1 \ldots 1$.

   En la tabla C.4 se muestran ordenados los códigos de un formato de coma flotante. El símbolo x significa cualquier bit 0 o 1.

6. Cabe notar que cuantos más bits se usen para representar la mantisa, mayor precisión se tiene en la representación (menor error relativo). Cuantos más bits usemos para representar el exponente mayor rango numérico abarcaremos.

Si cada computador usase su propio sistema de representación en coma flotante, podríamos tener valores diferentes al ejecutar el mismo programa en computadores diferentes. Para evitar este problema, la representación en coma flotante está estandarizada por la organización IEEE (*Institute of Electrical and Electronics Engineers*) bajo el estándar IEEE-754. Actualmente todos los computadores cumplen con dicho estándar.

C.3.1 Problemas derivados de la representación en coma flotante

Para representar números reales en un computador siempre usaremos una secuencia finita de bits. En consecuencia, nunca podremos representar todos los números. Al subconjunto de números que podemos representar en un formato coma flotante se le denomina *rango de representación*.

Debido a que el rango de representación es finito, las operaciones aritméticas no tienen las propiedades habituales. Por ejemplo, la suma ya no es asociativa. Supongamos un formato de coma flotante donde el número máximo representable es el 8.0. Entonces tendremos que:

$$(8.0 + 2.0) - 4.0 \neq 8.0 + (2.0 - 4.0)$$
$$(8.0 + 2.0) - 4.0 = 10.0 - 4.0 \Rightarrow ERROR$$
$$8.0 + (2.0 - 4.0) = 8.0 - 2.0 = 6.0 \Rightarrow CORRECTO$$

A esta situación se la denomina *overflow* (desbordamiento por arriba), y ocurre cuando una operación genera un número mayor que el máximo representable. Además del *overflow* puede producirse el *underflow* (desbordamiento por abajo) cuando el resultado de una operación cae en la zona de números situados entre 0 y el número mínimo representable.

En conclusión, al realizar cálculos en coma flotante deberemos tener en cuenta el orden en el que se realizan las operaciones, para evitar las citadas situaciones de error.

Apéndice D

Tabla de caracteres ASCII

Además de poder representar valores numéricos, es preciso que un ordenador sea capaz de almacenar los signos alfanuméricos que usamos normalmente en el lenguaje humano. Para ello se usa un código de 8 bits llamado código ASCII. Cada combinación de este código representa un carácter. Los periféricos, como la pantalla o una impresora, son capaces de reconocer estos códigos y mostrar/imprimir el símbolo correspondiente.

Entre los caracteres representados además de los símbolos que usamos para escribir (letras, cifras, signos de puntuación, etc) también hay caracteres de control, necesarios para el funcionamiento de los periféricos. Por ejemplo, el carácter `\n` provoca un salto de línea cuando es mostrado en una pantalla.

Originalmente el código ASCII sólo usaba 7 bits para codificar caracteres, el octavo bit (bit de paridad) se usaba para control de errores. Por lo tanto, el código ASCII original sólo podía representar $128 = 2^7$ caracteres diferentes. Actualmente, este control de errores no es necesario debido al perfeccionamiento de los equipos, por ello el octavo bit puede ser usado también para codificar caracteres. A este código se le conoce como ASCII extendido y codifica $256 = 2^8$ caracteres.

La tabla D.1 muestra los primeros 128 caracteres de acuerdo con la codificación estándar ASCII.

Tabla D.1: Caracteres y sus códigos ASCII

ASCII	char	ASCII	char	ASCII	char	ASCII	char
0	NUL \0	32	SP ' '	64	@	96	‘
1	SOH	33	!	65	A	97	a
2	STX	34	”	66	B	98	b
3	ETX	35	#	67	C	99	c
4	EOT	36	$	68	D	100	d
5	ENQ	37	%	69	E	101	e
6	ACK	38	&	70	F	102	f
7	BEL \a	39	’	71	G	103	g
8	BS \b	40	(	72	H	104	h
9	HT \t	41	)	73	I	105	i
10	LF \n	42	∗	74	J	106	j
11	VT \v	43	+	75	K	107	k
12	FF \f	44	,	76	L	108	l
13	CR \r	45	–	77	M	109	m
14	SO	46	.	78	N	110	n
15	SI	47	/	79	O	111	o
16	DLE	48	0	80	P	112	p
17	DC1	49	1	81	Q	113	q
18	DC2	50	2	82	R	114	r
19	DC3	51	3	83	S	115	s
20	DC4	52	4	84	T	116	t
21	NAK	53	5	85	U	117	u
22	SYN	54	6	86	V	118	v
23	ETB	55	7	87	W	119	w
24	CAN	56	8	88	X	120	x
25	EM	57	9	89	Y	121	y
26	SUB	58	:	90	Z	122	z
27	ESC	59	;	91	[	123	{
28	FS	60	¡	92	\	124	—
29	GS	61	=	93	]	125	}
30	RS	62	¿	94	∧	126	∼
31	US	63	?	95	_	127	DEL

Apéndice E

Bibliografía y recursos WEB

El Lenguaje de programación C
Brian W. Kernighan y Dennis M. Ritchie
Prentice Hall, 1992, segunda edición.
`http://cm.bell-labs.com/cm/cs/cbook/index.html`

De obligada referencia, es el primer libro sobre lenguaje C. Uno de los autores, Dennis M. Ritchie, es uno de los creadores del lenguaje. Incluye la definición oficial de C y un gran número de ejemplos interesantes, aunque en algunos aspectos el material presentado es obsoleto. Presupone que el lector está familiarizado con la programación de sistemas.

Programación en C
Byron Gottfried
Mc Graw Hill, 1997, segunda edición.

Extenso y eshaustivo libro con multitud de ejemplos detallados. Está estructurado como un libro de texto, por lo que contiene gran cantidad de ejercicios, cuestiones de repaso, etc. Para principiantes.

The Annotated ANSI C Standard
Herbert Schildt
Osborne - Mc Graw Hill, 1993.

El estándar ANCI C comentado y anotado por Herbert Shildt, miembro observador del comité encargado de desarrollar el estándar. ANSI C determina las reglas fundamentales que todo programador en C debería observar para crear programas funcionales y portables. Para lectores avanzados.

C by example
Greg M. Perry
Que Corporation, 1993.

Interesante libro que enseña a programar en C, construyendo programas paso a paso, desde el primer momento. Separa los conceptos complicados del lenguaje en varios capítulos cortos, de fácil asimilación. Incluye gran cantidad de ejercicios. Para principiantes.

The Development of the C Language

Dennis M. Ritchie

History of Programming Languages Conference (HOPL-II), 1993.

`http://cm.bell-labs.com/cm/cs/who/dmr/chist.html`

Interesante artículo de divulgación que describe la historia de la creación del lenguaje C, allá por los años 70.

WEB: Programming in C

`http://www.lysator.liu.se/c/index.html`

Un sitio WEB muy interesante. Incluye multitud de enlaces a recursos en Internet sobre programación en C, historia del lenguaje, curiosidades, etc.

WEB: Frequently Asked Questions about C

`http://www.eskimo.com/ scs/C-faq/top.html`

En este sitio WEB encontraremos una recopilación de las preguntas más habituales sobre programación en lenguaje C del grupo de noticias en Internet `comp.lang.c`.