Curso de programación

C/C++

Curso de programación

C/C++

Fco. Javier Ceballos Sierra
Profesor titular de la
Escuela Universitaria Politécnica
Universidad de Alcalá de Henares

Curso de programación C/C++

Editado por:
RA-MA Editorial
Ctra. de Canillas, 144
28043 MADRID
Teléf.: (91) 381 03 00
Telefax: (91) 381 03 72
ISBN: 84-7897-200-5
Depósito Legal: M-27061-1995
Autoedición: Fco. Javier Ceballos
Imprime: Albadalejo, S.L.
Impreso en España
Primera impresión 1.500 ejemplares. Septiembre 1995

Dios me libre de los toros mansos,
que los bravos ya los veo yo venir.

Dedico esta obra
a María del Carmen, mi esposa,
y a mis hijos Francisco y Javier.

ÍNDICE

PRÓLOGO

Este es un libro para aprender C, llegando a alcanzar al final, un nivel elevado de conocimientos. La forma en la que se ha estructurado el libro ha sido precisamente, pensando en ese objetivo. El libro se ha dividido en doce capítulos que van presentando el lenguaje poco a poco, empezando por lo más sencillo, viendo cada tema a su tiempo, hasta llegar al final donde se habrá visto todo lo referente a la programación con el lenguaje C, sin apenas encontrar dificultades.

Se completa el estudio de C con un capítulo referente a estructuras dinámicas y otro de algoritmos de uso común.

Este libro posee varias características dignas de resaltar. Es breve en teoría y abundante en ejemplos, lo que le hará aún más fácil el aprendizaje. La metodología utilizada en el desarrollo de los programas está fundamentada en las técnicas de desarrollo para realizar una programación estructurada.

La materia total que compone el *Curso de programación con C/C++*, se ha dividido en los siguientes capítulos y apéndices:

1. Fases en el desarrollo de un programa
2. Elementos del lenguaje C
3. Estructura de un programa
4. Entrada y salida estándar
5. Sentencias de control
6. Tipos estructurados de datos
7. Punteros
8. Funciones
9. Funciones estándar de E/S
10. El preprocesador de C
11. Estructuras dinámicas de datos
12. Algoritmos

A. Depurar un programa
B. Visual C++
C. Códigos de caracteres
D. Índice alfabético

Todo esto se ha documentado con abundantes PROBLEMAS RESUELTOS, muchos de ellos válidos como parte integrante en el desarrollo de aplicaciones.

El lenguaje C ha ido evolucionando a lo largo de su historia. Producto de esta evolución fue el lenguaje C++ y finalmente, el diseño de una amplia biblioteca de funciones para el desarrollo de la programación visual. De ahí que este libro sea el primero de un conjunto de tres, que conducen a desarrollar aplicaciones con una interfaz gráfica de usuario, pasando por la programación orientada a objetos.

Puesto que C++ fue desarrollado a partir del lenguaje de programación C, con pocas excepciones, incluye a C, de ahí el título de este libro. Esta parte de C incluida en C++ es conocida como C–, y podría compilarse como C++ sin problemas. No obstante, cuando C++ se utiliza para lo que fue pensado, para realizar una programación orientada a objetos, los conocimientos nuevos que hay que adquirir son cuantiosos.

Si su propósito es llegar a desarrollar aplicaciones vistosas como lo son las aplicaciones a base de ventanas, más bien conocidas como aplicaciones para Windows, después de estudiar este libro tiene que aprender programación orientada a objetos utilizando C++, para lo que le recomiendo mi otro libro *Programación Orientada a Objetos con C++* publicado por RA-MA. Cuando sepa desarrollar programas orientados a objetos, puede dar el último paso, desarrollar aplicaciones para Windows, las cuales están fundamentadas en la programación orientada a objetos; para esta última parte le recomiendo mi otro libro *Visual C++, Aplicaciones para Windows* publicado también por la editorial RA-MA.

Si su objetivo no es la programación visual, sino profundizar en aplicaciones C para MS-DOS vea mi otro libro *Enciclopedia del Lenguaje C*. En él encontrará abundante información para trabajar con C en el sistema operativo MS-DOS.

Por el contrario, si lo que usted persigue es profundizar en aplicaciones C para UNIX, entonces le recomiendo el libro de Fco. Manuel Márquez García titulado *UNIX, Programación Avanzada*, publicado también por la editorial RA-MA.

Agradecimientos

He recibido ayuda de algunas personas durante la preparación de este libro, y por ello estoy francamente agradecido. En especial, quiero expresar mi agradecimiento a **Alfredo Gallego Gandarillas** y a **Francisco Manuel Márquez García**, profesores titulares de la Universidad de Alcalá, que participaron en la revisión de este libro y contribuyeron aportando ideas y ejemplos.

Francisco Javier Ceballos Sierra

CAPÍTULO 1

FASES EN EL DESARROLLO DE UN PROGRAMA

En este capítulo aprenderá lo que es un programa, cómo escribirlo y qué hacer para que el ordenador lo ejecute y muestre los resultados perseguidos.

QUÉ ES UN PROGRAMA

Probablemente alguna vez haya utilizado un ordenador para escribir un documento o para divertirse con algún juego. Recuerde que en el caso de escribir un documento, primero tuvo que poner en marcha un procesador de textos, y que si quiso divertirse con un juego, lo primero que tuvo que hacer fue poner en marcha el juego. Tanto el procesador de textos como el juego son *programas* de ordenador.

Poner un programa en marcha es lo mismo que ejecutarlo. Cuando ejecutamos un programa, nosotros sólo vemos los resultados que produce (el procesador de textos muestra sobre la pantalla el texto que escribimos; el juego visualiza sobre la pantalla las imágenes que se van sucediendo) pero no vemos lo que hace el programa para conseguir esos resultados.

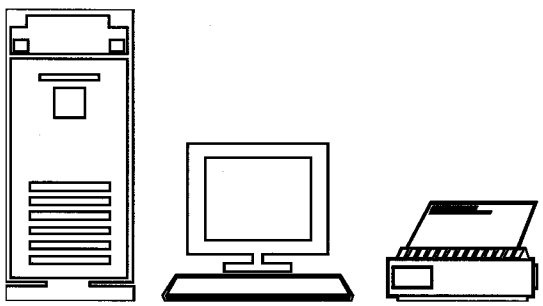
Si nosotros escribimos un programa, entonces sí que conocemos su interior y por lo tanto, sabemos cómo trabaja y por qué trabaja de esa forma. Esto es una forma muy diferente y curiosa de ver un programa de ordenador, lo cual no tiene nada que ver con la experiencia adquirida en la ejecución de distintos programas.

Ahora, piense en un juego cualquiera. La pregunta es ¿qué hacemos si queremos enseñar a otra persona a jugar? Lógicamente le explicamos lo que debe hacer; esto es, los pasos que tiene que seguir. Dicho de otra forma, le damos

instrucciones de cómo debe de actuar. Esto es lo que hace un programa de ordenador. Un *programa* no es nada más que una serie de instrucciones dadas al ordenador en un lenguaje entendido por él, para decirle exactamente lo que queremos hacer. Si el ordenador no entiende alguna instrucción, lo comunicará mediante un mensaje de error.

LENGUAJES DE PROGRAMACIÓN

Un programa tiene que escribirse en un lenguaje entendible por el ordenador. Desde el punto de vista físico, un ordenador es una máquina electrónica. Los elementos físicos (memoria, unidad de proceso, etc.) de que dispone el ordenador para representar los datos son de tipo binario; esto es, cada elemento puede diferenciar dos estados (dos niveles de voltaje). Cada estado se denomina genéricamente *bit* y se simboliza por *0* o *1*. Por lo tanto, para representar y manipular información numérica, alfabética y alfanumérica se emplean cadenas de *bits*. Según esto, se denomina *byte* a la cantidad de información empleada por un ordenador para representar un carácter; generalmente un *byte* es una cadena de ocho *bits*.



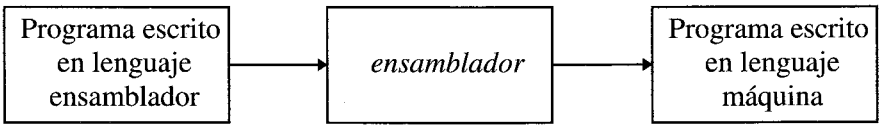
Así, por ejemplo, cuando un programa le dice al ordenador que visualice un mensaje sobre el monitor, o que lo imprima sobre la impresora, las instrucciones correspondientes para llevar a cabo esta acción, para que puedan ser entendibles por el ordenador, tienen que estar almacenadas en la memoria como cadenas de *bits*. Esto hace pensar que escribir un programa utilizando ceros y unos (lenguaje máquina), llevaría mucho tiempo y con muchas posibilidades de cometer errores. Por este motivo, se desarrollaron los lenguajes *ensambladores*.

Un lenguaje *ensamblador* utiliza *códigos nemotécnicos* para indicarle al hardware (componentes físicos del ordenador) las operaciones que tiene que realizar. Un código nemotécnico es una palabra o abreviatura fácil de recordar que representa una tarea que debe realizar el procesador del ordenador. Por ejemplo:

```
MOV AH, 4CH
```

El código *MOV* le dice al ordenador que mueva alguna información desde una posición de memoria a otra.

Para traducir un programa escrito en *ensamblador* a lenguaje máquina (código binario) se utiliza un programa llamado *ensamblador* que ejecutamos mediante el propio ordenador. Este programa tomará como datos nuestro programa escrito en lenguaje ensamblador y dará como resultado el mismo programa pero escrito en lenguaje máquina, lenguaje que entiende el ordenador.



Cada modelo de ordenador, dependiendo del procesador que utilice, tiene su propio lenguaje ensamblador. Debido a esto decimos que estos lenguajes están orientados a la máquina.

Hoy en día son más utilizados los lenguajes orientados al problema o lenguajes de alto nivel. Estos lenguajes utilizan una terminología fácilmente comprensible que se aproxima más al lenguaje humano.

Cada sentencia de un programa escrita en un lenguaje de alto nivel se traduce en general en varias instrucciones en lenguaje ensamblador. Por ejemplo:

```
printf("hola");
```

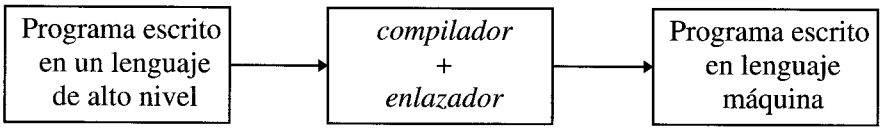
La función **printf** del lenguaje C le dice al ordenador que visualice en el monitor la cadena de caracteres especificada. Este mismo proceso escrito en lenguaje ensamblador necesitará de varias instrucciones.

A diferencia de los lenguajes ensambladores, la utilización de lenguajes de alto nivel no requiere en absoluto del conocimiento de la estructura del procesador que utiliza el ordenador, lo que facilita la escritura de un programa.

Compiladores

Para traducir un programa escrito en un lenguaje de alto nivel (programa fuente) a lenguaje máquina se utiliza un programa llamado *compilador*. Este programa tomará como datos nuestro programa escrito en lenguaje de alto nivel y dará como resultado el mismo programa pero escrito en lenguaje máquina, lenguaje que entiende el ordenador. Después de la traducción se ejecuta automáticamente un programa denominado *enlazador* encargado de incorporar las funciones de la

biblioteca del lenguaje utilizado necesarias para nuestro programa. Este último paso será explicado con más detalle más adelante. Si durante la traducción se detectan errores de sintaxis, el enlace no se efectúa.



Por ejemplo, un programa escrito en el lenguaje *C* necesita del compilador *C* para poder ser traducido. Posteriormente el programa traducido podrá ser ejecutado directamente por el ordenador.

Intérpretes

A diferencia de un compilador, un intérprete no genera un programa escrito en lenguaje máquina a partir del programa fuente, sino que efectúa la traducción y ejecución simultáneamente para cada una de las sentencias del programa. Por ejemplo, un programa escrito en el lenguaje *Basic* necesita el intérprete *Basic* para ser ejecutado. Durante la ejecución de cada una de las sentencias del programa, ocurre simultáneamente la traducción.

A diferencia de un compilador, un intérprete verifica cada línea del programa cuando se escribe, lo que facilita la puesta a punto del programa. En cambio la ejecución resulta más lenta ya que acarrea una traducción simultánea.

HISTORIA DEL LENGUAJE C

El C es un lenguaje de programación de propósito general. Sus principales características son:

- Programación estructurada.
- Economía en las expresiones.
- Abundancia en operadores y tipos de datos.
- Codificación en alto y bajo nivel simultáneamente.
- Reemplaza ventajosamente la programación en ensamblador.
- Utilización natural de las funciones primitivas del sistema.
- No está orientado a ningún área en especial.
- Producción de código objeto altamente optimizado.
- Facilidad de aprendizaje.

El lenguaje C nació en los laboratorios Bell de AT&T y ha sido estrechamente asociado con el sistema operativo UNIX, ya que su desarrollo se realizó en este sistema y debido a que tanto UNIX como el propio compilador C y la casi totalidad de los programas y herramientas de UNIX fueron escritos en C. Su eficiencia y claridad han hecho que el lenguaje ensamblador apenas haya sido utilizado en UNIX.

Este lenguaje está inspirado en el lenguaje B escrito por *Ken Thompson* en 1970 con intención de recodificar el UNIX, que en la fase de arranque estaba escrito en ensamblador, en vistas a su transportabilidad a otras máquinas. B era un lenguaje evolucionado e independiente de la máquina, inspirado en el lenguaje BCPL concebido por *Martin Richard* en 1967.

En 1972, *Dennis Ritchie* toma el relevo y modifica el lenguaje B, creando el lenguaje C y reescribiendo UNIX en dicho lenguaje. La novedad que proporcionó el lenguaje C sobre el B fue el diseño de tipos y estructuras de datos.

Los tipos básicos de datos eran **char** (carácter), **int** (entero), **float** (reales en simple precisión) y **double** (reales en doble precisión). Posteriormente se añadieron los tipos **short** (enteros de longitud $\leq$ longitud de un **int**), **long** (enteros de longitud $\geq$ longitud de un **int**), **unsigned** (enteros sin signo) y *enumeraciones*. Los tipos estructurados básicos de C son las *estructuras*, las *uniones* y los *arrays*. Estos permiten la definición y declaración de tipos derivados de mayor complejidad.

Las instrucciones de control de flujo de C son las habituales de la programación estructurada: **if**, **for**, **while**, **switch-case**, todas incluidas en su predecesor BCPL. C incluye también punteros y funciones y permite que cualquier función pueda ser llamada recursivamente.

Una de las peculiaridades de C es su riqueza de operadores. Puede decirse que prácticamente dispone de un operador para cada una de las posibles operaciones en código máquina.

Hay toda una serie de operaciones que pueden hacerse con el lenguaje C, que realmente no están incluidas en el compilador propiamente dicho, sino que las realiza un *preprocesador* justo antes de la compilación. Las dos más importantes son **#define** (directriz de sustitución simbólica o de definición) e **#include** (directriz de inclusión en el fichero fuente).

Finalmente, C, que ha sido pensado para ser altamente transportable y para programar lo improgramable, igual que otros lenguajes tiene sus inconvenientes. Carece de instrucciones de entrada/salida, de instrucciones para manejo de cadenas de caracteres, entre otras, con lo que este trabajo queda para la biblioteca de funciones, con la consiguiente pérdida de transportabilidad.

Por otra parte, la excesiva libertad en la escritura de los programas puede llevar a errores en la programación que, por ser correctos sintácticamente no se detectan a simple vista. Por otra parte, las precedencias de los operadores convierten a veces las expresiones en pequeños rompecabezas. A pesar de todo, C ha demostrado ser un lenguaje extremadamente eficaz y expresivo.

Lenguaje C++

C++ fue desarrollado a partir del lenguaje de programación C y, con pocas excepciones, incluye a C. Esta parte de C incluida en C++ es conocida como C–, y puede compilarse como C++ sin problemas.

En 1980 se añaden al lenguaje C características como *clases* (concepto tomado de Simula67), comprobación del tipo de los argumentos de una función y conversión, si es necesaria, de los mismos, así como otras características; el resultado fue el lenguaje denominado *C con Clases*.

En 1983/84, *C con Clases* fue rediseñado, extendido y nuevamente implementado. El resultado se denominó *Lenguaje C++*. Las extensiones principales fueron *funciones virtuales*, *funciones sobrecargadas* (un mismo identificador puede representar distintas funciones), y *operadores sobrecargados* (un mismo operador puede utilizarse en distintos contextos y con distintos significados). Después de algún otro refinamiento más, C++ quedó disponible en 1985. Este lenguaje fue creado por *Bjarne Stroustrup* (AT&T Bell Laboratories) y documentado en varios libros suyos.

El nombre de C++ se debe a *Rick Mascitti*, significando *el carácter evolutivo de las transformaciones de C* ("++" es el operador de incremento de C).

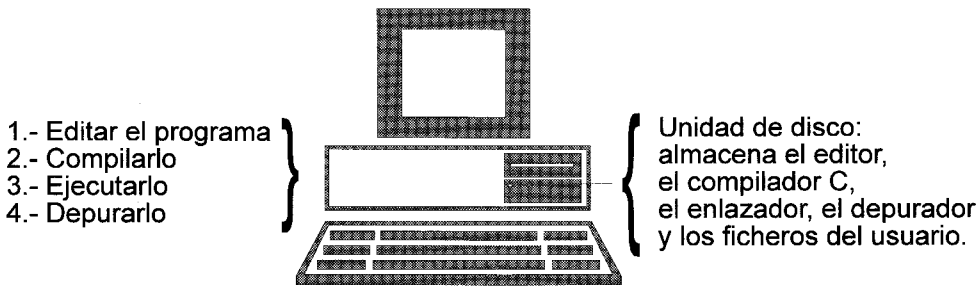
Posteriormente, C++ ha sido ampliamente revisado y refinado, lo que ha dado lugar a añadir nuevas características, como herencia múltiple, funciones miembro **static** y **const**, miembros **protected**, plantillas referidas a tipos y manipulación de excepciones. Se han revisado características como sobrecarga, enlace y manejo de la memoria. Además de esto, también se han hecho pequeños cambios para incrementar la compatibilidad con C.

C++ es, por lo tanto, un lenguaje híbrido, que, por una parte, ha adoptado todas las características de la OOP que no perjudiquen su efectividad; por ejemplo, funciones virtuales y la ligadura dinámica (*dynamic binding*), y por otra parte, mejora sustancialmente las capacidades de C. Esto dota a C++ de una potencia, eficacia y flexibilidad que lo convierten en un estándar dentro de los lenguajes de programación orientados a objetos.

En este libro no abordaremos las nuevas aportaciones de C++ encaminadas a una programación orientada a objetos, sino que nos limitaremos a realizar programas estructurados utilizando lo que hemos denominado C– o simplemente C. Cuando haya aprendido a programar con C, puede dar un paso más e introducirse en la programación orientada a objetos para lo cual le recomiendo mi libro titulado *Programación orientada a objetos con C++* publicado por la editorial RA-MA. Una vez que sepa programación orientada a objetos podrá introducirse en la programación visual y desarrollar aplicaciones gráficas y vistosas estilo Windows. Para esta última fase le recomiendo mi otro libro *Visual C++, aplicaciones para Windows*, publicado también por la editorial RA-MA.

REALIZACIÓN DE UN PROGRAMA EN C

En este apartado se van a exponer los pasos a seguir en la realización de un programa, por medio de un ejemplo. La siguiente figura, muestra lo que un usuario de C debe hacer para desarrollar un programa.



Según lo que hemos visto, un ordenador sólo puede ejecutar programas escritos en lenguaje máquina. Por lo tanto, es necesario disponer de herramientas que permitan la traducción de un programa escrito en un lenguaje de alto nivel, en nuestro caso en C, a lenguaje máquina.

Por lo tanto, en la unidad de disco de nuestro sistema tienen que estar almacenadas las herramientas necesarias para editar, compilar, y depurar nuestro programa. Por ejemplo, supongamos que queremos escribir un programa denominado *saludo.c*. Las herramientas (programas) que tenemos que utilizar y los ficheros que producen son:

Programa	**produce el fichero**
Editor	*saludo.c*
Compilador C	*saludo.obj* en MS-DOS o *saludo.o* en UNIX
Enlazador	*saludo.exe* en MS-DOS o *a.out* por defecto en UNIX
Depurador	ejecuta paso a paso el programa ejecutable

La tabla anterior indica que una vez editado el fichero fuente *saludo.c*, se compila obteniéndose el fichero objeto *saludo.obj* o *saludo.o*, el cual es enlazado con las rutinas necesarias de la biblioteca de C dando lugar a un único fichero ejecutable *saludo.exe* o *a.out*.

Edición de un programa

Para *editar* un programa, primeramente llamaremos, para su ejecución, al programa editor o procesador de textos que vayamos a utilizar. Podemos utilizar el procesador de textos suministrado con el compilador o nuestro propio procesador. El nombre del fichero elegido para guardar el programa en el disco, debe tener como extensión *.c*.

El paso siguiente, es escribir el texto correspondiente al programa fuente. Cada *sentencia* del lenguaje C finaliza con un *punto y coma* y cada *línea del programa* la finalizamos pulsando la tecla *Entrar* (*Enter* o ↵).

Como ejercicio para practicar lo hasta ahora expuesto, escribir el siguiente ejemplo:

```
/*************************** Saludo ***************************/
// saludo.c
#include <stdio.h>

main()
{
  printf("Hola, qué tal estáis.\n");
}
```

¿Qué hace este programa?

Comentamos brevemente cada línea de este programa. No apurarse si algunos de los términos no quedan muy claros ya que todos ellos se verán con detalle en capítulos posteriores.

Las dos primeras líneas son simplemente comentarios. Los comentarios no son tenidos en cuenta por el compilador.

La tercera línea incluye el fichero de cabecera *stdio.h* que contiene las declaraciones necesarias para las funciones de entrada-salida (E/S) que aparecen en el programa; en nuestro caso para **printf**. Esto significa que, como regla general, antes de invocar a una función hay que declararla. Las palabras reservadas de C que empiezan con el símbolo # reciben el nombre de *directrices* del compilador y son procesadas por el *preprocesador* de C cuando se invoca al compilador, pero antes de iniciarse la compilación.

A continuación se escribe la función principal **main**. Observe que una función se distingue por el modificador () que aparece después de su nombre y que el cuerpo de la misma empieza con el carácter { y finaliza con el carácter }.

La función **printf**, es una función de la biblioteca de C que escribe sobre el monitor la expresión que aparece especificada entre comillas. La secuencia de escape \n que aparece a continuación de la cadena de caracteres indica al ordenador que después de escribir el mensaje, avance el cursor de la pantalla al principio de la línea siguiente. Observe que la sentencia finaliza con punto y coma.

Guardar el programa escrito en el disco

El programa editado está ahora en la memoria. Para que este trabajo pueda tener continuidad, el programa escrito se debe grabar en el disco utilizando la orden correspondiente del editor.

Compilar y ejecutar el programa

El siguiente paso es *compilar* el programa; esto es, traducir el programa fuente a lenguaje máquina para posteriormente enlazarlo con las funciones necesarias de la biblioteca de C y obtener así un programa ejecutable. La orden correspondiente para compilar y el enlazar el programa *saludo.c* es la siguiente:

```
cl saludo.c
```

La orden **cl** de MS-DOS invoca al compilador C y al enlazador para producir el fichero ejecutable *saludo.exe*.

```
cc saludo.c -o saludo
```

La orden **cc** de UNIX invoca al compilador C y al enlazador para producir el fichero ejecutable *saludo*. Si no hubiéramos añadido la opción *-o saludo*, el fichero ejecutable se denominaría, por defecto, *a.out*.

Al compilar un programa, se pueden presentar *errores de compilación*, debidos a que el programa escrito no se adapta a la sintaxis y reglas del compilador. Estos errores se irán corrigiendo hasta obtener una compilación sin errores.

Para ejecutar el fichero resultante, escriba el nombre de dicho fichero a continuación del símbolo del sistema, en nuestro caso *saludo*, y pulse *Entrar* (↵). El resultado es que se visualizará sobre la pantalla el mensaje:

```
Hola, qué tal estáis.
```

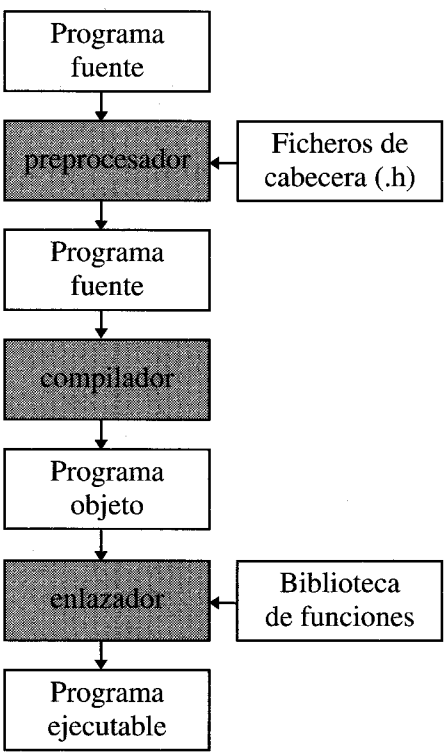
Biblioteca de funciones

Como ya dijimos anteriormente, C carece de instrucciones de E/S, de instrucciones para manejo de cadenas de caracteres, etc. con lo que este trabajo queda para la biblioteca de funciones provista con el compilador. Una función es un conjunto de instrucciones que realizan una tarea específica. Una biblioteca es un fichero separado en el disco (generalmente con extensión *.lib* en MS-DOS o con extensión *.a* en UNIX) que contiene las funciones que realizan las tareas más comunes, para que nosotros no tengamos que escribirlas. Como ejemplo, hemos visto anteriormente la función **printf**. Si esta función no existiera, sería labor nuestra el escribir el código necesario para visualizar los resultados sobre la pantalla.

Para utilizar una función de la biblioteca simplemente hay que invocarla utilizando su nombre y pasar los argumentos necesarios entre paréntesis. Por ejemplo:

```
printf("Hola, qué tal estais.\n");
```

La figura siguiente muestra como el código correspondiente a las funciones de biblioteca invocadas en nuestro programa es añadido por el *enlazador* cuando se está creando el programa ejecutable.



Guardar el programa ejecutable en el disco

Como hemos visto, cada vez que se realiza el proceso de *compilación* y *enlace* del programa actual, C genera automáticamente sobre el disco un fichero ejecutable. Este fichero puede ser ejecutado directamente desde el sistema operativo, sin el soporte de C, escribiendo el nombre del fichero a continuación del símbolo del sistema (*prompt* del sistema) y pulsando *Entrar*.

Cuando se crea un fichero ejecutable, primero se utiliza el compilador C para compilar el programa fuente, dando lugar a un fichero intermedio conocido como fichero objeto (con extensión *.obj* en MS-DOS o *.o* en UNIX). A continuación se utiliza el programa *enlazador* (*link*) para unir en un único fichero ejecutable, el módulo o los módulos del programa compilados separadamente y las funciones de la biblioteca del compilador C que el programa utilice.

Al ejecutar el programa, pueden ocurrir *errores durante la ejecución*. Por ejemplo, puede darse una división por cero. Estos errores solamente pueden ser detectados por C cuando se ejecuta el programa y serán notificados con el correspondiente mensaje de error.

Hay *otro tipo de errores* que no dan lugar a mensaje alguno. Por ejemplo: un programa que no termine nunca de ejecutarse, debido a que presenta un lazo, donde no se llega a dar la condición de terminación. Para detener la ejecución se tienen que pulsar las teclas *Ctrl+C*.

Depurar un programa

Una vez ejecutado el programa, la solución puede ser incorrecta. Este caso exige un análisis minucioso de cómo se comporta el programa a lo largo de su ejecución; esto es, hay que entrar en la fase de *depuración* del programa.

La forma más sencilla y eficaz para realizar este proceso, es utilizar un programa *depurador*. En el apéndice A, se explica como utilizar el depurador *Code View* de Microsoft con un programa MS-DOS y el depurador *gdb* con un programa UNIX.

UN EJEMPLO SIMPLE

Vamos a preparar un programa formado por un solo módulo fuente, con el fin de depurarlo. El primer paso será editar el programa. Como ejemplo, escriba el siguiente programa que imprime la suma, la diferencia, el producto y el cociente de

dos números. Este ejemplo no le costará entenderlo puesto que las sentencias tienen la misma apariencia que en la vida ordinaria.

Edición

Ejecute el procesador de textos y edite el programa ejemplo que se muestra a continuación.

```
/****************** Operaciones aritméticas ******************/
// aritmeti.c

#include <stdio.h>

main()
{
  int dato1, dato2, resultado;

  dato1 = 20;
  dato2 = 10;

  // Suma
  resultado = dato1 + dato2;
  printf("%d + %d = %d\n", dato1, dato2, resultado);

  // Resta
  resultado = dato1 - dato2;
  printf("%d - %d = %d\n", dato1, dato2, resultado);

  // Producto
  resultado = dato1 * dato2;
  printf("%d * %d = %d\n", dato1, dato2, resultado);

  // Cociente
  resultado = dato1 / dato2;
  printf("%d / %d = %d\n", dato1, dato2, resultado);

  return 0;
}
```

Una vez editado el programa, guárdelo en el disco con el nombre *aritmeti.c*.

¿Qué hace este programa?

Fijándonos en la función principal, **main**, vemos que se han declarado tres variables enteras: *dato1*, *dato2* y *resultado*.

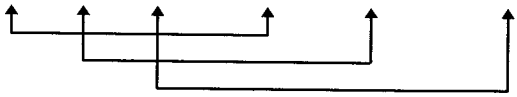
```
int dato1, dato2, resultado;
```

El siguiente paso asigna el valor 20 a la variable *dato1* y el valor 10 a la variable *dato2*.

```
dato1 = 20;
dato2 = 10;
```

A continuación se realiza la suma de esos valores y se escriben los datos y el resultado.

```
resultado = dato1 + dato2;
printf("%d + %d = %d\n", dato1, dato2, resultado);
```



La expresión que aparece entre comillas en la función **printf** indica los caracteres que queremos visualizar sobre el monitor, con una particularidad, que un carácter % le indica al compilador que el siguiente carácter (una *d* indica valor entero en base 10) no es un carácter normal que tiene que visualizar, sino que se trata de un especificador de formato que será sustituido por el valor correspondiente de la lista de valores especificada a continuación. Según esto, la función **printf** anterior producirá el siguiente resultado:

```
20 + 10 = 30
```

Un proceso similar se sigue para calcular la diferencia, el producto y el cociente.

Como verá en un capítulo posterior, tal cual se ha definido la función **main**, C interpreta que por defecto tiene que devolver un entero, razón por la que aparece la sentencia:

```
return 0;
```

Si la sentencia anterior no se especifica, el compilador nos mostrará un aviso (*warning*) indicándolo. Otra solución es indicar al compilador C que la función **main** no va a devolver ningún valor. Esto se hace así:

```
void main()
{
  ...;
}
```

Compilación

Como siguiente paso, compilaremos el programa con las opciones necesarias para posteriormente poder depurarlo. La orden correspondiente para compilar y enlazar el programa *aritmeti.c* de forma que incluya la información necesaria para realizar la depuración, es la siguiente:

```
cl /Zi /Od aritmeti.c
```

La orden **cl** de MS-DOS invoca al compilador C y al enlazador para producir el fichero ejecutable *aritmeti.exe*. La opción */Zi* hace que se incluya en el fichero ejecutable resultante, la información necesaria para realizar la depuración y la opción */Od* impide la optimización, la cual puede dificultar la depuración.

```
cc -g aritmeti.c -o aritmeti
```

La orden **cc** de UNIX invoca al compilador C y al enlazador para producir el fichero ejecutable *aritmeti*. La opción *-g* hace que se incluya en el fichero ejecutable resultante, la información necesaria para realizar la depuración.

Depuración

Cuando finaliza el proceso de compilación y enlace, invocamos al depurador (*debugger*).

```
cv aritmeti
```

La orden **cv** de MS-DOS invoca al depurador *Code View* de Microsoft.

```
gdb aritmeti
```

El programa **gdb** de UNIX es un depurador para programas C escritos bajo UNIX. Otros depuradores de UNIX son **sdb** (depurador de UNIX System V) o **dbx** (depurador del UNIX de SUN Microsystem).

Las operaciones mínimas que debe incluir un depurador son las siguientes:

Rastrear

Permite ver la sentencia del programa que es ejecutada. *Code View* incluye las siguientes opciones:

- Ejecutar una sentencia cada vez, incluidas funciones definidas por el usuario. Esta modalidad se activa y se continúa, pulsando la tecla *F8*. Cuando la sentencia a ejecutar coincide con una llamada a una función definida por el usuario y no queremos que ésta se ejecute paso a paso, utilizaremos la tecla *F10* en vez de *F8*.

- Si pulsamos la tecla *F5*, la ejecución continúa hasta el final del programa o hasta el primer punto de parada, si éste existe.

El depurador *gdb* utiliza la orden *next* (abreviadamente *n*) para ejecutar la sentencia siguiente. La orden *run* inicia la ejecución y la orden *c* la continúa, por ejemplo, después de un punto de parada.

Punto de parada

Un punto de parada (*breakpoint*) es una pausa que se hace en un lugar determinado dentro del programa. Esto permite verificar los valores de las variables en ese instante. Colocar los puntos de parada donde se sospeche que está el error.

En *Code View*, para poner o quitar una pausa, se coloca el cursor en el lugar donde va a tener lugar la pausa y se pulsa *F9*.

El depurador *gdb* utiliza la orden *break* (abreviadamente *b*) para establecer un punto de parada.

La orden *break* [*fichero:*]*función* establece un punto de parada en la función especificada y la orden *break* [*fichero:*]*n_línea* establece un punto de parada en la línea especificada por el número *n_línea*.

Expresiones de seguimiento

Las expresiones de seguimiento (*watch*) permiten observar los valores de las variables o de expresiones del programa mientras este se ejecuta. *Code View* utiliza la orden *Add Watch* para especificar las variables o expresiones cuyos valores se desean ver.

El depurador *gdb* utiliza la orden *print expresión* (abreviadamente *p*) para visualizar el valor de una variable o de una expresión.

EJERCICIOS PROPUESTOS

Practique la edición, la compilación y la depuración con el programa *aritmeti.c* o con un programa similar.

CAPÍTULO 2

ELEMENTOS DEL LENGUAJE C

En este capítulo veremos los elementos que aporta C (caracteres, secuencias de escape, tipos de datos, operadores, etc.) para escribir un programa. El introducir este capítulo ahora es porque dichos elementos los tenemos que utilizar desde el principio; algunos ya han aparecido en los ejemplos del capítulo 1. Considere este capítulo como soporte para el resto de los capítulos; esto es, lo que se va a exponer en él, lo irá utilizando en menor o mayor medida en los capítulos sucesivos. Por lo tanto, limítese ahora simplemente a realizar un estudio para saber de forma genérica los elementos con los que contamos para desarrollar nuestros programas.

PRESENTACIÓN DE LA SINTAXIS DE C

Las palabras clave aparecerán en negrita y cuando se utilicen deben escribirse exactamente como aparecen. Por ejemplo,

```
char a;
```

El texto que no aparece en negrita, significa que ahí debe ponerse la información indicada por ese texto. Por ejemplo,

```
typedef declaración_tipo sinónimo[, sinónimo]...;
```

Una información encerrada entre corchetes "[]" es opcional. Los puntos suspensivos "..." indican que pueden aparecer más elementos de la misma forma.

Cuando dos o más opciones aparecen entre llaves "{}" separadas por "|", se elige una, la necesaria dentro de la sentencia. Por ejemplo,

```
constante_entera[{L|U|UL}]
```

CARACTERES DE C

Los caracteres de C pueden agruparse en letras, dígitos, espacios en blanco, caracteres especiales, signos de puntuación y secuencias de escape.

Letras, dígitos y carácter de subrayado

Estos caracteres son utilizados para formar las *constantes*, los *identificadores* y las *palabras clave* de C. Son los siguientes:

- Letras mayúsculas del alfabeto inglés:

  A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

- Letras minúsculas del alfabeto inglés:

  a b c d e f g h i j k l m n o p q r s t u v w x y z

- Dígitos decimales:

  0 1 2 3 4 5 6 7 8 9

- Carácter de subrayado "_"

El compilador C trata las letras mayúsculas y minúsculas como caracteres diferentes. Por ejemplo los identificadores *Pi* y *PI* son diferentes.

Espacios en blanco

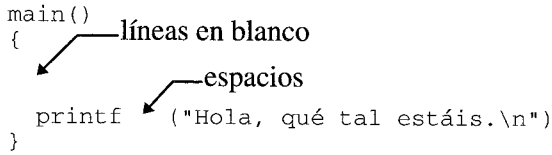
Los caracteres espacio en blanco, tabulador horizontal, tabulador vertical, avance de página y nueva línea, son caracteres denominados *espacios en blanco*, porque la labor que desempeñan es la misma que la del espacio en blanco: actuar como separadores entre los elementos de un programa, lo cual permite escribir programas más legibles. Por ejemplo, el siguiente código:

```
main() { printf("Hola, qué tal estáis.\n"); }
```

puede escribirse de una forma más legible así:

```
main()
{
  printf("Hola, qué tal estáis.\n");
}
```

Los espacios en blanco en exceso son ignorados por el compilador. Por ejemplo, el código siguiente se comporta exactamente igual que el anterior:



El carácter *Ctrl+Z* en MS-DOS o *Ctrl+D* en UNIX, es tratado por el compilador como un indicador de fin de fichero (*End Of File*).

Caracteres especiales y signos de puntuación

Este grupo de caracteres se utiliza de diferentes formas; por ejemplo, para indicar que un identificador es una función o un array; para especificar una determinada operación aritmética, lógica o de relación; etc. Son los siguientes:

```
, . ; : ? ' " ( ) [ ] { } < ! | / \ ~ + # % & ^ * - = >
```

Secuencias de escape

Cualquier carácter de los anteriores puede también ser representado por una *secuencia de escape*. Una secuencia de escape está formada por el carácter \ seguido de una *letra* o de una *combinación de dígitos*. Son utilizadas para acciones como nueva línea, tabular y para hacer referencia a caracteres no imprimibles.

El lenguaje C tiene predefinidas las siguientes secuencias de escape:

Secuencia	Nombre
\n	Ir al principio de la siguiente línea
\t	Tabulador horizontal
\v	Tabulador vertical (sólo para impresora)
\b	Retroceso (*backspace*)
\r	Retorno de carro sin avance de línea
\f	Alimentación de página (sólo para impresora)
\a	Alerta, pitido
\'	Comilla simple
\"	Comilla doble
\\	Barra invertida (*backslash*)
\ddd	Carácter ASCII. Representación octal
\xdd	Carácter ASCII. Representación hexadecimal

Observe en la llamada a **printf** del ejemplo anterior la secuencia de escape \n.

TIPOS DE DATOS

Recuerde las operaciones aritméticas que realizaba el programa *aritmeti.c* que vimos en el capítulo anterior. Por ejemplo, una de las operaciones que realizábamos era la suma de dos valores:

```
dato1 = 20;
dato2 = 10;
resultado = dato1 + dato2;
```

Para que el compilador C reconozca esta operación es necesario especificar previamente el tipo de cada uno de los operandos que intervienen en la misma, así como el tipo del resultado. Para ello, escribiremos una línea como la siguiente:

```
int dato1, dato2, resultado;
```

La declaración anterior le dice al compilador C que *dato1*, *dato2* y *resultado* son de tipo entero (**int**).

Hay dos clases de tipos: tipos *fundamentales* y tipos *derivados*.

Tipos fundamentales

Hay varios tipos fundamentales de datos. Los ficheros de cabecera *limits.h* y *float.h* especifican los valores máximo y mínimo para cada tipo. Los podemos clasificar en:

Tipos enteros:	**char**, **short**, **int**, **long** y **enum**
Tipos reales:	**float**, **double** y **long double**
Otros:	**void**

Cada tipo entero puede ser calificado por las palabras clave **signed** o **unsigned**, lo que da lugar a los siguientes tipos extras:

```
signed char,   unsigned char
signed short,  unsigned short
signed int,    unsigned int
signed long,   unsigned long
```

Un entero calificado **signed** es un entero con signo; esto es, un valor entero positivo o negativo. Un entero calificado **unsigned** es un valor entero sin signo, el cual es manipulado como un valor entero positivo.

Si los calificadores **signed** y **unsigned** se utilizan sin un tipo específico, se asume el tipo **int**. Por este motivo, las siguientes declaraciones de *x* y de *y* son equivalentes:

```
signed x;        /* es equivalente a */
signed int x;

unsigned y;      /* es equivalente a */
unsigned int y;
```

char (carácter - 1 byte)

El tipo **char** es utilizado para almacenar un valor entero en el rango –128 a 127. Los valores 0 a 127 son equivalentes a un carácter del código ASCII. El tipo **char** es la abreviación de **signed char**.

De forma análoga el tipo **unsigned char** puede almacenar valores en el rango de 0 a 255, valores correspondientes a los números ordinales de los 256 caracteres ASCII.

El siguiente ejemplo declara una variable *car* de tipo **char**. Cuando trabajemos con esta variable, deberemos tener en cuenta que sólo puede contener valores enteros entre –128 y 127.

```
char car;
```

A continuación se declara la variable *a* de tipo **char** a la que se le asigna el carácter *'z'* como valor inicial (observe que hay una diferencia entre *'z'* y *z*; *z* entre comillas simples es interpretada por el compilador C como un valor, un carácter, y *z* sin comillas sería interpretada como una variable). Así mismo, se declara la variable *b* de tipo **signed char** con un valor inicial de *7* expresado en hexadecimal (*0x07*) y la variable *c* de tipo **unsigned char** con un valor inicial de *32*.

```
char a = 'z';
signed char b = 0x07;
unsigned char c = 32;
```

Las definiciones anteriores son equivalentes a las siguientes:

```
char a = 122;           /* la z es el ASCII 122 */
signed char b = 7;      /* 7 en base 16 (0x07) es 7 en base 10 */
unsigned char c = ' '; /* el espacio en blanco es el ASCII 32 */
```

La razón es que un carácter es representado internamente por un entero, que puede ser expresado en decimal, hexadecimal u octal (vea en los apéndices del libro la tabla de caracteres ASCII).

short (entero formato corto - 2 bytes)

El tipo **short**, abreviación de **signed short int**, proporciona un entero en el rango de valores:

$$-32768 \text{ a } 32767 \ (-2^{15} \text{ a } 2^{15}-1)$$

De forma similar el tipo **unsigned short** puede almacenar valores en el rango 0 a 65535 (0 a $2^{16}-1$).

El siguiente ejemplo declara *i* y *j*, como variables enteras que pueden tomar valores entre –32768 y 32767.

```
short i, j;
```

Otros ejemplos son:

```
short int a = -500;
signed short b = 1990;
unsigned short int c = 0xf000;
```

int (entero - 2 o 4 bytes)

Un **int**, abreviación de **signed int**, es para C un número sin punto decimal. El tamaño en bytes depende de la arquitectura de la máquina. Igualmente ocurre con el tipo **unsigned int**. Por ejemplo, para una máquina con un procesador de 16 bits el rango de valores es de:

$$-32768 \text{ a } 32767 \ (-2^{15} \text{ a } 2^{15}-1) \text{ para el tipo } \textbf{int}$$
$$0 \text{ a } 65535 \ (\quad 0 \text{ a } 2^{16}-1) \text{ para el tipo } \textbf{unsigned}$$

El uso de enteros produce un código compacto y rápido. Para una máquina de 16 bits este tipo es equivalente al tipo **short** y solamente ocupa **2 bytes** de memoria.

En general, podemos afirmar que:

$$\text{tamaño}(\textbf{short}) \leq \text{tamaño}(\textbf{int})$$

El siguiente ejemplo declara las variables *n* y *x* de tipo entero.

```
int n, x;
```

Otros ejemplos son:

```
int a = 2000;
signed int b = -30;
unsigned int c = 0xf003;
unsigned d;
```

long (entero formato largo - 4 u 8 bytes)

El tipo **long**, abreviación de **signed long int**, es idóneo para aplicaciones de gestión. Al igual que los tipos anteriores, son números sin punto decimal. Para el caso de que tengan cuatro bytes de longitud, el rango de valores es el siguiente:

–2147483648 a 2147483647 (-2^{31} a $2^{31}-1$) para el tipo **long**
0 a 4294967295 (0 a $2^{32}-1$) para el tipo **unsigned long**

En general, podemos afirmar que:

tamaño(**int**) ≤ tamaño(**long**)

El siguiente ejemplo declara las variables *n* y *m* de tipo entero, pudiendo tomar valores entre –2147483648 y 2147483647.

```
long n, m;
```

Otros ejemplos son:

```
long a = -1L; /* L indica que la constante -1 es long */
signed long b = 125;
unsigned long int c = 0x1f00230f;
```

enum

La declaración de un *tipo enumerado* es simplemente una lista de valores que pueden ser tomados por una variable de ese tipo. Los valores de un tipo enumerado se representarán con identificadores, que serán las constantes del nuevo tipo. Por ejemplo,

```
enum dia_semana
            {
              lunes,
              martes,
              miercoles,
              jueves,
              viernes,
              sabado,
              domingo
            } hoy;

enum dia_semana ayer;
```

Este ejemplo declara las variables *hoy* y *ayer* del tipo enumerado *dia_semana*. Estas variables pueden tomar cualquier valor de los especificados, *lunes* a *domingo*. Los valores de las constantes comienzan en cero y aumentan en uno según se lee la declaración de arriba a abajo o de izquierda a derecha. Según esto el valor de *lunes* es *0*, el valor de *martes* es *1*, etc.

Creación de una enumeración

Crear una enumeración es definir un nuevo tipo de datos, denominado *tipo enumerado* y declarar una variable de este tipo. La sintaxis es la siguiente:

```
enum tipo_enumerado
  {
    /* definición de nombres de constantes enteras */
  };
```

donde *tipo_enumerado* es un identificador que nombra el nuevo tipo definido.

Después de definir un tipo enumerado, podemos declarar una o más variables de ese tipo, de la forma:

```
enum tipo_enumerado [variable[, variable]...];
```

El siguiente ejemplo declara una variable llamada *color* del tipo enumerado *colores*, la cual puede tomar cualquier valor de los especificados en la lista.

```
enum colores
  {
    azul, amarillo, rojo, verde, blanco, negro
  };
enum colores color;

color = azul;
```

Como ya hemos dicho, cada identificador de la lista de constantes en una enumeración, tiene asociado un valor. Por defecto, el primer identificador tiene asociado el valor *0*, el siguiente el valor *1*, y así sucesivamente. Según esto,

`color = verde;` es equivalente a `color = 3;`

Nota: Para ANSI C un tipo enumerado es un tipo **int**. Sin embargo, para C++ un tipo enumerado es un nuevo tipo entero diferente de los anteriores. Esto significa que en C++ un valor de tipo **int** no puede ser asignado directamente a un tipo enumerado, sino que habría que hacer una conversión explícita de tipo (vea "Conversión explícita del tipo de una expresión" al final de este capítulo).

```
color = (colores)3; /* conversión explícita al tipo colores */
```

A cualquier identificador de la lista se le puede asignar un valor inicial entero por medio de una expresión constante. Los identificadores sucesivos tomarán valores correlativos a partir de éste. Por ejemplo,

```
enum colores
  {
    azul, amarillo, rojo, verde = 0, blanco, negro
  } color;
```

Este ejemplo define un tipo enumerado llamado *colores* y declara una variable *color* de ese tipo. Los valores asociados a los identificadores son los siguientes: *azul* = 0, *amarillo* = 1 *rojo* = 2, *verde* = 0, *blanco* = 1 y *negro* = 2.

A los miembros de una enumeración se les aplica las siguientes reglas:

- Dos o más miembros pueden tener un mismo valor.
- Un identificador no puede aparecer en más de un tipo.
- Desafortunadamente, no es posible leer o escribir directamente un valor de un tipo enumerado. El siguiente ejemplo aclara este detalle.

```
// enum.c
#include <stdio.h>
enum colores
  {
    azul, amarillo, rojo, verde, blanco, negro
  };

main()
{
  enum colores color;
  // Leer un color introducido desde el teclado
  scanf("%d", &color);
  // Visualizar un color
  printf("%d\n", color);
}
```

En el siguiente capítulo verá con detalle la función **scanf**; ahora limítese a saber que esta función le permite asignar un valor introducido por el teclado, a la variable especificada.

Anteriormente hemos indicado que no es posible, por ejemplo, asignar a la variable *color* directamente el valor *verde*, sino que hay que asignarle la constante entera *3* equivalente. Igualmente, **printf** no escribirá *verde*, sino que escribirá *3*. Según esto, se preguntará ¿qué aportan, entonces, los tipos enumerados? Los tipos enumerados ayudan a acercar más el lenguaje de alto nivel a nuestra forma de expresarnos. Como podrá ver más adelante, la expresión "si el color es verde, ..." dice más que la expresión "si el color es 3, ...".

float (reales de precisión simple - 4 bytes)

Los datos reales de precisión simple son los más recurridos en un lenguaje de programación. Un valor real de precisión simple es un número que puede tener un punto decimal y que puede estar comprendido en el rango de:

−3.402823E+38 a −1.175494E−38 para números negativos
1.175494E−38 a 3.402823E+38 para números positivos

Un número real de precisión simple no tiene más de 7 dígitos significativos.

El siguiente ejemplo declara la variable *x* de tipo real de precisión simple.

```
float x;
```

Otros ejemplos son:

```
float a = 3.14159;
float b = 2.2e-5; /* 2.2e-5 = 2.2 x 10^-5 */
```

double (reales de precisión doble - 8 bytes)

Un dato real de precisión doble es un valor que puede tener un punto decimal y puede estar comprendido en el rango:

−1.79769E+308 a −2.22507E−308 para números negativos
2.22507E−308 a 1.79769E+308 para números positivos

Un valor real de precisión doble puede tener hasta 16 dígitos significativos, lo que da lugar a cálculos más exactos.

El siguiente ejemplo declara la variable *x* de tipo real de precisión doble.

```
double x;
```

Otros ejemplos son:

```
double a = 3.1415926;
double b = 2.2e-8;
```

long double (reales de precisión doble formato largo - 10 bytes)

Los valores para este tipo están comprendidos en el rango de:

−1.189731E+4932 a −3.362103E−4932 para números negativos
3.362103E−4932 a 1.189731E+4932 para números positivos

Un número real de precisión doble formato largo puede tener hasta 19 dígitos significativos. Algunos ejemplos son:

```
long double x;
long double y = 3.17e+425;
```

void

El tipo **void** especifica un conjunto vacío de valores. En realidad **void** no es un tipo, aunque por la forma de utilizarlo si lo comparamos con la forma de utilizar los otros tipos fundamentales, se considera como tal. Por esta razón, no se puede declarar una variable de tipo **void**.

```
void a; /* error: no se puede declarar una variable de tipo void */
```

El tipo **void** se utiliza:

- Para indicar que una función no acepta argumentos. En el siguiente ejemplo, **void** indica que la función *fx* no tiene argumentos.

  ```
  double fx(void);
  ```

- Para declarar funciones que no retornan un valor. En el siguiente ejemplo, **void** indica que la función *fy* no retorna un valor.

  ```
  void fy(int, int);
  ```

- Para declarar un puntero genérico, como veremos más adelante; esto es, un puntero a un objeto de tipo aún desconocido.

  ```
  void *p;
  ```

Los ejemplos anteriores declaran la función *fx*, como una función sin argumentos que devuelve un valor de tipo real de doble precisión; la función *fy*, como una función con dos argumentos de tipo **int** que no devuelve valor alguno; y un puntero genérico *p*.

Tipos derivados

Los tipos derivados son construidos a partir de los tipos fundamentales. Algunos de ellos son: *punteros*, *estructuras*, *uniones*, *arrays* y *funciones*. Cada uno de estos tipos será estudiado con detalle en capítulos posteriores.

NOMBRES DE TIPO

Utilizando la declaración **typedef** podemos declarar nuevos nombres de tipo de datos; esto es, sinónimos de otros tipos ya sean fundamentales o derivados, los cuales pueden ser utilizados más tarde para declarar variables de esos tipos. La sintaxis de **typedef** es la siguiente

```
typedef declaración_tipo sinónimo[, sinónimo]...;
```

donde *declaración_tipo* es cualquier tipo definido en C, fundamental o derivado, y *sinónimo* es el nuevo nombre elegido para el tipo especificado.

Por ejemplo, la sentencia siguiente declara el nuevo tipo *ulong* como sinónimo del tipo fundamental **unsigned long**.

```
typedef unsigned long ulong;
```

De acuerdo con esta declaración,

`unsigned long dni;` es equivalente a `ulong dni;`

Las declaraciones **typedef** permiten parametrizar un programa para evitar problemas de portabilidad. Si utilizamos **typedef** con los tipos que pueden depender de la instalación, cuando se lleve el programa a otra instalación sólo se tendrán que cambiar estas declaraciones.

El siguiente ejemplo declara el tipo enumerado *t_colores* y define la variable *color* de este tipo.

```
enum colores
  {
    azul, amarillo, rojo, verde, blanco, negro
  };

typedef enum colores t_colores;

main()
{
  t_colores color;
  // ...
}
```

La declaración del tipo *t_colores* puede realizarse también así:

```
typedef enum colores
  {
    azul, amarillo, rojo, verde, blanco, negro
  } t_colores;
```

o simplemente así:

```
typedef enum
  {
    azul, amarillo, rojo, verde, blanco, negro
  } t_colores;
```

CONSTANTES

Una constante es un valor que, una vez fijado por el compilador, no cambia durante la ejecución del programa. Una constante en C puede ser un *entero*, un *real*, un *carácter* o una *cadena de caracteres*.

Constantes enteras

El lenguaje C permite especificar un entero en base 10, 8 y 16.

En general, si la constante es positiva, el signo + es opcional y si es negativa, lleva el signo –. El tipo de una constante entera depende de su base, de su valor y de su sufijo. La sintaxis para especificar una constante entera es:

```
{[+]|-}constante_entera[{L|U|UL}]
```

Si es decimal y no tiene sufijo, su tipo es el primero de los tipos **int**, **long int** y **unsigned long int** en el que su valor pueda ser representado.

Si es octal o hexadecimal y no tiene sufijo, su tipo es el primero de los tipos **int**, **unsigned int**, **long int** y **unsigned long int** en el que su valor pueda ser representado.

También se puede indicar explícitamente el tipo de una constante entera, añadiendo los sufijos *L*, *U*, o *UL* (mayúsculas o minúsculas).

Si el sufijo es *L*, su tipo es **long** cuando el valor puede ser representado en este tipo, si no es **unsigned long**. Si el sufijo es *U*, su tipo es **unsigned int** cuando el valor puede ser representado en este tipo, si no es **unsigned long**. Si el sufijo es *UL*, su tipo es **unsigned long**. Por ejemplo,

`1522U`	constante entera **unsigned int**
`1000L`	constante entera de tipo **long**
`325UL`	constante entera de tipo **unsigned long**

Una *constante decimal* puede tener uno o más dígitos del *0* a *9*, de los cuales el primero de ellos es distinto de cero. Por ejemplo,

```
4326        constante entera int
432600      constante entera long
```

Una *constante octal* puede tener 1 o más dígitos del *0* a *7*, precedidos por *0* (*cero*). Por ejemplo,

```
0326        constante entera int
```

Una *constante hexadecimal* puede tener 1 o más caracteres del *0* a *9* y de la *A* a la *F* (en mayúsculas o en minúsculas) precedidos por *0x* o *0X* (*cero* más *x*). Por ejemplo,

```
256         especifica el nº  256 en decimal
0400        especifica el nº  256 en octal
0x100       especifica el nº  256 en hexadecimal
-0400       especifica el nº –256 en octal
-0x100      especifica el nº –256 en hexadecimal
```

Constantes reales

Una constante real está formada por una *parte entera*, seguida por un *punto decimal*, y una *parte fraccionaria*. También se permite la notación científica, en cuyo caso se añade al valor una *e* o *E*, seguida por un exponente positivo o negativo.

```
{[+]|-}parte-entera.parte-fraccionaria[{e|E}{[+]|-}exponente]
```

donde *exponente* representa cero o más dígitos del *0* al *9* y *E* o *e* es el símbolo de exponente de la base 10 que puede ser positivo o negativo ($2E{-}5 = 2 \times 10^{-5}$). Si la constante real es positiva no es necesario especificar el signo y si es negativa lleva el signo menos (–). Por ejemplo,

```
-17.24
17.244283
.008e3
27E-3
```

Una constante real tiene siempre tipo **double**, a no ser que se añada a la misma una *f* o *F*, en cuyo caso será de tipo **float**, o una *l* o *L* para indicar que es de tipo **long double**. Por ejemplo,

```
17.24F      constante real de tipo float
```

Constante de un solo carácter

Las constantes de un solo carácter son de tipo **char**. Este tipo de constantes está formado por un único carácter encerrado entre *comillas simples*. Una secuencia de escape es considerada como un único carácter. Algunos ejemplos son:

`' '`	espacio en blanco
`'x'`	letra minúscula x
`'\n'`	nueva línea
`'\x1B'`	carácter ASCII Esc

El valor de una constante de un solo carácter es el valor que le corresponde en el juego de caracteres de la máquina.

Constante de caracteres

Una constante de caracteres es una cadena de caracteres encerrados entre *comillas dobles*. Por ejemplo,

```
"Esto es una constante de caracteres"
"3.1415926"
"Paseo Pereda 10, Santander"
```

En el ejemplo siguiente el carácter \n fuerza a que la cadena "*O pulse Entrar*" se escriba en una nueva línea.

```
printf("Escriba un número entre 1 y 5\nO pulse Entrar");
```

Cuando una cadena de caracteres es demasiado larga puede utilizarse el carácter "\" como carácter de continuación. Por ejemplo,

```
printf("Esta cadena de caracteres es dema\↵
siado larga.\n");
```

Este ejemplo daría lugar a una sola línea:

```
Esta cadena de caracteres es demasiado larga.
```

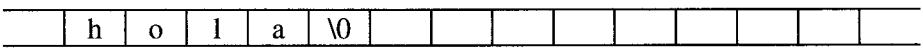
Dos o más cadenas separadas por un espacio en blanco serían concatenadas en una sola cadena. Por ejemplo,

```
printf("Primera cadena, "
       "segunda cadena.\n");
```

Este ejemplo daría lugar a una sola cadena:

```
Primera cadena, segunda cadena.
```

Los caracteres de una cadena de caracteres son almacenados en localizaciones sucesivas de memoria. Cada carácter ocupa un byte. Cada cadena de caracteres es finalizada automáticamente por el carácter nulo representado por la secuencia de escape *\0*. Por ejemplo, la cadena "*hola*" sería representada en memoria así:



IDENTIFICADORES

Los identificadores son nombres dados a constantes, variables, tipos, funciones y etiquetas de un programa. La sintaxis para formar un identificador es la siguiente:

```
{letra|_}[{letra|dígito|_}]...
```

lo cual indica que un identificador consta de uno o más caracteres (letras, dígitos y el carácter de subrayado) y que el *primer carácter* debe ser una *letra* o el *carácter de subrayado.*

Las letras pueden ser mayúsculas o minúsculas. Para C una letra mayúscula es un carácter diferente a esa misma letra en minúscula. Por ejemplo, los identificadores *Suma*, *suma* y *SUMA* son diferentes.

Los identificadores pueden tener cualquier número de caracteres pero dependiendo del compilador que se utilice solamente los *n* caracteres primeros (n≥31) son significativos. Esto quiere decir que un identificador es distinto de otro cuando difieren al menos en uno de los *n* primeros caracteres. Algunos ejemplos son:

```
Suma
suma
Calculo_Numeros_Primos
fn_ordenar
ab123
```

PALABRAS CLAVE

Las palabras clave son identificadores predefinidos que tienen un significado especial para el compilador C. Un identificador definido por el usuario, no puede tener el mismo nombre que una palabra clave. El lenguaje C, tiene las siguientes palabras clave:

```
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`

Las palabras clave deben escribirse siempre en minúsculas, como están.

COMENTARIOS

Un comentario es un mensaje a cualquiera que lea el código fuente. Añadiendo comentarios se hace más fácil la comprensión del programa. La finalidad de los comentarios es explicar el código fuente. Casi todos los compiladores C soportan comentarios estilo C y estilo C++.

Un comentario estilo C empieza con los caracteres **/*** y finaliza con los caracteres ***/**. Estos comentarios pueden ocupar más de una línea, pero no pueden anidarse. Por ejemplo,

```
main()  /* Función principal */

/* Este es un comentario
 * que ocupa varias
 * líneas.
 */
```

Un comentario estilo C++ comienza con los caracteres **//** y termina al final de la línea. Estos comentarios no pueden ocupar más de una línea. Por ejemplo,

```
main()  // Función principal
```

Un comentario estilo C puede aparecer en cualquier lugar donde se permita aparecer un espacio en blanco. El compilador trata un comentario como a un espacio en blanco.

DECLARACIÓN DE CONSTANTES SIMBÓLICAS

Declarar una constante simbólica significa decirle al compilador C el nombre de la constante y su valor. Esto se hace generalmente antes de la función **main** utilizando la directriz **#define**, cuya sintaxis es así:

```
#define NOMBRE VALOR
```

El siguiente ejemplo declara la constante real *PI* con el valor 3.14159, la constante de un solo carácter *NL* con el valor '\n' y la constante de caracteres *MENSAJE* con el valor "Pulse una tecla para continuar\n".

```
#define PI 3.14159
#define NL '\n'
#define MENSAJE "Pulse una tecla para continuar\n"
```

Observe que no hay un punto y coma después de la declaración. Esto es así, porque una directriz no es una sentencia C, sino una orden para el preprocesador.

El tipo de una constante es el tipo del valor asignado. Suele ser habitual escribir el nombre de una constante en mayúsculas.

Constantes C++

C++ y algunos compiladores C admiten una forma adicional de declarar una constante; anteponer el calificador **const** al nombre de la constante. Utilizando el calificador **const** se le dice al compilador C el tipo de la constante, su nombre y su valor. Por ejemplo,

const int *K = 12;*

El ejemplo anterior declara la constante entera *K* con el valor 12.

Una vez que se haya declarado un objeto constante no se le puede asignar un valor. Por ello, al declararlo debe ser inicializado. Por ejemplo, como *K* ha sido declarado como constante, las siguientes sentencias darían lugar a un error:

```
K = 100;     /* error */
K++;         /* error */
```

¿Por qué utilizar constantes?

Utilizando constantes es más fácil modificar un programa. Por ejemplo, supongamos que el programa utiliza una constante de valor *100*, veinte veces. Si hemos declarado una constante *K = 100* y posteriormente necesitamos cambiar el valor de la constante a *120*, tendremos que modificar una sola línea, la que declara la constante. En cambio, si no hemos declarado *K*, sino que hemos utilizado el valor *100* directamente veinte veces, tendríamos que hacer veinte cambios.

DECLARACIÓN DE UNA VARIABLE

El valor de una variable, a diferencia de una constante, puede cambiar a lo largo de la ejecución de un programa. Cada variable de un programa, debe declararse antes de ser utilizada.

La declaración de una variable consiste en enunciar el nombre de la misma y asociarle un tipo. El tipo determina los valores que puede tomar la variable así como las operaciones que con ella pueden realizarse.

La sintaxis correspondiente a la declaración de una variable es la siguiente:

tipo identificador[, *identificador*]...

donde *tipo* especifica el tipo fundamental o derivado de la variable (**char**, **int**, **float**, **double**, ...) e *identificador* indica el nombre de la variable. Por ejemplo,

```
int contador;
main()
{
  int dia, mes, anyo;
  float suma, precio;

  // ...
}
```

El ejemplo anterior declara cuatro variables de tipo **int** y dos variables de tipo **float**. Observe que hay dos lugares donde se puede realizar la declaración de una variable: fuera de todo bloque, entendiendo por bloque un conjunto de sentencias encerradas entre el carácter '{' y el carácter '}', y dentro de un bloque de sentencias, al principio del mismo en ANSI C y en cualquier parte en caso de C++.

En nuestro ejemplo, se ha declarado la variable *contador* antes de la función **main**, fuera de todo bloque, y las variables *dia*, *mes*, *anyo*, *suma* y *precio* dentro del cuerpo de la función, dentro de un bloque de sentencias.

Una variable declarada fuera de todo bloque es por defecto *global* y es accesible en el resto del fichero fuente en el que está declarada. Por el contrario, una variable declarada dentro de un bloque, es por defecto *local* y es accesible solamente dentro de éste. Para comprender esto, piense que generalmente en un programa habrá más de un bloque de sentencias. No obstante, esto lo veremos con más detalle en el capítulo siguiente.

Según lo expuesto, la variable *contador* es global y las variables *dia*, *mes*, *anyo*, *suma* y *precio* son locales.

Inicialización de una variable

Si queremos que algunas o todas las variables que intervienen en un programa tengan un valor inicial cuando éste comience a ejecutarse, tendremos que inicializar dichas variables. Una variable puede ser inicializada, cuando se declara o, si está declarada dentro de un bloque, a continuación de ser declarada. A diferencia de las constantes, este valor puede cambiarse a lo largo de la ejecución del programa. Por ejemplo,

```
int contador = 1;
main()
{
  int dia = 20, mes = 9, anyo = 1995;
  float suma = 0, precio;
  precio = 100;
  // ...
}
```

No hay ninguna razón para no inicializar una variable. Tiene que saber que el compilador C inicializa automáticamente las variables globales a cero, pero no hace lo mismo con las variables locales. Las variables locales no son inicializadas por el compilador por lo que tendrán un valor, para nosotros, indefinido; se dice entonces que contienen basura (un valor que en principio no sirve).

EXPRESIONES NUMÉRICAS

Una expresión es una secuencia de operadores y operandos que especifican una operación determinada. Por ejemplo,

```
a++
suma += c
cantidad * precio
7 * sqrt(a) - b / 2     (sqrt indica raíz cuadrada)
```

OPERADORES

Los operadores son símbolos que indican como son manipulados los datos. Se pueden clasificar en los siguientes grupos: aritméticos, lógicos, relacionales, unitarios, lógicos para manejo de bits, de asignación, operador condicional y otros.

Operadores aritméticos

Los operadores aritméticos los utilizamos para realizar operaciones matemáticas y son los siguientes:

Operador	Operación
+	Suma. Los operandos pueden ser enteros o reales.
–	Resta. Los operandos pueden ser enteros o reales.
*	Multiplicación. Los operandos pueden ser enteros o reales.
/	División. Los operandos pueden ser enteros o reales. Si ambos operandos son enteros el resultado es entero. En el resto de los casos el resultado es real.
%	Módulo o resto de una división entera. Los operandos tienen que ser enteros.

El siguiente ejemplo muestra como utilizar estos operadores. Como ya hemos venido diciendo, observe que primero se declaran las variables y después se realizan las operaciones deseadas con ellas.

```
main()
{
  int a = 10, b = 3, c;
  float x = 2.0, y;

  y = x + a;  /* el resultado es 12.0 de tipo float */
  c = a / b;  /* el resultado es 3 de tipo int */
  c = a % b;  /* el resultado es 1 de tipo int */
  y = a / b;  /* el resultado es 3 de tipo int. Se
                 convierte a float para asignarlo a y */
  c = x / y;  /* el resultado es 0.666667 de tipo float. Se
                 convierte a int para asignarlo a c (c = 0) */
}
```

Cuando en una operación aritmética los operandos son de diferentes tipos, ambos son convertidos al tipo del operando de precisión más alta; por ejemplo, para realizar la suma $x+a$ el valor del entero a es convertido a **float**, tipo de x. No se modifica a, sino que su valor es convertido a **float** sólo para realizar la suma.

El resultado obtenido en una operación aritmética es convertido al tipo de la variable que almacena dicho resultado. Por ejemplo, del resultado de x/y sólo la parte entera es asignada a c, ya que c es de tipo **int**. Esto indica que los reales son convertidos a enteros, truncando la parte fraccionaria.

Un resultado real es redondeado. Observe la operación x/y para x igual a 2 e y igual a 3. El resultado es 0.666667 en lugar de 0.666666 porque la primera cifra decimal suprimida es 6. Cuando la primera cifra decimal suprimida es 5 o mayor de 5, la última cifra decimal conservada se incrementa en una unidad.

Operadores lógicos

El resultado de una operación lógica (AND, OR y NOT) es un valor verdadero o falso (1 o 0). Por definición, un valor distinto de cero es siempre verdadero y un valor cero es siempre falso. Los operadores lógicos de C son los siguientes:

Operador	Operación
&&	AND. Da como resultado el valor lógico 1 si ambos operandos son distintos de cero. Si uno de ellos es cero el resultado es el valor lógico 0. Si el primer operando es igual a cero, el segundo operando no es evaluado.
||	OR. El resultado es 0 si ambos operandos son 0. Si uno de los operandos tiene un valor distinto de 0, el resultado es 1. Si el primer operando es distinto de cero, el segundo operando no es evaluado (el carácter | es el ASCII 124).
!	NOT. El resultado es 0 si el operando tiene un valor distinto de cero, y 1 en caso contrario.

El resultado de una operación lógica es de tipo **int**. Los operandos pueden ser enteros, reales o punteros. Por ejemplo,

```
main()
{
  int p = 10, q = 0, r = 0;
  r = p && q;    /* da como resultado 0 */
  r = p || q;    /* da como resultado 1 */
  r = !p;        /* da como resultado 0 */
}
```

Operadores de relación

El resultado de una operación de relación es un valor verdadero o falso (1 o 0). Los operadores de relación son los siguientes:

Operador	Operación
<	Primer operando *menor que* el segundo.
>	Primer operando *mayor que* el segundo.
<=	Primer operando *menor o igual que* el segundo.
>=	Primer operando *mayor o igual que* el segundo.
!=	Primer operando *distinto que* el segundo.
==	Primer operando *igual que* el segundo.

Los operandos pueden ser de tipo entero, real o puntero. Por ejemplo,

```
main()
{
  int x = 10, y = 0, r = 0;
  r = x == y;    /* da como resultado 0 */
  r = x > y;     /* da como resultado 1 */
  r = x != y;    /* da como resultado 1 */
}
```

Un operador de relación equivale a una pregunta relativa a cómo son dos operandos entre sí. Por ejemplo, la expresión *x==y* equivale a la pregunta *¿x* es igual a *y*? Una respuesta *sí* equivale a un valor 1 (verdadero) y una respuesta *no* equivale a un valor 0 (falso).

Expresiones de Boole

Una expresión de *Boole* da como resultado 1 (verdadero - *true*) o 0 (falso - *false*). Desde un análisis riguroso, los operadores booleanos son los operadores lógicos && (AND), || (OR) y ! (NOT). Ahora bien, por ejemplo, piense que las comparaciones producen un resultado de tipo *boolean*. Quiere esto decir que el resultado de una comparación puede utilizarse como operando en una expresión de *Boole*. Según esto, podemos enunciar que los operadores que intervienen en una expresión de *Boole* pueden ser de relación (<, >, <=, >=, ==, y !=) y lógicos (&&, ||, y !).

El ejemplo siguiente, combina en una expresión operadores lógicos y operadores de relación. Más adelante, en este mismo capítulo, veremos la precedencia de los operadores.

```
main()
{
  int r = 0, s = 0;
  float x = 15, y = 18, z = 20;
  r = (x < y) && (y <= z) || s;  /* resultado r = 1 */
}
```

El resultado es 1 (verdadero) porque *x < y* es verdadero (1), *y <= z* es verdadero (1) y *s*, como vale 0, es falso (un valor distinto de cero es siempre verdadero y un valor cero es siempre falso). Por lo tanto, *1 && 1 || 0* = 1 || 0 = 1.

Operadores unitarios

Los operadores unitarios se aplican a un solo operando y son los siguientes: *****, **&**, **!**, **–** y **~**. El operador **!** ya lo hemos visto y los operadores ***** y **&** los veremos un poco más adelante.

Operador	Operación
–	Cambia de signo al operando (complemento a dos). El operando puede ser entero o real.
~	Complemento a 1. El operando tiene que ser entero (carácter ASCII 126).

El siguiente ejemplo muestra como utilizar estos operadores.

```
main()
{
  int a = 2, b = 0, c = 0;
  c = -a;  /* resultado c = -2 */
  c = ~b;  /* resultado c = -1 */
}
```

Operadores lógicos para manejo de bits

Estos operadores permiten realizar operaciones AND, OR, XOR y desplazamientos, bit por bit de los operandos.

Operador	Operación
&	Operación AND a nivel de bits.
|	Operación OR a nivel de bits (carácter ASCII 124).
^	Operación XOR a nivel de bits.
<<	Desplazamiento a la izquierda.
>>	Desplazamiento a la derecha.

Los operandos para este tipo de operaciones tienen que ser de tipo entero (**char**, **int**, **long**, o **enum**), no pueden ser reales. Por ejemplo,

```
main()
{
  unsigned char a = 255, r = 0, m = 32;
  r = a & 017;  /* r=15. Pone a cero todos los bits de a
                   excepto los 4 bits de menor peso */
  r = r | m;    /* r=47. Pone a 1 todos los bits de r que
                   están a 1 en m */
  r = a & ~07;  /* r=248. Pone a 0 los 3 bits de menor peso de a */
  r = a >> 7;   /* r=1. Desplazamiento de 7 bits a la derecha */
}
```

En las operaciones de desplazamiento el primer operando es desplazado tantas posiciones como indique el segundo. Si el desplazamiento es a izquierdas, se rellena con ceros por la derecha; si el desplazamiento es a derechas, se rellena con ceros por la izquierda si el operando es de tipo **unsigned**, en otro caso se rellena con el bit de signo.

Operadores de asignación

El resultado de una operación de asignación es el valor almacenado en el operando izquierdo, lógicamente después de que la asignación se ha realizado. El valor que se asigna es convertido al tipo del operando de la izquierda. Incluimos aquí los operadores de incremento y decremento porque implícitamente estos operadores realizan una asignación sobre su operando.

Operador	Operación
++	Incremento.
—	Decremento.
=	Asignación simple.
*=	Multiplicación más asignación.
/=	División más asignación.
%=	Módulo más asignación.
+=	Suma más asignación.
–=	Resta más asignación.
<<=	Desplazamiento a izquierdas más asignación.
>>=	Desplazamiento a derechas más asignación.
&=	Operación AND sobre bits más asignación.
|=	Operación OR sobre bits más asignación.
^=	Operación XOR sobre bits más asignación.

A continuación se muestran algunos ejemplos con estos operadores.

```
main()
{
  int x = 0, n = 10, i = 1;
  x++;          /* incrementa el valor de x en 1                */
  ++x;          /* incrementa el valor de x en 1                */
  x = --n;      /* decrementa n en 1 y asigna el resultado a x */
```

```
  x = n--;      /* asigna el valor de n a x y después          */
                /* decrementa n en 1                          */
  i += 2;       /* realiza la operación i = i + 2             */
  x *= n - 3;   /* realiza la operación x = x * (n-3) y no     */
                /* x = x * n - 3                              */
  n >>= 1;      /* realiza la operación n = n >> 1 la cual des-*/
                /* plaza el contenido de n un bit a la derecha */
}
```

Operador condicional

C tiene un operador ternario (*?:*) que se utiliza en expresiones condicionales, las cuales tienen la forma:

operando1 **?** *operando2* **:** *operando3*

El valor resultante de la expresión *operando1* debe ser de tipo entero, real o puntero. La ejecución se realiza de la siguiente forma:

- Si el resultado de la evaluación de *operando1* es distinta de 0, el resultado de la expresión condicional es *operando2*.
- Si el resultado de la evaluación de *operando1* es 0, el resultado de la expresión condicional es *operando3*.

El siguiente ejemplo asigna a *mayor* el resultado de *(a > b) ? a : b*, que será *a* si *a* es mayor que *b* y *b* si *a* no es mayor que *b*.

```
main()
{
  float a = 10.2, b = 20.5, mayor = 0;
  mayor = (a > b) ? a : b;  /* mayor de a y b */
}
```

Operador coma

Un par de expresiones separadas por una coma son evaluadas de izquierda a derecha. Todos los efectos secundarios de la expresión de la izquierda son ejecutados antes de evaluar la expresión de la derecha, a continuación el valor de la expresión de la izquierda es descartado. El tipo y el valor del resultado son el tipo y el valor del operando de la derecha. Algunos ejemplos son:

```
aux = v1, v1 = v2, v2 = aux;
for (a = 256, b = 1; b < 512; a/=2, b *=2)
```

En las líneas del ejemplo anterior, la coma simplemente hace que las expresiones separadas por la misma se evaluen de izquierda a derecha.

```
fx(a, (b = 10, b - 3), c, d)
```

Esta línea es una llamada a una función *fx*. En la llamada se pasan cuatro argumentos, de los cuales el segundo (*b = 10, b - 3*) tiene un valor 7.

Operador dirección-de

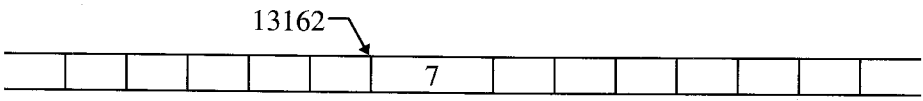
El operador & (dirección de) da la dirección de su operando. Por ejemplo,

```
int a = 7; /* la variable entera 'a' almacena el valor 7 */
printf("dirección de memoria = %d, dato = %d\n", &a, a);
```

El resultado de las sentencias anteriores puede ser similar al siguiente:

```
dirección de memoria = 13162, dato = 7
```

El resultado desde el punto de vista gráfico puede verlo en la figura siguiente. La figura representa un segmento de memoria de *n* bytes. En este segmento localizamos el entero 7 de dos bytes de longitud, en la dirección 13162. La variable *a* representa al valor 7 y *&a* (dirección de *a*; esto es, donde se localiza *a*) es 13162.



Este operador no se puede aplicar a un campo de bits perteneciente a una estructura o a un identificador declarado con el calificador **register**.

Operador de indirección

El operador * (indirección) accede a un valor indirectamente a través de un puntero. El resultado es el valor direccionado (apuntado) por el operando.

Un *puntero* es una variable capaz de contener una dirección de memoria que indica dónde se localiza un objeto de un tipo especificado (por ejemplo, un entero). La sintaxis para definir un puntero es:

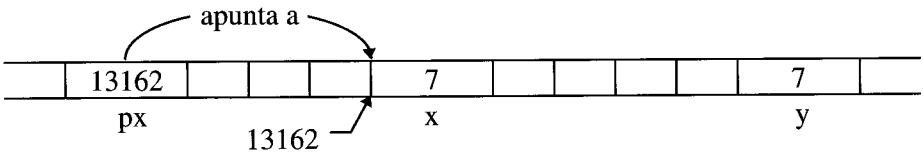
```
tipo *identificador;
```

donde *tipo* es el tipo del objeto apuntado e *identificador* el nombre del puntero.

El siguiente ejemplo declara un puntero *px* a un valor entero *x* y después asigna este valor al entero *y*.

```
main()
{
  int *px, x = 7, y = 0;    /* px es un puntero a un valor entero */
  px = &x;                  /* en el puntero px se almacena la dirección de x */
  y = *px;                  /* a y se le asigna el valor localizado en la
                               dirección almacenada en px */
}
```

Suponiendo que la dirección de *x* es 13162, el resultado expresado de una forma gráfica es el siguiente:



lo que indica que el contenido de *px* (**px*) es 7. La sentencia *y* = **px* se lee "*y* igual al contenido de *px*". De una forma más explícita diríamos "*y* igual al contenido de la dirección dada por *px*".

Operador sizeof

El operador **sizeof** da como resultado el tamaño en bytes de su operando. El operando o es el *identificador* del objeto o es el *tipo* del objeto. Por ejemplo,

```
#include <stdio.h>
main()
{
  int a = 0, t = 0;
  t = sizeof a;
  printf("El tamaño del entero a es: %d\n", t);
  printf("El tamaño de un float es: %d\n", sizeof(float));
}
```

Observe que los paréntesis son opcionales, excepto cuando el operando se corresponde con un tipo de datos.

El operador **sizeof** se puede aplicar a cualquier objeto de un tipo fundamental o de un tipo definido por el usuario, excepto al tipo **void**, a un array de dimensión no especificada, a un campo de bits o a una función.

PRIORIDAD Y ORDEN DE EVALUACIÓN

La tabla que se presenta a continuación, resume las reglas de prioridad y asociatividad de todos los operadores. Los operadores escritos sobre una misma línea

tienen la misma prioridad. Las líneas se han colocado de mayor a menor prioridad.

Una expresión entre paréntesis, siempre se evalúa primero. Los paréntesis tienen mayor prioridad y son evaluados de más internos a más externos.

Operador	Asociatividad
() [] . -> sizeof	izquierda a derecha
- ~ ! * & ++ — (tipo)	derecha a izquierda
* / %	izquierda a derecha
+ -	izquierda a derecha
<< >>	izquierda a derecha
< <= > >=	izquierda a derecha
== !=	izquierda a derecha
&	izquierda a derecha
^	izquierda a derecha
|	izquierda a derecha
&&	izquierda a derecha
||	izquierda a derecha
?:	derecha a izquierda
= *= /= %= += -= <<= >>= &= |= ^=	derecha a izquierda
,	izquierda a derecha

En el siguiente ejemplo, primero se asigna *z* a *y* y a continuación *y* a *x*. Esto es así porque la asociatividad para este operador es de derecha a izquierda.

```
void main()
{
  int x = 0, y = 0, z = 15;
  x = y = z; /* resultado x = y = z = 15 */
}
```

CONVERSIÓN DE TIPOS

Cuando los operandos que intervienen en una operación son de tipos diferentes, antes de realizar la operación especificada, se convierten a un tipo común, de acuerdo con las reglas que se exponen a continuación.

Las reglas que se exponen, se aplican en ese orden, para cada operación binaria perteneciente a una expresión, siguiendo el orden de evaluación expuesto anteriormente.

1. Si un operando es de tipo **long double**, el otro operando es convertido a tipo **long double**.
2. Si un operando es de tipo **double**, el otro operando es convertido a tipo **double**.
3. Si un operando es de tipo **float**, el otro operando es convertido a tipo **float**.
4. Un **char** o un **short**, con o sin signo, se convertirán a un **int**, si el tipo **int** puede representar todos los valores del tipo original, o a **unsigned int** en caso contrario.
5. Si un operando es de tipo **unsigned long**, el otro operando es convertido a **unsigned long**.
6. Si un operando es de tipo **long**, el otro operando es convertido a tipo **long**.
7. Si un operando es de tipo **unsigned int**, el otro operando es convertido a tipo **unsigned int**.

Por ejemplo,

```
long a;
unsigned char b;
int c;
float d;
int f;
f = a + b * c / d;
```

En la expresión anterior se realiza primero la multiplicación, después la división y por último la suma. Según esto, el proceso de evaluación será de la forma siguiente:

1. *b* es convertido a **int** (paso 4).
2. *b* y *c* son de tipo **int**. Se ejecuta la multiplicación (*) y se obtiene un resultado de tipo **int**.
3. Como el *d* es de tipo **float**, el resultado de *b* * *c*, es convertido a **float** (paso 3). Se ejecuta la división (/) y se obtiene un resultado de tipo **float**.
4. *a* es convertido a **float** (paso 3). Se ejecuta la suma (+) y se obtiene un resultado de tipo **float**.
5. El resultado de *a* + *b* * *c* / *d*, para ser asignado a *f*, es pasado a entero por truncamiento, esto es, eliminando la parte fraccionaria.

Resumen

- Los operandos que intervienen en una determinada operación, son convertidos al tipo del operando de precisión más alta.
- En C, las constantes reales son de tipo **double** por defecto.
- Una expresión de Boole da como resultado 1 si es cierta y 0 si es falsa.
- En una asignación, el valor de la parte derecha es convertido al tipo de la variable de la izquierda, de acuerdo con las siguientes reglas:
  - ◊ Los caracteres se convierten a enteros con o sin extensión de signo, dependiendo esto de la instalación. Generalmente la conversión se hace con extensión de signo.
  - ◊ Los enteros se convierten a caracteres preservando los bits de menor peso, esto es desechando los bits de mayor peso en exceso.
  - ◊ Los reales son convertidos a enteros, truncando la parte fraccionaria.
  - ◊ Un **double** pasa a **float**, redondeando y perdiendo precisión si el valor **double** no puede ser representado exactamente como **float**.
- También ocurre conversión cuando un valor es pasado como argumento a una función. Estas conversiones son ejecutadas independientemente sobre cada argumento en la llamada.

CONVERSIÓN EXPLÍCITA DEL TIPO DE UNA EXPRESIÓN

En C, está permitida una conversión explícita del tipo de una expresión mediante una construcción denominada *cast*, que tiene la forma:

```
(nombre-de-tipo) expresión
```

La expresión es convertida al tipo especificado si esa conversión está permitida; en otro caso, se obtendrá un error. La utilización apropiada de construcciones *cast* garantiza una evaluación consistente, pero siempre que se pueda, es mejor evitarla ya que suprime la verificación de tipo proporcionada por el compilador y por consiguiente puede conducir a resultados inesperados.

Por ejemplo, la función raíz cuadrada (**sqrt**), espera como argumento un tipo **double**. Para evitar resultados inesperados en el caso de pasar un argumento de otro tipo, podemos escribir:

```
sqrt((double)(n+2))
```

En C++ una construcción *cast* puede expresarse también así:

nombre-de-tipo(expresión)

sintaxis similar a la llamada a una función, por lo que recibe el nombre de *notación funcional*. De acuerdo con esto, el ejemplo anterior lo escribiríamos así:

```
sqrt(double(n+2))
```

La notación funcional sólo puede ser utilizada con tipos que tengan un nombre simple. Si no es así, utilice **typedef** para asignar al tipo derivado un nombre simple. Por ejemplo,

```
typedef int * pint;
int *p = pint(0x1F5);
```

EJERCICIOS PROPUESTOS

1. Escriba un programa que visualice en el monitor los siguientes mensajes:

```
Bienvenido al mundo del C.
Podrás dar solución a muchos problemas.
```

2. Defina un tipo enumerado *vehículos*.

3. ¿Qué resultados se obtienen al realizar las operaciones siguientes?

```
int a = 10, b = 3, c, d, e;
float x, y;
x = a / b;
c = a < b && 25;
d = a + b++;
e = ++a - b;
y = (float)a / b;
```

4. Escriba las sentencias necesarias para visualizar el tamaño en *bytes* de cada uno de los tipos fundamentales de C.

5. Escriba un programa que visualice su nombre, dirección y teléfono en líneas diferentes y centrados en la pantalla.

6. Decida qué tipos de datos necesita para escribir un programa que calcule la suma y la media de cuatro números de tipo **int**.

7. Escriba el código necesario para evaluar la expresión:

$$\frac{b^2 - 4ac}{2a}$$

para valores de $a = 1$, $b = 5$ y $c = 2$.

CAPÍTULO 3

ESTRUCTURA DE UN PROGRAMA

En este capítulo estudiará cómo es la estructura de un programa C. Partiendo de un programa ejemplo sencillo analizaremos cada una de las partes que componen su estructura, así tendrá un modelo para realizar sus propios programas. También veremos cómo se construye un programa a partir de varios módulos fuente. Por último, estudiaremos las clases de variables que puede utilizar y su accesibilidad.

ESTRUCTURA DE UN PROGRAMA C

Un programa fuente C está formado por una o más funciones. Recuerde que una función es un conjunto de instrucciones que realizan una tarea específica. Muchas de las funciones que utilizaremos pertenecen a la biblioteca de C, por lo tanto ya están escritas y compiladas. Otras tendremos que escribirlas nosotros mismos cuando necesitemos ejecutar una tarea que no esté en la biblioteca de C.

Todo programa C o C++ debe contener una función nombrada **main**, como se muestra a continuación:

```
main()
{
  // escriba aquí el código que quiere ejecutar
}
```

Los paréntesis después de **main** indican que ese identificador corresponde a una función. Esto es así para todas las funciones. A continuación y antes de la primera línea de código fuente hay que escribir una llave de apertura - { - es el punto por donde empezará a ejecutarse el programa. Después se escriben las sen-

tencias a que da lugar la tarea a ejecutar y se finaliza con una llave de cierre - } - es el punto donde finalizará la ejecución del programa.

Según lo expuesto hasta ahora, la solución de cualquier problema no debe considerarse inmediatamente en términos de sentencias correspondientes a un lenguaje, sino de elementos naturales del problema mismo, abstraídos de alguna manera, que darán lugar al desarrollo de las funciones mencionadas.

El diseño *top down* de programas, consiste precisamente en encontrar la solución de un problema mediante la aplicación sistemática de descomposición del problema en subproblemas cada vez más simples, aplicando la máxima de dividir para vencer.

El empleo de esta técnica de desarrollo de programas, así como la utilización únicamente de estructuras secuenciales, alternativas y repetitivas, nos conduce a la denominada *programación estructurada*. Esta ha sido la técnica empleada para desarrollar todos los ejemplos de este libro.

Para explicar como es la estructura de un programa C, vamos a plantear un ejemplo sencillo de un programa que presente una tabla de equivalencia entre grados centígrados y grados fahrenheit, como indica la figura siguiente:

```
-30 C     -22.00 F
-24 C     -11.20 F
  .
  .
  .
 90 C     194.00 F
 96 C     204.80 F
```

La relación entre los grados *centígrados* y los grados *fahrenheit* viene dada por la expresión *GradosFahrenheit = 9/5 * nGradosCentígrados + 32*. Los cálculos los vamos a realizar para un intervalo de –30 a 100 grados centígrados con incrementos de 6.

Analicemos el problema. ¿Qué piden? Escribir cuántos grados fahrenheit son –30 C, –24 C, ..., *nGradosCentigrados*, ..., 96 C. Y ¿cómo hacemos esto? Aplicando la fórmula

```
GradosFahrenheit = 9/5 * nGradosCentígrados + 32
```

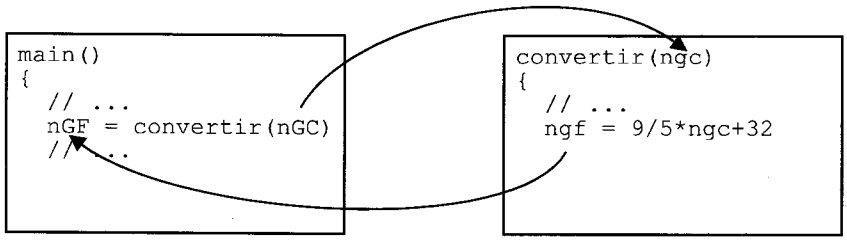
una vez para cada valor de *nGradosCentígrados*, desde –30 a 100 con incrementos de 6. Hacemos un alto y pensamos en un problema análogo; por ejemplo, cuando nos piden calcular el logaritmo de 2, en general de *n* ¿qué hacemos? Utilizar la función *log*; esto es,

```
x = log(n)
```

Análogamente, si tuviéramos una función *convertir* que hiciera los cálculos para convertir *n* grados centígrados en grados fahrenheit, escribiríamos,

```
GradosFahrenheit = convertir(nGradosCentígrados)
```

Sin casi darnos cuenta estamos haciendo una descomposición del problema general en subproblemas más sencillos de resolver. Recordando que un programa C tiene que tener una función **main**, además de otras funciones si lo consideramos necesario, ¿cómo se ve de una forma gráfica la sentencia anterior? La figura siguiente da respuesta a esta pregunta:



Análogamente a como hacíamos con la función logaritmo, aquí, desde la función **main**, llamamos a la función *convertir* que recibe como argumento el valor en grados centígrados a convertir. La función logaritmo nos daba como resultado el logaritmo del valor pasado. La función *convertir* nos da el valor en grados fahrenheit de los grados centígrados pasados. Sólo queda escribir en el monitor este resultado y repetir el proceso para cada uno de los valores descritos.

Seguro que pensará que todo el proceso se podría haber hecho utilizando solamente la función **main**, lo cual es cierto. Pero, lo que se pretende es que pueda ver de una forma clara que, en general, un programa C es un conjunto de funciones que se llaman entre sí con el fin de obtener el resultado perseguido, y que la forma sencilla de resolver un problema es descomponerlo en subproblemas más pequeños y por lo tanto más fáciles de solucionar; cada subproblema será resuelto por una función C.

Otro detalle importante es que las funciones que escriba deben ser autónomas, para posteriormente poderlas utilizar sin dificultades en otros programas.

Una función es autónoma cuando depende única y exclusivamente de sus propios argumentos. Por ejemplo, independientemente de que no sepamos cómo está hecha la función logaritmo, cuando la invocamos nos limitamos a pasar el valor del cual queremos calcular el logaritmo. Nuestra calculadora no requiere de

ninguna otra operación previa. Análogamente tienen que actuar las funciones que nosotros escribamos, porque si además de pasar los argumentos tuviéramos que realizar alguna operación externa a la misma, por ejemplo, declarar en **main** una determinada variable, sería una fuente de errores que al usuario de la función se le olvidara este requisito. En cambio, cuando una función es autónoma sólo tenemos que conocer qué valores hay que pasar y qué resultado devuelve la misma; sirva de ejemplo la función *convertir*.

Una vez analizado el problema, vamos a escribir el código. Ahora lo que importa es que aprenda cómo es la estructura de un programa, no por qué se escriben unas u otras sentencias, cuestión que aprenderá más tarde en éste y en sucesivos capítulos. Este ejemplo le servirá como plantilla para inicialmente escribir sus propios programas. Posiblemente su primer programa utilice solamente la función **main**, pero con este ejemplo tendrá un concepto más real de lo que es un programa C.

```
/* Paso de grados Centígrados a Fahrenheit (F = 9/5 * C + 32)
 *
 * grados.c
 */

/* Directrices para el preprocesador */
#include <stdio.h> /* declaraciones de las funciones C estándar
                    * de E/S.
                    */

/* Definición de constantes */
#define INF -30     /* límite inferior del intervalo de ºC */
#define SUP 100     /* límite superior */

/* Declaración de funciones */
float convertir(int c);    /* prototipo de la función convertir */

void main()          /* función principal - comienzo del programa */
{
  /* Declaración de variables locales */
  int nGradosCentigrados = 0;
  int incremento = 6; /* inicializar incremento con el valor 6 */
  float GradosFahrenheit = 0;

  nGradosCentigrados = INF;  /* sentencia de asignación */
  while (nGradosCentigrados <= SUP)
  {
    /* Se llama a la función convertir */
    GradosFahrenheit = convertir(nGradosCentigrados);
    /* Se escribe la siguiente línea de la tabla */
    printf("%10d C %10.2f F\n", nGradosCentigrados, GradosFahrenheit);
    /* Siguiente valor a convertir */
    nGradosCentigrados += incremento;
  }
}   /* fin de la función principal y del programa */
```

```
/*************** Definición de la función convertir *************/
float convertir(int gcent)
{
  /* Declaración de variables locales */
  float gfahr; /* variable local accesible solamente aquí,
                  en la función */
  /* los operandos son convertidos al tipo del operando de
     precisión más alta (float) */
  gfahr = (float)9 / (float)5 * gcent + 32;
  return (gfahr);  /* valor que retorna la función convertir */
} /* Fin de la función de convertir */
```

En el ejemplo realizado podemos observar que un programa C consta de:

- Directrices para el preprocesador
- Definiciones y/o declaraciones
- Función **main**
- Otras funciones

El orden establecido no es esencial, aunque sí bastante habitual. Así mismo, cada función consta de:

- Definiciones y/o declaraciones
- Sentencias a ejecutar

Cuando se escribe una función C, las definiciones y/o declaraciones hay que realizarlas antes de escribir la primera sentencia. En cambio, C++ permite realizar las declaraciones en cualquier lugar; esto es, en el momento en que se necesiten.

Directrices para el preprocesador

La finalidad de las directrices es facilitar el desarrollo, la compilación y el mantenimiento de un programa. Una directriz se identifica porque empieza con el carácter #. Las más usuales son la directriz de inclusión, **#include**, y la directriz de sustitución, **#define**. Las directrices son procesadas por el *preprocesador* de C, que es invocado por el compilador antes de proceder a la traducción del programa fuente. Más adelante, dedicaremos un capítulo para ver con detalle todas las directrices para el preprocesador de C.

Directriz de inclusión

En general, cuando se invoca a una función antes de su definición, es necesario incluir previamente a la llamada el prototipo de la función. Como ejemplo observe la función *convertir* del programa anterior. La definición de *convertir* está escrita a continuación de la función **main** y la función **main** llama a la función

escribir. Como la llamada está antes que la definición, es necesario escribir previamente el prototipo de la función llamada. Esto es así para todas las funciones, incluidas las funciones de la biblioteca de C, como **printf**. Las declaraciones de las funciones de la biblioteca de C se localizan en los ficheros de cabecera (ficheros con extensión *.h*). Estos ficheros generalmente se encuentran en el directorio predefinido *include* o en el directorio de trabajo.

Según lo expuesto en el apartado anterior, para incluir la declaración de una función de la biblioteca de C, basta con incluir el fichero de cabecera que la contiene antes de que ella sea invocada. Esto se hace utilizando la directriz **#include**.

Si el fichero de cabecera se encuentra en el directorio predefinido *include*, utilice **#include** <*fichero.h*>. Por ejemplo:

```
#include <stdio.h> /* declaraciones de las funciones C estándar
                      de E/S */
```

Si el fichero se encuentra, indistintamente, en el directorio de trabajo o en el directorio *include*, utilice **#include** "*fichero.h*". Por ejemplo:

```
#include "misfuncs.h"
```

La directriz anterior busca el fichero especificado primero en el directorio de trabajo y si no lo localiza, sigue la búsqueda en el directorio *include*.

Directriz de sustitución

Mediante la directriz **#define** *identificador valor* se indica al compilador, que toda aparición en el programa de *identificador*, debe ser sustituida por *valor*. Por ejemplo:

```
#define INF -30 /* límite inferior del intervalo ºC */
#define SUP 100 /* límite superior */
// ...
  nGradosCentigrados = INF;  /* sentencia de asignación */
  while (nGradosCentigrados <= SUP)
  {
    // ...
```

Cuando el *preprocesador* de C procese las directrices del ejemplo anterior, todas las apariciones de *INF* y de *SUP* en el programa fuente son sustituidas, como puede ver a continuación, por sus valores correspondientes.

```
nGradosCentigrados = -30;  /* sentencia de asignación */
while (nGradosCentigrados <= 100)
{
  // ...
```

Definiciones y declaraciones

Una declaración introduce uno o más nombres en un programa. Una declaración es una definición, a menos que no haya asignación de memoria como ocurre cuando se declara una función sin especificar el cuerpo, cuando se define un nuevo tipo, cuando se declara un sinónimo de un tipo o cuando con una variable definida en alguna parte se utiliza el calificador **extern** para hacerla accesible.

Toda variable debe ser definida antes de ser utilizada. La definición de una variable, declara la variable y además le asigna memoria. Además, una variable puede ser inicializada en la propia definición:

```
int nGradosCentigrados = 0;
int incremento = 6; /* inicializar incremento con el valor 6 */
float GradosFahrenheit = 0;
```

La definición de una función, declara la función y además incluye el cuerpo de la misma:

```
float convertir(int gcent)
{
  float gfahr;
  gfahr = (float)9 / (float)5 * gcent + 32;
  return (gfahr);
}
```

La declaración o la definición de una variable, así como la declaración de una función, pueden realizarse a *nivel interno* (dentro de la definición de una función) o a *nivel externo* (fuera de toda definición de función). La definición de una función, siempre ocurre a nivel externo.

Función main

Todo programa C tiene una función denominada **main**. Esta función es el punto de entrada al programa y también el punto de salida. Aunque algunos compiladores no requieren que se especifique el tipo retornado por esta función, generalmente **int** o **void**, conviene hacerlo porque hay compiladores que dan un mensaje de aviso si no se hace; concretamente, los compiladores que incluyen la característica de comprobación de tipos. Nos referimos a los compiladores de C++ y a algunos compiladores de C. Es una buena costumbre indicar explícitamente si la función devuelve o no un resultado. Por ejemplo:

```
int main()
{
  // ...
  return 0;
}
```

El ejemplo anterior indica que cuando la función **main** finalice, devolverá un cero. Este valor es devuelto al proceso que invocó al programa para su ejecución. También podemos especificar que no devuelve nada, así:

```
void main()
{
  // ...
}
```

Sentencia simple

Una *sentencia simple* es la unidad ejecutable más pequeña de un programa C. Las sentencias controlan el flujo u orden de ejecución. Una sentencia C puede formarse a partir de: una palabra clave (**for**, **while**, **if** ... **else**, etc.), expresiones, declaraciones o llamadas a funciones. Cuando se escriba una sentencia hay que tener en cuenta las siguientes consideraciones:

- Toda sentencia simple termina con un *punto y coma* (;).
- Dos o más sentencias pueden aparecer sobre una misma línea, separadas por una de otra por un punto y coma.
- Una *sentencia nula* consta solamente de un *punto y coma*. Cuando veamos la sentencia **while**, podrá ver su utilización.

Sentencia compuesta o bloque

Una *sentencia compuesta* o bloque, es una colección de sentencias simples incluidas entre llaves - { } -. Un bloque puede contener a otros bloques. Un ejemplo de una sentencia de este tipo es el siguiente:

```
{
  GradosFahrenheit = convertir(nGradosCentigrados);
  printf("%10d C %10.2f F\n", nGradosCentigrados, GradosFahrenheit);
  nGradosCentigrados += incremento;
}
```

Funciones

Una función es una colección de sentencias que ejecutan una tarea específica. En un programa C se distinguen dos clases de funciones, las funciones definidas por el usuario y las funciones de biblioteca. En C, la definición de una función nunca puede contener a la definición de otra función.

Puesto que la función es la unidad fundamental de un programa C, vamos a describir cómo se declaran, cómo se definen y cómo se invocan. Posteriormente dedicaremos un capítulo para estudiarlas con más detalle.

Declaración de una función

La declaración de una función, también conocida como *prototipo de la función*, indica cuántos parámetros tiene la función y de qué tipo son, así como el tipo del valor retornado. Su sintaxis es:

```
tipo-resultado nombre-función ([lista de parámetros]);
```

El prototipo de una función es una plantilla que se utiliza para asegurar que una sentencia de invocación escrita antes de la definición de la función, es correcta; esto es, que son especificados los argumentos adecuados y que el valor retornado se trata correctamente. Esto chequeo de tipos y número de argumentos permite detectar durante la compilación si se ha cometido algún error.

Por ejemplo, la siguiente sentencia declara la función *convertir* para indicar que cuando sea invocada tiene que recibir un argumento entero y que cuando finalice su ejecución, retornará un valor real.

```
float convertir(int c);
```

En conclusión, la declaración de una función da características de la misma, pero no define el proceso que realiza.

Una función puede ser declarada implícitamente o explícitamente. La *declaración implícita* se da cuando la función es llamada y no existe una declaración previa (prototipo de la función). En este caso, C, por defecto, construye una función prototipo con tipo **int** para el resultado y la lista de tipos de argumentos se construye, según los parámetros formales especificados en la definición de la función. Esto obliga a que el tipo del resultado en la definición de la función sea **int**. La declaración implícita de una función no se contempla en C++. Por ello, se recomienda realizar siempre una declaración explícita de la función.

La *declaración explícita*, especifica el número y el tipo de los argumentos de la función, así como el tipo del valor retornado.

La *lista de argumentos* normalmente consiste en una lista de identificadores con sus tipos, separados por comas. En el caso de que se trate del prototipo de una función, se pueden omitir los identificadores. Por ejemplo:

```
float funcion_x(int, float, char);
```

En cambio, si se especifican, su ámbito queda restringido a la propia declaración; esto es, no son accesibles en otra parte. Por ejemplo, en la declaración siguiente, *a*, *b* y *c* no son accesibles en ninguna otra parte.

```
float funcion_x(int a, float b, char c);
```

De lo expuesto se deduce que los identificadores utilizados en la declaración de la función y los utilizados después en la definición de la misma función no necesariamente tienen que ser los mismos. Vea como ejemplo la declaración y la definición de la función *convertir* del programa anterior.

Para asegurar la portabilidad de los programas, se puede utilizar el tipo **void** para especificar que una función no acepta argumentos, ya que a diferencia de C++, en el lenguaje C estándar la ausencia de parámetros y la no especificación de **void** indica cualquier número y tipo de argumentos. Según esto, las declaraciones siguientes tienen la misma validez, aunque con significados diferentes.

```
float fnEscribir(void); /* función sin argumentos */
float fnEscribir();     /* función con cualquier número y tipo
                           de argumentos */
```

Las declaraciones de las funciones pertenecientes a la biblioteca estándar de C, como **printf**, son proporcionadas por los ficheros de cabecera o ficheros *.h*. Por eso, cuando un programa utiliza, por ejemplo, la función **printf** observará que se incluye el fichero de cabecera *stdio.h*.

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

Al especificar la sintaxis de las funciones de la biblioteca de C, también se indica el fichero de cabecera donde está declarada.

Definición de una función

La definición de una función consta de una *cabecera* de función y del *cuerpo* de la función encerrado entre llaves.

```
tipo-resultado nombre-función ([parámetros formales])
{
   declaraciones de variables locales;
   sentencias;
   [return [(]expresión[)]];
}
```

Las variables declaradas en el cuerpo de la función son locales y por definición solamente son accesibles dentro del mismo.

El *tipo del resultado* especifica qué tipo de datos retorna la función. Este, puede ser cualquier tipo fundamental, o tipo definido por el usuario, pero no puede ser un array o una función. Si no se especifica, se supone que es **int**. El resultado de una función es devuelto a la sentencia de llamada, por medio de la sentencia:

```
return [(]expresión[)];
```

Esta sentencia puede ser o no la última y puede aparecer más de una vez en el cuerpo de la función. En el caso de que la función no retorne un valor, se puede omitir o especificar simplemente **return**. Por ejemplo,

```
void fnEscribir()
{
  // ...
  return;
}
```

Los *parámetros formales* de una función son las variables que reciben los valores de los argumentos especificados en la llamada a la función. Consisten en una lista de identificadores con sus tipos, separados por comas. A continuación puede ver un ejemplo:

```
float convertir(int gcent) /* cabecera de la función */
{
  /* cuerpo de la función */
  float gfahr;
  gfahr = (float)9 / (float)5 * gcent + 32;
  return (gfahr); /* valor retornado */
}
```

La cabecera de la función también se podía haber escrito utilizando el estilo antiguo, de la siguiente forma:

```
float conversion(gcent)
int gcent;
{
  float gfahr;
  gfahr = (float)9 / (float)5 * gcent + 32;
  return (gfahr);
}
```

Observe que la declaración de los parámetros formales se hace a continuación de la cabecera de la función y antes de la llave de apertura del cuerpo de la función.

Llamada a una función

Para ejecutar una función hay que llamarla. La llamada a una función consta del nombre de la misma y de una lista de argumentos, denominados *parámetros actuales*, encerrados entre paréntesis y separados por comas. Por ejemplo:

```
/* Se llama a la función convertir */
   GradosFahrenheit = convertir(nGradosCentigrados);
```

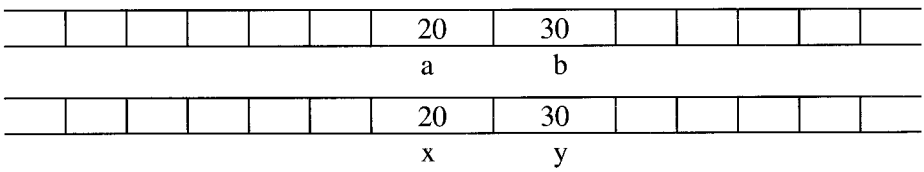
Pasando argumentos a las funciones

Cuando se llama a una función, el valor del primer parámetro actual es pasado al primer parámetro formal, el valor del segundo parámetro actual es pasado al segundo parámetro formal y así sucesivamente. Por defecto, todos los argumentos, excepto los arrays, son pasados *por valor*. Esto significa que a la función se pasa una copia del valor del argumento, no su dirección. Esto hace que la función invocada, no pueda alterar las variables que contienen los valores pasados.

En el ejemplo anterior, cuando se llama a la función *convertir*, el parámetro formal *gcent* recibe el valor del parámetro actual *nGradosCentigrados*.

En el siguiente ejemplo, puede observar que la función **main** llama a la función *intercambio* y le pasa los argumentos *a* y *b*. La función *intercambio* almacena en *x* el valor de *a* y en *y* el valor de *b*. Esto significa que los datos *a* y *b* se han duplicado. Esto es, ocurren las siguientes asignaciones:

```
x = a
y = b
```

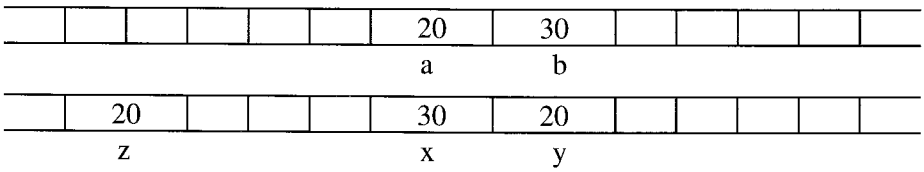


Los parámetros formales *x* e *y* son variables locales a la función *intercambio*, por lo tanto sólo son accesibles dentro de la propia función. Esto significa que las variables locales se crean cuando se ejecuta la función y se destruyen cuando finaliza la ejecución de la función.

```
/* valor.c - Paso de parámetros por valor */
#include <stdio.h>

void intercambio(int, int); /* prototipo de la función */
```

```
void main()
{
  int a = 20, b = 30;
  intercambio(a, b); /* a y b son pasados por valor */
  printf("a es %d y b es %d\n", a, b);
}

void intercambio(int x, int y)
{
  int z = x;
  x = y;
  y = z;
}
```

Después de ejecutarse la función *intercambio* el valor de *y* ha sido copiado en *x* y el valor de *x* ha sido copiado en *y*. Lo importante, es que vea que estas operaciones no afectan a los valores de *a* y de *b*.



Si lo que se desea es alterar los contenidos de los argumentos especificados en la llamada, entonces hay que pasar dichos argumentos *por referencia*. Esto es, a la función se le pasa la *dirección* de cada argumento y no su valor, lo que exige que los parámetros formales correspondientes sean punteros. Para pasar la dirección de un argumento, utilizaremos el operador **&**. Por ejemplo:

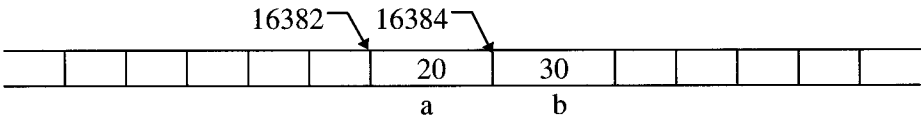
```
/* referen.c - Paso de parámetros por referencia */
#include <stdio.h>
void intercambio(int *, int *); /* prototipo de la función */

void main()
{
  int a = 20, b = 30;

  intercambio(&a, &b); /* a y b son pasados por referencia */
  printf("a es %d y b es %d\n", a, b);
}

void intercambio(int *x, int *y)
{
  int z = *x; /* z = contenido de la dirección x */
  *x = *y;    /* contenido de x = contenido de y */
  *y = z;     /* contenido de y = z */
}
```

En el ejemplo expuesto podemos ver que la función *intercambio* tiene dos parámetros *x* e *y* de tipo *puntero a un entero*, que reciben las direcciones de *a* y de *b*, respectivamente. Esto quiere decir que al modificar el contenido de las direccio-

nes *x* e *y*, indirectamente estamos modificando los valores *a* y *b*. Recordar la definición de *puntero* y las definiciones de los operadores de *indirección* (*) y *dirección-de* (&) vistas en el capítulo anterior.

La explicación gráfica del ejemplo que acabamos de ver es la siguiente:

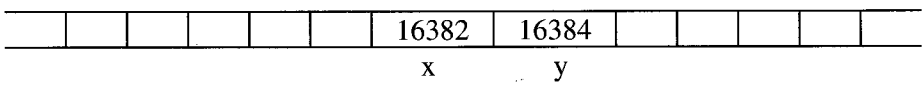
1. La función **main** define las variables *a* y *b* y llama a la función *intercambio* pasando las direcciones de dichas variables como argumento. Supongamos que las direcciones en memoria de estas variables son 16382 y 16384, respectivamente.



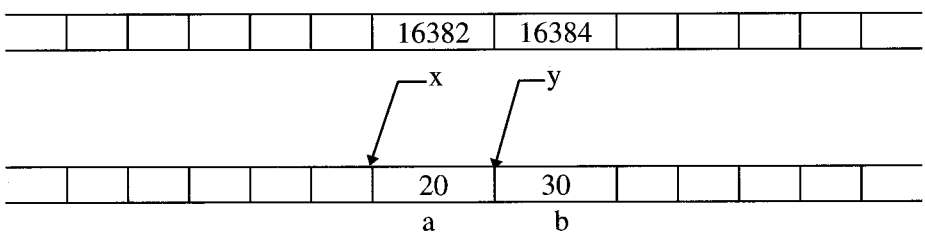
2. Cuando se llama a la función *intercambio* el valor del primer parámetro actual es pasado al primer parámetro formal y el valor del segundo parámetro actual es pasado al segundo parámetro formal. Esto es,

```
x = &a
y = &b
```

Los parámetros formales son variables locales a la función que se crean en el instante en el que ésta es llamada. Según esto,



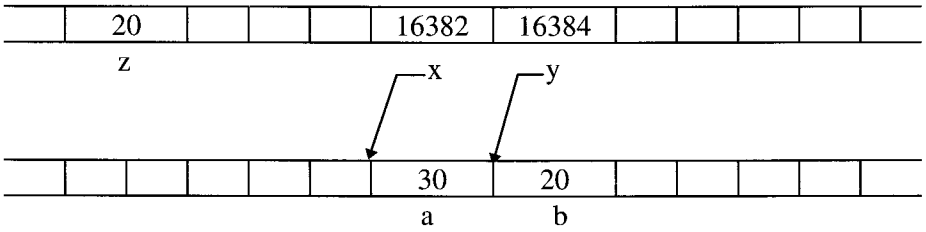
3. Ahora *x* apunta al dato *a*; esto es, el valor de *x* especifica el lugar donde se localiza el dato *a* dentro de la memoria. Análogamente, diremos que *y* apunta al dato *b*.



Observe que **x* hace referencia al mismo dato de memoria que *a* y que **y* hace referencia al mismo dato de memoria que *b*.

4. A continuación se ejecutan las sentencias:

```
z = *x;  /* se asigna a la variable z el valor de a */
*x = *y; /* se asigna a la variable a el valor de b */
*y = z   /* se asigna a la variable b el valor de z */
```



Cuando la función *intercambio* finaliza, los valores de *a* y *b* han sido intercambiados y las variables *x* e *y*, por el hecho de ser locales, son destruidas.

Por lo tanto, pasar parámetros por referencia a una función es hacer que la función acceda indirectamente a las variables pasadas. A diferencia de cuando se pasan los parámetros por valor, no hay duplicidad de datos.

Cuando se trate de funciones de la biblioteca de C, también se le presentarán ambos casos. Por ejemplo, suponga que desea escribir un programa que lea dos valores *a* y *b* y que escriba la suma de ambos. La solución es la siguiente:

```
/* sumar.c - Sumar dos valores */
#include <stdio.h>

void main()
{
  int a = 0, b = 0, s = 0;
  printf("Valores de a y b: ");
  scanf("%d %d", &a, &b); /* leer desde el teclado a y b */
  s = a + b;
  printf("La suma es %d\n", s);
}
```

La ejecución de este programa puede ser así:

```
Valores de a y b: 10 20↵
La suma es 30
```

La función **scanf** lee valores introducidos por el teclado y los asigna a las variables especificadas; en el ejemplo, a las variables *a* y *b*. Ahora, observe que dichas variables son pasadas por referencia y no por valor. Por el contrario, la función **printf** escribe los valores de las variables o expresiones especificadas; en este caso, observe cómo la variable *s* es pasada por valor.

Más adelante estudiaremos con más detalle las funciones **printf** y **scanf**.

PROGRAMA C FORMADO POR MÚLTIPLES FICHEROS

Según lo que hemos visto, un programa C es un conjunto de funciones que se llaman entre sí. Lo que no debemos pensar es que todo el programa tiene que estar escrito en un único fichero *.c*. De hecho no es así, ya que además del fichero *.c*, intervienen uno o más ficheros de cabecera. Por ejemplo, en el último programa está claro que intervienen los ficheros *sumar.c* y *stdio.h*, pero ¿dónde está el código de las funciones de la biblioteca de C invocadas? El programa *sumar* invoca a dos funciones de la biblioteca de C, **scanf** y **printf**, que no están escritas en el fichero *sumar.c*, sino que están escritas en otro fichero separado que forma parte de la biblioteca de C, al que se accede durante el proceso de enlace para obtener el código correspondiente a dichas funciones. Análogamente, nosotros podemos hacer lo mismo; esto es, podemos optar por escribir las funciones que nos interesen en uno o más ficheros separados y utilizar para las declaraciones y/o definiciones uno o más ficheros de cabecera.

Un fichero fuente puede contener cualquier combinación de directrices para el compilador, declaraciones y definiciones. Pero, una función o una estructura, no puede ser dividida entre dos ficheros fuente. Por otra parte, un fichero fuente no necesita contener sentencias ejecutables; esto es, un fichero fuente puede estar formado, por ejemplo, solamente por definiciones de variables que son referenciadas desde otros ficheros fuentes.

Como ejemplo de lo expuesto, vamos a escribir un programa C que nos dé como resultado el mayor de tres valores dados. Para ello, escribiremos una función *max* que devuelva el mayor de dos valores pasados como argumentos en la llamada. Esta función será invocada dos veces por la función **main**; la primera para calcular el mayor de los dos primeros valores, y la segunda para calcular el mayor del resultado anterior y del tercer valor. Los tres valores serán introducidos por el teclado. El código correspondiente lo escribiremos en dos ficheros:

- El fichero *modulo01.c* contendrá la función **main**, además de otras declaraciones.
- El fichero *modulo02.c* contendrá la función *max*.

Escriba el código siguiente en un fichero llamado *modulo01.c*.

```
/************************ modulo01.c ***************************
              Fichero fuente 1 - función principal
***************************************************************/
#include <stdio.h>

/* Declaración de funciones */
int max(int x, int y);
```

```
void main () /* función principal */
{
  int a = 0, b = 0, c = 0; /* definición de variables */
  int mayor = 0;

  printf("Valores a, b y c: ");
  scanf("%d %d %d", &a, &b, &c);

  mayor = max(a, b);      /* mayor de a y b */
  mayor = max(mayor, c); /* mayor del resultado anterior y de c */
  printf("%d\n", mayor);
}
```

A continuación, escriba este otro código en un fichero llamado *modulo02.c*.

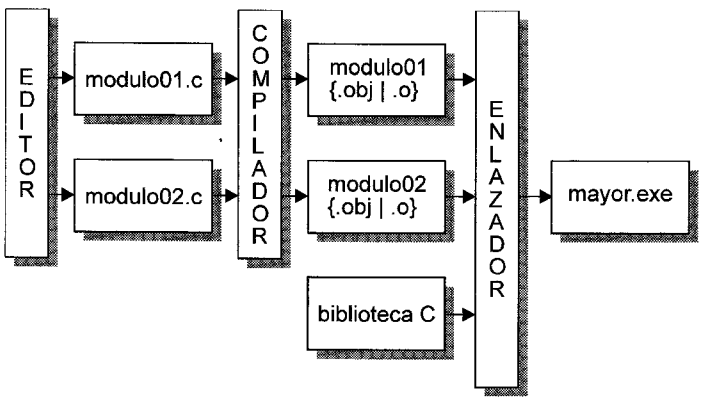
```
/************************ modulo02.c ***************************
                    Fichero fuente 2 - función max
*****************************************************************/
/* Función max. Toma dos valores, x e y, y devuelve el mayor */
int max(int x, int y)
{
  int z = 0;
  z = (x > y) ? x : y;
  return z;
}
```

Para compilar un programa formado por varios ficheros, se deben compilar por separado cada uno de los ficheros y, a continuación, enlazarlos para formar un único fichero ejecutable. Por ejemplo, para compilar y enlazar los ficheros *modulo01.c* y *modulo02.c*, utilice alguna de las órdenes siguientes:

`cl modulo01.c modulo02.c /Femayor` en MS-DOS (Microsoft C) o

`cc modulo01.c modulo02.c -o mayor.exe` en UNIX.

El resultado es el programa ejecutable *mayor.exe*. La siguiente figura muestra paso a paso la forma de obtener el fichero ejecutable.



La figura anterior indica que una vez editados los ficheros *modulo01.c* y *modulo02.c*, se compilan obteniéndose como resultado los ficheros objeto *modulo01* y *modulo02*, los cuales se enlazan con las funciones necesarias de la biblioteca de C, obteniéndose finalmente el fichero ejecutable *mayor.exe*.

ACCESIBILIDAD DE VARIABLES

Se denomina *ámbito* de una variable a la parte de un programa donde dicha variable puede ser referenciada por su nombre. Una variable puede ser limitada a un bloque, a un fichero, a una función, o a una declaración de una función.

Variables globales y locales

Cuando una variable se declara fuera de todo bloque en un programa, es accesible desde su punto de definición o declaración hasta el final del fichero fuente. Esta variable recibe el calificativo de *global*.

Una *variable global* existe y tiene valor desde el principio hasta el final de la ejecución del programa. Las funciones tienen todas carácter global.

Si la declaración de una variable se hace dentro de un bloque, el acceso a dicha variable queda limitado a ese bloque y a los bloques contenidos dentro de éste por debajo de su punto de declaración. En este caso, la variable recibe el calificativo de *local* o *automática*.

Una *variable local* existe y tiene valor desde su punto de declaración hasta el final del bloque donde está definida. Cada vez que se ejecuta el bloque que la contiene, la variable local es nuevamente definida, y cuando finaliza la ejecución del mismo, la variable local deja de existir. Un elemento con carácter local es accesible solamente dentro del bloque al que pertenece.

El siguiente ejemplo muestra el ámbito de las variables, dependiendo de si están definidas en un bloque o fuera de todo bloque. En este ejemplo, al tratar de definir el ámbito de una variable, distinguimos cuatro niveles:

- El nivel externo (fuera de todo bloque).

  Las variables definidas en este nivel, son accesibles desde el punto de definición hasta el final del programa.

- El nivel del bloque de la función **main**.

  Las variables definidas en este nivel, solamente son accesibles desde la propia función **main** y, por lo tanto, son accesibles en los bloques 1 y 2.

- El nivel del bloque 1 (sentencia compuesta).

  Las variables definidas en este nivel, solamente son accesibles en el interior del bloque 1 y, por lo tanto, en el bloque 2.

- El nivel del bloque 2 (sentencia compuesta).

  Las variables definidas en este nivel, solamente son accesibles en el interior del bloque 2.

```
/* vars01.c - Variables globales y locales
 */
#include <stdio.h>

/* Definición de var1 como variable GLOBAL */
int var1 = 50;

void main()
{ /* COMIENZO DE main Y DEL PROGRAMA */

  printf("%d\n", var1); /* se escribe 50 */

  { /* COMIENZO DEL BLOQUE 1 */
    /* Definición de var1 y var2 como variables
       LOCALES en el BLOQUE 1 y en el BLOQUE 2 */

    int var1 = 100, var2 = 200;

    printf("%d %d\n", var1, var2);
    /* escribe 100 y 200 */

    { /* COMIENZO DEL BLOQUE 2 */
      /* Redefinición de la variable LOCAL var1 */
      int var1 = 0;

      printf("%d %d\n", var1, var2);
      /* escribe 0 y 200 */

    } /* FINAL DEL BLOQUE 2 */

    printf("%d\n", var1); /* se escribe 100 */

  } /* FINAL DEL BLOQUE 1 */

 .printf("%d\n", var1); /* se escribe 50 */

} /* FINAL DE main Y DEL PROGRAMA */
```

En el ejemplo anterior se observa que una variable *global* y otra *local* pueden tener el *mismo nombre*, pero no guardan relación una con otra, lo cual da lugar a que la variable *global* quede anulada en el ámbito de accesibilidad de la *local* del mismo nombre. Como ejemplo observe lo que ocurre en el programa anterior con *var1*.

Los parámetros declarados en la lista de parámetros de la declaración de una función o prototipo de la función, tienen un ámbito restringido a la propia declaración de la función.

```
int max(int x, int y); /* declaración de la función max */
```

Los parámetros *x* e *y* en la declaración de la función *max* están restringidos a la propia declaración. Por esta razón se pueden omitir; esto es, la siguiente línea sería equivalente a la anterior:

```
int max(int, int); /* declaración de la función max */
```

Los *parámetros formales* declarados en la lista de parámetros de la definición de una función, son locales a la función.

```
int max(int x, int y)  /* definición de la función max */
{
  int z = 0;
  z = (x > y) ? x : y;
  return z;
}
```

Los parámetros *x* e *y* en la definición de la función *max* son locales a la función; esto es, *x* e *y* se crean cuando se llama a la función para su ejecución y dejan de existir cuando finaliza la ejecución de la función. Esta es la razón por la que *x* e *y* sólo son accesibles dentro de la propia función.

Clases de almacenamiento

Por defecto, todas las variables llevan asociada una clase de almacenamiento que determina su accesibilidad y existencia. Los conceptos de accesibilidad y de existencia tanto para las variables como para las funciones, pueden alterarse por los calificadores:

auto	almacenamiento automático
register	almacenamiento en un registro
static	almacenamiento estático
extern	almacenamiento externo

Los calificadores **auto** o **register** pueden ser utilizados solamente con variables locales; el calificador **extern** puede ser utilizado solamente con variables globales o funciones; y el calificador **static** puede ser utilizado con variables locales, globales o funciones.

Variables declaradas a nivel externo

En una variable declarada a nivel externo, esto es, fuera de toda definición de función, se pueden utilizar los calificadores **static** o **extern**, o bien *omitir* el calificador, en cuyo caso es como si se hubiera especificado **extern**. A nivel externo no se pueden utilizar los calificadores **auto** o **register**.

Una variable declarada a nivel externo es una *definición* de la variable o una *referencia* a una variable definida en otra parte. Esto quiere decir que la declaración de una variable externa inicializa la variable a cero (valor por defecto) o a un valor especificado.

A una variable definida a nivel externo, se la puede hacer accesible antes de su definición o en otro fichero fuente, utilizando el calificador **extern**. Esto quiere decir, que la utilización del calificador **extern** tiene sentido cuando la variable ha sido definida a nivel externo, una vez, y solamente una, en cualquier parte del programa y queremos tener acceso a ella en otra parte donde no es visible.

El siguiente ejemplo, formado por dos ficheros fuente, muestra lo expuesto con claridad. El fichero fuente *UNO.C* define la variable *var* y le asigna el valor 5. Así mismo, para hacer visible *var* antes de su definición utiliza la declaración **extern int** *var*.

```
/*****************************************************************
                        Fichero fuente UNO.C
*****************************************************************/
#include <stdio.h>
void funcion_1();
void funcion_2();

extern int var; /* declaración de var. Referencia a la variable var
                   definida a continuación */
void main()
{
  var++;
  printf("%d\n", var); /* se escribe 6 */
  funcion_1();
}

int var = 5; /* definición de var */

void funcion_1()
{
  var++;
  printf("%d\n", var); /* se escribe 7 */
  funcion_2();
}
```

El fichero fuente *DOS.C* utiliza la declaración **extern int** *var* para poder acceder a la variable *var* definida en el fichero fuente *UNO.C*.

```
/*************************************************************
                        Fichero fuente DOS.C
*************************************************************/
#include <stdio.h>
extern int var; /* declaración de var. Referencia a la variable var
                   definida a en el fichero UNO.C */
void funcion_2()
{
  var++;
  printf("%d\n", var); /* se escribe 8 */
}
```

Observe que en el programa anterior formado por los ficheros fuente *UNO.C* y *DOS.C*:

1. Existen tres declaraciones externas de *var*.
2. La variable *var*, se define e inicializa a nivel externo una sola vez.
3. La declaración **extern** en el fichero *UNO.C*, permite acceder a la variable *var*, antes de su definición. Sin la declaración **extern**, la variable global *var* no sería accesible en la función **main**.
4. La declaración **extern** en el fichero *DOS.C*, permite acceder a la variable *var* en este fichero.
5. Si la variable *var* no hubiera sido inicializada explícitamente, C le asignaría automáticamente el valor 0 por ser global.

Si se utiliza el calificador **static** en la declaración de una variable a nivel externo, ésta solamente es accesible dentro de su propio fichero fuente. Esto permite declarar otras variables **static** con el mismo nombre en otros ficheros correspondientes al mismo programa.

Como ejemplo, sustituya en los ficheros *UNO.C* y *DOS.C* del programa anterior, el calificador **extern** por **static**. Si ahora ejecuta el programa observará que la solución es 6, 7 y 1 en lugar de 6, 7 y 8, lo que demuestra que el calificador **static** restringe el acceso a la variable, al propio fichero fuente.

Variables declaradas a nivel interno

En una variable declarada a nivel interno, esto es, dentro de un bloque, se pueden utilizar cualquiera de los cuatro calificadores, u omitir el calificador, en cuyo caso se considera la variable como **auto** (local o automática).

Una variable declarada como **auto** solamente es visible dentro del bloque donde está definida. Este tipo de variables no son inicializadas automáticamente, por lo que hay que inicializarlas explícitamente, cuando sea necesario.

Una variable declarada a nivel interno como **static**, solamente es visible dentro del bloque donde está definida; pero, a diferencia de las automáticas, su existencia es permanente, en lugar de aparecer y desaparecer al iniciar y finalizar la ejecución del bloque que la contiene.

Una variable declarada **static** es inicializada solamente una vez, cuando comienza la ejecución del programa. No es reinicializada cada vez que se ejecuta el bloque que la contiene. Si la variable no es inicializada explícitamente, C la inicializa automáticamente a 0.

Una declaración **register** indica al compilador que la variable será almacenada, si es posible, en un registro de la máquina, lo que producirá programas más cortos y más rápidos. El número de registros utilizables para este tipo de variables, depende de la máquina. Si no es posible almacenar una variable **register** en un registro, se la da el tratamiento de automática. Este tipo de declaración es válido para variables de tipo **int** y de tipo *puntero*, debido al tamaño del registro.

Una variable declarada como **register** solamente es visible dentro del bloque donde está definida. Este tipo de variables no son inicializadas automáticamente, por lo que hay que inicializarlas explícitamente, si es necesario.

Una variable declarada **extern**, referencia a una variable definida con el mismo nombre a nivel externo en cualquier parte del programa. La declaración **extern** a nivel interno es utilizada para hacer accesible una variable externa, en una función o módulo en el cual no lo es.

El siguiente ejemplo clarifica lo anteriormente expuesto.

```
/* vars03.c - Variables declaradas a nivel interno
 */
#include <stdio.h>

void funcion_1();

void main()
{
  /* se hace referencia a la variable var1 */
  extern int var1;

  /* var2 es accesible solamente dentro de main.
     Su valor inicial es 0. */
  static int var2;
```

```
  /* var3 es almacenada en un registro si es posible */
  register int var3 = 0;

  /* var4 es declarada auto, por defecto */
  int var4 = 0;

  var1 += 2;

  /* se escriben los valores 7, 0, 0, 0 */
  printf("%d  %d  %d  %d\n", var1, var2, var3, var4);
  funcion_1();
}

int var1 = 5;

void funcion_1()
{
  /* se define la variable local var1 */
  int var1 = 15;

  /* var2 es accesible solamente dentro de funcion_1 */
  static var2 = 5;

  var2 += 5;

  /* se escriben los valores 15, 10 */
  printf("%d  %d\n", var1, var2);
}
```

En este ejemplo, la variable *var1* está definida a nivel externo. En la función **main** se utiliza una declaración **extern**, para hacer accesible dentro de ésta, la variable *var1*. La variable *var2* declarada **static** es inicializada, por defecto, a 0.

En la función denominada *funcion_1* se define la variable local *var1*, anulando así a la variable externa *var1*. La variable *var2*, declarada **static**, es inicializada a 5. Esta definición no entra en conflicto con la variable *var2* de la función **main**, ya que las variables **static** a nivel interno son visibles solamente dentro del bloque donde están declaradas. A continuación la variable *var2* es incrementada en 5, de tal forma que si *funcion_1* fuera llamada otra vez, el valor inicial para esta variable sería de 10, ya que las variables internas declaradas **static**, conservan los valores adquiridos durante la última ejecución.

Declaración de funciones a nivel interno y a nivel externo

Una función declarada **static** es accesible solamente dentro del fichero fuente en el que está definida.

Una función declarada **extern** es accesible desde todos los ficheros fuentes que componen un programa.

La declaración de una función **static** o **extern**, puede hacerla en la función prototipo o en la definición de la función. Una función es declarada por defecto **extern**. Por ejemplo, la siguiente declaración indica que *funcion_1* va a ser **static**.

```
static void funcion_1();
```

EJERCICIOS PROPUESTOS

1. Escriba el programa *grados.c* y compruebe cómo se ejecuta.

2. De acuerdo con lo que se expuso en el capítulo 1 acerca del depurador, pruebe a ejecutar el programa anterior paso a paso y verifique los valores que van tomando las variables a lo largo de la ejecución.

3. Modifique los límites inferior y superior de los grados centígrados, el incremento, y ejecute de nuevo el programa.

4. Modifique la sentencia

   ```
   gfahr = (float)9 / (float)5 * gcent + 32;
   ```

   y escríbala así:

   ```
   gfahr = 9 / 5 * gcent + 32;
   ```

   Explique lo que sucede.

5. Escriba el programa formado por los ficheros *modulo01.c* y *modulo02.c* y construya el fichero ejecutable.

CAPÍTULO 4

ENTRADA Y SALIDA ESTÁNDAR

El proceso de entrada obtiene datos de algún medio externo (por ejemplo, del teclado o de un fichero en disco) y los envía al ordenador para que sean procesados por un programa. Cuando se introduce un dato, se almacena en una variable en memoria. Esto quiere decir que habrá que utilizar variables en cualquier proceso de un programa.

La operación de salida envía una copia de los datos que hay en la memoria del ordenador a otro lugar; por ejemplo, los visualiza en el monitor, los escribe por la impresora o los guarda en un fichero en disco. La operación de salida no borra los datos de la memoria ni cambia la forma en la que están almacenados. Simplemente hace una copia de los mismos para enviarlos a otro lugar.

En este capítulo estudiaremos, además de la sentencia de asignación, las funciones de la biblioteca de C que permiten realizar operaciones de entrada y de salida sobre los dispositivos estándar del ordenador; esto es, funciones para introducir datos desde el teclado y funciones para visualizar datos por el monitor.

SINTAXIS DE LAS SENTENCIAS Y FUNCIONES DE C

Para presentar los formatos de las sentencias, macros y funciones de C, se aplicarán las mismas reglas enunciadas al principio del capítulo 2.

Cuando se trate de presentar la sintaxis correspondiente a una macro o a una función, se dará la siguiente información:

1. Fichero de cabecera que contiene las declaraciones y definiciones relativas a esa función y afines.

2. Prototipo de la función para indicar el tipo del resultado y el número y tipo de los argumentos.
3. Compatibilidad (ANSI, UNIX, MS-DOS).

Por ejemplo, la sintaxis de la función **sqrt** (raíz cuadrada) es:

```
#include <math.h>          fichero de declaraciones y definiciones
double sqrt(double x);     prototipo de la función
Compatibilidad: ANSI, UNIX y MS-DOS
```

SENTENCIA DE ASIGNACIÓN

Una sentencia de asignación tiene la forma:

variable operador-de-asignación expresión

La sentencia de asignación es asimétrica. Esto quiere decir que se evalúa la expresión de la derecha y el resultado se asigna a la variable especificada a la izquierda. Por ejemplo,

```
total = 0;
area = 3.141592 * r * r;
cuenta += 1;
```

Según lo expuesto, la siguiente sentencia no sería válida:

```
3.141592 * r * r = area;
```

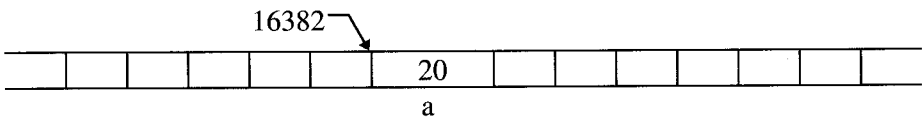
Si la variable es de tipo *puntero*, solamente se le puede asignar una dirección de memoria, la cual será siempre distinta de 0. Un valor 0 (se simboliza con **NULL**) sirve para indicar que esa variable puntero no apunta a un dato válido. Por ejemplo:

```
int a = 10, *p;
p = &a;          /* se asigna a p la dirección de a */
```

No tiene sentido asignar un entero a una variable de tipo puntero.

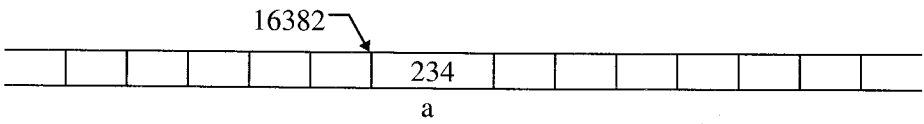
Cuando se asigna un valor a una variable estamos colocando el valor en una localización de memoria asociada con esa variable.

```
a = 20;
```



Lógicamente, cuando la variable tiene asignado un valor y se le asigna uno nuevo, el valor anterior es destruido ya que el valor nuevo pasa a ocupar la misma localización de memoria.

```
a = 234;
```



ENTRADA Y SALIDA ESTÁNDAR

Las operaciones de entrada y de salida (E/S) no forman parte del conjunto de sentencias de C, sino que pertenecen al conjunto de funciones de la biblioteca estándar de C. Por ello, todo fichero fuente que utilice funciones de E/S correspondientes a la biblioteca estándar de C, necesita de los prototipos de las funciones correspondientes a éstas, por lo que deberá contener la línea:

```
#include "stdio.h"
```

Las dobles comillas significan que el fichero especificado, debe ser buscado en el directorio actual de trabajo y si no se encuentra, la búsqueda debe continuar en el directorio estándar para los ficheros con extensión *.h* (directorio *include*).

Si el fichero de cabecera especificado, en lugar de escribirlo entre comillas, lo escribimos entre ángulos:

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

la búsqueda de dicho fichero se efectúa solamente en el directorio estándar para los ficheros con extensión *.h* (directorio *include*).

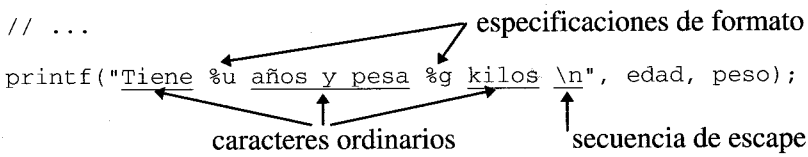
SALIDA CON FORMATO

La función **printf** escribe utilizando el formato especificado, una serie de caracteres en el fichero estándar de salida referenciado por **stdout**. Esta función devuelve un valor entero igual al número de caracteres escritos.

```
#include <stdio.h>
int printf(const char *formato[, argumento]...);
Compatibilidad: ANSI, UNIX y MS-DOS
```

formato Especifica cómo va a ser la salida. Es una cadena de caracteres formada por caracteres ordinarios, secuencias de escape y especificaciones de formato. El formato se lee de izquierda a derecha.

```
unsigned int edad = 0;
float peso = 0;

// ...
printf("Tiene %u años y pesa %g kilos \n", edad, peso);
```



argumento Representa el valor o valores a escribir. Cada argumento debe tener su correspondiente especificación de formato y en el mismo orden. Si hay más argumentos que especificaciones de formato, los argumentos en exceso se ignoran.

```
printf("Tiene %u años y pesa %g kilos\n", edad, peso);
```

Cuando se ejecute la sentencia anterior, los caracteres ordinarios se escribirán tal cual, las especificaciones de formato serán sustituidas por los valores correspondientes a la lista de argumentos y las secuencias de escape darán lugar al carácter o acción que representan. Así, para *edad* igual a 20 y *peso* igual 70.5 el resultado será:

```
Tiene 20 años y pesa 70.5 kilos↵
```

Una especificación de formato está compuesta por:

```
%[flags][ancho][.precisión][{h|l|L}]tipo
```

Una especificación de formato siempre comienza con **%**. El significado de cada uno de los elementos se indica a continuación.

flags	*significado*
–	Justifica el resultado a la izquierda, dentro del *ancho* especificado. Por defecto la justificación se hace a la derecha.
+	Antepone el signo + o – al valor de salida. Por defecto sólo se pone signo – a los valores negativos.

0	Rellena la salida con ceros no significativos hasta alcanzar el ancho mínimo especificado.
blanco	Antepone un espacio en blanco al valor de salida si es positivo. Si se utiliza junto con + entonces se ignora.
#	Cuando se utiliza con la especificación de formato **o**, **x**, o **X**, antepone al valor de salida **0**, **0x**, o **0X**, respectivamente. Cuando se utiliza con la especificación de formato **e**, **E**, o **f**, fuerza a que el valor de salida contenga un punto decimal en todos los casos. Cuando se utiliza con la especificación de formato **g**, o **G**, fuerza a que el valor de salida contenga un punto decimal en todos los casos y evita que los ceros arrastrados sean truncados. Se ignora con **c**, **d**, **i**, **u**, o **s**.
ancho	Mínimo número de posiciones para la salida. Si el valor a escribir ocupa más posiciones de las especificadas, el ancho es incrementado en lo necesario.
precisión	El significado depende del tipo de la salida.
tipo	es uno de los siguientes caracteres:
carácter	*salida*
d	(**int**) enteros con signo, en base 10.
i	(**int**) enteros con signo, en base 10.
u	(**int**) enteros sin signo, en base 10.
o	(**int**) enteros sin signo, en *base* 8.
x	(**int**) enteros sin signo, en base 16 (01...abcdef).
X	(**int**) enteros sin signo, en base 16 (01...ABCDEF).
f	(**double**) valor con signo de la forma: [−]*dddd.dddd*. El número de dígitos antes del punto decimal depende de la magnitud del número y el número de decimales de la precisión, la cual es 6 por defecto.
e	(**double**) valor con signo, de la forma [−]*d.dddd*e[±]*ddd*
E	(**double**) valor con signo, de la forma [−]*d.dddd*E[±]*ddd*
g	(**double**) valor con signo, en formato **f** o **e** (el que sea más compacto para el valor y precisión dados).
G	(**double**) igual que **g**, excepto que **G** introduce el exponente **E** en vez de **e**.

c	(**int**) un sólo carácter, correspondiente al byte menos significativo.
s	(*cadena de caracteres*) escribir una cadena de caracteres hasta el primer carácter nulo ('\0').
n	(*puntero a un entero*). En el entero correspondiente a esta especificación de formato, se almacena el número de caracteres hasta ahora escritos en el *buffer*.
p	(*puntero a void*). Válido sólo en MS-DOS. Escribe la dirección apuntada por el argumento. Si se especifica **%p** o **%Np** se escribe sólo el desplazamiento de la dirección y si se especifica **%Fp** o **%lp** se escribe una dirección segmentada (*xxxx:yyyy*). En este último caso se espera un puntero *far* a un valor, por ello bajo el *modelo small* (opción elegida al compilar) hay que utilizar con el argumento a escribir, la construcción *cast*: *(tipo* **far** **)arg*.

El siguiente ejemplo clarifica lo más significativo de lo expuesto hasta ahora.

```
#include <stdio.h>

void main()
{
  int i = 0, a = 12345;
  float b = 54.865;

  printf("%d\n%n", a, &i); /* hasta ahora hay en el buffer seis
                              caracteres: 12345\n */
  printf("%d\n", i);       /* escribe 6 */
  printf("\n%10s\n%10s\n", "abc", "abcdef");
  printf("\n%-10s\n%-10s\n", "abc", "abcdef");

  printf("\n");            /* avanzar una línea */
  printf("%.2f\n", b);     /* escribir b con dos decimales */
}
```

Al ejecutar este programa se obtendrán los siguientes resultados:

```
12345
6
(línea en blanco)
       abc
    abcdef
(línea en blanco)
abc
abcdef
(línea en blanco)
54.87
```

A continuación damos una explicación de cada uno de los formatos empleados. La línea

```
printf("%d\n%n", a, &i); /* hasta ahora hay en el buffer seis
                            caracteres: 12345\n */
```

utiliza un formato compuesto por:

- *%d* para escribir el entero *a* (*d* indica base decimal).
- *\n* para avanzar a la línea siguiente.
- *%n* para almacenar en el entero *i*, el número de caracteres hasta ahora escritos. Observe que con este formato el argumento no es el entero sino la dirección del entero, *&i*.

La línea: `printf("%d\n", i);`

utiliza un formato compuesto por:

- *%d* para escribir el valor del entero *i* (*d* indica base decimal).
- *\n* para avanzar a la línea siguiente.

La línea: `printf("\n%10s\n%10s\n", "abc", "abcdef");`

utiliza un formato compuesto por:

- *\n* para avanzar a la línea siguiente.
- *%10s* para escribir la cadena *"abc"* sobre un ancho de 10 posiciones. La cadena se ajusta por defecto a la derecha.
- *\n* para avanzar a la línea siguiente.
- *%10s* para escribir la cadena *"abcdef"* sobre un ancho de 10 posiciones. La cadena se ajusta por defecto a la derecha.
- *\n* para avanzar a la línea siguiente.

La línea: `printf("\n%-10s\n%-10s\n", "abc", "abcdef");`

utiliza un formato igual que el anterior, con la diferencia de que ahora se ha añadido el *flag* – después del carácter de formato para ajustar las cadenas a la izquierda.

La línea: `printf("\n");`

utiliza un formato compuesto por:

- *\n* para avanzar a la línea siguiente.

La línea: `printf("%.2f\n", b);`

utiliza un formato compuesto por:

- *%.2f* para escribir el valor del real *b* (*f* representa a un valor real, **float** o **double**). El formato no especifica el ancho, lo que significa que se utilizarán tantas posiciones como se necesiten para visualizar el resultado, pero sí se especifica el número de decimales, dos. Para escribir la parte decimal se truncan las cifras decimales que sobran y se redondea el resultado; esto es, si la primera cifra decimal truncada es cinco o mayor, la anterior se incrementa en una unidad.
- \n para avanzar a la línea siguiente.

En una especificación de formato, el *ancho* y/o la *precisión* pueden ser sustituídos por un *. Si el *ancho* y/o la *precisión* se especifican con el carácter *, el valor para estos campos se toma del siguiente argumento entero. Por ejemplo:

```
int ancho = 15, precision = 2;
double valor = -12.346;
printf("%*.*f", ancho, precision, valor);
```

									–	1	2	.	3	5

El resultado queda justificado por defecto a la derecha en un ancho de 15 posiciones, de las cuales dos de ellas son decimales. Para que el resultado hubiera quedado justificado a la izquierda tendríamos que haber utilizado el formato:

```
printf("%-*.*f", ancho, precision, valor);
```

Continuando con la explicación, la *precisión*, en función del tipo, tiene el siguiente significado:

Tipo	*Significado de la precisión*
d,i,u,o,x,X	Especifica el mínimo número de dígitos que se tienen que escribir. Si es necesario se rellena con ceros a la izquierda. Si el valor excede de la precisión, no se trunca.
e,E,f	Especifica el número de dígitos que se tienen que escribir después del punto decimal. Por defecto es 6. El valor es redondeado.
g,G	Especifica el máximo número de dígitos significativos (por defecto 6) que se tienen que escribir.
c	La precisión no tiene efecto.
s	Especifica el máximo número de caracteres que se escribirán. Los caracteres que excedan este número, se ignoran.

h Se utiliza como prefijo con los tipos **d**, **i**, **o**, **x**, y **X**, para especificar que el argumento es **short int**, o con **u** para especificar un **short unsigned int**.

l Se utiliza como prefijo con los tipos **d**, **i**, **o**, **x**, y **X**, para especificar que el argumento es **long int**, o con **u** para especificar un **long unsigned int**. También se utiliza con los tipos **e**, **E**, **f**, **g**, y **G** para especificar un **double** antes que un **float**.

L Se utiliza como prefijo con los tipos **e**, **E**, **f**, **g**, y **G**, para especificar **long double**.

Las siguientes sentencias muestran algunos ejemplos de cómo utilizar la función **printf.**

```
#include <stdio.h>

void main()
{
  char car;
  static char nombre[] = "La temperatura ambiente";
  int a, b, c;
  float x, y, z;

  car = 'C'; a = 20; b = 350; c = 1994;
  x = 34.5; y = 1234; z = 1.248;

  printf("\n%s es de ", nombre);
  printf("%d grados %c\n", a, car);
  printf("\n");
  printf("a = %6d\tb = %6d\tc = %6d\n", a, b, c);

  printf("\nLos resultados son los siguientes:\n");
  printf("\n%5s\t\t%5s\t\t%5s\n", "x", "y", "z");
  printf("_____________________________________________\n");
  printf("\n%8.2f\t%8.2f\t%8.2f", x, y, z);
  printf("\n%8.2f\t%8.2f\t%8.2f\n", x+y, y/5, z*2);
  printf("\n\n");
  z *= (x + y);
  printf("Valor resultante: %.3f\n", z);
}
```

Como ejercicio, escriba los resultados que se tienen que visualizar cuando se ejecute el programa anterior. Recuerde que \t es una secuencia de escape que da lugar a un tabulador horizontal. El resultado correcto se muestra a continuación.

```
La temperatura ambiente es de 20 grados C

a =     20      b =    350      c =   1995

Los resultados son los siguientes:
```

```
    x              y              z
-------------------------------------

  34.50        1234.00           1.25
1268.50         246.80           2.50

Valor resultante: 1583.088
```

ENTRADA CON FORMATO

La función **scanf** lee datos de la entrada estándar referenciada por **stdin**, los interpreta de acuerdo con el formato indicado y los almacena en los argumentos especificados. Cada argumento debe ser un puntero a una variable cuyo tipo debe corresponderse con el tipo especificado en el formato; dicho de otra forma, el argumento no es la variable, sino la dirección de la variable.

Esta función devuelve un entero correspondiente al número de datos leídos y asignados de la entrada. Si este valor es 0, significa que no han sido asignados datos. Cuando se intenta leer un carácter fin de fichero (*end-of-file* - marca de fin de fichero) la función **scanf** retorna la constante **EOF**, definida en el fichero *stdio.h*. Más adelante, en este mismo capítulo, explicaremos este último concepto.

```
#include <stdio.h>
int scanf(const char *formato[, argumento]...);
Compatibilidad: ANSI, UNIX y MS-DOS
```

formato Interpreta cada dato de entrada. Está formado por caracteres que genéricamente se denominan espacios en blanco (' ', \t, \n), caracteres ordinarios y especificaciones de formato. El formato se lee de izquierda a derecha.

Cada argumento debe tener su correspondiente especificación de formato y en el mismo orden (vea también, la función **printf**).

Si un carácter en la entrada estándar no se corresponde con la entrada especificada por el formato, se interrumpe la entrada de datos.

argumento Es un puntero a la variable que se quiere leer.

Cuando se especifica más de un argumento, los valores correspondientes en la entrada hay que separarlos por uno o más espacios en blanco (' ', \t, \n) o por el carácter que se especifique en el formato.

Un espacio en blanco antes o después de una especificación de formato hace que **scanf** lea, pero no almacene, todos los caracteres espacio en blanco, hasta en-

contrar un carácter distinto de espacio en blanco. Por ejemplo, siendo *a* de tipo **int**, *b* de tipo **float** y *c* de tipo **char**, vamos a analizar el comportamiento de las sentencias de la tabla siguiente:

sentencia	*entrada de datos*
`scanf("%d %f %c", &a, &b, &c);`	`5 23.4 z↵`
`scanf("%d , %f , %c", &a, &b, &c);`	`5, 23.4 , z↵`
`scanf("%d : %f : %c", &a, &b, &c);`	`5:23.4 : z↵`

La primera sentencia leerá del teclado un valor entero (*%d*) para *a*, un valor real (*%f*) para *b* y un carácter (*%c*) para *c*. Observe que las especificaciones de formato ("*%d %f %c*") van separadas por un espacio en blanco; esto es lo más habitual. Quiere esto decir, que los datos tecleados para las variables citadas tienen que introducirse separados por al menos un espacio en blanco. Si se utiliza otro separador, como ocurre en las dos sentencias siguientes, entonces en la entrada de datos se utilizará ese separador. Es conveniente que cuando utilice otro separador ponga en la especificación de formato un espacio antes y otro después de él. Esto le permitirá en la entrada separar los datos por el separador especificado seguido y precedido, si lo desea, por espacios en blanco.

Las especificaciones de formato que no incluyan espacios en blanco como separadores, no son aconsejables por ser muy rígidas en su uso. Por ejemplo,

sentencia	*entrada de datos*
`scanf("%d%f%c", &a, &b, &c);`	`5 23.4z↵`
`scanf("%d,%f,%c", &a, &b, &c);`	`5,23.4,z↵`

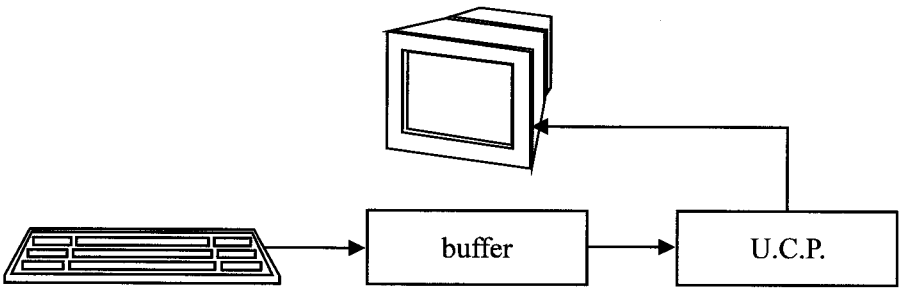
La primera sentencia del ejemplo anterior no incluye separadores entre las especificaciones de formato. Esto hace que en la entrada los valores numéricos, sí tengan que ser separados por al menos un espacio en blanco para diferenciar uno de otro, pero no sucede lo mismo con el carácter '*z*'. Si se pone un espacio en blanco antes de *z*, a la variable *c* se le asignará el espacio en blanco y no la *z*. La segunda sentencia incluye como separador el carácter coma sin espacios en blanco. Por lo tanto, por la misma razón expuesta anteriormente, no podemos escribir un espacio en blanco antes del carácter *z*.

Cuando se ejecuta una sentencia de entrada, por ejemplo

```
scanf("%d %f %c", &a, &b, &c);
```

la ejecución del programa se detiene hasta que escribamos los datos que hay que introducir. Los datos que se escriben desde el teclado no son inmediatamente

asignados a las variables especificadas y procesados; piense que si esto sucediera no tendríamos opción a corregir un dato equivocado. Realmente, los datos se escriben en un *buffer* o memoria intermedia asociada con el *stream* **stdin**, que a su vez está ligado con el teclado, y son enviados a la unidad central de proceso cuando se pulsa la tecla *Entrar* (↵) para ser asignados a las variables y ser procesados. Esos datos, según se escriben, son visualizados en el monitor con el fin de ver lo que estamos haciendo.



Según lo expuesto, lo que realmente hay en el *buffer* es una cadena de caracteres. Para diferenciar un dato de otro dentro de la cadena, utilizamos los caracteres genéricamente denominados espacios en blanco (espacio en blanco, nueva línea, etc.). Por ejemplo, cuando se ejecuta una sentencia como

```
scanf("%d %f %c", &a, &b, &c);
```

lo que hacemos es escribir una cadena de caracteres similar a la siguiente:

```
5 23.4 z↵
```

y finalizarla con *Entrar*. Dicha cadena también podría introducirse así:

```
5↵
23.4↵
z↵
```

La diferencia es que ahora el separador es el carácter nueva de línea, en lugar del un espacio en blanco.

Cuando se pulsa *Entrar*, lo que hace **scanf** es leer caracteres del *buffer* y convertirlos según el formato de la variable donde hay que almacenarlos. La asignación a una variable finaliza cuando se llega a un separador. Este proceso se repite para cada una de las variables especificadas.

La ejecución de la función **scanf** finaliza cuando se han asignado valores a todas las variables o cuando se lee un carácter que no se corresponde con la entrada especificada por el formato. Por ejemplo, si introducimos los datos

```
cinco 23.4 z↵
```

la ejecución de **scanf** se interrumpe porque el formato *%d* espera un carácter válido para formar un entero y *'c'* no lo es. El resultado es que no se asigna ningún valor y la ejecución continua en la siguiente sentencia del programa, con los valores que tengan por defecto las variables *a*, *b* y *c*. Si introducimos los datos

```
5 tm 23.4 z↵
```

la ejecución de **scanf** se interrumpe porque el formato *%f* espera un carácter válido para formar un real y *'t'* no lo es. El resultado es que *a* vale 5 y no se asigna ningún valor ni a *b* ni a *c*. La ejecución continua en la siguiente sentencia del programa, con los valores que tengan las variables *a*, *b* y *c*. Si introducimos los datos

```
5 23,4 z↵
```

se asigna a la variable *a* el entero 5, a *b* el real 23 y a *c* la coma (','). El resultado final será inesperado porque no eran estos los valores que deseábamos leer.

Recuerde que la función **scanf** devuelve el número de datos leídos y asignados. Esto es, si escribimos las sentencias

```
int a, r; float b; char c;
// ...
r = scanf("%d %f %c", &a, &b, &c);
```

el valor de *r* será 0, 1, 2 o 3. Por ejemplo,

Entrada	*valor de r*
`5 23.4 z↵`	3
`cinco 23.4 z↵`	0
`5 tm 23.4 z↵`	1
`5 23,4 z↵`	3

No se puede escribir una sentencia como la siguiente:

```
scanf("Introducir los valores de a, b y c: %d %f %c", &a, &b, &c);
```

ya que esta forma de proceder no visualizaría el mensaje especificado, sino que obligaría a escribir la cadena "*Introducir los valores de a, b y c:* " como separador, antes de escribir el valor de *a*. Si lo que quiere es visualizar este mensaje para informar al usuario de lo que tiene que hacer, proceda así:

```
printf("Introducir los valores de a, b y c: ");
scanf("%d %f %c", &a, &b, &c);
```

Lógicamente, cuando una variable tiene asignado un valor y utilizando una sentencia de entrada se le asigna uno nuevo, el valor anterior es destruido porque el nuevo valor pasa a ocupar la misma localización de memoria.

Una especificación de formato está compuesta por:

```
%[*][ancho][{h/l}]tipo
```

Una especificación de formato siempre comienza con **%**. El resto de los elementos que puede especificar se explican a continuación:

***** Un *asterisco* a continuación del símbolo **%** suprime la asignación del siguiente dato en la entrada. Por ejemplo,

```
scanf("%d %*s %d %*s", &horas, &minutos);
```

Para una entrada como *12 horas 30 minutos*↵ el resultado es: *horas=12* y *minutos=30*. Las cadenas "horas" y "minutos" especificadas después de los valores 12 y 30, no se asignan.

ancho Máximo número de caracteres a leer de la entrada. Los caracteres en exceso no se tienen en cuenta.

h Se utiliza como prefijo con los tipos **d**, **i**, **n**, **o** y **x** para especificar que el argumento es **short int**, o con **u** para especificar que es **short unsigned int**.

l Se utiliza como prefijo con los tipos **d**, **i**, **n**, **o** y **x** para especificar que el argumento es **long int**, o con **u** para especificar que es **long unsigned int**. También se utiliza con los tipos **e**, **f** y **g** para especificar que el argumento es **double**.

tipo El tipo determina cómo tiene que ser interpretado el dato de entrada: como un carácter, como una cadena de caracteres o como un número. El formato más simple contiene el símbolo **%** y el *tipo*. Por ejemplo, **%i**. Los tipos que puede utilizar son los siguientes:

Carácter	*El argumento es un puntero a*	*Entrada esperada*
d	**int**	enteros con signo, en base 10.
o	**int**	enteros con signo, en base 8.
x, X	**int**	enteros con signo, en base 16.
i	**int**	enteros con signo en base 10, 16 u 8. Si el entero comienza con **0** se toma el valor en octal y si empieza con **0x** o **0X** el valor se toma en hexadecimal.
u	**unsigned int**	enteros sin signo, en base 10.
f **e, E** **g, G**	**float**	valor con signo de la forma [–]*d.dddd*[{e|E}[±]*ddd*]
c	**char**	un solo carácter.
s	**char**	cadena de caracteres.

n	**int**	en el entero es almacenado el número de caracteres leídos del *buffer* o del fichero. Por ejemplo:
		`long a; int r;`
		`scanf("%ld%n", &a, &r);`
		`printf("Caracteres leídos: %ld\n", r);`

p	*puntero a* **void**	(sólo para MS-DOS) lee una dirección de la forma *xxxx:yyyy* expresada en dígitos hexadecimales en mayúsculas y la almacena en el argumento. Por ejemplo:
		`int *a;`
		`scanf("%p", &a);`

Las siguientes sentencias muestran algunos ejemplos de cómo utilizar la función **scanf.**

```
#include <stdio.h>

void main()
{
  int a, r; float b; char c;

  printf("Introducir un valor entero, un real y un char\n=>");
  r = scanf("%d %f %c", &a, &b, &c);
  printf("\nNúmero de datos leídos: %d\n", r);
  printf("Datos leídos: %d %f %c\n", a, b, c);
```

```
  printf("\nValor hexadecimal: ");
  scanf("%i", &a);
  printf("Valor decimal:     %i\n", a);
}
```

Cuando ejecute este programa, el resultado que se visualizará será así:

```
Introducir un valor entero, un real y un char
=>1880 3.14159 z

Número de datos leídos: 3
Datos leídos: 1880 3.141590 z

Valor hexadecimal: -0x100
Valor decimal:     -256
```

Con la especificación de formato **%c**, se puede leer cualquier carácter, incluidos los caracteres denominados genéricamente espacios en blanco (' ', \t, \n). Por ejemplo, si en un instante determinado durante la ejecución de un programa quiere hacer una pausa, puede intercalar las dos sentencias siguientes:

```
printf("Pulse <Entrar> para continuar ");
scanf("%c", &car);
```

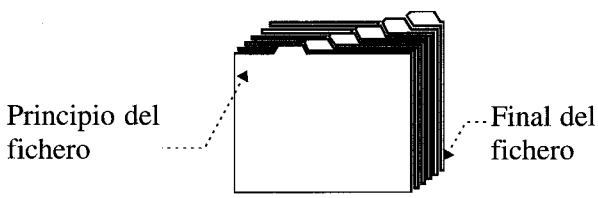
Cuando se ejecuten estas sentencias se visualizará el mensaje,

```
Pulse <Entrar> para continuar
```

y se hará una pausa mientras **scanf** no tenga un carácter en el *buffer* de entrada que leer. Cuando pulsemos la tecla *Entrar* (↵) habremos enviado el carácter nueva línea al *buffer* de entrada, que será leído por la función **scanf** y asignado a la variable *car*, reanudándose la ejecución del programa.

CARÁCTER FIN DE FICHERO

Los dispositivos de entrada y de salida estándar (teclado y monitor) son tratados por el lenguaje C como si de un fichero de datos en el disco se tratara. Un fichero de datos es una colección de información. Los datos que introducimos por el teclado son una colección de información y los datos que visualizamos en el monitor son también una colección de información.



Todo fichero tiene un principio y un final. ¿Cómo sabe un programa que está leyendo datos de un fichero, que se ha llegado al final del mismo y por lo tanto no hay más datos? Por una marca de fin de fichero. En el caso de un fichero grabado en un disco esa marca estará escrita al final del mismo. En el caso del teclado la información procede de lo que nosotros tecleamos, por lo tanto si nuestro programa requiere detectar la marca de fin de fichero, tendremos que teclearla cuando demos por finalizada la introducción de información. Esto se hace pulsando las teclas *Ctrl+D* en UNIX o *Ctrl+Z* en MS-DOS.

Recuerde que cuando la función **scanf** intenta leer un carácter fin de fichero, retorna la constante **EOF** definida en el fichero *stdio.h*. El siguiente ejemplo escribe el mensaje "Fin de la entrada de datos" si al mensaje "Precio: " respondemos, pulsando las teclas correspondientes, con el carácter *fin de fichero* seguido de la tecla *Entrar*.

```
#include <stdio.h>
void main()
{
  int r = 0;
  float precio = 0;

  printf("Precio: ");
  r = scanf("%g", &precio);

  (r == EOF) ? printf("Fin de la entrada de datos\n")
             : printf("%g\n", precio);
}
```

En capítulos posteriores utilizaremos el carácter fin de fichero como condición para finalizar la entrada de un número de datos, en principio indeterminado.

Indicador de fin de fichero

Cuando se detecta el final de un fichero el sistema activa un indicador de fin de fichero, asociado con el *stream* ligado con ese fichero, para notificarlo al programa. Cualquier intento de lectura posterior sobre ese fichero será fallido, lo que será notificado por el valor retornado por la función que se halla utilizado para leer. Por ejemplo,

```
#include <stdio.h>
void main()
{
  int r = 0, opcion;
  float precio = 0;

  printf("Precio: ");
  r = scanf("%g", &precio);
```

```
  (r == EOF) ? printf("Fin de la entrada de datos\n")
             : printf("%g\n", precio);

  printf("Opción: ");
  scanf("%d", &opcion);
}
```

Cuando ejecute este programa, si responde al mensaje "Precio: " con el carácter fin de fichero, la siguiente llamada a **scanf** no esperará por una entrada para *opcion* y la función **scanf** devolverá el valor **EOF**.

Para desactivar el indicador de fin de fichero y cualquier otro indicador de error se utiliza la función de la biblioteca de C **clearerr**. A continuación se indica la sintaxis de esta función. Más adelante entenderá el tipo de parámetro que requiere. Ahora limítese a ver cómo se utiliza.

```
#include <stdio.h>
void clearerr(FILE *stream);
```
Compatibilidad: ANSI, UNIX y MS-DOS

Según lo expuesto, para desactivar el indicador de fin de fichero (en nuestro caso, asociado con **stdin**) modificaremos el programa anterior así:

```
#include <stdio.h>
void main()
{
  int r = 0, opcion;
  float precio = 0;

  printf("Precio: ");
  r = scanf("%g", &precio);

  (r == EOF) ? printf("Fin de la entrada de datos\n")
             : printf("%g\n", precio);

  clearerr(stdin);
  printf("Opción: ");
  scanf("%g", &opcion);
}
```

Como ya hemos indicado anteriormente, esta forma de proceder tiene sentido cuando utilicemos el carácter fin de fichero como condición para finalizar la entrada de un número de datos en principio indeterminado, lo que veremos en capítulos posteriores.

Si cuando se ejecute la función **clearerr** no hay ningún indicador activado, no ocurre ninguna acción y la ejecución del programa continua normalmente. Los indicadores de error son automáticamente desactivados cuando finaliza la ejecución del programa.

CARÁCTER NUEVA LÍNEA

Cuando se están introduciendo datos a través del teclado y pulsamos la tecla *Entrar* (↵) se introduce también el carácter denominado *nueva línea*, que en C se representa por medio de la secuencia de escape \n. Por ejemplo, el programa siguiente lee un número entero:

```
#include <stdio.h>
void main()
{
  float precio = 0;

  printf("Precio: ");
  scanf("%g", &precio);
  printf("Precio = %g\n", precio);
}
```

Cuando se ejecute la función **scanf** del programa anterior, si tecleamos:

```
1000↵
```

antes de la lectura, habrá en el *buffer* de entrada la siguiente información:

1	0	0	0	\n										

y después de la lectura,

\n														

ya que el carácter nueva línea no es un carácter válido para la especificación de formato *%g*; por lo tanto, aquí se interrumpe la lectura. Este carácter sobrante puede ocasionarnos problemas si a continuación se ejecuta otra sentencia de entrada que admita datos que sean caracteres. Por ejemplo,

```
#include <stdio.h>
void main()
{
  float precio = 0;
  char car = 0;

  printf("Precio: ");
  scanf("%g", &precio);

  printf("Pulse <Entrar> para continuar ");
  scanf("%c", &car);

  printf("Precio = %g\n", precio);
}
```

Si ejecutamos este programa y tecleamos el dato 1000, se producirá el siguiente resultado:

```
Precio: 1000
Pulse <Entrar> para continuar Precio = 1000
```

A la vista del resultado, se observa que no se ha hecho una pausa ¿Por qué? Porque el carácter sobrante nueva línea es un carácter válido para la especificación de formato *%c*, razón por la que **scanf** no necesita esperar a que introduzcamos un carácter para la variable *car*.

Hay varias soluciones al problema planteado. La primera se deduce directamente de la exposición hecha anteriormente para la función **scanf**, donde decíamos: "un espacio en blanco antes o después de una especificación de formato hace que **scanf** lea, pero no almacene, todos los caracteres espacio en blanco, hasta encontrar un carácter distinto de espacio en blanco". También hemos dicho en el capítulo 2 que el carácter nueva línea se comporta como un espacio en blanco, porque hace de separador.

Según lo expuesto, bastará con introducir un espacio antes de la especificación de formato *%c*,

```
scanf(" %c", &car);
```

pero esto nos obliga a introducir un carácter que no se comporte como un espacio en blanco y después pulsar *Entrar*. Por lo tanto procederemos de otra forma. Al explicar la especificación de formato para **scanf** dijimos que un * a continuación del símbolo % suprime la asignación del siguiente dato en la entrada. Según esto, la sentencia siguiente resolverá el problema surgido.

```
scanf("%*c%c", &car);
```

Otra solución es limpiar el buffer de la entrada estándar.

Limpiar el buffer de la entrada estándar

Para limpiar el *buffer* asociado con la entrada estándar (*stream* **stdin**) hay que utilizar la función de la biblioteca de C **fflush**. A continuación se indica la sintaxis de esta función. Más adelante entenderá el tipo de parámetro que requiere. Ahora limítese a ver cómo se utiliza.

```
#include <stdio.h>
int fflush(FILE *stream);
Compatibilidad: ANSI, UNIX y MS-DOS
```

Cuando el *stream* está asociado con un fichero de entrada, como ocurre con **stdin** que está ligado con el teclado, la función **fflush** simplemente limpia el *buffer* (en algunos compiladores C en UNIX, **fflush** no debe emplearse con ficheros de entrada). Cuando el *stream* está ligado con un fichero de salida, **fflush** escribe el contenido del *buffer* en el fichero y limpia el *buffer*.

Según lo expuesto, el problema anterior podría resolverse también así:

```
#include <stdio.h>
void main()
{
  float precio = 0;
  char car = 0;

  printf("Precio: ");
  scanf("%g", &precio);
  fflush(stdin);
  printf("Pulse <Entrar> para continuar ");
  scanf("%c", &car);
  printf("Precio = %g\n", precio);
}
```

La función **fflush** retorna un valor 0 si se ejecuta satisfactoriamente o el valor **EOF** si ocurre un error.

Un *buffer* es automáticamente limpiado cuando está lleno, cuando se cierra el *stream* o cuando el programa finaliza normalmente.

LEER UN CARÁCTER DE LA ENTRADA ESTÁNDAR

Para leer un carácter de la entrada estándar (**stdin**) C proporciona la función **getchar**. Cada vez que se ejecute la función **getchar** se leerá el siguiente carácter al último leído. La sintaxis para esta función es:

```
#include <stdio.h>
int getchar(void);
Compatibilidad: ANSI, UNIX y MS-DOS
```

La función **getchar** devuelve el carácter leído, o un **EOF** si se detecta el final del fichero o si ocurre un error.

Observe que la función no tiene argumentos, por lo el carácter leído se corresponde con el valor **int** devuelto por la función. Por ejemplo:

```
car = getchar();  /* lee un carácter y lo almacena en
                     la variable car */
```

Suponiendo que el *buffer* de entrada está limpio, cuando se ejecute la sentencia anterior, la ejecución del programa se detendrá hasta que introduzcamos un carácter y pulsemos la tecla ↵. El carácter leído será almacena en la variable *car*.

Esta sentencia es equivalente a

```
scanf("%c", &car);
```

por lo que todo lo expuesto para **scanf** con respecto a los caracteres nueva línea y fin de fichero, también es aplicable a **getchar**.

ESCRIBIR UN CARÁCTER EN LA SALIDA ESTÁNDAR

Para escribir un carácter en la salida estándar (**stdout**) C proporciona la función **putchar**. Cada vez que se ejecute la función **putchar** se escribirá en el monitor un carácter a continuación del último escrito. La sintaxis para esta función es:

```
#include <stdio.h>
int putchar(int c);
Compatibilidad: ANSI, UNIX y MS-DOS
```

La función **putchar** devuelve el carácter escrito, o un **EOF** si ocurre un error.

Observe que la función tiene un argumento de tipo entero que almacena el carácter que se quiere escribir. Por ejemplo:

```
putchar('\n')   /* avanza a la siguiente línea */
putchar(car);   /* escribe el carácter contenido en la
                   variable car */
```

Las sentencias anteriores son equivalentes a

```
printf("\n");
printf("%c", car);
```

FUNCIONES getche y getch

La función **getch** lee un carácter del teclado, sin visualizarlo en el monitor (sin eco); la función **getche** lee un carácter del teclado visualizándolo en el monitor (con eco). La sintaxis para estas funciones es la siguiente:

```
#include <conio.h>
int _getch(void);
int _getche(void);
Compatibilidad: MS-DOS
```

Ambas funciones leen un carácter del *buffer* asociado con el teclado. Cuando se ejecuta una función de éstas, la ejecución se detiene hasta que se pulse una tecla. No es necesario pulsar *Entrar* (↵). El resultado es *un byte* cuando la tecla pulsada se corresponde con uno de los caracteres de la tabla de códigos de caracteres ASCII; por ejemplo, la tecla *A* da lugar a un byte correspondiente al carácter 'a' o 'A'. El resultado son *dos bytes* cuando la tecla o combinación de teclas pulsadas se corresponden con alguna de las especificadas en la tabla de los códigos extendidos que puede ver en los apéndices; por ejemplo, *F1* produce dos bytes, el primero es cero y el segundo es el que identifica a esta tecla. Para este último caso, hay que llamar a la función dos veces, ya que es la segunda llamada, la que proporciona el código deseado (segundo código).

El siguiente ejemplo hace que la ejecución se detenga cuando se ejecute *_getche* y continúe después de pulsar una tecla, la cual será visualizada en el monitor.

```
printf("Pulse una tecla para continuar ");
_getche();
```

El siguiente ejemplo, almacena en la variable *byte2*, el código extendido de la tecla de función, tecla de movimiento del cursor, combinación de teclas etc., que se pulse (vea los ejemplos de la tabla siguiente).

```
char byte1, byte2;
printf("pulse la combinación de teclas cuyo código\n"
       "extendido desea conocer\n");
byte1 = _getch(); byte2 = _getch();
printf("%d \t %d\n", byte1, byte2);
```

Teclas pulsadas	*Resultado*	
F1	0	59
Alt+A	0	30
Shift+F10	0	93
Ctrl+Inicio	0	119
Flecha hacia arriba	0	72

LIMPIAR LA PANTALLA

C proporciona la función **system** que permite enviar cualquier orden al sistema operativo; por ejemplo, la orden de limpiar la pantalla. Esta función tiene un argumento que es una cadena de caracteres. Cuando se invoca a la función **system** la cadena de caracteres es pasada al intérprete de órdenes del sistema operativo, que ejecuta la orden especificada por la cadena. La sintaxis es:

```
#include <stdlib.h>
int system(const char *cadena-de-caracteres);
Compatibilidad: ANSI, UNIX y MS-DOS
```

Las siguientes sentencias permiten limpiar la pantalla:

```
system("cls");     // limpiar la pantalla en MS-DOS
system("clear");   // limpiar la pantalla en UNIX
```

EJERCICIOS RESUELTOS

1. Realizar un programa que dé como resultado los intereses producidos y el capital total acumulado de una cantidad *c*, invertida a un interés *r* durante *t* días.

   La fórmula utilizada para el cálculo de los intereses es:

$$I = \frac{c.r.t}{360.100}$$

   siendo:

   I = Total de intereses a pagar
   c = Capital
   r = Tasa de interés nominal en tanto por ciento
   t = Período de cálculo en días

   La solución de este problema puede ser de la siguiente forma:

   - Primero definimos las variables que vamos a utilizar en los cálculos.

```
double c, intereses, capital;
float r;
int t;
```

   - A continuación leemos los datos *c*, *r* y *t*.

```
printf("Capital invertido        ");
scanf("%lf", &c);
printf("\nA un %% anual del          ");
scanf("%f", &r);
printf("\nDurante cuántos días     ");
scanf("%d", &t);
```

     Para tratar un carácter especial para el compilador, como es %, como un carácter ordinario, hay que duplicarlo. En el ejemplo anterior, %% da lugar a que se visualice un %.

   - Conocidos los datos, realizamos los cálculos. Nos piden los intereses producidos y el capital acumulado. Los intereses producidos los obtenemos aplicando

directamente la fórmula. El capital acumulado es el capital inicial más los intereses producidos.

```
intereses = c * r * t / (360L * 100);
capital = c + intereses;
```

Observe que la constante 360 la hemos declarado explícitamente **long** para que el resultado de 360*100 se calcule en este tipo. De no hacerlo así, el resultado de esa operación será **int**, pero en un ordenador de 16 bits este valor está fuera de rango, lo que producirá un error.

- Finalmente, escribimos el resultado.

```
printf("Intereses producidos...%10.0f\n", intereses);
printf("Capital acumulado......%10.0f\n", capital);
```

Los resultados se escriben sobre una columna de ancho 10, ajustados a la derecha y sin decimales. El resultado se redondea automáticamente.

Observe que el desarrollo de un programa, en general consta de tres bloques colocados en el siguiente orden:

El programa completo se muestra a continuación.

```
/********************* Capital e Intereses *********************/
/* capital.c
 */
#include <stdio.h>
#include <stdlib.h>

void main()
{
  double c, intereses, capital;
  float r;
  int t;

  system("cls"); /* limpiar pantalla */

  /* Entrada de datos */
  printf("Capital invertido        ");
  scanf("%lf", &c);
  printf("\nA un %% anual del         ");
  scanf("%f", &r);
  printf("\nDurante cuántos días     ");
  scanf("%d", &t);
  printf("\n\n\n");
```

```
    /* Cálculos */
    intereses = c * r * t / (360L * 100);
    capital = c + intereses;

    /* Escribir resultados */
    printf("Intereses producidos...%10.0f\n", intereses);
    printf("Capital acumulado......%10.0f\n", capital);
}
```

1. Realizar un programa que dé como resultado las soluciones reales *x1* y *x2* de una ecuación de segundo grado, de la forma:

$$ax^2 + bx + c = 0$$

Las soluciones de una ecuación de segundo grado vienen dadas por la fórmula:

$$x_i = \frac{-b \pm \sqrt{b^2 - 4.a.c}}{2.a}$$

Las soluciones son reales sólo si $b^2 - 4.a.c$ es mayor o igual que cero. La solución de este problema puede desarrollarse de la forma siguiente:

- Primero definimos las variables necesarias para los cálculos.

```
double a, b, c, d, x1, x2;
```

- A continuación leemos los coeficientes *a*, *b* y *c* de la ecuación.

```
printf("Introducir coeficientes a b c: ");
scanf("%lf %lf %lf", &a, &b, &c);
```

Observe que el tipo especificado es *lf* (**double**). Especificar un tipo *f* sería un error, porque las variables han sido definidas de tipo **double**. El formato utilizado por el ordenador internamente para almacenar un **float** es diferente al utilizado para almacenar un **double**.

- Nos piden calcular las raíces reales. Para que existan raíces reales tiene que cumplirse que $b^2 - 4.a.c \geq 0$; si no, las raíces son complejas conjugadas. Si hay raíces reales las calculamos; en otro caso, salimos del programa.

Para salir de un programa, en general para salir de un proceso sin hacer nada más, C proporciona la función

```
void exit(int estado);
```

Igual que sucedía con el valor retornado por la función **main**, el argumento *estado* es el valor que se devuelve al proceso que invocó al programa para su ejecución.

```
d = b * b - 4 * a * c;
(d < 0) ? printf("Las raíces son complejas\n"), exit(0)
        : printf("Las raíces reales son:\n");
```

- Si hay raíces reales las calculamos aplicando la fórmula.

```
d = sqrt(d);
x1 = (-b + d) / (2 * a);
x2 = (-b - d) / (2 * a);
```

La función **sqrt** calcula la raíz cuadrada de su argumento. En el ejemplo, se calcula la raíz cuadrada de *d* y se almacena el resultado de nuevo en *d*.

- Por último escribimos los resultados obtenidos.

```
printf("x1 = %g\nx2 = %g\n", x1, x2);
```

El programa completo se muestra a continuación.

```
/********** Solución de una ecuación de segundo grado **********/
/* ecuacion.c
 */
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

void main()
{
  double a, b, c, d, x1, x2;

  system("cls");

  /* Entrada de datos */
  printf("Introducir coeficientes a b c: ");
  scanf("%lf %lf %lf", &a, &b, &c);

  /* Comprobar si las raíces son reales */
  d = b * b - 4 * a * c;
  (d < 0) ? printf("Las raíces son complejas\n"), exit(0)
          : printf("Las raíces reales son:\n");

  /* Calculo de las soluciones */
  d = sqrt(d);
  x1 = (-b + d) / (2 * a);
  x2 = (-b - d) / (2 * a);

  /* Escribir resultados */
  printf("x1 = %g\nx2 = %g\n", x1, x2);
}
```

EJERCICIOS PROPUESTOS

1. Realizar un programa que calcule el volumen de una esfera, que viene dado por la fórmula:

$$v = \frac{4}{3}\pi r^3$$

2. Realizar un programa que pregunte el nombre y el año de nacimiento y dé como resultado:

```
Hola nombre, en el año 2030 tendrás n años
```

3. Realizar un programa que evalúe el polinomio

$$p = 3x^5 - 5x^3 + 2x - 7$$

y visualizar el resultado con el siguiente formato:

```
Para x = valor, 3x^5 - 5x^3 + 2^x - 7 = resultado
```

4. Realizar el mismo programa anterior, pero empleando ahora coeficientes variables *a*, *b* y *c*.

5. Ejecute el siguiente programa, explique lo que ocurre y realice las modificaciones que sean necesarias para su correcto funcionamiento.

```
#include <stdio.h>
void main()
{
  int car = 0;
  car = getchar();
  putchar(car);
  car = getchar();
  putchar(car);
}
```

6. Indique qué resultado da el siguiente programa. A continuación ejecute el programa y compare los resultados.

```
#include <stdio.h>
void main()
{
  char car1 = 'A', car2 = 65, car3 = 0;
  car3 = car1 + 'a' - 'A';
  printf("%d %c\n", car3, car3);
  car3 = car2 + 32;
  printf("%d %c\n", car3, car3);
}
```

CAPÍTULO 5

SENTENCIAS DE CONTROL

Cada función de los programas que hemos hecho hasta ahora, era un conjunto de sentencias que se ejecutaban en el orden que se habían escrito, entendiendo por *sentencia* una secuencia de expresiones que especifica una o varias operaciones. Pero esto no es siempre así; seguro que en algún momento nos ha surgido la necesidad de ejecutar unas sentencias u otras en función de unos criterios especificados por nosotros. Por ejemplo, en el capítulo anterior, cuando calculábamos las raíces de una ecuación de segundo grado, vimos que en función del valor del discriminante las raíces podían ser reales o complejas. En un caso como éste, surge la necesidad de que sea el propio programa el que tome la decisión, en función del valor del discriminante, de si lo que tiene que calcular son dos raíces reales o dos raíces complejas conjugadas.

Así mismo, en más de una ocasión necesitaremos ejecutar un conjunto de sentencias un número determinado de veces o hasta que se cumpla una condición impuesta por nosotros. Por ejemplo, en el capítulo anterior hemos visto cómo leer un carácter de la entrada estándar. Pero si lo que queremos es leer, no un carácter sino todos los que escribamos por el teclado hasta detectar la marca de fin de fichero, tendremos que utilizar una sentencia repetitiva.

En este capítulo aprenderá a escribir código para que un programa tome decisiones y para que sea capaz de ejecutar bloques de sentencias repetidas veces.

SENTENCIA if

La sentencia **if** permite a un programa tomar una decisión para ejecutar una acción u otra, basándose en el resultado verdadero o falso de una expresión. La sintaxis para utilizar esta sentencia es la siguiente:

```
if(condición)
  sentencia 1;
[else
  sentencia 2];
```

donde *condición* es una expresión numérica, relacional o lógica y *sentencia 1* y *sentencia 2* representan a una sentencia simple o compuesta. Cada sentencia simple debe finalizar con un punto y coma.

Una sentencia **if** se ejecuta de la forma siguiente:

1. Se evalúa la *condición.*
2. Si el resultado de la evaluación de la *condición* es verdadero (resultado distinto de cero) se ejecutará lo indicado por la *sentencia 1.*
3. Si el resultado de la evaluación de la *condición* es falso (resultado cero), se ejecutará lo indicado por la *sentencia 2*, si la cláusula **else** se ha especificado.
4. Si el resultado de la evaluación de la *condición* es falso, y la cláusula **else** se ha omitido, la *sentencia 1* se ignora.
5. En cualquier caso, la ejecución continúa en la siguiente sentencia ejecutable.

A continuación se exponen algunos ejemplos para que vea de una forma sencilla cómo se utiliza la sentencia **if.**

```
if (x)          /* es lo mismo que if (x != 0) */
  b = a / x;
b = b + 1;
```

En este ejemplo, la condición viene impuesta por una expresión numérica *x.* Entonces *b = a / x*, que sustituye a la *sentencia 1* del formato general, se ejecutará si la expresión es verdadera (*x* distinta de 0) y no se ejecutará si la expresión es falsa (*x* igual a 0). En cualquier caso, se continúa la ejecución en la línea siguiente, *b = b + 1.*

```
if (a < b) c = c + 1;
```

En este otro ejemplo, la condición viene impuesta por una expresión de relación. Si al evaluar la condición se cumple que *a* es menor que *b*, entonces se ejecuta la sentencia *c = c + 1.* En otro caso, esto es, si *a* es mayor o igual que *b*, se continúa en la línea siguiente, ignorándose la sentencia *c = c + 1.*

```
if (a && b)
  x = i;
```

En este ejemplo, la condición viene impuesta por una expresión lógica. Si al evaluar la condición se cumple que *a* y *b* son distintas de cero, entonces se ejecuta

la sentencia *x = i*. En otro caso, la sentencia *x = i* se ignora, continuando la ejecución en la línea siguiente.

```
if (a == b * 5)
{
  x = 4;
  a = a + x;
}
else
  b = 0;
```

En el ejemplo anterior, si se cumple que *a* es igual a *b*5*, se ejecutan las sentencias *x = 4* y *a = a + x*. En otro caso, se ejecuta la sentencia *b = 0*. En ambos casos, la ejecución continúa en la siguiente línea de programa. Un error típico es escribir, en lugar de la condición del ejemplo anterior, la siguiente:

```
if (a = b * 5)
  // ...
```

que equivale a

```
a = b * 5;
if (a)
  // ...
```

El ejemplo anterior demuestra que si por error utiliza el operador de asignación en lugar del operador de relación == los resultados pueden ser inesperados.

```
if (car == 's')
  break;
```

En este otro ejemplo, la sentencia **break** se ejecutará solamente cuando *car* sea igual al carácter *'s'*.

ANIDAMIENTO DE SENTENCIAS if

Las sentencias **if ... else** pueden estar anidadas. Según el formato general de la sentencia **if**, esto quiere decir que como *sentencia 1* o *sentencia 2* se puede escribir otra sentencia **if**. Por ejemplo:

```
if (condición 1)
{
  if (condición 2)
    sentencia 1;
}
else
  sentencia 2;
```

Al evaluarse las condiciones anteriores, pueden presentarse los casos que se indican en la tabla siguiente:

condición 1	*condición 2*	*sentencia 1*	*sentencia 2*
F	F	no	si
F	V	no	si
V	F	no	no
V	V	si	no

(V = verdadero, F = falso, no = no se ejecuta, si = si se ejecuta)

En el ejemplo anterior las llaves definen perfectamente que la cláusula **else** está emparejada con el primer **if**. ¿Qué sucede si quitamos las llaves?

```
if (condición 1)
  if (condición 2)
    sentencia 1;
  else
    sentencia 2;
```

Ahora podríamos dudar de a qué **if** pertenece la cláusula **else**. Cuando en el código de un programa aparecen sentencias **if ... else** anidadas, la regla para diferenciar cada una de estas sentencias es que "cada **else** se corresponde con el **if** más próximo que no haya sido emparejado". Según esto la cláusula **else** está emparejada con el segundo **if**. Entonces, al evaluarse ahora las condiciones 1 y 2, pueden presentarse los casos que se indican en la tabla siguiente:

condición 1	*condición 2*	*sentencia 1*	*sentencia 2*
F	F	no	no
F	V	no	no
V	F	no	si
V	V	si	no

(V = verdadero, F = falso, no = no se ejecuta, si = si se ejecuta)

Un ejemplo es el siguiente segmento de programa que escribe un mensaje indicando cómo es un número *a* con respecto a otro *b* (mayor, menor o igual):

```
if (a > b)
  printf("%d es mayor que %d", a, b);
else if (a < b)
       printf("%d es menor que %d", a, b);
     else
       printf("%d es igual a %d", a, b);
/* siguiente línea del programa */
```

Es importante que observe que una vez que se ejecuta una acción como resultado de haber evaluado las condiciones impuestas, la ejecución del programa continúa en la siguiente línea a la estructura a que dan lugar las sentencias **if ... else** anidadas. En el ejemplo anterior si se cumple que *a* es mayor que *b*, se escribe el mensaje correspondiente y se continúa en la siguiente línea del programa.

Así mismo, si en el siguiente ejemplo ocurre que *a* no es igual a 0, la ejecución continúa en la siguiente línea del programa.

```
if (a == 0)
  if (b != 0)
    s = s + b;
  else
    s = s + a;
/* siguiente línea del programa */
```

Si en lugar de la solución anterior, lo que deseamos es que se ejecute $s = s + a$ cuando *a* no es igual a *0*, entonces tendremos que incluir entre llaves el segundo **if** sin la cláusula **else**; esto es,

```
if (a == 0)
{
  if (b != 0)
    s = s + b;
}
else
  s = s + a;
/* siguiente línea del programa */
```

Como aplicación de la teoría expuesta, vamos a realizar un programa que dé como resultado el menor de tres números *a*, *b* y *c*.

La forma de proceder es comparar cada número con los otros dos una sola vez. La simple lectura del código que se muestra a continuación es suficiente para entender el proceso seguido.

```
/*************** Menor de tres números a, b y c ***************/
/* menor.c
 */
#include <stdio.h>

void main()
{
  float a, b, c, menor;

  printf("Números a b c : ");
  scanf("%g %g %g", &a, &b, &c);

  if (a < b)
    if (a < c)
```

```
      menor = a;
    else
      menor =c;
  else
    if (b < c)
      menor = b;
    else
      menor = c;

  printf("Menor = %g\n", menor);
}
```

ESTRUCTURA else if

La estructura presentada a continuación, aparece con bastante frecuencia y es por lo que se le da un tratamiento por separado. Esta estructura es consecuencia de las sentencias **if** anidadas. Su formato general es:

```
if (condición 1)
  sentencia 1;
else if (condición 2)
  sentencia 2;
else if (condición 3)
  sentencia 3;
.
.
.
else
  sentencia n
```

La evaluación de esta estructura sucede así: si se cumple la *condición 1*, se ejecuta la *sentencia 1* y si no se cumple se examinan secuencialmente las condiciones siguientes hasta el último **else**, ejecutándose la sentencia correspondiente al primer **else if**, cuya condición sea cierta. Si todas las condiciones son falsas, se ejecuta la *sentencia n* correspondiente al último **else**. En cualquier caso, se continúa en la primera sentencia ejecutable que haya a continuación de la estructura. Las sentencias 1, 2, ..., n pueden ser sentencias simples o compuestas.

Por ejemplo, al efectuar una compra en un cierto almacén, si adquirimos más de 100 unidades de un mismo artículo, nos hacen un descuento de un 40 %; entre 25 y 100 un 20 %; entre 10 y 24 un 10 %; y no hay descuento para una adquisición de menos de 10 unidades. Se pide calcular el importe a pagar. La solución se presentará de la siguiente forma:

```
Código artículo..... 111
Cantidad comprada..... 100
Precio unitario..... 100
```

```
Artículo    Cantidad      P. U.       Dto.       Total
     111         100    100.00        20%     8000.00
```

En la solución presentada como ejemplo, se puede observar que como la cantidad comprada está entre 25 y 100, el descuento aplicado es de un 20%.

La solución de este problema puede ser de la siguiente forma:

- Primero definimos las variables que vamos a utilizar en los cálculos.

```
int ar, cc; /* código y cantidad */
float pu;   /* precio unitario */
```

- A continuación leemos los datos *ar*, *cc* y *pu*.

```
printf("Código artículo..... ");
scanf("%d", &ar);
printf("Cantidad comprada..... ");
scanf("%d", &cc);
printf("Precio unitario..... ");
scanf("%f", &pu);
```

- Conocidos los datos, realizamos los cálculos y simultáneamente escribimos el resultado. Esto exige escribir primero la cabecera mostrada en la solución ejemplo, y los datos leídos.

```
printf("\n\n%10s %10s %10s %10s %10s\n\n",
        "Artículo", "Cantidad", "P. U.", "Dto.", "Total");
printf("%10d %10d %10.2f", ar, cc, pu);
if (cc > 100)
  printf(" %9d%% %10.2f\n", 40, cc * pu * 0.6);
else if (cc >= 25)
  printf(" %9d%% %10.2f\n", 20, cc * pu * 0.8);
else if (cc >= 10)
  printf(" %9d%% %10.2f\n", 10, cc * pu * 0.9);
else
  printf(" %10s %10.2f\n", "--", cc * pu);
```

Observe que las condiciones se han establecido desde la cantidad tope para los descuentos mayor a la menor. Como ejercicio, piense o pruebe que ocurriría si establece las condiciones desde la cantidad menor a la mayor.

El programa completo se muestra a continuación.

```
/**** Cantidad a pagar en función de la cantidad comprada ****/
/* else_if.c
 */
#include <stdio.h>
```

```
void main()
{
  int ar, cc;
  float pu;

  printf("Código artículo..... ");
  scanf("%d", &ar);
  printf("\nCantidad comprada..... ");
  scanf("%d", &cc);
  printf("\nPrecio unitario..... ");
  scanf("%f", &pu);

  printf("\n\n%10s %10s %10s %10s %10s\n\n",
       "Artículo", "Cantidad", "P. U.", "Dto.", "Total");
  printf("%10d %10d %10.2f", ar, cc, pu);

  if (cc > 100)
    printf(" %9d%% %10.2f\n", 40, cc * pu * 0.6);
  else if (cc >= 25)
    printf(" %9d%% %10.2f\n", 20, cc * pu * 0.8);
  else if (cc >= 10)
    printf(" %9d%% %10.2f\n", 10, cc * pu * 0.9);
  else
    printf(" %10s %10.2f\n", "--", cc * pu);
}
```

Para poder imprimir un símbolo que tiene un significado especial para C, tiene que ser duplicado en la expresión correspondiente. Como ejemplo, observe en el programa anterior el formato *%9d%%*; distinguimos dos partes: *%9d* es el formato utilizado para escribir el tanto por ciento de descuento, y %% hace que se escriba a continuación el carácter "%".

SENTENCIA switch

La sentencia **switch** permite ejecutar una de varias acciones, en función del valor de una expresión. Es una sentencia especial para decisiones múltiples. La sintaxis para utilizar esta sentencia es:

```
switch (expresión)
{
   [declaraciones]
    case expresión-constante 1:
      [sentencia 1;]
   [case expresión-constante 2:]
      [sentencia 2;]
   [case expresión-constante 3:]
      [sentencia 3;]
   .
   .
   .
   [default:]
      [sentencia n;]
}
```

donde *expresión* es una expresión entera y *expresión-constante* es una constante entera, una constante de un solo carácter o una expresión constante, en cuyo caso, el valor resultante tiene que ser entero. Por último, *sentencia* es una sentencia simple o compuesta.

La sentencia **switch** evalúa la expresión entre paréntesis y compara su valor con las constantes de cada **case**. La ejecución de las sentencias del bloque de la sentencia **switch**, comienza en el **case** cuya constante coincida con el valor de la *expresión* y continúa hasta el final del bloque o hasta una sentencia que transfiera el control fuera del bloque de **switch**; por ejemplo, **break** o **return**. La sentencia **switch** puede incluir cualquier número de cláusulas **case**.

Si no existe una constante igual al valor de la *expresión*, entonces se ejecutan las sentencias que están a continuación de **default**, si esta cláusula ha sido especificada. La cláusula **default** puede colocarse en cualquier parte del bloque y no necesariamente al final.

Igual que en cualquier otro bloque, en el bloque de la sentencia **switch** es posible hacer *declaraciones* al principio. No obstante, las inicializaciones si las hubiere, son ignoradas.

Para ilustrar lo expuesto, vamos a realizar un programa que lea una fecha representada por dos enteros, mes y año, y dé como resultado los días correspondientes al mes. Esto es,

```
Introducir mes (##) y año (####): 5 1995

El mes 5 del año 1995 tiene 31 días
```

Hay que tener en cuenta que Febrero puede tener 28 días o 29 si el año es bisiesto. Un año es bisiesto cuando es múltiplo de 4 y no de 100 o cuando es múltiplo de 400. Por ejemplo, el año 2000 por las dos primeras condiciones no sería bisiesto, pero sí lo es porque es múltiplo de 400; el año 2100 no es bisiesto porque aunque sea múltiplo de 4, también lo es de 100 y no es múltiplo de 400.

La solución de este problema puede ser de la siguiente forma:

- Primero definimos las variables que vamos a utilizar en los cálculos.

```
unsigned int dd = 0, mm = 0, aa = 0;
```

- A continuación leemos los datos *mm* y *aa*.

```
printf("Introducir mes (##) y año (####): ");
scanf("%d %d", &mm, &aa);
```

- Después comparamos el mes *mm* con las constantes 1, 2, ..., 12. Si *mm* es 1, 3, 5, 7, 8, 10 o 12 asignamos a *dd* el valor 31. Si *mm* es 4, 6, 9 u 11 asignamos a *dd* el valor 30. Si *mm* es 2, verificaremos si el año es bisiesto, en cuyo caso asignamos a *dd* el valor 29 y si no es bisiesto, asignamos a *dd* el valor 28. Si *dd* no es ningún valor de los anteriores enviaremos un mensaje al usuario indicándole que el mes no es válido. Todo este proceso lo realizaremos con una sentencia **switch**.

```
switch (mm)
{
  case 1: case 3: case 5: case 7: case 8: case 10: case 12:
    dd = 31;
    break;
  case 4: case 6: case 9: case 11:
    dd = 30;
    break;
  case 2:
    if ((aa % 4 == 0) && (aa % 100 != 0) || (aa % 400 == 0))
      dd = 29;
    else
      dd = 28;
    break;
  default:
    printf("\nEl mes no es válido\n");
    break;
}
```

Cuando una constante coincida con el valor de *mm*, se ejecutan las sentencias especificadas a continuación de la misma, continuando la ejecución del programa en el siguiente **case**, a no ser que se tome una acción explícita para abandonar el bloque de la sentencia **switch**. Esta es precisamente la función de la sentencia **break**.

- Por último si el mes es válido, escribimos el resultado solicitado.

```
if (mm >= 1 && mm <= 12)
  printf("\nEl mes %d del año %d tiene %d días\n",mm,aa,dd);
```

El programa completo se muestra a continuación.

```
/******* Días correspondientes a un mes de un año dado *******/
/* switch.c
 */
#include <stdio.h>

void main()
{
  unsigned int dd = 0, mm = 0, aa = 0;

  printf("Introducir mes (##) y año (####): ");
```

```
    scanf("%d %d", &mm, &aa);

    switch (mm)
    {
      case 1:
      case 3:
      case 5:
      case 7:
      case 8:
      case 10:
      case 12:
        dd = 31;
        break;
      case 4:
      case 6:
      case 9:
      case 11:
        dd = 30;
        break;
      case 2:
        if ((aa % 4 == 0) && (aa % 100 != 0) || (aa % 400 == 0))
         dd = 29;
        else
         dd = 28;
        break;
      default:
        printf("\nEl mes no es válido\n");
        break;
    }
    if (mm >= 1 && mm <= 12)
      printf("\nEl mes %d del año %d tiene %d días\n",mm,aa,dd);
}
```

El que las cláusulas **case** estén una a continuación de otra o una debajo de otra no es más que una cuestión de estilo, ya que C interpreta cada carácter nueva línea como un espacio en blanco; esto es, el código al que llega el compilador es el mismo en cualquier caso.

La sentencia **break** que se ha puesto a continuación de la cláusula **default** no es necesaria; simplemente obedece a un buen estilo de programación. Así, cuando tengamos que añadir otro caso ya tenemos puesto **break**, con lo que hemos eliminado una posible fuente de errores.

SENTENCIA break

La sentencia **break** finaliza la ejecución de una sentencia **switch**, **while**, **do**, o **for**, en la cual aparece. Su sintaxis es:

```
break;
```

Cuando las sentencias **switch**, **while**, **do**, o **for** estén anidadas, la sentencia **break** solamente finaliza la ejecución de la sentencia **switch**, **while**, **do**, o **for** donde esté incluida.

Como ejercicio vamos a realizar un programa que calcule el importe a pagar por un vehículo al circular por una autopista. El vehículo puede ser una bicicleta, una moto, un coche o un camión. Para definir el conjunto vehículos utilizaremos un tipo enumerado (vea en el capítulo 2 los tipos enumerados). El importe se calculará según los siguientes datos:

- Un importe fijo de 100 unidades para las bicicletas.
- Las motos y los coches pagarán 30 unidades por Km.
- Los camiones pagarán 30 unidades por Km. más 25 unidades por Tm.

La presentación en pantalla de la solución, será de la forma siguiente:

```
        1 - bicicleta
        2 - moto
        3 - coche
        4 - camión

        Pulse la opción deseada 3

¿Kilómetros? 20

Importe = 600
```

La solución de este problema puede ser de la siguiente forma:

- Primero declaramos el tipo enumerado *tvehiculos* y a continuación definimos las variables que vamos a utilizar en los cálculos.

```
typedef enum tipo_vehiculo
  {
    bicicleta = 1,
    moto,
    coche,
    camion
  } tvehiculo;

tvehiculo vehiculo;
int km, tm, importe;
```

- A continuación presentamos el menú que proporcionará el valor relacionado con el tipo de vehículo del que queremos calcular el importe que tiene que pagar.

```
printf("\t1 - bicicleta\n");
printf("\t2 - moto\n");
```

```
printf("\t3 - coche\n");
printf("\t4 - camión\n");
printf("\n\tPulse la opción deseada ");
scanf("%d", &vehiculo);
```

- Después comparamos la variable *vehiculo* con las constantes *bicicleta*, *moto*, *coche* y *camion*, cuyos valores asociados son 1, 2, 3 y 4, respectivamente, con el fin de calcular el importe a pagar por el vehículo de que se trate. Este cálculo, en el caso de motos y coches necesita del dato Km., y en el caso de camiones de los datos Km. y Tm., datos que leeremos desde el teclado. Para este proceso utilizaremos una sentencia **switch**.

```
switch (vehiculo)
{
  case bicicleta:
    importe = 100;
    break;
  case moto:
  case coche:
    printf("\n¿Kilómetros? ");
    scanf("%d", &km);
    importe = 30 * km;
    break;
  case camion:
    printf("\n¿Kilómetros y toneladas? ");
    scanf("%d %d", &km, &tm);
    importe = 30 * km + 25 * tm;
    break;
  default:
    printf("\nLa opción no es correcta\n");
    return; /* error; salir de main */
}
```

- Por último, escribimos el resultado.

```
printf("\nImporte = %d\n", importe);
```

El programa completo se muestra a continuación.

```
/* Importe a pagar por un vehículo al circular por una autopista
 *
 * break.c
 */
#include <stdio.h>
#include <stdlib.h>

void main()
{
  typedef enum tipo_vehiculo
    {
      bicicleta = 1,
      moto,
      coche,
```

```
        camion
    } tvehiculo;

  tvehiculo vehiculo;
  int km, tm, importe;

  system("cls");
  printf("\t1 - bicicleta\n");
  printf("\t2 - moto\n");
  printf("\t3 - coche\n");
  printf("\t4 - camión\n");
  printf("\n\tPulse la opción deseada ");
  scanf("%d", &vehiculo);

  switch (vehiculo)
  {
    case bicicleta:
      importe = 100;
      break;
    case moto:
    case coche:
      printf("\n¿Kilómetros? ");
      scanf("%d", &km);
      importe = 30 * km;
      break;
    case camion:
      printf("\n¿Kilómetros y toneladas? ");
      scanf("%d %d", &km, &tm);
      importe = 30 * km + 25 * tm;
      break;
    default:
      printf("\nLa opción no es correcta\n");
      return; /* error; salir de main */
  }
  printf("\nImporte = %d\n", importe);
}
```

SENTENCIA while

La sentencia **while** ejecuta una sentencia, simple o compuesta, cero o más veces, dependiendo del valor de una expresión. Su sintaxis es:

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

donde *condición* es cualquier expresión numérica, relacional o lógica y *sentencia* es una sentencia simple o compuesta.

La ejecución de la sentencia **while** sucede así:

1. Se evalúa la *condición.*

2. Si el resultado de la evaluación es cero (falso), la *sentencia* no se ejecuta y se pasa el control a la siguiente sentencia en el programa.
3. Si el resultado de la evaluación es distinto de cero (verdadero), se ejecuta la *sentencia* y el proceso descrito se repite desde el punto 1.

Por ejemplo, la rutina siguiente solicita obligatoriamente una de las dos respuestas posibles: s/n (sí o no).

```
#include <stdio.h>
#include <conio.h>
void main()
{
  char car = '\0';

  printf("\nDesea continuar s/n (sí o no) ");
  car = _getche();
  while (car != 's' && car != 'n')
  {
    printf("\nDesea continuar s/n (sí o no) ");
    car = _getche();
  }
}
```

Observe que antes de ejecutarse la sentencia **while** se visualiza el mensaje "Desea continuar s/n (sí o no) " y se inicializa la condición; esto es, se asigna un carácter a la variable *car* que interviene en la condición de la sentencia **while**.

La sentencia **while** se interpreta de la forma siguiente: mientras el valor de *car* no sea igual ni al carácter 's' ni al carácter 'n', visualizar el mensaje "Desea continuar s/n (sí o no) " y leer otro carácter. Esto obliga al usuario a escribir el carácter 's' o 'n'.

El ejemplo expuesto, puede escribirse también así:

```
#include <stdio.h>
#include <conio.h>
void main()
{
  char car = '\0';

  printf("\nDesea continuar s/n (sí o no) ");
  while ((car = _getche()) != 's' && car != 'n')
    printf("\nDesea continuar s/n (sí o no) ");
}
```

La diferencia de este ejemplo con respecto al anterior es que ahora la condición incluye la lectura de la variable *car*, la cual se ejecuta primero por estar entre paréntesis. A continuación se compara *car* con los caracteres 's' y 'n'.

El siguiente ejemplo, que visualiza el código ASCII de un carácter, da lugar a un bucle infinito, porque la condición es siempre cierta (valor distinto de cero). Para salir del bucle infinito tiene que pulsar las teclas *Ctrl+C*.

```
/************** Código ASCII de un carácter **************/
#include <stdio.h>

void main()
{
  while (1)  /* condición siempre cierta */
  {
    char car = 0; /* car = carácter nulo (\0) */

    printf("Introduzca un carácter: ");
    car = getchar();
    printf("\nEl código ASCII de %c es %d\n", car, car);
  }
}
```

En este ejemplo se ha definido la variable *car* local al bloque **while**. Esto se ha hecho así, simplemente por definirla en el bloque donde se va a utilizar, pero se podría haber definido exactamente igual, local a la función **main**.

A continuación ejecutamos el programa, introducimos, por ejemplo, el carácter 'a' y observamos los siguientes resultados:

```
Introduzca un carácter: a↵

El código ASCII de a es 97

Introduzca un carácter:
El código ASCII de ←——nueva línea
 es 10

Introduzca un carácter: Ctrl+C
```

Este resultado nos demuestra que cuando escribimos 'a' y después pulsamos ↵ para validar la entrada, en el *buffer* de entrada hay dos caracteres, 'a' y nueva línea (vea el apartado "carácter fin de línea" en el capítulo 4). El que se hayan leído todos los caracteres hasta que el *buffer* quedó vacío, induce a pensar que también podríamos introducir no un carácter sino un texto cualquiera. Por ejemplo:

```
Introduzca un carácter: hola

El código ASCII de h es 104

Introduzca un carácter:
El código ASCII de o es 111

Introduzca un carácter:
El código ASCII de l es 108
```

```
Introduzca un carácter:
El código ASCII de a es 97

Introduzca un carácter:
El código ASCII de
 es 10

Introduzca un carácter:
```

El resultado obtenido indica que el bucle **while** se está ejecutando sin pausa mientras hay caracteres en el *buffer*. Cuando el *buffer* de entrada queda vacío, la ejecución se detiene en la llamada a **getchar**, a la espera de nuevos datos.

Según lo expuesto, vamos a modificar el ejemplo anterior para que solicite la introducción de un texto, en lugar de un carácter. También, en lugar de establecer un bucle infinito, vamos a establecer como condición, que la entrada de datos finalice cuando se detecte la marca de fin de fichero. Recuerde que para el fichero estándar de entrada, esta marca se produce cuando se pulsan las teclas *Ctrl+D* en UNIX o *Ctrl+Z* en MS-DOS, y que cuando **getchar** lee una marca de fin de fichero, devuelve el valor **EOF** (capítulo 4).

```
/******** Código ASCII de los caracteres de un texto ********/
#include <stdio.h>

void main()
{
  char car = 0; /* car = carácter nulo (\0) */

  printf("Introduzca un texto: ");
  while ((car = getchar()) != EOF)
    printf("\nEl código ASCII de %c es %d\n", car, car);
}
```

Una solución posible de este programa es la siguiente:

```
Introduzca un texto: hola•  <-- marca de fin de fichero

El código ASCII de h es 104

El código ASCII de o es 111

El código ASCII de l es 108

El código ASCII de a es 97
```

También, se podría haber efectuado la entrada como se indica a continuación. La diferencia está en que además de los caracteres anteriores, se lee el carácter '\n'.

```
Introduzca un texto: hola↵  <-- nueva línea
•
  <-- marca de fin de fichero
```

Bucles anidados

Cuando se incluye una sentencia **while** dentro de otra sentencia **while**, en general una sentencia **while**, **do**, o **for** dentro de otra de ellas, estamos en el caso de bucles anidados. Por ejemplo:

```
#include <stdio.h>

void main()
{
  int i = 1, j = 1;

  while ( i <= 3 )    /* mientras i sea menor o igual que 3 */
  {
    printf("Para i = %d: ", i);
    while ( j <= 4 ) /* mientras j sea menor o igual que 4 */
    {
      printf("j = %d, ", j);
      j++;           /* aumentar j en una unidad */
    }
    printf("\n");    /* avanzar a una nueva línea */
    i++;             /* aumentar i en una unidad */
    j = 1;           /* inicializar j de nuevo a 1 */
  }
}
```

Al ejecutar este programa se obtiene el siguiente resultado:

```
Para i = 1: j = 1, j = 2, j = 3, j = 4,
Para i = 2: j = 1, j = 2, j = 3, j = 4,
Para i = 3: j = 1, j = 2, j = 3, j = 4,
```

Este resultado demuestra que el bucle exterior se ejecute tres veces y por cada una de estas, el bucle interior se ejecuta cuatro veces. De esta forma es como se ejecutan los bucles anidados.

Observe también que cada vez que finaliza la ejecución de la sentencia **while** interior, avanzamos a una nueva línea, incrementamos el valor de *i* en una unidad e inicializamos de nuevo *j* al valor 1.

Como aplicación de lo expuesto, vamos a realizar un programa que imprima los números *z*, comprendidos entre 1 y 50, que cumplan la expresión:

$$z^2 = x^2 + y^2$$

donde *z*, *x* e *y* son números enteros positivos. El resultado se presentará de la forma siguiente:

```
   Z          X          Y
-------------------------------
   5          3          4
  13          5         12
  10          6          8
 ...        ...        ...
  50         30         40
```

La solución de este problema puede ser de la siguiente forma:

- Primero definimos las variables que vamos a utilizar en los cálculos.

```
unsigned int x = 1, y = 1, z = 0;
```

- A continuación escribimos la cabecera de la solución.

```
printf("%10s %10s %10s\n", "Z", "X", "Y");
printf("      ____________________________\n");
```

- Después, para $x = 1$, e $y = 1, 2, 3, 4, ...$, para $x = 2$, e $y = 2, 3, 4, ...$, para $x = 3$, e $y = 3, 4, ...$, hasta $x = 50$, calculamos la $\sqrt{x^2 + y^2}$; llamamos a este valor z (observe que y es igual o mayor que x para evitar que se repitan pares de valores como *x=3, y=4* y *x=4, y=3*). Si z es exacto, escribimos z, x e y. Esto es, para los valores descritos de x e y, hacemos los cálculos,

```
z = sqrt(x * x + y * y);     /* z es una variable entera */
if (z * z == x * x + y * y) /* ¿la raíz cuadrada fue exacta? */
  printf("%10d %10d %10d\n", z, x, y);
```

Además, siempre que obtengamos un valor z mayor que 50 lo desecharemos y continuaremos con un nuevo valor de x y los correspondientes valores de y.

El programa completo se muestra a continuación.

```
/* Teorema de Pitágoras.
 * whileani.c
 */
#include <stdio.h>
#include <math.h>

void main()
{
  unsigned int x = 1, y = 1, z = 0;

  printf("%10s %10s %10s\n", "Z", "X", "Y");
  printf("      ____________________________\n");

  while (x <= 50)
  {
    /* Calcular z. Como z es un entero, almacena
       la parte entera de la raíz cuadrada */
    z = sqrt(x * x + y * y);
```

```
    while (y <= 50 && z <= 50)
    {
      /* Si la raíz cuadrada anterior fue exacta,
         escribir z, x e y */
      if (z * z == x * x + y * y)
       printf("%10d %10d %10d\n", z, x, y);
      y = y + 1;
      z = sqrt(x * x + y * y);
    }
    x = x + 1; y = x;
  }
}
```

SENTENCIA do

La sentencia **do** ejecuta una sentencia, simple o compuesta, una o más veces dependiendo del valor de una expresión. Su sintaxis es la siguiente:

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

donde *condición* es cualquier expresión numérica, relacional o lógica y *sentencia* es una sentencia simple o compuesta. Observe que la estructura **do - while** finaliza con un punto y coma.

La ejecución de una sentencia **do** sucede de la siguiente forma:

1. Se ejecuta el bloque (sentencia simple o compuesta) de **do**.
2. Se evalúa la expresión correspondiente a la *condición* de finalización del bucle.
3. Si el resultado de la evaluación es cero (falso), se pasa el control a la siguiente sentencia en el programa.
4. Si el resultado de la evaluación es distinto de cero (verdadero), el proceso descrito se repite desde el punto 1.

Por ejemplo, el siguiente programa obliga al usuario a introducir un valor positivo:

```
#include <stdio.h>
void main()
{
  double n;
  do  /* ejecutar las sentencias siguientes */
  {
    printf("Número: ");
    scanf("%lf", &n);
  }
  while ( n < 0 ); /* mientras n sea menor que cero */
}
```

Cuando se utiliza una estructura **do - while** el bloque de sentencias se ejecuta al menos una vez, porque la condición se evalúa al final. En cambio, cuando se ejecuta una estructura **while** puede suceder que el bloque de sentencias no se ejecute, lo que ocurrirá siempre que la condición sea inicialmente falsa.

Como aplicación, vamos a realizar un programa que calcule la raíz cuadrada de un número *n* por el método de Newton. Este método se enuncia así: sea r_i la raíz cuadrada aproximada de *n*. La siguiente raíz aproximada r_{i+1} se calcula en función de la anterior así:

$$r_{i+1} = \frac{\frac{n}{r_i} + r_i}{2}$$

El proceso descrito se repite hasta que la diferencia en valor absoluto de las dos últimas aproximaciones calculadas, sea tan pequeña como nosotros queramos (teniendo en cuenta los límites establecidos por el compilador). Según esto, la última aproximación será una raíz válida, cuando se cumpla que:

$$abs(r_i - r_{i+1}) \leq \varepsilon$$

La solución de este problema puede ser de la siguiente forma:

- Primero definimos las variables que vamos a utilizar en los cálculos.

```
double n;         /* número */
double aprox;     /* aproximación a la raíz cuadrada */
double antaprox;  /* anterior aproximación a la raíz cuadrada */
double epsilon;   /* coeficiente de error */
```

- A continuación leemos los datos *n*, *aprox* y *epsilon*.

```
do
{
  printf("Número:                    ");
  scanf("%lf", &n);
}
while ( n < 0 );
printf("Raíz cuadrada aproximada: ");
scanf("%lf", &aprox);
printf("Coeficiente de error:     ");
scanf("%lf", &epsilon);
```

Para no permitir la entrada de número negativos, se ha utilizado una estructura **do - while** que preguntará por el valor de *n* mientras se introduzca un número negativo.

- Después, se aplica la fórmula de Newton.

```
do
{
  antaprox = aprox;
  aprox = (n/antaprox + antaprox) / 2;
}
while (fabs(aprox - antaprox) >= epsilon);
```

Al aplicar la fórmula por primera vez, la variable *antaprox* contiene el valor aproximado, que hemos leído, a la raíz cuadrada. Para sucesivas veces, *antaprox* contiene la última aproximación calculada.

La función de la biblioteca de C **fabs** devuelve un valor **double** que se corresponde con el valor absoluto de su argumento, también de tipo **double**.

- Cuando la condición especificada en la estructura **do - while** anterior sea falsa, el proceso habrá terminado. Sólo queda imprimir el resultado.

```
printf("\nLa raíz cuadrada de %.2lf es %.2lf\n", n, aprox);
```

El programa completo se muestra a continuación.

```
/****** Raíz cuadrada de un número. Método de Newton ******/
/* do.c
 */
#include <stdio.h>
#include <math.h>
void main()
{
  double n;         /* número */
  double aprox;     /* aproximación a la raíz cuadrada */
  double antaprox; /* anterior aproximación a la raíz cuadrada */
  double epsilon;  /* coeficiente de error */
  do
  {
    printf("Número:                    ");
    scanf("%lf", &n);
  }
  while ( n < 0 );
  printf("Raíz cuadrada aproximada: ");
  scanf("%lf", &aprox);
  printf("Coeficiente de error:     ");
  scanf("%lf", &epsilon);
  do
  {
    antaprox = aprox;
    aprox = (n/antaprox + antaprox) / 2;
  }
  while (fabs(aprox - antaprox) >= epsilon);
  printf("\nLa raíz cuadrada de %.2lf es %.2lf\n", n, aprox);
}
```

Si ejecuta este programa para un valor de *n* igual a 10, obtendrá la siguiente solución:

```
Número:                    10
Raíz cuadrada aproximada: 1
Coeficiente de error:      1e-4

La raíz cuadrada de 10.00 es 3.16
```

SENTENCIA for

La sentencia **for** permite ejecutar una sentencia simple o compuesta, repetidamente un número de veces conocido. Su sintaxis es la siguiente:

```
for ([v1=e1, [v2=e2]...];[condición];[progresión-condición])
  sentencia;
```

donde *v1*, *v2*, ..., representan variables que serán inicializadas con los valores de las expresiones *e1*, *e2*, ...; *condición* es una expresión de Boole (operandos unidos por operadores relacionales y/o lógicos) que si se omite, se supone verdadera; *progresión-condición* es una expresión cuyo valor evoluciona en el sentido de que se cumpla la condición para finalizar la ejecución de la sentencia **for** y *sentencia* es una sentencia simple o compuesta que forma lo que llamamos bloque de sentencias.

La ejecución de la sentencia **for** sucede de la siguiente forma:

1. Se inicializan las variables *v1*, *v2*, ...
2. Se evalúa la *condición*.
   a) Si el resultado es distinto de cero (verdadero), se ejecuta el bloque de sentencias, se evalúa la expresión que da lugar a la progresión de la condición y se vuelve al punto 2.
   b) Si el resultado de 2 es cero (falso), la ejecución de la sentencia **for** se da por finalizada y se pasa el control a la siguiente sentencia del programa.

Por ejemplo, la siguiente sentencia **for** imprime los números del 1 al 100. Literalmente dice: desde *i* igual a 1, mientras *i* sea menor o igual que 100, aumentando la *i* de uno en uno, escribir el valor de *i*.

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

El siguiente ejemplo imprime los múltiplos de 7 que hay entre 7 y 112.

```
for (k = 7; k <= 112; k += 7)
  printf("%d  ", k);
```

Este otro ejemplo que ve a continuación, imprime los valores desde 1 hasta 10 con incrementos de 0.5.

```
float i;
for (i = 1; i <= 10; i += 0.5)
  printf("%g ", i);
```

El siguiente ejemplo imprime los números del 9 al 1.

```
for (a = 9; a >= 1; a--)
  printf("%d  ", a);
```

En el ejemplo que se muestra a continuación, observe como se inicializan las variables *filas* y *car*.

```
for (filas = 1, car = '\x20'; filas <= nfilas; filas++)
```

El ejemplo que ve a continuación indica cómo realizar un bucle infinito. Para salir de un bucle infinito tiene que pulsar las teclas *Ctrl+C*.

```
for (;;)
{
  sentencias;
}
```

Como aplicación de la sentencia **for** vamos a imprimir un tablero de ajedrez en el que las casillas blancas se simbolizarán con una B y las negras con una N. Así mismo, el programa deberá marcar con * las casillas a las que se puede mover un alfil desde una posición dada. La solución será similar a la siguiente:

```
Posición del alfil (fila, columna): 3 4

B * B N B * B N
N B * B * B N B
B N B * B N B N
N B * B * B N B
B * B N B * B N
* B N B N B * B
B N B N B N B *
N B N B N B N B
```

Desarrollo del programa:

- Primero definimos las variables que vamos a utilizar en los cálculos.

```
int falfil, calfil;   /* posición del alfil */
int fila, columna;    /* posición actual */
```

- Leer la fila y la columna en la que se coloca el alfil.

```
printf("Posición del alfil (fila, columna): ");
scanf("%d %d", &falfil, &calfil);
```

- Partiendo de la fila 1, columna 1 y recorriendo el tablero por filas imprimir un *, una B o una N dependiendo de las condiciones especificadas a continuación.

```
for (fila = 1; fila <= 8; fila++)
{
  for (columna = 1; columna <= 8; columna++)
  {
    /* Pintar el tablero de ajedrez */
  }
  printf("\n");  /* cambiar de fila */
}
```

Imprimir un * si se cumple, que la suma o diferencia de la fila y columna actuales, coincide con la suma o diferencia de la fila y columna donde se coloca el alfil.

Imprimir una B si se cumple que la fila más columna actuales es par.

Imprimir una N si se cumple que la fila más columna actuales es impar.

```
/* Pintar el tablero de ajedrez */
if ((fila + columna == falfil + calfil) ||
    (fila - columna == falfil - calfil))
  printf("* ");
else if ((fila + columna) % 2 == 0)
  printf("B ");
else
  printf("N ");
```

El programa completo se muestra a continuación.

```
/********************* Tablero de Ajedrez *********************/
/* for.c
 */
#include <stdio.h>

void main()
{
  int falfil, calfil;  /* posición del alfil */
  int fila, columna;   /* posición actual */

  printf("Posición del alfil (fila, columna): ");
  scanf("%d %d", &falfil, &calfil);
  printf("\n");  /* dejar una línea en blanco */

  /* Pintar el tablero de ajedrez */
  for (fila = 1; fila <= 8; fila++)
  {
    for (columna = 1; columna <= 8; columna++)
    {
```

```
      if ((fila + columna == falfil + calfil) ||
          (fila - columna == falfil - calfil))
        printf("* ");
      else if ((fila + columna) % 2 == 0)
        printf("B ");
      else
        printf("N ");
    }
    printf("\n");  /* cambiar de fila */
  }
}
```

SENTENCIA continue

La sentencia **continue** obliga a ejecutar la siguiente iteración en el bucle correspondiente a la sentencia **while**, **do**, o **for**, en el que está contenida. Su sintaxis es:

```
continue;
```

Como ejemplo, vea el siguiente programa que imprime todos los números entre 1 y 100 que no sean múltiplos de 5.

```
#include <stdio.h>

void main()
{
  int n;
  for (n = 0; n <= 100; n++)
  {
    if (n % 5 == 0) /*si n es múltiplo de 5, siguiente iteración*/
       continue;
    printf("%d  ", n);
  }
}
```

Ejecute este programa y observe que cada vez que se ejecuta la sentencia **continue**, se inicia la ejecución del bloque de sentencias de **for** para un nuevo valor de *n*.

SENTENCIA goto

La sentencia **goto** transfiere el control a una línea específica del programa, identificada por una *etiqueta*. Su sintaxis es la siguiente:

```
goto etiqueta;
.
.
.
etiqueta: sentencia;
```

Si la línea a la que se transfiere el control es una sentencia ejecutable, se ejecuta esa sentencia y las que le siguen. Si no es ejecutable, la ejecución se inicia en la primera sentencia ejecutable que se encuentre a continuación de dicha línea.

No se puede transferir el control fuera del cuerpo de la función en la que nos encontramos.

Un uso abusivo de esta sentencia da lugar a programas difíciles de interpretar y de mantener. Por ello, en programación estructurada, se utiliza solamente en ocasiones excepcionales. La función que desempeña una sentencia **goto**, puede suplirse utilizando las sentencias **if...else**, **do**, **for**, **switch**, **while**.

El uso más normal consiste en abandonar la ejecución de alguna estructura profundamente anidada, cosa que no puede hacerse mediante la sentencia **break**, ya que ésta se limita únicamente a un solo nivel de anidamiento.

El siguiente ejemplo muestra cómo se utiliza la sentencia **for**. Consta de dos bucles **for** anidados. En el bucle interior hay una sentencia **goto** que se ejecutará si se cumple la condición especificada. Si se ejecuta la sentencia **goto** el control es transferido a la primera sentencia ejecutable que haya a continuación de la etiqueta *salir*.

```
/********** goto salir **********/
#include <stdio.h>
#define K 8

void main()
{
  int f, c, n;

  printf("Valor de n: ");
  scanf("%d", &n);

  for (f = 0; f < K; f++)
  {
    for (c = 0; c < K; c++)
    {
      if (f*c > n) goto salir;
    }
  }
  salir:
  if (f < K && c < K)
    printf("(%d, %d)\n", f, c);
}
```

EJERCICIOS RESUELTOS

1. Realizar un programa que calcule las raíces de la ecuación:

$$ax^2 + bx + c = 0$$

teniendo en cuenta los siguientes casos:

1. Si a es igual a 0 y b es igual a 0, imprimiremos un mensaje diciendo que la ecuación es degenerada.

2. Si a es igual a 0 y b no es igual a 0, existe una raíz única con valor $-c / b$.

3. En los demás casos, utilizaremos la fórmula siguiente:

$$x_i = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

La expresión $d = b^2 - 4ac$ se denomina discriminante.

- Si d es mayor o igual que 0 entonces hay dos raíces reales.

- Si d es menor que 0 entonces hay dos raíces complejas de la forma:

$$x + yi,\ x - yi$$

Indicar con literales apropiados, los datos a introducir, así como los resultados obtenidos.

La solución de este problema puede ser de la siguiente forma:

- Primero definimos las variables que vamos a utilizar en los cálculos.

```
double a, b, c;  /* coeficientes de la ecuación */
double d;        /* discriminante */
double re, im;   /* parte real e imaginaria de la raíz */
```

- A continuación leemos los datos a, b y c.

```
printf("Coeficientes a, b y c de la ecuación: ");
scanf("%lf %lf %lf", &a, &b, &c);
```

- Leídos los coeficientes, pasamos a calcular las raíces.

```
if (a == 0 && b == 0)
  printf("La ecuación es degenerada\n");
else if (a == 0)
  printf("La única raíz es: %.2lf\n", -c / b);
else
{
  /* Evaluar la fórmula. Cálculo de d, re e im */
  if (d >= 0)
  {
    /* Imprimir las raíces reales */
  }
  else
  {
    /* Imprimir las raíces complejas conjugadas */
  }
}
```

- Cálculo de $\frac{-b}{2a} \pm \frac{\sqrt{b^2-4ac}}{2a}$ (re, im)

```
re = -b / (2 * a);
d = b * b - 4 * a * c;
im = sqrt(fabs(d)) / (2 * a);
```

- Imprimir las raíces reales.

```
printf("Raíces reales:\n");
printf("%.2lf    %.2lf\n", re + im, re - im);
```

- Imprimir las raíces complejas conjugadas.

```
printf("Raíces complejas:\n");
printf("%.2lf + %.2lf i\n", re, fabs(im));
printf("%.2lf - %.2lf i\n", re, fabs(im));
```

El programa completo se muestra a continuación.

```
/******. Calcular las raíces de una ecuación de 2º grado ******/
/* ecu2gra.c
 */
#include <stdio.h>
#include <math.h>

void main()
{
  double a, b, c;  /* coeficientes de la ecuación */
  double d;        /* discriminante */
  double re, im;   /* parte real e imaginaria de la raíz */

  printf("Coeficientes a, b y c de la ecuación: ");
```

```
    scanf("%lf %lf %lf", &a, &b, &c);
    printf("\n\n");
    if (a == 0 && b == 0)
      printf("La ecuación es degenerada\n");
    else if (a == 0)
      printf("La única raíz es: %.2lf\n", -c / b);
    else
    {
      re = -b / (2 * a);
      d = b * b - 4 * a * c;
      im = sqrt(fabs(d)) / (2 * a);
      if (d >= 0)
      {
        printf("Raíces reales:\n");
        printf("%.2lf    %.2lf\n", re + im, re - im);
      }
      else
      {
        printf("Raíces complejas:\n");
        printf("%.2lf + %.2lf i\n", re, fabs(im));
        printf("%.2lf - %.2lf i\n", re, fabs(im));
      }
    }
}
```

2. Escribir un programa para que lea un texto y dé como resultado el número de palabras con al menos cuatro vocales diferentes. Suponemos que una palabra está separada de otra por uno o más espacios (' '), tabuladores (\t) o caracteres nueva línea (\n). La entrada de datos finalizará cuando se detecte la marca de fin de fichero. La ejecución será de la forma siguiente:

```
Introducir texto. Para finalizar introducir la marca EOF

En la universidad hay muchos
estudiantes de telecomunicación
^Z  marca de fin de fichero
Número de palabras con 4 vocales distintas: 3
```

La solución de este problema puede ser de la siguiente forma:

- Primero definimos las variables que vamos a utilizar en el programa.

```
int np = 0;  /* número de palabras con 4 vocales distintas */
int a = 0, e = 0, i = 0, o = 0, u = 0;
char car;
```

- A continuación leemos el texto carácter a carácter.

```
printf("Introducir texto. Para finalizar introducir la marca EOF\n\n");
```

```
while ((car = getchar()) != EOF)
{
  /*
     Si el carácter leído es una 'a' hacer a = 1
     Si el carácter leído es una 'e' hacer e = 1
     Si el carácter leído es una 'i' hacer i = 1
     Si el carácter leído es una 'o' hacer o = 1
     Si el carácter leído es una 'u' hacer u = 1
     Si el carácter leído es un espacio en blanco,
     un \t o un \n, acabamos de leer una palabra. Entonces,
     si a+e+i+o+u >= 4 incrementar el contador de palabras
     de cuatro vocales diferentes y poner a, e, i, o y u de
     nuevo a cero.
   */
} /* fin del while */
```

- Si la marca de fin de fichero está justamente a continuación de la última palabra (no se pulsó ↵ después de la última palabra), entonces se sale del bucle **while** sin verificar si esta palabra tenía o no cuatro vocales diferentes. Por eso este proceso hay que repetirlo fuera del **while**.

```
if ((a + e + i + o + u) >= 4) np += 1;
```

- Finalmente, escribimos el resultado.

```
printf("\nNúmero de palabras con 4 vocales distintas: %d", np);
```

El programa completo se muestra a continuación.

```
/******** Palabras con cuatro o más vocales diferentes ********/
/* vocales.c
 */
#include <stdio.h>
void main()
{
  int np = 0;  /* número de palabras con 4 vocales distintas */
  int a = 0, e = 0, i = 0, o = 0, u = 0;
  char car;

  printf("Introducir texto. Para finalizar introducir la marca EOF\n\n");
  while ((car = getchar()) != EOF)
  {
    switch (car)
    {
      case 'A': case 'a': case 'á':
        a = 1;
        break;
      case 'E': case 'e': case 'é':
        e = 1;
        break;
      case 'I': case 'i': case 'í':
        i = 1;
        break;
```

```
        case 'O': case 'o': case 'ó':
          o = 1;
          break;
        case 'U': case 'u': case 'ú':
          u = 1;
          break;
        default:
          if (car == ' ' || car == '\t' || car == '\n')
          {
            if ((a + e + i + o + u) >= 4) np += 1;
            a = e = i = o = u = 0;
          }
      } /* fin del switch */
    } /* fin del while */
    if ((a + e + i + o + u) >= 4) np += 1;
    printf("\nNúmero de palabras con 4 vocales distintas: %d", np);
}
```

3. Escribir un programa para que lea un texto y dé como resultado el número de caracteres, el número de palabras y el número de líneas del mismo. Suponemos que una palabra está separada de otra por uno o más espacios (' '), caracteres tab (\t) o caracteres nueva línea (\n). La ejecución será de la forma siguiente:

```
Introducir texto. Pulse Entrar después de cada línea.
Para finalizar introducir la marca EOF.

Este programa cuenta los caracteres, las palabras y
las líneas de un documento.
•
80 13 2
```

El programa completo se muestra a continuación. Como ejercicio analice paso a paso el código del programa y justifique la solución anterior presentada como ejemplo.

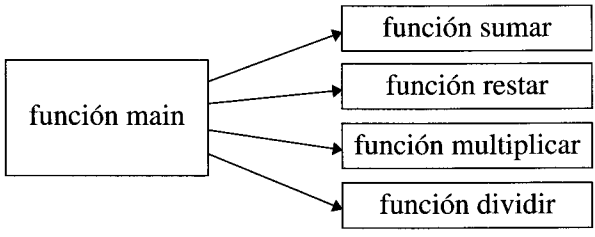
```
/****** Contar caracteres, palabras y líneas en un texto ******/
/* palabras.c
 */
#include <stdio.h>

void main()    /* función principal */
{
  const int SI = 1;
  const int NO = 0;

  char car;
  int palabra = NO;
  int ncaracteres = 0, npalabras = 0, nlineas = 0;

  printf("Introducir texto. Pulse Entrar después de cada línea\n");
  printf("Para finalizar introducir la marca EOF.\n\n");
```

```
  while ((car = getchar()) != EOF)
  {
    ++ncaracteres;          /* contador de caracteres */
    /* Eliminar blancos, tabuladores y finales de línea
       entre palabras */
    if (car == ' ' || car == '\n' || car == '\t')
      palabra = NO;
    else if (palabra == NO) /* comienza una palabra */
    {
      ++npalabras;          /* contador de palabras */
      palabra = SI;
    }

    if (car == '\n')        /* finaliza una línea */
      ++nlineas;            /* contador de líneas */
  }
  printf("%d %d %d\n", ncaracteres, npalabras, nlineas);
}
```

4. Realizar un programa que a través de un menú permita realizar las operaciones de *sumar*, *restar*, *multiplicar*, *dividir* y *salir*. Las operaciones constarán solamente de dos operandos y cada una de ellas será realizada por una función que recibirá como parámetros los dos operandos y devolverá el resultado de la operación. El menú también será visualizado por una función sin argumentos, que devolverá como resultado la opción elegida. La ejecución será de la forma siguiente:

```
        1. sumar
        2. restar
        3. multiplicar
        4. dividir
        5. salir

Seleccione la operación deseada: 3

Teclear dato 1: 2.5
Teclear dato 2: 10

Resultado = 25

Pulse <Entrar> para continuar
```

La solución de este problema puede ser de la siguiente forma:



- Primero definimos las variables y los prototipos de las funciones que van a intervenir en el programa.

```
double dato1, dato2, resultado;
int operacion;

/* Prototipos de las funciones */
double sumar(double dato1, double dato2);
double restar(double dato1, double dato2);
double multiplicar(double dato1, double dato2);
double dividir(double dato1, double dato2);
int menu(void);
```

- A continuación presentamos el menú en la pantalla para poder elegir la operación a realizar.

```
operacion = menu();
```

- Si la operación elegida no ha sido *salir*, leemos los operandos *dato1* y *dato2*.

```
if (operacion != 5)
{
  printf("\nTeclear dato 1: ");
  scanf("%lf", &dato1);
  printf("Teclear dato 2: ");
  scanf("%lf", &dato2);
  fflush(stdin);
  /* realizar la operación e imprimir el resultado */
}
else
  break; /* salir */
```

- A continuación, realizamos la operación elegida con los datos leídos e imprimimos el resultado.

```
switch (operacion)
{
  case 1:
    resultado = sumar(dato1, dato2);
    break;
  case 2:
    resultado = restar(dato1, dato2);
    break;
  case 3:
    resultado = multiplicar(dato1, dato2);
    break;
  case 4:
    resultado = dividir(dato1, dato2);
    break;
}
printf("\nResultado = %g\n", resultado);
printf("\nPulse <Entrar> para continuar ");
getchar();
```

- Las operaciones descritas formarán parte de un bucle infinito formado por una sentencia **while** con el fin de poder encadenar distintas operaciones.

```
while (1)
{
  /* sentencias */
}
```

El programa completo, así como las definiciones de las funciones declaradas, se muestra a continuación.

```
/***************** Simulación de una calculadora ***************/
/* calcula.c
 */
#include <stdio.h>
#include <stdlib.h>

void main()
{
  double dato1, dato2, resultado;
  int operacion;

  /* Prototipos de las funciones */
  double sumar(double dato1, double dato2);
  double restar(double dato1, double dato2);
  double multiplicar(double dato1, double dato2);
  double dividir(double dato1, double dato2);
  int menu(void);

  while (1)
  {
    operacion = menu();

    if (operacion != 5)
    {
      printf("\nTeclear dato 1: ");
      scanf("%lf", &dato1);
      printf("Teclear dato 2: ");
      scanf("%lf", &dato2);
      fflush(stdin);

      switch (operacion)
      {
      case 1:
        resultado = sumar(dato1, dato2);
        break;
      case 2:
        resultado = restar(dato1, dato2);
        break;
      case 3:
        resultado = multiplicar(dato1, dato2);
        break;
      case 4:
        resultado = dividir(dato1, dato2);
        break;
      }
```

```
      printf("\nResultado = %g\n", resultado);
      printf("\nPulse <Entrar> para continuar ");
      getchar();
    }
    else
      break;
  }
}

int menu()
{
  int op;
  do
  {
    system("cls");
    printf("\t1.  sumar\n");
    printf("\t2.  restar\n");
    printf("\t3.  multiplicar\n");
    printf("\t4.  dividir\n");
    printf("\t5.  salir\n");
    printf("\nSeleccione la operación deseada: ");
    scanf("%d", &op);
  }
  while (op < 1 || op > 5);
  return op;
}

double sumar(double a, double b)
{
  double c;
  c = a + b;
  return(c);
}

double restar(double a, double b)
{
  double c;
  c = a - b;
  return(c);
}

double multiplicar(double a, double b)
{
  double c;
  c = a * b;
  return(c);
}

double dividir(double a, double b)
{
  double c;
  c = a / b;
  return(c);
}
```

EJERCICIOS PROPUESTOS

1. Realizar un programa que calcule e imprima la suma de los múltiplos de 5 comprendidos entre dos valores *a* y *b*. El programa no permitirá introducir valores negativos para *a* y *b*, y verificará que *a* es menor que *b*. Si *a* es mayor que *b*, intercambiará estos valores.

2. Realizar un programa que permita evaluar la serie:

$$\sum_{a=0}^{b} \frac{1}{x+ay}$$

3. Si quiere averiguar su número de Tarot, sume los números de su fecha de nacimiento y a continuación redúzcalos a un solo dígito; por ejemplo si su fecha de nacimiento fuera *17 de Octubre de 1970*, los cálculos a realizar serían:

$$17 + 10 + 1970 = 1997 \Rightarrow 1 + 9 + 9 + 7 = 26 \Rightarrow 2 + 6 = 8$$

lo que quiere decir que su número de Tarot es el 8.

Realizar un programa que pida una fecha, de la forma:

dd de *mm* de *aaaa*

donde *dd*, *mm* y *aaaa* son enteros, y de como resultado el número de Tarot. El programa verificará si la fecha es correcta.

4. Realizar un programa que genere la siguiente secuencia de dígitos:

```
                    1
                  2 3 2
                3 4 5 4 3
              4 5 6 7 6 5 4
            5 6 7 8 9 8 7 6 5
          6 7 8 9 0 1 0 9 8 7 6
        7 8 9 0 1 2 3 2 1 0 9 8 7
      8 9 0 1 2 3 4 5 4 3 2 1 0 9 8
    9 0 1 2 3 4 5 6 7 6 5 4 3 2 1 0 9
  0 1 2 3 4 5 6 7 8 9 8 7 6 5 4 3 2 1 0
1 2 3 4 5 6 7 8 9 0 1 0 9 8 7 6 5 4 3 2 1
2 3 . . . . . .
```

El número de filas estará comprendido entre 11 y 20 y el resultado aparecerá centrado en la pantalla como se indica en la figura.

5. Un centro numérico es un número que separa una lista de números enteros (comenzando en 1) en dos grupos de números, cuyas sumas son iguales. El primer centro numérico es el 6, el cual separa la lista (1 a 8) en los grupos: (1, 2, 3, 4, 5) y (7, 8) cuyas sumas son ambas iguales a 15. El segundo centro numérico es el 35, el cual separa la lista (1 a 49) en los grupos: (1 a 34) y (36 a 49) cuyas sumas son ambas iguales a 595. Escribir un programa que calcule los centros numéricos entre 1 y *n*.

6. Realizar un programa que solicite un texto (suponer que los caracteres que forman el texto son solamente letras, espacios en blanco, comas y el punto como final del texto) y a continuación lo escriba modificado de forma que, a la A le corresponda la K, a la B la L, ... , a la O la Y, a la P la Z, a la Q la A, ... y a la Z la J, e igual para las letras minúsculas. Suponga que la entrada no excede de una línea y que finaliza con un punto.

   Al realizar este programa tenga en cuenta que el tipo **char** es un tipo entero, por lo tanto las afirmaciones en los ejemplos siguientes son correctas:

   - 'A' es menor que 'a', que es equivalente a decir que 65 es menor que 97, porque el valor ASCII de 'A' es 65 y el de 'a' es 97.

   - 'A' + 3 es igual a 'D', que es equivalente a decir que 65 + 3 es igual a 68 y este valor es el código ASCII del carácter 'D'.

CAPÍTULO 6

TIPOS ESTRUCTURADOS DE DATOS

Las variables que hemos manipulado hasta ahora pueden almacenar sólo un valor cada vez. ¿Qué hacemos entonces para almacenar un conjunto de valores? Por ejemplo, si quisiéramos calcular la temperatura media del mes de agosto tendríamos que introducir 31 valores utilizando una variable, uno cada vez, y acumular la suma en otra variable. Pero ¿qué ocurre con los valores que introducimos? que cuando introducimos el segundo valor, el primero se pierde; cuando introducimos el tercero, el segundo se pierde, y así sucesivamente. Cuando hayamos introducido todos podremos calcular la media, pero las temperaturas correspondientes a cada día se habrán perdido. Si quisiéramos conservar todos los valores, tendríamos que utilizar 31 variables diferentes lo que incrementaría enormemente el código.

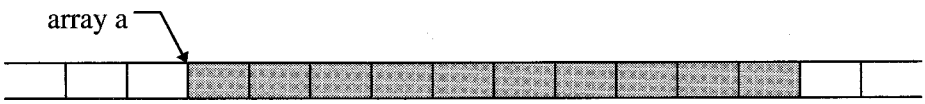
En este capítulo, aprenderá a coleccionar valores del mismo tipo en unas variables especiales llamadas *arrays*. Así mismo, aprenderá a coleccionar caracteres en *arrays de caracteres*; esto es, a trabajar con cadenas de caracteres.

Los arrays permiten entonces coleccionar valores todos del mismo tipo (**int**, **float**, **double**, **char**, etc.). Entonces ¿qué hacemos para almacenar un conjunto de valores de diferentes tipos, relacionados entre sí? Por ejemplo, si quisiéramos almacenar los datos relativos a una persona como su *nombre*, *dirección*, *teléfono*, etc. tendríamos que utilizar tantas variables diferentes como datos y la manipulación de los mismos como puede ser una simple copia en otra parte, incrementaría enormemente el código.

En este capítulo, aprenderá también a agrupar datos de diferentes tipos en unas variables especiales llamadas *estructuras*.

ARRAYS

Un array es una estructura homogénea, compuesta por varios elementos, todos del mismo tipo y almacenados consecutivamente en memoria. Cada elemento puede ser accedido directamente por el nombre de la variable array seguido de uno o más subíndices encerrados entre corchetes.



La representación de los arrays se hace mediante variables suscritas o de subíndices y pueden tener una o varias dimensiones (subíndices). A los arrays de una dimensión se les llama también listas y a los de dos dimensiones, tablas.

Desde el punto de vista matemático, en más de una ocasión necesitaremos utilizar variables, tales como:

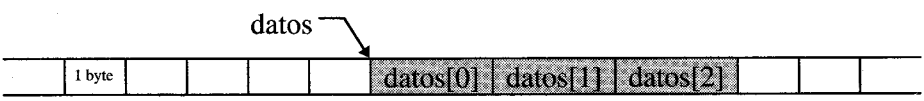
$$a_1 \quad a_2 \quad a_3 \ \ldots \ a_i \ \ldots \ a_n$$

en el caso de un subíndice, o bien

$$\begin{array}{l} a_{11} \ a_{12} \ a_{13} \ \ldots \ a_{1j} \ \ldots \ a_{1n} \\ a_{21} \ a_{22} \ a_{23} \ \ldots \ a_{2j} \ \ldots \ a_{2n} \\ \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \\ a_{i1} \ a_{i2} \ a_{i3} \ \ldots \ a_{ij} \ \ldots \ a_{in} \end{array}$$

si se utilizan dos subíndices. Para realizar esta misma representación en C, tendremos que recurrir a los arrays que acabamos de definir y que a continuación se estudian.

Por ejemplo, supongamos que tenemos un array unidimensional de enteros llamado *datos*, el cual contiene tres elementos. Estos elementos se identificarán de la siguiente forma:



Observe que los subíndices son enteros consecutivos, y que el primer subíndice vale 0. Un subíndice puede ser cualquier expresión entera.

Así mismo, un array de dos dimensiones se representa mediante una variable con dos subíndices (filas, columnas); un array de tres dimensiones se representa mediante una variable con tres subíndices etc. El número máximo de dimensiones

o el número máximo de elementos para un array depende de la memoria disponible.

DECLARACIÓN DE UN ARRAY

La declaración de un array especifica el nombre del array, el número de elementos del mismo y el tipo de éstos. Según su dimensión, cabe distinguir entre arrays unidimensionales y arrays multidimensionales.

Arrays unidimensionales

La declaración de un array de una dimensión, se hace así:

```
tipo nombre[tamaño];
tipo nombre[];
```

donde *tipo* indica el tipo de los elementos del array, los cuales pueden ser de cualquier tipo excepto **void**; *nombre* es un identificador que nombra al array; y *tamaño* es una constante entera que especifica el número de elementos del array. Los corchetes modifican la definición normal del identificador para que sea interpretado por el compilador como un array.

El tamaño puede omitirse cuando se inicializa el array, cuando se declara como un parámetro formal en una función o cuando se hace referencia a un array declarado en otra parte del programa. Vea algunos ejemplos mostrados a continuación.

```
int lista[100]; /* definición del array lista con 100 elementos */
```

El ejemplo anterior declara una variable array denominada *lista* con 100 elementos (subindicados del 0 al 99), cada uno de ellos de tipo **int**. El primer elemento es *lista[0]* (se lee *lista sub-cero*), el segundo *lista[1]*, etc.

```
char nombre[40]; /* definición del array nombre con 40 elementos */
```

Este otro ejemplo declara una variable array denominada *nombre* con 40 elementos (subindicados del 0 al 39), cada uno de ellos de tipo **char**.

```
extern int vector[]; /* declaración del array vector */
```

Este ejemplo declara el array *vector* con elementos de tipo **int**. La definición actual de *vector* tiene que estar hecha en otra parte del programa.

```
void fnEscribir(int a[], int n)
{
  /* cuerpo de la función */
}
```

Este ejemplo define una función *fnEscribir* con un parámetro *a* que es un array de enteros. Cuando la función sea invocada, como primer parámetro se pasará un array de enteros unidimensional. Veremos esto con detalle más adelante.

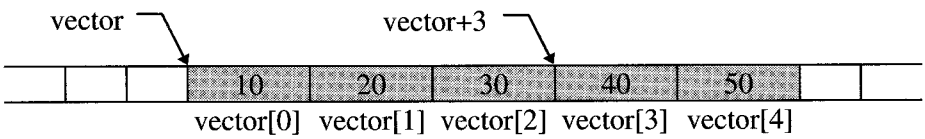
Un array puede ser inicializado en el momento de su definición así:

```
int vector[5] = {10, 20, 30, 40, 50};
```

o también así:

```
int vector[] = {10, 20, 30, 40, 50};
```

La diferencia entre las definiciones anteriores está en que en la última, no se ha indicado explícitamente el número de elementos. En este caso el compilador lo calcula por el número de valores especificados.



La figura anterior nos dice algo más; *vector*, que es el nombre del array, es la dirección simbólica donde se localiza el array en memoria. Para acceder a un elemento se utiliza un subíndice que le indica al compilador cuántos elementos a partir de *vector* hay que desplazarse para localizar dicho elemento. Así, para acceder al elemento *vector[0]* hay que desplazarse 0 elementos a partir de *vector*, para acceder a *vector[1]* hay que desplazarse un elemento a partir de *vector*, y así sucesivamente. El número de bytes que hay que desplazarse lo calcula el compilador en función del tipo de los elementos del array; por ejemplo, en un ordenador de 16 bits si el tipo de los elementos es **int**, avanzar un elemento equivale a desplazarse dos bytes.

Según lo expuesto podemos afirmar que *el nombre de un array se corresponde con la dirección de comienzo del array.*

Un elemento de un array se puede utilizar exactamente igual que una variable. Por ejemplo, en las operaciones que se muestran a continuación intervienen variables subindicadas; esto es, elementos de un array:

```
int lista[100], k = 0, a = 0;
// ...
```

```
a = lista[1] + lista[99];
k = 50;
lista[k] += 1;
lista[k+1] = lista[k];
```

Observe que para referenciar un elemento de un array se puede emplear como subíndice una constante, una variable o una expresión de tipo entero.

Para practicar la teoría expuesta hasta ahora, vamos a realizar un programa que asigne datos a un array unidimensional *a* de *N_ELEMENTOS* elementos y, a continuación, escriba el contenido de dicho array. La solución será similar a la siguiente:

```
Introducir los valores del array.
a[0]= 1
a[1]= 2
a[2]= 3
...

1  2  3 ...

Fin del proceso.
```

Para ello, en primer lugar definimos la constante *N_ELEMENTOS* que indica el número de elementos del array.

```
#define N_ELEMENTOS 31
```

A continuación declaramos el array *a* y el subíndice *i* para acceder a los elementos del array.

```
int a[N_ELEMENTOS];   /* array a */
int i = 0;            /* subíndice */
```

El paso siguiente es asignar un valor desde el teclado a cada elemento del array.

```
for (i = 0; i < N_ELEMENTOS; i++)
{
  printf("a[%d]= ", i);
  scanf("%d", &a[i]);
}
```

Una vez leído el array lo visualizamos para comprobar el trabajo realizado.

```
for (i = 0; i < n; i++)
  printf("%d  ", a[i]);
```

El programa completo se muestra a continuación.

```
/************* Creación de un array unidimensional *************/
/* array01.c
 */
#include <stdio.h>
#define N_ELEMENTOS 31 /* número de elementos del array */

void main()
{
  int a[N_ELEMENTOS];  /* array a */
  int i = 0;           /* subíndice */

  printf("Introducir los valores del array.\n");

  for (i = 0; i < N_ELEMENTOS; i++)
  {
    printf("a[%d]= ", i);
    scanf("%d", &a[i]);
  }

  /* Salida de datos */
  printf("\n\n");
  for (i = 0; i < N_ELEMENTOS; i++)
    printf("%d  ", a[i]);
  printf("\n\nFin del proceso.\n");
}
```

Cuando definimos un array, el número de elementos del mismo tiene que ser especificado por una constante entera y no por una variable. Esto es, un intento de definir una array como se indica a continuación daría lugar a un error:

```
int n = 0;
printf("Número de elementos del array: ");
scanf("%d", &n);
int a[n];
```

Error 1: las declaraciones tienen que ir al principio.
Error 2: n tiene que ser una constante entera.

Con un compilador ANSI C, el ejemplo anterior presenta dos errores, uno debido a que hay sentencias escritas antes de una declaración y las declaraciones deben estar al principio (en C++ no sucede lo mismo) y otro, que para definir un array el número de elementos tiene que ser especificado por una constante entera.

Para trabajar con un número de elementos variable, lo que hay que hacer es definir el array con el número máximo de elementos que preveamos que va tener, por ejemplo *N_ELEMENTOS*, y utilizar después sólo *n* elementos, siendo *n* menor o igual que *N_ELEMENTOS*.

Como ejemplo vamos a modificar el programa anterior para que ahora permita trabajar con un número *n* variable de elementos del array *a*, siendo *n* menor o igual que *N_ELEMENTOS*.

Para ello, en primer lugar definimos la constante *N_ELEMENTOS* que indica el número máximo de elementos del array.

```
#define N_ELEMENTOS 50
```

A continuación declaramos las variables:

```
int a[N_ELEMENTOS];   /* array a */
int i = 0;            /* subíndice */
int n = 0;            /* número de elementos leídos */
```

El paso siguiente es asignar un valor desde el teclado a cada elemento del array. La entrada de datos se dará por finalizada cuando hayamos asignado valores a todos los elementos del array (*n* igual a *N_ELEMENTOS*), o cuando tecleemos un valor no numérico, en cuyo caso **scanf** devolverá un cero.

```
printf("a[%d]= ", n);
while (n < N_ELEMENTOS && scanf("%d", &a[n]))
{
  n++;
  printf("a[%d]= ", n);
}
```

Una vez leído el array lo visualizamos para comprobar el trabajo realizado.

```
for (i = 0; i < n; i++)
  printf("%d  ", a[i]);
```

El programa completo se muestra a continuación.

```
/************* Creación de un array unidimensional *************/
/* array02.c
 */

#include <stdio.h>
#define N_ELEMENTOS 50 /* máximo número de elementos del array */

void main()
{
  int a[N_ELEMENTOS];   /* array a */
  int i = 0;            /* subíndice */
  int n = 0;            /* número de elementos leídos */

  printf("Introducir los valores del array.\n");
  printf("La entrada finalizará cuando se hayan introducido\n");
  printf("el total de los elementos o cuando se introduzca\n");
  printf("un valor no numérico.\n\n");

  printf("a[%d]= ", n);
  while (n < N_ELEMENTOS && scanf("%d", &a[n]))
  {
    n++;
```

```
    printf("a[%d]= ", n);
  }
  fflush(stdin);

  /* Salida de datos */
  printf("\n\n");
  for (i = 0; i < n; i++)
    printf("%d  ", a[i]);
  printf("\n\nFin del proceso.\n");
}
```

La ejecución de este programa presenta un aspecto como la siguiente:

```
Introducir los valores del array.
La entrada finalizará cuando se hayan introducido
el total de los elementos o cuando se introduzca
un valor no numérico.

a[0]= 1
a[1]= 2
a[2]= 3
a[3]= fin


1  2  3

Fin del proceso.
```

Para este problema no es necesario llamar a la función **fflush** después de ejecutar por última vez la función **scanf**, pero el hacerlo significa que tenemos conocimiento de que en el *buffer* de entrada hay información que puede ser indeseable cuando en otros casos, haya que ejecutar otras funciones de entrada.

El ejercicio anterior nos enseña cómo leer un array y cómo escribirlo. El paso siguiente es aprender a trabajar con los valores almacenados en el array. Por ejemplo, pensemos en un programa que lea las notas correspondientes a los alumnos de un determinado curso, las almacene en un array y dé como resultado la nota media del curso.

Igual que hemos visto en los programas anteriores, en primer lugar definiremos el array con un número máximo de elementos. En este caso nos interesa que el array sea de tipo **float** para que sus elementos puedan almacenar un valor real. También definiremos un índice para acceder a los elementos del array, una variable que contenga el número de alumnos y otra para almacenar la suma total de todas las notas.

```
float notas[ALUM_MAX]; /* array notas */
int i = 0;             /* índice */
int nalumnos = 0;      /* número de alumnos */
float suma = 0;        /* suma total de todas las notas */
```

A continuación preguntamos al usuario del programa por el número de alumnos y obligamos a que este valor sea mayor que cero y menor o igual que el número máximo de elementos del array.

```
do
{
  printf("Número de alumnos: ");
  scanf("%d", &nalumnos);
}
while (nalumnos < 1 || nalumnos > ALUM_MAX);
```

Después, almacenaremos en el array las notas introducidas desde el teclado.

```
for (i = 0; i < nalumnos; i++)
{
  printf("Alumno número %3d, nota final: ", i+1);
  scanf("%f", &notas[i]);
}
```

El paso siguiente es sumar todas las notas. Utilizaremos para ello la variable *suma*. Una variable utilizada de esta forma recibe el nombre de acumulador. Es importante que observe que inicialmente su valor es cero.

```
for (i = 0; i < nalumnos; i++) suma += notas[i];
```

Por último, calculamos la media y la visualizamos.

```
printf("\n\nNota media del curso: %5.2f\n", suma / nalumnos);
```

El programa completo se muestra a continuación.

```
/******************** Nota media del curso ********************/
/* notas.c
 */

#include <stdio.h>
#define ALUM_MAX 100  /* número máximo de alumnos */

void main()
{
  float notas[ALUM_MAX]; /* array notas */
  int i = 0;             /* índice */
  int nalumnos = 0;      /* número de alumnos */
  float suma = 0;        /* suma total de todas las notas */

  do
  {
    printf("Número de alumnos: ");
    scanf("%d", &nalumnos);
  }
  while (nalumnos < 1 || nalumnos > ALUM_MAX);
```

```
  /* Entrada de datos */
  for (i = 0; i < nalumnos; i++)
  {
    printf("Alumno número %3d, nota final: ", i+1);
    scanf("%f", &notas[i]);
  }

  /* Sumar las notas */
  for (i = 0; i < nalumnos; i++)
    suma += notas[i];

  /* Escribir resultados */
  printf("\n\nNota media del curso: %5.2f\n", suma / nalumnos);
}
```

Los dos bucles **for** del programa anterior podrían reducirse a uno como se indica a continuación. No se ha hecho por motivos didácticos.

```
for (i = 0; i < nalumnos; i++)
{
  printf("Alumno número %3d, nota final: ", i+1);
  scanf("%f", &notas[i]);
  suma += notas[i];
}
```

Arrays multidimensionales

Un array multidimensional, como su nombre indica, es un array de dos o más dimensiones. La declaración de un array de varias dimensiones se hace así:

```
tipo nombre [expr-cte-1][expr-cte-2]...;
tipo nombre [][expr-cte]...;
```

El número de elementos de un array multidimensional es el producto de las dimensiones indicadas por *expr-cte-1*, *expr-cte-2*, ... Por ejemplo,

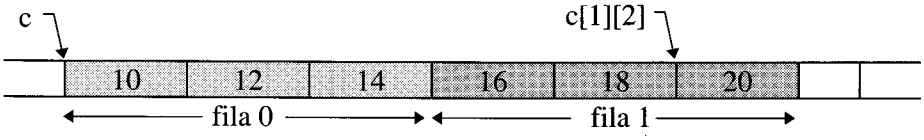
```
int a[2][3][4][5][3];
```

Este ejemplo define un array *a* de cinco dimensiones con 2×3×4×5×3 = 360 elementos.

La primera *expr-cte* puede omitirse para los mismos casos que expusimos al hablar de arrays unidimensionales; esto es, cuando se inicializa el array, cuando se declara como un parámetro formal en una función o cuando se hace referencia a un array declarado en otra parte del programa. Por ejemplo,

```
int c[][3] = {10, 12, 14, 16, 18, 20};
```

Los valores en un array se almacenan por filas. Según esto, el ejemplo anterior define un array *c* de 2 filas por 3 columnas; total 6 elementos. Su disposición en memoria se vería así:



Desde nuestro punto de vista, cuando se trate de arrays de dos dimensiones, es más fácil pensar en ellos como si de una tabla de *m* filas por *n* columnas se tratara. Por ejemplo,

c

	columna 0	columna 1	columna 2
fila 0	10	12	14
fila 1	16	18	20

Para acceder a los elementos del array *c*, puesto que se trata de un array de dos dimensiones, utilizaremos dos subíndices, el primero indicará la fila y el segundo la columna donde se localiza el elemento. Según esto, los elementos del array *c* son:

c

	columna 0	columna 1	columna 2
fila 0	c[0][0]	c[0][1]	c[0][2]
fila 1	c[1][0]	c[1][1]	c[1][2]

Análogamente a lo expuesto para los arrays unidimensionales, para acceder a un elemento en un array de dos dimensiones, se utiliza un subíndice que le indica al compilador cuántas filas hay que desplazarse, y otro que le indica cuántos elementos hay que avanzar en la fila actual, para situarse en dicho elemento. Así, para acceder al elemento *c[1][2]* hay que desplazarse a partir de *c* 1 fila, y avanzar 2 elementos sobre la fila actual. En definitiva, el cálculo que hace el compilador para saber cuántos elementos tiene que avanzar para acceder a un elemento cualquiera *c[fila][col]* en un array de dos dimensiones es:

$$fila \times elementos\ por\ fila + col$$

Como ejemplo de aplicación de arrays multidimensionales, vamos a realizar un programa que asigne datos a un array *c* de dos dimensiones y a continuación escriba las sumas correspondientes a las filas del array. La ejecución del programa presentará el aspecto siguiente:

```
Número de filas del array: 2
Número de columnas del array: 2
c[0][0] = 2
c[0][1] = 5
c[1][0] = 3
c[1][1] = 6


Fila 0, suma = 7
Fila 1, suma = 9
```

Para ello, en primer lugar definimos como constante el número máximo de filas y de columnas del array.

```
#define FILAS_MAX 10  /* número máximo de filas */
#define COLS_MAX  10  /* número máximo de columnas */
```

A continuación declaramos las variables:

```
float c[FILAS_MAX][COLS_MAX];  /* array c de dos dimensiones */
float sumafila;          /* suma de los elementos de una fila */
int filas, cols;     /* filas y columnas del array de trabajo */
int fila, col;        /* fila y columna del elemento accedido */
```

Después, leemos el número de filas y de columnas que en realidad vamos a utilizar, comprobando que estos valores estén dentro del rango permitido; esto es:

```
do
{
  printf("Número de filas del array: ");
  scanf("%d", &filas);
}
while (filas < 1 || filas > FILAS_MAX);
do
{
  printf("Número de columnas del array: ");
  scanf("%d", &cols);
}
while (cols < 1 || cols > COLS_MAX);
```

El paso siguiente es asignar un valor desde el teclado a cada elemento del array.

```
for (fila = 0; fila < filas; fila++)
{
  for (col = 0; col < cols; col++)
  {
    printf("c[%d][%d] = ", fila, col);
    scanf("%f", &c[fila][col]);
  }
}
```

Una vez leído el array lo visualizamos para comprobar el trabajo realizado.

```
for (fila = 0; fila < filas; fila++)
{
  sumafila  = 0;
  for (col = 0; col < cols; col++)
    sumafila += c[fila][col];
  printf("Fila %d, suma = %g\n", fila, sumafila);
}
```

El programa completo se muestra a continuación.

```
/********* Suma de las filas de un array bidimensional *********/
/* arraybi.c
 */
#include <stdio.h>
#define FILAS_MAX 10  /* número máximo de filas */
#define COLS_MAX  10  /* número máximo de columnas */

void main()
{
  float c[FILAS_MAX][COLS_MAX];  /* array c de dos dimensiones */
  float sumafila;          /* suma de los elementos de una fila */
  int filas, cols;     /* filas y columnas del array de trabajo */
  int fila, col;       /* fila y columna del elemento accedido */

  do
  {
    printf("Número de filas del array: ");
    scanf("%d", &filas);
  }
  while (filas < 1 || filas > FILAS_MAX);

  do
  {
    printf("Número de columnas del array: ");
    scanf("%d", &cols);
  }
  while (cols < 1 || cols > COLS_MAX);

  /* Entrada de datos */
  for (fila = 0; fila < filas; fila++)
  {
    for (col = 0; col < cols; col++)
    {
      printf("c[%d][%d] = ", fila, col);
      scanf("%f", &c[fila][col]);
    }
  }

  /* Escribir la suma de cada fila */
  printf("\n\n");
  for (fila = 0; fila < filas; fila++)
  {
    sumafila  = 0;
    for (col = 0; col < cols; col++)
```

```
      sumafila += c[fila][col];
    printf("Fila %d, suma = %g\n", fila, sumafila);
  }
}
```

Seguramente habrá pensado que la suma de cada fila se podía haber hecho simultáneamente a la lectura tal como se indica a continuación.

```
for (fila = 0; fila < filas; fila++)
{
  sumafila  = 0;
  for (col = 0; col < cols; col++)
  {
    printf("c[%d][%d] = ", fila, col);
    scanf("%f", &c[fila][col]);
    sumafila += c[fila][col];
  }
  printf("Fila %d, suma = %g\n", fila, sumafila);
}
```

No obstante, esta forma de proceder presenta una diferencia a la hora de visualizar los resultados, y es que la suma de cada fila se presenta a continuación de haber leído los datos de la misma.

```
Número de filas del array: 2
Número de columnas del array: 2
c[0][0] = 2
c[0][1] = 5
Fila 0, suma = 7
c[1][0] = 3
c[1][1] = 6
Fila 1, suma = 9
```

Arrays asociativos

Cuando el índice de un array es a su vez un dato, se dice que el array es asociativo. En estos casos, la solución del problema se hace más fácil si utilizamos esta coincidencia. Por ejemplo, vamos a realizar un programa que cuente el número de veces que aparece cada una de las letras de un texto introducido por el teclado y a continuación imprima el resultado. Para hacer el ejemplo sencillo, vamos a suponer que el texto sólo contiene letras minúsculas del alfabeto inglés (no hay ni letras acentuadas, ni la ll, ni la ñ). La solución podría ser de la forma siguiente:

```
Introducir texto. Para finalizar introducir la marca EOF

los arrays más utilizados son los unidimensionales
y los bidimensionales.
•
  a  b  c  d  e  f  g  h  i  j  k  l  m  n  o  p  q  r  s  t  u  v  w  x  y  z
 ---------------------------------------------------------------------------
  5  1  0  3  4  0  0  0  8  0  0  6  3  6  7  0  0  2 11  1  2  0  0  0  2  1
```

Antes de empezar el problema vamos a ver algunos de los detalles que después utilizaremos en el programa. Por ejemplo:

```
#define N_ELEMENTOS 'z'-'a'+1 /* N_ELEMENTOS = 26 */
```

La directriz anterior define la constante *N_ELEMENTOS* con el valor 26. Recuerde, cada carácter tiene asociado un valor entero (código ASCII) que es el que utiliza la máquina internamente para manipularlo. Así por ejemplo la 'z' tiene asociado el entero 122, la 'a' el 97, etc. Según esto, la interpretación que ha hecho el preprocesador C de la directriz anterior es:

```
#define N_ELEMENTOS 122-97+1 /* N_ELEMENTOS = 26 */
```

Por la misma razón, si realizamos las declaraciones:

```
int c[256], car = 'a'; /* car tiene asignado el valor 97 */
```

la siguiente sentencia asigna a *c[97]* el valor diez,

```
c['a'] = 10;
```

y esta otra sentencia que se muestra a continuación realiza la misma operación, porque *car* tiene asignado el carácter 'a'.

```
c[car] = 10;
```

Entonces, si leemos un carácter (de la 'a' a la 'z')

```
car = getchar();
```

y a continuación realizamos la operación

```
c[car]++;
```

¿qué elemento del array *c* se ha incrementado? La respuesta es, el de índice igual al código correspondiente al carácter leído. Hemos hecho coincidir el carácter leído con el índice del array. Así cada vez que leamos una 'a' se incrementará el contador *c[97]* o lo que es lo mismo *c['a']*; tenemos un contador de 'a'. Análogamente diremos para el resto de los caracteres.

Pero ¿qué pasa con los elementos *c[0]* a *c[96]*? Según hemos planteado el problema inicial (con qué frecuencia aparecen los caracteres de la 'a' a la 'z') quedarían sin utilizar. Esto, aunque no presenta ningún problema, se puede evitar así:

```
c[car - 'a']++;
```

Para *car* igual a 'a' se trataría del elemento *c[0]* y para *car* igual a 'z' se trataría del elemento *c[26]*. De esta forma podemos definir un array de enteros justamente con un número de elementos igual al número de caracteres de la 'a' a la 'z'. El primer elemento será el contador de 'a', el segundo el de 'b', y así sucesivamente.

Un contador es una variable que inicialmente vale cero (suponiendo que la cuenta empieza desde uno) y que después se incrementa en una unidad cada vez que ocurre el suceso que se desea contar.

El programa completo se muestra a continuación.

```
/************ Frecuencia de las letras en un texto ************/
/* letras.c
 */
#include <stdio.h>
#define N_ELEMENTOS 'z'-'a'+1  /* número de elementos */

void main()
{
  int c[N_ELEMENTOS]; /* array c */
  char car; /* índice */

  /* Poner los elementos del array a cero */
  for (car = 'a'; car <= 'z'; car++)
    c[car - 'a'] = 0;

  /* Entrada de datos y cálculo de la tabla de frecuencias */
  printf("Introducir texto. Para finalizar introducir la marca EOF\n\n");
  while ((car = getchar()) != EOF)
  {
    /* Si el carácter leído está entre la 'a' y la 'z'
       incrementar el contador correspondiente */
    if (car >= 'a' && car <= 'z')
      c[car - 'a']++;
  }

  /* Escribir la tabla de frecuencias */
  for (car = 'a'; car <= 'z'; car++)
    printf("  %c", car);
  printf("\n  ------------------------------------"
         "-----------------------------\n");

  for (car = 'a'; car <= 'z'; car++)
    printf("%3d", c[car - 'a']);
  putchar('\n');
}
```

Arrays internos static

Las variables **static** internas son locales a una función, pero a diferencia de las variables automáticas (**auto**) su existencia es permanente, en lugar de definirse al

activarse la función y eliminarse al finalizar la misma. Esto es, las variables **static** internas proporcionan un medio de almacenamiento permanente y privado en una función y al igual que las variables globales son inicializadas automáticamente a cero. Hay otra diferencia, las variables automáticas se definen en un área de memoria denominada pila (*stack*), mientras que las variables estáticas se definen en el área de memoria destinada a los datos globales.

Según lo expuesto, si en el programa anterior definimos el array *c* estático, no es necesario poner sus elementos a cero. Por ejemplo,

```
void main()
{
  static int c[N_ELEMENTOS]; /* array c */
  char car; /* índice */

  /* Entrada de datos y cálculo de la tabla de frecuencias */
  // ...
}
```

En un compilador C para MS-DOS, si el array definido en el interior de la función es **auto** y es muy grande, puede ser que necesite aumentar el tamaño de la pila (vea las opciones del enlazador de su compilador). En UNIX no tendrá estos problemas.

Copiar un array en otro

La biblioteca de C no incluye una función que permita copiar un array en otro, excepto para los arrays de caracteres o cadenas de caracteres que sí incluye una, que veremos más adelante. Por lo tanto, para copiar un array en otro tendremos que añadir a nuestro programa el código correspondiente.

Como ejemplo, vamos a realizar un programa que lea un array *a* de dos dimensiones, copie el array *a* en otro array *c*, y visualice el array *c* por filas.

Leer un array y escribirlo ya lo hemos hecho en programas anteriores. Copiar un array en otro es un proceso similar pero utilizando una sentencia de asignación. Así, para copiar un array *a* en un array *c* podemos proceder de la forma siguiente:

```
/* Copiar el array a en c */
for (fila = 0; fila < FILAS; fila++)
{
  for (col = 0; col < COLS; col++)
    c[fila][col] = a[fila][col];
}
```

El programa completo se muestra a continuación.

```
/****************** Copiar un array en otro ******************/
/* copiar.c
 */
#include <stdio.h>
#define FILAS 3
#define COLS 3

void main()
{
  static float a[FILAS][COLS], c[FILAS][COLS];
  int fila = 0, col = 0;

  /* Leer datos para el array a */
  for (fila = 0; fila < FILAS; fila++)
  {
    for (col = 0; col < COLS; col++)
    {
      printf("a[%d][%d] = ", fila, col);
      scanf("%f", &a[fila][col]);
    }
  }
  /* Copiar el array a en c */
  for (fila = 0; fila < FILAS; fila++)
  {
    for (col = 0; col < COLS; col++)
      c[fila][col] = a[fila][col];
  }
  /* Escribir los datos del array c */
  for (fila = 0; fila < FILAS; fila++)
  {
    /* Escribir una fila */
    for (col = 0; col < COLS; col++)
      printf("%10g", c[fila][col]);
    printf("\n"); /* fila siguiente */
  }
}
```

Según hemos dicho anteriormente, la biblioteca de C proporciona una función para copiar arrays de caracteres. Análogamente, podemos realizar una función que copie un array en otro y modificar el programa anterior para que el proceso de copiar lo haga utilizando esta función.

Una función que copie un array en otro tiene que tener dos parámetros, el array destino y el array origen. Según esto, podría ser así:

```
void CopiarArray( float destino[][COLS], float origen[][COLS] )
{
  int fila = 0, col = 0;
  for (fila = 0; fila < FILAS; fila++)
  {
    for (col = 0; col < COLS; col++)
      destino[fila][col] = origen[fila][col];
  }
}
```

Al hablar de arrays multidimensionales dijimos que la primera dimensión de un array se podía omitir cuando se declara el array como un parámetro formal en una función; y eso es lo que hemos hecho en la función *CopiarArray*. Un inconveniente de esta función es que no es autónoma, porque depende de las constantes *FILAS* y *COLS*. En el capítulo dedicado a funciones aprenderemos a solucionar este problema. Las sentencias que forman el cuerpo de la función tienen el mismo aspecto que las utilizadas en el programa anterior para copiar el array.

El programa completo se muestra a continuación.

```
/******************* Copiar un array en otro ******************/
/* copiar02.c
 */
#include <stdio.h>
#define FILAS 3
#define COLS 3
void CopiarArray( float destino[][COLS], float origen[][COLS] );

void main()
{
  static float a[FILAS][COLS], c[FILAS][COLS];
  int fila = 0, col = 0;

  /* Leer datos para el array a */
  for (fila = 0; fila < FILAS; fila++)
  {
    for (col = 0; col < COLS; col++)
    {
      printf("a[%d][%d] = ", fila, col);
      scanf("%f", &a[fila][col]);
    }
  }
  /* Copiar el array a en c */
  CopiarArray(c, a);
  /* Escribir los datos del array c */
  for (fila = 0; fila < FILAS; fila++)
  {
    /* Escribir una fila */
    for (col = 0; col < COLS; col++)
      printf("%10g", c[fila][col]);
    printf("\n"); /* fila siguiente */
  }
}

void CopiarArray( float destino[][COLS], float origen[][COLS] )
{
  int fila = 0, col = 0;
  for (fila = 0; fila < FILAS; fila++)
  {
    for (col = 0; col < COLS; col++)
      destino[fila][col] = origen[fila][col];
  }
}
```

Características generales de los arrays

El lenguaje C no comprueba los límites de un array. Es responsabilidad del programador el realizar este tipo de operaciones. Por ejemplo, el siguiente código inicializa el índice *i* en cero y controla que el último elemento que puede ser leído es el *a[99]*.

```
int a[100];
int i = 0;
while (i < 100 && scanf("%d", &a[i])) i++;
```

Para dimensionar un array se pueden emplear constantes o expresiones a base de constantes de cualquier tipo entero. Por ejemplo,

```
#define FILAS 3
// ...
static float a[FILAS][FILAS * 2 + 1];
```

Para acceder a un elemento de un array, se hace mediante el nombre del array seguido de uno o más subíndices, dependiendo de las dimensiones del mismo, cada uno de ellos encerrado entre corchetes. Un subíndice puede ser una constante, una variable o una expresión cualquiera. Por ejemplo,

```
a[1][0] = a[f++][c+k];
```

La sentencia anterior equivale a ejecutar las dos sentencias siguientes:

```
a[1][0] = a[f][c+k];
f++;
```

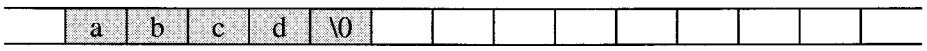
CADENAS DE CARACTERES

Una cadena de caracteres es un array unidimensional, en el que todos sus elementos son de tipo **char**.

Igual que sucedía con los arrays numéricos, un array de caracteres puede ser inicializado en el momento de su definición. Por ejemplo,

```
char cadena[] = "abcd";
printf("%s\n", cadena);
```

Este ejemplo define el array de caracteres *cadena* con cinco elementos (*cadena[0]* a *cadena[4]*) y asigna al primer elemento el carácter 'a', al segundo el carácter 'b', al tercero el carácter 'c', al cuarto el carácter 'd' y al quinto el carácter nulo (valor ASCII 0 o secuencia de escape \0), con el que C finaliza todas las cadenas de caracteres.



Observe que el carácter nulo de terminación lo añade C automáticamente, por lo tanto no es necesario especificarlo en la inicialización. La llamada a **printf** con la especificación de formato *%s* permite visualizar la cadena. Se visualizan todos los caracteres que hay desde el primero hasta el carácter nulo. Esto es, una función capaz de manipular una cadena sabe dónde empieza por el nombre del array y sabe que termina en el primer carácter nulo que encuentre según avance sobre la misma, byte a byte.

Puesto que cada carácter es un entero entre 0 y 255, las dos líneas anteriores son equivalentes a las siguientes:

```
char cadena[] = {97, 98, 99, 100, 0}; /* 'a' = 97, ... */
printf("%s\n", cadena);
```

Esto es así porque un array de caracteres es un array de enteros (los datos de tipo **char** son un subconjunto de los enteros); cada carácter tiene asociado un entero entre 0 y 255. Por la misma razón, las dos líneas anteriores pueden escribirse también así:

```
char cadena[] = {'a', 'b', 'c', 'd', '\0'};
printf("%s\n", cadena);
```

Si se especifica el tamaño del array de caracteres y la cadena asignada es más larga que el tamaño especificado, se obtiene un error en el momento de la compilación indicándolo. Por ejemplo:

```
char cadena[3] = "abcd";
```

Este ejemplo daría lugar a un mensaje de error, indicándonos que hemos excedido los límites del array.

Si la cadena asignada es más corta que el tamaño del array de caracteres, el resto de los elementos del array son inicializados con caracteres nulos.

Según lo expuesto, si queremos leer una cadena de caracteres desde el teclado utilizando la función **scanf** con la especificación de formato *%s*, hay que declarar primero un array de tipo **char** de tamaño igual al número de caracteres máximo que puede contener la cadena, más uno correspondiente al carácter nulo de terminación. Por ejemplo, si queremos leer un nombre de 40 caracteres de longitud máxima, debemos primero declarar el array y después leerlo, así:

```
char nombre[41];
scanf("%s", nombre);
printf("%s\n", nombre);
```

En este caso, la variable *nombre* no necesita ser precedida por el operador **&**, porque como ya hemos dicho anteriormente, el identificador de un array es la dirección de comienzo del array. En este caso, *nombre* es la dirección simbólica de comienzo de la cadena de caracteres.

Ahora, si ejecuta las sentencias anteriores y realiza una entrada como las siguiente,

```
Francisco Javier↵
```

no se sorprenda cuando al visualizar la cadena vea que sólo se escribe *Francisco*. Recuerde que la función **scanf** lee datos delimitados por espacios en blanco. Para solucionar este problema **scanf** admite una especificación de formato personalizada que tiene la sintaxis siguiente:

```
%[caracteres]
```

Esta especificación de formato indica leer caracteres hasta que se lea uno que no esté especificado en el conjunto indicado por *caracteres*. Lo inverso sería

```
%[^caracteres]
```

que indica leer caracteres hasta que se lea uno que esté especificado en el conjunto indicado por *caracteres*. Por ejemplo,

```
scanf("%[^\n]", nombre);
```

La sentencia anterior leerá caracteres de la entrada estándar hasta encontrar un carácter \n (↵). Para este caso hay una solución más sencilla que es utilizar la función **gets** de la biblioteca de C y que veremos más adelante.

Otra forma de leer esta cadena sería carácter a carácter, así:

```
int i = 0;
char nombre[41];
while ((nombre[i++] = getchar()) != '\n' && i < 41)
    ; ← sentencia nula
nombre[i] = 0;
printf("%s\n", nombre);
```

En el capítulo tres se indicó que una sentencia nula estaba formada por un punto y coma. En este ejemplo puede ver su utilización. Se ha escrito en una sola línea por didáctica; lo normal es escribirla a continuación del paréntesis cerrado de la condición de la sentencia **while**.

Si para leer un solo carácter utiliza **scanf** en lugar de **getchar**, no se olvide de anteponer el operador **&**. Por ejemplo,

```
scanf("%c", &nombre[i]);
```

El nombre de un array es la dirección de comienzo del array pero *nombre[i]* hace referencia al contenido de ese elemento, no a la dirección.

Leer una cadena de caracteres

La función de la biblioteca de C **gets** lee una línea de la entrada estándar, **stdin**, y la almacena en la cadena de caracteres especificada. Su sintaxis es la siguiente:

```
#include <stdio.h>
char *gets(char *var);
Compatibilidad: ANSI, UNIX y MS-DOS
```

La variable *var* representa la cadena de caracteres que contendrá todos los caracteres tecleados, excepto el carácter nueva línea (*\n*), que es automáticamente reemplazado por el carácter nulo (*\0*), con el cual C finaliza toda cadena de caracteres. El siguiente ejemplo lee y visualiza la cadena de caracteres simbolizada por *nombre*.

```
char nombre[41];
gets(nombre);
printf("%s\n", nombre);
```

Observe que el parámetro *var* esta definido como un puntero a un **char**; esto es, una dirección que hace referencia al lugar donde está almacenado un carácter. Esto es así, porque como ya hemos dicho en más de una ocasión, el nombre de un array es la dirección de comienzo del array. Para el caso de un array de caracteres, esa dirección coincide con la dirección del primer carácter; el final del array viene marcado por un carácter nulo.

La función **gets** devuelve un puntero a la cadena de caracteres leída; dicho de otra forma, devuelve la cadena de caracteres leída. Un valor nulo para este puntero, indica un error o una condición de fin de fichero (*eof*). Un puntero nulo viene definido en *stdio.h* por la constante **NULL**.

El siguiente ejemplo lee cadenas de caracteres de la entrada estándar hasta que se introduzca la marca de fin de fichero.

```
char *c = NULL; /* para almacenar el valor retornado por gets */
char cadena[41];
c = gets(cadena);
```

```
while (c != NULL)
{
  /* operaciones con la cadena leída */
  c = gets(cadena);
}
```

La función **gets**, a diferencia de la función **scanf**, permite la entrada de una cadena de caracteres formada por varias palabras separadas por espacios en blanco, sin ningún tipo de formato. Recordar que para **scanf**, el espacio en blanco actúa como separador de datos en la entrada.

Escribir una cadena de caracteres

La función **puts** de la biblioteca de C escribe una cadena de caracteres en la salida estándar, **stdout**, y reemplaza el carácter nulo de terminación de la cadena (*\0*) por el carácter nueva línea (*\n*), lo que quiere decir que después de escribir la cadena, se avanza automáticamente a la siguiente línea.

```
#include <stdio.h>
int *puts(const char *var);
Compatibilidad: ANSI, UNIX y MS-DOS
```

La función **puts** retorna un valor positivo si se ejecuta satisfactoriamente; en caso contrario, retorna el valor **EOF**.

El siguiente ejemplo lee y visualiza la cadena de caracteres simbolizada por *nombre*.

```
char nombre[41];
gets(nombre);
puts(nombre);
```

Este otro ejemplo que se muestra a continuación, utiliza el valor retornado por la función **gets** para visualizar la cadena leída, Así mismo, demuestra como **printf** con una especificación de formato *%s\n* es equivalente a **puts**.

```
#include <stdio.h>
char linea[81];
char *pc;
void main()
{
  printf("Introduce una cadena de caracteres: ");
  pc = gets(linea);
  printf("\nLa línea introducida es:\n");
  printf("%s\n", linea);
  puts("\nLa escribo por segunda vez:");
  puts(pc);
}
```

En el capítulo 4 fue discutido el efecto que produce el carácter *nueva línea* que queda en el *buffer* de entrada al ejecutar la función **scanf** o **getchar**, si a continuación se ejecutaba otra vez cualquiera de ellas con la intención de leer caracteres. Lo que sucedía es que \n es una entrada válida para cualquiera de esas funciones, con lo que parecía que no se ejecutaban. Esto mismo ocurrirá con la función **gets**, si cuando vaya a ejecutarse hay un carácter \n en el *buffer* de entrada, debido a que previamente se ha ejecutado alguna de las funciones mencionadas. La solución a esto es limpiar el *buffer* asociado con **stdin** después de una lectura con las funciones **scanf** o con **getchar**.

El siguiente ejemplo, trata de aclarar los conceptos expuestos en el párrafo anterior. En él se combinan las funciones **scanf**, **getchar** y **gets** para ver la necesidad de utilización de la función **fflush**.

```
/************* Limpiar el buffer asociado con stdin ************/
/* fflush.c
 */
#include <stdio.h>
void main()
{
  int entero;
  double real;
  char respuesta = 's', cadena[81];

  /* Introducir números */
  printf("Introducir un nº entero y un nº real:\n");
  scanf("%d %lf", &entero, &real);
  printf("%d + %f = %f\n\n", entero, real, entero + real);

  /* Leer 4 cadenas de caracteres */
  printf("Introducir 4 cadenas de caracteres para scanf:\n");
  for (entero = 0; entero < 4; entero++)
  {
    scanf("%s", cadena);
    printf("%s\n", cadena);
  }
  /* Limpiar el buffer de entrada y leer una cadena con gets */
  fflush(stdin);

  printf("Introducir cadenas para gets.\n");
  while (respuesta == 's' && gets(cadena) != NULL)
  {
    printf("%s\n", cadena);
    do
    {
      printf("¿ Desea continuar  (s/n) ");
      respuesta = getchar();
      /* Limpiar el buffer de entrada */
      fflush( stdin );
    }
    while ((respuesta != 's') && (respuesta != 'n'));
  }
}
```

Este programa utiliza la función **scanf**, para leer datos numéricos y cadenas de caracteres. Según lo expuesto anteriormente, después de que se ejecute la última llamada a **scanf**, en el *buffer* asociado con la entrada estándar queda un carácter nueva línea. De no limpiar este *buffer*, el carácter nueva línea sería leído por la función **gets** que tiene que ejecutarse a continuación, sustituyendo así, a la primera cadena que tendría que leer dicha función. Con la función **getchar** ocurre el mismo problema, tanto si a continuación se ejecuta ella misma, como si se ejecuta la función **gets**.

Utilización de gets y puts

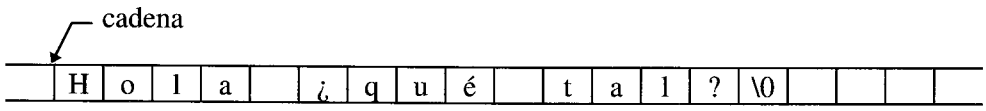
El siguiente ejemplo lee una cadena de caracteres y a continuación visualiza la dirección, el carácter y el valor ASCII de cada uno de los caracteres de la cadena. La solución será de la forma:

```
Escriba una cadena de caracteres:
Hola ¿qué tal?
Dirección = 13180, carácter = 'H', código ASCII =  72
Dirección = 13181, carácter = 'o', código ASCII = 111
...
```

El problema consiste en definir una cadena de caracteres, *cadena*, y asignarle datos desde el teclado utilizando la función **gets**. Una vez leída la cadena, se accede a cada uno de sus elementos (no olvide que son elementos de un array) y por cada uno de ellos se visualiza su dirección, su contenido y el valor ASCII correspondiente. El programa completo se muestra a continuación.

```
/*** Examinar una cadena de caracteres almacenada en memoria ***/
/* cadena.c
 */
#include <stdio.h>

void main()
{
  char cadena[41]; /* almacena la cadena leída */
  int i = 0; /* índice */

  puts("Escriba una cadena de caracteres:");
  gets(cadena);

  /* Examinar la cadena */
  do
  {
    printf("Dirección = %5u, carácter = '%c', código ASCII = %4u\n",
        &cadena[i], cadena[i], cadena[i] & 0x00FF);
    i++;
  }
  while (cadena[i] != '\0');
}
```

Cuando un usuario ejecute este programa, se le solicitará que introduzca una cadena. Por ejemplo,



Observe que el bucle utilizado para examinar la cadena para *i* igual a 0 accede al primer elemento de array, para *i* igual a 1 al segundo, y así hasta llegar al carácter nulo que indica el final de la cadena. Este bucle podría haberse escrito también así:

```
do
{
  printf("Dirección = %5u, carácter = '%c', código ASCII = %4u\n",
      &cadena[i], cadena[i], cadena[i] & 0x00FF);
}
while (cadena[++i] != 0);
```

¿Qué ha cambiado? El índice *i* ahora se incrementa en la propia condición del **while** y antes que ésta se evalúe, porque el operador ++ está como prefijo. También, en lugar del carácter '\0' se ha puesto su valor decimal, cero.

El código del carácter será generalmente ASCII, aunque esta característica depende del sistema que esté utilizando. Ahora ¿por qué se ha utilizado la expresión *cadena[i] & 0x00FF* para escribir dicho código? La respuesta es, porque no hay una especificación de formato para escribir un valor entero en base diez de un byte. Por lo tanto, al utilizar una especificación de formato para un valor **unsigned int**, el sistema convierte implícitamente el valor de tipo **char** (*cadena[i]*) a un valor de tipo **unsigned int**, con extensión del signo; esto es, si un valor de tipo **unsigned int** tiene 2 bytes de longitud, después de la conversión, el byte menos significativo se corresponde con *cadena[i]* y se completa hasta 16 bits con el bit más significativo de *cadena[i]*, el bit que está en la posición del signo. Por ejemplo, el carácter 0x41 sería convertido al valor 0x0041, porque el bit más significativo es 0, con lo que el resultado visualizado es el mismo; en cambio, el carácter 0x82 sería convertido al valor 0xFF82, porque el bit más significativo es 1, que da lugar a un valor decimal diferente. Para visualizar el valor correcto tenemos que filtrar lo ocho bits menos significativos así: *0xFF82 & 0x00FF = 0x0082*.

En el siguiente ejemplo se trata de escribir un programa que lea una línea de la entrada estándar y la almacene en un array de caracteres. A continuación, utilizando una función, deseamos convertir los caracteres escritos en minúsculas, a mayúsculas.

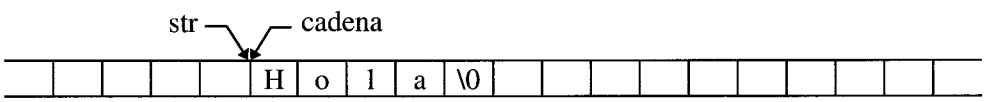
Pasar un carácter de minúsculas a mayúsculas supone restar al valor entero asociado con el carácter la diferencia entre los valores enteros asociados con ambos caracteres, el del carácter en minúscula y el del carácter en mayúscula. La diferencia entre los caracteres en minúsculas y los mismos caracteres en mayúsculas es constante y vale *desp* = *'a' -'A'*. Lógicamente si al valor entero asociado con *'a'* le restamos *desp* nos da *'A'* (97 - 32 = 65). Para más detalles vea el apartado "arrays asociativos" expuesto anteriormente en este mismo capítulo.

La función que realice esta operación recibirá como parámetro el array de caracteres que contiene el texto a convertir. Si la función se llama *MinusculasMayusculas* y el array *cadena*, la llamada será así:

```
MinusculasMayusculas(cadena);
```

A continuación la función accederá al primer elemento del array y comprobará si se trata de una minúscula en cuyo caso, cambiará el valor de dicho elemento por el valor correspondiente a la mayúscula; esto es,

```
if (str[i] >= 'a' && str[i] <= 'z')
  str[i] = str[i] - desp;
```



La función completa es así:

```
void MinusculasMayusculas(char str[])
{
  int i = 0, desp = 'a' - 'A';

  for (i = 0; str[i] != '\0'; ++i)
    if (str[i] >= 'a' && str[i] <= 'z')
      str[i] = str[i] - desp;
}
```

Observe que cuando se llama a la función *MinusculasMayusculas*, lo que en realidad se pasa es la dirección de comienzo del array. Vea en la figura que *str* y *cadena* apuntan a la misma localización de memoria. Por lo tanto todos los cambios que haga la función, los hace sobre el array original.

El programa completo se muestra a continuación.

```
/*********** Conversión de minúsculas a mayúsculas ***********/
/* strupr.c
 */
#include <stdio.h>
#define LONG_MAX 81          /* longitud máxima de la cadena */
```

```
void MinusculasMayusculas(char str[]);

void main() /* función principal */
{
  char cadena[LONG_MAX];
  int i = 0;

  printf ("Introducir una cadena: ");
  gets(cadena);
  MinusculasMayusculas(cadena); /* llamada a la función */
  printf ("%s\n", cadena);
}

/*************************************************************
                  Función MinúsculasMayúsculas
*************************************************************/
/* Convierte minúsculas a mayúsculas */

void MinusculasMayusculas(char str[])
{
  int i = 0, desp = 'a' - 'A';

  for (i = 0; str[i] != '\0'; ++i)
    if (str[i] >= 'a' && str[i] <= 'z')
      str[i] = str[i] - desp;
}
```

FUNCIONES PARA TRABAJAR CON CADENAS DE CARACTERES

La biblioteca de C proporciona un amplio número de funciones que permiten realizar diversas operaciones con cadenas de caracteres, como copiar una cadena en otra, añadir una cadena a otra, etc. A continuación se describen las más utilizadas.

strcat

```
#include <string.h>
char *strcat( char *cadena1, const char *cadena2 );
```
Compatibilidad: ANSI, UNIX y MS-DOS

La función **strcat** añade la *cadena2* a la *cadena1*, finaliza la cadena resultante con el carácter nulo y devuelve un puntero a *cadena1*.

strcpy

```
#include <string.h>
char *strcpy( char *cadena1, const char *cadena2 );
```
Compatibilidad: ANSI, UNIX y MS-DOS

La función **strcpy** copia la *cadena2*, incluyendo el carácter de terminación nulo, en la *cadena1* y devuelve un puntero a *cadena1*.

```
/* Este programa utiliza strcpy y strcat
 * strcpy.c
 */
#include <stdio.h>
#include <string.h>
void main(void)
{
  char cadena[81];
  strcpy( cadena, "Hola " );
  strcat( cadena, "strcpy " );
  strcat( cadena, "y " );
  strcat( cadena, "strcat os saludan!" );
  printf( "cadena = %s\n", cadena );
}
```

Este programa da como solución:

```
cadena = Hola, strcpy y strcat te saludan!
```

strchr

```
#include <string.h>
char *strchr( const char *cadena, int c );
```
Compatibilidad: ANSI, UNIX y MS-DOS

La función **strchr** devuelve un puntero a la primera ocurrencia de *c* en *cadena* o un valor **NULL** si el carácter no es encontrado. El carácter *c* puede ser el carácter nulo ('\0').

strrchr

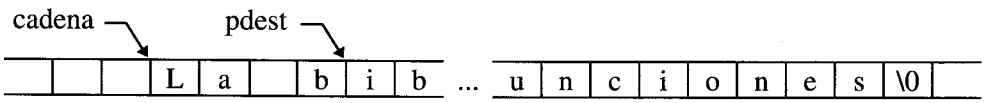
```
#include <string.h>
char *strrchr( const char *cadena, int c );
```
Compatibilidad: ANSI, UNIX y MS-DOS

La función **strrchr** devuelve un puntero a la última ocurrencia de *c* en *cadena* o un valor **NULL** si el carácter no se encuentra. El carácter *c* puede ser un carácter nulo ('\0').

```
/* Este programa ilustra como buscar un carácter con strchr
 * (hacia adelante) o con strrchr (hacia atrás).
/* strchr.c
 */
#include <stdio.h>
#include <string.h>
```

```
void main(void)
{
  int  car = 'i';
  char cadena[] = "La biblioteca de C proporciona muchas funciones";
  char dec1[] =   "         1         2         3         4         5";
  char uni2[] =   "12345678901234567890123456789012345678901234567890";
  char *pdest;    /* valor devuelto por strchr */
  int resu;

  printf( "Cadena en la que se busca: \n%s\n", cadena );
  printf( "%s\n%s\n\n", dec1, uni2 );
  printf( "Buscar el carácter: %c\n\n", car );

  /* Buscar de adelante hacia atrás */
  pdest = strchr( cadena, car );
  resu = pdest - cadena + 1;
  if( pdest != NULL )
    printf( "La %c primera está en la posición %d\n", car, resu );
  else
    printf( "%c no se encuentra en la cadena\n" );

  /* Buscar desde atrás hacia adelante */
  pdest = strrchr( cadena, car );
  resu = pdest - cadena + 1;
  if( pdest != NULL )
    printf( "La última %c está en la posición %d\n\n", car, resu );
  else
    printf( "%c no se encuentra en la cadena\n" );
}
```

Anteriormente dijimos que los elementos de un array de caracteres, igual que los de cualquier otro array, ocupan posiciones sucesivas en memoria, cuestión que comprobamos en el programa *cadena.c*. También dijimos que el nombre de un array es la dirección de comienzo del array y coincide con la dirección del primer carácter (dónde se localiza). Así mismo, observe que el valor retornado por **strchr** y **strrchr** esta definido como un puntero a un **char**; esto es, una dirección que hace referencia al lugar donde está almacenado el carácter que se busca.



Por lo tanto, una sentencia como

```
resu = pdest - cadena + 1;
```

da como resultado la posición 1, 2, 3, ... del carácter buscado dentro de la cadena, que es lo que hace el programa planteado.

El resultado del programa que acabamos de exponer es el siguiente:

```
Cadena en la que se busca:
La biblioteca de C proporciona muchas funciones
         1         2         3         4         5
12345678901234567890123456789012345678901234567890

Buscar el carácter: i

La i primera está en la posición 5
La última i está en la posición 43
```

strcmp

```
#include <string.h>
int strcmp( const char *cadena1, const char *cadena2 );
Compatibilidad: ANSI, UNIX y MS-DOS
```

La función **strcmp** compara la *cadena1* con la *cadena2* lexicográficamente y devuelve un valor:

<0 si la *cadena1* es menor que la *cadena2*,
=0 si la *cadena1* es igual a la *cadena2* y
>0 si la *cadena1* es mayor que la *cadena2*.

En otras palabras, la función **strcmp** nos permite saber si una cadena está en orden alfabético antes (es menor) o después (es mayor) que otra y el proceso que sigue es el mismo que nosotros ejercitamos cuando lo hacemos mentalmente, comparar las cadenas carácter a carácter.

La función **strcmp** diferencia las mayúsculas de las minúsculas. Las mayúsculas están antes por orden alfabético. Esto es así porque en la tabla ASCII las mayúsculas tienen asociado un valor entero menor que las minúsculas.

```
/* strcmp.c */
#include <stdio.h>
#include <string.h>

void main(void)
{
  char cadena1[] = "La provincia de Santander es muy bonita";
  char cadena2[] = "La provincia de SANTANDER es muy bonita";
  char temp[20];
  int resu;

  /* Se diferencian mayúsculas de minúsculas */
  printf("Comparar las cadenas:\n\n%s\n%s\n\n", cadena1, cadena2);
  resu = strcmp( cadena1, cadena2 );
  if( resu > 0 )
    strcpy( temp, "mayor que" );
  else if( resu < 0 )
    strcpy( temp, "menor que" );
```

```
  else
    strcpy( temp, "igual a" );
  printf( "strcmp: cadena 1 es %s cadena 2\n", temp );
}
```

la solución de este problema es que la *cadena1* es mayor que la *cadena2* porque alfabéticamente *Santander* está después de *SANTANDER*.

strcspn

```
#include <string.h>
size_t strcspn( const char *cadena1, const char *cadena2 );
Compatibilidad: ANSI, UNIX y MS-DOS
```

La función **strcspn** da como resultado la posición (subíndice) del primer carácter de *cadena1*, que pertenece al conjunto de caracteres contenidos en *cadena2*. Este valor corresponde a la longitud de la subcadena de *cadena1* formada por caracteres no pertenecientes a *cadena2*. Si ningún carácter de *cadena1* pertenece a *cadena2*, el resultado es la posición del carácter de terminación (\0) de *cadena1*; esto es, la longitud de *cadena1*.

```
/* strcspn.c */
#include <stdio.h>
#include <string.h>

void main(void)
{
   char cadena[] = "xyzabc";
   int  pos;
   pos = strcspn( cadena, "abc" );
   printf("Primer a, b o c en %s es el carácter %d\n", cadena, pos);
}
```

La solución de este programa es (recuerde que la primera posición es la 0):

```
Primer a, b o c en xyzabc es el carácter 3
```

strlen

```
#include <string.h>
size_t strlen( char *cadena );
Compatibilidad: ANSI, UNIX y MS-DOS
```

La función **strlen** devuelve la longitud en bytes de *cadena*, no incluyendo el carácter de terminación nulo. El tipo **size_t** es sinónimo de **unsigned int**.

```
/* strlen.c */
#include <stdio.h>
```

```
#include <string.h>
void main(void)
{
  char cadena[80] = "Hola";

  printf("El tamaño de cadena es %d\n", strlen(cadena));
}
```

La solución de este programa es:

```
El tamaño de cadena es 4
```

strncat

```
#include <string.h>
char *strncat( char *cadena1, const char *cadena2, size_t n );
Compatibilidad: ANSI, UNIX y MS-DOS
```

La función **strncat** añade los primeros *n* caracteres de *cadena2* a la *cadena1*, termina la cadena resultante con el carácter nulo y devuelve un puntero a *cadena1*. Si *n* es mayor que la longitud de *cadena2*, se utiliza como valor de *n* la longitud de *cadena2*.

strncpy

```
#include <string.h>
char *strncpy( char *cadena1, const char *cadena2, size_t n );
Compatibilidad: ANSI, UNIX y MS-DOS
```

La función **strncpy** copia *n* caracteres de la *cadena2*, en la *cadena1* (sobreescribiendo los caracteres de *cadena1*) y devuelve un puntero a *cadena1*. Si *n* es menor que la longitud de *cadena2*, no se añade automáticamente un carácter nulo a la cadena resultante. Si *n* es mayor que la longitud de *cadena2*, la *cadena1* es rellenada con caracteres nulos ('\0') hasta la longitud *n*.

strncmp

```
#include <string.h>
int strncmp( const char *cadena1, const char *cadena2, size_t n );
Compatibilidad: ANSI, UNIX y MS-DOS
```

La función **strncmp** compara lexicográficamente los primeros *n* caracteres de *cadena1* y de *cadena2*, distinguiendo mayúsculas y minúsculas, y devuelve un valor:

<0 si la *cadena1* es menor que la *cadena2*,
=0 si la *cadena1* es igual a la *cadena2* y
>0 si la *cadena1* es mayor que la *cadena2*.

Si *n* es mayor que la longitud de la *cadena1*, se toma como valor la longitud de la *cadena1*.

strspn

```
#include <string.h>
size_t strspn( const char *cadena1, const char *cadena2 );
Compatibilidad: ANSI, UNIX y MS-DOS
```

La función **strspn** da como resultado la posición (subíndice) del primer carácter de *cadena1*, que no pertenece al conjunto de caracteres contenidos en *cadena2*. Esto es, el resultado es la longitud de la subcadena inicial de *cadena1*, formada por caracteres pertenecientes a *cadena2*.

strstr

```
#include <string.h>
char *strstr( const char *cadena1, const char *cadena2 );
Compatibilidad: ANSI y MS-DOS
```

La función **strstr** devuelve un puntero a la primera ocurrencia de *cadena2* en *cadena1* o un valor **NULL** si la *cadena2* no se encuentra en la *cadena1*.

strtok

```
#include <string.h>
char *strtok( char *cadena1, const char *cadena2 );
Compatibilidad: ANSI, UNIX y MS-DOS
```

La función **strtok** lee la *cadena1* como una serie de cero o más elementos básicos separados por cualquiera de los caracteres expresados en *cadena2*, los cuales son interpretados como delimitadores.

Una vez leído el primer elemento de *cadena1*, para leer el siguiente elemento se llama a **strtok** pasando como primer argumento **NULL**. Observar el ejemplo que se expone a continuación.

Esta función devuelve un puntero por cada elemento en *cadena1*. Cuando no hay más elementos, se devuelve un puntero nulo.

Puede ponerse más de un delimitador entre elemento y elemento. También pueden variarse el conjunto de caracteres que actúan como delimitadores, de una llamada a otra.

```
/********************** Función strtok **********************/
/* strtok.c
 */
#include <stdio.h>
#include <string.h>

void main(void)
{
  char *cadena = "Esta cadena, está formada por varias palabras";
  char *elemento;
  elemento = strtok(cadena," ,");
  while (elemento != NULL)
  {
    printf("%s\n", elemento);
    elemento = strtok(NULL," ,");
  }
}
```

Si ejecuta este programa obtendrá la siguiente solución:

```
Esta
cadena
está
formada
por
varias
palabras
```

La cadena se ha divido en elementos separados por un espacio en blanco o por una coma.

strlwr

```
#include <string.h>
char *_strlwr(char *cadena );
Compatibilidad: MS-DOS
```

La función **strlwr** convierte las letras mayúsculas de cadena, en minúsculas. El resultado es la propia cadena en minúsculas.

strupr

```
#include <string.h>
char *_strupr(char *cadena );
Compatibilidad: MS-DOS
```

La función **strupr** convierte las letras minúsculas de *cadena*, en mayúsculas. El resultado es la propia cadena en mayúsculas.

FUNCIONES PARA CONVERSIÓN DE DATOS

Las funciones de la biblioteca de C que se muestran a continuación permiten convertir cadenas de caracteres a números y viceversa, suponiendo que la conversión sea posible.

atof

```
#include <stdlib.h>
double atof( const char *cadena );
Compatibilidad: ANSI, UNIX y MS-DOS
```

La función **atof** convierte una cadena de caracteres a un valor en doble precisión.

atoi

```
#include <stdlib.h>
int atoi( const char *cadena );
Compatibilidad: ANSI, UNIX y MS-DOS
```

La función **atoi** convierte una cadena de caracteres a un valor entero.

atol

```
#include <stdlib.h>
long atol( const char *cadena );
Compatibilidad: ANSI, UNIX y MS-DOS
```

La función **atol** convierte una cadena de caracteres a un valor entero largo.

Cuando las funciones **atof**, **atoi** y **atol** toman de la variable *cadena* un carácter que no es reconocido como parte de un número, interrumpen la conversión.

```
/* Este programa muestra como los números almacenados como
 * cadenas de caracteres pueden ser convertidos a valores
 * numéricos utilizando las funciones atof, atoi, y atol.
 *
 * atof.c
 */
#include <stdio.h>
#include <stdlib.h>
```

```
void main(void)
{
   char *s = NULL; double x = 0; int i = 0; long l = 0;
   s = "  -3208.15E-13";     /* para ver como trabaja atof */
   x = atof( s );
   printf( "atof: cadena ASCII: %-17s float: %e\n", s, x );
   s = "8.7195642337X120";  /* para ver como trabaja atof */
   x = atof( s );
   printf( "atof: cadena ASCII: %-17s float: %e\n", s, x );
   s = "  -8995 libros";     /* para ver como trabaja atoi */
   i = atoi( s );
   printf( "atoi: cadena ASCII: %-17s int  : %d\n", s, i );
   s = "89954 pesetas";      /* para ver como trabaja atol */
   l = atol( s );
   printf( "atol: cadena ASCII: %-17s long : %ld\n", s, l );
}
```

La solución de este programa es:

```
atof: cadena ASCII:   -3208.15E-13    float: -3.208150e-010
atof: cadena ASCII: 8.7195642337X120  float: 8.719564e+000
atoi: cadena ASCII:   -8995 libros    int  : -8995
atol: cadena ASCII: 89954 pesetas     long : 89954
```

fcvt

```
#include <stdlib.h>
char *fcvt( double valor, int decs, int *pdec, int *signo );
Compatibilidad: UNIX y MS-DOS
```

La función **fcvt** convierte un número real a una cadena de caracteres. La cadena de caracteres será finalizada con el carácter nulo. Esta función devuelve un puntero a la cadena de caracteres.

valor es el número real a convertir.

decs número de dígitos después del punto decimal. Si es necesario, se añaden ceros.

pdec devuelve un puntero a un valor entero que especifica la posición del punto decimal.

signo devuelve un puntero a un valor 0, si el número es positivo; o un valor distinto de 0, si el número es negativo.

```
/* Este programa convierte una constante 3.141592653 a
 * una cadena.
 *
 * fcvt.c
 */
#include <stdio.h>
#include <stdlib.h>
```

```
void main(void)
{
  int  puntodecimal, signo;
  char *cadena;
  double valor = 3.141592653;
  cadena = fcvt( valor, 8, &puntodecimal, &signo );
  printf( "Valor: %2.9f, cadena: '%s', punto decimal: %d, "
          "signo: %d\n", valor, cadena, puntodecimal, signo );
}
```

Si ejecuta este programa obtendrá la siguiente solución:

```
Valor: 3.141592653, cadena: '31415926', punto decimal: 1, signo: 0
```

sprintf

```
#include <stdlib.h>
int sprintf( char *buffer, const char *formato [, argumento] ... );
Compatibilidad: ANSI, UNIX y MS-DOS
```

La función **sprintf** convierte los valores de los argumentos especificados, a una cadena de caracteres que almacena en *buffer*. La cadena de caracteres finaliza con el carácter nulo. Cada argumento es convertido y almacenado de acuerdo con el formato correspondiente que se haya especificado. La descripción de *formato*, es la misma que se especificó para **printf**.

La función **sprintf** devuelve como resultado un entero correspondiente al número de caracteres almacenados en *buffer* sin contar el carácter nulo de terminación. Por ejemplo,

```
/* sprintf.c. Este programa utiliza sprintf para almacenar
 * en buffer la cadena de caracteres:
 *
 *    Salida:
 *            Cadena:    ordenador
 *            Carácter:  /
 *            Entero:    40
 *            Real:      1.414214
 *
 *    Número de caracteres = 72
 */
#include <stdio.h>

void main(void)
{
  char  buffer[200], s[] = "ordenador", c = '/';
  int   i = 40, j;
  float f = 1.414214F;

  j  = sprintf( buffer,     "\tCadena:    %s\n", s );
  j += sprintf( buffer + j, "\tCarácter:  %c\n", c );
```

```
  j += sprintf( buffer + j, "\tEntero:    %d\n", i );
  j += sprintf( buffer + j, "\tReal:      %f\n", f );
  printf( "Salida:\n%s\nNúmero de caracteres = %d\n", buffer, j );
}
```

FUNCIONES PARA CONVERSIÓN DE CARACTERES

Las funciones de la biblioteca de C que se exponen a continuación actúan sobre un entero para dar como resultado un carácter.

toascii

```
#include <ctype.h>
int toascii( int c );
```
Compatibilidad: UNIX y MS-DOS

La función **toascii** pone a 0 todos los bits de *c*, excepto los 7 bits de menor orden. Dicho de otra forma, convierte *c* a un carácter ASCII.

tolower

```
#include <stdlib.h>
int tolower( int c );
```
Compatibilidad: ANSI, UNIX y MS-DOS

La función **tolower** convierte *c* a una letra minúscula, si procede.

toupper

```
#include <stdlib.h>
int toupper( int c );
```
Compatibilidad: ANSI, UNIX y MS-DOS

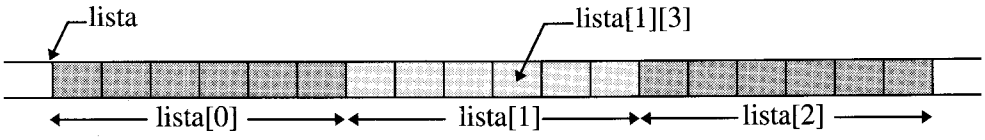
La función **toupper** convierte *c* a una letra mayúscula, si procede.

```
char car;
do
{
  printf("¿Desea continuar?  s/n  ");
  car = getchar();
  fflush(stdin);
}
while (tolower(car) != 'n' && tolower(car) != 's');
```

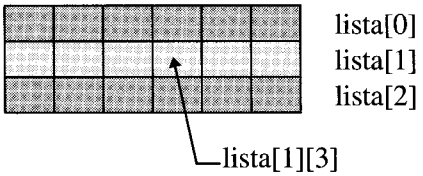
Este ejemplo admite una respuesta si o no (s|S|n|N) en minúsculas o en mayúsculas, pero la comparación se hace en minúsculas.

ARRAYS DE CADENAS DE CARACTERES

Un array de cadenas de caracteres es un array donde cada elemento es a su vez un array de caracteres. Dicho de otra forma, es un array de dos dimensiones de tipo **char**. Su disposición en memoria es de la forma siguiente:



aunque quizá le resulte más sencillo imaginárselo así :



La definición del array de cadenas de caracteres de la figura sería así:

```
char lista[3][6]; /* tres cadenas de seis caracteres cada una */
```

Es importante que asimile que *lista[0]*, *lista[1]* y *lista[2]* son tres cadenas de caracteres y que, por ejemplo, *lista[1][3]* es un carácter; el que está en la fila 1, columna 3.

El siguiente ejemplo declara el array denominado *nombres* como un array de cadenas de caracteres. Esto es, *nombres* es un array de 100 filas, cada una de las cuales es una cadena de caracteres de longitud máxima 60.

```
char nombres[100][60];
```

Para ilustrar la forma de trabajar con cadenas de caracteres, vamos a realizar un programa que lea una lista de nombres y los almacene en un array. Una vez construido el array, visualizaremos su contenido.

La solución tendrá el aspecto siguiente:

```
Escriba los nombres que desea introducir.
Para finalizar introduzca la marca de fin de fichero.

Nombre y apellidos: Mª del Carmen
Nombre y apellidos: Francisco
Nombre y apellidos: Javier
Nombre y apellidos: •
```

```
¿Desea visualizar el contenido del array? (s/n) S

Mª del Carmen
Francisco
Javier
```

La solución pasa por realizar los siguientes puntos:

1. Definir el array de cadenas, los índices y demás variables necesarias.
2. Establecer un bucle para leer las cadenas de caracteres utilizando la función **gets**. La entrada de datos finalizará al introducir la marca de fin de fichero.
3. Preguntar al usuario del programa si quiere visualizar el contenido del array.
4. Si la respuesta anterior es afirmativa, establecer un bucle para visualizar las cadenas de caracteres almacenadas en el array.

El programa completo se muestra a continuación.

```
/****************** Leer una lista de nombres ******************/
/* cadenas.c
 */
#include <stdio.h>
#include <stdlib.h>
#define MAX 10    /* número máximo de filas */
#define LONG 60   /* número máximo de caracteres por fila */

void main(void)
{
  char lista[MAX][LONG];
  char *fin = NULL; /* valor devuelto por gets */
  int i = 0, n = 0; /* índices */
  int resp = 0;     /* respuesta sí o no (s|S|n|N) */

  puts("Escriba los nombres que desea introducir.");
  puts("Para finalizar introduzca la marca de fin de fichero.\n");
  do
  {
    printf("Nombre y apellidos: ");
    fin = gets(lista[i++]);
    /* Cuando gets lea la marca eof, fin es igual a NULL */
  }
  while (fin != NULL && i < MAX);
  clearerr(stdin); /* desactivar el indicador de fin de fichero */

  if (i < MAX) i--; /* eliminar la última entrada: eof */

  do
  {
    printf("¿Desea visualizar el contenido del array? (s/n) ");
    resp = getchar();
    fflush(stdin);
  }
  while (tolower(resp) != 's' && tolower(resp) != 'n');
```

```
  if ( tolower(resp) == 's' )
  {
    /* Escribir el contenido del array de cadenas */
    printf("\n");
    for (n = 0; n < i; n++)
      printf("%s\n", lista[n]);
  }
}
```

Observe la sentencia *fin = gets(lista[i++])*. Es equivalente a

```
fin = gets(lista[i]);
i++;
```

y observe *gets(lista[i])*. El identificador *lista* hace referencia a un array de caracteres de dos dimensiones. Una fila de este array es una cadena de caracteres (un array de caracteres unidimensional) y la biblioteca de C provee la función **gets** para leer cadenas de caracteres. Por eso, para leer una fila (una cadena de caracteres) utilizamos un sólo índice. No sucede lo mismo con los arrays numéricos, porque la biblioteca de C no proporciona funciones para ello.

Siguiendo con el análisis del programa anterior, si la entrada de datos finaliza porque se ha introducido la marca de fin de fichero, el indicador de fin de fichero asociado con la entrada estándar queda activado. Esto da lugar a que no se ejecuten más sentencias que tengan que leer de la entrada estándar, hasta que se desactive dicho indicador. Esta es la razón de utilizar **clearerr** (vea el apartado "carácter fin de fichero" en el Capítulo 4).

Así mismo, la marca de fin de fichero ha sido introducida en respuesta a la llamada a **gets** para almacenar datos en la siguiente fila del array; por lo tanto este último elemento no es una cadena válida. Por eso al finalizar la entrada y siempre que no se hayan asignado datos a la totalidad de los elementos, decrementamos el valor del índice de las filas en una unidad. En el caso de asignar datos a la totalidad de los elementos, no se finaliza con la marca de fin de fichero (*Ctrl+D* en UNIX o *Ctrl+Z* en MS-DOS)

TIPO ARRAY Y TAMAÑO DE UN ARRAY

En el capítulo 2 vimos que utilizando la declaración **typedef** podíamos declarar sinónimos de otros tipos fundamentales o derivados. El tipo array es un tipo derivado (por ejemplo, **int []** o **char [][***C***]**). Pues bien, para declarar un sinónimo de un determinado tipo array proceda de forma análoga a como se indica en el ejemplo siguiente:

```
typedef char tarray2dchar[100][81];
tarray2dchar d;
```

Las dos líneas anteriores son equivalentes a realizar la declaración,

```
char d[100][81];
```

Así mismo, vimos que el operador **sizeof** daba como resultado el tamaño en bytes de su operando. Pues bien, cuando el operando es un array, el resultado es el tamaño en bytes de dicho array. Por ejemplo, el siguiente programa visualiza el tamaño y el número de elementos de cada uno de los arrays definidos en él.

```
/* Tamaño y número de elementos de un array */
/* sizeof.c
 */
#include <stdio.h>
typedef char tarray2dchar[100][81];

void main()
{
  int a[100];
  float b[10][10];
  char c[81];
  tarray2dchar d;

  printf("Tamaño de a: %5d, ", sizeof(a));
  printf("número de elementos de a: %d\n", sizeof(a)/sizeof(int));
  printf("Tamaño de b: %5d, ", sizeof(b));
  printf("número de elementos de b: %d\n", sizeof(b)/sizeof(float));
  printf("Tamaño de c: %5d, ", sizeof(c));
  printf("número de elementos de c: %d\n", sizeof(c)/sizeof(char));
  printf("Tamaño de d: %5d, ", sizeof(d));
  printf("número de elementos de d: %d\n", sizeof(d)/sizeof(char));
}
```

El resultado al ejecutar este programa es el siguiente (el programa se ha compilado y ejecutado en un sistema donde un **int** ocupa dos bytes):

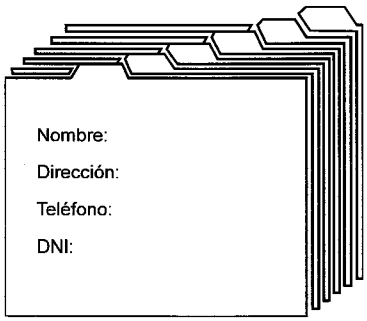
```
Tamaño de a:   200, número de elementos de a: 100
Tamaño de b:   400, número de elementos de b: 100
Tamaño de c:    81, número de elementos de c: 81
Tamaño de d:  8100, número de elementos de d: 8100
```

Puede comprobar estos resultados multiplicando el número de elementos de cada array por el tamaño en bytes de uno de sus elementos.

ESTRUCTURAS

Todas las variables que hemos utilizado hasta ahora han sido de un único tipo, incluso los arrays son variables con todos los elementos del mismo tipo. La finalidad de una estructura es agrupar una o más variables, generalmente de diferentes tipos, bajo un mismo nombre para hacer más fácil su manejo.

El ejemplo típico de una estructura es una ficha que almacena datos relativos a una persona como, nombre, apellidos, dirección, teléfono, etc. En otros compiladores diferentes de C, este tipo de construcciones son conocidas como *registros*.



Algunos de estos datos podrían ser a su vez estructuras. Por ejemplo, la fecha de nacimiento podría ser una estructura con los datos día, mes y año.

Crear una estructura

Para crear una estructura hay que definir un nuevo tipo de datos y declarar una variable de este tipo. La declaración de un tipo estructura, incluye tanto los elementos que la componen como sus tipos. Cada elemento de una estructura recibe el nombre de *miembro* (*campo* del registro). La sintaxis es la siguiente:

```
struct tipo_estructura
{
  /* declaraciones de los miembros */
};
```

donde *tipo_estructura* es un identificador que nombra el nuevo tipo definido. La declaración de un miembro de una estructura no puede contener calificadores de clase de almacenamiento como **extern**, **static**, **auto** o **register** y no puede ser inicializado. Su tipo puede ser: fundamental, array, puntero, unión, estructura o función.

En C una estructura sólo puede contener declaraciones de variables. C++ permite que una estructura contenga, además, miembros que sean funciones.

Las reglas para utilizar el nuevo tipo son las mismas que las seguidas para los tipos predefinidos como **float**, **int** y **char**, entre otros. Esto es, después de definir un tipo estructura, podemos declarar una variable de ese tipo, así:

```
struct tipo_estructura [variable[, variable]...];
```

El siguiente ejemplo clarifica lo expuesto hasta ahora:

```
struct tficha /* declaración del tipo de estructura tficha */
{
  char nombre[40];
  char direccion[40];
  long telefono;
};

struct tficha var1, var2; /* definición de las estructuras var1 y var2 */
```

Este ejemplo define las variables *var1* y *var2*, de tipo **struct** *tficha*; por lo tanto, cada una de las variables es una estructura de datos con los miembros *nombre*, *dirección* y *teléfono*. Observe que en la declaración de *var1* y *var2* se ha especificado la palabra **struct** cuando parece lógico escribir,

```
tficha var1, var2;
```

Esto no es posible en ANSI C, pero si se permite en C++. No obstante utilizando **typedef**, como veremos a continuación, podemos conseguir la forma de declaración anterior. Por ejemplo,

```
struct ficha
{
  char nombre[40];
  char direccion[40];
  long telefono;
};
typedef struct ficha tficha;

tficha var1, var2;
```

La declaración **typedef** anterior declara un sinónimo *tficha* de **struct** *ficha*. Esto mismo puede hacerse de la forma siguiente:

```
typedef struct ficha
{
  char nombre[40];
  char direccion[40];
  long telefono;
} tficha;

tficha var1, var2;
```

La definición de las estructuras *var1* y *var2*, puede realizarse también justamente a continuación de la declaración del nuevo tipo, así:

```
struct tficha
{
  char nombre[40];
  char direccion[40];
  long telefono;
} var1, var2;
```

o también, como se indica a continuación, sin dejar constancia del nuevo tipo declarado, forma que no se aconseja porque posteriormente no podríamos definir otras variables de este tipo.

```
struct
{
  char nombre[40];
  char direccion[40];
  long telefono;
} var1, var2;
```

Para referirse a un determinado miembro de la estructura, se utiliza la notación *estructura.miembro*. Por ejemplo, *var1.telefono* se refiere al miembro *telefono* de la estructura *var1*.

Un miembro de una estructura, se utiliza exactamente igual que cualquier otra variable. Por ejemplo,

```
var1.telefono = 232323;
gets(var2.nombre);
```

La primera sentencia asigna el valor 232323 al miembro *teléfono* de *var1* y la segunda sentencia lee de la entrada estándar información para el miembro *nombre* de la estructura *var2*.

El nombre de un miembro de una estructura es local a la misma y puede ser utilizado solamente después del operador punto o después del operador –> que veremos en el capítulo de punteros. Por ejemplo,

```
#include <stdio.h>

typedef struct ficha
{
  char nombre[40];
  char direccion[40];
  long telefono;
} tficha;

void main(void)
{
  tficha var1;
  char nombre[40] = "Javier";
  int ficha = 1;

  gets(var1.nombre);

  printf("%s\n", var1.nombre);
  printf("%s\n", nombre);
  printf("%d\n", ficha);
}
```

En este programa se ha declarado el tipo *tficha*. Es bueno declarar el nuevo tipo a nivel global para que después podamos utilizarlo en cualquier función del programa. Observe que en la función **main** se ha definido un array con el mismo nombre que un miembro de la estructura y una variable entera *ficha* con el mismo nombre empleado para declarar la estructura. Según lo dicho anteriormente, esto no supone ningún problema. Así, por ejemplo, si ejecuta el programa anterior e introduce el dato *Carmen* para que sea leído por **gets**, el resultado será:

```
Carmen
Javier
1
```

donde observa que no hay conflicto al utilizar identificadores iguales a los utilizados por los miembros de la estructura o por el nombre empleado en la declaración de la misma, lo que corrobora que el nombre de un miembro de una estructura es local a la misma.

Miembros que son estructuras

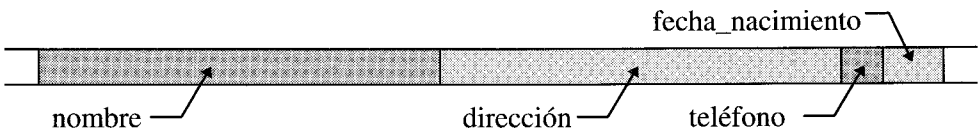
Para declarar un miembro como una estructura, es necesario haber declarado previamente ese tipo de estructura. En particular un tipo de estructura *st* no puede incluir un miembro del mismo tipo *st*, pero sí puede contener un puntero o referencia a un objeto de tipo *st*. Por ejemplo,

```
struct fecha
{
  int dia, mes, anyo;
};

struct ficha
{
  char nombre[40];
  char direccion[40];
  long telefono;
  struct fecha fecha_nacimiento;
};

struct ficha persona;
```

Este ejemplo define la estructura *persona*, en la que el miembro *fecha_nacimiento* es a su vez una estructura. Para acceder, por ejemplo, al miembro *anyo* de *persona* escribiremos,

```
persona.fecha_nacimiento.anyo
```

Los miembros de una estructura son almacenados secuencialmente, en el mismo orden en el que son declarados. Vea la figura siguiente:



Operaciones con estructuras

Con una variable declarada como una estructura, pueden realizarse las siguientes operaciones:

- Inicializarla en el momento de definirla.

```
struct ficha persona = { "Francisco", "Santander 1", 232323, 25, 8, 1982 };
```

- Obtener su dirección por medio del operador **&**.

```
struct ficha *ppersona = &persona;
```

- Acceder a uno de sus miembros.

```
long tel = persona.telefono;
```

- Asignar una estructura a otra con el operador de asignación.

```
struct ficha otra_persona = persona;
```

Cuando se asigna una estructura a otra estructura se copian uno a uno todos los miembros de la estructura fuente en la estructura destino, independientemente de cual sea el tipo de los miembros; esto es, se duplica la estructura.

Por ejemplo, el siguiente programa define la estructura *persona* del tipo *tficha*, asigna datos introducidos a través del teclado a cada uno de sus miembros, copia la estructura *persona* en otra estructura *otra_persona* del mismo tipo y visualiza en pantalla los datos almacenados en esta última estructura.

```
/* estruct.c */
#include <stdio.h>

struct fecha
{
  int dia, mes, anyo;
} tfecha;

struct ficha
{
  char nombre[40];
  char direccion[40];
  long telefono;
  struct fecha fecha_nacimiento;
} tficha;
```

```
void main()
{
  struct ficha persona, otra_persona;

  /* Introducir datos */
  printf("Nombre:          ");
  gets(persona.nombre);
  printf("Dirección:       ");
  gets(persona.direccion);
  printf("Teléfono:        ");
  scanf("%ld", &persona.telefono);
  printf("Fecha de nacimiento:\n");
  printf("  Día:           ");
  scanf("%d", &persona.fecha_nacimiento.dia);
  printf("  Mes:           ");
  scanf("%d", &persona.fecha_nacimiento.mes);
  printf("  Año:           ");
  scanf("%d", &persona.fecha_nacimiento.anyo);

  /* Copiar una estructura en otra */
  otra_persona = persona;

  /* Escribir los datos de la nueva estructura */
  printf("\n\n");
  printf("Nombre:      %s\n", otra_persona.nombre);
  printf("Dirección:   %s\n", otra_persona.direccion);
  printf("Teléfono:    %ld\n", otra_persona.telefono);
  printf("Fecha de nacimiento:\n");
  printf("  Día:       %d\n", otra_persona.fecha_nacimiento.dia);
  printf("  Mes:       %d\n", otra_persona.fecha_nacimiento.mes);
  printf("  Año:       %d\n", otra_persona.fecha_nacimiento.anyo);
}
```

Arrays de estructuras

Cuando los elementos de un array son de tipo estructura, el array recibe el nombre de array de estructuras o array de registros. Esta es una construcción muy útil y potente ya que nos permite manipular los datos en bloques.

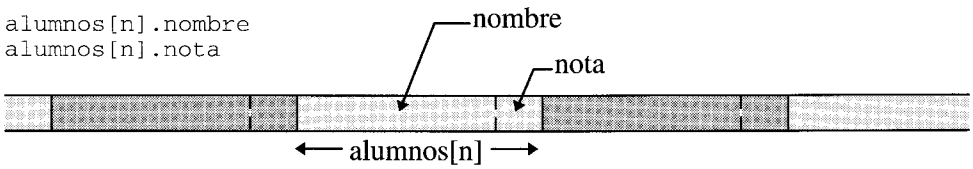
Para definir un array de estructuras, declare el tipo de estructura que coincide con el tipo de los elementos del array; por ejemplo,

```
typedef struct ficha
{
  char nombre[60];
  float nota;
} tficha;
```

y defina el array análogamente a como se hace en el ejemplo siguiente:

```
tficha alumnos[100];
```

Este ejemplo define el array de estructuras denominado *alumnos* con 100 elementos (*alumnos[0]*, *alumnos[1]*, ..., *alumnos[n]*, ..., *alumnos[99]*) cada uno de los cuales es una estructura con los datos miembro *nombre* y *nota*. Para acceder al *nombre* y a la *nota* del elemento *n* del array escribiremos:



Por ejemplo, realizar un programa que lea una lista de alumnos y sus correspondientes notas de final de curso y que dé como resultado el tanto por ciento de los alumnos aprobados y suspendidos.

Los pasos a seguir para realizar este programa son:

- Declarar el tipo de la estructura y definir el array de estructuras. Definir otras variables necesarias.
- Establecer un bucle para leer y almacenar en el array el *nombre* y la *nota* de cada alumno.
- Establecer un bucle para recorrer todos los elementos del array y contar los aprobados (*nota* mayor o igual que 5) y los suspendidos (el resto).

```
for (i = 0; i < n; i++)
   if (alumnos[i].nota >= 5)
     aprobados++;
   else
     suspendidos++;
```

- Escribir el tanto por ciento de aprobados y suspendidos.

El programa completo se muestra a continuación.

```
/********** Calcular el % de aprobados y suspendidos **********/
/* array_st.c
 */
#include <stdio.h>
#define NA 100       /* número máximo de alumnos */

typedef struct ficha
{
  char nombre[60];
  float nota;
} tficha;

void main()
{
  static tficha alumnos[NA]; /*array de estructuras o registros*/
```

```
    int n = 0, i = 0;
    char *fin = NULL; /* para almacenar el valor devuelto por gets */
    int aprobados = 0, suspendidos = 0;

    /* Entrada de datos */
    printf("Introducir datos. ");
    printf("Para finalizar teclear la marca de fin de fichero\n\n");
    printf("Nombre  ");
    fin = gets(alumnos[n].nombre);
    while (n < NA && fin != NULL)
    {
      printf("Nota    ");
      scanf("%f", &alumnos[n++].nota);
      fflush(stdin); /* eliminar el carácter \n */
      /* Siguiente alumno */
      printf("Nombre  ");
      fin = gets(alumnos[n].nombre);
    }

    /* Escribir resultados */
    for (i = 0; i < n; i++)
      if (alumnos[i].nota >= 5)
        aprobados++;
      else
        suspendidos++;

    printf("Aprobados %.3g %%\n", (float)aprobados/n*100);
    printf("Suspendidos %.3g %%\n", (float)suspendidos/n*100);
}
```

Como las variables *aprobados* y *suspendidos* son enteras, para hacer los cálculos del tanto por ciento de aprobados y suspendidos tendremos que convertir explícitamente estas variables al tipo **float** para que los cálculos se hagan en esta precisión. Si no se hace esa conversión explícita, el cociente de la división de enteros que interviene en los cálculos dará siempre cero, excepto cuando el número de aprobados sea *n*, que dará uno, o el número de suspendidos sea *n*, que también dará uno.

UNIONES

Una *unión* es una variable que puede contener, en distintos instantes de la ejecución del programa, datos de diferentes tipos. Esto permite manipular diferentes tipos de datos utilizando una misma zona de memoria, la reservada para la variable *unión*.

La declaración de una unión tiene la misma forma que la declaración de una estructura, excepto que en lugar de la palabra reservada **struct** se pone la palabra reservada **union**. Por lo tanto, todo lo expuesto para las estructuras es aplicable a las uniones, con la excepción de que de los miembros especificados, en un instante sólo está uno presente. La sintaxis para declarar una unión es así:

```
union tipo_union
{
  /* declaraciones de los miembros */
};
```

donde *tipo_union* es un identificador que nombra el nuevo tipo definido.

Después de definir un tipo unión, podemos declarar una o más variables de ese tipo, así:

```
union tipo_union [variable[, variable]...];
```

Para referirse a un determinado miembro de la unión, se utiliza la notación:

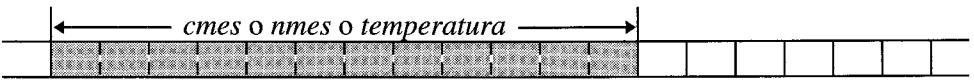
```
variable_unión.miembro
```

El siguiente ejemplo clarifica lo expuesto.

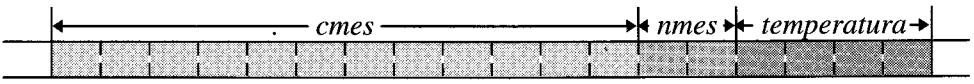
```
union tmes
{
  char  cmes[12];
  int   nmes;
  float temperatura;
};

union tmes var1;
```

Este ejemplo define una unión *var1* que puede almacenar una cadena de caracteres, un entero o un real; un dato sólo, no los tres a la vez.

Por lo tanto, para almacenar los miembros de una unión, se requiere una zona de memoria igual a la que ocupa el miembro más largo de la unión. En el ejemplo anterior, el espacio de memoria reservado para *var1* son 12 bytes, la longitud de la cadena *cmes*. El miembro almacenado ocupará los bytes requeridos por él.



Si *var1* fuera una estructura en lugar de una unión, se requeriría una zona de memoria igual a 12+2+4 bytes, suponiendo que un entero ocupa 2 bytes.

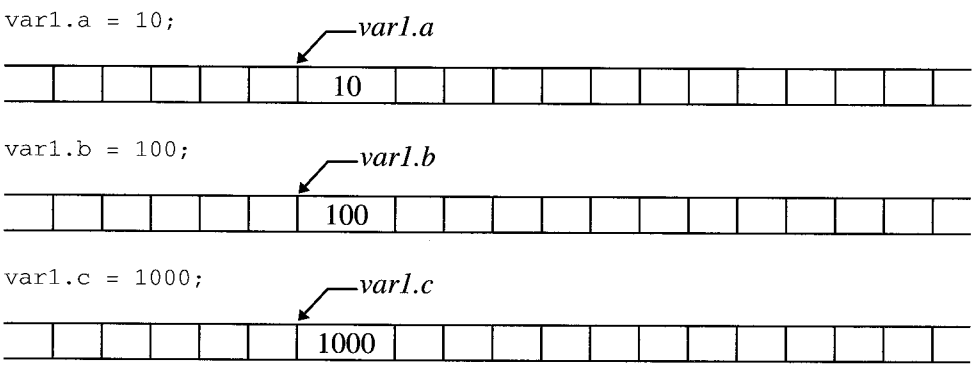


Todos los miembros son almacenados en el mismo espacio de memoria y comienzan en la misma dirección. El valor almacenado es sobreescrito cada vez

que se asigna un valor al mismo miembro o a un miembro diferente. Por ejemplo, ¿qué resultado piensa que da el siguiente programa?

```
#include <stdio.h>

union tunion
{
  int a;
  int b;
  int c;
};

void main()
{
  union tunion var1;

  var1.a = 10;
  var1.b = 100;
  var1.c = 1000;
  printf("%d  ", var1.a);
  printf("%d  ", var1.b);
  printf("%d\n", var1.c);
}
```

Este ejemplo define una unión *var1* que ocupa 2 bytes en memoria, suponiendo que un **int** ocupa dos bytes. La ejecución es de la forma siguiente:



Puesto que el espacio de memoria reservado para la unión *var1* es compartido por los miembros *a*, *b* y *c*, el resultado será el último valor asignado.

```
1000  1000  1000
```

Estructuras variables

Una estructura variable permite utilizar unos miembros u otros en función de las necesidades durante la ejecución. Para ello, alguno de sus miembros tiene que ser una unión.

Por ejemplo, supongamos que deseamos diseñar una ficha para almacenar datos relativos a los libros o revistas científicas de una biblioteca. Por cada libro o revista, figurará la siguiente información:

1. Número de referencia.
2. Título.
3. Nombre del autor.
4. Editorial.
5. Clase de publicación (libro o revista).
6. Número de edición (sólo libros).
7. Año de publicación (sólo libros).
8. Nombre de la revista (sólo revistas).

Esta claro que cada ficha contendrá siempre los miembros 1, 2, 3, 4 y 5 y además, si se trata de un libro, los miembros 6 y 7, o si se trata de una revista, el miembro 8. Esta disyunción da lugar a una unión con dos miembros: una estructura con los miembros 6 y 7 y el miembro 8. Veamos:

```
struct libro
{
  unsigned edicion;
  unsigned anyo;
};

union libro_revista
{
 struct libro libros;
 char nomrev[30];
};
```

Según lo expuesto, el diseño de la ficha quedará de la forma siguiente:

```
struct ficha
{
  unsigned numref;
  char titulo[30];
  char autor[20];
  char editorial[25];
  int clase_publicacion;
  union libro_revista lr;
};
```

La declaración del tipo **struct** *ficha*, declara una estructura variable, apoyándose en una variable *lr* de tipo **unión**. Esta variable contendrá, o bien los datos *edicion* y *anyo*, o bien *nomrev*.

La estructura anterior podría escribirse también de una sola vez así:

```
struct ficha
{
  unsigned numref;
  char titulo[30];
  char autor[20];
  char editorial[25];
  enum clases libro_revista;
  union
  {
    struct
    {
      unsigned edicion;
      unsigned anyo;
    } libros;
    char nomrev[30];
  } lr;
};
```

Como aplicación de lo expuesto vamos a realizar un programa que utilizando la estructura *ficha* anterior permita:

- Almacenar en un array la información correspondiente a la biblioteca.
- Listar dicha información.

La estructura del programa constará de las funciones siguientes:

1. Una función principal **main** que llamará a una función *leer* para introducir los datos para los elementos del array y a una función *escribir* para visualizar todos los elementos del array.

2. Una función *leer* con el prototipo siguiente:

```
int leer(struct ficha bibli[], int n);
```

Esta función recibe como parámetros el array donde hay que almacenar los datos de los libros o revistas leídos y el número máximo de elementos que admite dicho array. La función devolverá como resultado el número de elementos leídos (valor menor o igual que el número máximo de elementos). Cada vez que se introduzcan los datos de un libro o revista, la función visualizará un mensaje preguntando si se quieren introducir más datos.

3. Una función *escribir* con el prototipo siguiente:

```
void escribir(struct ficha bibli[], int n);
```

Esta función recibirá como parámetros el array cuyos elementos hay que visualizar y el número real de elementos que tiene el array. Cada vez que se vi-

sualice un libro o revista se visualizará un mensaje que diga "pulse la tecla <Entrar> para continuar" de forma que al pulsar la tecla *Entrar* se limpie la pantalla y se visualice el siguiente elemento del array.

El programa completo se muestra a continuación. Observe que las funciones dependen sólo de sus parámetros.

```
/************************* BIBLIOTECA ************************/
/* estr_var.c
 */
#include <stdio.h>
#include <stdlib.h>
#define N 100 /* máximo número de elementos de array */

enum clase /* tipo enumerado */
{
  libro, revista
};

typedef struct ficha /* estructura variable */
{
  unsigned numref;
  char titulo[30];
  char autor[20];
  char editorial[25];
  enum clase libro_revista;
  union
  {
    struct
    {
      unsigned edicion;
      unsigned anyo;
    } libros;
    char nomrev[30];
  } lr;
} tficha;

/* Prototipos de funciones */
void escribir(tficha bibli[], int n);
int leer(tficha bibli[], int n);

void main() /* función principal */
{
  static tficha biblioteca[N]; /* array de estructuras */
  int n = 0; /* actual número de elementos del array */

  system("cls");
  printf("INTRODUCIR DATOS\n\n");
  n = leer(biblioteca, N);

  system("cls");
  printf("LISTADO DE LIBROS Y REVISTAS\n");
  escribir(biblioteca, n); /* listar todos los libros y revistas */
}
```

```
/*****************************************************************
      Función para leer los datos de los libros y revistas
*****************************************************************/
int leer(tficha bibli[], int NMAX)
{
  int clase;
  char resp = 's';
  int k = 0; /* número de elementos introducidos */

  while( tolower(resp) == 's' && k < NMAX )
  {
    system("cls");
    printf("Número de refer. ");
    scanf("%u",&bibli[k].numref); fflush(stdin);
    printf("Título           "); gets(bibli[k].titulo);
    printf("Autor            "); gets(bibli[k].autor);
    printf("Editorial        "); gets(bibli[k].editorial);
    do
    {
      printf("Libro o revista (0 = libro, 1 = revista) ");
      scanf("%d", &clase); fflush(stdin);
    }
    while (clase != 0 && clase != 1);
    if (clase == libro)
    {
      bibli[k].libro_revista = libro;
      printf("Edición          ");
      scanf("%u", &bibli[k].lr.libros.edicion);
      printf("Año de public.   ");
      scanf("%u", &bibli[k].lr.libros.anyo); fflush(stdin);
    }
    else
    {
      bibli[k].libro_revista = revista;
      printf("Nombre revista   "); gets(bibli[k].lr.nomrev);
    }

    k++;

    do
    {
      printf("\n\n¿Más datos a introducir? s/n ");
      resp = getchar();
      fflush(stdin);
    }
    while( tolower(resp) != 's' && tolower(resp) != 'n' );
  }
  return k;
}

/*****************************************************************
      Función para listar todos los elementos del array
*****************************************************************/
void escribir(tficha bibli[], int n)
{
  int k = 0;
```

```
  for (k = 0; k < n; k++)
  {
    printf("%d  %s\n", bibli[k].numref, bibli[k].titulo);
    printf("%s - Ed. %s\n", bibli[k].autor,  bibli[k].editorial);

    switch (bibli[k].libro_revista)
    {
      case libro :
       printf("Edición %u - año %u\n",
       bibli[k].lr.libros.edicion,
       bibli[k].lr.libros.anyo);
       break;
      case revista :
       printf("%s\n", bibli[k].lr.nomrev);
    }
    printf("\n\nPulse <Entrar> para continuar");
    getchar();
    system("cls");
  }
}
```

Observe que el tipo *tficha* se ha declarado a nivel global con el fin de poder utilizarlo en cualquier función del programa donde sea necesario.

CAMPOS DE BITS

Un *campo de bits* es un conjunto de bits adyacentes dentro de una unidad direccionable. La sintaxis para declarar un campo de bits es la siguiente:

```
tipo [identificador] : expresión_constante
```

La *expresión constante* especifica el número de bits correspondientes al campo y debe ser un valor entero no negativo. El *tipo* tiene que ser entero. A diferencia de ANSI C, que restringe los campos de bits a tipos **int**, **signed int** o **unsigned int**, C++ permite que un campo de bits sea de cualquier tipo entero; es decir, **char**, **short**, **int**, **long**, con signo o sin signo, o un tipo enumerado. No se permiten arrays de campos de bits, punteros a campos de bits o funciones que retornen un campo de bits. El nombre de un campo es opcional. Los campos sin nombre sirven de relleno.

Es posible definir como miembros de una estructura (no de una unión) campos de bits. El tamaño de un campo de bits no debe sobrepasar el tamaño físico de la palabra máquina; es decir, el espacio ocupado por un entero.

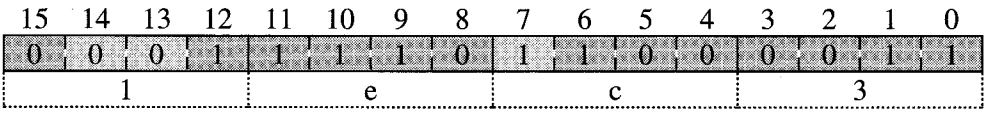
La asignación de los campos de bits depende del hardware. Los campos de bits son asignados del más significativo al menos significativo o del menos significativo al más significativo, caso del ejemplo mostrado a continuación, según la máquina que se emplee.

```
/********************** Campos de bits **********************/
/* campobit.c
 */
#include <stdio.h>

struct palabra            /* palabra de 16 bits - 0 a 15 */
{
  unsigned car_ascii    : 7; /* bits 0 a 6 */
  unsigned bit_paridad  : 1; /* bit 7 */
  unsigned operacion    : 5; /* bits 8 a 12 */
  unsigned              : 2; /* bits 13 a 14 de relleno */
  unsigned bit_signo    : 1; /* bit 15 */
};

void main()  /* función principal */
{
  struct palabra cb = { 'C', 1, 0x1E, 0 };

  printf("campos de bits : %x\n\n", cb);
  printf("bit de signo   : %x\n", cb.bit_signo);
  printf("operación      : %x\n", cb.operacion);
  printf("bit de paridad : %x\n", cb.bit_paridad);
  printf("carácter %c     : %x\n", cb.car_ascii, cb.car_ascii);
}
```

La ejecución de este programa visualiza la siguiente solución:

```
campos de bits : 1ec3

bit de signo   : 0
operación      : 1e
bit de paridad : 1
carácter C     : 43
```

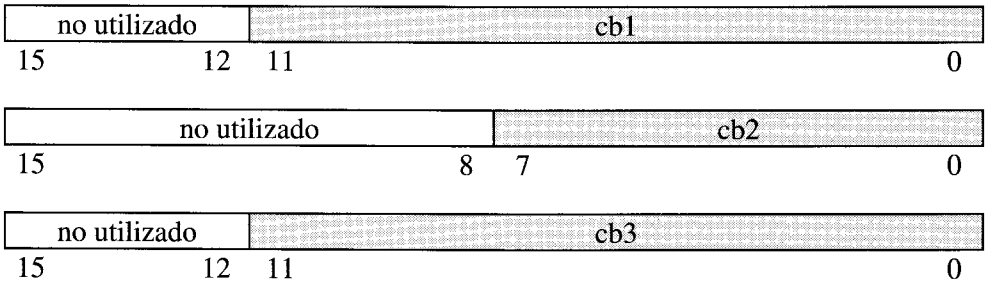
La asignación en memoria se ha efectuado de la forma siguiente:



Si al definir una estructura de campos de bits, el espacio que queda en la unidad direccionable no es suficiente para ubicar un siguiente campo de bits, este se alinea con respecto al siguiente entero. Un campo de bits sin nombre y de tamaño cero, garantiza que el siguiente miembro de la lista comience en el siguiente espacio para un entero. Por ejemplo, considere:

```
struct bits
{
  unsigned cb1 : 12;
  unsigned cb2 : 8;
  unsigned cb3 : 5;
};
```

En un sistema de 16 bits que asigna los campos de bits del menos significativo al más significativo, el objeto se almacenará en palabras adyacentes como éstas:



Utilizar un campo de bits para economizar espacio, es una tarea ingenua y puede conducir a una pérdida de tiempo de ejecución. En muchas máquinas un byte o una palabra es la cantidad de memoria más pequeña que se puede acceder sin tiempo de procesamiento adicional. En cambio, extraer un campo de bits puede suponer instrucciones extra y por lo tanto un tiempo adicional. Quiere esto decir que un campo de bits debe utilizarse en problemas que lo requieran; por ejemplo cuando necesitamos utilizar un conjunto de indicadores de un bit agrupados en un entero (*flags*).

Como aplicación vamos a realizar un programa que solicite introducir un carácter por el teclado y que dé como resultado el carácter reflejado en binario. La solución del problema será análoga a la siguiente:

```
Introduce un carácter: A
carácter A, ASCII 41h, en binario: 01000001

Carácter reflejado:
carácter é, ASCII 82h, en binario: 10000010
```

La estructura del programa constará de las funciones siguientes:

1. Una función principal **main** que llamará a una función *presentar* para visualizar el carácter introducido y el reflejado de la forma expuesta anteriormente (simbólicamente, en hexadecimal y en binario) y a una función *ReflejarByte* que invierta el orden de los bits (el bit 0 pasará a ser el bit 7, el bit 1 pasará a ser el bit 6, el bit 2 pasará a ser el bit 5, etc).

2. Una función *presentar* con el prototipo siguiente:

```
void presentar( unsigned char c );
```

Esta función recibirá como parámetro el carácter que se quiere visualizar y lo presentará simbólicamente, en hexadecimal y en binario.

3. Una función *ReflejarByte* con el prototipo siguiente:

```
unsigned char ReflejarByte( union byte b );
```

Esta función recibirá como parámetro una unión *byte* que permita disponer del carácter como un dato de tipo **unsigned char** o bit a bit, y devolverá como resultado el byte reflejado.

```
union byte
{
  unsigned char byte;
  struct
  {
    unsigned char b0 : 1;
    unsigned char b1 : 1;
    unsigned char b2 : 1;
    unsigned char b3 : 1;
    unsigned char b4 : 1;
    unsigned char b5 : 1;
    unsigned char b6 : 1;
    unsigned char b7 : 1;
  } bits;
};
```

El programa completo se muestra a continuación.

```
/********************** Reflejar un byte **********************/
/* reflejar.c
 */
#include <stdio.h>

union byte
{
  unsigned char byte;
  struct
  {
    unsigned char b0 : 1;
    unsigned char b1 : 1;
    unsigned char b2 : 1;
    unsigned char b3 : 1;
    unsigned char b4 : 1;
    unsigned char b5 : 1;
    unsigned char b6 : 1;
    unsigned char b7 : 1;
  } bits;
};

void presentar( unsigned char c );
unsigned char ReflejarByte( union byte b );
```

```
void main()
{
  union byte b;

  printf("Introduce un carácter: ");
  b.byte = getchar();
  presentar(b.byte);

  printf("\nCarácter reflejado:\n");
  b.byte = ReflejarByte(b);
  presentar(b.byte);
}

void presentar( unsigned char c )
{
  int i = 0;

  printf("carácter %c, ASCII %Xh, en binario: ", c, c);
  /* A continuación se imprime otra vez en binario */
  for (i = 7; i>=0; i--)
    printf("%d", (c & (1 << i)) ? 1 : 0);
  printf("\n");
}

unsigned char ReflejarByte( union byte b )
{
  union byte c;

  c.bits.b0 = b.bits.b7;
  c.bits.b1 = b.bits.b6;
  c.bits.b2 = b.bits.b5;
  c.bits.b3 = b.bits.b4;
  c.bits.b4 = b.bits.b3;
  c.bits.b5 = b.bits.b2;
  c.bits.b6 = b.bits.b1;
  c.bits.b7 = b.bits.b0;

  return (c.byte);
}
```

Si recuerda como trabaja el operador condicional, sabrá que la operación

```
(c & (1 << i)) ? 1 : 0
```

da como resultado, en este caso, *1* si la expresión *(c & (1 << i))* es cierta (valor distinto de cero) y *0* si la expresión es falsa (valor cero). La expresión *1 << i* desplaza el *1* (0...01), *i* veces a la izquierda. El objetivo es visualizar *c* en binario. La ejecución del bucle **for** que la contiene ocurre de la forma siguiente:

c	i	1 << i	c & (1 << i)	(c&(1<<i)) ? 1 : 0
01000001	7	10000000	00000000	0
01000001	6	01000000	01000000	1
01000001	5	00100000	00000000	0

c	i	1 << i	c & (1 << i)	(c&(1<<i)) ? 1 : 0
01000001	4	00010000	00000000	0
01000001	3	00001000	00000000	0
01000001	2	00000100	00000000	0
01000001	1	00000010	00000000	0
01000001	0	00000001	00000001	1

EJERCICIOS RESUELTOS

1. Realizar un programa que lea una lista de valores introducida por el teclado. A continuación, y sobre la lista, buscar los valores máximo y mínimo, y escribirlos.

   La solución de este problema puede ser de la siguiente forma:

   - Definimos el array que va a contener la lista de valores y el resto de las variables necesarias en el programa.

```
float a[DIM_MAX]; /* lista de valores */
float max, min;   /* valor máximo y valor mínimo */
int numval = 0;   /* numval = número de valores leídos */
int i = 0;        /* índice */
```

   - A continuación leemos los valores que forman la lista.

```
printf("a[%d] = ", numval);
while (numval < DIM_MAX && scanf("%f", &a[numval]) != EOF)
{
  numval++;
  printf("a[%d]= ", numval);
}
```

   - Una vez leída la lista de valores, calculamos el máximo y el mínimo. Para ello suponemos inicialmente que el primer valor es el máximo y el mínimo (como si todos los valores fueran iguales). Después comparamos cada uno de estos valores con los restantes de la lista. El valor de la lista comparado pasará a ser el nuevo mayor si es más grande que el mayor actual y pasará a ser el nuevo menor si es más pequeño que el menor actual.

```
max = min = a[0];
for (i = 0; i < numval; i++)
{
  if (a[i] > max)
    max = a[i];
  if (a[i] < min)
    min = a[i];
}
```

- Finalmente, escribimos el resultado.

```
printf("\n\nValor máximo: %g, valor mínimo: %g\n", max, min);
```

El programa completo se muestra a continuación.

```
/** Encontrar el máximo y el mínimo de un conjunto de valores **/
/* maxmin.c
 */
#include <stdio.h>
#define DIM_MAX 100 /* máximo número de valores */

void main()
{
  float a[DIM_MAX]; /* lista de valores */
  float max, min;   /* valor máximo y valor mínimo */
  int numval = 0;   /* numval = número de valores leídos */
  int i = 0;        /* índice */

  /* Entrada de datos */
  printf("Introducir datos. Para finalizar introducir la marca EOF\n\n");
  printf("a[%d] = ", numval);
  while (numval < DIM_MAX && scanf("%f", &a[numval]) != EOF)
  {
    numval++;
    printf("a[%d]= ", numval);
  }

  /* Encontrar los valores máximo y mínimo */
  if (numval > 0)
  {
    max = min = a[0];
    for (i = 0; i < numval; i++)
    {
      if (a[i] > max)
        max = a[i];
      if (a[i] < min)
        min = a[i];
    }

    /* Escribir resultados */
    printf("\n\nValor máximo: %g, valor mínimo: %g\n", max, min);
  }
  else
    printf("\n\nNo hay datos.\n");
}
```

2. Escribir un programa que dé como resultado la frecuencia con la que aparece cada una de las parejas de letras adyacentes, de un texto introducido por el teclado. No se hará diferencia entre mayúsculas y minúsculas. El resultado se presentará en forma de tabla, de la manera siguiente:

```
   a  b  c  d  e  f        ...        z
a  0  4  0  2  1  0        ...        1
b  8  0  0  0  3  1        ...        0
c              .
d              .
e              .
f              .

.
.
.

z
```

Por ejemplo, la tabla anterior dice que la pareja de letra *ab* ha aparecido 4 veces. La tabla resultante contempla todas las parejas posibles de letras, desde la *aa* hasta la *zz*.

Las parejas de letras adyacentes de *hola que tal* son: *ho*, *ol*, *la*, *a* blanco no se contabiliza por estar el carácter espacio en blanco fuera del rango 'a' - 'z', blanco *q* no se contabiliza por la misma razón, *qu*, etc.

Para realizar este problema, en función de lo expuesto necesitamos un array de enteros de dos dimensiones. Cada elemento actuará como contador de la pareja de letras correspondiente. Por lo tanto, todos los elementos del array deben valer inicialmente cero.

```
#define DIM ('z' - 'a' + 1) /* filas/columnas de la tabla */
int tabla[DIM][DIM]; /* tabla de contingencias */
```

Para que la solución sea fácil, aplicaremos el concepto de arrays asociativos visto anteriormente en este mismo capítulo; es decir, la pareja de letras a contabilizar serán los índices del elemento del array que actúa como contador de dicha pareja. Observe la tabla anterior y vea que el contador de la pareja *aa* es el elemento (0,0) del supuesto array. Esto supone restar una constante de valor *'a'* a los valores de los índices (*carant*, *car*) utilizados para acceder a un elemento. La variable *carant* contendrá el primer carácter de la pareja y *car* el otro carácter.

```
if ((carant>='a' && carant<='z') && (car>='a' && car<='z'))
  tabla[carant - 'a'][car - 'a']++;
```

El problema completo se muestra a continuación.

```
/**** Tabla de frecuencias de letras adyacentes en un texto ****/
/* parejas.c
 */
#include <stdio.h>
#include <ctype.h>
#define DIM ('z' - 'a' + 1) /* filas/columnas de la tabla */
```

```
void main()
{
  static int tabla[DIM][DIM]; /* tabla de contingencias */
  char f, c;                  /* índices */
  char car;                   /* carácter actual */
  char carant = ' ';          /* carácter anterior */

  printf("Introducir texto. Para finalizar introducir la marca EOF\n\n");
  while ((car = getchar()) != EOF)
  {
    car = tolower(car); /* convertir a minúsculas si procede */
    if ((carant>='a' && carant<='z') && (car>='a' && car<='z'))
      tabla[carant - 'a'][car - 'a']++;
    carant = car;
  }

  /* Escribir tabla de frecuencias */
  printf(" ");
  for (c = 'a'; c <= 'z'; c++)
    printf("  %c", c);
  putchar('\n');
  for (f = 'a'; f <= 'z'; f++)
  {
    putchar(f);
    for (c = 'a'; c <= 'z'; c++)
      printf("%3d", tabla[f - 'a'][c - 'a']);
    putchar('\n');
  }
}
```

El código que escribe la tabla de frecuencias utiliza un bucle **for** para visualizar la primera línea, la que actúa como cabecera (a b c ...) para especificar el último carácter de la pareja, y dos bucles **for** anidados para escribir los valores del array *tabla* por filas. Observe que antes de cada fila se escribe el carácter correspondiente al primer carácter de la pareja (a b c ...).

3. Visualizar la representación interna de un valor *float*. Por ejemplo, para un valor de −10.5 el resultado que se obtendría al ejecutar el programa sería:

```
real = 1100 0001 0010 1000 0000 0000 0000 0000
```

Este resultado está representado en coma flotante, bajo el formato estándar IEEE; el cual emplea, mantisa fraccionaria normalizada en signo y magnitud, y sin almacenar el bit implícito que es igual 1. El exponente está representado en exceso 127. Por tanto, el valor viene dado por $(-1)^S \times 1.M \times 2^{E-127}$ para 0<E<255.

```
Signo:     1 ............................... S = 1
Exponente: 100 0001 0 ...................... E = 3
Mantisa:   010 1000 0000 0000 0000 0000 ..... M = 0.3125
```

Aplicado a nuestro ejemplo, obtenemos:

$(-1)^1 \times 1.3125 \times 2^3 = -10.5$

Más que entender la fórmula de conversión, el propósito de este problema es visualizar el valor binario de un número **float** almacenado en una variable en memoria. recuerde que un **float** ocupa cuatro bytes.

Para realizar este problema de una manera sencilla utilizaremos una unión *ufl* que permita disponer del valor real como un dato de tipo **float** o bit a bit, lo que conseguiremos con una estructura cuyos miembros sean campos be bits de un bit.

```
union ufl  /* unión con dos miembros: un real y una estructura */
{
  float x; /* valor real de tipo float */
  struct   /* valor bit a bit */
  {
    unsigned b0 : 1;
    ...
    unsigned b31: 1;
  } s;
} real;
```

El programa completo se muestra a continuación.

```
/********** Manipulación de un valor float bit a bit **********/
/* float.c
 */
#include <stdio.h>

void main()
{
  struct sfl
  {
    unsigned b0 : 1; unsigned b1 : 1;
    unsigned b2 : 1; unsigned b3 : 1;
    unsigned b4 : 1; unsigned b5 : 1;
    unsigned b6 : 1; unsigned b7 : 1;
    unsigned b8 : 1; unsigned b9 : 1;
    unsigned b10: 1; unsigned b11: 1;
    unsigned b12: 1; unsigned b13: 1;
    unsigned b14: 1; unsigned b15: 1;
    unsigned b16: 1; unsigned b17: 1;
    unsigned b18: 1; unsigned b19: 1;
    unsigned b20: 1; unsigned b21: 1;
    unsigned b22: 1; unsigned b23: 1;
    unsigned b24: 1; unsigned b25: 1;
    unsigned b26: 1; unsigned b27: 1;
    unsigned b28: 1; unsigned b29: 1;
    unsigned b30: 1; unsigned b31: 1;
  };
```

```
  union ufl
  {
    float x;
    struct sfl s;
  };

  union ufl real;

  real.x = -10.5;
  printf("real = ");
  printf("%d", real.s.b31); printf("%d",  real.s.b30);
  printf("%d", real.s.b29); printf("%d ", real.s.b28);
  printf("%d", real.s.b27); printf("%d",  real.s.b26);
  printf("%d", real.s.b25); printf("%d ", real.s.b24);
  printf("%d", real.s.b23); printf("%d",  real.s.b22);
  printf("%d", real.s.b21); printf("%d ", real.s.b20);
  printf("%d", real.s.b19); printf("%d",  real.s.b18);
  printf("%d", real.s.b17); printf("%d ", real.s.b16);
  printf("%d", real.s.b15); printf("%d",  real.s.b14);
  printf("%d", real.s.b13); printf("%d ", real.s.b12);
  printf("%d", real.s.b11); printf("%d",  real.s.b10);
  printf("%d", real.s.b9 ); printf("%d ", real.s.b8 );
  printf("%d", real.s.b7 ); printf("%d",  real.s.b6 );
  printf("%d", real.s.b5 ); printf("%d ", real.s.b4 );
  printf("%d", real.s.b3 ); printf("%d",  real.s.b2 );
  printf("%d", real.s.b1 ); printf("%d\n",real.s.b0 );
}
```

4. Queremos escribir un programa para operar con matrices de números complejos. Las estructuras de datos que vamos a manejar están basados en las siguientes definiciones:

```
#define MAX 10 /* máximo número de filas y columnas */
typedef struct
{
  float r; /* parte real de un número complejo */
  float i; /* parte imaginaria de un número complejo */
} tcomplejo;

typedef struct
{
  int filas;     /* filas que actualmente tiene la matriz */
  int columnas; /* columnas que actualmente tiene la matriz */
  tcomplejo c[MAX][MAX]; /* matriz de complejos */
} tmatriz;
```

Se pide:

a) Escribir una función para leer una matriz. El prototipo de esta función será de la forma:

```
tmatriz LeerMatriz();
```

Esta función solicitará el número de *filas* y de *columnas* de la matriz y leerá todos sus elementos.

b) Escribir una función que visualice una matriz determinada. El prototipo de esta función será así:

```
void VisualizarMatriz(tmatriz m);
```

c) Escribir una función para multiplicar dos matrices. El prototipo de esta función será:

```
tmatriz Multiplicar(tmatriz a, tmatriz b);
```

Para invocar a la función multiplicar proceda como se indica a continuación:

```
tmatriz a, b, c;
// ...
c = Multiplicar(a, b);
```

Tenga presente que la matriz es de números complejos. Para hacer fácil el desarrollo de esta función escriba previamente dos funciones: una que sume dos complejos y devuelva como resultado la suma y otra que multiplique dos complejos y devuelva como resultado el producto. Los prototipos de estas funciones serán así:

```
tcomplejo SumCompl(tcomplejo a, tcomplejo b);
tcomplejo MulCompl(tcomplejo a, tcomplejo b);
```

d) Utilizando las funciones anteriores, escribir un programa que lea dos matrices y visualice el producto de ambas.

El programa completo se muestra a continuación.

```
/*************** Matrices de números complejos **************/
 /* complejo.c
  */
#include <stdio.h>
#include <stdlib.h>
#define MAX 10 /* máximo número de filas y columnas */

typedef struct
{
  float r; /* parte real de un número complejo */
  float i; /* parte imaginaria de un número complejo */
} tcomplejo;
```

```
typedef struct
{
  int filas;     /* filas que actualmente tiene la matriz */
  int columnas; /* columnas que actualmente tiene la matriz */
  tcomplejo c[MAX][MAX]; /* matriz de complejos */
} tmatriz;

tmatriz LeerMatriz()
{
  tmatriz m;
  int f = 0, c = 0;
  int r = 0;
  do
  {
    printf("\nNúmero de filas: ");
    r = scanf ("%d", &m.filas);
    fflush(stdin);
  }
  while (r != 1 || m.filas > MAX);
  do
  {
    printf("Número de columnas: ");
    r = scanf("%d", &m.columnas);
    fflush(stdin);
  }
  while (r != 1 || m.columnas > MAX);
  /* Leer los datos para la matriz */
  printf("\nIntroducir datos de la forma: x yj\n");
  printf("x e y son valores reales positivos o negativos\n\n");
  for (f = 0; f < m.filas; f++)
  {
    for (c = 0; c < m.columnas; c++)
    {
      do
      {
        printf("elemento [%d][%d] = ", f, c);
        r = scanf("%f %f", &m.c[f][c].r,&m.c[f][c].i);
        fflush(stdin);
      }
      while (r != 2);
    }
  }
  return m;
}

void VisualizarMatriz(tmatriz m)
{
  int f, c;
  printf("\n");
  for (f = 0; f < m.filas; f++)
  {
    for (c = 0; c < m.columnas; c++)
      printf ("%8.2f%+8.2fj", m.c[f][c].r, m.c[f][c].i);
    printf ("\n");
  }
}
```

```
tcomplejo SumCompl(tcomplejo a, tcomplejo b)
{
  tcomplejo c;

  c.r = a.r + b.r;
  c.i = a.i + b.i;
  return c;
}

tcomplejo MulCompl(tcomplejo a, tcomplejo b)
{
  tcomplejo c;

  c.r = a.r * b.r - a.i * b.i;
  c.i = a.r * b.i + a.i * b.r;
  return c;
}

tmatriz Multiplicar(tmatriz a, tmatriz b)
{
  tmatriz m;
  int f, c, k;

  if (a.columnas != b.filas)
  {
    printf ("No se pueden multiplicar las matrices.\n");
    exit(-1);
  }

  m.filas = a.filas;
  m.columnas = b.columnas;

  /* Multiplicar las matrices */
  for (f = 0; f < m.filas; f++)
  {
    for (c = 0; c < m.columnas; c++)
    {
      m.c[f][c].r = 0;
      m.c[f][c].i = 0;
      for (k = 0; k < a.columnas; k++)
        m.c[f][c] = SumCompl(m.c[f][c], MulCompl(a.c[f][k], b.c[k][c]));
    }
  }
  return m;
}

void main(void)
{
  static tmatriz a, b, c;

  a = LeerMatriz();
  b = LeerMatriz();
  c = Multiplicar(a, b);
  VisualizarMatriz(c);
}
```

Una diferencia que observará en la estructura de este programa con respecto a los que hemos escrito hasta ahora, es que las definiciones de las funciones se han escrito antes de la función **main**. En C, el orden en el que se escriban las funciones no es trascendente. Además, recuerde que los prototipos de las funciones sólo son necesarios cuando las llamadas a las mismas se hacen antes de su definición. Por eso, observará que en este programa no hay funciones prototipo.

5. Queremos escribir una función para ordenar alfabéticamente un array de cadenas de caracteres.

Para ello, primero diríjase al capítulo "Algoritmos" y estudie, si aún no lo conoce, el algoritmo de ordenación basado en el método de la burbuja. Segundo, utilice el código del programa *cadenas.c* que hemos realizado anteriormente para leer y visualizar un array de cadenas de caracteres. Tercero, utilizando el método de la burbuja, escriba una función *ordenar* que se ajuste al prototipo siguiente:

```
void ordenar(char cad[MAX][LONG], int nc);
```

El argumento *cad* es el array de cadenas de caracteres que queremos ordenar alfabéticamente y *nc* es el número total de cadenas a ordenar.

La función *ordenar* será invocada desde la función **main** una vez leído el array de cadenas de caracteres.

El programa completo se muestra a continuación.

```
/****************** Leer una lista de nombres ******************/
/* ordcads.c
 */
#include <stdio.h>
#include <stdlib.h>
#define MAX 10   /* número máximo de filas */
#define LONG 60  /* número máximo de caracteres por fila */
void ordenar(char cad[][LONG], int nc);

void main(void)
{
  char lista[MAX][LONG];
  char *fin = NULL; /* valor devuelto por gets */
  int i = 0, n = 0; /* índices */
  int resp = 0;     /* respuesta sí o no (s|S|n|N) */

  puts("Escriba los nombres que desea introducir.");
  puts("Para finalizar introduzca la marca de fin de fichero.\n");
  do
  {
    printf("Nombre y apellidos: ");
```

```
    fin = gets(lista[i++]);
    /* Cuando gets lea la marca eof, fin es igual a NULL */
  }
  while (fin != NULL && i < MAX);
  clearerr(stdin); /* desactivar el indicador de fin de fichero */

  if (i < MAX) i--; /* eliminar la última entrada: eof */

  ordenar(lista, i);

  do
  {
    printf("¿Desea visualizar el contenido del array? (s/n) ");
    resp = getchar();
    fflush(stdin);
  }
  while (tolower(resp) != 's' && tolower(resp) != 'n');

  if ( tolower(resp) == 's' )
  {
    /* Escribir el contenido del array de cadenas */
    printf("\n");
    for (n = 0; n < i; n++)
      printf("%s\n", lista[n]);
  }
}

/*****************************************************************
                        Función ordenar
*****************************************************************/

/* Ordenar cadenas de caracteres por orden alfabético */
void ordenar(char cad[][LONG], int nc)
{
  char aux[LONG];
  int i, s = 1;

  while ((s == 1) && (--nc > 0))
  {
    s = 0;  /* no permutación */
    for (i = 1; i <= nc; i++)
      if (strcmp(cad[i-1], cad[i]) > 0)
      {
        strcpy(aux, cad[i-1]);
        strcpy(cad[i-1], cad[i]);
        strcpy(cad[i], aux);
        s = 1;  /* permutación */
      }
  }
}
```

EJERCICIOS PROPUESTOS

1. Se desea realizar un histograma con los pesos de los alumnos de un determinado curso.

```
Peso        Número de alumnos
 21         **
 22         ****
 23         ***************
 24         ******
 ...        ...
```

El número de asteriscos se corresponde con el número de alumnos del peso especificado.

Realizar un programa que lea los pesos de los alumnos e imprima el histograma correspondiente. Suponer que los pesos están comprendidos entre los valores 10 y 100 Kgs. En el histograma sólo aparecerán los pesos que se corresponden con 1 o más alumnos.

2. Realizar un programa que lea una cadena de *n* caracteres e imprima dicha cadena, cada vez que realicemos una rotación de un carácter a la derecha sobre la misma. El proceso finalizará cuando se haya obtenido nuevamente la cadena de caracteres original. Por ejemplo,

```
HOLA   AHOL   LAHO   OLAH   HOLA
```

3. Realizar un programa que lea una cadena de caracteres y la almacene en un array. A continuación, utilizando una función, convertir los caracteres escritos en mayúsculas a minúsculas e imprimir el resultado.

4. La mediana de una lista de *n* números se define como el valor que es menor o igual que los valores correspondientes a la mitad de los números, y mayor o igual que los valores correspondientes a la otra mitad. Por ejemplo, la mediana de:

   16 12 99 95 18 87 10

   es 18, porque este valor es menor que 99, 95 y 87 (mitad de los números) y mayor que 16, 12 y 10 (otra mitad).

   Realizar un programa que lea un número impar de valores y de como resultado la mediana. La entrada de valores finalizará cuando se introduzca la marca de fin de fichero.

5. Escribir un programa, que utilice una función para leer una línea de la entrada y que dé como resultado, la línea leída y su longitud o número de caracteres.

6. Analice el programa que se muestra a continuación e indique el significado que tiene el resultado que se obtiene.

```
#include <stdio.h>

void Visualizar( unsigned char c );
unsigned char fnxxx( unsigned char c );

void main()
{
  unsigned char c;

  printf("Introduce un carácter: ");
  c = getchar();
  Visualizar(c);

  printf("\nCarácter resultante:\n");
  c = fnxxx(c);
  Visualizar(c);
}

void Visualizar( unsigned char c )
{
  int i = 0;

  for (i = 7; i>=0; i--)
    printf("%d", (c & (1 << i)) ? 1 : 0);
  printf("\n");
}

unsigned char fnxxx( unsigned char c )
{
  return ((((c)&0x01) << 7) | (((c)&0x02) << 5) |
          (((c)&0x04) << 3) | (((c)&0x08) << 1) |
          (((c)&0x10) >> 1) | (((c)&0x20) >> 3) |
          (((c)&0x40) >> 5) | (((c)&0x80) >> 7));
}
```

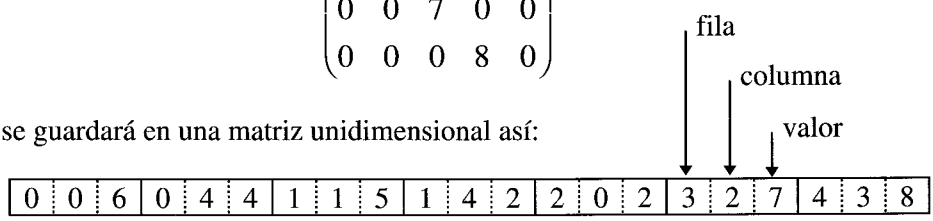
7. En el apartado de ejercicios resueltos se ha presentado un programa para visualizar la representación interna de un valor *float*. ¿Se podría resolver este problema utilizando una función como el siguiente? Razone la respuesta.

```
void presentar( float c )
{
  int i = 0;

  for (i = 31; i>=0; i--)
    printf("%d", (c & (1 << i)) ? 1 : 0);
  printf("\n");
}
```

8. Para almacenar una matriz bidimensional que generalmente tiene muchos elementos nulos (matriz *sparse*) se puede utilizar una matriz unidimensional en la que sólo se guardarán los elementos no nulos precedidos por sus índices, fila y columna, lo que redunda en un aprovechamiento de espacio. Por ejemplo, la matriz

$$\begin{pmatrix} 6 & 0 & 0 & 0 & 4 \\ 0 & 5 & 0 & 0 & 2 \\ 2 & 0 & 0 & 0 & 0 \\ 0 & 0 & 7 & 0 & 0 \\ 0 & 0 & 0 & 8 & 0 \end{pmatrix}$$

se guardará en una matriz unidimensional así:



Para trabajar con esta matriz utilizaremos la siguiente declaración:

```
typedef int tMatrizU[300]
```

Se pide:

a) Escribir una función que lea una matriz bidimensional por filas y la almacene en una matriz *m* de tipo *tMatrizU*. El prototipo de esta función será:

```
int CrearMatrizUni( tMatrizU m, int fi, int co );
```

Los parámetros *fi* y *co* se corresponden con el número de filas y de columnas de la supuesta matriz bidimensional.

b) Escribir una función llamada *Visualizar* que permita representar en pantalla la matriz bidimensional por filas y columnas. El prototipo de esta función será:

```
int Visualizar(int f, int c, tMatrizU m, int n );
```

Los parámetros *f* y *c* se corresponden con la fila y la columna del elemento que se visualiza. El valor del elemento que se visualiza se obtiene, lógicamente de la matriz unidimensional creada en el apartado a, así: buscamos por los índices *f* y *c*; si se encuentran, la función *Visualizar* devuelve el valor almacenado justamente a continuación; si no se encuentran, entonces devuelve un cero. El parámetro *n* indica el número de elementos no nulos de la matriz bidimensional.

Escribir un programa que utilizando la función *CrearMatrizUni* cree una matriz unidimensional a partir de una supuesta matriz *sparse* bidimensional y que a continuación, utilizando la función *Visualizar* presente en pantalla la matriz bidimensional.

CAPÍTULO 7

PUNTEROS

Un puntero es una variable que contiene la *dirección* de memoria de un dato o de otra variable que contiene al dato. Quiere esto decir, que el puntero apunta al espacio físico donde está el dato o la variable. Un puntero puede apuntar a un objeto de cualquier tipo, como por ejemplo, a una estructura o a una función. Los punteros se pueden utilizar para referenciar y manipular estructuras de datos, para referenciar bloques de memoria asignados dinámicamente y para proveer el paso de argumentos por referencia en las llamadas a funciones.

Muchas funciones de la biblioteca de C tienen parámetros que son punteros y devuelven un puntero. Como ejemplo, recuerde la sintaxis de la función **scanf** o de la función **strcpy**.

Cuando se trabaja con punteros son frecuentes los errores debidos a la creación de punteros que apuntan a alguna parte inesperada, produciéndose una violación de memoria. Por lo tanto, debe ponerse la máxima atención para que esto no ocurra, inicializando adecuadamente cada uno de los punteros que utilicemos.

CREACIÓN DE PUNTEROS

Un puntero se declara anteponiendo al identificador que nombra al puntero, el modificador *, el cual significa "*puntero a*". Un puntero inicializado correctamente siempre apunta a un objeto de un tipo particular. Un puntero no inicializado no se sabe a donde apunta. La sintaxis para declarar un puntero es:

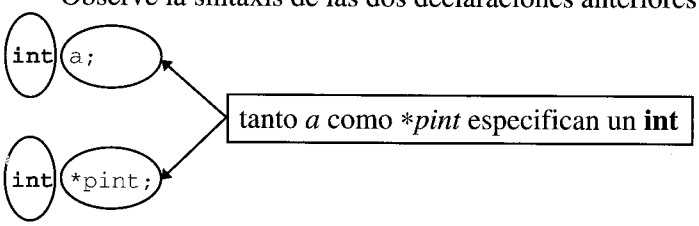
```
tipo *var-puntero;
```

donde *tipo* especifica el tipo del objeto apuntado, puede ser cualquier tipo incluyendo tipos definidos por el usuario, y *var-puntero* es el identificador de la variable puntero. Por ejemplo,

```
int a;          /* a es una variable entera */
int *pint;      /* pint es un puntero a un entero */
pint = &a;      /* pint igual a la dirección de a; entonces,
                   pint apunta a la variable a    */
```

La declaración de *a* ya nos es familiar. La declaración del puntero *pint*, aunque también la hemos visto en más de una ocasión, es nueva en lo que se refiere a su utilización como una variable más.

Observe la sintaxis de las dos declaraciones anteriores



Observamos que **pint* es un nemotécnico que hace referencia a un objeto de tipo **int**, por lo tanto puede aparecer en los mismos lugares donde puede aparecer un entero. Es decir, si *pint* apunta al entero *a*, entonces **pint* puede aparecer en cualquier lugar donde pueda hacerlo *a*.

En conclusión **pint* es un entero que está localizado en la dirección de memoria especificada por *pint*.

Este razonamiento es aplicable a cualquier declaración por complicada que sea. Otro ejemplo,

```
double sqrt(double), *p, d, x;
p = &x;
// ...
d = sqrt(x) + *p;
printf("%g\n", *p); /* imprime el valor actual de x */
```

Este ejemplo pone de manifiesto que *sqrt(x)* y **p* representan valores de tipo **double** en la expresión donde aparecen; es decir, son equivalentes a variables de tipo **double**.

El espacio de memoria requerido para un puntero, es el número de bytes necesarios para especificar una dirección máquina. Son valores típicos 2 o 4 bytes.

Operadores

Los ejemplos que hemos visto hasta ahora, ponen de manifiesto que en las operaciones con punteros intervienen frecuentemente el operador *dirección de* (**&**) y el operador de *indirección* (*).

El operador unitario **&**, devuelve como resultado la dirección de su operando y el operador unitario *, toma su operando como una dirección y nos da como resultado su contenido (para más detalles, vea el Capítulo 2). Por ejemplo,

```
#include <stdio.h>
void main()
{
  /* La siguiente línea declara las variables enteras a y b
   * y los punteros p y q a enteros
   */
  int a = 10, b, *p, *q;
  q = &a;  /* asigna la dirección de a, a la variable q */
           /* q apunta a la variable entera a */
  b = *q;  /* asigna a b el valor de la variable a */
  *p = 20; /* error: asignación no válida */
           /* ¿a dónde apunta p? */
  printf("En la dirección %.4X está el dato %d\n", q, b);
  printf("En la dirección %.4X está el dato %d\n", p, *p);
}
```

En teoría, si ejecutamos este programa el resultado sería análogo al siguiente:

```
En la dirección 0D96 está el dato 10
En la dirección 54E6 está el dato 20
```

Pero, en la práctica estamos cometiendo un error por utilizar un puntero *p* sin saber a dónde apunta. Sabemos que *q* apunta a la variable *a*; dicho de otra forma, *q* contiene una dirección válida, pero no podemos decir lo mismo de *p* ya que no ha sido inicializado y por lo tanto su valor es desconocido para nosotros; posiblemente sea una dirección ocupada por el sistema y entonces lo que estamos haciendo es sobreescribir el contenido de esa dirección con el valor 20, lo que seguramente nos ocasionará problemas. En el caso de UNIX, el sistema realiza un control exhaustivo de la memoria asignada con lo que el uso de un puntero no inicializado correctamente produce un error, "violación de segmento" o "fallo de segmento", durante la ejecución.

Importancia del tipo del objeto al que se apunta

¿Cómo sabe C cuántos bytes tiene que asignar a una variable desde una dirección? La respuesta es que C toma como referencia el tipo del objeto apuntado por el puntero y asigna el número de bytes correspondiente a ese tipo. Por ejemplo,

```
/* Este programa no opera correctamente */
#include <stdio.h>
void main()
{
  float a = 10.33F, b = 0; /* a y b son variables de tipo real */

  int *p;  /* p es un puntero a un entero */

  p = &a;
  b = *p;
  printf("b = %g\n", b);
}
```

Al compilar este programa, se nos presentará el mensaje,

```
aviso: indirección a diferentes tipos.
```

debido a la sentencia *p = &a*, ya que *p* apunta a un **int** y *a* es una variable de tipo **float**. Cuando se ejecuta el programa, el resultado es inesperado porque al ejecutarse la sentencia *b = *p*, C toma como referencia el tipo **int** (*p* apunta a un **int**) y asigna el número de bytes correspondiente a este tipo (dos o cuatro bytes), cuando en realidad tendría que asignar cuatro bytes interpretados como un valor real. Aún, en el supuesto de que asignaran los cuatro bytes, estaríamos cometiendo un error y es que el contenido de los cuatro bytes se tomaría como un valor entero, cuando lo que hay almacenado es un valor real.

OPERACIONES CON PUNTEROS

Si *p* es un puntero, *p++* incrementa *p* para que apunte al siguiente elemento de los que apunta *p*; es decir, C sabe que tiene que avanzar un número de bytes igual al tamaño de un objeto de los que apunta *p*. La aritmética de direcciones es una de las principales virtudes de C.

Operación de asignación

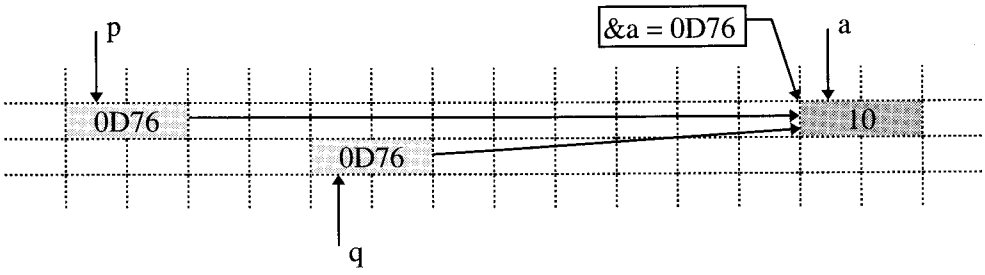
A un puntero se le puede asignar otro puntero. Por ejemplo:

```
#include <stdio.h>
void main()
{
  int a = 10, *p, *q;

  p = &a;
  q = p; /* la dirección que contiene p se asigna a q */
  printf("En la dirección %.4X está el valor %d", q, *q);
}
```

El resultado de este programa será análogo al siguiente:

```
En la dirección 0D76 está el valor 10
```

Gráficamente puede observar que después de ejecutarse la asignación, *p* y *q* apuntan a la misma localización de memoria, a la variable *a*. Por lo tanto, *a*, **p* y **q* son el mismo dato; es decir, 10.

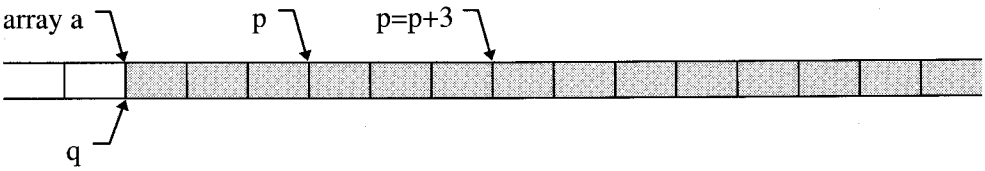


Operaciones aritméticas

A un puntero se le puede sumar o restar un entero. Por ejemplo,

```
int a[100];
int *p, *q; /* declara p y q como un puntero a un entero */
p = &a[3];  /* p apunta a a[3] */
q = &a[0];  /* q apunta a a[0] */
p++;        /* hace que p apunte al siguiente entero; a a[4] */
p--;        /* hace que p apunte al entero anterior; a a[3] */
p = p + 3;  /* hace que p avance tres enteros; apunta a a[6] */
p = p - 3;  /* hace que p retroceda tres enteros; apunta a a[3] */
```

Si *p* y *q* son variables de tipo puntero y apuntan a elementos del mismo array, la operación *p* – *q* es válida; en el ejemplo, el resultado es 3. No se permite sumar, multiplicar, dividir o rotar punteros y tampoco se permite sumarles un real.



Comparación de punteros

Es posible comparar dos punteros en una expresión de relación. Esta operación tiene sentido si ambos punteros apuntan a elementos del mismo array. Por ejemplo:

```
int n, *p, *q, a[100];
// ...
p = a[99];
q = a[0];
// ...
if (q + n <= p)
  q += n;

if (q != NULL && q <= p) /* NULL es una constante que identifica
  q++;                       a un puntero nulo */
```

La primera sentencia **if** indica que el puntero *q* avanzará *n* elementos si se cumple que la dirección *q+n* es menor o igual que *p*. La segunda sentencia **if** indica que *q* pasará a apuntar al siguiente elemento del array si la dirección por él especificada no es nula y es menor o igual que la especificada por *p*.

Ejemplos con punteros

Los operadores unitarios * y & tienen prioridad mayor que los operadores aritméticos + y – e igual prioridad que ++ y ––.

Definimos el array *a* y dos punteros a datos de tipo entero *pa* y *pb*.

```
int a[100], *pa, *pb;
// ...
a[50] = 10;
pa = &a[50];

b = *pa + 1;    /* el resultado de la suma del entero *pa más 1
                   se asigna a b; es decir, b = a[50] + 1 */

b = *(pa + 1); /* el siguiente entero al entero apuntado por pa,
                  es asignado a b (b = a[51]) */

pb = &a[10];    /* a pb se le asigna la dirección de a[10] */

*pb = 0;        /* el elemento a[10] es puesto a 0 */

*pb += 2;       /* a[10] se incrementa en dos unidades */

(*pb)--;        /* a[10] se decrementa en una unidad */

a[0] = *pb--;   /* a a[0] se le asigna el valor de a[10] y pb
                   pasa a apuntar al entero anterior (a a[9]) */
```

Punteros genéricos

Un puntero a cualquier objeto no constante se puede convertir a tipo **void** *. Por eso, un puntero de tipo **void** * recibe el nombre de puntero genérico. Por ejemplo,

```
void main()
{
  void *p = NULL; /* puntero genérico */
  int a = 10, *q = &a; /* se declara q y se le asigna la dirección de a */
  p = q; /* implícitamente se convierte (int *) a (void *) */
  // ...
}
```

ANSI C permite una conversión implícita de un puntero a **void** a un puntero a otro tipo de objeto; en C++ un puntero a **void** no puede ser asignado a un puntero a otro objeto sin realizar una conversión explícita. Por ejemplo, lo siguiente está permitido en ANSI C, pero no en C++:

```
#include <stdio.h>

void fx(int *pi, void *pg)
{
  pi = pg; /* permitido en ANSI C pero no en C++ */
  printf("%d\n", *pi);
}

void main()
{
  void *p = NULL; /* puntero genérico */
  int a = 10, *q = &a; /* se declara q y se le asigna la dirección de a */
  p = q;
  fx(NULL, p);
}
```

Para que la función *fx* fuera válida en C++ tendríamos que escribirla así:

```
void fx(int *pi, void *pg)
{
  pi = (int *)pg; /* válido en ANSI C y en C++ */
  printf("%d\n", *pi);
}
```

En este ejemplo se observa que para convertir un puntero a **void** a un puntero a un objeto de tipo **int** se ha hecho una conversión explícita (conversión, *cast*).

Puntero nulo

En general, un puntero se puede inicializar como cualquier otra variable, aunque los únicos valores significativos son **NULL** o la dirección de un objeto previamente definido. **NULL** es una constante definida en el fichero *stdio.h* así:

```
#define NULL ((void *)0)   /* definición de NULL en C */
#define NULL 0             /* definición de NULL en C++ */
```

El lenguaje C garantiza que un puntero que apunte a un objeto válido nunca tendrá un valor cero. El valor cero se utiliza para indicar que ha ocurrido un error; en otras palabras, que una determinada operación no se ha podido realizar. Por ejemplo, recuerde que la función **gets** cuando lee la marca de fin de fichero retorna un puntero nulo, indicando así que no hay más datos para leer.

En general, no tiene sentido asignar enteros a punteros porque quien gestiona la memoria es el sistema operativo y por lo tanto es él, el que sabe en todo momento qué direcciones están libres y cuáles están ocupadas. Por ejemplo,

```
int *px = 10382; /* se inicializa px con la dirección 10382 */
```

La asignación anterior no tiene sentido porque ¿qué sabemos nosotros acerca de la dirección 10382?

Punteros constantes

Una declaración de un puntero precedida por **const**, hace que el objeto apuntado sea una constante, no sucediendo lo mismo con el puntero. Por ejemplo,

```
const char *pc = "abcd"; /* objeto "abcd" constante y pc variable*/
pc[0] = 'z';     /* error; la cadena apuntada por pc es constante */
pc = "efg";     /* correcto; pc pasa a apuntar a una nueva cadena */
```

Si lo que se pretende es declarar un puntero constante, procederemos así:

```
char *const pc = "abcd"; /* objeto "abcd" variable y pc constante*/
pc[0] = 'z';  /* correcto; la cadena apuntada por pc es variable */
pc = "efg";                              /* error; pc es constante */
```

Para hacer que tanto el puntero como el objeto apuntado sean constantes, procederemos como se indica a continuación:

```
const char *const pc = "abcd";     /* objeto y puntero constantes */
pc[0] = 'z';              /* error; el objeto "abcd" es constante */
pc = "efg";                  /* error; el puntero pc es constante */
```

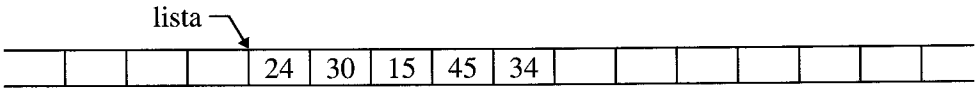
PUNTEROS Y ARRAYS

En C existe una relación entre punteros y arrays tal que cualquier operación que se pueda realizar mediante la indexación de un array, se puede realizar también con punteros.

Para clarificar lo expuesto, analicemos el siguiente programa, realizado primeramente con arrays y a continuación con punteros.

```
/* Escribir los valores de un array. Versión utilizando un array */
/* punt01.c
 */
#include <stdio.h>

void main()
{
  static int lista[] = {24, 30, 15, 45, 34};
  int ind;

  for (ind = 0; ind < 5; ind++)
    printf("%d  ", lista[ind]);
}
```

En este ejemplo se ha utilizado la indexación del array para acceder a sus elementos; expresión *lista[ind]*. Cuando C interpreta esta expresión sabe que a partir de la dirección de comienzo del array (a partir de *lista*) tiene que avanzar *ind* elementos para acceder al contenido del elemento especificado por ese índice.



A continuación se expone la versión con punteros:

```
/* Escribir los valores de un array. Versión con punteros */
/* punt02.c
 */
#include <stdio.h>

void main()
{
  static int lista[] = {24, 30, 15, 45, 34};
  int ind;
  int *plista = &lista[0];

  for (ind = 0; ind < 5; ind++)
    printf("%d  ", *(plista+ind));
}
```

Esta versión es idéntica a la anterior, excepto que la expresión para acceder a los elementos del array es ahora **(plista+ind)*.

La asignación *plista = &lista[0]* hace que *plista* apunte al primer elemento de *lista*; es decir, *plista* contiene la dirección del primer elemento, que coincide con la dirección de comienzo del array; esto es, con *lista*. Por lo tanto, en lugar de la expresión **(plista+ind)*, podríamos utilizar también la expresión **(lista+ind)*. Según lo expuesto, las siguientes expresiones dan lugar a idénticos resultados:

```
lista[ind], *(lista+ind), plista[ind], *(plista+ind)
```

El mismo resultado se obtendría con esta otra versión del programa:

```
/* Escribir los valores de un array. Versión con punteros */
/* punt03.c
 */
#include <stdio.h>

void main()
{
  static int lista[] = {24, 30, 15, 45, 34};
  int ind;
  int *plista = lista;

  for (ind = 0; ind < 5; ind++)
    printf("%d  ", *plista++);
}
```

La asignación *plista = lista* hace que *plista* apunte al comienzo del array; es decir, al elemento primero del array, y la expresión **plista++* equivale a **plista* que es el valor apuntado por *plista* y a *plista++* que hace que *plista* pase a apuntar al siguiente elemento del array. Esto es, el bucle se podría escribir también así:

```
for (ind = 0; ind < 5; ind++)
{
  printf("%d  ", *plista);
  plista++;
}
```

Sin embargo, hay una diferencia entre el nombre de un array y un puntero. El nombre de un array es una constante y un puntero es una variable. Esto quiere decir que el siguiente bucle daría lugar a un error, porque *lista* es constante y no puede cambiar de valor.

```
for (ind = 0; ind < 5; ind++)
  printf("%d  ", *lista++);
```

Punteros a cadenas de caracteres

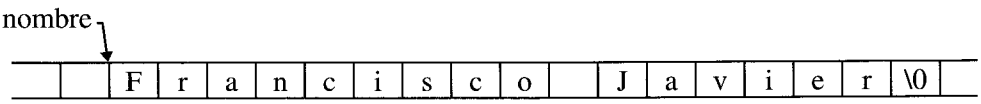
Puesto que una cadena de caracteres es un array de caracteres, es correcto pensar que la teoría expuesta anteriormente, es perfectamente aplicable a cadenas de caracteres. La forma de definir un puntero a una cadena de caracteres es:

```
char *cadena;
```

El identificador del array de caracteres es la dirección de comienzo del array y para que una función manipuladora de una cadena de caracteres pueda saber dón-

de finaliza dicha cadena, el compilador añade al final de la cadena el carácter \0. El siguiente ejemplo define e inicializa la cadena de caracteres *nombre*.

```
char *nombre = "Francisco Javier";
printf("%s", nombre);
```



En el ejemplo anterior *nombre* es un puntero a los caracteres. El compilador C asigna la dirección de comienzo de la cadena "Francisco Javier" al puntero *nombre* y finaliza la cadena con el carácter \0. Por lo tanto, la función **printf** sabe que la cadena de caracteres que tiene que visualizar empieza en la dirección contenida en *nombre* y que a partir de aquí, tiene que ir accediendo a posiciones sucesivas de memoria hasta encontrar el carácter \0.

Es importante tomar nota de que *nombre* no contiene una copia de la cadena asignada, sino un puntero a la cadena almacenada en algún lugar de la memoria. Según esto, en el ejemplo siguiente, *nombre* apunta inicialmente a la cadena de caracteres "Francisco Javier" y a continuación reasigna el puntero para que apunte a una nueva cadena. La cadena anterior se pierde porque el contenido de la variable *nombre* ha sido sobreescrito con una nueva dirección, la de la cadena "Carmen".

```
char *nombre = "Francisco Javier";
printf("%s", nombre);
nombre = "Carmen";
```

El siguiente ejemplo presenta una función que devuelve como resultado el número de caracteres de una cadena, lo que se conoce como longitud de una cadena de caracteres. Recuerde que la biblioteca de C proporciona la función **strlen** que da el mismo resultado.

```
/*** Función "longcad" que devuelve la longitud de una cadena ***/
/* longcad.c
 */
#include <stdio.h>

int longcad(char *);

void main()
{
  static char *cadena = "abcd"; /* el carácter de terminación '\0'
                                   es añadido automáticamente */
  printf("%d\n", longcad(cadena));
}
```

```
int longcad(char *cad)
{
  char *p = cad;                /* p apunta al primer carácter */
  while (*p != '\0')
    p++;                        /* siguiente carácter */
  return (p - cad);
}
```

El valor *p - cad* devuelto por la función *longcad*, da la longitud de la cadena. Esta función realiza las siguientes operaciones:

1. Asigna a *p* la dirección del primer carácter de la cadena, que coincide con la dirección de comienzo de la misma, e inicia la ejecución del bucle **while**.

2. Cuando se ejecuta la condición del bucle **while**, se compara el carácter **p* apuntado por *p* con el carácter nulo. Si **p* es el carácter nulo el bucle finaliza; si no, se incrementa el valor de *p* en una unidad para que apunte al siguiente carácter, y se vuelve a evaluar la condición.

La expresión **p != '\0'* es cierta (valor distinto de cero) cuando **p* es un carácter distinto de nulo y es falsa cuando **p* es el carácter nulo (ASCII cero). Por lo tanto, en lugar de hacer la comparación explícitamente, se podría hacer implícitamente así:

```
int longcad(char *cad)
{
  char *p = cad;
  while (*p)
    p++;
  return (p - cad);
}
```

Ahora la condición está formada por la expresión **p*. Si esta expresión es distinta de cero (carácter distinto de nulo) la condición es cierta. En cambio si es cero (carácter nulo) la condición es falsa. También podríamos escribir:

```
int longcad(char *cad)
{
  char *p = cad;
  while (*p++);
  return (p - cad - 1);
}
```

Ahora el resultado vendrá dado por la expresión *p - cad - 1*, ya que después de examinar sí **p* es cero, se efectúa la operación *p++*. La expresión **++p* no daría el mismo resultado en todos los casos, porque primero se efectúa la operación *++p* y después se examina **p* y esto no sería válido para cadenas de longitud 1.

Esto es así porque los operadores * y ++ tienen la misma prioridad pero la asociatividad es de derecha a izquierda.

Para completar la discusión anterior sobre el orden de evaluación de los operadores * y ++, analicemos la siguiente tabla:

la expresión	es equivalente a	char x[] = "ab", p = x, c; (p = 3390)
*c = *p++;*	*c = *p; p++;*	*c = 'a', p = 3391 y *p = 'b'*
*c = *++p;*	*++p; c = *p;*	*p = 3391, c = 'b' y *p = 'b'*
*c = ++*p;*	**p += 1; c = *p;*	*p = 3390 y c = 'b' y *p = 'b'*
*c = (*p)++;*	*c = *p; (*p)++;*	*c = 'a', p = 3390 y *p = 'b'*

El siguiente ejemplo presenta una función que copia una cadena en otra. Recuerde que la biblioteca de C proporciona la función **strcpy** para realizar la misma operación. Primeramente se presenta una versión utilizando arrays y después otra con punteros.

La versión con arrays del problema planteado es:

```
/*********** Función para copiar una cadena en otra ***********/
/* copicad.c
 */
#include <stdio.h>

void copicad(char [], char []);

void main()
{
  char cadena1[81], cadena2[81];

  printf("Introducir cadena: ");
  gets(cadena1);
  copicad(cadena2, cadena1);  /* copia la cadena1 en la cadena2 */
  printf("\nLa cadena copiada es: %s\n", cadena2);
}

void copicad(char p[], char q[])  /* copia q en p */
{
  int i = 0;

  while ((p[i] = q[i]) != '\0')
    i++;
}
```

La función *copicad* recibe dos parámetros, el array destino y el array fuente o cadena a copiar. Observe la llamada a la función:

```
copicad(cadena2, cadena1);
```

Como ya hemos indicado en otras ocasiones anteriormente, *p* recibe el argumento *cadena2*, que es la dirección de comienzo del array destino, no una copia del array, y lo mismo diremos con respecto a *q*. Esto es lo que en el capítulo 3 denominamos paso de parámetros por referencia.

Así mismo, observe que en la condición del bucle **while** hay implícitas dos operaciones, una asignación y una comparación. Por lo tanto, podríamos escribirlo también así:

```
p[i] = q[i];
while (p[i] != '\0')
{
  i++;
  p[i] = q[i];
}
```

La solución de este mismo problema utilizando una función *copicad* con dos parámetros que sean punteros a caracteres, es así:

```
/*********** Función para copiar una cadena en otra ***********/
/* copicad1.c
 */
#include <stdio.h>
void copicad(char *, char *);

void main()
{
  char cadena1[81], cadena2[81];

  printf("Introducir cadena: ");
  gets(cadena1);
  copicad(cadena2, cadena1);  /* copia la cadena1 en la cadena2 */
  printf("\nLa cadena copiada es: %s\n", cadena2);
}

void copicad(char *p, char *q)  /* copia q en p */
{
  while ((*p = *q) != '\0')
  {
    p++;
    q++;
  }
}
```

Aplicando los mismos razonamientos que hicimos al exponer tanto la función *longcad* del ejemplo anterior, como la función *copicad* de la versión con arrays, una nueva versión de la función *copicad* podría ser así:

```
void copicad(char *p, char *q)  /* copia q en p */
{
  while (*p++ = *q++);
}
```

En la versión con punteros de la función *copicad* resulta más evidente que los parámetros *p* y *q* son dos variables que tienen que recibir dos direcciones, las de los arrays destino y fuente, respectivamente.

ARRAYS DE PUNTEROS

En capítulos anteriores, hemos trabajado con arrays multidimensionales, aunque en la práctica lo más habitual es trabajar con arrays de punteros por las ventajas que esto reporta, como verá más adelante.

Se puede definir un array, para que sus elementos contengan en lugar de un dato, una dirección o puntero. Por ejemplo:

```
int *a[10], v;
a[0] = &v;
printf("%d", *a[0]);
```

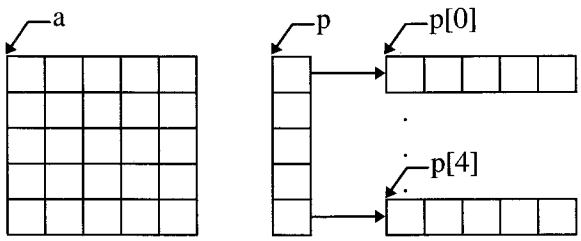
Este ejemplo define un array *a* de 10 elementos, cada uno de los cuales es un puntero a un **int**, y una variable entera *v*. A continuación asigna al elemento *a[0]*, la dirección de *v* y escribe su contenido.

Un array de dos dimensiones y un array de punteros, se pueden utilizar de forma parecida, pero no son lo mismo. Por ejemplo,

```
int a[5][5];
int *p[5];     /* array de punteros */
```

Las declaraciones anteriores dan lugar a que el compilador de C reserve memoria para un array *a* de 25 elementos de tipo entero y para un array *p* de 5 elementos declarados como punteros a objetos de tipo entero.

Supongamos ahora que cada uno de los objetos apuntados por los elementos del array *p* es a su vez un array de 5 elementos de tipo entero; esto es, la ocupación de memoria será la de los 5 elementos de *p* más la de los 25 elementos de los 5 arrays de enteros. Más adelante aprenderemos cómo asignar estos objetos a cada uno de los elementos del array de punteros.



El acceso a los elementos del array *p* puede hacerse utilizando la notación de punteros o utilizando la indexación igual que lo haríamos con *a*. Por ejemplo, para asignar valores a los enteros referenciados por el array *p* y después visualizarlos, podríamos escribir el siguiente código:

```
for (i = 0; i < 5; i++)
  for (j = 0; j < 5; j++)
    scanf("%d", &p[i][j]);

for (i = 0; i < 5; i++)
{
  for (j = 0; j < 5; j++)
    printf("%7d", p[i][j]);
  printf("\n");
}
```

Según lo expuesto ¿que diferencias hay entre *a* y *p*? Las ventajas que tiene el array *p* sobre el *a*, es que el acceso a un elemento se efectúa mediante una indirección a través de un puntero, en lugar de hacerlo mediante una multiplicación y una suma, y que los arrays apuntados pueden ser de longitud variable. El inconveniente es que el espacio ocupado por *p* y los objetos apuntados, es un poco mayor que el ocupado por *a*.

Otro detalle a tener en cuenta desde el punto de vista de direcciones, es que para acceder a un entero a través de *p* hay que pasar por una doble indirección; es decir, observando la figura anterior, vemos que *p*, dirección de comienzo del array, nos conduce a uno de sus elementos, *p[i]*, que a su vez contiene la dirección de un array de enteros. Por lo tanto, si quisiéramos acceder a los enteros referenciados por *p* utilizando una variable puntero, esta variable tendría que ser un puntero a un puntero. Esto es, para realizar una asignación como

```
px = p;
```

la variable *px* tiene que ser declarada como un puntero a un puntero.

Punteros a punteros

Para especificar que una variable es un puntero a un puntero, la sintaxis utilizada es la siguiente:

```
tipo **varpp;
```

donde *tipo* especifica el tipo del objeto apuntado después de una doble indirección, puede ser cualquier tipo incluyendo tipos definidos por el usuario, y *varpp* es el identificador de la variable puntero a puntero. Por ejemplo,

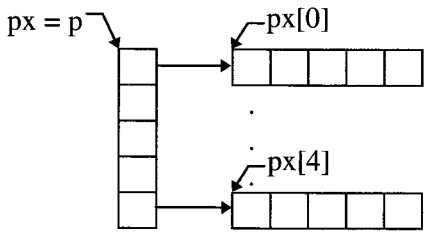
```
int **pp; /* pp es un puntero a un puntero a un objeto int */
```

Por ejemplo, el siguiente código resuelve el ejemplo anterior, pero utilizando ahora una variable *px* declarada como un puntero a un puntero. El acceso a los elementos del array *p* utilizando el puntero *px* puede hacerse igual que antes, utilizando la indexación igual que lo haríamos con un array convencional de dos dimensiones o utilizando la notación de punteros.

La versión utilizando la indexación es:

```
int *p[5];     /* array de punteros */
int **px = p; /* px es un puntero a un puntero */
/* Supongamos que a cada elemento de p se le ha asignado la
 * dirección de un array unidimensional de enteros, cuestión que
 * veremos más adelante al hablar de asignación dinámica de memoria
 */
for (i = 0; i < 5; i++)
  for (j = 0; j < 5; j++)
    scanf("%d", &px[i][j]);

for (i = 0; i < 5; i++)
{
  for (j = 0; j < 5; j++)
    printf("%7d", px[i][j]);
  printf("\n");
}
```

Para reescribir este código y utilizar la notación de punteros en lugar de la notación array, nos planteamos únicamente la cuestión de cómo escribir la expresión *px[i][j]*, utilizando la notación de punteros. Pues bien, como vimos en la figura anterior, un array de punteros es un array de una dimensión, donde cada elemento apunta a su vez a un array de una dimensión.



En el ejemplo anterior, la dirección de comienzo del array es *p* o *px* después de la asignación *px=p*. Este array tiene 5 elementos (filas), *px[0]* a *px[4]*. El contenido de estos elementos se corresponde con la dirección de cada elemento fila (array de 5 elementos) que utilizando *px* como dirección base, podemos especificar también así:

```
*(px+0), *(px+1), *(px+2), *(px+3), *(px+4)
```

Si elegimos una fila, por ejemplo *px[1]*, o en notación puntero **(px+1)*, interpretamos esta expresión como un puntero a un array de 5 elementos; esto quiere decir que **(px+1)*, es la dirección del primer elemento de esa fila, *px[1][0]*, o en notación puntero **(*(px+1)+0)*. Observe que las direcciones *px+1* y **(px+1)* tienen significado diferente. Por ejemplo,

px+1+2	se refiere a la dirección del elemento *px[3]* del array de punteros.
**(px+1)+2*	se refiere a la dirección del elemento *px[1][2]*.
**(*(px+1)+2)*	se refiere al valor del elemento *px[1][2]*.

De acuerdo con lo expuesto la versión con punteros del ejemplo anterior, presenta solamente la siguiente modificación:

```
int *p[5];     /* array de punteros */
int **px = p; /* px es un puntero a un puntero */
/* Supongamos que a cada elemento de p se le ha asignado la
 * dirección de un array unidimensional de enteros, cuestión que
 * veremos más adelante al hablar de asignación dinámica de memoria
 */
for (i = 0; i < 5; i++)
  for (j = 0; j < 5; j++)
    scanf("%d", *(px+i)+j);

for (i = 0; i < 5; i++)
{
  for (j = 0; j < 5; j++)
    printf("%7d", *(*(px+i)+j));
  printf("\n");
}
```

El nombre de un array convencional de dos dimensiones no se puede considerar como un puntero a un puntero. Por ejemplo, la siguiente asignación daría lugar a un error:

```
int a[5][5];
int **pa = a; /* aviso: diferentes niveles de indirección */
```

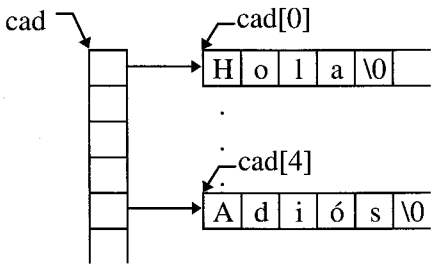
Esto no significa que no podamos utilizar la aritmética de direcciones expuesta en el ejemplo anterior. Estamos diciendo que sí es posible escribir **(*(a+i)+j)* en lugar de *a[i][j]*, aunque por facilidad es preferible utilizar esta última notación.

Array de punteros a cadenas de caracteres

En un array de punteros a cadenas de caracteres cada elemento apunta a un carácter. Por lo tanto, la declaración correspondiente a un array de este tipo es así:

```
char *cad[10];
cad[0] = "Hola";
// ...
cad[4] = "Adiós";
// ...
```

Este ejemplo declara el array de punteros *cad*. Cada elemento es un puntero a un carácter, el primero de un array de caracteres, de ahí la denominación de array de punteros a cadenas de caracteres. Gráficamente se puede ver así:



Como ya hemos indicado anteriormente, una asignación de la forma

```
cad[0] = "Hola";
```

asigna la dirección de la cadena indicada al elemento del array especificado, que tiene que ser un puntero a un **char**. Por otra parte, si en algún momento ha pensado en escribir una sentencia como

```
cad[1] = 'a';
```

obtendrá un error porque el valor asignado a la variable puntero no se corresponde con la dirección de una cadena de caracteres, sino que se trata de un entero (valor ASCII del carácter a) y a un puntero sólo se le puede asignar otro puntero.

Quizá también, piense en escribir el siguiente código para leer todas las cadenas de caracteres:

```
for (i = 0; i < 10; i++)
  gets(cad[i]);
```

Siendo *cad* un array de punteros, escribir el código anterior es un error ¿Por qué? La función **gets** lee una cadena de caracteres de la entrada estándar y la almacena en la cadena de caracteres especificada, en nuestro caso la tiene que colocar a partir de la dirección especificada por *cad[i]*, pero ¿qué dirección es ésta si el array no ha sido inicializado? Posiblemente sea una dirección ya ocupada.

La solución a este problema es asignar dinámicamente memoria para cada cadena que leamos, cuestión que veremos un poco más adelante. No obstante, para aclarar un poco más lo expuesto, vea la diferencia entre el ejemplo anterior y el siguiente (le remito al apartado "Arrays de cadenas de caracteres" visto en el capítulo anterior):

```
char cad[10][81];
for (i = 0; i < 10; i++)
  gets(cad[i]);
```

La declaración **char** **cad[10]* hace que el compilador reserve memoria para un array de punteros a cadenas de caracteres, pero no reserva memoria para cada una de las cadenas, de ahí que tengamos que hacerlo nosotros durante la ejecución. En cambio, la declaración **char** *cad[10][81]* hace que el compilador reserve memoria para 10 cadenas de 81 caracteres cada una.

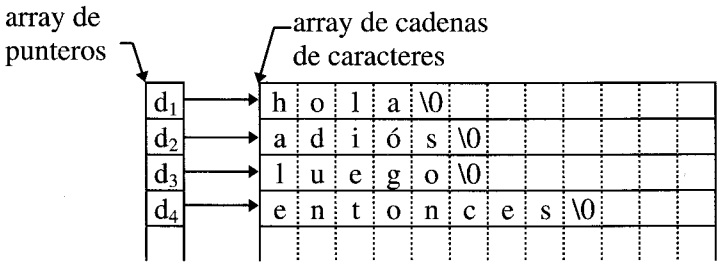
Vamos a estudiar mediante un ejemplo, cómo inicializar un array de punteros a cadenas de caracteres. Consideremos el problema de escribir una función que reciba como parámetro un número entero correspondiente a un mes y nos devuelva como resultado un puntero a una cadena de caracteres que se corresponda con el nombre de dicho mes.

```
/*** Función que devuelve el nombre del mes 1 a 12 dado ***/
/* ptrcads.c
 */
#include <stdio.h>
/* Prototipo de la función. Devuelve una cadena de caracteres */
char *nombre_mes(unsigned int mm);

void main()
{
  unsigned int dia, mes, anyo;
  char *m;

  printf("Introduce fecha (dd-mm-aaaa): ");
  /* Los datos en la entrada irán separados por '-' */
  scanf("%u-%u-%u", &dia, &mes, &anyo);
  m = nombre_mes(mes);
  printf("\nMes: %s\n", m);
}

char *nombre_mes(unsigned int mm)
{
  /* mes es un array de punteros a cadenas de caracteres */
  static char *mes[] = { "Mes no correcto",
                         "Enero", "Febrero", "Marzo",
                         "Abril", "Mayo", "Junio", "Julio",
                         "Agosto", "Septiembre", "Octubre",
                         "Noviembre", "Diciembre"                };
  return ((mm > 0 && mm <= 12) ? mes[mm] : mes[0]);
}
```

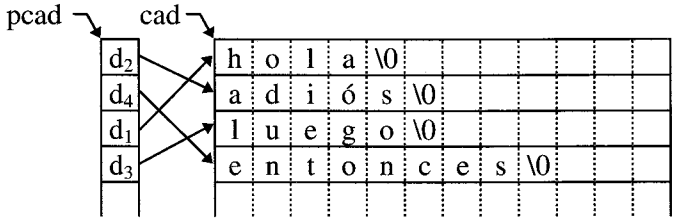
En este ejemplo, *mes* es un array de 13 elementos (0 a 12) que son punteros a cadenas de caracteres. Como se ve, éstas son de diferente longitud y todas son finalizadas automáticamente con el carácter nulo. Si en lugar de utilizar un array de punteros, hubiéramos utilizado un array de dos dimensiones, el número de columnas sería el de la cadena más larga, mas uno para el carácter nulo, con lo que la ocupación de memoria sería mayor.

Las declaraciones **char** **mes[]* y **char** ***mes* no son equivalentes. La primera declara un array de punteros a cadenas de caracteres y la segunda un puntero a un puntero a una cadena de caracteres. Por lo tanto, sería un error intercambiarlas.

Para ilustrar la utilización de los arrays de punteros a cadenas de caracteres vamos a escribir un programa que ordene alfabéticamente un conjunto de cadenas de caracteres referenciadas por un array de punteros. Para realizar este proceso de una forma eficiente, vamos a almacenar las cadenas en un array de caracteres de dos dimensiones y las direcciones de las mismas en un array de punteros.



Esto permite realizar la ordenación comparando las cadenas y modificando el orden de los elementos del array de punteros, evitando así cambiar el orden de las cadenas de caracteres, lo que da lugar a una mayor velocidad de ejecución. Más adelante realizaremos este mismo proceso, almacenando las cadenas de caracteres dinámicamente en memoria con lo que ganaremos en velocidad y optimizaremos el espacio requerido para almacenamiento.



Observe en la figura anterior que el orden de los elementos del array de punteros se corresponde con el orden alfabético de las cadenas de caracteres. Por ejemplo, si visualizamos las cadenas de caracteres utilizando el array de punteros, las cadenas aparecerán en pantalla ordenadas alfabéticamente.

La estructura del programa estará formada por la función **main** y por las funciones:

```
int  LeerCadena(char cad[][CMAX], char *pcad[], int nmc);
void Ordenar(char *pcad[], int nc);
void Visualizar(char *pcad[], int nc);
```

La función *LeerCadena* recibe como parámetros el array donde hay que almacenar las cadenas de caracteres, el array de punteros a las cadenas y el número máximo de cadenas que se pueden almacenar en el array. Esta función devolverá el número de cadenas leídas o el valor –1 si el número de cadenas a ordenar supera el número máximo de cadenas que se pueden almacenar en el array. Por cada cadena leída, se almacenará su dirección en el array de punteros. La entrada finalizará cuando al introducir una nueva cadena pulsemos solamente la tecla *<Entrar>*. Según lo expuesto, la función puede escribirse así:

```
int LeerCadena(char cad[][CMAX], char *pcad[], int nmc)
{
  /* nmc = número máximo de cadenas que se pueden leer */
  int longitud, ncads = 0;

  while ((longitud = strlen(gets(cad[ncads]))) > 0)
  {
    if (ncads >= nmc)
      return (-1);  /* demasiadas cadenas a ordenar */
    else
      /* guardar la dirección de comienzo de la cadena en el array */
      pcad[ncads++] = cad[ncads];
  }
  return (ncads);  /* número de cadenas leídas */
}
```

La variable *longitud* valdrá cero cuando al introducir una nueva cadena pulsemos solamente la tecla *<Entrar>*, finalizando así el bucle **while**.

La función *Ordenar* recibe como parámetros el array de punteros a las cadenas de caracteres y el número de cadenas a ordenar. Esta función ya fue desarrollada en los "ejercicios resueltos" del capítulo anterior (vea también en el capítulo "Algoritmos" el algoritmo de ordenación basado en el método de la burbuja). El código de esta función puede ser el siguiente:

```
void Ordenar(char *pcad[], int nc)
{
  char *aux;  /* puntero auxiliar */
  int i, s = 1;

  while ((s == 1) && (--nc > 0))
  {
    s = 0;  /* no permutación */
```

```
    for (i = 1; i <= nc; i++)
      if (strcmp(pcad[i-1], pcad[i]) > 0)
      {
        aux = pcad[i-1];
        pcad[i-1] = pcad[i];
        pcad[i] = aux;
        s = 1;  /* permutación */
      }
  }
}
```

Observe que el bucle **while** finaliza cuando al recorrer el array para comparar las cadenas entre sí, no se detecta ninguna desordenación (la variable *s* que inicialmente vale cero, seguirá valiendo cero) o, en el caso más desfavorable, cuando se han comparado todas las cadenas con todas, lo que ocurre al recorrer el array *nc–1* veces.

La función *Visualizar* recibe como parámetros el array de punteros a las cadenas de caracteres y el número de elementos del mismo, que coincide con el número de cadenas de caracteres. Esta función puede ser así:

```
void Visualizar(char *pcad[], int nc)
{
  /* nc = número de cadenas a visualizar */
  while (--nc >= 0)
    printf("%s\n", *pcad++);
}
```

La función **main** utilizará las funciones anteriores para realizar el proceso descrito. El programa completo se muestra a continuación.

```
/**************** Ordenar cadenas de caracteres ***************/
/* ordcads.c
 */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define NMAX  25    /* número máximo de cadenas */
#define CMAX  81    /* número máximo de caracteres por cadena */

void main()   /* función principal */
{
  char cad[NMAX][CMAX];  /* array de cadenas */
  char *pcad[NMAX];      /* array de punteros a las cadenas */
  int  ncads;            /* número de cadenas leídas */

  int  LeerCadena(char cad[][CMAX], char *pcad[], int nmc);
  void Ordenar(char *pcad[], int nc);
  void Visualizar(char *pcad[], int nc);

  printf("Ordenación de cadenas de caracteres.\n");
  printf("Introduzca las cadenas a ordenar. Pulse <Entrar> para salir.\n");
```

```
  if ((ncads = LeerCadena(cad, pcad, NMAX)) > 0)
  {
    printf("Proceso de ordenación.\n\n");
    Ordenar(pcad, ncads);
    Visualizar(pcad, ncads);
  }
  else
    printf("Cero o demasiadas cadenas para ordenar\n");
}

/*****************************************************************
                    Función leer cadenas
*****************************************************************/

int LeerCadena(char cad[][CMAX], char *pcad[], int nmc)
{
  /* nmc = número máximo de cadenas que se pueden leer */
  int longitud, ncads = 0;

  while ((longitud = strlen(gets(cad[ncads]))) > 0)
  {
    if (ncads >= nmc)
      return (-1);  /* demasiadas cadenas a ordenar */
    else
      /* guardar el apuntador a la cadena en el array */
      pcad[ncads++] = cad[ncads];
  }
  return (ncads);  /* número de cadenas leídas */
}

/*****************************************************************
                    Función ordenar cadenas
*****************************************************************/

void Ordenar(char *pcad[], int nc)
{
  char *aux;  /* puntero auxiliar */
  int i, s = 1;

  while ((s == 1) && (--nc > 0))
  {
    s = 0;  /* no permutación */
    for (i = 1; i <= nc; i++)
      if (strcmp(pcad[i-1], pcad[i]) > 0)
      {
        aux = pcad[i-1];
        pcad[i-1] = pcad[i];
        pcad[i] = aux;
        s = 1;  /* permutación */
      }
  }
}

/*****************************************************************
                  Función visualizar cadenas
*****************************************************************/
```

```
void Visualizar(char *pcad[], int nc)
{
  /* nc = número de cadenas a Visualizar */
  while (--nc >= 0)
    printf("%s\n", *pcad++);
}
```

ASIGNACIÓN DINÁMICA DE MEMORIA

C cuenta fundamentalmente con dos métodos para almacenar información en la memoria. El primero utiliza variables globales y locales. En el caso de variables globales, el espacio es fijado para ser utilizado a lo largo de toda la ejecución del programa; y en el caso de variables locales, la asignación se hace a través del **stack**; en este caso, el espacio es fijado temporalmente, mientras la variable existe. El segundo método utiliza funciones pertenecientes a la biblioteca de C, como **malloc** y **free**. Como es lógico, estas funciones utilizan el área de memoria libre para realizar las asignaciones de memoria.

La *asignación dinámica de memoria* consiste en asignar la cantidad de memoria necesaria para almacenar un objeto durante la ejecución del programa, en vez de hacerlo en el momento de la compilación del mismo. Cuando se asigna memoria para un objeto de un tipo cualquiera, se devuelve un puntero a la zona de memoria asignada. Según esto, lo que tiene que hacer el compilador es asignar una cantidad fija de memoria para almacenar la dirección del objeto asignado dinámicamente, en vez de hacer una asignación para el objeto en sí. Esto implica declarar un puntero a un tipo de datos igual al tipo del objeto que se quiere asignar dinámicamente. Por ejemplo, si queremos asignar memoria dinámicamente para un array de enteros, el objeto apuntado será el primer entero lo que implica declarar un puntero a un entero; esto es,

```
int *pa;
```

Funciones para asignación dinámica de memoria

La biblioteca de C proporciona fundamentalmente una función para asignar memoria dinámicamente, **malloc**, y otra para liberar el espacio de memoria asignado para un objeto cuando este ya no sea necesario, **free**.

malloc

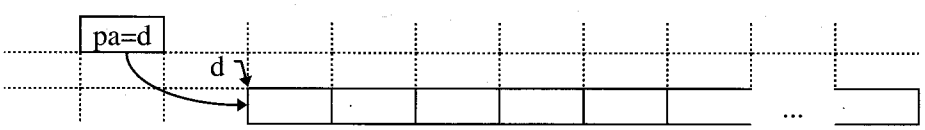
```
#include <stdlib.h>
void *malloc( size_t nbytes );
```
Compatibilidad: ANSI, UNIX y MS-DOS

La función **malloc** permite asignar un bloque de memoria de *nbytes* bytes para almacenar uno o más objetos de un tipo cualquiera. Esta función devuelve un puntero a **void** que referencia el espacio asignado. En ANSI C, el valor devuelto será convertido implícitamente a un puntero a un objeto del tipo especificado en la declaración de la variable utilizada para almacenar ese valor; en C++ y en algunos compiladores de C, esta conversión hay que realizarla explícitamente (conversión *cast*). Si hay insuficiente espacio de memoria o el argumento *nbytes* es 0, la función **malloc** retorna un puntero nulo (valor **NULL**). El bloque de memoria asignado se corresponde con un conjunto de bytes consecutivos en memoria.

Por ejemplo, las siguientes líneas de código asignan memoria para 100 enteros. Como los bytes de memoria asignados por **malloc** están consecutivos en memoria, el espacio asignado se corresponde con un array de 100 enteros. La dirección de comienzo del array es el puntero utilizado para almacenar la dirección del bloque de memoria asignado, que es devuelta por **malloc**.

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

int main()
{
  int *pa = NULL;
  int nbytes = 100 * sizeof(int);

  pa = malloc(nbytes);
  if (pa == NULL )
  {
    printf("Insuficiente espacio de memoria\n");
    return -1;
  }
  printf("Se han asignado %u bytes de memoria\n", nbytes);
  // ...
}
```

Observe que el número de bytes que hay que asignar dinámicamente a *pa* es el número de elementos del array por el tamaño de un elemento y que después de invocar a la función **malloc** verificamos si ha sido posible realizar la asignación de memoria solicitada. Si el valor devuelto por **malloc** es un puntero nulo (valor **NULL**) quiere decir que la asignación de memoria no se ha podido realizar.



La función **malloc** devuelve la dirección *d* del bloque de memoria asignado.

free

```
#include <stdlib.h>
void free( void *vpuntero );
Compatibilidad: ANSI, UNIX y MS-DOS
```

La función **free** libera un bloque de memoria asignado por las funciones **malloc**, **calloc** o **realloc** (éstas dos últimas, las veremos a continuación). Si el puntero que referencia el bloque de memoria que deseamos liberar es nulo, se ignora.

Si la memoria liberada por **free** no ha sido previamente asignada con **malloc**, **calloc** o **realloc**, se pueden producir errores durante la ejecución del programa. Por ejemplo, si a un puntero le asignamos la dirección de un array estático, ese espacio de memoria no hay que liberarlo.

El siguiente ejemplo es otra versión del ejemplo anterior, que además de asignar un bloque de memoria, utiliza la función **free** para liberarlo. Es un buen estilo de programación liberar la memoria asignada cuando ya no se necesite (en el sistema operativo MS-DOS la memoria no liberada crea lagunas de memoria o fugas de memoria; esto es, los bloques de memoria no liberados no están disponibles hasta que no se reinicie la máquina).

```
#include <stdio.h>
#include <stdlib.h>
void main()
{
  int *pa = NULL;
  unsigned int nbytes = 100 * sizeof(int);

  if ((pa = (int *)malloc(nbytes)) == NULL )
  {
    printf("Insuficiente espacio de memoria\n");
    exit(-1);
  }
  printf("Se han asignado %u bytes de memoria\n", nbytes);
  /* Operaciones */
  free(pa);
}
```

En este caso se ha utilizado una conversión *cast* para convertir el puntero a **void** devuelto por **malloc**, a un puntero a **int** (tipo de los objetos apuntados). La asignación de memoria se ha integrado en la condición de la sentencia **if**. También, como otra alternativa, se ha utilizado la función **exit** en lugar de la sentencia **return**. La función **exit** finaliza el programa; en cambio la sentencia **return** devuelve el control a la función que a su vez llamó a la que contiene la sentencia **return**; si la sentencia **return** pertenece a la función **main**, lógicamente el programa finaliza.

ARRAYS DINÁMICOS

Hasta ahora todos los arrays que hemos manipulado, excepto en el ejemplo anterior, eran estáticos. Esto exigía conocer en el momento de escribir el código del programa, cuál era la dimensión del array y expresar esta dimensión como una constante entera. Por ejemplo,

```
#define NMAX 100
// ...
int a[NMAX];
```

Ahora, utilizando la técnica de asignar memoria dinámicamente, podremos decidir durante la ejecución cuántos elementos queremos que tenga nuestro array. Este tipo de arrays recibe el nombre de *arrays dinámicos*; esto es, arrays creados durante la ejecución del programa. Igual que ocurría con los arrays estáticos, los elementos de un array dinámico pueden ser de cualquier tipo.

Para asignar memoria dinámicamente para un array, además de la función **malloc**, la biblioteca de C proporciona la función **calloc** cuya sintaxis es:

```
#include <stdlib.h>
void *calloc( size_t nelementos, size_t tamelem );
Compatibilidad: ANSI, UNIX y MS-DOS
```

El primer parámetro, *nelementos*, especifica el número de elementos del array y el segundo, *tamelem*, el tamaño en bytes de cada elemento. La función **calloc** devuelve un puntero a **void** que referencia el espacio de memoria asignado. Según esto, el código

```
int *pa = NULL;

if ((pa = (int *)malloc(100 * sizeof(int))) == NULL )
{
  printf("Insuficiente espacio de memoria\n");
  exit(-1);
}
```

es equivalente a

```
int *pa = NULL;

if ((pa = (int *)calloc(100, sizeof(int))) == NULL )
{
  printf("Insuficiente espacio de memoria\n");
  exit(-1);
}
```

Arrays dinámicos enteros o reales

La función **malloc** asigna un bloque de memoria especificado en bytes y devuelve un puntero a **void** que es convertido implícita o explícitamente a un puntero a un objeto del tipo especificado en la declaración de la variable utilizada para referenciar el bloque de memoria asignado. Por lo tanto, para crear un array durante la ejecución del programa es necesario haber declarado una variable que apunte a objetos del tipo de los elementos del array.

Como ejemplo vamos a presentar otra versión del programa anterior para que ahora solicite la entrada del número de elementos del array desde el teclado.

```
/*************** Array dinámico de una dimensión ***************/
/* arrdim01.c
 */
#include <stdio.h>
#include <stdlib.h>

void main()
{
  int *pa = NULL;
  unsigned int nbytes = 0, Ok = 0, nElementos = 0, i = 0;

  do
  {
    printf("Número de elementos del array: ");
    Ok = scanf("%u", &nElementos);
    fflush(stdin);
  }
  while ( !Ok );

  nbytes = nElementos * sizeof(int);

  if ((pa = (int *)malloc(nbytes)) == NULL )
  {
    printf("Insuficiente espacio de memoria\n");
    exit(-1);
  }
  printf("Se han asignado %u bytes de memoria\n", nbytes);

  /* Inicializar el array a cero */
  for ( i = 0; i < nElementos; i++ )
    pa[i] = 0;

  /* Operaciones */

  free(pa);
}
```

El ejemplo que acabamos de exponer solicita al usuario del mismo, que teclee el número de elementos que desea que tenga el array; el valor introducido es verificado para asegurar que se trata de un valor numérico. A continuación se asigna

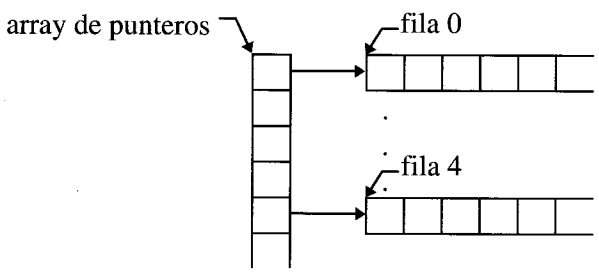
memoria para el array y si la asignación es posible, se inicializan los elementos del array con el valor cero.

Es importante que observe que puede utilizar la indexación del array, *pa[i]*, para acceder a los elementos del mismo, concepto que ya fue explicado anteriormente en este mismo capítulo. También tiene que saber que cuando se accede a un elemento de un array dinámico, C toma como referencia el tipo del objeto apuntado por el puntero y asigna para el elemento referenciado el número de bytes correspondiente a ese tipo. En el ejemplo que estamos analizando, *pa* es un puntero a un **int**, por lo tanto *pa[i]* accede al contenido de un número de bytes igual al tamaño de un **int** (2 o 4 bytes) localizados a partir de la dirección *pa+i*.

Para asignar memoria para un array de dos dimensiones, el proceso se divide en dos partes:

- Asignar memoria para un array de punteros, cuyos elementos referenciarán cada una de las filas del array.
- Asignar memoria para cada una de las filas. El número de elementos de cada fila puede ser variable.

Gráficamente podemos ver que tenemos que construir una estructura como la siguiente:



Esta estructura hace el mismo papel que un array de dos dimensiones, con una ventaja, que las filas pueden ser de cualquier longitud.

Para crear el array de punteros, primero tenemos que declarar un puntero a un puntero, puesto que sus elementos van a ser punteros a objetos de un determinado tipo. Por ejemplo, si suponemos que los objetos van a ser enteros, declararemos un puntero así:

```
int **ppa; /* puntero que referencia el array de punteros */
```

El paso siguiente es asignar memoria para el array de punteros. Supongamos que el array de dos dimensiones que deseamos construir tiene *nFilas*. Esto implica que el array de punteros tiene que tener *nFilas* elementos y que cada uno de los elementos será un puntero a un entero (al primer entero de cada fila). Según esto, para asignar memoria para el array de punteros escribiremos:

```
ppa = malloc(nFilas * sizeof(int *));
```

Y por último escribimos el código necesario para asignar memoria para cada una de las filas. Supongamos que todas tienen *nCols* elementos de tipo **int**.

```
for (f = 0; f < nFilas; f++)
  ppa[f] = malloc(nCols * sizeof(int));
```

A continuación se muestra un ejemplo completo, que asigna memoria para un array de dos dimensiones, inicializa todos sus elementos a cero y finalmente libera la memoria asignada.

```
/************** Array dinámico de dos dimensiones **************/
/* arrdim02.c
 */
#include <stdio.h>
#include <stdlib.h>

void main()
{
  int **ppa = NULL;
  unsigned int nFilas = 0, nCols = 0;
  unsigned int Ok = 0, f = 0, c = 0;

  do
  {
    printf("Número de filas del array:    ");
    Ok = scanf("%u", &nFilas);
    fflush(stdin);
  }
  while ( !Ok );
  do
  {
    printf("Número de columnas del array: ");
    Ok = scanf("%u", &nCols);
    fflush(stdin);
  }
  while ( !Ok );

  /* Asignar memoria para el array de punteros */
  if ((ppa = (int **)malloc(nFilas * sizeof(int *))) == NULL)
  {
    printf("Insuficiente espacio de memoria\n");
    exit(-1);
  }

  /* Asignar memoria para cada una de las filas */
```

```
  for (f = 0; f < nFilas; f++)
  {
    if ((ppa[f] = (int *)malloc(nCols * sizeof(int))) == NULL)
    {
      printf("Insuficiente espacio de memoria\n");
      exit(-1);
    }
  }

  /* Inicializar el array a cero */
  for ( f = 0; f < nFilas; f++ )
    for ( c = 0; c < nCols; c++ )
      ppa[f][c] = 0;

  /* Operaciones */

  /* Liberar la memoria asignada a cada una de las filas */
  for ( f = 0; f < nFilas; f++ )
    free(ppa[f]);
  /* Liberar la memoria asignada al array de punteros */
  free(ppa);
}
```

Observe que para liberar la memoria ocupada por el array, el proceso que se sigue es inverso al realizado para crear el array; esto es, primero liberamos la memoria asignada a cada una de las filas y después la asignada al array de punteros.

Arrays dinámicos de cadenas de caracteres

Un array dinámico de cadenas de caracteres es un array de dos dimensiones cuyos elementos son de tipo **char**. Por lo tanto, su construcción es idéntica a los arrays de dos dimensiones que acabamos de ver en el apartado anterior.

Los arrays de cadenas de caracteres son un caso típico donde las filas tienen un número de elementos variable, dependiendo esto de la longitud de la cadena que se almacene en cada fila.

Para ilustrar cómo se trabaja con este tipo de arrays, vamos a realizar un ejemplo que cree un array dinámico de cadenas de caracteres, asigne las cadenas de caracteres correspondiente, las ordene alfabéticamente en orden ascendente y finalmente, visualice las cadenas de caracteres ordenadas. La estructura del programa estará formada por la función **main** y las funciones:

```
unsigned int LeerCadenas(char **pcad, unsigned nFilas);
void OrdenarCadenas(char **pcad, unsigned filas);
void VisualizarCadenas(char **pcad, unsigned filas);
```

El argumento *pcad* es un puntero al array de punteros a las cadenas de caracteres, *nFilas* es el número máximo de cadenas que podemos almacenar y *filas* es el número real de cadenas que contiene el array. La función *LeerCadenas* devolverá el número de cadenas leídas.

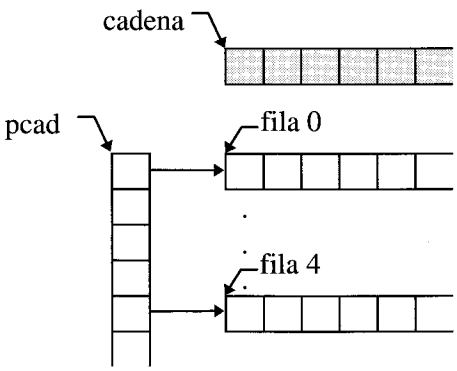
El proceso que seguiremos para realizar este problema es:

- Asignamos memoria para el array de punteros.

```
if ((pcad = (char **)malloc(nFilas * sizeof(char *))) == NULL)
{
  printf("Insuficiente espacio de memoria\n");
  exit(-1);
}
```

No asignamos memoria para cada una de las cadenas porque no conocemos su longitud. Por lo tanto este proceso lo desarrollaremos paralelamente a la lectura de cada una de las ellas.

- Leemos las cadenas de caracteres. Para poder leer una cadena, necesitamos definir una array de caracteres que vamos a denominar *cadena*. Este será un array estático de longitud 81 caracteres. Una vez leída la cadena, calculamos su longitud, reservamos memoria para almacenar un número de caracteres igual a la longitud de la cadena más uno (el carácter nulo), asignamos el puntero al bloque de memoria reservado al siguiente elemento vacío del array de punteros y copiamos *cadena* en el nuevo bloque asignado (fila del array de cadenas). Este proceso lo repetiremos para cada una de las cadenas de caracteres que leamos.




```
unsigned int LeerCadenas(char **pcad, unsigned nFilas)
{
  unsigned int f = 0, longitud = 0;
  char cadena[81];
```

```
    printf("Introducir cadenas de caracteres.\n");
    printf("Para finalizar introduzca una cadena nula.\n");
    printf("Esto es, pulse sólo <Entrar>.\n\n");
    while ((longitud = strlen(gets(cadena))) > 0 && f < nFilas)
    {
      /* Asignar espacio para una cadena de caracteres */
      if ((pcad[f] = (char *)malloc(longitud + 1)) == NULL)
      {
        printf("Insuficiente espacio de memoria disponible\n");
        exit(-1);       /* terminar el proceso */
      }
      /* copiar la cadena en el espacio de memoria asignado */
      strcpy(pcad[f], cadena);
      f++;
    }
    return(f);
  }
```

La sentencia *pcad[f] = (char *)malloc(longitud+1)* asigna un espacio de memoria de *longitud+1* bytes, para la cadena de caracteres almacenada en *cadena*. Recuerde que la función **strlen** no contabiliza el carácter nulo de terminación, de ahí la expresión *longitud+1*.

- Una vez leído el array lo ordenamos (este proceso ya ha sido expuesto en este capítulo y en el anterior) y finalmente lo visualizamos y liberamos la memoria ocupada por el array de cadenas de caracteres.

El programa completo se muestra a continuación.

```
/*********** Array dinámico de cadenas de caracteres ***********/
/* arrdim03.c
 */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

unsigned int LeerCadenas(char **pcad, unsigned nFilas);
void OrdenarCadenas(char **pcad, unsigned filas);
void VisualizarCadenas(char **pcad, unsigned filas);

void main()
{
  char **pcad = NULL;
  unsigned int nFilas = 0;
  unsigned int Ok = 0, filas = 0, f = 0;

  do
  {
    printf("Número de filas del array:    ");
    Ok = scanf("%u", &nFilas);
    fflush(stdin);
  }
  while ( !Ok );
```

```
  /* Asignar memoria para el array de punteros */
  if ((pcad = (char **)malloc(nFilas * sizeof(char *))) == NULL)
  {
    printf("Insuficiente espacio de memoria\n");
    exit(-1);
  }

  /* Operaciones */
  filas = LeerCadenas(pcad, nFilas);
  OrdenarCadenas(pcad, filas);
  VisualizarCadenas(pcad, filas);

  /* Liberar la memoria asignada a cada una de las filas */
  for ( f = 0; f < filas; f++ )
    free(pcad[f]);
  /* Liberar la memoria asignada al array de punteros */
  free(pcad);
}

unsigned int LeerCadenas(char **pcad, unsigned nFilas)
{
  unsigned int f = 0, longitud = 0;
  char cadena[81];

  printf("Introducir cadenas de caracteres.\n");
  printf("Para finalizar introduzca una cadena nula.\n");
  printf("Esto es, pulse sólo <Entrar>.\n\n");

  while ((longitud = strlen(gets(cadena))) > 0 && f < nFilas)
  {
    /* Asignar espacio para una cadena de caracteres */
    if ((pcad[f] = (char *)malloc(longitud + 1)) == NULL)
    {
      printf("Insuficiente espacio de memoria disponible\n");
      exit(-1);   /* terminar el proceso */
    }
    /* copiar la cadena en el espacio de memoria asignado */
    strcpy(pcad[f], cadena);
    f++;
  }
  return(f);
}

void OrdenarCadenas(char **pcad, unsigned filas)
{
  char *aux;  /* puntero auxiliar */
  unsigned int i = 0, s = 1;

  while ((s == 1) && (--filas > 0))
  {
    s = 0;  /* no permutación */
    for (i = 1; i <= filas; i++)
      if (strcmp(pcad[i-1], pcad[i]) > 0)
      {
        aux = pcad[i-1];
        pcad[i-1] = pcad[i];
        pcad[i] = aux;
```

```
            s = 1;  /* permutación */
          }
      }
  }

void VisualizarCadenas(char **pcad, unsigned filas)
{
  unsigned int f = 0;

  for ( f = 0; f < filas; f++ )
    printf("%s\n", pcad[f]);
}
```

REASIGNAR UN BLOQUE DE MEMORIA

En alguna ocasión necesitaremos cambiar el tamaño de un bloque de memoria previamente asignado. Para realizar esto, la biblioteca de C proporciona la función **realloc** que tiene la siguiente sintaxis:

```
#include <stdlib.h>
void *realloc( void *pBlomem, size_t nBytes );
Compatibilidad: ANSI, UNIX y MS-DOS
```

El parámetro *pBlomem* es un puntero que apunta al comienzo del bloque de memoria actual. Si *pBlomem* es **NULL**, esta función se comporta igual que **malloc** y asigna un nuevo bloque de *nBytes* bytes. Si *pBlomem* no es un puntero nulo, entonces tiene que ser un puntero devuelto por las funciones **malloc**, **calloc** o por la propia función **realloc**. El bloque ha podido, incluso, ser liberado por la función **free**. El argumento *nBytes*, da el nuevo tamaño del bloque en bytes. El contenido del bloque no cambia en el espacio conservado.

La función **realloc** devuelve un puntero al espacio asignado. El bloque puede ser movido al modificar el tamaño, esto quiere decir que *pBlomem* puede cambiar.

Si hay insuficiente espacio de memoria o si *nBytes* es 0, la función retorna un puntero nulo. Cuando esto ocurre, el bloque original es liberado.

El siguiente programa muestra cómo realizar una reasignación de memoria y pone de manifiesto que después de una reasignación, la información no varía en el espacio de memoria conservado. Por último, el bloque de memoria es liberado.

```
/******************** Función realloc ********************/
/* realloc.c
 */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
```

```
void main()
{
  const int BYTES = 40;
  char *p;

  /* Asignar espacio para una cadena de caracteres */
  p = malloc(BYTES * sizeof(char));

  /* Copiar una cadena de caracteres en el espacio asignado */
  strcpy(p, "abcdef");

  /* Reasignar el bloque para que pueda contener más caracteres */
  if (p != NULL)
    p = realloc(p, BYTES * 2 * sizeof(char));

  if (p != NULL)
  {
    printf("Bloque reasignado\n");
    /* Escribir la cadena original */
    printf("%s\n", p);
    /* Liberar el espacio de memoria */
    free(p);
    printf("\nEl bloque ha sido liberado\n");
  }
  else
  {
    printf("La reasignación no ha sido posible\n");
    printf("El espacio ocupado por el bloque ha sido liberado");
  }
}
```

PUNTEROS A ESTRUCTURAS

Los punteros a estructuras se declaran igual que los punteros a otros tipos de datos. Para referirse a un miembro de una estructura apuntada por un puntero hay que utilizar el operador –>.

Por ejemplo, el siguiente programa declara un puntero *hoy* a una estructura de tipo **struct** *fecha*, asigna memoria para la estructura, lee valores para cada miembro de la misma y, apoyándose en una función, escribe su contenido.

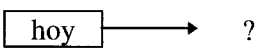
```
/****************** Punteros a estructuras ******************/
/* pstruct.c
 */
#include <stdio.h>
#include <stdlib.h>

struct fecha
{
  unsigned int dd;
  unsigned int mm;
  unsigned int aa;
};
```

```
void escribir(struct fecha *f);

void main()
{
  struct fecha *hoy; /* hoy es un puntero a una estructura */

  /* asignación de memoria para la estructura */
  hoy = (struct fecha *)malloc(sizeof(struct fecha));

  printf("Introducir fecha  (dd-mm-aa): ");
  scanf("%u-%u-%u", &hoy->dd, &hoy->mm, &hoy->aa);
  escribir(hoy);
}

void escribir(struct fecha *f)
{
  printf("Día %u del mes %u del año %u\n", f->dd, f->mm, f->aa);
}
```

Observe que el tipo **struct** *fecha* se ha declarado al principio, antes de cualquier función, lo que permite utilizarlo en cualquier parte.

Otro detalle importante es comprender que el simple hecho de declarar un puntero a una estructura no significa que dispongamos de la estructura; es necesario asignar al puntero un bloque de memoria del tamaño de la estructura donde se almacenarán los datos de la estructura (este concepto es aplicable a cualquier tipo de objetos). Esto es, la declaración siguiente crea un puntero para apuntar a una estructura, pero no la estructura.

```
struct fecha *hoy;
```



Por lo tanto sería un error ejecutar una sentencia como

```
scanf("%u-%u-%u", &hoy->dd, &hoy->mm, &hoy->aa);
```

porque *dd*, *mm* y *aa* ¿a qué estructura pertenecen? La respuesta es, a ninguna por que al puntero *hoy* no se le ha asignado una estructura. Si hubiéramos hecho la siguiente declaración,

```
struct fecha f, *hoy = &f;
```

si sería válido ejecutar la sentencia

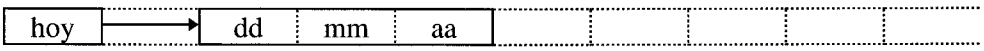
```
scanf("%u-%u-%u", &hoy->dd, &hoy->mm, &hoy->aa);
```

porque ahora *hoy* apunta a la estructura *f*. Pero se preguntará, y con razón, para qué queremos el puntero a la estructura; por qué no utilizar directamente la estructura *f* así:

```
scanf("%u-%u-%u", &f.dd, &f.mm, &f.aa);
```

Esto evidencia que cuando declaramos un puntero a un objeto, casi siempre es porque el objeto va a ser creado en tiempo de ejecución. Es decir,

```
struct fecha *hoy;
hoy = malloc(sizeof(struct fecha));
```



Ahora a *hoy* se le ha asignado un bloque de memoria capaz de almacenar una estructura del tipo **struct** *fecha*. Por lo tanto sí es correcto ejecutar una sentencia como

```
scanf("%u-%u-%u", &hoy->dd, &hoy->mm, &hoy->aa);
```

Los punteros a uniones se manipulan exactamente igual que los punteros a estructuras.

Como conclusión podemos decir que declarar un puntero a un objeto de cualquier tipo no sirve de nada mientras no le asignemos un bloque de memoria capaz de almacenar un objeto de ese tipo. Esto es, en la mayoría de los casos, la declaración de un puntero implica llamar a continuación a la función **malloc** para asignarle el bloque de memoria que va a contener el objeto apuntado.

DECLARACIONES COMPLEJAS

Una declaración compleja es un identificador calificado por más de un operador (array: [], puntero: *, o función: ()). Se pueden aplicar varias combinaciones con estos operadores sobre un identificador; sin embargo, los elementos de un array no pueden ser funciones y una función no puede devolver como resultado un array o una función.

Para interpretar estas declaraciones, hay que saber que los corchetes y paréntesis (operadores a la derecha del identificador) tienen prioridad sobre los asteriscos (operadores a la izquierda del identificador). Los paréntesis y corchetes tienen la misma prioridad y se evalúan de izquierda a derecha. Como último paso se aplica el tipo especificado. Utilizando paréntesis, podemos cambiar el orden de prioridades. Las expresiones entre paréntesis se evalúan primero, de más internas a más externas.

Una forma simple de interpretar declaraciones complejas es leerlas desde dentro hacia afuera, de acuerdo con los siguientes pasos:

1. Comenzar con el identificador y mirar si hacia la derecha hay corchetes o paréntesis.
2. Interpretar esos corchetes o paréntesis y mirar si hacia la izquierda hay asteriscos.
3. Dentro de cada nivel de paréntesis, de más internos a más externos, aplicar las reglas 1 y 2.

El siguiente ejemplo clarifica lo expuesto. En él se han enumerado el identificador *var*, los calificadores *[]*, *()* y *, y el tipo **char**, en el orden de interpretación resultado de aplicar las reglas anteriores.

```
char *(*(*var)())[10]
  ↑  ↑ ↑ ↑ ↑  ↑  ↑
  7  6 4 2 1  3  5
```

La lectura que se hace al interpretar la declaración anterior es:

1. El identificador *var* es declarado como
2. un puntero a
3. una función que devuelve
4. un puntero a
5. un array de 10 elementos, los cuales son
6. punteros a
7. objetos de tipo **char**.

EJERCICIOS RESUELTOS

1. Se quiere escribir un programa para manipular polinomios. Para ello, vamos a utilizar una estructura de datos como la siguiente:

```
typedef struct
{
  int grado;    /* grado del polinomio */
  float *coef; /* coeficientes del polinomio */
} tpolinomio;
```

El miembro *grado* es un valor mayor que cero que especifica el grado del polinomio. El miembro *coef* es un puntero que referencia un array cuyos elementos contienen los coeficientes del polinomio. El número de elementos del array es el número de coeficientes del polinomio y depende del grado de éste. Por ejemplo, sea el polinomio

$x^5 + 5x^3 - 7x^2 + 4$

Como el grado del polinomio es 5, el array de los coeficientes tendrá 6 elementos cuyos valores serán: 1, 0, 5, –7, 0 y 4.

Se pide:

a.- Escribir una función *LeerPol* que lea a través del teclado un polinomio y lo almacene en una estructura de tipo *tpolinomio* anteriormente descrita. La función *LeerPol* devolverá el polinomio leído. Para el polinomio anterior la entrada de datos se efectuaría así:

```
Grado del polinomio: 5
Coeficientes de mayor a menor grado: 1 0 5 -7 0 4
```

El prototipo de la función es el siguiente:

```
tpolinomio LeerPol(void);
```

b.- Escribir una función *VisualizarPol* que visualice en pantalla un polinomio. Por ejemplo, el polinomio anterior sería visualizado así:

```
+1x^5 +5x^3 -7x^2 +4
```

El prototipo de la función es el siguiente:

```
void VisualizarPol(tpolinomio pol);
```

El parámetro *pol* es una estructura que especifica el polinomio a visualizar.

c.- Escribir una función *SumarPols* que devuelva como resultado la suma de dos polinomios. El prototipo de esta función es:

```
tpolinomio SumarPols(tpolinomio polA, tpolinomio polB);
```

Los parámetros *polA* y *polB* son estructuras que especifican los polinomios a sumar.

d.- Utilizando las funciones anteriores, escribir un programa que lea dos polinomios y visualice en pantalla su suma.

El programa completo se muestra a continuación.

```
/************************ Polinomios ************************/
/* polinom.c
 */
#include <stdio.h>
```

```
#include <stdlib.h>

typedef struct
{
  int grado;      /* grado del polinomio */
  float *coef;    /* coeficientes del polinomio */
} tpolinomio;

tpolinomio LeerPol(void)
{
  tpolinomio pol;
  int i = 0;

  printf("Grado del polinomio: ");
  scanf("%d", &pol.grado);
  /* Asignar memoria para el array de coeficientes */
  pol.coef = (float *)malloc((pol.grado + 1) * sizeof(tpolinomio));
  if ( pol.coef == NULL )
  {
    printf("Insuficiente memoria\n");
    exit(-1);
  }
  /* Leer los coeficientes de mayor a menor grado */
  printf("Coeficientes de mayor a menor grado: ");
  for ( i = pol.grado; i >= 0; i--)
    scanf("%g", &pol.coef[i]);

  return (pol);
}

void VisualizarPol(tpolinomio pol)
{
  int i = 0;

  /* Escribir los términos de pol de mayor a menor grado */
  for ( i = pol.grado; i > 0; i--)
    if ( pol.coef[i] ) printf("%+gx^%d ", pol.coef[i], i);
  /* Escribir el término independiente */
  if ( pol.coef[i] ) printf("%+g\n", pol.coef[i]);
}

tpolinomio SumarPols(tpolinomio polA, tpolinomio polB)
{
  int i = 0;
  tpolinomio polresu, polaux;

  /* Hacer que polA sea el de mayor grado */
  if ( polA.grado < polB.grado)
  {
    polaux = polA;
    polA = polB;
    polB = polaux;
  }

  /* El polinomio resultante tendrá como grado, el mayor */
  polresu.grado = polA.grado;
```

```
  /* Asignar memoria para el array de coeficientes de polresu */
  polresu.coef = (float *)malloc((polresu.grado + 1) * sizeof(tpolinomio));
  if ( polresu.coef == NULL )
  {
    printf("Insuficiente memoria\n");
    exit(-1);
  }
  /* Sumar polB con los coeficientes correspondientes de polA */
  for ( i = 0; i <= polB.grado; i++)
    polresu.coef[i] = polB.coef[i] + polA.coef[i];
  /* A partir del valor actual de i, copiar
     los coeficientes restantes de polA */
  for ( ; i <= polA.grado; i++)
    polresu.coef[i] = polA.coef[i];

  return (polresu);
}

void main(void)
{
  tpolinomio polA, polB, polR;

  polA = LeerPol();
  polB = LeerPol();
  polR = SumarPols(polA, polB);
  VisualizarPol(polR);

  /* Liberar la memoria asignada */
  free(polA.coef);
  free(polB.coef);
  free(polR.coef);
}
```

2. Números seudoaleatorios. Un algoritmo que genere una secuencia aleatoria o aparentemente aleatoria de números, se llama un generador de números aleatorios. Muchos ejemplos requieren de este método.

El método más comúnmente utilizado para generar números aleatorios es el método de congruencia lineal. Cada número en la secuencia r_k, es calculado a partir de su predecesor r_{k-1}, utilizando la siguiente fórmula:

$$r_k = (multiplicador * r_{k-1} + incremento) \% módulo$$

donde % es el operador módulo o resto de una división entera.

La secuencia, así generada, es llamada más correctamente secuencia seudoaleatoria, ya que cada número generado, depende del anteriormente generado.

El siguiente algoritmo, presentado como una función C, genera 65536 números aleatorios y no causará sobrepasamiento en un ordenador que admita un rango de enteros de -2^{31} a $2^{31}-1$.

```
void rnd(long *prandom)
{
  *prandom = (25173 * *prandom + 13849) % 65536;
}
```

La llamada a esta función, pasa el parámetro por referencia, con la finalidad de generar un número random diferente cada vez. La función genera números enteros comprendidos entre 0 y 65535.

Para la mayoría de las aplicaciones, estos números deberían estar comprendidos dentro de un intervalo requerido. Por ejemplo, si el problema es simular la tirada de un dado, podríamos escribir:

```
/********************* Tirada de un dado *********************/
/* dado.c
 */
#include <stdio.h>
#include <ctype.h>
void rnd(long *prandom);

void main()
{
  unsigned int inicio;  /* inicio contiene un valor entre 0 y 65535 */
  long random = inicio; /* random = número entre 0 y 65535 */
  long tirada;
  char c;

  printf("Para tirar el dado, pulse una <Entrar>\n");
  printf("Para finalizar pulse <f>.\n\n");
  c = getchar();
  fflush(stdin);
  while (tolower(c) != 'f')
  {
    rnd(&random); /* random contiene un número seudoaleatorio */
    tirada = random % 6 + 1; /* número entre 1 y 6 */
    printf("%10d\n", tirada);
    c = getchar();
    fflush(stdin);
  }
}

void rnd(long *prandom)
{
  *prandom = (25173 * *prandom + 13849) % 65536;
}
```

3. Frecuentemente requerimos de un valor aleatorio entre 0 y 1. Para este propósito podemos utilizar una versión modificada como la que se expone a continuación:

```
/********************* Valores entre 0 y 1 *********************/
/* rnd.c
 */
#include <stdio.h>
double rnd(long *prandom);

void main()
{
  unsigned int inicio;  /* inicio contiene un valor entre 0 y 65535 */
  long random = inicio; /* random = número entre 0 y 65535 */
  double n;
  int i;

  for (i = 10; i; i--)
  {
    n = rnd(&random);
    printf("%.8g\n", n);
  }
}

double rnd(long *prandom)
{
  *prandom = (25173 * *prandom + 13849) % 65536;
  return((double)*prandom / (double)65535);
}
```

Esta función da valores aleatorios en el rango 0 a 1.

4. Supongamos que tenemos un sólido irregular *S*, el cual puede encerrarse en un cubo *C*. Puede demostrarse que la probabilidad de que un punto al azar dentro de *C*, esté también dentro de *S* es:

```
Volumen Sólido/Volumen Cubo
```

Supongamos un cubo *C* definido por $x<=1$, $y<=1$, $z<=1$, y la esfera definida por $x^2 + y^2 + z^2 <= 1$.

Un octavo de la esfera, así definida, está dentro del cubo de lado 1. Por lo que si generamos un punto al azar, la probabilidad de que éste se encuentre también dentro del sector esférico es:

```
P = (Volumen esfera / 8) / 1 = Volumen esfera / 8
```

Por lo tanto, para saber el volumen de la esfera, basta calcular esa probabilidad. Para calcular esta probabilidad la función **main** invocará a una función denominada *DentroEsfera* que generará *TOTAL* puntos (x, y, z) y contará cuantos de puntos de esos están dentro del octavo de esfera. Por lo tanto, la probabilidad *P* y el volumen de la esfera vendrán dados por las expresiones:

```
P = dentro/TOTAL;
volumen = 8.0 * P;
```

El prototipo de la función *DentroEsfera* es el siguiente:

```
int DentroEsfera(const int);
```

Esta función tiene un parámetro que se corresponde con el número total de puntos a generar. Para generar los valores *x*, *y*, *z* (valores entre 0 y 1) la función *DentroEsfera* invocará a la función *rnd* descrita anteriormente.

El programa que se muestra a continuación, da solución a este problema.

```
/****** Números seudoaleatorios - Volumen de una esfera ******/
/* esfera.c
 */
#include <stdio.h>
double rnd(long *);
int DentroEsfera(const int);

void main()
{
  const int TOTAL = 1000; /* ensayos a realizar */
  double volumen;  /* volumen de la esfera */
  int dentro;      /* número de puntos dentro de la esfera */

  printf("Ensayos a realizar:  %d\n\n", TOTAL);
  dentro = DentroEsfera(TOTAL);
  /* Es necesario poner 8.0 para que el resultado sea real */
  volumen =  8.0 * dentro / TOTAL;
  printf("\n\n\nVolumen estimado = %g\n", volumen);
}

/* Puntos generados dentro de la esfera */
int DentroEsfera(const int total)
{
  unsigned int inicio;
  long random = inicio;
  int i, dentro = 0;
  double x, y, z;

  for (i = 1; i <= total; i++)
  {
    printf("Realizando cálculos... %d%c", i, '\r');
    x = rnd(&random); y = rnd(&random); z = rnd(&random);
    if (x*x + y*y + z*z <= 1)
      dentro = dentro + 1;
  }
  return(dentro);
}

// Generador de números seudoaleatorios
double rnd(long *prandom)
{
  *prandom = (25173 * *prandom + 13849) % 65536;
  return((double)*prandom / (double)65535);
}
```

4. Queremos generar un diccionario inverso. Estos diccionarios se caracterizan por presentar las palabras en orden alfabético ascendente pero observando las palabras desde su último carácter hasta el primero (por ejemplo: hola → aloh). En la tabla siguiente podemos ver un ejemplo de este tipo de ordenación:

DICCIONARIO NORMAL	DICCIONARIO INVERSO
adiós	hola
camión	rosa
geranio	camión
hola	geranio
rosa	tractor
tractor	adiós

Una aplicación de este curioso diccionario es buscar palabras que rimen. Para escribir un programa que genere un diccionario de este tipo, se pide:

1. Escribir la función *Comparar* cuyo prototipo es el siguiente:

```
int comparar(char *cad1, char *cad2);
```

Esta función comparará *cadena1* y *cadena2*, pero observando las palabras desde su último carácter hasta el primero. La función devolverá los siguientes resultados:

> 0 Si *cadena1* está alfabéticamente después que *cadena2*.
0 Si *cadena1* y *cadena2* son iguales.
< 0 Si *cadena1* está alfabéticamente antes que *cadena2*.

2. Escribir la función *OrdenarCadenas* con el prototipo que se indica a continuación, que ordene las palabras del array *palabras* en orden alfabético ascendente:

```
void OrdenarCadenas(char **pcad, int filas);
```

El parámetro *filas* indica el número total de palabras que tiene el array *pcad*. Para ordenar las palabras se empleará el *método de inserción* (para detalles sobre este método de ordenación, vea el capítulo de "Algoritmos").

3. Escribir un programa que lea palabras de la entrada estándar y las almacene en un array dinámico de cadenas de caracteres, y tras ordenarlas utilizando las funciones anteriores, las visualice en la salida estándar. Para ello escriba, además de las funciones anteriores, las siguientes funciones:

```
unsigned int LeerCadenas(char **pcad, unsigned nFilas);
void VisualizarCadenas(char **pcad, unsigned filas);
```

El programa completo se muestra a continuación.

```
/******************** Diccionario inverso *********************/
/* dicinver.c
 */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

unsigned int LeerCadenas(char **pcad, unsigned nFilas);
int Comparar(char *cad1, char *cad2);
void OrdenarCadenas(char **pcad, int filas);
void VisualizarCadenas(char **pcad, unsigned filas);

void main()
{
  char **pcad = NULL;
  unsigned int nFilas = 0;
  unsigned int Ok = 0, filas = 0, f = 0;

  do
  {
    printf("Número de filas del array:    ");
    Ok = scanf("%u", &nFilas);
    fflush(stdin);
  }
  while ( !Ok );

  /* Asignar memoria para el array de punteros */
  if ((pcad = (char **)malloc(nFilas * sizeof(char *))) == NULL)
  {
    printf("Insuficiente espacio de memoria\n");
    exit(-1);
  }

  /* Operaciones */
  filas = LeerCadenas(pcad, nFilas);
  OrdenarCadenas(pcad, filas);
  VisualizarCadenas(pcad, filas);

  /* Liberar la memoria asignada a cada una de las filas */
  for ( f = 0; f < filas; f++ )
    free(pcad[f]);
  /* Liberar la memoria asignada al array de punteros */
  free(pcad);
}

unsigned int LeerCadenas(char **pcad, unsigned nFilas)
{
  unsigned int f = 0, longitud = 0;
  char cadena[81];

  printf("Introducir cadenas de caracteres.\n");
  printf("Para finalizar introduzca una cadena nula.\n");
  printf("Esto es, pulse sólo <Entrar>.\n\n");
```

```
  while ((longitud = strlen(gets(cadena))) > 0 && f < nFilas)
  {
    /* Asignar espacio para una cadena de caracteres */
    if ((pcad[f] = (char *)malloc(longitud + 1)) == NULL)
    {
      printf("Insuficiente espacio de memoria disponible\n");
      exit(-1);   /* terminar el proceso */
    }
    /* copiar la cadena en el espacio de memoria asignado */
    strcpy(pcad[f], cadena);
    f++;
  }
  return(f);
}

int Comparar(char *cad1, char *cad2)
{
  int i, j;
  i = strlen(cad1) - 1;
  j = strlen(cad2) - 1;
  /* Comparar las cadenas de atrás hacia adelante */
  while( i > 0 && j > 0 )
  {
    if ( cad1[i] != cad2[j] )
      return (cad1[i] - cad2[j]);
    i--;
    j--;
  }
  return (cad1[i] - cad2[j]);
}

void OrdenarCadenas(char **pcad, int filas)
{
  char *aux;  /* puntero auxiliar */
  int i = 0, k = 0;

  /* Método de inserción */
  for ( i = 1; i < filas; i++ )
  {
    aux = pcad[i];
    k = i - 1;
    while ( (k >= 0) && (Comparar(aux, pcad[k]) < 0) )
    {
      pcad[k+1] = pcad[k];
      k--;
    }
    pcad[k+1] = aux;
  }
}

void VisualizarCadenas(char **pcad, unsigned filas)
{
  unsigned int f = 0;

  for ( f = 0; f < filas; f++ )
    printf("%s\n", pcad[f]);
}
```

EJERCICIOS PROPUESTOS

1. Realizar un programa que solicite una frase y a continuación la escriba modificada de forma que a la A le corresponda la K, a la B la L, ... , a la O la Y, a la P la Z, a la Q la A, ... y a la Z la J.

2. Realizar un programa que permita utilizar el terminal, como un diccionario Inglés-Español, esto es, al introducir una palabra en inglés, se escribirá la correspondiente palabra en español. El número de parejas de palabras es variable, pero limitado a un máximo de 100. La longitud máxima de cada palabra será de 40 caracteres. Por ejemplo, introducimos las siguientes parejas de letras:

   book libro
   green verde
   mouse ratón

   Una vez finalizada la introducción de la lista, pasamos al modo traducción, de forma que si tecleamos *green*, la respuesta ha de ser *verde*. Si la palabra no se encuentra se emitirá un mensaje que lo indique.

   El programa constará al menos de dos funciones:

   a) *CrearDiccionario*. Esta función creará el diccionario.
   b) *Traducir*. Esta función realizará la labor de traducción.

3. Un cuadrado mágico se compone de números enteros comprendidos entre *1* y n^2, donde n es un número impar que indica el orden de la matriz cuadrada que contiene los números que forman dicho cuadrado mágico. La matriz que forma este cuadrado mágico, cumple que la suma de los valores que componen cada fila, cada columna y cada diagonal es la misma. Por ejemplo, un cuadrado mágico de orden 3, puede ser el siguiente:

   Un valor de $n = 3$ implica una matriz de 3 por 3. Por lo tanto, los valores de la matriz estarán comprendidos entre 1 y 9 y dispuestos de la siguiente forma:

   8 1 6
   3 5 7
   4 9 2

   Realizar un programa que visualice un cuadrado mágico de orden impar n. El programa verificará que n es impar y está comprendido entre 3 y 15.

Una forma de construirlo consiste en situar el número 1 en el centro de la primera línea, el número siguiente en la casilla situada encima y a la derecha, y así sucesivamente. Es preciso tener en cuenta que el cuadrado se cierra sobre si mismo, esto es, la línea encima de la primera es la última y la columna a la derecha de la última es la primera. Siguiendo esta regla, cuando el número caiga en una casilla ocupada, se elige la casilla situada debajo del último número situado.

Se deberán realizar al menos las siguientes funciones:

a) *Es_impar*. Esta función verificará si *n* es impar.
b) *Cuadrado_mágico*. Esta función construirá el cuadrado mágico.

4. Se define la función *f(x)* como:

$$f(x)=\begin{cases}\int_0^x e^{-t^2}\,dt & si \quad x\geq 0\\ 0 & si \quad x<0\end{cases}$$

Se pide escribir un programa que nos permita evaluar F(x). Para ello se realizarán los siguientes pasos:

a) Escribir una función que nos permita evaluar el integrando e^{-t^2}. El prototipo de esta función será así:

```
double f(double x);
```

Para implementar esta función se aconseja utilizar la función *exp(x)*, que permite evaluar e^x, y que se encuentra declarada en el fichero de cabecera *math.h* así:

```
double exp(double x);
```

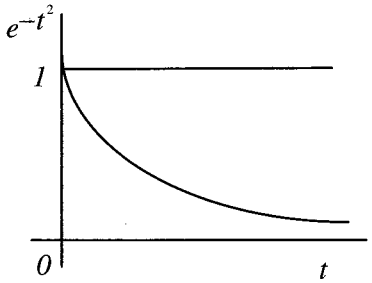
b) Escribir una función que genere números seudoaleatorios dentro de un determinado rango. El prototipo de esta función será así:

```
double rnd(double x);
```

La función *rnd*, cada vez que se invoque, generará un número seudoaleatorio comprendido entre *0* y *x*.

c) Escribir un programa que pida un valor de *x* y utilizando las funciones anteriores calcule *f(x)*.

- La función e^{-t^2} es positiva monótona decreciente. En *0* vale *1* y tiende a *0* cuando *t* crece. La primitiva de esta función, no se conoce.



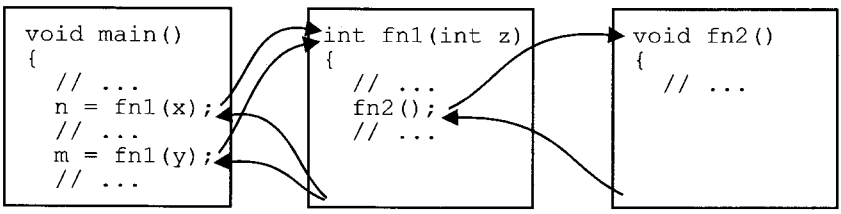
- La forma de solucionar este problema es análoga al cálculo del volumen de una esfera realizado en el apartado "ejercicios resueltos".

CAPÍTULO 8

FUNCIONES

Una función es una colección independiente de declaraciones y sentencias, generalmente enfocadas a realizar una tarea específica. Todo programa C consta al menos de una función, la función **main**. Además de ésta, puede haber otras funciones cuya finalidad es, fundamentalmente, descomponer el problema general en subproblemas más fáciles de resolver y de mantener. No obstante, independientemente del número de funciones y del orden en el que hayan sido escritas, la ejecución de un programa comienza siempre por la función **main**.

Cuando se llama a una función, el sistema pasa el control a la misma para su ejecución; y cuando finaliza, el control es devuelto de nuevo a la función que llamó, para continuar con la ejecución de la misma a partir de la sentencia que efectuó la llamada.



En el capítulo 3 ya fueron expuestos los conceptos de declaración, definición y llamada a una función; también se expuso cómo pasar argumentos por valor y por referencia a las funciones. En este capítulo, estudiaremos con detalle y veremos ejemplos de cómo pasar argumentos de distintos tipos de datos; por ejemplo arrays, estructuras o punteros. También estudiaremos cómo pasar argumentos a través de la línea de órdenes, funciones recursivas, punteros a funciones y diversas funciones de la biblioteca de C.

PASAR UN ARRAY A UNA FUNCIÓN

Ya hemos dicho en numerosas ocasiones que el nombre de un array es la dirección de comienzo de dicho array y también hemos visto que cuando pasamos un array a una función lo que se escribe como argumento en la llamada a esa función es el nombre del array. Esto significa que el argumento que se pasa es la dirección del array, por lo tanto el parámetro formal correspondiente en la definición de la función tiene que ser también un array, el cual, después de la llamada, queda inicializado con esa dirección. Por eso decimos que los arrays son siempre pasados por referencia, porque no se pasa una copia de todos sus elementos; esto es, tanto la función que invoca, como la función invocada trabajan sobre el mismo espacio de memoria, en otras palabras, sobre el mismo array. Por ejemplo, en el capítulo 6 implementamos funciones como estás:

```
// ...
void main() /* función principal */
{
  static tficha biblioteca[N]; /* array de estructuras */
  int n = 0; /* actual número de elementos del array */

  printf("INTRODUCIR DATOS\n\n");
  n = leer(biblioteca, N);

  printf("LISTADO DE LIBROS Y REVISTAS\n");
  escribir(biblioteca, n); /* listar todos los libros y revistas */
}

int leer(tficha bibli[], int NMAX)
{
  // ...
}

void escribir(tficha bibli[], int n)
{
  // ...
}
```

En este ejemplo, el primer parámetro de la función *leer* y de la función *escribir* es un array de una dimensión. Recuerde que cuando se declara un array como parámetro de una función, si es unidimensional no se requiere que se especifique su dimensión, lo que indica que no hay que hacer una reserva de memoria para una copia total del array; conclusión, lo que se pasa es la dirección del array. Si el array es multidimensional, entonces no se requiere que se especifique la primera dimensión, pero sí las restantes, como puede ver a continuación:

```
// ...
void main()
{
  static float a[FILAS][COLS], c[FILAS][COLS];
  int fila = 0, col = 0;
```

```
  /* Leer datos para el array a */
  for (fila = 0; fila < FILAS; fila++)
  {
    for (col = 0; col < COLS; col++)
    {
      printf("a[%d][%d] = ", fila, col);
      scanf("%f", &a[fila][col]);
    }
  }

  /* Copiar el array a en c */
  CopiarArray(c, a);
  // ...
}

void CopiarArray( float destino[][COLS], float origen[][COLS] )
{
  // ...
}
```

En este otro ejemplo, desarrollado también en el capítulo 6, los dos parámetros de la función *CopiarArray* son arrays de dos dimensiones. El tener que especificar la segunda dimensión hace que la función dependa de ese valor externo lo que supone declarar esa constante cuando utilicemos esta función en otros programas. Esto podría solucionarse con un fichero de cabecera en el que se incluyera tanto el prototipo de la función como la definición de la constante.

Cuando alguno de los parámetros de una función representa un array de una dimensión es indiferente declararlo como array o como puntero. En el siguiente ejemplo, puede ver el parámetro *bibli* declarado primeramente como un array y a continuación como un puntero; el comportamiento es el mismo.

```
int leer(tficha bibli[], int NMAX)
{
  // ...
}

int leer(tficha *bibli, int NMAX)
{
  // ...
}
```

El identificador de un array y un puntero no son lo mismo, pero en el caso de arrays de una dimensión podemos utilizar en lugar del nombre del array, un puntero que almacene la dirección de comienzo del array para acceder a los elementos del mismo. Como aplicación vamos a realizar un programa que:

1. Almacene en una array, el número de matrícula, apellidos, nombre y dirección de cada uno de los alumnos de un determinado curso. La estructura de cada uno de los elementos del array será del siguiente tipo:

```
typedef struct
{
  char matricula[10];
  char apellidos[30];
  char nombre[20];
  char direccion[30];
} ficha;
```

2. Busque la ficha correspondiente a un alumno, por su número de matrícula. Para esto escribiremos una función con el prototipo siguiente:

```
int leer(ficha *, const int);
```

El primer parámetro representa el puntero que recibirá la dirección de comienzo del array y el segundo parámetro es una constante entera que indicará el número máximo de elementos del array. La función *leer* devuelve como resultado el número de alumnos almacenados en el array.

3. Busque la ficha correspondiente a un alumno, por sus apellidos. Este trabajo lo realizaremos con una función cuyo prototipo es:

```
void buscar(ficha *, char *, int, int);
```

El primer parámetro representa un puntero que recibirá la dirección de comienzo del array, el segundo parámetro es un puntero a la cadena de caracteres que se desea buscar (*matrícula* o *nombre*), el tercer parámetro es el número de alumnos almacenados en el array por la función *leer* y el cuarto parámetro es un entero que especifica qué opción del menú se ha elegido (buscar por matrícula o buscar por apellidos).

4. La operación a realizar, es decir, leer los datos para los elementos del array, buscar por el número de matrícula, buscar por apellido y finalizar, será elegida de un menú visualizado por una función con el siguiente prototipo:

```
int menu(void);
```

La función *menú* devuelve como resultado la opción elegida.

El programa completo se muestra a continuación.

```
/********************** Programa Alumnos **********************/
/* arrays01.c
 */
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include <string.h>
```

```
#define N 100  /* número máximo de alumnos */

typedef struct
{
  char matricula[10];
  char apellidos[30];
  char nombre[20];
  char direccion[30];
} ficha;

int leer(ficha *, const int);
void buscar(ficha *, char *, int, int);
int menu(void);

void main()
{
  static ficha lista[N];
  char dato[30];  /* dato a buscar */
  int opcion;     /* opción elegida en el menú */
  int n = 0;      /* número de alumnos leídos */

  while (1)       /* bucle infinito. Se sale con break. */
  {
    opcion = menu();
    if (opcion != 4)
    {
      switch (opcion)
      {
        case 1:    /* entrada de los datos de los alumnos */
          n = leer(lista, N);
          break;
        case 2:    /* búsqueda por el número de matrícula */
          system("cls");
          printf("Número de matrícula "); gets(dato);
          buscar(lista, dato, n, opcion);
          break;
        case 3:    /* Búsqueda por los apellidos */
          system("cls");
          printf("Apellidos..... "); gets(dato);
          buscar(lista, dato, n, opcion);
          break;
      }
    }
    else
      break;
  }
}

/*****************************************************************
              Función para visualizar el menú
*****************************************************************/

int menu(void)
{
  int op;
```

```
  do
  {
    system("cls");
    printf("\n\t1. Entrada de datos de alumnos\n");
    printf("\n\t2. Búsqueda por nro. de matrícula\n");
    printf("\n\t3. Búsqueda por apellidos\n");
    printf("\n\t4. Fin\n");
    printf("\n\nTeclee la opción deseada  ");
    scanf("%d", &op);
    fflush(stdin);
  }
  while (op < 1 || op > 4);

  return (op);
}

/*****************************************************************
     Función para leer los datos correspondientes a un alumno
*****************************************************************/

int leer(ficha *lista, const int NMAX)
{
  int n = 0;
  char resp = 's';

  while (tolower(resp) == 's' && n < NMAX)
  {
    do
    {
      system("cls");
      printf("Alumno número %d\n\n", n+1);
      printf("Número de matrícula "); gets(lista[n].matricula);
      printf("Apellidos..... "); gets(lista[n].apellidos);
      printf("Nombre..... "); gets(lista[n].nombre);
      printf("Dirección..... "); gets(lista[n].direccion);
      printf("\n\n¿ Datos correctos ? s/n ");
      resp = getchar();
      fflush(stdin);
    }
    while (tolower(resp) != 's');
    n++;

    printf("\n\n¿ Más datos a introducir ? s/n ");
    resp = getchar();
    fflush(stdin);
  }
  return (n);
}

/*****************************************************************
        Función para buscar si existe o no un dato
*****************************************************************/

void buscar(ficha *lista, char *x, int alumnos, int opcion)
{
  const int NO = 0;
  const int SI = 1;
```

```
  int existe = NO, i = 0;
  char resp;

  switch (opcion)
  {
    case 2:       /* búsqueda por número de matrícula */
      while (!existe && i < alumnos)
        if (strcmp(lista[i++].matricula, x) == 0)
          existe = SI;
      break;
    case 3:       /* Búsqueda por apellidos */
      while (!existe && i < alumnos)
        if (strcmp(lista[i++].apellidos, x) == 0)
          existe = SI;
      break;
  }
  if (existe)
    printf("\n%s\n%s %s\n%s\n", lista[i-1].matricula,
                                lista[i-1].apellidos,
                                lista[i-1].nombre,
                                lista[i-1].direccion);
  else
    printf("\n%s no existe", x);

  printf("\n\nPulse <Entrar> para continuar ");
  resp = getchar();
  fflush(stdin);
}
```

Como ya hemos dicho anteriormente, el identificador de un array y un puntero no son lo mismo, lo que imposibilita en el caso de arrays de dos dimensiones utilizar en lugar del nombre del array, un puntero a un puntero que almacene la dirección de comienzo del array con el fin de acceder a los elementos del mismo.

Como aplicación vamos a realizar un programa que, partiendo de dos arrays de cadenas de caracteres clasificados en orden ascendente, construya y visualice un array también clasificado, fusión de los dos anteriores. Primero realizamos la versión con arrays convencionales.

Para ello, la función **main** proporcionará los dos arrays e invocará a una función cuyo prototipo es el siguiente:

```
int fusionar(char [][CPL], int,
             char [][CPL], int,
             char [][CPL], const int);
```

El primer parámetro de la función *fusionar* es uno de los arrays de partida y el segundo parámetro indica su número de elementos; el parámetro tercero es otro de los arrays de partida y el cuarto parámetro su número de elementos; el quinto parámetro es el array que almacenará los elementos de los dos anteriores y el sexto parámetro es su número máximo de elementos.

El proceso de fusión consiste en:

1. Tomar un elemento de cada uno de los arrays de partida.

2. Comparar los dos elementos (uno de cada array) y almacenar en el array resultado el menor.

3. Tomar otro elemento del array al que pertenecía el elemento almacenado en el array resultado, y volver al punto 2.

4. Cuando no queden más elementos en uno de los dos arrays de partida, se copian directamente en el array resultado, todos los elementos que queden en el otro array de partida.

El programa completo se muestra a continuación.

```
/************** Fusionar dos listas clasificadas **************/
/* arrays02.c
 * Versión arrays
 */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define NML 120 /* número máximo de líneas */
#define CPL 60  /* caracteres por línea */

int fusionar(char [][CPL], int,
             char [][CPL], int,
             char [][CPL], const int);
void Error(void);

void main()
{
  /* Inicializamos las listas a clasificar con el fin de no tener
   * que leer los datos y realizar así, una prueba rápida.
   */

  static char listaActual[][CPL] =
                { "Ana", "Carmen", "David",
                  "Francisco", "Javier", "Jesús",
                  "José", "Josefina", "Luís",
                  "María", "Patricia", "Sonia" };

  static char listaNueva[][CPL] =
                { "Agustín", "Belén",
                  "Daniel", "Fernando", "Manuel",
                  "Pedro", "Rosa", "Susana" };

  /* Calcular el número de elementos de los arrays anteriores */
  const int dimA = sizeof(listaActual)/sizeof(listaActual[0]);
  const int dimN = sizeof(listaNueva)/sizeof(listaNueva[0]);
```

```
  /* Definir el array resultante de fusionar los anteriores */
  static char listaFinal[NML][CPL];
  int ind, r;

  /* Fusionar listaActual y listaNueva y almacenar el resultado en
   * listaFinal. La función "fusionar" devuelve un 0 si no se
   * pudo realizar la fusión.
   */
  r = fusionar(listaActual, dimA, listaNueva, dimN, listaFinal, NML);

  /* Escribir el array resultante */
  if (r)
  {
    for (ind = 0; ind < dimA+dimN; ind++)
      printf("%s\n", listaFinal[ind]);
  }
  else
    Error();
}

/*****************************************************************
                        F U S I O N A R
*****************************************************************/
int fusionar(char listaA[][CPL], int dimA,
             char listaN[][CPL], int dimN,
             char listaF[][CPL], const int nml)
{
  int ind = 0, indA = 0, indN = 0, indF = 0;

  if (dimA + dimN == 0 || dimA + dimN > nml)
    return(0);

  while (indA < dimA && indN < dimN)
    if (strcmp(listaA[indA], listaN[indN]) < 0)
      strcpy(listaF[indF++], listaA[indA++]);
    else
      strcpy(listaF[indF++], listaN[indN++]);

  /* Los dos lazos siguientes son para prever el caso de que,
   * lógicamente una lista finalizará antes que la otra.
   */
  for (ind = indA; ind < dimA; ind++)
    strcpy(listaF[indF++], listaA[ind]);

  for (ind = indN; ind < dimN; ind++)
    strcpy(listaF[indF++], listaN[ind]);

  return(1);
}

void Error(void)
{
  puts("Longitud no válida de la lista resultante");
  exit(1);
}
```

Si ahora cambiamos en la declaración y en la definición de la función *fusionar* los parámetros que identifican los arrays de dos dimensiones por punteros a punteros ¿qué sucede?.

```
int fusionar(char **listaA, int dimA,
             char **listaN, int dimN,
             char **listaF, const int nml)
{
  // ...
}
```

Lo que sucede es que cuando el compilador analice la llamada

```
r = fusionar(listaActual, dimA, listaNueva, dimN, listaFinal, NML);
```

notificará que entre los argumentos 1, 3 y 5 de la llamada y los parámetros 1, 3 y 5 de la definición existen diferentes niveles de indirección porque los tipos de los parámetros actuales (argumentos en la llamada) y los tipos de los correspondientes parámetros formales son diferentes, lo que ratifica la afirmación de que el identificador de un array de dos dimensiones y un puntero a un puntero no son lo mismo (vea "punteros a punteros" el capítulo 7).

Un parámetro formal de una función que sea un puntero a un puntero si puede corresponderse con un parámetro actual que sea un identificador de un array de punteros, o con otro puntero a puntero (vea en el capítulo 7 "Arrays de punteros a cadenas de caracteres" y "Arrays dinámicos de cadenas de caracteres"). Como ejemplo, a continuación se muestra la versión con punteros del mismo programa anterior.

```
/************** Fusionar dos listas clasificadas **************/
/* arrays03.c
 * Versión con punteros
 */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define NML 120 /* número máximo de líneas */
#define CPL 60  /* caracteres por línea */

int fusionar(char **, int,
             char **, int,
             char **, const int);
void Error(void);

void main()
{
  /* Inicializamos las listas a clasificar con el fin de no tener
   * que leer los datos y realizar así, una prueba rápida.
   */
```

```
  static char *listaActual[] =
                  { "Ana", "Carmen", "David",
                    "Francisco", "Javier", "Jesús",
                    "José", "Josefina", "Luís",
                    "María", "Patricia", "Sonia" };

  static char *listaNueva[] =
                  { "Agustín", "Belén",
                    "Daniel", "Fernando", "Manuel",
                    "Pedro", "Rosa", "Susana" };

  /* Calcular el número de elementos de los arrays anteriores */
  const int dimA = sizeof(listaActual)/sizeof(listaActual[0]);
  const int dimN = sizeof(listaNueva)/sizeof(listaNueva[0]);
  /* Definir el array resultante de fusionar los anteriores */
  static char **listaFinal; // referencia el array resultante
  int ind, r;

  /* Asignar memoria para el array de punteros listaFinal */
  listaFinal = (char **)malloc((dimA+dimN)*sizeof(char *));
  if (listaFinal == NULL) Error();
  /* Inicializa el array de punteros. Esto evita problemas al
     liberar memoria, en el supuesto de un error por falta de
     memoria */
  for (ind = 0; ind < dimA+dimN; ind++)
    listaFinal[ind] = NULL;

  /* Fusionar listaActual y listaNueva y almacenar en resultado en
   * listaFinal. La función "fusionar" devuelve un 0 si no se
   * pudo realizar la fusión.
   */
  r = fusionar(listaActual, dimA, listaNueva, dimN, listaFinal, NML);

  /* Escribir el array resultante */
  if (r)
  {
    for (ind = 0; ind < dimA+dimN; ind++)
      printf("%s\n", listaFinal[ind]);
  }
  else
    Error();

  /* Liberar la memoria ocupada por el array listaFinal */
  for (ind = 0; ind < dimA+dimN; ind++)
    free(listaFinal[ind]);
  free(listaFinal);

}

/*****************************************************************
                          F U S I O N A R
*****************************************************************/
int fusionar(char **listaA, int dimA,
             char **listaN, int dimN,
             char **listaF, const int nml)
{
  int ind = 0, indA = 0, indN = 0, indF = 0;
```

```
  while (indA < dimA && indN < dimN)
    if (strcmp(listaA[indA], listaN[indN]) < 0)
    {
      listaF[indF] = (char *)malloc(strlen(listaA[indA]) + 1);
      if (listaF[indF] == NULL) return 0;
      strcpy(listaF[indF++], listaA[indA++]);
    }
    else
    {
      listaF[indF] = (char *)malloc(strlen(listaN[indN]) + 1);
      if (listaF[indF] == NULL) return 0;
      strcpy(listaF[indF++], listaN[indN++]);
    }

  /* Los dos lazos siguientes son para prever el caso de que,
   * lógicamente una lista finalizará antes que la otra.
   */
  for (ind = indA; ind < dimA; ind++)
  {
    listaF[indF] = (char *)malloc(strlen(listaA[indA]) + 1);
    if (listaF[indF] == NULL) return 0;
    strcpy(listaF[indF++], listaA[ind]);
  }

  for (ind = indN; ind < dimN; ind++)
  {
    listaF[indF] = (char *)malloc(strlen(listaN[indN]) + 1);
    if (listaF[indF] == NULL) return 0;
    strcpy(listaF[indF++], listaN[ind]);
  }

  return(1);
}

void Error(void)
{
  puts("Longitud no válida de la lista resultante");
  exit(1);
}
```

PASAR UN PUNTERO COMO ARGUMENTO A UNA FUNCIÓN

Un puntero, igual que otros tipos de variables, puede ser pasado por valor o por referencia. Por valor significa que el valor almacenado (una dirección) en el parámetro actual (argumento especificado en la llamada a la función) se copia en el parámetro formal correspondiente; si ahora modificamos el valor de este parámetro formal, el parámetro actual correspondiente no se ve afectado. Por referencia lo que se copia, no es la dirección almacenada en el parámetro actual, sino la dirección de donde se localiza ese parámetro actual, de tal forma que una referencia al contenido del parámetro formal es una referencia al parámetro actual.

Para clarificar lo expuesto vamos a realizar un programa que utilice una función *fnAsigMem* para asignar memoria para estructuras de tipo

```
typedef struct
{
  char c[20];
  int n;
} testruc;
```

La función *fnAsigMem* tiene como misión asignar memoria para una estructura de tipo *testruc* e inicializar los miembros de la estructura a unos valores determinados. Tiene un único parámetro que es un puntero a una estructura de tipo *testruc*, que referenciará el espacio de memoria asignado. Supongamos que dicha función la escribimos así:

```
void fnAsigMem(testruc *p)
{
  p = (testruc *)malloc(sizeof(testruc));
  strcpy(p->c, "cadena"), p->n = 1;
}
```

Para invocar a esta función, escribiremos

```
testruc *psta = NULL;
fnAsigMem(psta);
```

donde *psta* es el puntero que deseamos que apunte a la estructura de datos creada por la función *fnAsigMem*.

El programa completo se muestra a continuación.

```
/*************** Pasando punteros a funciones ***************/
/* punteros.c
 */
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
typedef struct
{
  char c[20];
  int n;
} testruc;

void fnAsigMem(testruc *p);

void main()
{
  testruc *psta = NULL;
  fnAsigMem(psta);
  printf("Estructura apuntada por psta: %-20s %5d\n", psta->c, psta->n);
}
```

```
void fnAsigMem(testruc *p)
{
  p = (testruc *)malloc(sizeof(testruc));
  strcpy(p->c, "cadena"), p->n = 1;
}
```

Cuando se ejecuta este programa, observamos que los resultados no se corresponden con los valores "*cadena*" y *1*, con los que hemos inicializado la estructura.

```
Estructura apuntada por psta: (null)                26956
```

¿Qué es lo que ha ocurrido? Observe la llamada a la función. El parámetro actual *psta*, independientemente de que sea un puntero, es pasado por valor; por lo tanto, el parámetro formal *p* recibe el valor **NULL** con el que ha sido inicializado *psta*. Después, la función *fnAsigMem* asigna a *p* memoria para una estructura; esto es, almacena en *p* la dirección del bloque de memoria asignado. Significa esto que el valor de *p* ha cambiado, pero no el de *psta*(vea "Pasando argumentos a las funciones" en el capítulo 3). Para poder modificar el valor de *psta* tendremos que pasar esta variable por referencia como se puede ver en la siguiente versión del mismo programa.

```
/*************** Pasando punteros a funciones ***************/
/* punteros.c
 */
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
typedef struct
{
  char c[20];
  int n;
} testruc;

void fnAsigMem(testruc **p);

void main()
{
  testruc *psta = NULL;
  fnAsigMem(&psta);
  printf("Estructura apuntada por psta: %-20s %5d\n", psta->c, psta->n);
}

void fnAsigMem(testruc **p)
{
  *p = (testruc *)malloc(sizeof(testruc));
  strcpy((*p)->c, "cadena"), (*p)->n = 1;
}
```

Ahora **p* es sinónimo de *psta*; es decir, si *p* es la dirección donde se localiza *psta*, el contenido de esta dirección es el valor de *psta*; en otras palabras, *p* apunta a *psta*. Según esto, cuando asignamos a **p* la dirección devuelta por **malloc**, en

realidad se la estamos asignando a *psta*. Por lo tanto, cuando ejecute el programa, el resultado será el esperado:

```
Estructura apuntada por psta: cadena                    1
```

Otro detalle que cabe resaltar, es la utilización de la función **strcpy** para copiar una cadena en otra. Seguramente algunos intentarían resolver esta cuestión utilizando, por ejemplo, la sentencia:

```
p->c = "cadena a";  /* error */
```

Cuando el compilador analiza esta sentencia visualiza un error porque *c* es el nombre de un array y como tal es una constante; como la sentencia, lo que trata es de asignar la dirección de "*cadena a*" a la constante *c*, se produce un error. En cambio, la función **strcpy** copia contenidos, no direcciones. Si *c* hubiera sido un puntero a una cadena de caracteres, sí se podría haber realizado esa asignación.

PASAR UNA ESTRUCTURA A UNA FUNCIÓN

Una estructura puede ser pasada a una función, igual que cualquier otra variable, por valor o por referencia. Cuando pasamos una estructura por valor, el parámetro actual que representa la estructura se copia en el correspondiente parámetro formal, produciéndose un duplicado de la estructura. Por eso, si alguno de los miembros del parámetro formal se modifica, estos cambios no afectan al parámetro actual correspondiente. Si pasamos la estructura por referencia, lo que recibe la función es el lugar de la memoria donde se localiza dicha estructura. Entonces, conociendo su dirección, sí es factible alterar su contenido.

Como ejemplo vamos a realizar un programa que utilizando una función *SumarComplejos*, permita visualizar la suma de dos complejos. La estructura de un complejo queda definida de acuerdo con la siguiente declaración de tipo:

```
typedef struct
{
  float real;
  float imag;
} tcomplejo;
```

La función *SumarComplejos* tiene el prototipo siguiente:

```
void SumarComplejos(tcomplejo c1, tcomplejo c2, tcomplejo *p);
```

Para invocar a esta función, escribiremos

```
SumarComplejos(ca, cb, &cc);
```

donde se observa que los complejos *ca* y *cb* son pasados por valor y el complejo *cc* por referencia. Los argumentos *ca* y *cb* son los complejos que queremos sumar y el complejo *cc* es donde almacenaremos el resultado, de ahí que lo pasemos por referencia.

El programa completo se muestra a continuación.

```
/***************** Operaciones con complejos ****************/
/* complejo.c
 */
#include <stdio.h>

typedef struct
{
  float real;
  float imag;
} tcomplejo;

void SumarComplejos(tcomplejo c1, tcomplejo c2, tcomplejo *p);

void main()
{
  tcomplejo ca, cb, cc;

  printf("\nIntroducir datos de la forma: x yj\n");
  printf("ca = ");
  scanf("%f %f", &ca.real,&ca.imag); fflush(stdin);
  printf("cb = ");
  scanf("%f %f", &cb.real,&cb.imag); fflush(stdin);

  SumarComplejos(ca, cb, &cc);
  printf("Resultado: %g%+gj\n", cc.real, cc.imag);
}

void SumarComplejos(tcomplejo c1, tcomplejo c2, tcomplejo *p)
{
  p->real = c1.real + c2.real;
  p->imag = c1.imag + c2.imag;
}
```

Está claro que el parámetro *p* es un puntero que apunta a la estructura *cc*. Puesto que *p* contiene la dirección de *cc*, la función *SumarComplejos* podría escribirse también así:

```
void SumarComplejos(tcomplejo c1, tcomplejo c2, tcomplejo *p)
{
  (*p).real = c1.real + c2.real;
  (*p).imag = c1.imag + c2.imag;
}
```

Cuando uno o más parámetros actuales se pasan por referencia a una función, es porque necesitamos disponer de ellos con su contenido actualizado por dicha función. Además, desde el punto de vista de optimización del tiempo de ejecu-

ción, el paso de parámetros por referencia es más rápido que el paso de los mismos por valor, puesto que no hay que hacer una copia de los datos. Ahora en el caso de que sólo sea un parámetro el pasado por referencia, como sucede en el ejemplo anterior, otra solución puede ser utilizar el valor retornado por la función para realizar la misma operación de actualización. Por ejemplo, siguiendo este criterio la función *SumarComplejos* podría escribirse así:

```
tcomplejo SumarComplejos(tcomplejo c1, tcomplejo c2)
{
  tcomplejo c;
  c.real = c1.real + c2.real;
  c.imag = c1.imag + c2.imag;
  return c;
}
```

Observe que ahora la función devuelve una estructura de tipo *tcomplejo*, por lo tanto, ahora la llamada a la función será así:

```
cc = SumarComplejos(ca, cb);
```

UNA FUNCIÓN QUE RETORNA UN PUNTERO

Cuando una función retorna un puntero a un objeto, el objeto debe persistir después de finalizar la función de lo contrario estaremos cometiendo un error.

Por ejemplo, modifiquemos el programa anterior para que la función *SumarComplejos* retorne un puntero a una estructura de tipo *tcomplejo*.

```
/*************** Operaciones con complejos ***************/
/* complej2.c
 */
#include <stdio.h>
#include <stdlib.h>

typedef struct
{
  float real;
  float imag;
} tcomplejo;

tcomplejo *SumarComplejos(tcomplejo c1, tcomplejo c2);

void main()
{
  tcomplejo ca, cb, *pcr;

  printf("\nIntroducir datos de la forma: x yj\n");
  printf("ca = ");
  scanf("%f %f", &ca.real,&ca.imag); fflush(stdin);
  printf("cb = ");
  scanf("%f %f", &cb.real,&cb.imag); fflush(stdin);
```

```
  pcr = SumarComplejos(ca, cb);
  printf("Resultado: %g%+gj\n", pcr->real, pcr->imag);
}

tcomplejo *SumarComplejos(tcomplejo c1, tcomplejo c2)
{
  tcomplejo cx;
  cx.real = c1.real + c2.real;
  cx.imag = c1.imag + c2.imag;

  return &cx;
}
```

Si ejecuta este programa observará que los resultados no son los esperados ¿Qué sucede? La función *SumarComplejo*, para calcular la suma utiliza un complejo local *cx* del cual retorna su dirección. Pero cuando la función finalice el complejo *cx* se destruirá automáticamente, con lo que el puntero *pcr* que apunta al resultado, estará apuntando a un objeto inexistente, razón por la cual los resultados son inesperados. Como vemos estamos en el caso de un objeto que no persiste a lo largo de la ejecución del programa.

Una solución al problema planteado es hacer que la función *SumarComplejos* cree un objeto que persista a lo largo de la ejecución del programa y esto se consigue asignando memoria dinámicamente para el objeto. Por ejemplo,

```
tcomplejo *SumarComplejos(tcomplejo c1, tcomplejo c2)
{
  tcomplejo *pcx;

  /* Asignar memoria para el complejo suma */
  pcx = (tcomplejo *)malloc(sizeof(tcomplejo));
  if ( pcx == NULL )
  {
    printf("Memoria insuficiente\n");
    exit(-1);
  }
  pcx->real = c1.real + c2.real;
  pcx->imag = c1.imag + c2.imag;

  return pcx;
}
```

Ahora la función *SumarComplejo* devuelve la dirección de un objeto asignado dinámicamente. Para destruir este objeto añadiremos al final de la función **main** la sentencia:

```
free(pcr);
```

ARGUMENTOS EN LA LÍNEA DE ÓRDENES

Muchas veces, cuando invocamos a un programa desde el sistema operativo, necesitamos escribir uno o más argumentos a continuación del nombre del programa, separados por blancos. Por ejemplo, piense en la orden *ls -l* del sistema operativo UNIX o en la orden *dir /p* del sistema operativo MS-DOS. Tanto *ls* como *dir* son programas; *-l* y */p* son opciones o argumentos en la línea de órdenes que pasamos al programa para que tenga un comportamiento diferente al que tiene por defecto; es decir, cuando no se pasan argumentos.

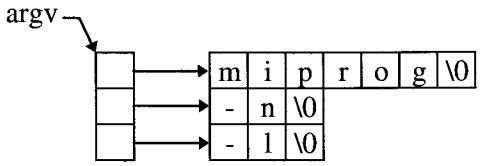
De la misma forma, nosotros podemos construir programas que admitan argumentos a través de la línea de órdenes ¿Qué función recibirá esos argumentos? La función **main**, ya que es por esta función por donde empieza a ejecutarse un programa C. Quiere esto decir que la función **main** tiene que tener parámetros formales donde se almacenen los argumentos pasados, igual que ocurre con cualquier otra función C. Así, el prototipo de la función **main** en general es de la forma siguiente:

```
int main(int argc, char *argv[]);
```

El argumento *argc* es un entero que indica el número de argumentos pasados a través de la línea de órdenes, incluido el nombre del programa. El *argv* es un array de punteros a cadenas de caracteres. La función **main** retorna por defecto un **int**. Cada elemento del array *argv* apunta a un argumento, de manera que *argv[0]* contiene el nombre del programa, *argv[1]* el primer argumento de la línea de órdenes, *argv[2]* el segundo argumento, etc. Por ejemplo, supongamos que tenemos un programa C denominado *miprog* que acepta como argumentos *-n* y *-l*. Para invocar a este programa podríamos escribir en la línea de órdenes del sistema operativo,

```
miprog -n -l
```

Esto hace que automáticamente *argc* tome el valor tres (nombre del programa más dos argumentos) y que el array de punteros a cadenas de caracteres sea:



Los argumentos de la función **main** se han denominado *argc* y *argv* por convenio. Esto quiere decir que podríamos utilizar otros identificadores para nombrarlos.

Para clarificar lo expuesto vamos a realizar un programa que simplemente visualice los valores del entero *argc* y del array *argv*. Esto nos dará una idea de cómo acceder desde un programa a los argumentos pasados desde la línea de órdenes. Supongamos que el programa se denomina *args01* y que admite los argumentos *-n*, *-k* y *-l*. Esto quiere decir que podremos especificar de cero a tres argumentos. Por lo tanto, para invocar a este programa desde la línea de órdenes escribiremos una línea similar a la siguiente:

```
args01 -n -l↵
```

Aquí sólo se han especificado dos parámetros.

El código del programa se muestra a continuación.

```
/*************** Argumentos en línea de órdenes ***************/
/* args01.c
 */
#include <stdio.h>

void main(int argc, char *argv[])
{
  int i;

  /* Verificar si se han pasado argumentos */
  if (argc < 2)
    printf("No hay argumentos para %s\n", *argv);
  else
  {
    printf("Nombre del programa: %s\n\n", argv[0]);
    printf("Argumentos pasados: %d\n\n", argc-1);
    for (i = 1; i < argc; i++)
      printf("  argumento %d: %s\n", i, argv[i]);
  }
}
```

Al ejecutar este programa, invocándolo como se ha indicado anteriormente, se obtendrá el siguiente resultado:

```
Nombre del programa: args01.exe

Argumentos pasados: 2

  argumento 1: -n
  argumento 2: -l
```

Por ser *argv* un array de punteros a cadenas, existen varias formas de acceder al contenido de sus elementos. Estas son:

```
argv[i], *(argv+i) o *argv++
```

REDIRECCIÓN DE LA ENTRADA Y DE LA SALIDA

Redireccionar la entrada significa que los datos pueden ser obtenidos de un medio diferente a la entrada estándar; por ejemplo, de un fichero en el disco. Si suponemos que tenemos un programa denominado *redir.c* que admite datos de la entrada estándar, la orden siguiente ejecutaría el programa *redir* y obtendría los datos de entrada de un fichero en el disco denominado *fdatos.ent.*

```
redir < fdatose.ent
```

Igualmente, redireccionar la salida significa enviar los resultados que produce un programa a un dispositivo diferente a la salida estándar; por ejemplo, a un fichero en disco. Tomando como ejemplo el programa *redir.c*, la orden siguiente ejecutaría el programa *redir* y escribiría los resultados en un fichero *fdatos.sal.*

```
redir > fdatos.sal
```

Observe que el programa se ejecuta desde la línea de órdenes, que para redireccionar la entrada se utiliza el símbolo "<" y que para redireccionar la salida se utiliza el ">". También es posible redireccionar la entrada y la salida simultáneamente. Por ejemplo:

```
redir < fdatos.ent > fdatos.sal
```

Como aplicación de lo expuesto, vamos a realizar un programa que lea un conjunto de números y los escriba de una forma o de otra, en función de los argumentos pasados a través de la línea de órdenes. Esto es, si el programa se llama *redir.c*, la orden

```
redir
```

visualizará el conjunto de números sin más; pero la orden

```
redir -l
```

visualizará el conjunto de números escribiendo a continuación de cada uno de ellos un mensaje que indique si es par o impar. Por ejemplo,

```
24          es par
345         es impar
7           es impar
...         ...
```

El código de este programa se muestra a continuación.

```
/************** Argumentos en la línea de órdenes **************/
/* args02.c
 */
#include <stdio.h>

void main( int argc, char *argv[] )
{
  int n;

  while (scanf("%d", &n) != EOF)
  {
    printf("%6d", n);
    if (argc > 1 && argv[1][0] == '-' && argv[1][1] == 'l')
      printf((n%2) ? " es impar" : " es par");
    printf("\n");
  }
}
```

Como hemos dicho, para ejecutar este programa puede escribir en la línea de órdenes:

```
redir
redir -l
```

En ambos casos tendrá que introducir datos por el teclado, por ejemplo, así:

```
24 345 7 41 89 -72 5
```

y finalizar la entrada con la marca de fin de fichero. Si no se introduce el argumento *-l* simplemente se visualizarán los valores tecleados. Si se introduce el argumento *-l* se visualizarán los valores tecleados seguidos cada uno de ellos de la cadena " es par" o " es impar", dependiendo de que el número sea par o impar.

También, podemos editar un fichero *fdatos.ent* que contenga, por ejemplo,

```
24 345 7 41 89 -72 5
```

e invocar al programa *redir* de alguna de las formas siguientes:

```
redir < fdatos.ent
redir > fdatos.sal
redir < fdatos.ent > fdatos.sal
```

La primera orden leería los datos del fichero *fdatos.ent* y visualizaría los resultados por la pantalla, la segunda orden leería los datos del teclado y escribiría los resultados en el fichero *fdatos.sal* y la tercera orden leería los datos del fichero *fdatos.ent* y escribiría los resultados en el fichero *fdatos.sal*. Sepa que cuando se edita un fichero y se almacena, el sistema le añade automáticamente al final del mismo, la marca de fin de fichero.

FUNCIONES RECURSIVAS

Se dice que una función es recursiva, si se llama a sí misma. El compilador C permite cualquier número de llamadas recursivas a una función. Cada vez que la función es llamada, los parámetros formales y las variables **auto** y **register** son inicializadas. Notar que las variables **static** solamente son inicializadas una vez, en la primera llamada.

¿Cuándo es eficaz escribir una función recursiva? La respuesta es sencilla, cuando el proceso a programar sea por definición recursivo. Por ejemplo, el cálculo del factorial de un número es por definición un proceso recursivo,

```
n! = n(n-1)!
```

por lo tanto, la forma idónea de programar este problema es implementando una función recursiva. Como ejemplo, a continuación se muestra un programa que visualiza el factorial de un número. Para ello, se ha escrito una función *factorial* que recibe como parámetro un número entero positivo y devuelve como resultado el factorial de dicho número.

```
/************* Cálculo del factorial de un número *************/
/* facto.c
 */
#include <stdio.h>

unsigned long factorial(int n);

void main()
{
  int numero;
  unsigned long fac;

  do
  {
    printf("¿Número?  ");
    scanf("%d", &numero);
  }
  while (numero < 0 || numero > 12);
  fac = factorial(numero);
  printf("\nEl factorial de %2d es %ld\n", numero, fac);
}

unsigned long factorial(int n)
{
  if (n == 0)
    return 1;
  else
    return n*factorial(n-1);
}
```

En la tabla siguiente se ve el proceso seguido por la función, durante su ejecución para *n = 4*.

Nivel de recursión	*Proceso de ida*	*Proceso de vuelta*
0	factorial(4)	24
1	4 * factorial(3)	4 * 6
2	3 * factorial(2)	3 * 2
3	2 * factorial(1)	2 * 1
4	1 * factorial(0)	1 * 1
	factorial(0)	1

Cada llamada a la función *factorial* aumenta en una unidad el nivel de recursión. Cuando se llega a *n = 0*, se obtiene como resultado el valor 1 y se inicia la vuelta hacia el punto de partida, reduciendo el nivel de recursión en una unidad cada vez.

Los algoritmos recursivos son particularmente apropiados cuando el problema a resolver o los datos a tratar se definen en forma recursiva. Sin embargo, el uso de la recursión debe evitarse cuando haya una solución obvia por iteración.

En aplicaciones prácticas es imperativo demostrar que el nivel máximo de recursión es, no sólo finito, sino realmente pequeño. La razón es que, por cada ejecución recursiva de la función, se necesita cierta cantidad de memoria para almacenar las variables locales y el estado en curso del proceso de cálculo con el fin de recuperar dichos datos cuando se acabe una ejecución y haya que reanudar la anterior. En el capítulo de "Algoritmos" volveremos a tratar la recursión.

Ajustando el tamaño del STACK

Como hemos indicado anteriormente, el llamar a una función recursivamente, consume mucho espacio de pila debido a que por cada llamada las variables que intervienen en la función son salvadas en la pila para posteriormente poder iniciar la vuelta. Esto indica que puede ser necesario ajustar el tamaño de la pila. En un sistema operativo como MS-DOS el tamaño de la pila se ajusta mediante la opción */F* de la orden *CL*. Por ejemplo,

```
CL /F 2000 prog.c
```

En este ejemplo se fija el tamaño del stack a 8K (2000H bytes).

PUNTEROS A FUNCIONES

Igual que sucedía con los arrays, el nombre de una función representa la dirección donde se localiza esa función; quiere esto decir que como tal dirección, puede pasarse como argumento a una función, almacenarla en un elemento de un array, etc. La sintaxis para declarar un puntero a una función es así:

```
tipo(*p_identif)();
```

donde *tipo* es el tipo del valor devuelto por la función y *p_identif* es el nombre de una variable de tipo puntero. Esta variable recibirá la dirección de comienzo de una función, dada por el propio nombre de la función. Por ejemplo,

```
int (*pfn)(int);
```

indica que *pfn* es un puntero a una función que devuelve un entero. Si *pfn* es un puntero a una función, **pfn* es la función y *(*pfn)(argumentos)* o simplemente *pfn(argumentos)* es la llamada. Observe la declaración de *pfn*; los paréntesis que envuelven al identificador son fundamentales. Si no los ponemos, el significado cambia completamente. Por ejemplo,

```
int *pfn(int);
```

indica que *pfn* es una función que devuelve un puntero a entero.

La forma de utilizar un puntero a una función es como se indica en el ejemplo que se expone a continuación. En este ejemplo, la función **main** define un puntero *pfn* a una función; después asigna a *pfn* la dirección de la función *cuadrado* y utiliza ese puntero para llamar a dicha función. A continuación asigna a *pfn* la dirección de la función *pot* y utiliza de nuevo ese puntero para llamar a dicha función.

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

int cuadrado(int);
int pot(int, int);

void main()
{
  int (*pfn)();  /* pfn es un puntero a una función */
  int x = 5, y = 3, r = 0;

  pfn = cuadrado;     /* pfn apunta a la función cuadrado */

  r = (*pfn)(x);      /* llamada a la función cuadrado */
  printf("%d\n", r);
```

```
  pfn = pot;          /* pfn apunta a la función pot */

  printf("%d\n", (*pfn)(x, y));
}

int cuadrado(int a)  /* función cuadrado */
{
  return (a*a);
}

int pot(int x, int y)  /* función potenciación */
{
  return (int)(exp(y * log(x)));
}
```

Observe que podemos utilizar el puntero *pfn* con distintas funciones independiente de su número y tipo de parámetros. Lo que no es posible es utilizar el puntero para invocar a una función con un tipo del resultado diferente al especificado al declarar dicho puntero. Por ejemplo, si declaramos la función *pot* así,

```
double pot(double, double);
```

la asignación *pfn = pot* daría lugar a un error porque el puntero representa a una función que devuelve un **int** y *pot* es una función que devuelve un **double**.

Así mismo, una llamada explícita como *(*pfn)(x)* puede realizarse también de una forma implícita así:

```
pfn(x);
```

El siguiente ejemplo clarifica el trabajo con punteros a funciones. El programa que se muestra a continuación, busca y escribe el valor menor de una lista de datos numérica o de una lista de datos alfanumérica. Si en la línea de órdenes se especifica un argumento "*–n*", se interpreta la lista como numérica; en los demás casos se interpreta como alfanumérica. Por ejemplo, si el programa se llama *puntsfns.c* la orden

```
puntsfns -n
```

invoca al programa para trabajar con una lista de datos numérica.

El programa consta fundamentalmente de una función *fmenor* que busca en una lista de datos, el menor, independientemente del tipo de comparación (numérica o alfanumérica). Para ello, la función *fmenor* recibe como parámetro una función; la adecuada para comparar los datos según sean estos numéricos o alfanuméricos. Para comparar datos numéricos, se utiliza la función *compnu* y para comparar alfanuméricos, se utiliza la función *compal*.

```
char *fmenor(int c, char *pcadena[], char *(*comparar)())
{
  /* Buscar el dato menor de una lista */
  char *menor;
  menor = *pcadena;  /* menor = primer dato */
  while ( --c > 0)
    /* comparar menor con el siguiente dato */
    menor = (*comparar)(menor, *++pcadena);
  return (menor);
}
```

El argumento *c* es el número de datos a comparar, *pcadena* es el array de punteros a esos datos y *comparar* es un puntero a una función que hace referencia a la función utilizada para comparar (*compnu* o *compal*). La llamada a la función *fmenor* es de la forma siguiente:

```
if (argc > 1 && argv[1][0] == '-' && argv[1][1] == 'n')
  p = fmenor(c, pcadena, compnu);
else
  p = fmenor(c, pcadena, compal);
```

El programa completo se muestra a continuación.

```
/******************** Punteros a funciones ********************/
/* puntsfns.c
 */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#define FMAX 100          /* número máximo de filas */

char *compnu(char *px, char *py);
char *compal(char *px, char *py);
char *fmenor(int n, char **pa, char *(*pfn)());

void main(int argc, char *argv[])
{
  char *pcadena[FMAX]; /* array de punteros a los datos */
  char dato[81];       /* dato */
  char *p;
  int c = 0, longitud;
  /* Leer la lista de datos numéricos o alfanuméricos */
  printf("Introducir datos y finalizar con <Entrar>\n\n");
  while (c < FMAX)
  {
    printf("Dato %d: ", c + 1);
    if ((longitud = strlen(gets(dato))) == 0)
      break;
    else
    {
      pcadena[c] = (char *)malloc(longitud+1);
      if (pcadena[c] == NULL)
      {
```

```
            printf("Insuficiente espacio de memoria\n");
            exit(-1);
          }
          else
            strcpy(pcadena[c++], dato);
        }
      }
      /* argv[1] != "-n" -> búsqueda en una lista alfanumérica,
       * argv[1]  = "-n" -> búsqueda en una lista numérica
       */
      if (argc > 1 && argv[1][0] == '-' && argv[1][1] == 'n')
        p = fmenor(c, pcadena, compnu);
      else
        p = fmenor(c, pcadena, compal);
      printf("\n\nEl elemento menor de la lista es: %s\n", p);
}

char *fmenor(int c, char *pcadena[], char *(*comparar)())
{
      /* Buscar el dato menor de una lista */
      char *menor;

      menor = *pcadena;  /* menor = primer dato */
      while ( --c > 0)
        /* comparar menor con el siguiente dato */
        menor = (*comparar)(menor, *++pcadena);

      return (menor);
}

char *compnu(char *px, char *py)
{
      /* Comparar dos datos numéricamente */
      if (atof(px) > atof(py))
        return (py);
      else
        return (px);
}

char *compal(char *px, char *py)
{
      /* Comparar dos datos alfanuméricamente */
      if (strcmp(px, py) > 0)
        return (py);
      else
         return (px);
}
```

FUNCIONES PREDEFINIDAS EN C

En este capítulo hemos estudiado cómo el usuario puede definir sus propias funciones. No obstante C dispone en sus librerías de otras muchas funciones; algunas de ellas ya las hemos visto, como las funciones para entrada/salida, las funciones

para manipular cadenas de caracteres etc., y otras las iremos viendo en éste y en sucesivos capítulos.

Funciones matemáticas

Las declaraciones para las funciones matemáticas que a continuación se describen, están en el fichero a incluir *math.h*. Quiere esto decir, que cuando se utilice una función matemática en un programa, debe especificarse la directriz:

#include <math.h>

Los argumentos para estas funciones son de tipo **double** y el resultado devuelto es también de tipo **double**. Por ello, en muchos casos utilizaremos una conversión explícita de tipo (conversión *cast*) para convertir explícitamente los argumentos al tipo deseado. Por ejemplo, suponiendo que *valor* es un **int**,

```
a = acos((double)valor);
```

calcula el arco coseno de *valor*, pero el argumento pasado a la función *acos* es *valor* convertido explícitamente tipo **double**.

Las funciones matemáticas las podemos clasificar en las siguientes categorías:

- Funciones trigonométricas.
- Funciones hiperbólicas.
- Funciones exponencial y logarítmica.
- Funciones varias.

acos

La función **acos** da como resultado el arco, en el rango 0 a π, cuyo coseno es *x*. El valor de *x* debe estar entre −1 y 1; de lo contrario se obtiene un error (argumento fuera del dominio de la función).

```
#include <math.h>
double acos( double x );
Compatibilidad: ANSI, UNIX y MS-DOS
```

asin

La función **asin** da como resultado el arco, en el rango -π/2 a π/2, cuyo seno es *x*. El valor de *x* debe estar entre −1 y 1; si no se obtiene un error (argumento fuera del dominio de la función).

```
#include <math.h>
double asin( double x );
Compatibilidad: ANSI, UNIX y MS-DOS
```

atan

La función **atan** da como resultado el arco, en el rango -π/2 a π/2, cuya tangente es *x*.

```
#include <math.h>
double atan( double x );
Compatibilidad: ANSI, UNIX y MS-DOS
```

atan2

La función **atan2** da como resultado el arco, en el rango –π a π, cuya tangente es *y/x*. Si ambos argumentos son 0, se obtiene un error (argumento fuera del dominio de la función).

```
#include <math.h>
double atan2( double x );
Compatibilidad: ANSI, UNIX y MS-DOS
```

```
#include <math.h>
void main()
{
  double valor = -1;
  do
  {
    printf("%lf  %lf\n", acos(valor), atan2(valor, 1.0));
    valor += 0.1;
  }
  while (valor <= 1.0);
}
```

cos

La función **cos** da como resultado el coseno de *x* (*x* en radianes).

```
#include <math.h>
double cos( double x );
Compatibilidad: ANSI, UNIX y MS-DOS
```

sin

La función **sin** da como resultado el seno de *x* (*x* en radianes).

```
#include <math.h>
```

```
double sin( double x );
Compatibilidad: ANSI, UNIX y MS-DOS
```

tan

La función **tan** da como resultado la tangente de x (x en radianes).

```
#include <math.h>
double tan( double x );
Compatibilidad: ANSI, UNIX y MS-DOS
```

cosh

La función **cosh** da como resultado el coseno hiperbólico de x (x en radianes).

```
#include <math.h>
double cosh( double x );
Compatibilidad: ANSI, UNIX y MS-DOS
```

sinh

La función **sinh** da como resultado el seno hiperbólico de x (x en radianes).

```
#include <math.h>
double sinh( double x );
Compatibilidad: ANSI, UNIX y MS-DOS
```

tanh

La función **tanh** da como resultado la tangente hiperbólica de x (x en radianes).

```
#include <math.h>
double tanh( double x );
Compatibilidad: ANSI, UNIX y MS-DOS
```

exp

La función **exp** da como resultado el valor de e^x (e = 2.718282).

```
#include <math.h>
double exp( double x );
Compatibilidad: ANSI, UNIX y MS-DOS
```

log

La función **log** da como resultado el logaritmo natural de x.

```
#include <math.h>
double log( double x );
Compatibilidad: ANSI, UNIX y MS-DOS
```

log10

La función **log10** da como resultado el logaritmo en base 10 de *x*.

```
#include <math.h>
double log10( double x );
Compatibilidad: ANSI, UNIX y MS-DOS
```

ceil

La función **ceil** da como resultado un valor **double**, que representa el entero más pequeño que es mayor o igual que *x*.

```
#include <math.h>
double ceil( double x );
Compatibilidad: ANSI, UNIX y MS-DOS

double x = 2.8, y = -2.8;
printf("%g  %g\n", ceil(x), ceil(y));

3  -2
```

fabs

La función **fabs** da como resultado el valor absoluto de *x*. El argumento *x*, es un valor real en doble precisión. Igualmente, **abs** y **labs** dan el valor absoluto de un **int** y un **long** respectivamente.

```
#include <math.h>
double fabs( double x );
Compatibilidad: ANSI, UNIX y MS-DOS
```

floor

La función **floor** da como resultado un valor **double**, que representa el entero más grande que es menor o igual que *x*.

```
#include <math.h>
double floor( double x );
Compatibilidad: ANSI, UNIX y MS-DOS

double x = 2.8, y = -2.8;
printf("%g  %g\n", floor(x), floor(y));
```

```
2  -3
```

pow

La función **pow** da como resultado x^y. Si x es 0 e y negativo o si x e y son 0 o si x es negativo e y no es entero, se obtiene un error (argumento fuera del dominio de la función). Si x^y da un resultado superior al valor límite para el tipo **double**, el resultado es este valor límite (1.79769e+308).

```
#include <math.h>
double pow( double x );
Compatibilidad: ANSI, UNIX y MS-DOS

double x = 2.8, y = -2.8;
printf("%g\n", pow(x, y));

0.0559703
```

sqrt

La función **sqrt** da como resultado la raíz cuadrada de x. Si x es negativo, ocurre un error (argumento fuera del dominio de la función).

```
#include <math.h>
double sqrt( double x );
Compatibilidad: ANSI, UNIX y MS-DOS
```

matherr

Las funciones de la librería matemática llaman a la función **matherr** cuando ocurre un error.

```
#include <math.h>
int matherr( struct _exception *except );
Compatibilidad: UNIX y MS-DOS

struct exception          Información:
{
  int type;               tipo de error
  char *name;             función donde se origina el error
  double arg1, arg2;      valores que causan el error
  double retval;          valor devuelto
} *excep;
```

El tipo de error es uno de los valores siguientes:

Valor	*Significado*
DOMAIN	El argumento está fuera del dominio de la función.
OVERFLOW	El resultado es demasiado grande para ser representado.
PLOSS	Pérdida PARCIAL de significación.
SING	Un argumento de la función ha tomado un valor no permitido.
TLOSS	Pérdida total de significación.
UNDERFLOW	El resultado es demasiado pequeño para ser representado.

En el ejemplo siguiente se muestra la forma de utilizar la función **matherr**, la cual será llamada automáticamente si alguna función matemática da lugar a un error.

```
/********************** Función matherr **********************/
/* matherr.c
 */
#include <stdio.h>
#include <math.h>
#include <string.h>

void main(void)
{
   /* Realizar algunas operaciones matemáticas que producen
    * errores. La función matherr se ejecuta cuando una función
    * matemática produce un error.
    */
   printf( "log( -3.0 ) = %g\n", log( -3.0 ) );
   printf( "log10( -4.0 ) = %g\n", log10( -4.0 ) );
   printf( "log( 0.0 ) = %g\n", log( 0.0 ) );
}

/* Manipular errores producidos por pasar argumentos negativos
 * a las funciones logaritmo. Cuando esto ocurre la función
 * matherr calcula el logaritmo del valor absoluto y suprime
 * el mensaje usual. Lo normal es que matherr devuelva un 0
 * para indicar un error, y un valor distinto de cero para
 * indicar que se ha manipulado correctamente el error.
 */
int matherr(struct exception *excep)
{
   /* Manipular el error para log y log10 */
   if ( excep->type == DOMAIN )
   {
      if ( strcmp( excep->name, "log" ) == 0 )
      {
         excep->retval = log( -(excep->arg1) );
         printf( "%s utiliza el valor absoluto\n", excep->name );
         return 1;
      }
      else if ( strcmp( excep->name, "log10" ) == 0 )
      {
         excep->retval = log10( -(excep->arg1) );
         printf( "%s utiliza el valor absoluto\n", excep->name );
```

```
            return 1;
        }
    }
    else
    {
        printf( "Acción por defecto " );
        return 0;
    }
}
```

Funciones varias

La biblioteca de C proporciona también funciones para generar números aleatorios, para manipular la fecha y la hora, etc. A continuación vemos algunas de ellas.

rand

La función **rand** da como resultado un número seudoaleatorio entero, entre 0 y el valor máximo para un **int**.

```
#include <stdlib.h>
int rand(void);
Compatibilidad: ANSI, UNIX y MS-DOS
```

srand

La función **srand** fija el punto de comienzo para generar números seudoaleatorios; en otras palabras, inicializa el generador de números seudoaleatorios en función del valor de su argumento. Cuando esta función no se utiliza, el valor del primer número seudoaleatorio generado siempre es el mismo para cada ejecución (corresponde a un argumento de valor 1).

```
#include <stdlib.h>
void srand(unsigned int arg);
Compatibilidad: ANSI, UNIX y MS-DOS
```

clock

La función **clock** indica el tiempo empleado por el procesador en el proceso en curso. Este tiempo expresado en segundos, es el resultado de dividir el valor devuelto por **clock** entre la constante *CLOCKS_PER_SEC*. Si no es posible obtener este tiempo, la función **clock** devuelve el valor (**clock_t**)–1. El tipo **clock_t** está declarado así:

```
typedef long clock_t;
```

En MS-DOS esta función devuelve el tiempo transcurrido desde que el proceso arrancó. Este tiempo puede no ser igual al tiempo de procesador, realmente utilizado por el proceso.

```
#include <time.h>
clock_t clock(void);
Compatibilidad: ANSI, UNIX y MS-DOS
```

time

La función **time** da como resultado el número de segundos transcurridos desde las 0 horas del 1 de Enero de 1970.

```
#include <time.h>
time_t time(time_t *seg);
Compatibilidad: ANSI, UNIX y MS-DOS
```

El tipo **time_t** está definido así:

```
typedef long time_t;
```

ctime

La función **ctime** convierte un tiempo almacenado como un valor de tipo **time_t**, en una cadena de caracteres de la forma:

```
Wed Jan 02 02:03:55 1980\n\0
```

```
#include <time.h>
char *ctime(const time_t *seg);
Compatibilidad: ANSI, UNIX y MS-DOS
```

Esta función devuelve un puntero a la cadena de caracteres resultante o un puntero nulo si *seg* representa un dato anterior al 1 de Enero de 1970. Por ejemplo, el siguiente programa presenta la fecha actual y a continuación genera cinco números seudoaleatorios, uno cada segundo.

```
/********** Generar un número aleatorio cada segundo **********/
/* time.c
 */
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

void main(void)
{
  long x, tm;
  time_t segundos;
```

```
  time(&segundos);
  printf("\n%s\n", ctime(&segundos));
  srand((unsigned)(segundos % 65536));
  for (x = 1; x <= 5; x++)
  {
    do      /* tiempo de espera igual a 1 segundo */
      tm = clock();
    while (tm/CLOCKS_PER_SEC < x);
    /* se genera un número aleatorio cada segundo */
    printf("Iteración %ld: %d\n", x, rand());
  }
}
```

localtime

La función **localtime** convierte el número de segundos transcurridos desde la 0 horas del 1 de Enero de 1970, valor obtenido por la función **time**, a la fecha y hora correspondiente (corregida en función de la zona horaria en la que nos encontremos). El resultado es almacenado en una estructura de tipo **tm**, definida en *time.h*.

```
#include <time.h>
struct tm *localtime(const time_t *seg);
Compatibilidad: ANSI, UNIX y MS-DOS
```

La función **localtime** devuelve un puntero a la estructura que contiene el resultado, o un puntero nulo si el tiempo no puede ser interpretado. Los miembros de la estructura son los siguientes:

Campo	*Valor almacenado*
tm_sec	Segundos (0 - 59)
tm_min	Minutos (0 - 59)
tm_hour	Horas (0 - 23)
tm_mday	Día del mes (1 - 31)
tm_mon	Mes (0 - 11; Enero = 0)
tm_year	Año (actual menos 1900)
tm_wday	Día de la semana (0 - 6; Domingo = 0)
tm_yday	Día del año (0 - 365; 1 de Enero = 0)

El siguiente ejemplo muestra cómo se utiliza esta función.

```
/* localtim.c */
#include <stdio.h>
#include <time.h>

void main()
{
```

```
    struct tm *fh;
    time_t segundos;
    time(&segundos);
    fh = localtime(&segundos);
    printf("%d horas, %d minutos\n", fh->tm_hour, fh->tm_min);
}
```

EJERCICIOS RESUELTOS

1. El calendario Gregoriano actual obedece a la reforma del calendario juliano que ordenó el papa Gregorio XIII en 1582. Se decidió, después de algunas modificaciones, que en lo sucesivo fuesen bisiestos todos los años múltiplos de cuatro, pero que de los años seculares (los acabados en dos ceros) sólo fuesen bisiestos aquéllos que fuesen múltiplos de cuatrocientos. Según estos conceptos, construir un programa para que dada una fecha (día, mes y año) devuelva como resultado el día correspondiente de la semana.

   La estructura del programa estará formada, además de por la función **main**, por las funciones:

```
void LeerFecha(int *dia, int *mes, int *anyo);
void EntradaDatos(int *dia, int *mes, int *anyo);
int  DatosValidos(int dia, int mes, int anyo);
int  AnyoBisiesto(int anyo);
void EscribirFecha( int dd, int mm, int aa);
int DiaSemana(int dia, int mes, int anyo);
```

   La función *LeerFecha* llama a la función *EntradaDatos* para leer los datos día, mes y año, y a la función *DatosValidos* para asegurar que los datos introducidos se corresponden con una fecha correcta.

   La función *EntradaDatos* llama a su vez a la función *AnyoBisiesto* para verificar si el año es o no bisiesto.

   La función *EscribirFecha* llama a la función *DiaSemana* para calcular el día de la semana al que corresponde la fecha introducida y visualiza el resultado.

   El programa completo se muestra a continuación.

```
/*****************************************************************
                        CALENDARIO PERPETUO
*****************************************************************/

/* Dada una fecha (dia, mes, año)
 * indicar el día correspondiente de la semana.
 *
 * calendar.c
 */
```

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

void LeerFecha (int *dia, int *mes, int *anyo);
void EntradaDatos (int *dia, int *mes, int *anyo);
int  DatosValidos (int dia, int mes, int anyo);
int  AnyoBisiesto (int anyo);
void EscribirFecha (int dd, int mm, int aa);
int DiaSemana (int dia, int mes, int anyo);

void main(void)  /* Función Principal */
{
  int dia, mes, anyo;

  LeerFecha(&dia, &mes, &anyo);
  EscribirFecha(dia, mes, anyo);
}

/*****************************************************************
                          FUNCIONES
*****************************************************************/

void LeerFecha(int *dia, int *mes, int *anyo)
{
  int datos_validos;

  do
  {
    EntradaDatos(dia, mes, anyo);
    datos_validos = DatosValidos(*dia, *mes, *anyo);
  }
  while (!datos_validos);
}

void EntradaDatos(int *dia, int *mes, int *anyo)
{
  printf("Día (1 - 31)    "); scanf("%d", dia);
  printf("Mes (1 - 12)    "); scanf("%d", mes);
  printf("Año (1582 -->) "); scanf("%d", anyo);
}

int DatosValidos(int dia, int mes, int anyo)
{
  int r, anyoB, mesB, diaB;

  anyoB = (anyo >= 1582);
  mesB = (mes >= 1) && (mes <= 12);
  switch (mes)
  {
    case 2:
      if (r = AnyoBisiesto(anyo))
        diaB = (dia >= 1) && (dia <= 29);
      else
        diaB = (dia >= 1) && (dia <= 28);
      break;
    case 4: case 6: case 9: case 11:
      diaB = (dia >= 1) && (dia <= 30);
```

```
        break;
      default:
        diaB = (dia >= 1) && (dia <= 31);
    }
    if (!(diaB && mesB && anyoB))
    {
      printf("\nDATOS NO VÁLIDOS\n\n");
      printf("Pulse <Entrar> para continuar ");
      r = getchar(); fflush(stdin);     return (0);
    }
    else
      return (1);
}

int AnyoBisiesto(int anyo)
{
  int verdad = 1, falso = 0;

  if ((anyo % 4 == 0) && (anyo % 100 != 0) || (anyo % 400 == 0))
    return (verdad);
  else
    return (falso);
}

void EscribirFecha(int dd, int mm, int aa)
{
  int d;

  static char dia[7][10] =  { "Sábado", "Domingo", "Lunes",
                              "Martes", "Miércoles", "Jueves",
                              "Viernes" };
  static char mes[12][11] = { "Enero", "Febrero", "Marzo",
                              "Abril", "Mayo", "Junio", "Julio",
                              "Agosto", "Septiembre", "Octubre",
                              "Noviembre", "Diciembre" };

  d = DiaSemana(dd, mm, aa);
  printf("\n%s %d de %s de %d\n",dia[d], dd, mes[mm-1], aa);
}

int DiaSemana(int dia, int mes, int anyo)
{
  if (mes <= 2)
  {
    mes = mes + 12;
    anyo = anyo - 1;
  }
  return ((dia+2*mes+3*(mes+1)/5+anyo+anyo/4-anyo/100+
           anyo/400+2)%7);
}
```

2. La transformada discreta de Fourier (DFT) de una secuencia de números ($x[n]$) se define así:

$$X[k] = \sum_{n=0}^{N-1} x[n] \cdot e^{-jk\frac{2\pi}{N}n};\ n = 0,1,\cdots,N-1;\ k = 0,1,\cdots,N-1$$

$x[n] \in \mathbb{R}$, (Cuerpo de los nros. reales)

$X[k] \in \mathbb{C}$, (Cuerpo de los nros. complejos)

Se desea escribir un programa que calcule la DFT de una secuencia de números reales. Para ello se pide:

a) Escribir las funciones

```
complejo sumar(complejo a, complejo b);
complejo multiplicar(complejo a, complejo b);
```

para trabajar con números complejos definidos de la forma:

```
typedef struct
{
  double r, i; /* Partes real e imaginaria del número */
} complejo;
```

La función *sumar* devuelve un complejo resultado de sumar el complejo *a* y el complejo *b* pasados como argumentos, y la función *multiplicar* devuelve el producto.

b) Escribir una función que calcule la DFT. La declaración de esta función es:

```
void DFT(complejo *X, double *x, int N);
```

Tenga en cuenta las siguientes consideraciones:

1. $e^{jx} = \cos(x) + j\,\mathrm{sen}(x)$
2. Para efectuar los cálculos se pueden utilizar las siguientes funciones declaradas en el fichero de cabecera *math.h*:

   exp(x), cos(x) y sin(x)
3. $\pi = 3.141592654$

c) Escribir un programa que lea del fichero estándar de entrada una secuencia de números reales y escriba en el fichero estándar de salida la secuencia correspondiente a la DFT.

El programa completo se muestra a continuación.

```
/*************** Transformada discreta de Fourier ***************/
/* fourier.c
 */
```

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

typedef struct
{
  double real, imag;
} complejo;

complejo sumar( complejo a, complejo b )
{
  complejo temp;
  temp.real = a.real + b.real;
  temp.imag = a.imag + b.imag;
  return temp;
}

complejo multiplicar( complejo a, complejo b )
{
  complejo temp;
  temp.real = a.real * b.real - a.imag * b.imag;
  temp.imag = a.real * b.imag + a.imag * b.real;
  return temp;
}

void DFT( complejo *X, double *x, int N )
{
  int n, k;
  double t, pi = 3.141592654;
  complejo a, b;

  for ( k = 0; k < N; k++ )
  {
    X[k].real = 0; X[k].imag = 0;
    for ( n = 0; n < N; n++ )
    {
      a.real = x[n]; a.imag = 0;
      t = k * 2 * pi / N * n;
      b.real = cos( -t ); b.imag = sin( -t );
      b = multiplicar( a, b );
      X[k] = sumar( X[k], b );
    }
  }
}

void main(void)
{
  complejo *X;
  double *x;
  int n, N;

  printf( "Cuántos valores reales desea introducir\n" );
  scanf( "%d", &N );

  /* Asignar memoria para el array de complejos */
  if ((X = (complejo *)malloc(N * sizeof(complejo))) == NULL)
  {
```

```
    printf( "Insuficiente memoria para asignación\n" );
    exit( 1 );
  }

  /* Asignar memoria para el array que almacenará la secuencia
   * de números reales
   */
  if ((x = (double *)malloc(N * sizeof(double))) == NULL)
  {
    printf( "Insuficiente memoria para asignación\n" );
    exit( 1 );
  }

  /* Introducir la secuencia de números reales */
  printf( "Introduzca los valores\n" );
  for ( n = 0; n < N; n++ )
    scanf( "%lf", &x[n] );

  /* Calcular la transformada discreta de Fourier */
  DFT(X, x, N);

  printf("Resultado:\n");
  for ( n = 0; n < N; n++ )
    printf( "%g%+g j\n", X[n].real, X[n].imag );
}
```

3. Partiendo de la declaración:

```
typedef struct
{
  float real;
  float imag;
} tcomplejo;
```

escribir un programa que calcule la suma de *n* complejos. Para ello se pide:

a) Escribir una función *AsigMem* que asigne memoria para *x* complejos y devuelva como resultado la dirección del bloque de memoria asignado. El prototipo de esta función será así:

```
tcomplejo *AsigMem(int x);
```

b) Escribir una función *SumarComplejos* que reciba como parámetros dos complejos y devuelva como resultado un puntero al complejo suma de los dos anteriores. El prototipo de esta función será:

```
tcomplejo *SumarComplejos(tcomplejo c1, tcomplejo c2);
```

c) Utilizando las funciones anteriores, escribir un programa que lea *n* complejos de la entrada estándar, los almacene en un array dinámico y visualice como resultado la suma total de los *n* complejos.

El programa completo se muestra a continuación.

```
/*************** Operaciones con complejos ***************/
/* complej3.c
 */
#include <stdio.h>
#include <stdlib.h>

typedef struct
{
  float real;
  float imag;
} tcomplejo;

tcomplejo *AsigMem(int x);
tcomplejo *SumarComplejos(tcomplejo c1, tcomplejo c2);

void main()
{
  tcomplejo *pcomplejo = NULL, *pcomsuma = NULL, *panterior = NULL;
  int n = 0, i = 0;

  printf("Número de elementos del array: ");
  scanf("%d", &n);

  /* Asignar memoria para el array de complejos */
  pcomplejo = AsigMem(n);
  if ( pcomplejo == NULL )
  {
    printf("Memoria insuficiente\n");
    exit(-1);
  }

  /* Asignar memoria para el complejo suma */
  pcomsuma = AsigMem(1);
  if ( pcomsuma == NULL )
  {
    printf("Memoria insuficiente\n");
    exit(-1);
  }
  pcomsuma->real = pcomsuma->imag = 0;

  printf("\nIntroducir datos de la forma: x yj\n");
  for ( i = 0; i < n; i++ )
  {
    printf("complejo %d: ", i+1);
    scanf("%f %f", &pcomplejo[i].real, &pcomplejo[i].imag);
    fflush(stdin);
    panterior = pcomsuma;
    pcomsuma = SumarComplejos(*pcomsuma, pcomplejo[i]);
    free(panterior);
  }
```

```
  printf("Resultado: %g%+gj\n", pcomsuma->real, pcomsuma->imag);

  /* Liberar la memoria asignada */
  free(pcomplejo);
  free(pcomsuma);
}

tcomplejo *AsigMem(int x)
{
  tcomplejo *p;
  p = (tcomplejo *)malloc(x * sizeof(tcomplejo));

  return p;
}

tcomplejo *SumarComplejos(tcomplejo c1, tcomplejo c2)
{
  tcomplejo *pcom;
  /* Asignar memoria para el complejo suma */
  pcom = AsigMem(1);
  if ( pcom == NULL )
  {
    printf("Memoria insuficiente\n");
    exit(-1);
  }
  pcom->real = c1.real + c2.real;
  pcom->imag = c1.imag + c2.imag;
  return pcom;
}
```

4. Escribir un programa que calcule la serie:

$$e^x = 1 + \frac{x}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + \ldots$$

Para un valor de x dado, se calcularán y sumarán términos sucesivos de la serie, hasta que el siguiente término a sumar sea menor que una constante de error especificada. Para ello se pide:

a) Escribir una función *recursiva* que tenga el siguiente prototipo:

```
float expn(float x, int n, float tp, float error);
```

Esta función deberá imprimir todos sus parámetros, como resultados parciales, en cada una de las etapas del proceso recursivo. La función *expn* devolverá como resultado el valor aproximado de e^x. El parámetro *tp* representa cada uno de los términos parciales que se van calculando.

b) Escribir un programa que ejecute una llamada a la función *expn* de la forma siguiente:

```
expn(x, 0, 1, error);
```

Probar el programa para *x* igual a 1 y *error* igual a 0.0001. El resultado será el número *e*.

El programa completo se muestra a continuación.

```
/********************** Cálculo de exp(x) *********************/
/* exp.c
 */
#include <stdio.h>

float expn(float x, int n, float tp, float error)
{
  if ( tp < 0.00001 )
    return 0;
  else
  {
    printf("x = %g, n = %d, tp = %g\n", x, n, tp);
    return tp + expn(x, ++n, tp*x/n, error);
  }
}

void main()
{
  float ex, x, error;

  printf("Valor de x: ");
  scanf("%f", &x);
  printf("Error: ");
  scanf("%f", &error);

  ex = expn(x, 0, 1, error);
  printf("\nexp(%g) = %g\n", x, ex);
}
```

EJERCICIOS PROPUESTOS

1. Realizar un programa que:

   a) Lea dos cadenas de caracteres denominadas *cadena1* y *cadena2* y un número entero *n*. Las cadenas finalizarán con el carácter nulo (ASCII 0).

   b) Llame a una función:

   ```
   int compcads(cadena1, cadena2, n);
   ```

que compare los *n* primeros caracteres de *cadena1* y de *cadena2* y devuelva como resultado un valor entero

0 si *cadena1* y *cadena2* son iguales
1 si *cadena1* es mayor que *cadena2* (los *n* primeros caracteres)
−1 si *cadena1* es menor que *cadena2* (los *n* primeros caracteres)

Si *n* es menor que 1 o mayor que la longitud de la menor de las cadenas, la comparación se hará sin tener en cuenta este parámetro.

c) Escriba la cadena que sea menor según los *n* primeros caracteres (esto es, la que esté antes por orden alfabético).

2. Suponiendo un texto escrito en minúsculas y sin signos de puntuación, es decir, una palabra estará separada de otra por un espacio en blanco, realizar un programa que lea texto de la entrada estándar (del teclado) y dé como resultado la frecuencia con que aparece cada palabra leída del texto. El resultado se almacenará en un array en el que cada elemento será una estructura del tipo siguiente:

```
typedef struct
{
  char *palabra; /* palabra */
  int contador;  /* número de veces que aparece en el texto */
} telem;
```

La estructura del programa estará formada por la función **main** y por las funciones siguientes:

```
int BuscarPalabra(char *a, char *palabra);
void InsertarPalabra(char *a, char *palabra);
void VisualizarArray(char *a);
```

La función **main** asignará memoria para un array de *n* elementos, inicializará los elementos del array a cero, utilizando las funciones anteriores calculará la frecuencia con la que aparece cada una de las palabras y visualizará el resultado.

La función *BuscarPalabra* verificará sí la palabra leída de la entrada estándar, está en el array *a*. Esta función devolverá un valor distinto de cero si la palabra está en el array y un cero en caso contrario.

La función *InsertarPalabra* permitirá añadir una nueva palabra al final del array *a*. Tenga en cuenta que cada palabra en el array está referenciada por un puntero.

La función *VisualizarPalabra* visualizará cada una de las palabras del array seguida del número de veces que apareció.

3. Modificar el programa anterior para que la función *InsertarPalabra* inserte cada nueva palabra en el orden que le corresponde alfabéticamente, moviendo las elementos necesarios un lugar hacia atrás. De esta forma, cuando finalice la lectura del texto, el array estará clasificado.

4. Modificar el programa *arrays03.c* realizado en este capítulo, para que la asignación de memoria para el array de punteros referenciado por *listaFinal* sea realizada en la función *fusionar* y no en la función **main**.

5. Realizar un programa que lea un conjunto de valores reales a través del teclado, los almacene en un array dinámico de *m* filas por *n* columnas y a continuación, visualice el array por filas. Cuando el programa finalice, se liberará la memoria ocupada por el array.

   La estructura del programa estará formada, además de por la función **main,** por las funciones siguientes:

   ```
   void Leer( float **x, int fi, int co );
   ```

   El parámetro *x* de la función *leer* es un puntero a un puntero que referenciará el array cuyos elementos deseamos leer. El array tiene *fi* filas y *co* columnas.

   ```
   float **AsigMem(int fi, int co );
   ```

   La función *AsigMem* tiene como fin asignar memoria para el array. El parámetro *fi* es el número de filas del array y el parámetro *co* el número de columnas. La función *AsigMem* devuelve un puntero a un puntero al espacio de memoria asignado.

6. Escribir un programa para evaluar la expresión $(ax + by)^n$. Para ello, tenga en cuenta las siguientes expresiones:

$$(ax + by)^n = \sum_{k=0}^{n} \binom{n}{k} (ax)^{n-k} (by)^k$$

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}$$

$$n! = n * (n-1) * (n-2)...2 * 1$$

   a) Escribir una función cuyo prototipo sea:

   ```
   long factorial( int n );
   ```

La función *factorial* recibe como parámetro un entero y devuelve el factorial del mismo.

b) Escribir una función con el prototipo,

```
long combinaciones( int n, int k );
```

La función *combinaciones* recibe como parámetros dos enteros n y k, y devuelve como resultado el valor de $\binom{n}{k}$.

c) Escribir una función que tenga el prototipo:

```
long potencia( int base, int exponente );
```

La función *potencia* recibe como parámetros dos enteros *base* y *exponente*, y devuelve como resultado el valor de $base^{exponente}$

d) Escribir la función principal:

```
void main()
```

La función **main** leerá los valores de a, b, n, x e y, y utilizando las funciones anteriores escribirá como resultado el valor de $(ax + by)^n$.

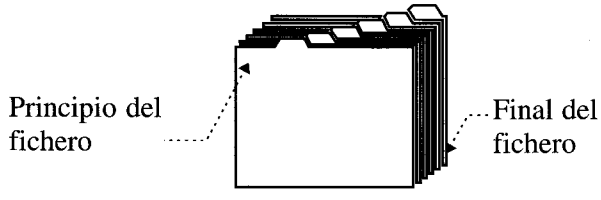
CAPÍTULO 9

FUNCIONES ESTÁNDAR DE E/S

Todos los programas realizados hasta ahora obtenían los datos necesarios para su ejecución de la entrada estándar y visualizaban los resultados en la salida estándar. Por otra parte, el programa retiene los datos que manipula mientras esté en ejecución; es decir, los datos introducidos se pierden cuando el programa finaliza.

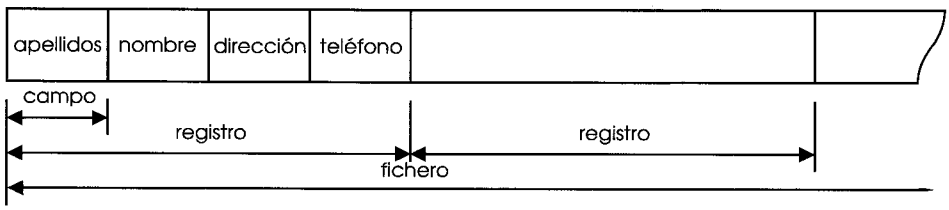
Por ejemplo, si hemos realizado un programa con la intención de construir una agenda, lo ejecutamos y almacenamos los datos *apellidos*, *nombre*, *dirección* y *teléfono* de cada uno de los componentes de la agenda en un array de estructuras, los datos estarán disponibles mientras el programa esté en ejecución. Si finalizamos la ejecución del programa y lo ejecutamos de nuevo, tendremos que volver a introducir de nuevo todos los datos.

La solución para hacer que los datos persistan de una ejecución para otra es almacenarlos en un fichero en el disco, en vez de en un array en memoria. Entonces, cada vez que se ejecute el programa que trabaja con esos datos, podrá leer del fichero los que necesite y manipularlos. Nosotros procedemos de forma análoga en muchos aspectos de la vida ordinaria; almacenamos los datos en fichas y guardamos el conjunto de fichas en lo que generalmente denominamos fichero o archivo.



Desde el punto de vista informático, un fichero es una colección de información que almacenamos en un soporte magnético para poder manipularla en cualquier momento. Esta información se almacena como un conjunto de registros (estructuras), conteniendo todos ellos, generalmente, los mismos campos (miembros de la estructura). Cada campo almacena un dato de un tipo predefinido o definido por el usuario. El registro más simple estaría formado por un único carácter.

Por ejemplo, si quisiéramos almacenar en un fichero los datos relativos a la agenda a la que nos hemos referido anteriormente, podríamos diseñar cada registro con los campos *apellidos*, *nombre*, *dirección* y *teléfono*. Gráficamente, puede imaginarse la estructura del fichero así:



Para dar soporte para el trabajo con ficheros, la biblioteca de C proporciona varias funciones de entrada/salida (E/S) que permiten leer y escribir datos a, y desde, ficheros y dispositivos. Este capítulo presenta las funciones estándar de E/S; su característica fundamental es que la E/S, en el procesamiento de ficheros, se realiza a través de un *buffer* o memoria intermedia. También, permiten la E/S con formato. C se refiere a éstas funciones en los términos "*streams* de E/S".

MANIPULACIÓN DE FICHEROS EN EL DISCO

Para poder escribir o leer sobre un fichero, primeramente hay que abrirlo. El fichero puede ser abierto para leer, para escribir o para leer y escribir.

En C y en C++ abrir un fichero significa definir un *stream* que permita el acceso al fichero en el disco para leer o escribir. La definición del *stream* acarrea la definición de un *buffer* (memoria intermedia) para conectar, a través de él, el *stream* con el fichero en el disco. Esto permite referirse al *stream* como si fuera el fichero (más adelante veremos que un *stream* se define como un puntero a una estructura de tipo *FILE*).

Puede imaginarse un *stream* como un fichero ingenioso que actúa como fuente y/o como destino para los bytes.

La utilización de un *buffer* para realizar las operaciones de E/S es una técnica implementada en software diseñada para hacer las operaciones de E/S más eficientes. Un *buffer* es un área de datos en la memoria (RAM) asignada por el programa que abre el fichero. La utilización de *buffers* en operaciones de E/S reduce el número de accesos al dispositivo físico asociado con el fichero necesarios para la transferencia de información entre el programa y el fichero; un acceso a un dispositivo físico consume mucho más tiempo que un acceso a la memoria RAM. Cuando un fichero no tiene asociado un *buffer*, cada byte escrito en, o leído desde, el fichero es físicamente transferido en el momento de la operación. En cambio, cuando un fichero tiene asociado un *buffer*, todas las operaciones de E/S requeridas son servidas desde ese *buffer*, y la transferencia física de datos se hace en múltiplos del tamaño del *buffer*.

Las operaciones de lectura y escritura siempre empiezan en una posición perfectamente definida por un puntero de lectura y por un puntero de escritura, respectivamente, o simplemente, por un puntero de lectura/escritura (L/E).

Después de haber finalizado el trabajo con un fichero, éste debe cerrarse. Si un fichero no se cierra explícitamente, es cerrado automáticamente cuando finaliza el programa. Sin embargo, es aconsejable cerrar un fichero cuando se ha finalizado con él, ya que el número de ficheros abiertos al mismo tiempo está limitado.

Un fichero en C/C++ está organizado secuencialmente, y el acceso al mismo puede ser secuencial o aleatorio. Se habla de acceso secuencial cuando se va accediendo a posiciones sucesivas; esto es, tras acceder a la posición *n*, se accede a la posición *n+1*, y se habla de acceso aleatorio cuando se accede directamente a la posición deseada sin necesidad de acceder a las posiciones que le preceden.

ABRIR UN FICHERO

Para poder escribir en un fichero o leer de un fichero, primeramente hay que abrirlo con las funciones **fopen** o **freopen**. El fichero puede ser abierto para leer, para escribir o para leer y escribir.

Cuando un programa comienza su ejecución, son abiertos automáticamente tres ficheros, que se corresponden con otros tres dispositivos. Estos ficheros, direccionados por *streams*, y los dispositivos asociados por defecto son:

Stream	*Dispositivo*
stdin	dispositivo de entrada estándar (teclado)
stdout	dispositivo de salida estándar (pantalla)
stderr	dispositivo de error estándar (pantalla)

■ En MS-DOS, además de estos tres ficheros, dependiendo de la configuración de la máquina, pueden estar presentes dos más, el dispositivo serie y el dispositivo de impresión paralelo:

stdaux	dispositivo auxiliar estándar (puerto serie)
stdprn	dispositivo de impresión estándar (impresora paralelo)

Los *streams* especificados anteriormente, están declarados como punteros constantes a una estructura de tipo **FILE**. Esta estructura define un *buffer* para conectar, a través de él, el *stream* con el fichero físico. Debido a esto en la mayoría de las ocasiones nos referiremos al *stream* como si se tratara del fichero. Estos *streams* pueden ser utilizados en cualquier función que requiera como argumento un puntero a una estructura de tipo **FILE**.

fopen

La función **fopen** abre el fichero especificado por *nomfi*. El argumento *modo* especifica cómo es abierto el fichero. Después de abrir un fichero, la posición de L/E está al principio del fichero, excepto para el modo añadir que está al final.

```
#include <stdio.h>
FILE *fopen(const char *nomfi, const char *modo);
Compatibilidad: ANSI, UNIX y MS-DOS
```

Modo	*Descripción*
"r"	Abrir un fichero para leer. Si el fichero no existe o no se encuentra, se obtiene un error.
"w"	Abrir un fichero para escribir. Si el fichero no existe, se crea; y si existe, su contenido se destruye para ser creado de nuevo.
"a"	Abrir un fichero para añadir información al final del mismo. Si el fichero no existe, se crea.
"r+"	Abrir un fichero para leer y escribir. El fichero debe existir.
"w+"	Abrir un fichero para leer y escribir. Si el fichero no existe, se crea; y si existe, su contenido se destruye para ser creado de nuevo.
"a+"	Abrir un fichero para leer y añadir. Si el fichero no existe, se crea.

■ Cuando estemos trabajando con un compilador C bajo el sistema operativo MS-DOS hay que tener en cuenta las siguientes consideraciones; por lo tanto, si es usuario de UNIX sáltese la letra pequeña. A diferencia de UNIX, en MS-DOS un fichero puede ser abierto en modo *texto* o en modo *binario*. La necesidad de dos modos diferentes, es por las incompatibilidades existentes entre C y MS-DOS ya que C fue diseñado originalmente para el sistema operativo UNIX. El modo *texto* es para ver los ficheros como si estuvieran bajo UNIX; y el modo *binario*, para verlos como si estuvieran bajo MS-DOS. En modo *texto*,

un final de línea es representado en C por un único carácter ('\n'), pero en un fichero de MS-DOS es representado por dos caracteres (*CR+LF*). Esto significa que, en MS-DOS, cuando un programa C escribe en un fichero convierte el carácter '\n' en los caracteres *CR+LF*; y cuando C lee de un fichero y encuentra los caracteres *CR+LF*, los convierte a '\n'; y cuando encuentra un *Ctrl+Z* lo interpreta como un **EOF**. La conversión puede ocasionar problemas cuando estos caracteres no se correspondan con texto, sino, por ejemplo, con un valor numérico que estemos recuperando de un fichero en disco y dos bytes de ese valor coincidan con la representación de los caracteres *CR+LF*. Para evitar este tipo de problemas, utilice el modo *binario*. En modo *binario* estas conversiones no tienen lugar.

■ Según lo expuesto en el párrafo anterior, a las formas de acceso mencionadas, se les puede añadir un carácter *t* o *b* (por ejemplo, *rb*, *a+b* o *ab+*), para indicar si el fichero se abre en modo texto o en modo binario. La opción *t*, no pertenece al lenguaje C estándar; sino que es una extensión de Microsoft C. Si *t* o *b* no se especifican, el modo texto o binario es definido por la variable global *_fmode* de C (modo texto por defecto).

■ En UNIX, la opción *b* es ignorada aunque sintácticamente es aceptada. Esto permite la transportabilidad de un programa hecho en MS-DOS a UNIX.

La función **fopen** devuelve un puntero a una estructura de tipo **FILE** que define, entre otros datos, el *buffer* asociado con el fichero abierto. Un puntero nulo indica un error. El puntero devuelto por **fopen** recibe el nombre de *stream* y es utilizado por las funciones estándar de E/S para leer y escribir datos en un fichero. Por eso, antes de invocar a la función **fopen** hay que definir un puntero que apunte a una estructura de tipo **FILE**. Por ejemplo,

```
#include <stdio.h>
FILE *pf;
pf = fopen("datos", "w");
if (pf == NULL)
  printf("Error: el fichero no se puede abrir\n");
```

Este ejemplo indica que se abre el fichero *datos* para escribir, y que en adelante será referenciado por el puntero *pf*. Para simplificar nos referiremos a este puntero diciendo que apunta al fichero abierto.

El tipo **FILE** está declarado en el fichero *stdio.h* así:

```
struct _iobuf
{
  char  *_ptr;    /* puntero al siguiente carácter del buffer */
  int   _cnt;     /* contador que indica los bytes que quedan en
                     el buffer de E/S */
  char  *_base;   /* puntero al buffer de E/S */
  char  _flag;    /* máscara que contiene el modo de acceso al
                     fichero y los errores que se producen al
                     acceder a él */
  char  _file;    /* es el descriptor del fichero */
};
typedef struct _iobuf FILE;
```

La variable *pf*, declarada en el ejemplo anterior, contiene la dirección de memoria de un elemento del array de estructuras *_iob[]* definido así:

```
extern FILE _iob[];
```

Esta asignación ocurre, por cada fichero que se abre. El array *_iob[]*, tiene un número de elementos que depende de la configuración del sistema. En UNIX el número de elementos es igual al número de entradas de la tabla de descriptores asociada con el proceso, y en MS-DOS es igual al valor especificado por la variable *FILES* declarada en el fichero de configuración del sistema, *config.sys*. Como ejemplo, observar cómo están definidas los *streams* estándar:

```
#define stdin  (&_iob[0])
#define stdout (&_iob[1])
#define stderr (&_iob[2])
```

freopen

La función **freopen** cierra el fichero actualmente asociado con el puntero *stream* y reasigna *stream* al fichero identificado por *nomfi*. Es utilizada para redireccionar *stdin, stdout* o *stderr* a ficheros especificados por el usuario. La descripción para el argumento *modo*, es la misma que la dada en la función **fopen**.

```
#include <stdio.h>
FILE *freopen(const char *nomfi, const char *modo, FILE *stream);
Compatibilidad: ANSI, UNIX y MS-DOS
```

La función **freopen** devuelve un puntero al fichero abierto nuevamente. Si ocurre un error, el fichero original es cerrado y se devuelve un puntero nulo. Por ejemplo,

```
#include <stdio.h>
FILE *pf;
pf = freopen("datos", "w", stdout);
if (pf == NULL)
  printf("Error: el fichero no se puede abrir\n");
// ...
printf("hola\n"); /* se escribe en el fichero "datos" */
```

Este ejemplo, reasigna **stdout** al fichero llamado *datos*. Ahora, lo que escribamos en **stdout**, será escrito en *datos*.

CERRAR UN FICHERO

Después de haber finalizado el trabajo con un fichero, éste debe cerrarse con la función **fclose**. Si un fichero no se cierra explícitamente, es cerrado automática-

mente cuando finaliza el programa. Sin embargo, es aconsejable cerrar un fichero cuando se ha finalizado con él, ya que el número de ficheros abiertos al mismo tiempo está limitado.

fclose

La función **fclose** cierra el *stream pf*, y por lo tanto cierra el fichero asociado con *pf*. Cualquier dato en el *buffer* asociado, se escribe en el fichero antes de cerrarlo.

```
#include <stdio.h>
int fclose(FILE *pf);
Compatibilidad: ANSI, UNIX y MS-DOS
```

Si la operación de cerrar el fichero se ejecuta satisfactoriamente, la función **fclose** devuelve un cero. Si ocurre un error entonces devuelve un **EOF**.

DETECCIÓN DE ERRORES

Cuando en una operación sobre un fichero ocurre un error, éste puede ser detectado por la función **ferror**. Cuando ocurre un error, el indicador de error permanece activado hasta que el fichero se cierra, a no ser que utilicemos la función **clearerr** o **rewind** para desactivarlo explícitamente.

ferror

La función **ferror** verifica si ha ocurrido un error en una operación con ficheros. Cuando ocurre un error, el indicador de error para ese fichero se pone activo y permanece en este estado, hasta que sea ejecutada la función **clearerr**.

```
#include <stdio.h>
int ferror(FILE *pf);
Compatibilidad: ANSI, UNIX y MS-DOS
```

La función **ferror** devuelve un cero si no ha ocurrido un error y un valor distinto de cero en caso contrario.

clearerr

La función **clearerr** desactiva el indicador de error y el indicador de fin de fichero para un determinado fichero, poniéndolos a valor 0.

```
#include <stdio.h>
void clearerr(FILE *pf);
Compatibilidad: ANSI, UNIX y MS-DOS
```

El siguiente ejemplo, muestra cómo utilizar algunas de las funciones explicadas hasta ahora. Lo que intenta hacer el ejemplo es abrir en el directorio actual un fichero llamado *datos* y escribir en él una cadena de caracteres. Cuando el programa finaliza, el fichero se cierra.

```
/* ferror.c */
#include <stdio.h>
#include <stdlib.h>

void main()
{
  FILE *pf;
  char *cadena = "Esta cadena nunca será escrita";

  if ((pf = fopen("datos", "r")) == NULL)
  {
    printf("Error: no se puede abrir el fichero\n");
    exit(1);
  }
  fprintf(pf, "%s\n", cadena);
  if (ferror(pf))
  {
    printf("Error al escribir en el fichero\n");
    clearerr(pf);
  }
  fclose(pf);
}
```

En el ejemplo anterior, cuando se invoca a la función **fopen** para abrir el fichero *datos* para leer (*r*) se verifica si la operación ha sido satisfactoria y en caso afirmativo se continúa. Para abrir un fichero para leer, el fichero debe existir. Si el fichero *datos* existe, se intenta escribir en él una cadena de caracteres y se verifica si la operación se ha efectuado correctamente. Para ello, la función **ferror** interroga los *flags* de error asociados con el fichero, detectando en este caso que ocurrió un error en la última operación de escritura. La función **ferror** manda un mensaje por la consola y la función **clearerr** desactiva el indicador de error para ese fichero. Este error se debe a que el fichero estaba abierto para leer, no para escribir. Si no hubiéramos hecho esta verificación, no nos hubiéramos enterado del error ya que el sistema no envía ningún mensaje.

feof

Cuando en una operación de lectura sobre un fichero se intenta leer más allá de la marca de fin de fichero, automáticamente el sistema activa el indicador de fin de fichero asociado con él. Para saber el estado de este indicador para un determinado fichero, hay que invocar a la función **feof**. La marca de fin de fichero es añadida automáticamente por el sistema cuando crea el fichero.

```
#include <stdio.h>
int feof(FILE *pf);
Compatibilidad: ANSI, UNIX y MS-DOS
```

La función **feof** devuelve un valor distinto de 0 cuando se intenta leer un elemento del fichero y nos encontramos con un *eof* (*end of file* - fin de fichero). En caso contrario devuelve un 0. Por ejemplo,

```
// ...
/* Leer el primer registro del fichero */
while (!feof(pf)) /* mientras no se llegue al final del fichero */
{
  // ...
  /* Leer el siguiente registro del fichero */
}
fclose(pf);
```

El bucle **while** del ejemplo anterior permite leer información del fichero referenciado por *pf* mientras no se llegue al final del fichero. Cuando se intente leer más allá del final del fichero el indicador de *eof* se activará y el bucle finalizará.

perror

La función **perror** escribe en la salida estándar **stderr** el mensaje especificado seguido por dos puntos, seguido del mensaje de error dado por el sistema y de un carácter nueva línea.

```
#include <stdio.h>
void perror(const char *mensaje);
Compatibilidad: ANSI, UNIX y MS-DOS
```

El siguiente ejemplo muestra cómo utilizar esta función. Observar que se trata del mismo ejemplo anterior, pero utilizando ahora la función **perror** en lugar de **printf**, para visualizar el mensaje de error.

```
/* perror.c */
#include <stdio.h>
#include <stdlib.h>

void main()
{
  FILE *pf;
  char *cadena = "Esta cadena nunca será escrita";

  if ((pf = fopen("datos", "r")) == NULL)
  {
    perror("datos");
    exit(1);
  }
  fprintf(pf, "%s\n", cadena);
```

```
  if (ferror(pf))
  {
    perror("Error al escribir en el fichero");
    clearerr(pf);
  }
  fclose(pf);
}
```

Al ejecutar este programa, si el fichero *datos* no existe, se visualiza el mensaje

```
datos: No such file or directory
```

En cambio, si el fichero datos existe, se visualiza el mensaje

```
Error al escribir en el fichero: Error 0
```

El número de error es almacenado en la variable del sistema **errno**. La variable del sistema **sys_errlist** es un array que contiene los mensajes de error ordenados por el número de error. El número de elementos del array viene dado por la variable del sistema **sys_nerr**. La función **perror** busca el mensaje de error en este array utilizando el valor de la variable **errno** como índice. Por ejemplo, según lo expuesto, otra versión del programa anterior es:

```
/* errno.c */
#include <stdio.h>
#include <stdlib.h>

void main()
{
  int i;
  FILE *pf;
  char *cadena = "Esta cadena nunca será escrita";
  if ((pf = fopen("datos", "r")) == NULL)
  {
    printf("datos: ");
    printf("%s\n", sys_errlist[errno]);
    exit(1);
  }
  fprintf(pf, "%s\n", cadena);
  if (ferror(pf))
  {
    printf("Error al escribir en el fichero: ");
    printf("%s\n", sys_errlist[errno]);
    clearerr(pf);
  }
  fclose(pf);
}
```

Las variables del sistema, **errno**, **sys_errlist** y **sys_nerr** están declaradas en el fichero de cabecera *stdlib.h*. No obstante, si no se incluye este fichero, podemos declararlas a nivel externo como se indica a continuación. Los números de error

puede verlos en el fichero de cabecera *errno.h*. El siguiente código le permitirá visualizar todos los mensajes de error.

```
/* errno.c */
#include <stdio.h>

extern int errno;            /* número de error */
extern char *sys_errlist[]; /* mensajes de error */
extern int sys_nerr;         /* elementos del array sys_errlist */

void main()
{
  int i;

  for (i = 0; i < sys_nerr; i++)
    printf("Error: %d, %s\n", i, sys_errlist[i]);
}
```

E/S CARÁCTER A CARÁCTER

Los datos pueden ser escritos en un fichero o leídos de un fichero, carácter a carácter con las funciones **fputc** y **fgetc**.

fputc

La función **fputc** escribe un carácter *car* en la posición indicada por el puntero de lectura/escritura (L/E) del fichero apuntado por *pf*.

```
#include <stdio.h>
int fputc(int car, FILE *pf);
Compatibilidad: ANSI, UNIX y MS-DOS
```

La función **fputc** devuelve el carácter escrito o un **EOF** si ocurre un error. No obstante, ya que **EOF** es un valor aceptado por *car*, utilizar la función **ferror** para verificar si ha ocurrido un error.

Por ejemplo, el siguiente programa crea un fichero denominado *texto* y escribe en él la cadena de caracteres almacenada en el array *buffer*. La escritura sobre el fichero se hace carácter a carácter utilizando la función **fputc**.

```
/****** Escribir datos en un fichero carácter a carácter *******/
/* fputc.c
 */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

void main()
{
```

```
  FILE *pf;
  char buffer[81];
  int i = 0;

  /* Abrir el fichero "texto" para escribir */
  if ((pf = fopen("texto", "w")) == NULL)
  {
    perror("El fichero no se puede abrir");
    exit(1);
  }
  strcpy(buffer, "Este es un texto para fputc!!\n");

  while (!ferror(pf) && buffer[i])
    fputc(buffer[i++], pf);

  if (ferror(pf))
    perror("Error durante la escritura");

  fclose(pf);
}
```

Observe que cada vez que se realiza una operación de escritura sobre el fichero, se invoca a la función **ferror** para verificar si ha ocurrido un error.

fgetc

La función **fgetc** lee un carácter de la posición indicada por el puntero de L/E del fichero apuntado por *pf* y avanza la posición de L/E al siguiente carácter a leer.

```
#include <stdio.h>
int fgetc(FILE *pf);
Compatibilidad: ANSI, UNIX y MS-DOS
```

La función **fgetc** devuelve el carácter leído o un **EOF**, si ocurre un error o se detecta el final del fichero. No obstante, ya que **EOF** es un valor aceptado, utilizar la función **ferror** o **feof** para distinguir si se ha detectado el final del fichero o si ha ocurrido un error.

Por ejemplo, el siguiente programa lee carácter a carácter toda la información escrita en un fichero llamado *texto* y la almacena en un array denominado *buffer*.

```
/******** Leer datos de un fichero carácter a carácter ********/
/* fgetc.c
 */
#include <stdio.h>

void main()
{
  FILE *pf;
  char buffer[81];
  int i = 0;
```

```
  /* Abrir el fichero "texto" para leer */
  if ((pf = fopen("texto", "r")) == NULL)
  {
    perror("El fichero no se puede abrir");
    exit(1);
  }
  while (!ferror(pf) && !feof(pf))
    buffer[i++] = fgetc(pf);
  buffer[--i] = '\0';

  if (ferror(pf))
    perror("Error durante la lectura");

  fclose(pf);

  printf("%s", buffer);
}
```

Como aplicación, vamos a realizar un programa, *conta*, que dé como resultado el número de caracteres de un fichero cualquiera del disco. El fichero en cuestión, será pasado como argumento en la línea de órdenes cuando se ejecute el programa *conta*. Por ejemplo,

```
conta texto↵
```

Esta orden ejecuta el programa *conta*, dando como resultado el número de caracteres del fichero *texto*. El proceso básicamente consiste en incrementar un contador una unidad por cada carácter que se lea del fichero *texto*.

```
/************* Contar los caracteres de un fichero *************/
/* conta.c
 */
#include <stdio.h>
#include <stdlib.h>

void main(int argc, char *argv[])
{
  FILE *pf;
  int conta = 0;

  /* Comprobar el número de argumentos pasados en la línea de
   * órdenes
   */
  if (argc != 2)
  {
    printf("Sintaxis: conta nombre_fichero\n");
    exit(1);
  }
  /* Abrir el fichero indicado por argv[1] */
  if ((pf = fopen(argv[1], "r")) == NULL)
  {
    printf("El fichero %s no puede abrirse\n", argv[1]);
    exit(1);
  }
```

```
  while (!ferror(pf) && !feof(pf))
  {
    fgetc(pf); conta++;    /* contar caracteres */
  }

  if (ferror(pf))
    perror("Error durante la lectura");
  else
    printf("El fichero %s tiene %d caracteres\n", argv[1],conta-1);

  fclose(pf);
}
```

E/S PALABRA A PALABRA

Los datos pueden ser escritos y leídos palabra a palabra con las funciones **putw** y **getw**. Se entiende por palabra, la palabra máquina o dato de tipo **int**.

putw

La función **putw** escribe un dato binario *entb* de tipo **int**, en el fichero apuntado por *pf*.

```
#include <stdio.h>
int putw(int entb, FILE *pf);
Compatibilidad: UNIX y MS-DOS
```

La función **putw** devuelve el valor escrito o un **EOF** si ocurre un error. No obstante, ya que **EOF** es un valor válido, utilizar la función **ferror** para verificar si ha ocurrido un error.

getw

La función **getw** lee el siguiente dato binario de tipo **int**, del fichero apuntado por *pf* y avanza el puntero de L/E al siguiente dato no leído.

```
#include <stdio.h>
int getw(FILE *pf);
Compatibilidad: UNIX y MS-DOS
```

La función **getw** devuelve el valor leído o un **EOF** si ocurre un error o se detecta el final del fichero. No obstante, ya que **EOF** es un valor válido, utilizar la función **ferror** o **feof** para distinguir si se ha detectado el final del fichero o si ha ocurrido un error.

Por ejemplo, el siguiente programa escribe el contenido de un array de enteros llamado *lista* en un fichero llamado *datos.bin* y a continuación visualiza el contenido de dicho fichero.

```
/**** Escribir y leer datos en un fichero palabra a palabra ****/
/* putw.c
 */
#include <stdio.h>
#include <stdlib.h>

void main()
{
  FILE *pf;
  static int lista[] = { -1, 10, 20, 30, 40, 50 };
  int elementos = sizeof(lista)/sizeof(int);
  int i;

  /* Abrir el fichero para leer y escribir */
  if ((pf = fopen("datos.bin", "w+")) == NULL)
  {
    perror("El fichero no se puede abrir");
    exit(1);
  }

  /* Escribir el array de enteros en el fichero */
  for (i = 0; i < elementos; i++)
  {
    putw(lista[i], pf);
    if (ferror(pf))
    {
      perror("Error durante la escritura");
      exit(1);
    }
  }

  /* Posicionar el puntero de L/E al principio */
  rewind(pf);

  /* Escribir el contenido del fichero */
  while (1)
  {
    i = getw(pf);
    if (feof(pf) || ferror(pf))
      break;
    printf("%d  ", i);
  }
  if (ferror(pf))
    perror("Error durante la lectura");
  fclose(pf);
}
```

Observe que para escribir el contenido del fichero, primeramente hay que situar el puntero de L/E al principio del mismo, lo cual se hace invocando a la fun-

ción **rewind**. La razón es que cuando terminamos de escribir los datos en el fichero, el puntero de L/E está apuntando al final del mismo.

E/S DE CADENAS DE CARACTERES

Los datos pueden ser escritos en un fichero y leídos de un fichero como cadenas de caracteres con las funciones **fputs** y **fgets**.

fputs

La función **fputs** copia la cadena de caracteres almacenada en *cadena*, en el fichero apuntado por *pf*. La terminación '\0' con la que finaliza toda cadena C, no se copia.

```
#include <stdio.h>
int fputs(const char *cadena, FILE *pf);
Compatibilidad: ANSI, UNIX y MS-DOS
```

La función **fputs** devuelve un valor 0 si se ejecuta satisfactoriamente. En caso contrario, devuelve un valor distinto de 0.

Para recuperar de una forma sencilla la información escrita en el fichero, es aconsejable copiar el carácter '\n' después de cada cadena escrita sobre el fichero. Por ejemplo,

```
while (gets(cadena) != NULL)
{
  fputs(cadena, pf); /* escribir la cadena en el fichero */
  fputc('\n', pf);   /* escribir a continuación, el carácter \n */
}
```

fgets

La función **fgets** lee una cadena de caracteres del fichero apuntado por *pf* y la almacena en *cadena*. Se entiende por cadena la serie de caracteres que va desde la posición actual dentro del fichero, hasta el primer carácter nueva línea ('\n') incluido éste, o hasta el final del fichero, o hasta que el número de caracteres sea igual a n–1. La terminación '\0' es añadida automáticamente a la cadena leída y el carácter '\n', si lo hay, no es eliminado.

```
#include <stdio.h>
char *fgets(char *cadena, int n, FILE *pf);
Compatibilidad: ANSI, UNIX y MS-DOS
```

La función **fgets** devuelve un puntero al principio de la cadena leída. Si el valor devuelto es **NULL**, quiere decir que ha ocurrido un error o que se ha detectado la marca de fin de fichero. Utilizar la función **feof** o **ferror** para determinar lo que ha ocurrido.

Por ejemplo, el siguiente programa lee líneas de texto y las almacena en un fichero. El programa preguntará por el nombre del fichero. Para hacer fácil la recuperación de cada una de las cadenas almacenadas en el fichero, escribiremos a continuación de cada una de ellas el carácter '\n'. Finalmente el programa visualiza el contenido del fichero creado.

```
/*********** Entrada/salida de cadenas de caracteres ***********/
/* fgets.c
 */
#include <stdio.h>
#include <stdlib.h>
#define N 81

void main()
{
  FILE *pf;
  char buffer[N], f[13];

  printf("Nombre del fichero: ");
  gets(f);
  f[12] = '\0'; /* truncar nombres superiores a 12 caracteres */
  /* Abrir el fichero f para escribir y leer */
  if ((pf = fopen(f, "w+")) == NULL)
  {
    printf("El fichero %s no puede abrirse.", f);
    exit(1);
  }
  printf("Fichero %s abierto\n", f);
  printf("Introducir datos. Finalizar cada línea con <Entrar>\n");
  printf("Para terminar introduzca la marca eof\n\n");
  while (gets(buffer) != NULL)
  {
    fputs(buffer, pf);
    fputc('\n', pf);
    if (ferror(pf))
    {
      perror("Error al escribir");
      exit(2);
    }
  }
  /* Visualizar el contenido del fichero */
  rewind(pf);  /* situarse al principio del fichero */
  /* Leer hasta un '\n' o hasta N-1 caracteres */
  while (fgets(buffer, N, pf) != NULL)
    printf("%s", buffer);
  if (ferror(pf))
    perror("Error durante la lectura");
  fclose(pf);
}
```

Sabemos que cuando la función **fgets** lee más allá de la marca de fin de fichero, devuelve el valor **NULL**. Otra forma de detectar el final del fichero, como puede ver a continuación, es utilizando la función **feof**. Por lo tanto, para interrogar si se ha alcanzado el final del fichero es necesario hacer antes una lectura.

```
fgets(buffer, N, pf);
while (!ferror(pf) && !feof(pf))
{
  printf("%s", buffer);
  fgets(buffer, N, pf);
}
```

ENTRADA/SALIDA CON FORMATO

Los datos pueden ser escritos con formato en un fichero y leídos con formato de un fichero utilizando las funciones **fprintf** y **fscanf**.

fprintf

La función **fprintf** escribe sus argumentos, *arg*, en el fichero apuntado por *pf*, con el formato especificado. La descripción de *formato*, es la misma que se especificó para **printf**.

```
#include <stdio.h>
int fprintf(FILE *pf, const char *formato[, arg]...);
Compatibilidad: ANSI, UNIX y MS-DOS
```

La función **fprintf** devuelve el número de caracteres escritos o un valor negativo si ocurre un error.

Si el *stream* especificado en la función **fprintf** es **stdout**, el resultado es el mismo que si hubiéramos invocado a la función **printf**. Esto es, las sentencias siguientes son equivalentes:

```
printf("n = %d\n", n);
fprintf(stdout, "n = %d\n", n);
```

Esto demuestra que un dispositivo físico recibe el mismo tratamiento que un fichero en el disco.

fscanf

La función **fscanf** lee sus argumentos, *arg*, del fichero apuntado por *pf*, con el formato especificado. La descripción de formato es la misma que se especificó para **scanf**. Cada argumento *arg*, debe ser un puntero a la variable en la que quere-

mos almacenar el valor leído. El tipo de cada una de estas variables debe corresponderse con la especificación de formato indicada para cada una de ellas.

```
#include <stdio.h>
int fscanf(FILE *pf, const char *formato[, arg]...);
Compatibilidad: ANSI, UNIX y MS-DOS
```

La función **fscanf** devuelve el número de argumentos que han sido leídos y asignados. Si el valor devuelto es un 0, significa que no se han asignado valores; y si es un **EOF**, significa que se ha detectado el final del fichero.

Si el *stream* especificado en la función **fscanf** es **stdin**, el resultado es el mismo que si hubiéramos invocado a la función **scanf**. Esto es, las sentencias siguientes son equivalentes:

```
scanf("%d", &n);
fscanf(stdin, "%d", &n);
```

El siguiente ejemplo, le enseña cómo utilizar las funciones **fprintf** y **fscanf**. No obstante, es importante conocer cómo la función **fprintf** almacena los datos sobre el disco. Los caracteres son almacenados uno por byte y los números enteros y reales en lugar de ocupar 2, 4 u 8 bytes dependiendo del tipo, requieren un byte por cada dígito. Por ejemplo el número –105.56 ocuparía 7 bytes. Por lo tanto, salvo excepciones, esta no es la forma idónea de almacenar datos ya que se ocupa mucho espacio en el disco.

```
/******* Escribir y leer datos con formato en un fichero *******/
/* fprintf.c
 */
#include <stdio.h>
#include <stdlib.h>

void main()
{
  char buffer[128];
  FILE *ptabla;

  long entl, total_entl;
  float real, total_real;
  int  i, c = 'A';

  /* Abrir un fichero para leer. Si no existe se crea. */
  if ((ptabla = fopen("tabla.d", "r")) != NULL)
  {
    /* Leer datos del fichero y totalizarlos */
    printf("RESULTADOS:\n\n");
    for (i = 0, total_entl = 0, total_real = 0.0F; i < 10; i++)
    {
      fscanf(ptabla, "%s %c: %ld %f", buffer,&c,&entl,&real);
      total_entl += entl;
      total_real += real;
```

```
      printf("\t%s %c: %7ld %9.2f\n", buffer, c, entl, real);
    }
    printf("\n\tTotal:   %7ld %9.2f\n", total_entl, total_real);
  }
  else
  {
    /* Si el fichero no existe lo creamos. */
    if ((ptabla = fopen( "tabla.d", "w" )) == NULL)
      exit(1);

    /* Se escribe la tabla deseada en el fichero. */
    for (i = 0, entl = 99999, real = 3.14F; i < 10; i++)
      fprintf(ptabla, "\tLínea %c: %7ld %9.2f\n",
                      c++, entl /= 2, real *= 2);
    printf("El fichero no existía y lo hemos creado.\n");
    printf("\nEjecute de nuevo el programa.\n");
  }
  fclose(ptabla);
}
```

E/S UTILIZANDO REGISTROS

Los datos pueden ser escritos y leídos como registros o bloques con las funciones **fwrite** y **fread**; esto es, como un conjunto de datos de longitud fija, tales como estructuras o elementos de un array. No obstante, aunque lo más habitual sea que un registro se corresponda con una estructura de datos, es factible también que un registro se corresponda con una variable de tipo **char**, **int**, **float** o con una cadena de caracteres, entre otros, lo que significa que estas funciones pueden sustituir en muchas ocasiones a las funciones estándar de E/S estudiadas hasta ahora.

fwrite

La función **fwrite** permite escribir *c* elementos de longitud *n* bytes almacenados en el *buffer* especificado, en el fichero apuntado por *pf*.

```
#include <stdio.h>
size_t fwrite(const void *buffer, size_t n, size_t c, FILE *pf);
```
Compatibilidad: ANSI, UNIX y MS-DOS

La función **fwrite** devuelve el número de elementos actualmente escritos. Si este número es menor que *c*, entonces es que ha ocurrido un error.

A continuación se muestran algunos ejemplos:

```
FILE *pf1, *pf2;
char car, cadena[36];
...
fwrite(&car, sizeof(char), 1, pf1);
fwrite(cadena, sizeof(cadena), 1, pf2);
```

Las sentencias anteriores son equivalentes a las que se exponen a continuación, sólo si la cadena escrita por **fputs** se completa con blancos (no con nulos) hasta su longitud menos uno (carácter nulo de terminación) porque **fwrite** escribe *sizeof(cadena)* caracteres; es decir, en el ejemplo, escribe 36 caracteres.

```
fputc(car, pf);
fputs(cadena, pf);
```

La función **fwrite**, almacena los datos numéricos en formato binario. Esto quiere decir que un **int** ocupa 2 o 4 bytes, un **long** ocupa 4 bytes, un **float** ocupa 4 bytes y un **double** ocupa 8 bytes.

- En MS-DOS, no hay que confundir el formato binario empleado para almacenar un dato numérico, con el modo binario en el que se abre un fichero, lo cual sólo indica que no se va a efectuar una conversión entre los caracteres '\n' y CR+LF.

- Con ficheros abiertos en modo texto, pueden producirse resultados inesperados debido a la conversión de '\n' en CR+LF. Por ello, en MS-DOS, es aconsejable trabajar en modo binario.

fread

La función **fread** permite leer *c* elementos de longitud *n* bytes, del fichero apuntado por *pf* y los almacena en el *buffer* especificado.

```
#include <stdio.h>
size_t fread(void *buffer, size_t n, size_t c, FILE *pf);
Compatibilidad: ANSI, UNIX y MS-DOS
```

La función **fread** devuelve el número de elementos actualmente leídos, que puede ser menor que *c* si ocurre un error. Utilizar las funciones **feof** o **ferror** para distinguir si se ha detectado el final del fichero o si ha ocurrido un error. Si *n* o *c* son 0, **fread** devuelve un 0 y el contenido del *buffer* permanece igual.

A continuación se muestran algunos ejemplos:

```
FILE *pf1, *pf2;
char car, cadena[36];
...
fread(&car, sizeof(char), 1, pf);
fread(cadena, sizeof(cadena), 1, pf);
```

Las sentencias anteriores son equivalentes a las que se exponen a continuación, con una particularidad, que la cadena leída por **fread** es una cadena de *sizeof(cadena)* caracteres; es decir, en el ejemplo, una cadena de 36 caracteres. Por lo tanto, como **fgets** lee *n–1* caracteres, *n* tiene que ser 37.

```
car = fgetc(pf);
fgets(cadena, 37, pf);
```

Estos ejemplos y los expuestos con **fwrite** demuestran que un registro puede ser una variable de tipo **char**, **int**, **float** o una cadena de caracteres, entre otros, pero lo más normal es que sea una estructura de datos, como puede ver a continuación.

Un ejemplo con registros

Después de los ejercicios realizados hasta ahora para trabajar con ficheros, habrá observado que la metodología de trabajo se repite. Es decir, para escribir datos en un fichero,

- Abrimos el fichero (función **fopen**).
- Leemos datos del dispositivo de entrada (o de otro fichero) y los escribimos en nuestro fichero. Este proceso se hace normalmente registro a registro. Para ello, utilizaremos las funciones estándar de E/S.
- Cerramos el fichero.

Para leer datos de un fichero existente,

- Abrimos el fichero (función **fopen**).
- Leemos datos del fichero y los almacenamos en variables de nuestro programa con el fin de trabajar con ellos. Este proceso se hace normalmente registro a registro. Para ello, utilizaremos las funciones estándar de E/S.
- Cerramos el fichero.

Esto pone de manifiesto que el fichero no es más que un medio de almacenamiento de datos permanente, dejando los datos disponibles para cualquier programa que necesite manipularlos. Es lógico que los datos serán recuperados del fichero con el mismo formato con el que fueron escritos, de lo contrario los resultados serán inesperados. Es decir, si en el ejercicio siguiente los datos son guardados en el orden una cadena de longitud 20 caracteres y un **long**, tendrán que ser recuperados en este orden y con este mismo formato. Sería un error recuperar primero un **long** y después una cadena de longitud 20 caracteres, o recuperar primero un cadena de longitud 15 caracteres y después un **float**.

El siguiente ejemplo lee de la entrada estándar registros del tipo

```
struct r
{
  char referencia[20];
  long precio;
};
```

y los almacena en un fichero llamado *datos*. Una vez creado el fichero, dispondremos de la opción de visualizar el fichero, registro a registro.

El programa completo se muestra a continuación.

```
/*** Escribir y leer datos en un fichero registro a registro ***/
/* fwrite.c
 */
#include <stdio.h>
#include <stdlib.h>

struct r                        /* declaración de un registro */
{
  char referencia[20];
  long precio;
};
typedef struct r registro;      /* tipo registro */

void main()
{
  registro reg;
  int bytesreg = sizeof(reg); /* tamaño de un registro */
  FILE *pf;                   /* puntero al fichero */
  char sprecio[10], respuesta;

  /* Abrir el fichero "datos" para escribir "w" */
  if ((pf = fopen("datos", "w")) == NULL)
  {
    printf("El fichero no puede abrirse.");
    exit(1);
  }
  /* Entrada de datos */
  printf ("Introduzca la marca eof para finalizar\n\n");
  printf ("Referencia:    ");
  while (gets(reg.referencia) != NULL)
  {
    printf("Precio:         ");
    gets(sprecio); reg.precio = atol(sprecio);
    /* Escribir un registro en el fichero */
    fwrite(&reg, bytesreg, 1, pf);
    if (ferror(pf))
    {
      perror("Error durante la escritura");
      exit(2);
    }
    printf("\nReferencia:    ");
  }
  fclose(pf);       /* cerrar el fichero */
  clearerr(stdin); /* desactivar el indicador eof de stdin */
  do
  {
      printf("¿Desea visualizar el fichero? (s/n)  ");
      respuesta = getchar();
      fflush(stdin);
  }
  while (tolower(respuesta) != 's' && tolower(respuesta) != 'n');
```

```
  /* Salida de datos */
  if (respuesta == 's')
  {
    /* abrir el fichero "datos" para leer "r" */
    if ((pf = fopen("datos", "r")) == NULL)
    {
      printf("El fichero no puede abrirse.");
      exit(1);
    }

    /* Leer el primer registro del fichero */
    fread(&reg, bytesreg, 1, pf);
    while (!ferror(pf) && !feof(pf))
    {
      printf("Referencia:    %s\n", reg.referencia);
      printf("Precio:        %ld\n\n", reg.precio);
      /* Leer el siguiente registro del fichero */
      fread(&reg, bytesreg, 1, pf);
    }
  }

  if (ferror(pf))
    perror("Error durante la lectura");
  fclose(pf);
}
```

El proceso de grabar registros en el fichero se centra en la sentencia

```
fwrite(&reg, bytesreg, 1, pf);
```

Cada vez que se ejecuta esta sentencia se graba un registro *reg* en el fichero referenciado por *pf*. Esto implica haber calculado previamente la longitud de un registro (*bytesreg*), haber abierto el fichero referenciado por *pf* para escribir y leer los miembros que componen la estructura *reg*; esto es:

```
reg.referencia;
reg.precio;
```

El hecho de utilizar las funciones **gets** y **atol** para leer y almacenar el dato *precio* de tipo entero, es una alternativa a las funciones **scanf** y **fflush**.

El proceso de leer registros del fichero se centra en la sentencia

```
fread(&reg, bytesreg, 1, pf);
```

Cada vez que se ejecuta esta sentencia se lee el siguiente registro del fichero referenciado por *pf* y se almacena en la estructura *reg*. Esto implica haber calculado previamente la longitud de un registro (*bytesreg*) y haber abierto el fichero referenciado por *pf*. Para manipular los datos leídos, por ejemplo para visualizarlos, simplemente tenemos que acceder a los miembros de la estructura *reg*; es decir, a

```
reg.referencia;
reg.precio;
```

CONTROL DEL BUFFER

Como sabemos, las funciones estándar de E/S utilizan un *buffer* para realizar las operaciones de E/S con el fin de hacerlas más eficientes. También sabemos que un *buffer* es un área de datos en la memoria (RAM) asignada por el programa que abre el fichero. Cuando un fichero no tiene asociado un *buffer*, cada byte escrito en, o leído desde, el fichero es físicamente transferido en el momento de la operación. En cambio, cuando un fichero tiene asociado un *buffer*, todas las operaciones de E/S requeridas son servidas desde ese *buffer*, y la transferencia física de datos se hace en múltiplos del tamaño del *buffer*. Las funciones que vemos a continuación permiten controlar la existencia o no de un *buffer*, así como su tamaño.

setbuf

La función **setbuf** permite al usuario controlar la memoria intermedia asignada al fichero apuntado por *pf*.

```
#include <stdio.h>
void setbuf(FILE *pf, char *buffer);
Compatibilidad: ANSI, UNIX y MS-DOS
```

La función **setbuf** debe ejecutarse una vez abierto el fichero y antes de cualquier operación de lectura o escritura. Si el argumento *buffer* es **NULL**, no se le asigna una memoria intermedia al fichero. En caso contrario, *buffer* es un array de caracteres de longitud *BUFSIZ*, constante definida en *stdio.h*. Esta memoria intermedia será la utilizada en lugar de la asignada por defecto por el sistema.

setvbuf

La función **setvbuf** permite al usuario controlar la existencia o no de un *buffer*, y el tamaño del mismo. Esta función debe ejecutarse una vez abierto el fichero y antes de cualquier operación de lectura o escritura.

```
#include <stdio.h>
int setvbuf(FILE *pf, char *buffer, int tipo, size_t n);
Compatibilidad: ANSI, UNIX y MS-DOS
```

El argumento *buffer* es un array de caracteres de longitud *n* bytes que desempeña la función de *buffer* o memoria intermedia; el argumento *tipo* puede ser:

Tipo	*Significado*
_IOFBF	Se utiliza un *buffer* de tamaño *n* bytes.
_IOLBF	Con MS-DOS, igual que _IOFBF.
_IONBF	No se utiliza una memoria intermedia (*buffer*).

La función **setvbuf** devuelve un 0 si no ocurre un error; y un valor distinto de 0, en caso contrario.

El siguiente programa cuenta las líneas de un fichero de texto. La cuenta la realiza tres veces:

- Utilizando un *buffer* de tamaño *BUFSIZE* bytes.
- Utilizando un *buffer* de tamaño 2048 bytes.
- Sin utilizar un *buffer*.

El programa da como resultado el tiempo, expresado en segundos, que tarda en realizar la cuenta del número de líneas del fichero pasado como argumento en la línea de órdenes.

El programa completo se muestra a continuación.

```
/********** Control del buffer asociado a un fichero **********/
/* setbuf.c
 */
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define MIBUFFER 2048

int cuenta_lineas(FILE *pf);
FILE * abrir(char *);

char buf1[BUFSIZ], buf2[MIBUFFER];  /* buffers para el fichero */

void main(int argc, char *argv[])
{
  time_t t_inicial;
  FILE *pf;
  int  c;

  if (argc != 2)  /* verificar el número de argumentos */
  {
    printf("Sintaxis: nombre_programa nombre_fichero.\n");
    exit(1);
  }

  pf = abrir(argv[1]);
```

```
  /*************************************************************
            Utilizando el buffer buf1, de tamaño BUFSIZ
  *************************************************************/
  setbuf(pf, buf1);
  t_inicial = clock();
  c = cuenta_lineas(pf);
  printf("Tiempo: %5.1f\tTamaño del Buffer: %4d\n",
       ((float)clock()-t_inicial)/CLK_TCK, BUFSIZ);

  pf = abrir(argv[1]);

  /*************************************************************
           Utilizando el buffer buf2, de tamaño MIBUFFER
  *************************************************************/
  setvbuf(pf, buf2, _IOFBF, sizeof(buf2));
  t_inicial = clock();
  c = cuenta_lineas(pf);
  printf("Tiempo: %5.1f\tTamaño del Buffer: %4d\t  mi buffer\n",
       ((float)clock()-t_inicial)/CLK_TCK, MIBUFFER);

  pf = abrir(argv[1]);

  /*************************************************************
                       Sin utilizar un buffer
  *************************************************************/
  setvbuf(pf, NULL, _IONBF, 0);
  t_inicial = clock();
  c = cuenta_lineas(pf);
  printf("Tiempo: %5.1f\tTamaño del Buffer: %4d\n",
       ((float)clock()-t_inicial)/CLK_TCK, 0);

  printf("\nEl fichero %s tiene %d líneas\n", argv[1], c);
}


/*************************************************************
              Contar líneas en un fichero de texto
*************************************************************/
int cuenta_lineas(FILE *pf)
{
  #define N 81
  char linea_buf[N];
  int c = 0;

  while (!ferror(pf) && !feof(pf))
  {
    fgets(linea_buf, N, pf);      /* lee una línea */
    c++;                          /* contar líneas */
  }

  if ( ferror(pf) )
  {
    printf("Ha ocurrido un error de lectura.");
    fclose(pf);
    exit(3);
  }
```

```
  putchar('\n');
  fclose(pf);
  return c;
}

/*****************************************************************
              Abrir el fichero indicado por argv[1]
*****************************************************************/
FILE *abrir(char *fichero)
{
  FILE *pf;
  if ((pf = fopen(fichero, "r" )) == NULL)
  {
    printf("El fichero %s no puede abrirse.\n", fichero);
    exit(2);
  }
  return pf;
}
```

Este programa, en primer lugar, asigna un *buffer* al fichero apuntado por *pf*, de tamaño fijado por el sistema; en segundo lugar, asigna un *buffer* de tamaño fijado por el usuario; y en tercer lugar, no asigna un *buffer* al fichero. La prueba más satisfactoria, es la segunda; en ella el tamaño del buffer es mayor. Esto no significa que cuanto más grande sea el *buffer* más satisfactoria es la respuesta, ya que a partir de un tamaño y en función del sistema que estemos utilizando, las diferencias son mínimas o ninguna.

fflush

La función **fflush** escribe en el fichero apuntado por *pf*, el contenido del *buffer* asociado al mismo. Si el fichero en lugar de estar abierto para escribir está abierto para leer, *fflush* borra el contenido del *buffer*.

```
#include <stdio.h>
int fflush(FILE *pf);
Compatibilidad: ANSI, UNIX y MS-DOS
```

La función **fflush** devuelve el valor 0 si no ocurre un error y un **EOF** en caso contrario.

FICHEROS TEMPORALES

Un fichero temporal es un fichero que sea crea durante la ejecución del programa y se destruye, como muy tarde, al finalizar la ejecución del programa. La función siguiente permite crear ficheros temporales.

tmpfile

La función **tmpfile** crea un fichero temporal. Este fichero es automáticamente borrado cuando el fichero es cerrado o cuando el programa termina normalmente. El fichero temporal es abierto en modo *w+b*.

```
#include <stdio.h>
FILE *tmpfile(void);
Compatibilidad: ANSI, UNIX y MS-DOS
```

La función **tmpfile** devuelve un puntero al fichero temporal creado, o un puntero nulo si no es posible crearlo. Por ejemplo,

```
FILE *pf;
if ((pf = tmpfile()) == NULL)
  printf("No se puede crear un fichero temporal");
```

ACCESO ALEATORIO A UN FICHERO

Hasta ahora, los programas que hemos realizado accedían a los ficheros de forma secuencial; esto es, después de acceder al primer registro, se accedía al segundo, al tercero y así sucesivamente. En cambio el acceso aleatorio permite acceder directamente al registro deseado del fichero, sin necesidad de tener que acceder a los registros que le preceden. Las funciones que proporcionan esta operatividad se estudian a continuación.

fseek

La función **fseek** mueve el puntero de L/E asociado con el fichero apuntado por *pf*, a una nueva localización desplazada *desp* bytes de la posición dada por el argumento *pos*. El desplazamiento *desp* puede ser positivo o negativo.

```
#include <stdio.h>
int fseek(FILE *pf, long desp, int pos);
Compatibilidad: ANSI, UNIX y MS-DOS
```

El argumento *pos* puede tomar alguno de los valores siguientes:

pos	*definición*
SEEK_SET	Principio del fichero
SEEK_CUR	Posición actual del puntero de L/E
SEEK_END	Final del fichero

La función **fseek** devuelve un 0 si no se ha producido un error y un valor distinto de 0 en caso contrario. Por ejemplo,

```
fseek(pfichero, 0L, SEEK_SET);
```

Esta sentencia sitúa el puntero de L/E, al principio del fichero apuntado por *pfichero*. Observar que el desplazamiento es 0 y está expresado como un valor de tipo **long** (L).

> ■ En MS-DOS, para ficheros abiertos en modo texto, **fseek** puede producir un resultado inesperado debido a la conversión de '\n' en CR+LF. Por lo tanto en modo texto, las operaciones con la función **fseek** serán buenas con un desplazamiento 0 (ir al principio o al final) o con un desplazamiento devuelto por la función **ftell** a partir del comienzo del fichero. Para evitar este tipo de problemas, es aconsejable trabajar en modo binario.

ftell

La función **ftell** da como resultado la posición actual del puntero de L/E, dentro del fichero apuntado por *pf*. Esta posición es relativa al principio del fichero.

```
#include <stdio.h>
long ftell(FILE *pf);
Compatibilidad: ANSI, UNIX y MS-DOS
```

La función **ftell** devuelve la posición actual del puntero de L/E, o el valor –1L si ocurre un error. Por ejemplo,

```
long pos;
pos = ftell(pf);
```

Este sentencia almacena en la variable *pos*, la posición actual del puntero de L/E del fichero apuntado por *pf*.

rewind

La función **rewind** pone el puntero de L/E del fichero apuntado por *pf*, al comienzo del mismo.

```
#include <stdio.h>
void rewind(FILE *pf);
Compatibilidad: ANSI, UNIX y MS-DOS
```

Una llamada a esta función equivale a:

```
(void)fseek(pf, 0L, SEEK_SET);
```

con la excepción de que **rewind** desactiva los indicadores de error y de fin de fichero, y **fseek** no.

Un ejemplo de acceso aleatorio

El siguiente programa ilustra el uso de las funciones **fseek**, **ftell** y **rewind**. El programa abre el fichero *datos*, que contiene registros de tipo *registro*, creado por el programa *fwrite.c*. Calcula el número de registros del fichero, presenta en pantalla un determinado registro previamente seleccionado y pregunta si se desea modificar; en caso afirmativo, solicita del teclado los nuevos datos y los almacena en la misma posición dentro del fichero. En cualquier caso, vuelve a preguntar por el número del siguiente registro a modificar; un valor 0, finaliza el programa.

El cálculo del número de registros de un fichero se hace así:

```
fseek(pf, 0L, SEEK_END);
totalreg = ftell(pf,/bytesreg;
```

Primero situamos la posición de L/E al principio del fichero para poder calcular con **ftell** la longitud del fichero y después, dividimos la longitud del fichero entre la longitud de un registro.

Para modificar un registro lo habitual es:

- Preguntar por el número de registro a modificar, situar la posición de L/E en ese registro, leerlo y visualizarlo en la pantalla para asegurarse de que se trata del registro requerido.

```
printf("Nº registro entre 1 y %d (0 para salir): ",totalreg);
scanf("%d", &nreg);

/* Cálculo del desplazamiento. El desplazamiento desde el
   principio al primer registro son 0 bytes. Como el primer
   registro es el 1, tenemos que utilizar la expresión: */
desp = (long)(nreg - 1) * bytesreg;
fseek(pf, desp, SEEK_SET);

fread(&reg, bytesreg, 1, pf);

printf("\nReferencia:  %s\n", reg.referencia);
printf("Precio:      %ld\n\n", reg.precio);
```

- Preguntar si se desea modificar el registro. En caso afirmativo, solicitar del teclado los nuevos datos, situar la posición de L/E en ese registro puesto que la lectura anterior hizo que se avanzara un registro y escribir el registro de nuevo en el fichero.

```
    /* Leer datos */
    printf("\nReferencia:  ");
    gets(reg.referencia);
    printf("Precio:      ");
    gets(sprecio); reg.precio = atol(sprecio);
    /* Posicionarse sobre el registro a reescribir */
    fseek(pf, -bytesreg, SEEK_CUR);
    /* Reescribir el registro con los nuevos datos */
    fwrite (&reg, bytesreg, 1, pf);
```

El programa completo se muestra a continuación.

```
/***************** Acceso aleatorio a un fichero ***************/
/* fseek.c
 */
#include <stdio.h>
#include <stdlib.h>

typedef struct r                    /* definición de un registro */
{
  char referencia[20];
  long precio;
} registro;

void main()
{
  registro reg;                     /* variable de tipo registro */
  int bytesreg = sizeof(reg);          /* tamaño de un registro */
  FILE *pf;                               /* puntero al fichero */
  int totalreg;                     /* número total de registros */
  int nreg;                               /* número de registro */
  long desp;                         /* desplazamiento en bytes */
  int c, respuesta;
  char sprecio[10];

  /* Abrir el fichero "datos" para leer y escribir "r+" */
  if ((pf = fopen("datos", "r+")) == NULL)
  {
    printf("Error: no se puede abrir el fichero\n");
    exit(1);
  }

  /* Calcular el número total de registros del fichero */
  fseek(pf, 0L, SEEK_END);
  totalreg = (int)ftell(pf)/bytesreg;

  /* Presentar un registro en pantalla */
  do
  {
    printf("Nº registro entre 1 y %d (0 para salir): ",totalreg);
    c = scanf("%d", &nreg);
    fflush(stdin);

    /* Visualizar un registro */
    if (c && (nreg >= 1) && (nreg <= totalreg))
    {
```

```
        desp = (long)(nreg - 1) * bytesreg;
        fseek(pf, desp, SEEK_SET);
        fread(&reg, bytesreg, 1, pf);
        if (ferror(pf))
        {
          printf("Error al leer un registro del fichero.\n");
          exit(2);
        }
        printf("\nReferencia:  %s\n", reg.referencia);
        printf("Precio:      %ld\n\n", reg.precio);

        /* Modificar el registro seleccionado */
        do
        {
          printf ("¿Desea modificar este registro? (s/n)  ");
          respuesta = getchar();
          fflush(stdin);
        }
        while (tolower(respuesta != 's') && tolower(respuesta) != 'n');

        if (respuesta == 's')
        {
          printf("\nReferencia:  ");
          gets(reg.referencia);
          printf("Precio:      ");
          gets(sprecio); reg.precio = atol(sprecio);
          /* Escribir un registro en el fichero */
          fseek(pf, -bytesreg, SEEK_CUR);
          fwrite (&reg, bytesreg, 1, pf);
          if (ferror(pf))
          {
            printf("Error al escribir un registro en el fichero.\n");
            exit(3);
          }
        }
      }
    }
    while (nreg);
    fclose(pf);
}
```

EJERCICIOS RESUELTOS

1. Queremos escribir un programa denominado *grep* que permita buscar palabras en uno o más ficheros de texto. Como resultado se visualizará, por cada uno de los ficheros, el nombre del fichero y el número de línea y contenido de la misma, para cada una de las líneas del fichero que contenga la palabra buscada.

   Para formar la estructura del programa se pide:

   a) Escribir una función que busque una cadena de caracteres dentro de otra. El prototipo de esta función será:

```
int BuscarCadena(char *cadena1, char *cadena2);
```

Esta función devolverá un 1 si *cadena2* se encuentra dentro de *cadena1*; en otro caso, devolverá un 0.

b) Escribir una función que busque una cadena de caracteres en un fichero de texto e imprima el número y el contenido de la línea que contiene a la cadena. El prototipo de esta función será:

```
void BuscarEnFich(char *nombref, char *cadena);
```

c) Escribir un programa que utilizando las funciones anteriores permita buscar una palabra en uno o más ficheros. La forma de invocar al programa será así:

```
grep palabra fichero_1 fichero_2 ... fichero_n
```

El programa completo se muestra a continuación.

```
/********** Buscar cadenas de caracteres en ficheros **********/
/* grep.c
 */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int BuscarCadena(char *cadena1, char *cadena2)
{
  /* Buscar la cadena2 en la cadena1 */
  int lon = strlen(cadena2);

  while ( *cadena1 )
  {
    if (strncmp(cadena2, cadena1, lon) == 0)
      return 1; /* se encontró */
    else
      cadena1++;
  }
  return 0;      /* no se encontró */
}

void BuscarEnFich(char *nombref, char *cadena)
{
  FILE *pf;
  char linea[256];
  int nrolinea = 0;

  /* Abrir el fichero nombref */
  if ((pf = fopen(nombref, "r")) == NULL)
  {
    perror(nombref);
```

```
    return;
  }

  /* Buscar cadena en el fichero pf */
  while (fgets(linea, sizeof(linea), pf) != NULL)
  {
    nrolinea++;
    if (BuscarCadena(linea, cadena))
      printf("%s[%d]: %s\n", nombref, nrolinea, linea);
  }
  fclose(pf);
}

void main(int argc, char *argv[])
{
  int i;

  /* Verificar el número de argumentos */
  if (argc < 3)
  {
    printf("Sintaxis: %s palabra f1 f2 ... fn\n", argv[0]);
    exit(-1);
  }

  /* Llamar a la función BuscarEnFich por cada fichero */
  for (i = 2; i < argc; i++)
    BuscarEnFich(argv[i], argv[1]);
}
```

2. En un fichero disponemos de los resultados obtenidos después de medir las temperaturas en un punto geográfico durante un intervalo de tiempo. El fichero consta de una cabecera que se define de acuerdo con la siguiente estructura:

```
struct cabecera
{
  struct posicion
  {
    int grados, minutos;
    float segundos;
  } latitud, longitud; /* Posición geográfica del punto. */
  int total_muestras;
};
```

A continuación de la cabecera vienen especificadas todas las temperaturas. Cada una de ellas es un número **float**. Se pide lo siguiente:

a) Escribir una función con la siguiente declaración:

```
struct temperaturas
{
  int total_temp; /* Tamaño del array de temperaturas. */
```

```
    float *temp; /* Puntero a un array con las temperaturas. */
};
struct temperaturas leer_temperaturas(char *nombref);
```

Esta función recibe el nombre del fichero donde están las temperaturas y devuelve una estructura que indica el total de muestras que hay y un puntero a un bloque de memoria donde se localizan las temperaturas obtenidas del fichero.

b) Escribir una función con la siguiente declaración:

```
float calcular_media(struct temperaturas temp);
```

que calcule la media del conjunto de temperaturas especificado en el parámetro *temp*. Recordamos que la media de un conjunto de muestras se define como sigue,

$$\eta_x = \frac{1}{n}\sum_{i=1}^{n} x_i, \begin{cases} \eta_x, & \text{media de las } x_i \text{ muestras} \\ n, & \text{total muestras} \\ x_i, & \text{cada una de las muestras} \end{cases}$$

c) Escribir una función con la siguiente declaración:

```
float calcular_varianza (struct temperaturas temp);
```

que calcule la varianza del conjunto de temperaturas especificado en el parámetro *temp*. Recordamos que la varianza de un conjunto de muestras se define como sigue,

$$\sigma_x^2 = \frac{1}{n}\sum_{i=1}^{n} (x_i - \eta_x)^2, \begin{cases} \sigma_x^2, & \text{varianza de las } x_i \text{ muestras} \\ \eta_x, & \text{media de las } x_i \text{ muestras} \\ n, & \text{total muestras} \\ x_i, & \text{cada una de las muestras} \end{cases}$$

d) Escribir un programa que pida el nombre de un fichero con el formato anterior y que presente por pantalla la media y la varianza de las temperaturas recogidas.

El programa completo se muestra a continuación.

```
/*************** Estadística de temperaturas ***************/
/* temp.c
 */
#include <stdio.h>
#include <stdlib.h>

struct cabecera  /* tipo del registro de cabecera del fichero */
{
  struct posicion
  {
    int grados, minutos;
    float segundos;
  } latitud, longitud; /* Posición geográfica del punto */

  int total_muestras;
};

struct temperaturas /* tipo de la estructura para almacenar las */
{                   /* temperaturas guardadas en el fichero */
  int total_temp;
  float *temp;
};

/* Almacenar las temperaturas en una estructura de
 * tipo struct temperaturas
 */
struct temperaturas leer_temperaturas(char *nombref)
{
  FILE *pf;
  struct cabecera cab;
  struct temperaturas temp;

  if ((pf = fopen(nombref, "r")) == NULL)
  {
    printf("No se puede abrir el fichero: %s\n", nombref);
    exit(-1);
  }

  /* leer el registro de cabecera */
  fread(&cab, 1, sizeof(struct cabecera), pf);

  /* construir la estructura de temperaturas */
  temp.total_temp = cab.total_muestras;
  temp.temp = (float *)malloc(temp.total_temp * sizeof(float));
  if (temp.temp == NULL)
  {
    printf("Insuficiente memoria.\n");
    exit(-1);
  }
  fread(temp.temp, temp.total_temp, sizeof(float), pf);
  fclose(pf);
  return (temp);
}

/*Media del conjunto de temperaturas */
float calcular_media(struct temperaturas temp)
{
```

```
  float suma = 0;
  int i;

  for (i = 0; i < temp.total_temp; i++)
    suma += temp.temp[i];

  return (suma/temp.total_temp);
}

/* Varianza del conjunto de temperaturas */
float calcular_varianza(struct temperaturas temp)
{
  float suma = 0, media = calcular_media(temp), aux;
  int i;

  for (i = 0; i < temp.total_temp; i++)
  {
    aux = temp.temp [i] - media;
    suma += aux*aux;
  }

  return (suma / temp.total_temp);
}

/* Función principal */
void main()
{
  char nombrefich[30];
  struct temperaturas temp;

  printf("Nombre fichero: ");
  scanf("%s", nombrefich);
  /* Construir la estructura temp */
  temp = leer_temperaturas(nombrefich);
  printf("Temperatura media = %g grados\n", calcular_media(temp));
  printf("Desviación = %g\n", calcular_varianza(temp));

  /* Liberar la memoria asignada */
  free(temp.temp);
}
```

3. Queremos escribir una aplicación para comprimir y descomprimir ficheros binarios con información gráfica.

   Cada fichero gráfico se compone de un conjunto de bytes. Cada byte es un valor de 0 a 255 que representa el nivel de gris de un pixel del gráfico.

   El algoritmo de compresión es el siguiente: "cada secuencia de uno o más bytes *del mismo valor* que haya en el fichero origen se va a representar con dos bytes, de tal forma que el primero representa el nivel de gris leído del fichero origen y el segundo el número total de bytes que hay en la secuencia". Si la secuencia es su-

perior a los 255 bytes, utilizar repetidamente la representación de dos bytes que sean necesarias.

Para implementar esta aplicación se pide:

a) Escribir una función que genere un fichero comprimido a partir de un fichero gráfico, en el que cada registro sea de la forma:

```
typedef struct r
{
  char pixel;
  char total_bytes;
} registro;
```

El prototipo de esta función es:

```
void comprimir(FILE *forigen, FILE *fcomprimido);
```

donde *forigen* es la *stream* que referencia al fichero gráfico que deseamos comprimir y *fcomprimido* es la *stream* que referencia al fichero donde almacenaremos los datos comprimidos.

b) Escribir una función con el prototipo siguiente:

```
void descomprimir(FILE *fcomprimido, FILE *fdestino);
```

para que descomprima los datos que hay en el fichero referenciado por *fcomprimido* y los deposite en el fichero referenciado por *fdestino*.

c) Escribir un programa de nombre *comp* que pueda ser usado de la forma siguiente:

```
comp -c forigen fdestino          para comprimir
comp -d forigen fdestino          para descomprimir
```

El programa completo se muestra a continuación.

```
/************* Comprimir/descomprimir un fichero *************/
/* comp.c
 */
#include <stdio.h>
#include <stdlib.h>

typedef struct r
{
  unsigned char pixel;
```

```
  unsigned char total_bytes;
} registro;

void comprimir( FILE *forigen, FILE *fcomprimido )
{
  unsigned char byte, byteanterior, total;
  registro reg;
  int bytesreg = sizeof(registro);

  /* Leer los bytes del fichero */
  byte = fgetc( forigen );
  while ( !ferror( forigen ) && !feof( forigen ) )
  {
    total = 0;
    byteanterior = byte;

    /* Contar bytes consecutivos repetidos */
    do
    {
      total++;
      byte = fgetc( forigen );
    }
    while ( byteanterior == byte && total < 255 &&
           !ferror( forigen ) && !feof( forigen ) );

    // Escribir el byte y el número de apariciones consecutivas
    reg.pixel = byteanterior;
    reg.total_bytes = total;
    fwrite( &reg, bytesreg, 1, fcomprimido );
  }
  if ( ferror( forigen ) )
    perror( "Error durante la lectura" );
}

void descomprimir( FILE *fcomprimido, FILE *fdestino )
{
  registro reg;
  int bytesreg = sizeof(registro);
  unsigned char i;

  /* Leer los datos del fichero comprimido */
  fread( &reg, bytesreg, 1, fcomprimido );
  while (  !ferror( fcomprimido ) && !feof( fcomprimido ) )
  {
    /* Descomprimir */
    for ( i = 0; i < reg.total_bytes; i++ )
      fputc( reg.pixel, fdestino );
    fread( &reg, bytesreg, 1, fcomprimido );
  }
  if ( ferror( fcomprimido ) )
    perror( "Error durante la lectura" );
}

/* Función principal */
void main( int argc, char *argv[] )
{
  FILE *forigen, *fdestino;
```

```
  if ( argc != 4 )
  {
    printf( "Sintaxis: comp {-c|-d} forigen fdestino\n" );
    exit( 1 );
  }

  if ( (forigen = fopen( argv[2], "r" )) == NULL )
  {
    printf( "El fichero %s no puede abrirse\n", argv[2] );
    exit( 1 );
  }

  if ( (fdestino = fopen( argv[3], "w" )) == NULL )
  {
    printf( "El fichero %s no puede abrirse\n", argv[3] );
    exit( 1 );
  }

  if ( argv[1][0] == '-' && argv[1][1] == 'c' )
    comprimir( forigen, fdestino );
  else if ( argv[1][0] == '-' && argv[1][1] == 'd' )
    descomprimir( forigen, fdestino );
  else
    printf( "Opción incorrecta\n" );

  fclose( forigen );
  fclose( fdestino );
}
```

4. Tenemos grabado en un fichero en disco, denominado "*lista*", la lista de los alumnos de un determinado curso. Cada registro es de la forma:

```
typedef struct
{
  char nombre[61];
  float nota;
  char borrado; /* 'b' = reg. borrado, '\0' = no borrado */
} registro;
```

Se pide, realizar un programa que permita añadir y borrar registros del fichero. Para ello, se creará un fichero denominado "*auxiliar*" con los registros de "*lista*". Sobre el fichero "*auxiliar*" se marcará con una '*b*' en el campo *borrado* los registros que se quieren eliminar. A continuación, a partir del fichero "*auxiliar*" construir de nuevo el fichero "*lista*" con los registros que no están marcados.

Para realizar el proceso descrito:

a) Escribir una función con el prototipo:

```
void CopiarFichero(FILE *fdestino, FILE *forigen);
```

que copie los registros de *forigen* que no tengan una '*b*' en el campo *borrado*, en el fichero referenciado por *fdestino*.

b) Escribir una función con el prototipo:

```
void MarcarRegistro(FILE *pf, int nreg);
```

que permita marcar el registro *nreg* del fichero referenciado por *pf*. Esta función visualizará el registro antes de marcarlo y preguntará si ese es el registro que se desea eliminar; en caso afirmativo asignará el carácter '*b*' al campo *borrado* de ese registro.

c) Escribir una función con el prototipo:

```
void AltaRegistro(FILE *pf);
```

que solicite del teclado los datos para un nuevo registro, pregunte, una vez tecleados, si son correctos y, en caso afirmativo, añada el registro al fichero referenciado por *pf*.

Utilizando las funciones anteriores construir la función **main** para que a través de un menú permita eliminar los registros que deseemos del fichero "*lista*", así como añadir nuevos registros. Así mismo, si cuando se ejecute el programa no existe un fichero *lista*, el programa preguntará si se desea crear un fichero *lista* inicialmente vacío. De esta forma, con la opción "añadir registros" podremos crear el fichero cuando no exista.

El programa completo se muestra a continuación. Observe que la función *CopiarFichero* detecta el final del fichero, no con la función **feof**, sino cuando la función **fread** devuelve un cero, lo que ocurre cuando se hace una lectura y no hay más registros.

La función *MarcarRegistro* visualiza el registro especificado, pregunta si se desea marcar y en caso afirmativo, asigna una '*b*' al campo *borrado* de ese registro y lo vuelve a escribir en el fichero; previamente hay que posicionarse de nuevo en dicho registro.

La función *AltaRegistro* solicita del teclado los datos para un nuevo registro, pregunta si los datos son correctos y en caso afirmativo, sitúa la posición de L/E al final del fichero y escribe el registro.

La función **main** abre el fichero *lista* para leer y si no existe pregunta si se desea crear vacío para después añadir registros; después abre el fichero *auxiliar* para escribir y llama a la función *CopiarFichero* para copiar el fichero *lista* en el fichero *auxiliar*. A continuación, visualiza el menú:

```
b - borrar registro
a - añadir registro
s - salir
```

La opción "borrar registro" llama a la función *MarcarRegistro*, la opción "añadir registro" llama a la función *AltaRegistro* y la opción "salir" pregunta si se desea actualizar el fichero *lista* con los cambios efectuados. En caso afirmativo, se cierran ambos ficheros y se vuelven a abrir, pero ahora el fichero *lista* para escribir y el fichero *auxiliar* para leer y se llama a la función *CopiarFichero* para copiar el fichero *auxiliar* en *lista*, finalizando así el proceso.

```
/*********** Añadir y borrar registros en un fichero **********/
/* lista.c
 */
#include <stdio.h>
#include <stdlib.h>

typedef struct
{
  char nombre[61];
  float nota;
  char borrado; /* 'b' = reg. borrado, '\0' = no borrado */
} registro;

/* Copiar un fichero en otro */
void CopiarFichero(FILE *fdestino, FILE *forigen)
{
  registro reg;

  while (fread(&reg, sizeof(registro), 1, forigen) == 1)
    if (reg.borrado != 'b')
      fwrite(&reg, sizeof(registro), 1, fdestino);
}

/* Visualizar un registro y marcarlo si procede */
void MarcarRegistro(FILE *pf, int nreg)
{
  registro reg;
  char borrado;

  /* Posicionarse en el registro */
  fseek(pf, (nreg - 1)*sizeof(registro), SEEK_SET);

  /* Leer el registro y visualizarlo */
  fread(&reg, sizeof(registro), 1, pf);
  printf("Registro nro. %d\n", nreg);
  printf("Apellidos y Nombre: %s", reg.nombre);
  printf("Nota: %f\n", reg.nota);
```

```
  do
  {
    printf("\n¿Desea borrar el registro? [s/n]: ");
    borrado = getchar();
    fflush(stdin);
  }
  while (borrado != 's' && borrado != 'n');

  /* Marcar el registro si procede */
  reg.borrado = (borrado == 's') ? 'b': 0;
  fseek(pf, (nreg - 1)*sizeof(registro), SEEK_SET);
  fwrite(&reg, sizeof(registro), 1, pf);
}

/* Añadir un registro a un fichero */
void AltaRegistro(FILE *pf)
{
  registro reg;
  char ok;

  printf("Apellidos y Nombre [Máx. 60 caracteres]: ");
  fgets(reg.nombre, 60, stdin);
  printf("Nota: ");
  scanf("%f", &reg.nota);
  fflush(stdin);
  reg.borrado = 0;

  do
  {
    printf("¿Son correctos los datos anteriores? [s/n]: ");
    ok = getchar();
    fflush(stdin);
  }
  while (ok != 's' && ok != 'n');

  if (ok == 's')
  {
    fseek(pf, 0, SEEK_END);
    /* Añadir el registro */
    fwrite(&reg, sizeof(registro), 1, pf);
  }
}

/* Función principal */
void main()
{
  FILE *lista, *auxiliar;
  char opcion;
  int nreg, totalregs, c;

  if ((lista = fopen("lista", "rb")) == NULL)
  {
    perror("lista");
    do
    {
      printf("¿Desea crearlo vacío y añadir registros? [s/n] ");
      opcion = getchar();
```

```
      fflush(stdin);
    }
    while (opcion != 's' && opcion != 'n');
    if (opcion == 's')
    {
      if ((lista = fopen("lista", "wb")) == NULL)
      {
        perror("lista");
        exit(-1);
      }
    }
    else
      exit(-1);
  }
  if ((auxiliar = fopen("auxiliar", "w+b")) == NULL)
  {
    perror("auxiliar");
    exit(-1);
  }

  /* Copia lista en auxiliar */
  CopiarFichero(auxiliar, lista);
  do
  {
    printf("b - borrar registro\n");
    printf("a - añadir registro\n");
    printf("s - salir\n");
    opcion = getchar();
    fflush(stdin);
    switch (opcion)
    {
      case 'b':
        /* Calcular el número total de registros del fichero */
        fseek(auxiliar, 0L, SEEK_END);
        totalregs = (int)ftell(auxiliar)/sizeof(registro);

        /* Marcar un registro */
        printf("Nº registro a borrar entre 1 y %d: ",totalregs);
        c = scanf("%d", &nreg);
        fflush(stdin);

        if (c && (nreg >= 1) && (nreg <= totalregs))
          MarcarRegistro(auxiliar, nreg);
        else
          printf("Nro. no válido\n");
        break;
      case 'a':
        /* Añade registros al fichero auxiliar */
        AltaRegistro(auxiliar);
        break;
    }
  }
  while (opcion != 's');
  do
  {
    printf("Desea salvar los cambios efectuados? [s/n]: ");
    opcion = getchar();
```

```
    fflush(stdin);
  }
  while (opcion != 's' && opcion != 'n');

  if (opcion == 's')
  {
    /* Actualizar lista con los registros de auxiliar */
    fclose(lista);
    fclose(auxiliar);
    if ((lista = fopen("lista", "wb")) == NULL)
    {
      perror("lista");
      exit(-1);
    }

    if ((auxiliar = fopen("auxiliar", "rb")) == NULL)
    {
      perror("auxiliar");
      exit(-1);
    }

    /* Copia auxiliar en lista */
    CopiarFichero(lista, auxiliar);
  }

  fclose(lista);
  fclose(auxiliar);
}
```

EJERCICIOS PROPUESTOS

1. Realizar un programa que permita construir el fichero de temperaturas descrito el problema 2 del apartado "ejercicios propuestos". Para ejecutar este programa se emitirá una orden de la forma:

   ```
   tempw temps.dat
   ```

   donde *tempw* es el nombre del programa y *temps.dat* es el fichero que deseamos construir.

2. Realizar un programa que permita visualizar el fichero de temperaturas construido en el problema 1 anterior. Para ejecutar este programa se emitirá una orden de la forma:

   ```
   tempr temps.dat
   ```

   donde *tempr* es el nombre del programa y *temps.dat* es el fichero que deseamos visualizar.

3. Tenemos un fichero en disco llamado "*alumnos*", donde cada registro es de la forma:

```
typedef struct
{
  unsigned int n_matricula;
  char nombre[40];
  char calificacion[2];
} registro;
```

La calificación viene dada por dos caracteres: *SS* (suspenso), *AP* (aprobado), *NT* (notable) y *SB* (sobresaliente). Realizar un programa que nos imprima el % de los alumnos suspendidos, aprobados, notables y sobresalientes.

4. Tenemos grabados en el disco dos ficheros denominados *alumnos* y *modifi* respectivamente. La estructura de cada uno de los registros para ambos ficheros es la siguiente:

```
struct alumno
{
  char nombre[61];
  float nota;
}
```

Ambos ficheros están clasificados ascendentemente por el campo *nombre*.

En el fichero *modifi* se han grabado las modificaciones que posteriormente realizaremos sobre el fichero *alumnos*. Los registros del fichero *modifi* son:

- registros que también están en el fichero *alumnos* pero que han variado en el campo nota.
- registros nuevos; esto es, registros que no están en el fichero *alumnos*.
- registros que también están en el fichero *alumnos* y que deseamos eliminar. Estos registros se distinguen porque su campo *nota* vale 1.
- Como máximo hay un registro en *modifi* por alumno.

Se pide realizar un programa denominado *actuali* con las funciones que se indican a continuación:

a) Escribir una función que obtenga a partir de los ficheros *alumnos* y *modifi* un tercer fichero siguiendo los criterios de actualización anteriormente descritos. La función prototipo será de la forma:

```
void actualizar(FILE *pfa, FILE *pfb, FILE *pfc);
```

donde *pfa* y *pfb* referencian los ficheros *alumnos* y *modifi*, respectivamente, y *pfc* referencia el fichero resultante de la actualización.

b) Escribir una función principal **main** que llame a la función *actualizar* para obtener el fichero actualizado y a continuación realice las operaciones necesarias para que este fichero quede renombrado con el nombre *alumnos*.

La llamada para ejecutar el programa *actuali* será de la forma:

```
actuali alumnos modifi
```

5. En un fichero disponemos de las cotas de altitud procedentes de la digitalización de un terreno realizada mediante fotografía aérea. Este fichero tiene una cabecera que se ajusta al siguiente formato:

```
typedef struct
{
  unsigned int total_cotas_eje_x;
  unsigned int total_cotas_eje_y;
  float resolucion;                /* Expresada en metros */
} cabecera;
```

El resto de la información del fichero está constituida por las cotas de altitud, en metros, cada una de las cuales es un número **float**.

Por ejemplo, si hemos digitalizado un terreno de 20 Km. de ancho por 10 Km. de largo, con una resolución de 100 m. de separación entre cotas, los datos de la cabecera del fichero serán:

```
total de cotas del eje x = 201
total de cotas del eje y = 101
resolucion = 100.0
```

A continuación vendrán un total de $201 \times 101 = 20301$ cotas de altitud.

Se pide realizar un programa que pida el nombre de un fichero con el formato anterior y presente por pantalla:

- Ancho y largo, en metros, del terreno digitalizado.
- Alturas máxima, mínima y media, en metros, del terreno digitalizado.

CAPÍTULO 10

EL PREPROCESADOR DE C

El *preprocesador* de C es un procesador de texto que manipula el texto de un fichero fuente como parte de la primera fase de compilación y antes de que ésta comience. La parte de texto manipulada se conoce como directrices para el preprocesador.

Una *directriz* es una instrucción para el preprocesador de C. Las directrices son utilizadas para hacer programas fáciles de cambiar y fáciles de compilar en diferentes entornos de ejecución. Las directrices le indican al preprocesador qué acciones específicas tiene que ejecutar. Por ejemplo, reemplazar elementos en el texto, insertar el contenido de otros ficheros en el fichero fuente, o suprimir la compilación de partes del fichero fuente.

El preprocesador de C reconoce las siguientes directrices:

```
#define     #endif      #ifdef      #line
#elif       #error      #ifndef     #pragma
#else       #if         #include    #undef
```

El símbolo **#** debe ser el primer carácter, distinto de espacio en blanco, en la línea que contiene a la directriz; entre el símbolo # y la primera letra de la directriz pueden aparecer caracteres espacio en blanco. Una directriz que ocupe más de una línea física puede ser continuada en una línea siguiente, colocando el carácter \ inmediatamente antes de cambiar a la línea siguiente. Por ejemplo,

```
#define T_INICIAL(descripcion) \
        printf("\n\npara     : %s", #descripcion);\
        inicial = clock();
```

Cualquier texto que siga a una directriz, excepto un argumento o un valor que forma parte de la directriz, tiene que ser incluido como un comentario (/* */).

```
#define T_INICIAL(descripcion) /* descripción es un parámetro */ \
        printf("\n\npara    : %s", #descripcion);\
        inicial = clock();
```

Una directriz puede escribirse en cualquier parte del fichero fuente, pero solamente se aplica desde su punto de definición hasta el final del programa fuente.

DIRECTRIZ #define

La directriz **#define** se utiliza para asociar identificadores con palabras clave, constantes, sentencias y expresiones. Cuando un identificador representa una constante se denomina *constante simbólica*; en cambio, cuando un identificador representa sentencias o expresiones se denomina *macro*. La sintaxis para esta directriz puede ser de dos formas, sin parámetros y con parámetros:

```
#define identificador texto
#define identificador(parámetros) texto
```

La directriz **#define** sustituye todas las apariciones de *identificador* o *identificador(parámetros)* en el fichero fuente por *texto*. *Parámetros*, representa una lista de parámetros formales entre paréntesis y separados por comas, que en la sustitución, serán reemplazados por sus correspondientes parámetros actuales. Entre el *identificador* y el paréntesis abierto no puede haber un espacio en blanco, para no confundir los parámetros con el *texto*.

Para mayor claridad del programa, las constantes simbólicas suelen expresarse en mayúsculas con el fin de distinguirlas de las otras variables.

A continuación se presentan algunos ejemplos para clarificar lo expuesto.

```
#define ANCHO 70
#define LONGITUD (ANCHO + 10)
```

Estas directrices definen una constante simbólica *ANCHO* de valor *70* y otra *LONGITUD* de valor *ANCHO + 10*; esto es, *70 + 10*. Cada aparición de *ANCHO* en el fichero fuente es sustituida por *70*, y cada aparición de *LONGITUD* por *(70 + 10)*. Los paréntesis son importantes, más bien necesarios, para obtener los resultados esperados. Por ejemplo, supongamos la siguiente sentencia:

```
var = LONGITUD * 15;
```

Después de la sustitución sería:

```
var = (70 + 10) * 15;
```

En el caso de no haber utilizado paréntesis en la definición de *LONGITUD*, el resultado sería:

```
var = 70 + 10 * 15
```

que daría lugar a un resultado diferente.

El siguiente es un ejemplo de una macro con dos parámetros:

```
#define MENOR(a, b) ((a) < (b)) ? (a) : (b)
```

Esta directriz define la macro denominada *MENOR*, con el fin de obtener el valor menor de entre los parámetros actuales, cuando se realice la sustitución. Por ejemplo, una ocurrencia en el programa fuente como

```
MENOR(1, 2);
```

donde los parámetros actuales son 1 y 2, sería sustituida por:

```
((1) < (2)) ? (1) : (2);
```

Se insiste, una vez más, en el uso de los paréntesis para evitar resultados inesperados. Por ejemplo,

```
#define MULT(a, b) ((a) * (b))
```

Una ocurrencia como

```
MULT(1 + 2, 3 + 4);
```

sería sustituida por:

```
((1 + 2) * (3 + 4));
```

De no haber puesto *a* y *b* entre paréntesis el resultado sería:

```
(1 + 2 * 3 + 4);
```

que da un valor diferente al esperado.

El trabajo realizado por medio de una macro también puede ser hecho por una función. Es mejor utilizar una función que una macro ya que las funciones no presentan los problemas de las macros parametrizadas como puede ver en el ejemplo siguiente.

```
/*************** Macros y funciones ***************/
/* macros.c
 */
#include <stdio.h>

#define MENOR(x, y) ((x) < (y)) ? (x) : (y)

int menor(int x, int y)
{
  return ((x < y) ? x : y );
}

void main()
{
  int m = 0, a = 10, b = 20;

  /* Utilizando la macro */
  m = MENOR(a--, b--); /* efecto colateral. El valor menor
                        * se decrementa dos veces
                        */
  printf("menor = %d, a = %d y b = %d\n", m, a, b);

  /* Llamando a la función */
  a = 10, b = 20;
  m = menor(a--, b--);
  printf("menor = %d, a = %d y b = %d\n", m, a, b);
}
```

Este ejemplo da lugar al siguiente resultado:

```
menor = 9, a = 8 y b = 19
menor = 10, a = 9 y b = 19
```

Después de la sustitución de la macro quedaría una sentencia como la siguiente:

```
((a--) < (b--)) ? (a--) : (b--);
```

Cuando se ejecuta esta sentencia sucede lo siguiente, en el orden indicado:

1. $((a--) < (b--))$ $\begin{cases} a < b \\ a-- \\ b-- \end{cases}$

2. *Si fue* $a < b$, $m = a--$ $\begin{cases} m = a \\ a-- \end{cases}$

3. *Si no fue* $a < b$, $m = b--$ $\begin{cases} m = b \\ b-- \end{cases}$

Aplicando lo anterior a nuestro ejemplo, se compara 10 y 20, el resultado es menor, y se decrementa *a* a 9 y *b* a 19; como *a* fue menor que *b*, se asigna *a* a *m*, valor 9, y se decrementa *a* a 8. El resultado es *m*=9, *a*=8 y *b*=19.

Macros predefinidas

ANSI C reconoce cinco macros predefinidas. El nombre de cada una de ellas va precedido y seguido por dos guiones de subrayado. Esta macros son:

Macro	*Descripción*
__DATE__	Esta macro es sustituida por una cadena de caracteres de la forma *Mmm dd aaaa* (nombre del mes, día y año). Por ejemplo: `printf("%s\n", __DATE__);`
__TIME__	Esta macro es sustituida por una cadena de caracteres de la forma *hh:mm:ss* (hora, minutos y segundos). Por ejemplo: `printf("%s\n", __TIME__);`
__FILE__	Esta macro es sustituida por el nombre del fichero fuente. Por ejemplo: `printf("%s\n", __FILE__);`
__LINE__	Esta macro es sustituida por el entero correspondiente al número de línea actual. Por ejemplo: `int nlinea = __LINE__;`
__STDC__	Esta macro es sustituida por el valor 1 si la opción */Za* o *-std1* ha sido especificada. Esta opción verifica que todo el código esté conforme a las normas ANSI C. Por ejemplo: `int ansi = __STDC__;`

El operador

El operador # es utilizado solamente con *macros* que reciben argumentos. Cuando este operador precede al nombre de un parámetro formal en la definición de la macro, el parámetro actual correspondiente es incluido entre comillas dobles y tratado como un literal. Por ejemplo:

```
#define LITERAL(s) printf(#s "\n")
```

Una ocurrencia como:

```
LITERAL(Pulse una tecla para continuar);
```

sería sustituida por:

```
printf("Pulse una tecla para continuar" "\n");  equivalente a:
printf("Pulse una tecla para continuar\n");
```

El operador #@

El operador #@ es utilizado solamente con *macros* que reciben argumentos. Cuando este operador precede al nombre de un parámetro formal en la definición de la macro, el parámetro actual correspondiente es incluido entre comillas simples y tratado como un carácter. Por ejemplo:

```
#define CARACTER(c) #@c
```

Una ocurrencia como:

```
car = CARACTER(b);
```

sería sustituida por:

```
car = 'b';
```

El operador

El operador ## al igual que el anterior, también es utilizado con macros que reciben argumentos. Este operador permite la concatenación de dos cadenas. Por ejemplo,

```
#define SALIDA(n) printf("elemento " #n " = %d\n", elemento##n)
```

Una ocurrencia como:

```
SALIDA(1);
```

sería sustituida por:

```
printf("elemento " "1" " = %d\n", elemento1);  equivalente a:
printf("elemento 1 = %d\n", elemento1);
```

DIRECTRIZ #undef

Como su nombre indica, la directriz **#undef** borra la definición de un identificador previamente creado con **#define**. La sintaxis es:

```
#undef identificador
```

La directriz **#undef** borra la definición actual de *identificador*. Consecuentemente, cualquier ocurrencia de *identificador* que pueda aparecer a continuación es ignorada por el preprocesador.

```
#define SUMA(a, b)  (a) + (b)
.
.
.
#undef SUMA  /* la definición de SUMA es borrada */
```

DIRECTRIZ #include

La directriz **#include** le dice al preprocesador que incluya el fichero especificado en el programa fuente, en el lugar donde aparece la directriz. La sintaxis es:

```
#include "fichero"
```

Cuando se utiliza esta sintaxis, el fichero se busca en primer lugar en el directorio actual de trabajo y posteriormente en los directorios estándar definidos. Es útil para incluir ficheros de cabecera, definidos por el usuario en el directorio de trabajo. Otra forma de escribir esta directriz es:

```
#include <fichero>
```

Si se utiliza esta otra forma, el fichero a incluir solamente es buscado en los directorios estándar definidos. Por ejemplo:

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

Estas sentencias añaden los contenidos de los ficheros *misfuncs.h* y *stdio.h* al programa fuente. El fichero *misfuncs.h* es buscado primero en el directorio actual y si no se encuentra, se busca en el directorio estándar definido para los ficheros de cabecera. El fichero *stdio.h* se busca directamente en el directorio estándar.

COMPILACIÓN CONDICIONAL

Las directrices **#if**, **#elif**, **#else** y **#endif** permiten compilar o no partes seleccionadas del fichero fuente. Su sintaxis es la siguiente:

```
#if expresión
  [grupo-de-líneas;]
[#elif expresión 1
   grupo-de-líneas;]
```

```
[#elif expresión 2
   grupo-de-líneas;]
 .
 .
 .
[#elif expresión N
   grupo-de-líneas;]
[#else
   grupo-de-líneas;]
 #endif
```

donde *grupo-de-líneas* representa cualquier número de líneas de texto de cualquier tipo.

El preprocesador selecciona un único *grupo-de-líneas* para pasarlo al compilador. El *grupo-de-líneas* seleccionado será aquel que se corresponda con un valor *verdadero* de la expresión que sigue a **#if** o **#elif**. Si todas las expresiones son falsas, entonces se ejecutará el *grupo-de-líneas* a continuación de **#else**. Por ejemplo:

```
/****** Compilación condicional ******/
/* compcon.c
 */
#include <stdio.h>

#define EEUU 1    /* Estados Unidos */
#define ESPA 2    /* España */
#define FRAN 3    /* Francia */

#define PAIS_ACTIVO ESPA

void main()
{
  #if PAIS_ACTIVO == EEUU
     char moneda[] = "dolar ";
  #elif PAIS_ACTIVO == ESPA
     char moneda[] = "peseta";
  #elif PAIS_ACTIVO == FRAN
     char moneda[] = "franco";
  #endif

  printf("%s", moneda);
}
```

Este programa define un array de caracteres con el nombre de la moneda a utilizar, que es función del valor de la constante simbólica *ESTADO_ACTIVO*. Para el programa expuesto, el compilador compilará la siguiente función **main**:

```
void main()
{
  char moneda[] = "peseta";

  printf("%s", moneda);
}
```

Cada directriz **#if** en un fichero fuente debe emparejarse con su correspondiente **#endif**.

Las *expresiones* deben ser de tipo entero y pueden incluir solamente constantes enteras, constantes de un sólo carácter y el operador **defined**.

Operador defined

El operador **defined** puede ser utilizado en una expresión de constantes enteras, de acuerdo con la siguiente sintaxis:

defined(*identificador***)**

La expresión constante a que da lugar este operador es considerada *verdadera* (distinta de cero) si el *identificador* está actualmente definido y es considerada *falsa*, en caso contrario. Por ejemplo:

```
#if defined(CREDIT)
   credit();
#elif defined(DEBIT)
   debit();
#else
   fnerror();
#endif
```

En este ejemplo, se compila la llamada a la función *credit* si el identificador *CREDIT* está definido; se compila la llamada a la función *debit* si el identificador *DEBIT* está definido; y se compila la llamada a la función *fnerror* si no está definido ninguno de los identificadores anteriores.

CONSTANTE DEFINIDA EN LA ORDEN DE COMPILACIÓN

La opción **–D***id*[=[*valor*]] en UNIX o **/D***id*[=[*valor*]] en MS-DOS, define la constante simbólica *id* para el preprocesador. Si *valor* no se especifica, el valor de *id* se supone que es 1 (*verdadero*). La forma de utilizar esta opción es:

```
cc -DTIEMPO nomprog.c          en UNIX
cl /DTIEMPO nomprog.c          en MS-DOS
```

Como aplicación vamos a diseñar una utilidad sencilla que permita medir el tiempo de ejecución de cualquier parte de nuestro programa C.

En primer lugar crearemos un fichero *tiempo.h* para que contenga las macros *T_INICIAL(descripción)* y *T_FINAL*. Además, diseñaremos el fichero *tiempo.h* para que el código fuente incluido por él en un programa, sólo sea compilado co-

mo parte de dicho programa, si la orden de compilación define la constante simbólica *TIEMPO*; en otro caso, las referencias a *T_INICIAL* y a *T_FINAL* serán nulas. Edite el código que ve a continuación y guárdelo con el nombre *tiempo.h*.

```
/** Definición de las macros: T_INICIAL(descripcion) y T_FINAL **/
/* tiempo.h
 */
#if !defined(TIEMPO_DEFINIDO)
  #if defined(TIEMPO)
    #include <stdio.h>
    #include <time.h>
    clock_t inicial, final;
    #define T_INICIAL(descripcion) \
      printf("\n\npara    : %s", #descripcion);\
      inicial = clock();
    #define T_FINAL final = clock();\
      printf("\ntiempo  : %g seg",\
      (double)(final-inicial)/(double)CLOCKS_PER_SEC);
    #define TIEMPO_DEFINIDO
  #else
    #define T_INICIAL(descripcion)
    #define T_FINAL
  #endif
#endif
```

En primer lugar, observamos la directriz

```
#if !defined(TIEMPO_DEFINIDO)
   ...
#endif
```

Si la constante *TIEMPO_DEFINIDO* está definida, la expresión será falsa y no se tendrá en cuenta el código dentro de la directriz. Esto sucederá cuando se intente compilar este código una segunda vez (por ejemplo, suponga que incluye el fichero *tiempo.h* dos veces).

Si la constante *TIEMPO_DEFINIDO* aún no está definida, el código escrito en el fichero *tiempo.h* será compilado sólo si la constante simbólica *TIEMPO* se ha definido; en nuestro caso, al ejecutar la orden de compilación del programa:

```
#if defined(TIEMPO)
...
#endif
```

Si se cumplen las dos expresiones de las directrices descritas anteriormente, el código siguiente pasará a formar parte del programa fuente:

```
#include <stdio.h>
#include <time.h>
clock_t inicial, final;
#define T_INICIAL(descripcion) \
  printf("\n\npara    : %s", #descripcion);\
```

```
  inicial = clock();
#define T_FINAL final = clock();\
  printf("\ntiempo  : %g seg",\
  (double)(final-inicial)/(double)CLOCKS_PER_SEC);
#define TIEMPO_DEFINIDO
```

Observe el carácter de continuación \, en cada una de las líneas que definen las macros *T_INICIAL* y *T_FINAL*.

En otro caso, alguna de las expresiones mencionadas es falsa, las macros serán definidas para no ejecutar ningún código. Es decir, así:

```
#define T_INICIAL(descripcion)
#define T_FINAL
```

Notar que el argumento *descripción* de la macro *T_INICIAL* sustituye a una cadena de caracteres, ya que utilizamos el operador #.

Un programa que utilice las macros *T_INICIAL* y *T_FINAL*, puede ser el siguiente:

```
/***************** Medir tiempos de ejecución ****************/
/* tiempo.c
 */
#include "tiempo.h"

void main( void )
{
  register unsigned long i;
  float k;

  T_INICIAL(lazo con variable register unsigned long);
    for (i = 0; i < 1000000; i++);
  T_FINAL;

  T_INICIAL(lazo con variable float);
    for (k = 0; k < 1000000; k++);
  T_FINAL;
}
```

Este programa mide el tiempo empleado por dos bucles que se ejecutan un mismo número de veces; uno con una variable entera y otro con una variable real. Cuando ejecute este programa obtendrá una solución similar a la siguiente:

```
para    : lazo con variable register unsigned long
tiempo  : 0 seg

para    : lazo con variable float
tiempo  : 0.28 seg
```

DIRECTRICES #ifdef e #ifndef

La sintaxis correspondiente a estas directrices es:

```
#ifdef identificador
#ifndef identificador
```

La directriz **#ifdef** comprueba si el *identificador* está definido e **#ifndef** comprueba si el *identificador* no está definido.

La directriz **#ifdef** *id* es equivalente a **#if defined**(*id*), y la línea **#ifndef** *id* es equivalente a **#if !defined**(*id*).

Estas directrices simplemente garantizan la compatibilidad con versiones anteriores de C, ya que su función es ejecutada perfectamente por el operador **defined**(*identificador*).

DIRECTRIZ #line

La sintaxis de la directriz **#line** es la siguiente:

```
#line cte-entera ["identificador"]
```

Una línea de la forma indicada pone las macros predefinidas *__LINE__* y *__FILE__* a los valores indicados por *cte-entera* e *identificador* respectivamente, lo cual hace que el compilador cambie su contador interno y su nombre de fichero de trabajo, por los valores especificados en estas constantes. Si se omite el *identificador* (normalmente un nombre de fichero) se utiliza el que tenga la constante *__FILE__* por defecto (el nombre del fichero fuente).

La información proporcionada por la directriz **#line** se utiliza simplemente con el objeto de dar mensajes de error más informativos. El compilador utiliza esta información para referirse a los errores que encuentra durante la compilación. Por ejemplo, cuando se compilen las sentencias

```
#line 30 "en el fichero xxxx, línea "
print("%s\n", __FILE__);
```

se visualizará el siguiente mensaje de error:

```
en el fichero xxxx, línea (30) : warning C4013: 'print' ...
```

Observe cómo se le aplica el número de línea especificado a la sentencia siguiente a la directriz y cómo en el mensaje de error se utiliza la cadena de caracte-

res especificada. A partir de esta sentencia, las demás tendrán números correlativos.

Normalmente el número de línea se refiere a la línea actual que se está compilando y el identificador al nombre del fichero actual. El número de línea se incrementa cada vez que termina de compilarse una línea.

DIRECTRIZ #error

La directriz **#error** es utilizada para abortar una compilación, al mismo tiempo que se visualiza el mensaje de error especificado a continuación de la misma. Su sintaxis es:

#error *mensaje*

Esta directriz tiene utilidad cuando en un programa incluimos un proceso de compilación condicional. Si se detecta una condición anormal, podemos abortar la compilación utilizando esta directriz, al mismo tiempo que se visualiza el mensaje de error especificado. Por ejemplo,

```
#if !defined(PAIS_ACTIVO)
  #error PAIS_ACTIVO no definido.
#endif
```

Cuando se compile el programa fuente que contiene las directrices anteriores, si la macro o constante *PAIS_ACTIVO* no está definida, se emitirá el mensaje de error especificado.

UTILIZACIÓN DE FICHEROS DE CABECERA

La directriz **#include** añade el contenido de un fichero de texto a un programa fuente. Por lo tanto, es útil escribir las definiciones de constantes, macros, declaraciones de variables, tipos de datos definidos por el usuario, prototipos de funciones, etc. susceptibles de ser utilizadas en diferentes programas, en ficheros que después incluiremos por medio de la directriz **#include**. Por ejemplo:

```
/*****************************************************************
                          MODULO PRINCIPAL
*****************************************************************/
/* modppal.c
 *
 * Las tres líneas siguientes incluyen los ficheros especificados
 */
#include "declara2.h"
#include "declara1.h"
#include <stdio.h>
```

```
void main()          /* FUNCIÓN PRINCIPAL */
{
  char mensaje[25];
  TipoReg registro;

  registro = LeerRegistro();
  Verificar(registro, mensaje);
  putchar('\n');
  puts(mensaje);
}
```

Este programa invoca a las funciones *LeerRegistro* y *Verificar*. Por lo tanto, es necesario especificar previamente los prototipos de dichas funciones. Esto es justamente lo que se hace al incluir el fichero *declara1.h*.

```
/*****************************************************************
                          DECLARACIONES
*****************************************************************/
/* declara1.h
 */
#if !defined(_DECLARA1_H)
#define _DECLARA1_H  1

TipoReg LeerRegistro(void);
void Verificar(TipoReg registro, char *mensaje);

#endif // _DECLARA1_H
```

Los prototipos de las funciones pasarán a formar parte del código del programa sólo si aún no han sido incluidos. Esto es lo que hacen las directrices:

```
#if !defined(_DECLARA1_H)
#define _DECLARA1_H  1
...
#endif
```

Si al incluir este fichero en nuestro programa la constante *_DECLARA1_H* no está definida, la definimos y los prototipos de las funciones pasan a formar parte de nuestro programa. Si ahora intentáramos incluir de nuevo el fichero *declara1.h*, nos encontraríamos con que la constante *_DECLARA1_H* ya está definida, la expresión de la directriz **#if** sería falsa, y no se incluirían de nuevo los prototipos de las funciones. Como ve, esto es una medida de seguridad para evitar durante la fase de compilación (no durante la fase de enlace) que las declaraciones contenidas en un fichero puedan ser incluidas dos veces en un mismo módulo, lo que daría lugar a un error de redefinición. Este caso podría darse, por ejemplo, cuando dos o más ficheros incluidos en un módulo, incluyen a su vez un mismo fichero.

También, observamos que los prototipos de las funciones *LeerRegistro* y *Verificar* utilizan el tipo definido por el usuario *TipoReg*. Por lo tanto, es necesa-

rio incluir la declaración del tipo *TipoReg*, previamente a los prototipos de las funciones. Esta declaración se localiza en el fichero *declara2.h*.

```
/*****************************************************************
                          DECLARACIONES
*****************************************************************/
/* declara2.h
 */
#if !defined(_DECLARA2_H)
#define _DECLARA2_H  1

typedef struct
{
  char denominacion[30];
  int existencias;
} TipoReg;

#endif // _DECLARA2_H
```

Para que el programa pueda ejecutarse, necesitamos también el código correspondiente a las funciones invocadas. Este código lo proporciona el fichero *funcs.c* que será enlazado al módulo principal durante la fase de enlace que se realiza para obtener el módulo ejecutable (vea en el Capítulo 3, el apartado "Programa C formado por múltiples ficheros").

```
/*****************************************************************
                            FUNCIONES
*****************************************************************/
/* funcs.c
 *
 * La siguiente línea incluye el fichero especificado
 */
#include "declara2.h"
#include <string.h>
#include <stdio.h>

void LeerRegistro()
{
  printf("Denominación "); gets(registro.denominacion);
  printf("Existencias  "); scanf("%d", &registro.existencias);
}

void Verificar(char *mensaje)
{
  if (registro.existencias < 5)
    strcpy(mensaje, "Por debajo de mínimos");
  else
    strcpy(mensaje, "Por encima de mínimos");
}
```

A su vez, las definiciones de las funciones *LeerRegistro* y *Verificar* necesitan de la declaración del tipo *TipoReg*. Esta declaración está contenida en el fichero *declara2.h*, razón por la que hemos incluido este fichero en el módulo *funcs.c*.

EJERCICIOS RESUELTOS

1. Realizar un programa que solicite introducir un carácter por el teclado y dé como resultado el carácter reflejado en binario. La solución del problema será análoga a la siguiente.

```
Introduce un carácter: A
carácter A, ASCII 41h, en binario: 01000001

Carácter reflejado:
carácter é, ASCII 82h, en binario: 10000010
```

La estructura del programa constará de las funciones siguientes:

a) Una función principal **main** que llamará a una función *presentar* para visualizar el carácter introducido y el reflejado de la forma expuesta anteriormente (simbólicamente, en hexadecimal y en binario) y a una macro *ReflejarByte* que invierta el orden de los bits (el bit 0 pasará a ser el bit 7, el bit 1 pasará a ser el bit 6, el bit 2 pasará a ser el bit 5, etc).

b) Una función *presentar* con el prototipo siguiente:

```
void presentar( unsigned char c );
```

Esta función recibirá como parámetro el carácter que se quiere visualizar y lo presentará simbólicamente, en hexadecimal y en binario.

c) Una macro *ReflejarByte*:

```
#define ReflejarByte( b )
```

Esta macro recibirá como parámetro un byte *b* y dará como resultado el byte reflejado.

El programa completo se muestra a continuación.

```
/*************** Reflejar un byte ***************/
/* refejar.c
 */
#include <stdio.h>
#define ReflejarByte( c )\
         ((((c)&0x01) << 7) | (((c)&0x02) << 5) |\
          (((c)&0x04) << 3) | (((c)&0x08) << 1) |\
          (((c)&0x10) >> 1) | (((c)&0x20) >> 3) |\
          (((c)&0x40) >> 5) | (((c)&0x80) >> 7))
```

```
void Visualizar( unsigned char c );

void main()
{
  unsigned char c;

  printf("Introduce un carácter: ");
  c = getchar();
  Visualizar(c);

  printf("\nCarácter resultante:\n");
  c = ReflejarByte(c);
  Visualizar(c);
}

void Visualizar( unsigned char c )
{
  int i = 0;

  for (i = 7; i>=0; i--)
    printf("%d", (c & (1 << i)) ? 1 : 0);
  printf("\n");
}
```

EJERCICIOS PROPUESTOS

1. Escribir un programa que permita cifrar un fichero de texto, de acuerdo con las siguientes especificaciones.

   El programa se invocará desde la línea de órdenes así:

   ```
   cifrar -c clave
   ```

   donde *–c clave* indica al programa cuál es la clave que se va a emplear para realizar el cifrado. La clave será un valor entero entre 0 y 255.

   El texto a cifrar se leerá del fichero estándar de entrada y el texto cifrado se visualizará en el fichero estándar de salida.

   El cifrado del texto se realizará byte a byte utilizando el siguiente algoritmo:

   - Se calculará la OR exclusiva entre los bytes de entrada y la *clave*.
   - Los bytes resultantes de la operación anterior se reflejarán; esto es, el bit 0 pasará a ser el.bit 7, el bit 1 pasará a ser el bit 6, el bit 2 pasará a ser el bit 5, etc.
   - Los bytes resultantes de la operación anterior serán complementados a 1 y estos serán los bytes cifrados.

Por ejemplo, si el byte de entrada es *b = 0x9a* (10011010) y la clave es *0x49* (01001001) el proceso sería:

```
b XOR c:              (10011010) XOR (01001001) = (11010011)
Reflejar              (11001011)
Complemento a 1       (00110100)
```

El byte cifrado resultante es *0x34*.

Se pide realizar un programa denominado *cifrar* con las macros y funciones que se indican a continuación:

a) Escribir una macro *ReflejarByte*:

```
#define ReflejarByte( b )
```

Esta macro recibirá como parámetro un *byte* y dará como resultado el byte reflejado.

b) Escribir una función *cifrar* de acuerdo con el siguiente prototipo:

```
unsigned char cifrar(unsigned char byte, unsigned char clave);
```

Esta función recibe como parámetros el *byte* a cifrar y la *clave* y devuelve como resultado el *byte* cifrado.

c) Escribir una función *descifrar* de acuerdo con el siguiente prototipo:

```
unsigned char descifrar(unsigned char byte, unsigned char clave);
```

Esta función recibe como parámetros el *byte* cifrado y la *clave* y devuelve como resultado el *byte* sin cifrar.

d) Escribir una función **main** que utilizando las funciones anteriores permita cifrar o descifrar un texto.

¿Cómo invocaría desde la línea de órdenes del sistema operativo al programa *cifrar* para redirigir la E/S y trabajar con ficheros distintos a los estándar?

2. Coger un programa cualquiera de los realizados hasta ahora y añadir el código que supuestamente se necesitaría para visualizar resultados intermedios en un proceso de depuración de dicho programa.

El programa se invocará desde la línea de órdenes así:

```
nombre_programa -DDEBUG          en UNIX
nombre_programa /DDEBUG          en MS-DOS
```

Por ejemplo, la siguiente función incluye código de depuración que sólo se compilará en el caso de que al ejecutar el programa se especifique la constante simbólica *DEBUG*.

```
void Visualizar( unsigned char c )
{
  int i = 0;

  for (i = 7; i>=0; i--)
  {
    #if defined(DEBUG)
      printf("\ni = %d ", i);
    #endif
    printf("%d", (c & (1 << i)) ? 1 : 0);
  }
  printf("\n");
}
```

CAPÍTULO 11

ESTRUCTURAS DINÁMICAS DE DATOS

La principal característica de las *estructuras dinámicas* es la facultad que tienen para variar su tamaño y hay muchos problemas que requieren de este tipo de estructuras. Esta propiedad las distingue claramente de las estructuras estáticas fundamentales (arrays y estructuras). Por tanto, no es posible asignar una cantidad fija de memoria para una estructura dinámica, y como consecuencia un compilador no puede asociar direcciones explícitas con las componentes de tales estructuras. La técnica que se utiliza más frecuentemente para resolver este problema consiste en realizar una asignación dinámica de memoria; es decir, asignación de memoria para las componentes individuales, al tiempo que son creadas durante la ejecución del programa, en vez de hacer la asignación durante la compilación del mismo.

Cuando se trabaja con estructuras dinámicas, el compilador asigna una cantidad fija de memoria para mantener la dirección del componente asignado dinámicamente, en vez de hacer una asignación para el componente en sí. Esto implica que debe haber una clara distinción entre datos y referencias a datos y que consecuentemente se deben emplear tipos de datos cuyos valores sean punteros o referencias a otros datos.

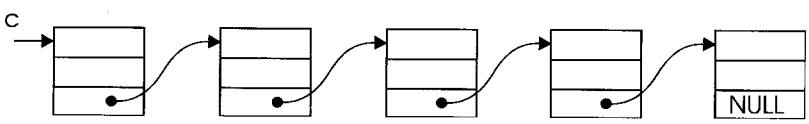
Cuando se asigna memoria dinámicamente para un objeto de un tipo cualquiera, se devuelve un puntero a la zona de memoria asignada. Para realizar esta operación disponemos en C de la función *malloc* (vea en el capítulo 7, el apartado "Asignación dinámica de memoria").

LISTAS LINEALES

Hasta ahora hemos trabajado con arrays estáticos y dinámicos. El espacio para los primeros es asignado durante la compilación y para los segundos durante la ejecución. A diferencia de un array estático, un array dinámico permite alterar su tamaño utilizando la función **realloc**; la reasignación, igual que la asignación, requiere de un número de bytes consecutivos en memoria igual al tamaño del bloque a reasignar y además, copiar en él los datos de la parte que se conserva del array, que será todo el array cuando el espacio reasignado sea más grande.

Si lo que deseamos es una lista de elementos u objetos de cualquier tipo, originalmente vacía, que durante la ejecución del programa vaya creciendo o decreciendo elemento a elemento, según las necesidades previstas en el programa, la construcción idónea es una *lista lineal.*

En una lista lineal cada elemento apunta al siguiente; esto es, cada elemento tiene información de dónde está el siguiente. Por este motivo, una lista lineal se denomina también *lista enlazada.* Esto permite crear cada elemento individualmente, lo cual hace que no sea necesario disponer de un número de bytes consecutivos en memoria igual al tamaño total de la lista lineal, que por otra parte no conocemos a priori. Gráficamente una lista enlazada podemos imaginarla así:



Lista lineal

Para construir una lista lineal primero tendremos que definir la clase de elementos que van a formar parte de la misma. De una forma genérica el tipo de cada uno de los elementos será de la forma:

```
typedef struct s
{
  /* declaración de los miembros de la estructura */
  struct s *siguiente; /* puntero al siguiente elemento */
} telemento;
```

La declaración anterior declara el tipo *telemento*, sinónimo de **struct** *s*. El miembro *siguiente* lo hemos declarado de tipo puntero a **struct** *s* y no de *telemento* porque en este instante, *telemento* aún está por definir. La razón de incluir este miembro en cada uno de los elementos de la lista, es para permitir que cada uno de ellos puede referenciar a su sucesor, formando así una lista enlazada.

A continuación asignaremos memoria para un nuevo elemento. Supongamos que para esta operación escribimos una función *NuevoElemento* así:

```
telemento *NuevoElemento(void)
{
  telemento *q =((telemento *)malloc(sizeof(telemento)));
  if (!q) error();
  return (q);
}
```

Esta función asigna memoria para un objeto de tipo *telemento* y devuelve un puntero al espacio de memoria asignado. Si la asignación de memoria no se puede realizar, entonces la función *NuevoElemento* invoca a la función *error*, la cual puede ser de la forma siguiente:

```
void error(void)
{
  perror("Insuficiente memoria");
  exit(1);
}
```

La función *error* visualiza un mensaje indicando que no hay memoria suficiente para la asignación y finaliza el programa.

Una vez definida la función *NuevoElemento*, la asignación de memoria para un elemento se haría así:

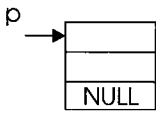
```
void main(void)
{
  telemento *p; /* puntero a un elemento */

  /* Asignar memoria para un elemento */
  p = NuevoElemento();
  p->siguiente = NULL;

  /* Operaciones */

  /* Liberar memoria */
  free(p);
}
```

La declaración *telemento *p* declara un puntero *p* a un objeto de tipo *telemento*. La sentencia *p = NuevoElemento()* crea (asigna memoria para) un objeto de tipo *telemento*, genera un puntero (dirección de memoria) que referencia este nuevo objeto y asigna esta dirección a la variable *p*. La sentencia *p–>siguiente = NULL* asigna al miembro *siguiente* del objeto apuntado por *p* el valor *NULL*, indicando así que después de este elemento no hay otro; esto es, que este elemento es el último de la lista.

El valor **NULL**, dirección nula, permite crear estructuras de datos finitas. Así mismo, suponiendo que *p* apunta al principio de la lista, diremos que dicha lista está vacía si *p* vale **NULL**. Por ejemplo, después de ejecutar las sentencias:

```
p = NULL; /* lista vacía */
p = NuevoElemento();
p->siguiente = NULL;
```

tenemos una lista de un elemento:



Para añadir un nuevo elemento a la lista, procederemos así:

```
q = NuevoElemento(); /* crear un nuevo elemento */
q->siguiente = p;    /* enlazar con el elemento apuntado por p */
p = q;               /* p apunta al principio de la lista */
```

donde *q* es un puntero a un objeto de tipo *telemento*. Ahora tenemos una lista de dos elementos. Observe que los elementos nuevos se añaden al principio de la lista. Para verlo con claridad analicemos las tres sentencias anteriores. Partimos de que tenemos una lista apuntada por *p*, en este caso de un elemento. La sentencia

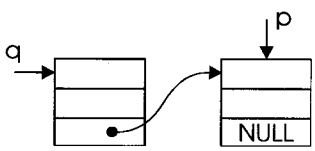
```
q = NuevoElemento();
```

crea un nuevo elemento

La sentencia

```
q->siguiente = p;
```

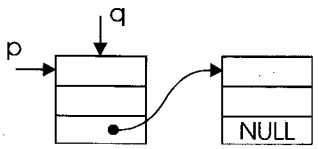
hace que el sucesor del elemento creado sea el anteriormente creado. Recuerde que para acceder a un miembro de una estructura referenciada por un puntero, hay que utilizar el operador –>. Observe que ahora *q–>siguiente* y *p* tienen el mismo valor; esto es, la misma dirección, por lo tanto, apuntan al mismo elemento.



Por último, la sentencia

```
p = q;
```

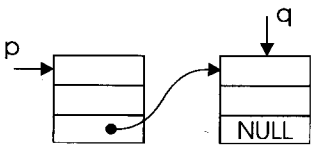
hace que la lista quede de nuevo apuntada por *p*; es decir, para nosotros *p* es siempre el primer elemento de la lista.



Ahora *p* y *q* apuntan al mismo elemento, al primero. Si ahora se ejecutara una sentencia como la siguiente ¿qué sucedería?

```
q = q->siguiente;
```

¿Quién es *q–>siguiente*? Es el miembro *siguiente* de la estructura apuntada por *q* que contiene la dirección de memoria donde se localiza el siguiente elemento al apuntado por *q*. Si este valor se lo asignamos a *q*, entonces *q* apuntará al mismo elemento que *q–>siguiente*. El resultado es que *q* apunta ahora al siguiente elemento como puede ver en la figura.



Esto nos da una idea de cómo avanzar elemento a elemento sobre una lista. Si ejecutamos de nuevo la misma sentencia

```
q = q->siguiente;
```

¿Qué sucede? Sucede que como *q–>siguiente* vale **NULL**, a *q* se le ha asignado el valor **NULL**. Conclusión, cuando utilizamos un puntero para ir de elmento en elmento de una lista, en el ejemplo anterior *q*, diremos que hemos llegado al final de la lista cuando *q* toma el valor **NULL**.

OPERACIONES BÁSICAS

Las operaciones que podemos realizar con listas incluyen fundamentalmente las siguientes:

1. Insertar un elemento en una lista.

2. Borrar un elemento de una lista.
3. Recorrer los elementos de una lista.
4. Borrar todos los elementos de una lista.
5. Buscar un elemento en una lista.

Partiendo de las declaraciones:

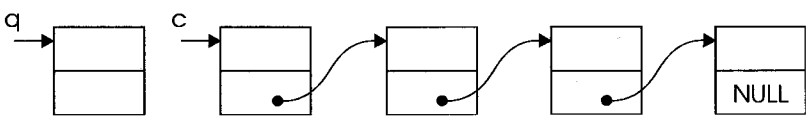
```
typedef struct datos telemento; /* declaración del tipo elemento*/
struct datos
{
  int dato;
  telemento *siguiente;
};

telemento *c, *p, *q;
```

en los siguientes apartados se indica la forma de realizar estas operaciones. Observe que por sencillez vamos a trabajar con una lista de enteros.

Inserción de un elemento al comienzo de la lista

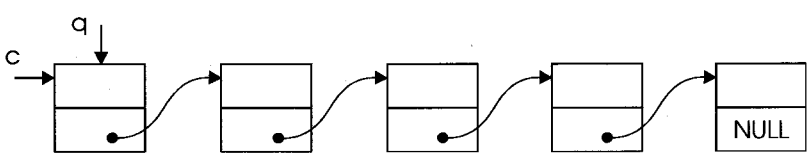
Supongamos una lista lineal apuntada por *c*. Para insertar un elemento al principio de la lista, primero se crea el elemento y después se reasignan los punteros, tal como se indica a continuación:

```
q = NuevoElemento();
```




```
q->dato = n;          /* asignación de valores */
q->siguiente = c;     /* reasignación de punteros */
c = q;
```

El orden en el que se realizan estas operaciones es esencial. El resultado es



Este concepto nos sugiere cómo *crear una lista*. Para ello, y partiendo de una lista vacía, no tenemos más que repetir la operación de insertar un elemento al comienzo de una lista. Veámoslo a continuación:

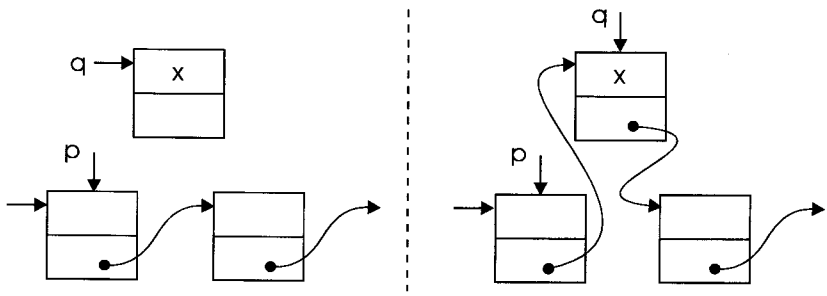
```
elemento *c, *q;
int n;

/* Crear una lista de enteros */
printf("Introducir datos. Finalizar con eof\n\n");
c = NULL; /* lista vacía */
printf("dato: ");
while (scanf("%d", &n) != EOF)
{
  q = NuevoElemento();
  q->dato = n;
  q->siguiente = c;
  c = q;
  printf("dato: ");
}
```

Notar que el orden de los elementos en la lista, es el inverso del orden en el que han llegado.

Inserción de un elemento en general

La inserción de un elemento en la lista, a continuación de otro elemento cualquiera apuntado por *p*, es de la forma siguiente:

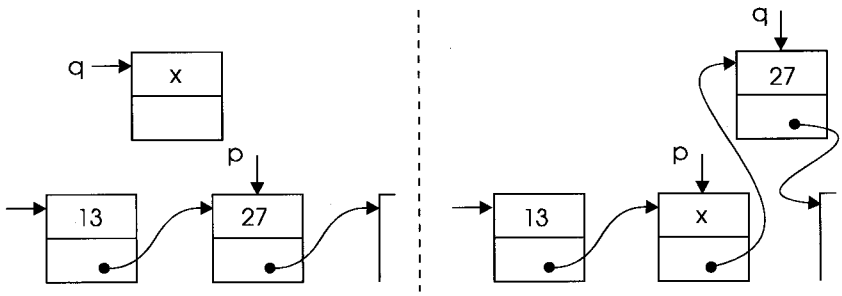
```
q = NuevoElemento();
q->dato = x;          /* valor insertado */
q->siguiente = p->siguiente;
p->siguiente = q;
```



Inserción en la lista detrás del elemento apuntado por p

La inserción de un elemento en la lista antes de otro elemento apuntado por *p*, se hace insertando un nuevo elemento detrás del elemento apuntado por *p*, intercambiando previamente los valores del nuevo elemento y del elemento apuntado por *p*.

```
q = NuevoElemento();
*q = *p;        /* copia miembro a miembro una estructura en otra */
p->dato = x;    /* valor insertado */
p->siguiente = q;
```

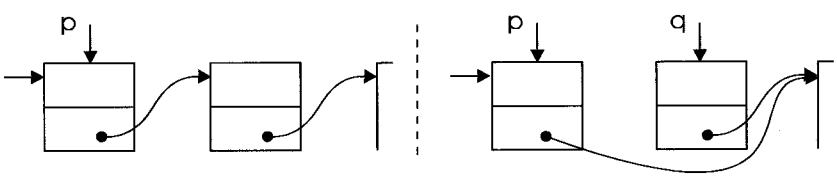


Inserción en la lista antes del elemento apuntado por p

Borrar un elemento de la lista

Para borrar el sucesor de un elemento apuntado por *p*, las operaciones a realizar son las siguientes:

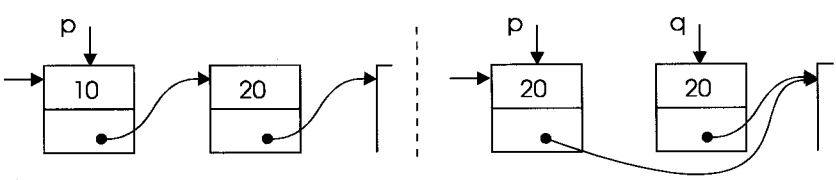
```
q = p->siguiente;                    /* q apunta al elemento a borrar */
p->siguiente = q->siguiente;         /* enlazar los elementos anterior y
                                        posterior al borrado */
free(q);                        /* liberar la memoria ocupada (borrar) */
```



Borrar el sucesor del elemento apuntado por p

Observe que para acceder a los miembros de un elemento, éste tiene que estar apuntado por un puntero. Por esta razón, lo primero que hemos hecho ha sido apuntar el elemento a borrar por *q*.

Para borrar un elemento apuntado por *p*, las operaciones a realizar son las siguientes:



Borrar el elemento apuntado por p

```
q = p->siguiente;
*p = *q;          /* copia miembro a miembro una estructura en otra */
free(q);
```

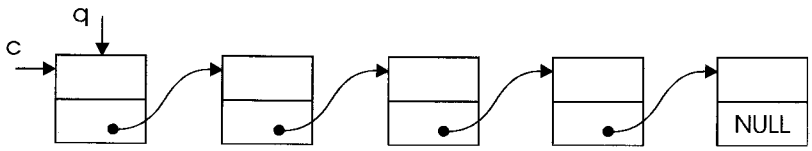
La sentencia *p = *q copia el contenido de la estructura apuntada por *q*, miembro a miembro, en la estructura apuntada por *p*.

Como ejercicio, escribir la secuencia de operaciones que permitan borrar el último elemento de una lista.

Recorrido de una lista

Supongamos que hay que realizar una operación con todos los elementos de una lista, cuyo primer elemento está apuntado por *c*. Por ejemplo, escribir el valor de cada elemento de la lista. La secuencia de operaciones es la siguiente:

```
q = c;    /* salvar el puntero que referencia la lista */
while (q != NULL)
{
  printf("%d  ", q->dato);
  q = q->siguiente;
}
```



Borrar todos los elementos de una lista

Borrar todos los elementos de una lista equivale a liberar la memoria ocupada por cada uno de los elementos de la misma. Supongamos que queremos borrar una lista, cuyo primer elemento está apuntado por *c*. La secuencia de operaciones es la siguiente:

```
q = c;                  /* q apunta al primer elemento de la lista */
while (q != NULL)
{
  c = c->siguiente;              /* c apunta al siguiente elemento */
  free(q);      /* liberar la memoria del elemento apuntado por q */
  q = c;                        /* q apunta al mismo elemento que c */
}
```

Observe que antes de borrar el elemento apuntado por *q*, hacemos que *c* apunte al siguiente elemento, porque si no perderíamos el resto de la lista; la referenciada por *q–>siguiente*.

Buscar en una lista un elemento con un valor x

La búsqueda es secuencial y termina cuando se encuentra el elemento, o bien, cuando se llega al final de la lista.

```
q = c;
printf("valor : "); scanf("%d", &x);
while (q != NULL && q->dato != x)
  q = q->siguiente;
```

Observe el orden de las expresiones que forman la condición del bucle **while**. Sabemos que en una operación AND, cuando una de las expresiones es falsa la condición ya es falsa, por lo que el resto de las expresiones no necesitan ser evaluadas. De ahí que cuando *q* valga **NULL**, la expresión *q–>dato* no tiene sentido que sea evaluada. En el caso de que fuera evaluada sería interpretada como un valor cero.

UN EJEMPLO CON LISTAS LINEALES

Como aplicación, realizaremos a continuación un programa que permita crear una lista lineal clasificada ascendentemente, en la que cada elemento conste de dos miembros: un entero y un puntero a un elemento del mismo tipo.

```
typedef struct datos /* elemento de una lista de enteros */
{
  int dato;
  struct datos *siguiente;
} telemento;
```

El programa incluirá las siguientes funciones:

1. Añadir un elemento. Esta función comprenderá dos casos: insertar un elemento al principio de la lista o insertar un elemento después de otro.

   ```
   void anadir(telemento **, int);
   ```

   La función *añadir* recibirá como parámetros la dirección del primer elemento de la lista, parámetro que será pasado por referencia puesto que puede variar cuando se inserte un elemento al principio, y el entero a insertar.

2. Borrar un elemento de la lista. Esta función comprenderá dos casos: borrar el elemento del principio de la lista o borrar otro elemento cualquiera.

   ```
   void borrar(telemento **, int);
   ```

La función *borrar* recibirá como parámetros la dirección del primer elemento de la lista, parámetro que será pasado por referencia puesto que puede variar cuando se borre el primer elemento, y el entero a borrar.

3. Buscar un elemento en la lista.

```
telemento *buscar(telemento *, int);
```

La función *buscar* recibirá como parámetros la dirección del primer elemento de la lista y el entero a buscar y devolverá como resultado la dirección del elemento si se encuentra, o un valor **NULL** si no se encuentra.

4. Visualizar el contenido de la lista.

```
void visualizar(telemento *);
```

La función *visualizar* recibirá como parámetros la dirección del primer elemento de la lista.

5. Presentar un menú con cada una de las operaciones anteriores.

```
int menu(void);
```

La función *menu* presentará un menú con las opciones: *añadir*, *borrar*, *buscar*, *visualizar* y *salir* y devolverá como resultado un entero correspondiente a la opción elegida.

Empecemos a desarrollar el problema, declarando el tipo de los elementos de la lista, escribiendo las funciones *NuevoElemento* y *error* comentadas al hablar de listas lineales, para a continuación escribir la función **main**.

La función **main** presentará el menú invocando a la función *menu* y según la opción elegida, llamará a la función *añadir*, *borrar*, *buscar* o *visualizar*, o si se eligió la opción *salir*, finalizará el programa. Antes de salir del programa, como último paso, liberaremos la memoria ocupada por la lista.

```
/******************* Operaciones con listas ******************/
/* lista1.c
 */
#include <stdio.h>
#include <stdlib.h>

/* Lista simplemente enlazada.
 * Cada elemento contiene un número entero */
typedef struct datos      /* elemento de una lista de enteros */
{
  int dato;
```

```
  struct datos *siguiente;
} telemento;

/* Funciones */
void error(void)
{
  perror("error: insuficiente espacio de memoria");
  exit(1);
}

telemento *NuevoElemento()
{
  telemento *q = (telemento *)malloc(sizeof(telemento));
  if (!q) error();
  return(q);
}

/* Funciones prototipo */
int menu(void);
void anadir(telemento **, int);
void borrar(telemento **, int);
telemento *buscar(telemento *, int);
void visualizar(telemento *);

void main()                                  /* función principal */
{
  telemento *cabecera = NULL;                      /* lista vacía */
  telemento *q;
  int opcion, dato;

  while (1)
  {
    opcion = menu();
    switch (opcion)
    {
      case 1:
        printf("añadir dato: ");
        scanf("%d", &dato);
        anadir(&cabecera, dato);
        break;
      case 2:
        printf("borrar dato: ");
        scanf("%d", &dato);
        borrar(&cabecera, dato);
        break;
      case 3:
        printf("buscar dato: ");
        scanf("%d", &dato);
        q = buscar(cabecera, dato);
        if (q)
          printf("El valor %d está en la lista\n", q->dato);
        else
          printf("El valor %d no está en la lista\n", dato);
        break;
      case 4:
        visualizar(cabecera);
```

```
        break;
      case 5:
        /* Liberar la memoria ocupada por la lista */
        q = cabecera; /*q apunta al primer elemento de la lista*/
        while (q != NULL)
        {
          cabecera = cabecera->siguiente;
          free(q);
          q = cabecera;
        }
        return;
    }
  }
}
```

La función *menu* utiliza un bucle **while** para presentar el menú y obligar al usuario a introducir una de las opciones visualizadas. Cualquier entrada que no se corresponda con las opciones presentadas, dará lugar a que se visualice de nuevo el menú, solicitando una opción correcta.

```
int menu()                                    /* menú de opciones */
{
  int op = 0;
  do
  {
    printf("\n\t1.  Añadir un elemento\n");
    printf("\n\t2.  Borrar un elemento\n");
    printf("\n\t3.  Buscar un elemento\n");
    printf("\n\t4.  Visualizar la lista\n");
    printf("\n\t5.  Salir\n");
    printf("\n\t    Elija la opción deseada ");
    scanf("%d", &op);
    fflush(stdin);
  }
  while ( op < 1 || op > 5 );
  return op;
}
```

La función *añadir* añade un nuevo elemento a la lista en el lugar que le corresponda para que dicha lista quede ordenada ascendentemente. Utiliza una variable local, *cabecera*, que es un puntero al primer elemento de la lista; esta variable es inicializada con la dirección del primer elemento de la lista que se pasa como argumento y que vale **cab*. Puesto que *cabecera* es una variable local, cuando la función finaliza, se hace la operación inversa para actualizar el puntero que apunta al primer elemento; esto es, **cab = cabecera*.

A la hora de insertar un nuevo elemento en la lista, distinguimos tres casos, cada uno de los cuales tiene un tratamiento diferente:

- Que la lista esté vacía.
- Que el elemento haya que insertarlo al principio.

- Que el elemento haya que insertarlo después de otro.

Los dos últimos casos requieren que se busque la posición donde hay que insertar el nuevo elemento, con el fin de que los valores queden ordenados ascendentemente. Para ello, utilizaremos dos punteros auxiliares, *actual* y *anterior*, para apuntar al elemento actual al que estamos accediendo y al anterior a éste, respectivamente. Inicialmente, ambos punteros apuntarán al primer elemento.

```
actual = anterior = cabecera;
while (actual != NULL && dato > actual->dato)
{
  anterior = actual;
  actual = actual->siguiente;
}
```

Si la búsqueda finaliza y los dos punteros siguen valiendo lo mismo, el nuevo elemento hay que insertarlo al principio. En este caso, el bucle **while** se habrá ejecutado cero veces porque *dato* no es mayor que *actual–>dato*. En otro caso, hay que insertarlo después del elemento apuntado por *anterior* debido al criterio de búsqueda expuesto; esto es, el valor del nuevo elemento es mayor que el valor del elemento apuntado por *anterior* y menor o igual que el valor del elemento apuntado por *actual*, excepto cuando el nuevo elemento hay que insertarlo al final de la lista; en este caso, *actual* vale **NULL**.

```
/* Introducir un elemento ordenadamente en la lista */
void anadir(telemento **cab, int dato)
{
  telemento *cabecera = *cab;
  telemento *actual=cabecera, *anterior=cabecera, *q;

  if (cabecera == NULL)   /* Si está vacía, crear un elemento */
  {
    cabecera = NuevoElemento();
    cabecera->dato = dato;
    cabecera->siguiente = NULL;
    *cab = cabecera;
    return;
  }
  /* Entrar en la lista y encontrar el punto de inserción */
  while (actual != NULL && dato > actual->dato)
  {
    anterior = actual;
    actual = actual->siguiente;
  }

  /* Dos casos:
   * 1) Insertar al principio de la lista
   * 2) Insertar después de anterior (incluye insertar al final)
   */
  q = NuevoElemento();          /* se genera un nuevo elemento */
  if (anterior == actual)             /* insertar al principio */
  {
```

```
    q->dato = dato;
    q->siguiente = cabecera;
    cabecera = q;
  }
  else                          /* insertar después de anterior */
  {
    q->dato = dato;
    q->siguiente = actual;
    anterior->siguiente = q;
  }
  *cab = cabecera;
}
```

Si ha comprendido la función *añadir* que acabamos de exponer, analice la función *borrar* que se presenta a continuación, ya que se rige por unos criterios análogos.

```
/* Encontrar un dato y borrarlo */
void borrar(telemento **cab, int dato)
{
  telemento *cabecera = *cab;
  telemento *actual=cabecera, *anterior=cabecera;
  if (cabecera == NULL)
  {
    printf("Lista vacía\n");
    return;
  }

  /* Entrar en la lista y encontrar el elemento a borrar */
  while (actual != NULL && dato != actual->dato)
  {
    anterior = actual;
    actual = actual->siguiente;
  }

  /* Si el dato no se encuentra, retornar */
  if (actual == NULL) return;

  /* Si el dato se encuentra, borrar el elemento */
  if (anterior == actual)    /* borrar el elemento de cabecera */
    cabecera = cabecera->siguiente;
  else                       /* borrar un elemento no cabecera */
    anterior->siguiente = actual->siguiente;

  free(actual);
  *cab = cabecera;
}
```

La función *buscar* utiliza un puntero *actual* que partiendo de la cabecera va avanzando elemento a elemento y verificando si el dato almacenado en el miembro *dato* del elemento apuntado por él, es el buscado. El valor devuelto es *actual*, que será **NULL** cuando el dato buscado no se encuentre en la lista.

```
/* Buscar un elemento determinado en la lista */
```

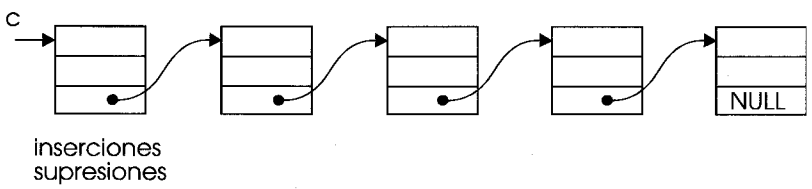
```
telemento *buscar(telemento *cabecera, int dato)
{
  telemento *actual = cabecera;
  while (actual != NULL && dato != actual->dato)
    actual = actual->siguiente;

  return (actual);
}
```

La función *Visualizar* utiliza un puntero *actual* para, partiendo de la cabecera, ir avanzando elemento a elemento e imprimiendo el valor almacenado en el miembro *dato* del elemento apuntado por él.

```
/* Visualizar la lista */
void visualizar(telemento *cabecera)
{
  telemento *actual = cabecera;
  if (cabecera == NULL)
    printf("Lista vacía\n");
  else
  {
    while (actual != NULL)
    {
      printf("%d  ", actual->dato);
      actual = actual->siguiente;
    }
    printf("\n");
  }
}
```

PILAS

Una pila es una lista lineal en la que todas las inserciones y supresiones (y normalmente todos los accesos), se hacen en un extremo de la lista.



Pila

Un ejemplo de esta estructura es una pila de platos. En ella, el añadir o quitar platos se hace siempre por la parte superior de la pila. Este tipo de listas reciben también el nombre de listas *LIFO* (*last in first out* - último en entrar, primero en salir).

Las operaciones de insertar y suprimir en una pila, son conocidas en los lenguajes ensambladores como *push* y *pop* respectivamente. La operación de recuperación de un elemento de la pila, lo elimina de la misma.

Como ejemplo de utilización de una pila, vamos a simular una calculadora capaz de realizar las operaciones de +, –, * y /. La mayoría de las calculadoras aceptan la notación *infija* y unas pocas la notación *postfija*. En estas últimas, para sumar 10 y 20 introduciríamos primero 10, después 20 y por último el +. Cuando se introducen los operandos, se colocan en una pila y cuando se introduce el operador, se sacan dos operandos de la pila, se calcula el resultado y se introduce en la pila. La ventaja de la notación *postfija* es que expresiones complejas pueden evaluarse fácilmente sin mucho código.

La calculadora de nuestro ejemplo utiliza la notación *postfija*, por lo que hemos desarrollado dos funciones: *push* y *pop*. La función *push* introduce un valor en la pila y la función *pop* saca un valor de la pila.

```
void push(telemento **p, double x);
double pop(telemento **p);
```

La función *push* recibirá como parámetros la dirección del primer elemento de la pila, parámetro que será pasado por referencia porque cambia cada vez que se añade un elemento, y el operando a añadir a la pila.

La función *pop* recibirá como parámetro la dirección del primer elemento de la pila, parámetro que será pasado por referencia porque cambia cada vez que se saca un elemento de la pila.

El programa realiza las siguientes operaciones:

1. Lee un dato, operando u operador, y lo almacena en la variable *op*.

2. Analiza *op*; si se trata de un operando lo mete en la pila utilizando la función *push*; y si se trata de un operador saca, utilizando la función *pop*, los dos últimos operandos de la pila, realiza la operación indicada por dicho operador y mete el resultado en la pila para encadenarlo con otra posible operación.

El programa completo se muestra a continuación.

```
/********** Programa calculadora. Aplicación de pilas **********/
/* pila.c
 */
#include <stdio.h>
#include <stdlib.h>
```

```
typedef struct datos telemento;              /* tipo telemento */
struct datos          /* estructura de un elemento de la pila */
{
  double dato;
  telemento *siguiente;
};
/* Funciones */
void error(void)
{
  perror("error: insuficiente espacio de memoria");
  exit(1);
}

telemento *NuevoElemento()
{
  telemento *q = (telemento *)malloc(sizeof(telemento));
  if (!q) error();
  return (q);
}
/* Prototipos de funciones */
void push(telemento **p, double x);/* añadir un dato a la pila */
double pop(telemento **p);          /* sacar un dato de la pila */

void main()  /* función principal */
{
  telemento *q, *cima = NULL; /* pila vacía */
  double a, b;
  char op[81];

  printf("Calculadora con las operaciones: + - * /\n");
  printf("Los datos serán introducidos de la forma:\n");
  printf(">operando 1\n");
  printf(">operando 2\n");
  printf(">operador\n\n");
  printf("Para salir pulse q\n\n");
  do
  {
    printf("> ");
    gets(op);                 /* leer un operando o un operador */
    switch (*op)                   /* *op es el primer carácter */
    {
      case '+':
        b = pop(&cima); a = pop(&cima);
        printf("%g\n", a + b);
        push( &cima, a+b );
        break;
      case '-':
        b = pop(&cima); a = pop(&cima);
        printf("%g\n", a - b);
        push( &cima, a-b );
      break;
      case '*':
        b = pop(&cima); a = pop(&cima);
        printf("%g\n", a * b);
        push( &cima, a*b );
      break;
      case '/':
```

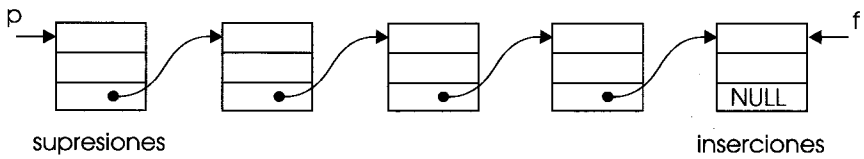
```
            b = pop(&cima); a = pop(&cima);
            if (b == 0)
            {
              printf("\nDivisión por cero\n");
              break;
            }
            printf("%g\n", a / b);
            push(&cima, a/b);
            break;
          default : /* es un operando */
            push(&cima, atof(op));
        }
      }
      while (*op != 'q');

      /* Liberar la memoria ocupada por la pila */
      q = cima;    /* q apunta al primer elemento de la lista */
      while (q != NULL)
      {
        cima = cima->siguiente;
        free(q);
        q = cima;
      }
}

/* Añadir un dato a la pila */
void push(telemento **p, double x)
{
  telemento *q, *cima;
  cima = *p;  /* cima de la pila */
  q = NuevoElemento();
  q->dato = x;
  q->siguiente = cima;
  cima = q;
  *p = cima;
}

/* Recuperar el dato de la cima de la pila */
double pop(telemento **p)
{
  telemento *cima;
  double x;
  cima = *p;  /* cima de la pila */
  if (cima == NULL)
  {
    printf("\nerror: pop de una pila vacía\n");
    return 0;
  }
  else
  {
    x = cima->dato;
    *p = cima->siguiente;
    free(cima);
    return (x);
  }
}
```

El resultado cuando ejecute este programa, es de la forma siguiente:

```
Calculadora con las operaciones: + - * /
Los datos serán introducidos de la forma:
>operando 1
>operando 2
>operador

Para salir pulse q

> 4.2
> 5
> *
21
> 10
> -
11
> q
```

COLAS

Una cola es una lista lineal en la que todas las inserciones se hacen por un extremo y todas las supresiones (y normalmente todos los accesos) se hacen por el otro extremo de la lista. Por ejemplo, una fila en un banco. Este tipo de listas reciben también el nombre de listas *FIFO* (*first in first out* - primero en entrar, primero en salir). Este orden, es la única forma de insertar y recuperar un elemento de la cola.



Cola

Tenga en cuenta que la operación de recuperación de un elemento de la cola, lo elimina de la misma.

Como aplicación, considere el problema de almacenar las citas diarias, con el fin de ejecutarlas en el día y hora señalados. Cuando una de las citas se ejecuta, se quita de la lista. El programa que se expone a continuación manipula este tipo de sucesos u otros. Para ello, realiza los procesos siguientes:

1. Presenta un menú con las opciones de: *introducir un suceso*, *realizar un suceso* y *salir*. A continuación solicita que elijamos una opción, lo que da lugar a que se llame a la función correspondiente. El prototipo de la función que visualiza el menú es:

```
char menu(void);
```

La función *menu* devolverá como resultado el primer carácter de la opción elegida.

2. Llama a la función *introducir* para añadir un suceso a la cola. La función *introducir* comprenderá dos casos: añadir un elemento a una cola vacía o añadir un elemento al final de la cola. El prototipo de esta función es:

```
void introducir(telemento **, telemento **, char *);
```

Esta función recibirá como parámetros la dirección del principio y del final de la cola, parámetros que serán pasados por referencia puesto que pueden variar cuando se ejecute una inserción, y el suceso a insertar.

3. Llama a la función *realizar* para sacar un suceso a la cola; se supone que el suceso recuperado, se ejecuta. Si la cola estuviese vacía, se indicará con un mensaje. El prototipo de esta función es:

```
char *realizar(telemento **, telemento **);
```

Esta función recibirá como parámetros la dirección del principio y del final de la cola, parámetros que serán pasados por referencia puesto que pueden variar cuando se ejecute una supresión. La función devuelve el suceso extraído o un cero (puntero nulo) si la cola está vacía.

Cuando se ejecute la orden *Salir* del menú, se liberará la memoria ocupada por la cola. El programa completo se muestra a continuación.

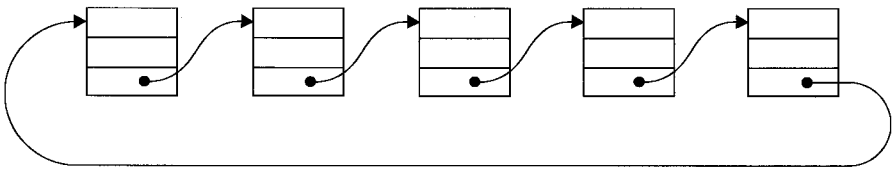
```
/********* Realización de sucesos. Aplicación de colas *********/
/* cola.c
 */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

typedef struct datos telemento;                    /* tipo telemento */
struct datos           /* estructura de un elemento de la cola */
{
  char suceso[81];
  telemento *siguiente;
};

/* Funciones */
void error(void)
{
  perror("error: insuficiente espacio de memoria");
  exit(1);
}
```

```
telemento *NuevoElemento()
{
  telemento *q = (telemento *)malloc(sizeof(telemento));
  if (!q) error();
  return (q);
}

/* Prototipos de las funciones */
char menu(void);
void introducir(telemento **, telemento **, char *);
char *realizar(telemento **, telemento **);

void main()                                    /* función principal */
{
  telemento *q, *principio, *final;
  char opcion, suceso[81];

  principio = final = NULL;                          /* cola vacía */
  while (1)
  {
    opcion = menu();                          /* visualizar el menú */

    switch (opcion)
    {
      case 'I':
        printf("\nIntroduzca suceso: ");
        gets(suceso);
        introducir(&principio, &final, suceso);
        break;
      case 'R':
        strcpy(suceso, realizar(&principio, &final));
        if (*suceso)          /* ¿primer carácter distinto de 0? */
          printf("\nRealizando el suceso %s\n", suceso);
          printf("\nPulse <Entrar> para continuar\n");
          getchar();
        break;
      case 'S':
        /* Liberar la memoria ocupada por la lista */
        q = principio; /*q apunta al primer elemento de la cola*/
        while (q != NULL)
        {
          principio = principio ->siguiente;
          free(q);
          q = principio;
        }
        return; /* salir del programa */
    }
  }
}

char menu()                                    /* menú de opciones */
{
  char op = 0;

  do
  {
    printf("\n\t  Introducir suceso\n");
```

```
    printf("\n\t  Realizar suceso\n");
    printf("\n\t  Salir\n");
    printf("\n\t  Elija la opción deseada ( I, R, S ): ");
    op = getchar();
    op = toupper(op);
    fflush(stdin);
  }
  while ( op != 'I' && op != 'R' && op != 'S' );
  return op;
}

/* Añadir un suceso a la cola */
void introducir(telemento **p, telemento **f, char *suceso)
{
  telemento *pc, *fc, *q;

  pc = *p;   /* principio de la cola */
  fc = *f;   /* final de la cola */

  q = NuevoElemento();
  strcpy(q->suceso, suceso);
  q->siguiente = NULL;
  if (fc == NULL)
    pc = fc = q;
  else
    fc = fc->siguiente = q;

  *p = pc;  *f = fc;
}

/* Recuperar un suceso de la cola */
char *realizar(telemento **p, telemento **f)
{
  telemento *pc, *fc, *q;
  char suceso[81];

  pc = *p;  /* principio de la cola */
  fc = *f;  /* final de la cola */

  if ( pc != NULL )                /* si la cola no está vacía */
  {
    q = pc;
    /* Copiar el suceso sacado de la cola */
    strcpy(suceso, q->suceso);
    /* actualizar la cola */
    pc = pc->siguiente;
    if (pc == NULL)
      fc = NULL;                        /* había un solo suceso */
    free(q);
    *p = pc; *f = fc;
    return (suceso);
  }
  printf("\nNo hay sucesos.\n");
  return 0;
}
```

LISTAS CIRCULARES

Una lista circular es una lista lineal, en la que el último elemento enlaza a su vez con el primero. De este modo, es posible acceder a cualquier elemento de la lista desde cualquier punto dado. Las operaciones sobre una lista circular resultan más sencillas, ya que se evitan casos especiales.

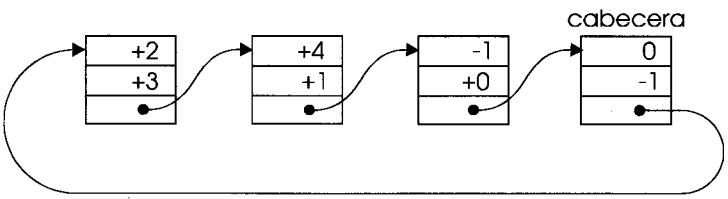


Lista circular

Cuando recorremos una lista circular, diremos que hemos llegado al final de la misma, cuando nos encontremos de nuevo en el punto de partida; esto nos hace suponer que el punto de partida se guarda de alguna manera; por ejemplo, utilizando un puntero que apunte constantemente al elemento de partida.

Una lista circular con un puntero al último elemento, es equivalente a una lista lineal con dos punteros, uno al principio y otro al final.

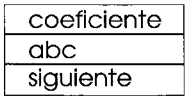
Otra posible solución al problema anterior sería poner en cada lista circular, un elemento especial identificable como lugar de parada. Este elemento especial recibe el nombre de *elemento de cabecera de la lista*. Esto presenta una ventaja, y es que la lista circular no estará nunca vacía. Por ejemplo, si utilizamos una lista circular para almacenar el polinomio $2x^3 + 4x - 1$, podríamos utilizar un elemento de cabecera que tuviera como coeficiente 0 y como exponente –1.



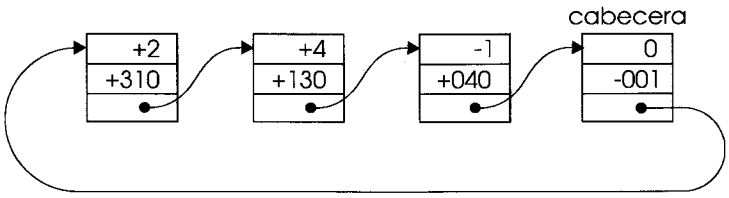
Como ejemplo de utilización de listas circulares, realizaremos la suma de ecuaciones algebraicas o polinómicas de las variables *x*, *y*, *z*. Por ejemplo:

$$2x^3y + 4xy^3 - y^4 \text{ más } 2xy^3 - xy \text{ igual a } 2x^3y + 6xy^3 - xy - y^4$$

Cada polinomio será representado por una lista circular en la que cada elemento almacena un término no nulo, como se indica a continuación:



donde *coeficiente* es el coeficiente del término $x^a y^b z^c$. Suponemos que los coeficientes y exponentes están dentro de los rangos permitidos. El miembro *abc* se utilizará para almacenar los exponentes *a*, *b* y *c* de cada término como un número entero *abc*. El signo de *abc* será siempre positivo, excepto para el elemento de cabecera. Para este elemento, *abc* = −1 y *coeficiente* = 0. Los elementos de la lista aparecerán sobre la misma en *orden decreciente* del miembro *abc*, siguiendo la dirección de los enlaces. Por ejemplo, el polinomio $2x^3y + 4xy^3 - y^4$ se representaría de la siguiente forma:



A continuación se muestra el programa correspondiente para sumar dos polinomios, almacenados en dos listas circulares, referenciadas por las estructuras *polP* y *polQ* de tipo *ListCir*.

```
typedef struct datos elemento;                    /* tipo elemento */
typedef elemento * pelemento;     /* tipo puntero a un elemento */
struct datos          /* estructura de un elemento de la lista */
{
  float coeficiente;          /* coeficiente del término en xyz */
  int abc;                              /* exponentes de x, y, z */
  pelemento siguiente;
};

typedef struct lista ListCir;
struct lista
{
  pelemento cabecera;          /* cabecera de la lista circular */
  pelemento anterior;            /* elemento anterior al actual */
  pelemento actual;           /* elemento actualmente apuntado */
};
```

El campo *abc* se corresponde con un entero igual *a*100 + b*10 + c*. Esto limita cada uno de los exponentes a un dígito. Si deseamos utilizar dos dígitos, necesitaríamos un entero de 6 dígitos. Formando el campo *abc* de esta manera, es muy sencillo, para aplicaciones posteriores, descomponerlo en los exponentes individuales.

Las estructuras *polP* y *polQ* de tipo *ListCir* que se utilizan en el programa que se muestra a continuación, definen los polinomios que queremos sumar. Para cada una de ellas, el miembro *cabecera* apunta al elemento que sirve como referencia para saber dónde empieza y dónde termina el polinomio. El miembro *actual* está en todo momento apuntado al término del polinomio sobre el que se va a realizar alguna operación. El miembro *anterior* apunta al elemento anterior al actual. En conclusión, una estructura de tipo *ListCir*, no sólo define un polinomio, sino que además utiliza un puntero para saber en todo momento sobre qué elemento se va a realizar la siguiente operación.

El programa al que hemos hecho referencia incluye las siguientes funciones:

1. *Leer polinomio*. Esta función añade, en primer lugar, el elemento de cabecera y a continuación el resto de los términos correspondientes a un polinomio definido por una estructura de tipo *ListCir*. La lectura de cada término se hace en orden creciente de los exponentes *abc*. Como consecuencia, se crea una lista circular con un elemento de cabecera seguido de los elementos que almacenan los términos del polinomio, colocados en orden decreciente del miembro *abc*.

```
void leer_polinomio(ListCir *);
```

2. *Inicializar*. Esta función inicializa los miembro *actual* y *anterior* de una estructura de tipo *ListCir* que define un polinomio, para que apunte al primer término de dicho polinomio (al que sigue al elemento de cabecera).

```
void inicializar(ListCir *);
```

3. *Sumar polinomios*. Esta función compara cada término del polinomio *polP* con los términos del polinomio *polQ* con el fin de sumar sobre *polQ* los términos de *polP* de igual exponente, y añadir a *polQ* en orden decreciente de *abc*, los términos de *polP* que no estén en *polQ*. Para estas operaciones se utilizan las funciones *sumar_coeficientes* e *insertar_nuevo_termino*, respectivamente.

```
void sumar_polinomios(ListCir *, ListCir *);
```

4. *Sumar coeficientes*. Suma dos términos, uno de *polP* y otro de *polQ*, con exponentes iguales. Los términos a sumar están referenciados por el miembro *actual* de cada una de las estructuras, *polP* y *polQ*. Si el resultado de la suma es cero, se elimina este término del polinomio *polQ*, para lo cual se invoca a la función *eliminar_termino*. Si el resultado es distinto de cero, el puntero *actual* de cada lista pasa a apuntar al siguiente elemento. Esta función es llamada por la función *Sumar_polinomios*.

```
void sumar_coeficientes(ListCir *, ListCir *);
```

5. *Insertar nuevo término.* Inserta un elemento de *polP* en la lista circular representada por *polQ*, debido a que el polinomio *polP* contiene un término que no existe en *polQ*. El elemento se inserta entre los elementos apuntados por los miembros *actual* y *anterior* de *polQ*. Esta función es llamada por la función *sumar_polinomios*.

```
void insertar_nuevo_termino(ListCir *, ListCir *);
```

6. *Eliminar término.* Esta función elimina de un polinomio, el elemento referenciado por el miembro *actual* de la estructura que define dicho polinomio. En nuestro programa se utiliza para eliminar un término nulo del polinomio *polQ*, resultado de sumar un término de *polP* con el correspondiente término de *polQ*.

```
void eliminar_termino(ListCir *);
```

7. *Escribir polinomio.* Visualiza un polinomio determinado.

```
void escribir_polinomio(ListCir);
```

8. *Liberar memoria.* Libera la memoria asignada a un determinado polinomio.

```
void liberar_memoria(ListCir);
```

Observe que en todas las funciones, menos en *escribir_polinomio* y *liberar_memoria*, los parámetros son punteros a estructuras de tipo *ListCir*. La razón es que las estructuras *polP* y *polQ* son pasadas por referencia porque sus miembros pueden variar.

La terminología empleada en el programa se interpreta de la forma siguiente:

polP: identifica al polinomio *P*. Es una estructura que contiene tres punteros: *cabecera* que apunta el elemento cabecera de la lista que contiene al polinomio *P*, *actual* que apunta el término del polinomio sobre el que estamos trabajando y *anterior* que apunta al término del polinomio anterior al actual.

polQ y *polX*: se definen de la misma forma que *polP*.

polP–>actual: hace referencia al puntero *actual* del polinomio *P*.

polP–>actual–>siguiente: hace referencia al miembro *siguiente* del elemento apuntado por el puntero *actual* del polinomio *P*.

El resto de la terminología empleada se interpreta de forma análoga.

El programa completo se muestra a continuación.

```
/* Listas circulares. Suma de ecuaciones algebraicas.
 * Cada término es función de las variables x, y, z
 * con exponentes a, b y c respectivamente, en el rango 0 a 9.
 *
 * listacir.c
 */
#include <stdio.h>
#include <stdlib.h>

typedef struct datos elemento;                    /* tipo elemento */
typedef elemento * pelemento;     /* tipo puntero a un elemento */
struct datos           /* estructura de un elemento de la lista */
{
  float coeficiente;          /* coeficiente del término en xyz */
  int abc;                                /* exponentes de x, y, z */
  pelemento siguiente;         /* puntero al siguiente elemento */
};

typedef struct lista ListCir;
struct lista
{
  pelemento cabecera;          /* cabecera de la lista circular */
  pelemento anterior;            /* elemento anterior al actual */
  pelemento actual;            /* elemento actualmente accedido */
};

/* Funciones */
void error(void)
{
  perror("error: insuficiente espacio de memoria");
  exit(1);
}
pelemento NuevoElemento()              /* crear un nuevo elemento */
{
  pelemento q = (pelemento)malloc(sizeof(elemento));
  if (!q) error();
  return (q);
}

void leer_polinomio(ListCir *);
void inicializar(ListCir *);
void sumar_polinomios(ListCir *, ListCir *);
void sumar_coeficientes(ListCir *, ListCir *);
void insertar_nuevo_termino(ListCir *, ListCir *);
void eliminar_termino(ListCir *);
void escribir_polinomio(ListCir);
void liberar_memoria(ListCir);

void main()                                   /* función principal */
{
  ListCir polP, polQ;    /* declarar los polinomios polP y polQ */

  /* Leer los polinomios polP y polQ */
  leer_polinomio(&polP);
```

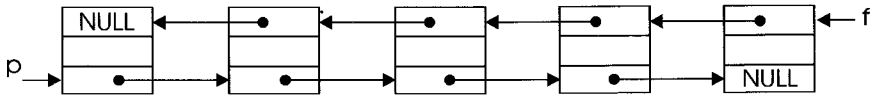
```
  leer_polinomio(&polQ);

  /* Inicializar */
  inicializar(&polP);
  inicializar(&polQ);

  /* Sumar polinomios */
  sumar_polinomios(&polP, &polQ);

  /* Escribir polinomio resultante Q */
  escribir_polinomio(polQ);

  /* Liberar la memoria ocupada por P y por Q */
  liberar_memoria(polP);
  liberar_memoria(polQ);
}

void leer_polinomio(ListCir *polX)
/* Los elementos de la lista se colocarán en orden decreciente
 * del campo abc, por lo que hay que introducirlos en orden
 * inverso, esto es en orden creciente.
 */
{
  int abc;                          /* exponentes de x, y, z */
  float coef;                /* coeficiente del término en xyz */
  pelemento q;                      /* puntero a un elemento */

  /* Añadir el elemento de cabecera */
  polX->cabecera = NuevoElemento();
  polX->cabecera->coeficiente = 0;
  polX->cabecera->abc = -001;
  polX->cabecera->siguiente = polX->cabecera;
  polX->anterior = polX->actual = NULL;

  /* Añadir los elementos restantes */
  printf("Introducir los términos del polinomio en\
  orden creciente de abc (x^a.y^b.z^c)\n\n");
  printf("Para finalizar, teclear 0 para coeficiente.\n\n");
  printf("coeficiente:                    ");
  scanf("%f", &coef);
  while (coef)
  {
    printf("exponentes abc (sin espacios):  ");
    scanf("%d", &abc);
    q = NuevoElemento();
    q->coeficiente = coef;
    q->abc = abc;
    q->siguiente = polX->cabecera->siguiente;
    polX->cabecera->siguiente = q;
    printf("coeficiente:                    ");
    scanf("%f", &coef);
  }
}

/* Inicializar proceso */
void inicializar(ListCir *polX)
{
  polX->anterior = polX->cabecera;
```

```
  polX->actual = polX->cabecera->siguiente;
}

/* Sumar los polinomios polP y polQ */
void sumar_polinomios(ListCir *polP, ListCir *polQ)
{
  while (!(polP->actual->abc < 0))
  {
    while (polP->actual->abc < polQ->actual->abc)
    {
      polQ->anterior = polQ->actual;
      polQ->actual = polQ->actual->siguiente;
    }

    if (polP->actual->abc == polQ->actual->abc)
      sumar_coeficientes(polP, polQ);
    else              /* polP->actual->abc > polQ->actual->abc */
    {
      insertar_nuevo_termino(polP, polQ);
      polP->actual = polP->actual->siguiente;
    }
  }
}

/* Sumar dos términos con exponentes iguales; uno de P y otro de Q */
void sumar_coeficientes(ListCir *polP, ListCir *polQ)
{
  if ( polP->actual->abc < 0 )
    return;
  else
  {
    polQ->actual->coeficiente += polP->actual->coeficiente;
    if ( polQ->actual->coeficiente == 0 )
    {
      eliminar_termino(polQ);
      polP->actual = polP->actual->siguiente;
    }
    else
    {
      polP->actual = polP->actual->siguiente;
      polQ->anterior = polQ->actual;
      polQ->actual = polQ->actual->siguiente;
    }
  }
}

/* El polinomio P contiene un término que no existe en Q */
void insertar_nuevo_termino(ListCir *polP, ListCir *polQ)
{
  /* Se inserta antes del actual */
  pelemento q;

  q = NuevoElemento();
  q->coeficiente = polP->actual->coeficiente;
  q->abc = polP->actual->abc;
  q->siguiente = polQ->actual;
  polQ->anterior = polQ->anterior->siguiente = q;
```

```
  return;                          /* retornar a sumar_polinomios */
}

/* Eliminar el término de coeficiente nulo */
void eliminar_termino(ListCir *polQ)
{
  pelemento q;

  q = polQ->actual;
  polQ->actual = polQ->actual->siguiente;
  polQ->anterior->siguiente = polQ->actual;
  free(q);   /* liberar la memoria ocupada por el término nulo */
  return;                      /* retornar a sumar_coeficientes */
}

void escribir_polinomio(ListCir polQ)
{
  printf("\n\nSuma de los polinomios:\n\n");
  polQ.cabecera = polQ.cabecera->siguiente;
  while (polQ.cabecera->abc != -1)
  {
    printf("coef: %+g   exps. de x y z %03d\n",
       polQ.cabecera->coeficiente, polQ.cabecera->abc);
    polQ.cabecera = polQ.cabecera->siguiente;
  }
}

void liberar_memoria(ListCir polX)
{
  pelemento q;                          /* puntero a un elemento */

  /* Liberar la memoria asignada al polinomio X */
  q = polX.cabecera; /*q apunta al primer elemento */
  while (q->coeficiente != 0 && q->abc != -1)
  {
    polX.cabecera = polX.cabecera->siguiente;
    free(q);
    q = polX.cabecera;
  }
  free(q); /* eliminar el elemento de cabecera */
}
```

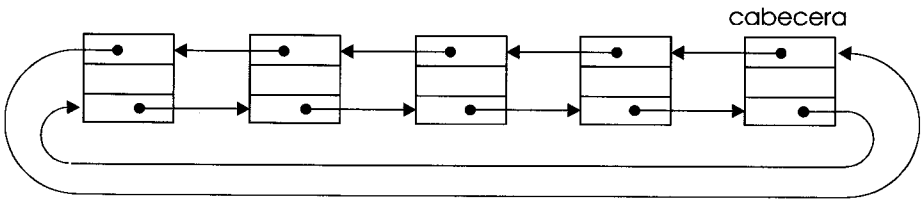
LISTAS DOBLEMENTE ENLAZADAS

Una lista doblemente enlazada, es una lista lineal en la que cada elemento tiene dos enlaces, uno al elemento siguiente y otro al elemento anterior. Esto permite avanzar sobre la lista en cualquier dirección.



Lista doblemente enlazada

Las operaciones sobre una lista doblemente enlazada, normalmente se realizan sin ninguna dificultad. Sin embargo, casi siempre es mucho más fácil la manipulación de las mismas, cuando se añade un *elemento de cabecera* y existe un doble enlace entre el último elemento y el primero a través de este elemento de cabecera. Esta estructura recibe el nombre de *lista circular doblemente enlazada.*



Como aplicación, vamos a realizar un ejemplo basado en el programa *listal.c* que hicimos al hablar de listas lineales. El programa que queremos desarrollar construirá una lista doblemente enlazada ordenada ascendentemente; cada elemento introducido se colocará automáticamente en el lugar que le corresponde.

Cada elemento de la lista tendrá la siguiente estructura:

```
typedef struct datos elemento;                    /* tipo elemento */
typedef elemento * pelemento;    /* tipo puntero a un elemento */
struct datos          /* estructura de un elemento de la lista */
{
  pelemento siguiente;
  char clave[12];
  pelemento anterior;
};
```

Una lista doblemente enlazada será representada por la siguiente estructura:

```
typedef struct lista ListDob;
struct lista
{
  pelemento princ;                    /* principio de la lista */
  pelemento final;                       /* final de la lista */
};
```

El programa estará formado fundamentalmente por tres funciones: *insertar*, *borrar*, *visualizar_lista* y *menu*:

```
void insertar(ListDob *, char *);
void borrar(ListDob *, char *);
void visualizar_lista(ListDob);
char menu(void);
```

La función *insertar* comprende los casos: insertar un elemento al principio, insertar un elemento entre otros dos e insertar un elemento al final.

La función *borrar* comprende los casos: borrar el primer elemento y borrar un elemento cualquiera que no sea el primero.

La función *visualizar_lista* permite ver en pantalla el contenido de cada uno de los elementos de la lista, ascendente o descendentemente.

La función *menu* presenta en pantalla un menú como el siguiente:

```
Introducir un nuevo elemento
Borrar un elemento
Visualizar la lista
Salir

Elija la opción deseada ( I, B, V, S ):
```

El programa completo se muestra a continuación. Si comprendió el programa *listal.c*, como ejercicio analice el código de éste otro. Tenga presente que ahora cada elemento utiliza dos enlaces.

```
/***** Lista doblemente enlazada ordenada ascendentemente *****/
/* listadob.c
 */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define ListaVacia (listaD->princ == NULL)

typedef struct datos elemento;                /* tipo elemento */
typedef elemento * pelemento;   /* tipo puntero a un elemento */
struct datos         /* estructura de un elemento de la lista */
{
  pelemento siguiente;
  char clave[12];
  pelemento anterior;
};

typedef struct lista ListDob;
struct lista
{
  pelemento princ;                    /* principio de la lista */
  pelemento final;                        /* final de la lista */
};

/* Funciones */
void error(void)
{
  perror("error: insuficiente espacio de memoria");
  exit(1);
}

pelemento NuevoElemento()
{
  pelemento q = (pelemento )malloc(sizeof(elemento));
```

```
  if (!q) error();
  return (q);
}

void insertar(ListDob *, char *);
void borrar(ListDob *, char *);
void visualizar_lista(ListDob);
char menu(void);

void main(void)
{
  ListDob listaD;
  char opcion, clave[12];

  listaD.princ = listaD.final = NULL;          /* lista vacía */
  while (1)
  {
    opcion = menu();
    switch (opcion)
    {
      case 'I':
        printf("\nIntroduzca la clave a añadir: ");
        gets(clave);
        insertar(&listaD, clave);
        break;
      case 'B':
        printf("\nIntroduzca la clave a borrar: ");
        gets(clave);
        borrar(&listaD, clave);
        break;
      case 'V':
        visualizar_lista(listaD);
        printf("\nPulse <Entrar> para continuar ");
        getchar();
        fflush(stdin);
        break;
      case 'S':
        /* Liberar la memoria ocupada por la lista */
        q = listaD.princ; /*q apunta al primer elemento de la lista*/
        while (q != NULL)
        {
          listaD.princ = listaD.princ->siguiente;
          free(q);
          q = listaD.princ;
        }
        return; /* salir del programa */    }
  }
}

/* Añadir un dato a la lista */
void insertar(ListDob *listaD, char *clave)
{
  pelemento q, pactual, panterior;

  /* Generar un elemento */
  q = NuevoElemento();
  strcpy(q->clave, clave);
```

```
  q->anterior = q->siguiente = NULL;

  if (ListaVacia)                                  /* lista vacía */
  {
    listaD->princ = listaD->final = q;
    return;
  }

  /* Buscar la posición donde hay que insertar el elemento */
  pactual = panterior = listaD->princ;
  while (pactual != NULL && strcmp(clave, pactual->clave ) > 0)
  {
    panterior = pactual;
    pactual = pactual->siguiente;
  }
  if (panterior == pactual)          /* insertar al principio */
  {
    q->siguiente = listaD->princ;
    listaD->princ = pactual->anterior = q;
  }
  else                          /* insertar después de panterior */
  {                                 /* incluye insertar al final */
    q->anterior = panterior;
    q->siguiente = pactual;
    panterior->siguiente = q;
    /* pactual será NULL cuando se inserta al final */
    if (pactual)              /* se añadió después de panterior */
      pactual->anterior = q;
    else                                    /* se añadió al final */
      listaD->final = q;
  }
}

/* Encontrar una determinada clave y borrar el elemento */
void borrar(ListDob *listaD, char *clave)
{
  pelemento panterior, pactual;

  if (ListaVacia)                                  /* lista vacía */
    return;

  /* Entrar en la lista y encontrar el elemento a borrar */
  panterior = pactual = listaD->princ;
  while (pactual != NULL && strcmp(clave, pactual->clave) != 0)
  {
    panterior = pactual;
    pactual = pactual->siguiente;
  }

  /* Si el dato no se encuentra retornar */
  if (pactual == NULL)
  {
    printf("%s no está en la lista\n", clave);
    printf("\nPulse <Entrar> para continuar ");
    getchar(); fflush(stdin);
    return;
  }
```

```
  /* Si el dato se encuentra, borrar el elemento */
  if (panterior == pactual)  /* el elemento está al principio */
  {
    listaD->princ = listaD->princ->siguiente;
    if (listaD->princ) listaD->princ->anterior = NULL;
    /* Si principio es igual a NULL había un solo elemento */
  }
  else          /* borrar un elemento que no está al principio */
  {
    /* Modificar el enlace siguiente */
    panterior->siguiente = pactual->siguiente;
    /* Modificar el enlace anterior excepto para el último */
    if (pactual->siguiente)
      pactual->siguiente->anterior = pactual->anterior;
  }
  free(pactual);
}

/* Visualizar el contenido de la lista */
void visualizar_lista(LlstDob listaD)
{
  int op;
  pelemento q;

  do
  {
    printf("\n\t  1. Ascendentemente\n");
    printf("\n\t  2. Descendentemente\n");
    printf("\n\t  Elija la opción deseada: ");
    scanf("%d", &op);
    fflush(stdin);
  }
  while (op < 1 || op > 2);

  if (op == 1) /* listado ascendente */
  {
    q = listaD.princ;
    while (q != NULL)
    {
      printf("%s\n", q->clave);
      q = q->siguiente;
    }
  }
  else          /* listado descendente */
  {
    q = listaD.final;
    while (q != NULL)
    {
      printf("%s\n", q->clave);
      q = q->anterior;
    }
  }
}

char menu(void)                          /* menú de opciones */
{
  char op;
```

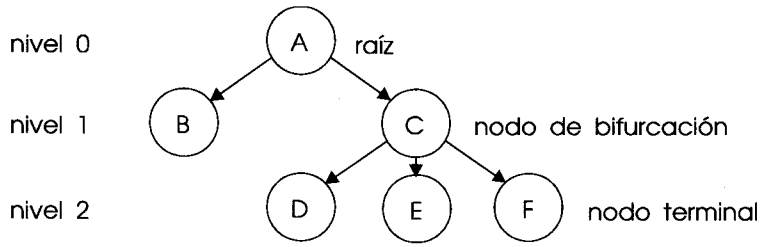
```
  do
  {
    printf("\n\t  Introducir un nuevo elemento\n");
    printf("\n\t  Borrar un elemento\n");
    printf("\n\t  Visualizar la lista\n");
    printf("\n\t  Salir\n");
    printf("\n\t  Elija la opción deseada ( I, B, V, S ): ");
    op = getchar();
    fflush(stdin);
    op = toupper(op);
  }
  while (op != 'I' && op != 'B' && op != 'V' && op != 'S');
  return (op);
}
```

ListDob es una estructura que identifica la lista doblemente enlazada que estamos creando. Contiene dos punteros que definen perfectamente la lista: *princ* que apunta al primer elemento, y *final* que apunta al último elemento. Para realizar las operaciones de inserción y borrado utilizamos dos punteros auxiliares: *pactual* que apunta al elemento identificado, y *panterior* que apunta al elemento anterior al identificado.

ÁRBOLES

Un árbol es una estructura no lineal formada por un conjunto de nodos y un conjunto de ramas.

En un árbol existe un nodo especial denominado *raíz*. Así mismo, un nodo del que sale alguna rama, recibe el nombre de *nodo de bifurcación* o *nodo rama* y un nodo que no tiene ramas recibe el nombre de *nodo terminal* o *nodo hoja*.



Árbol

De un modo más formal, diremos que un árbol es un conjunto finito de uno o más nodos tales que:

a) Existe un nodo especial llamado *raíz* del árbol, y

b) los nodos restantes están agrupados en $n > 0$ conjuntos disjuntos A_1, ... , A_n, cada uno de los cuales es a su vez un árbol que recibe el nombre de *subárbol de la raíz*.

La definición dada es recursiva, es decir, hemos definido un árbol como un conjunto de árboles. Esta es la forma más apropiada de definir un árbol.

De la definición se desprende, que cada nodo de un árbol es la raíz de algún subárbol contenido en la totalidad del mismo.

El número de ramas de un nodo recibe el nombre de *grado* del nodo.

El *nivel* de un nodo respecto al nodo raíz se define diciendo que la raíz tiene nivel 0 y cualquier otro nodo tiene un nivel igual a la distancia de ese nodo al nodo raíz. El máximo de los niveles se denomina *profundidad* o *altura* del árbol.

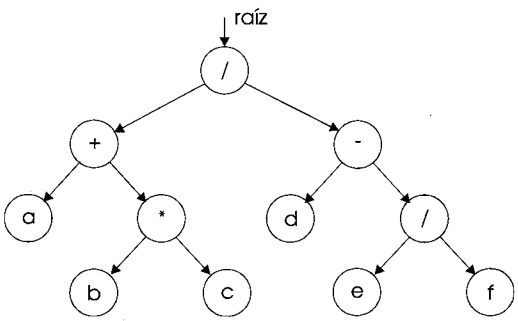
Es útil limitar los árboles en el sentido de que cada nodo sea a lo sumo de grado 2. De esta forma cabe distinguir entre subárbol izquierdo y subárbol derecho de un nodo. Los árboles así formados, se denominan árboles binarios.

Árboles binarios

Un árbol binario es un conjunto finito de nodos que consta de un nodo raíz que tiene dos subárboles binarios denominados *subárbol izquierdo* y *subárbol derecho*. Evidentemente, la definición dada es una definición recursiva, es decir, cada subárbol es un árbol binario.

Las fórmulas algebraicas, debido a que los operadores que intervienen son operadores binarios, nos dan un ejemplo de estructura en árbol binario. La figura siguiente nos muestra un árbol que corresponde a la expresión aritmética:

$$(a + b * c) / (d - e / f)$$

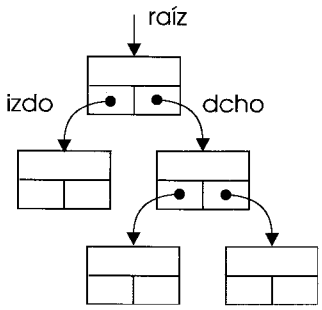


fórmulas algebraicas

Recorrido de árboles binarios

La definición dada de árbol binario, sugiere una forma natural de representar árboles binarios en un ordenador. Una variable *raíz* referenciará el árbol y cada nodo del árbol debe tener dos enlaces, *izdo* y *dcho*, uno para referenciar su subárbol izquierdo y otro para referenciar su subárbol derecho. Esto es, la declaración genérica de un nodo puede ser así:

```
typedef struct datos nodo;
struct datos    /* estructura de un nodo del árbol */
{
  /* declaración de los datos miembro */
  nodo *izdo;  /* direcciona el subárbol izquierdo */
  nodo *dcho;  /* direcciona el subárbol derecho */
};

nodo *raiz;     /* direcciona el árbol */
```

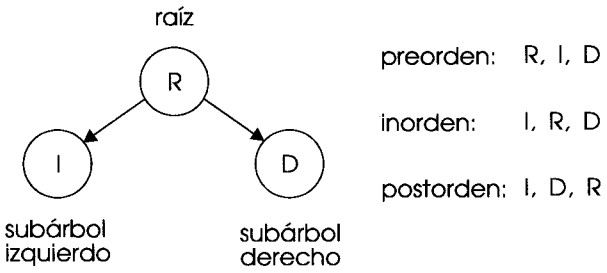


Un árbol representado como una estructura de datos

Si el árbol está vacío, *raíz* es igual a **NULL**; en caso contrario, *raíz* es un puntero que direcciona el nodo raíz del árbol, e *izdo* y *dcho* son punteros que direccionan los subárboles izquierdo y derecho del nodo raíz, respectivamente.

Hay varios algoritmos para el manejo de estructuras en árbol y un proceso que generalmente se repite en estos algoritmos es el de *recorrido de un árbol*. Este proceso consiste en examinar sistemáticamente los nodos de un árbol, de forma que cada nodo sea visitado solamente una vez.

Esencialmente pueden utilizarse tres formas para recorrer un árbol binario: *preorden*, *inorden* y *postorden*. Cuando se visitan los nodos en *preorden*, primero se visita la raíz, después el subárbol izquierdo y por último el subárbol derecho. Cuando se visitan los nodos en *inorden*, primero se visita el subárbol izquierdo, después la raíz y por último el subárbol derecho. Cuando se visitan los nodos en *postorden*, primero se visita el subárbol izquierdo, después el subárbol derecho y por último la raíz.



Evidentemente, las definiciones dadas son definiciones recursivas, ya que, recorrer un árbol utilizando cualquiera de ellas, implica recorrer sus subárboles empleando la misma definición.

Si se aplican estas definiciones al árbol binario de la figura "fórmulas algebraicas" anterior, se obtiene la siguiente solución:

```
Preorden:       / + a * b c - d / e f
Inorden:        a + b * c / d - e / f
Postorden:      a b c * + d e f / - /
```

El recorrido en preorden produce la notación *prefija*; el recorrido en inorden produce la notación *convencional*; y el recorrido en postorden produce la notación *postfija* o *inversa*.

Los nombres de preorden, inorden y postorden derivan del lugar en el que se visita la raíz con respecto a sus subárboles. Estas tres formas, se exponen a continuación como tres funciones recursivas:

```
void preorden(nodo *a);
void inorden(nodo *a);
void postorden(nodo *a);
```

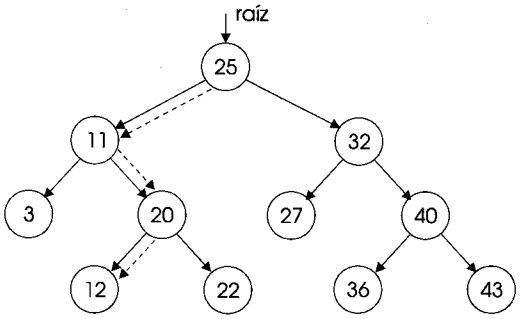
Las tres funciones tienen un único parámetro *a* que representa la dirección de la raíz del árbol cuyos nodos se quieren visitar.

```
void preorden(nodo *a)
{
  if (a != NULL)
  {
    /* operaciones con el nodo apuntado por a */
    preorden(a->izdo); /* se visita el subárbol izquierdo */
    preorden(a->dcho); /* se visita el subárbol derecho */
  }
}

void inorden(nodo *a)
{
  if (a != NULL)
  {
```

```
    inorden(a->izdo); /* se visita el subárbol izquierdo */
    /* operaciones con el nodo apuntado por a */
    inorden(a->dcho); /* se visita el subárbol derecho */
  }
}

void postorden(nodo *a)
{
  if (a != NULL)
  {
    postorden( a->izdo ); /* se visita el subárbol izquierdo */
    postorden( a->dcho ); /* se visita el subárbol derecho */
    /* operaciones con el nodo apuntado por a */
  }
}
```

ÁRBOLES BINARIOS DE BÚSQUEDA

Un árbol binario de búsqueda es un árbol ordenado; es decir, las ramas de cada nodo están ordenadas de acuerdo con las siguientes reglas: para todo nodo a_i, todas las claves del subárbol izquierdo de a_i son menores que la clave de a_i, y todas las claves del subárbol derecho de a_i son mayores que la clave de a_i.



Árbol de búsqueda

Con un árbol de estas características encontrar si un nodo de una clave determinada existe o no, es una operación muy sencilla. Por ejemplo, observando la figura anterior, localizar la clave 12 es aplicar la definición de árbol de búsqueda; esto es, si la clave buscada es menor que la clave del nodo en el que estamos, pasamos al subárbol izquierdo de este nodo, para continuar la búsqueda y si es mayor, pasamos al subárbol derecho. Este proceso continúa hasta encontrar la clave o hasta llegar a un subárbol vacío, árbol cuya raíz tiene un valor **NULL**.

Como aplicación, consideremos una secuencia de claves con el fin de determinar el número de veces que aparece cada una de ellas. Por ejemplo:

```
25 11 32 20 3 27 40 22 12 43 36 20 40 32 3 36 3
```

Esto significa que, empezando con un árbol vacío, se busca cada una de las claves en el árbol. Si se encuentra, se incrementa su contador y si no se encuentra, se inserta en el árbol como una nueva clave, con el contador correspondiente inicializado a 1. El tipo de cada uno de los nodos viene dado por la estructura,

```
typedef struct datos nodo;                          /* tipo nodo */
struct datos                  /* estructura de un nodo del árbol */
{
  int clave;
  int contador;
  nodo *izdo;      /* puntero a la raíz del subárbol izquierdo */
  nodo *dcho;        /* puntero a la raíz del subárbol derecho */
};
```

El proceso de búsqueda, función *buscar*, se formula como una función recursiva. La función tiene dos parámetros: la clave a buscar en el árbol y la raíz del mismo. Observe que el parámetro formal *raíz* está definido como un puntero a un puntero con el fin de pasarle su correspondiente parámetro actual por referencia, para hacer posible los enlaces entre los nodos.

```
void buscar(int x, nodo **raiz);
```

La función *buscar* se desarrolla aplicando estrictamente la definición de árbol de búsqueda; esto es, si la clave *x* buscada es menor que la clave del nodo en el que estamos, pasamos al subárbol izquierdo de este nodo para continuar la búsqueda (llamada recursiva para buscar en el subárbol izquierdo) y si es mayor, pasamos al subárbol derecho (llamada recursiva para buscar en el subárbol derecho). Este proceso continúa hasta encontrar la clave en cuyo caso se incrementa el contador asociado o hasta llegar a un subárbol vacío, árbol cuya raíz tiene un valor **NULL**, en cuyo caso se inserta en ese lugar un nuevo nodo con la clave *x* poniendo el contador asociado a valor uno.

Una vez construido el árbol, se utiliza la función *visualizar_arbol* para visualizar el contenido del mismo. Los nodos se visitan en *inorden* y la solución se presenta en forma de árbol, de acuerdo con el siguiente esquema:

```
                    subárbol izquierdo

        raíz

                    subárbol derecho
```

El prototipo de esta función es:

```
void visualizar_arbol(nodo *a, int n);
```

El parámetro *a* representa la raíz del árbol que se desea visualizar y *n* es la distancia de cada nodo que se visualiza, al nodo raíz (nivel del nodo).

Además de las funciones anteriores, implementaremos una más que permita liberar la memoria asignada a cada uno de los nodos del árbol. Para realizar este proceso recorreremos el árbol en postorden; piense que esta es la forma correcta de borrar un árbol: primero se borra el subárbol izquierdo, después el derecho y por último la raíz. El prototipo de la función es:

```
void borrar_arbol(nodo *a)
```

El programa completo se muestra a continuación.

```
/****************** Árbol binario de búsqueda ******************/
/* arbolbus.c
 */
#include <stdio.h>
#include <stdlib.h>

typedef struct datos nodo;                          /* tipo nodo */
struct datos                  /* estructura de un nodo del árbol */
{
  int clave;
  int contador;
  nodo *izdo;       /* puntero a la raíz del subárbol izquierdo */
  nodo *dcho;         /* puntero a la raíz del subárbol derecho */
};

/* Funciones */
void error(void)
{
  perror("error: insuficiente espacio de memoria");
  exit(1);
}

nodo *NuevoNodo()
{
  nodo *q = (nodo *)malloc(sizeof(nodo));
  if (!q) error();
  return (q);
}

void buscar(int, nodo **);
void visualizar_arbol(nodo *, int);
void borrar_arbol(nodo *a);

void main()                                    /* Función Principal */
{
  nodo *raiz=NULL;                 /* apunta a la raíz del árbol */
  int k;

  printf ("Introducir claves. Finalizar con eof\n\n");
  printf("clave:  ");
```

```
  while (scanf ("%d", &k) != EOF)
  {
    buscar(k, &raiz);           /* raíz se pasa por referencia */
    printf("clave:  ");
  }

  visualizar_arbol(raiz, 0);     /* nodo raíz y nivel del nodo */
  borrar_arbol(raiz);
}

/*****************************************************************
                    Buscar una clave en el árbol
*****************************************************************/
/* Buscar por un determinado nodo y si no está insertarlo.
 * El valor para raíz es pasado por referencia para hacer posibles
 * los enlaces entre nodo y nodo cuando se crea uno nuevo.
 */

#define ArbolVacio (a == NULL)

void buscar(int x, nodo **raiz)
{
  nodo *a;

  a = *raiz;                                    /* raíz del árbol */
  if (ArbolVacio) /* el nodo con clave x, no está en el árbol. */
  {               /* Insertarlo */
    a = NuevoNodo();
    a->clave  = x;
    a->contador = 1;
    a->izdo = a->dcho = NULL;
  }
  else
    if (x < a->clave)
          /* el valor buscado está a la izquierda de este nodo */
      buscar(x, &a->izdo);
    else
    {
      if (x > a->clave)
            /* el valor buscado está a la derecha de este nodo */
        buscar(x, &a->dcho);
      else                              /* el valor buscado existe */
        a->contador++;
    }
  *raiz = a;
}

/*****************************************************************
                      Visualizar el árbol
*****************************************************************/
/* Visualizar el árbol direccionado por a. n es nivel de cada nodo
 * que se visualiza. El árbol se recorre en inorden.
 */
void visualizar_arbol(nodo *a, int n)
{
  int i;
```

```
  if (!ArbolVacio)
  {
    visualizar_arbol(a->izdo, n+1);
    for (i = 1; i <= n; i++)
      printf("         ");
    printf("%d(%d)\n", a->clave, a->contador);
    visualizar_arbol(a->dcho, n+1);
  }
}

/*****************************************************************
                        Borrar el árbol
*****************************************************************/
/* Liberar la memoria asignada a cada uno de los nodos del árbol
 * direccionado por a. Se recorre el árbol en postorden.
 */
void borrar_arbol(nodo *a)
{
  if (!ArbolVacio)
  {
    borrar_arbol(a->izdo);
    borrar_arbol(a->dcho);
    free(a);
  }
}
```

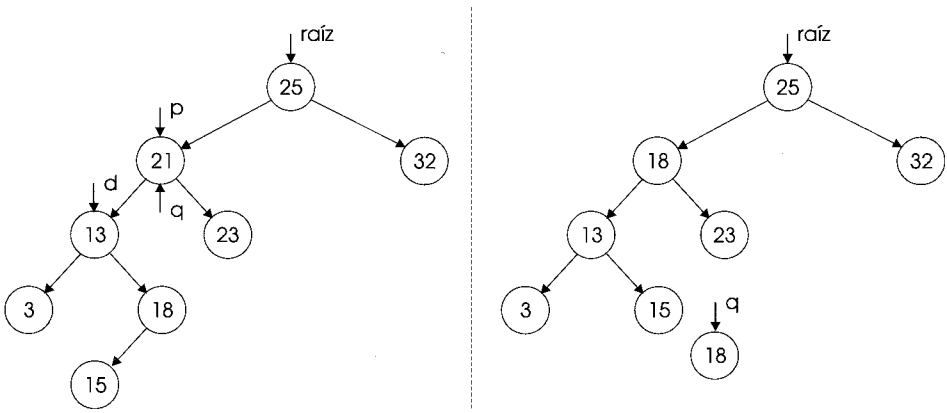
La función *buscar* puede también escribirse para que reciba como parámetros la clave a buscar y la raíz del árbol en el que se realiza la búsqueda, ambos pasados por valor, y devuelva como resultado la raíz del subárbol en el que se realiza la búsqueda o, si la clave no se encuentra, la dirección del nodo insertado (raíz de un subárbol de un solo nodo, el insertado). A continuación se muestra el código correspondiente a esta otra versión de la función.

```
/*****************************************************************
                  Buscar una clave en el árbol
*****************************************************************/
/* Buscar por un determinado nodo y si no está insertarlo.
 * La función devuelve un puntero al subárbol en el que se busca
 * para hacer posible el enlace del nuevo nodo insertado.
 */
nodo *buscar(int x, nodo *a)
{

  if (a == NULL)  /* el nodo con clave x, no está en el árbol. */
  {               /* Insertarlo */
    a = NuevoNodo();
    a->clave   = x;
    a->contador = 1;
    a->izdo = a->dcho = NULL;
  }
  else
    if (x < a->clave)
        /* el valor buscado está a la izquierda de este nodo */
      a->izdo = buscar(x, a->izdo);
    else
```

```
    {
      if (x > a->clave)
            /* el valor buscado está a la derecha de este nodo */
        a->dcho = buscar(x, a->dcho);
      else                                  /* el valor buscado existe */
        a->contador++;
    }
  return a;
}
```

Borrado en árboles

A continuación se estudia el problema de borrar el nodo con clave *x* de un árbol que tiene las claves ordenadas. Este proceso es una tarea fácil si el nodo a borrar es un nodo terminal o si tiene un único descendiente. La dificultad se presenta cuando deseamos borrar un nodo que tiene dos descendientes, ya que con un solo puntero no puede apuntarse en dos direcciones. En este caso, el nodo a borrar debe ser reemplazado, bien por el nodo más a la derecha en el subárbol izquierdo de dicho nodo, o bien por el nodo más a la izquierda en el subárbol derecho.



Borrar el nodo con clave 21

En el ejemplo que muestra la figura anterior, se desciende por el árbol hasta encontrar el nodo a borrar (21). La variable *p* representa la raíz del subárbol en el que continúa la búsqueda; inicialmente su valor es *raíz*. La variable *q* apunta al nodo a borrar una vez localizado, el cual es sustituido por el nodo más a la derecha en el subárbol izquierdo (18) del nodo apuntado por *q*. Finalmente, se borra el nodo que queda apuntado por *q*. El proceso detallado, se presenta a continuación y comprende los tres casos mencionados:

1. No hay un nodo con clave igual a *x*.
2. El nodo con clave *x* tiene un único descendiente.
3. El nodo con clave *x* tiene dos descendientes.

La función recursiva *borrar_nodo* se ejecuta solamente en el caso 3. En este caso, se desciende a lo largo de la rama más a la derecha del subárbol izquierdo del nodo apuntado por *q* que se va a borrar, y se reemplaza la información de interés en el nodo apuntado por *q* por los valores correspondientes del nodo apuntado por *d*, que es el nodo más a la derecha en el subárbol izquierdo. La función **free** libera la memoria, del nodo que ya no forma parte del árbol.

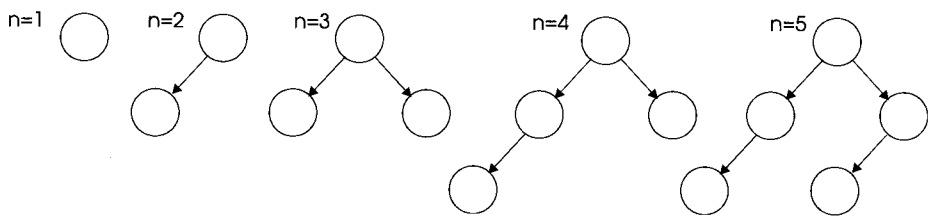
Observe que los valores para los parámetros formales *raíz* y *dr*, son pasados por referencia con el fin de realizar los enlaces necesarios. Así mismo, el valor de *q* también es pasado por referencia para poder borrar en la función *borrar*, el nodo que finalmente queda apuntado por él. La llamada a esta función será de la forma:

```
typedef struct datos nodo;                          /* tipo nodo */
...
borrar(x, &raiz);                          /* llamada a la función */
...

/*****************************************************************
                    Borrar un nodo del árbol
*****************************************************************/
/* Función para borrar un nodo cualquiera del árbol */
void borrar(int x, nodo **raiz)
{
  nodo *p = *raiz;
  nodo *q;                              /* puntero al nodo a borrar */

  /* Descender por el árbol de raíz p, para buscar el nodo
   * que se desea borrar
   */
  if (p == NULL) /* ¿árbol vacío? */
    printf("Esa componente no está en el árbol\n");
  else if (x < p->clave)
    borrar(x, &p->izdo);
  else if (x > p->clave)
    borrar( x, &p->dcho );
  else                              /* borrar el nodo apuntado por q */
  {
    q = p;
    if (q->dcho == NULL)
      p = q->izdo;
    else if (q->izdo == NULL)
      p = q->dcho;
    else                              /* nodo con dos descendientes */
      borrar_nodo(&q->izdo, &q);          /* subárbol izquierdo */
  }
  free(q);
  *raiz = p;
}

void borrar_nodo(nodo **dr, nodo **qr)
{
  nodo *d = *dr;
  nodo *q = *qr;
```

```
  /* Descender al nodo más a la derecha del subárbol d */
  if (d->dcho != NULL)
    borrar_nodo( &d->dcho, &q);
  else
  {
    /* Sustituir el nodo a borrar por el nodo más a la derecha
     * en el subárbol izquierdo
     */
    q->clave = d->clave;
    q->contador = d->contador;
    q = d;      /* q apunta al nodo que finalmente se eliminará */
    d = d->izdo;    /* para enlazar el subárbol izquierdo de d */
  }
  *dr = d;
  *qr = q;
}
```

ÁRBOLES BINARIOS PERFECTAMENTE EQUILIBRADOS

Un árbol binario está perfectamente equilibrado si, para todo nodo, el número de nodos en el subárbol izquierdo y el número de nodos en el subárbol derecho, difieren como mucho en una unidad.



Árboles perfectamente equilibrados

Como ejemplo, considere el problema de construir un árbol perfectamente equilibrado, siendo los valores de los nodos *n* números que se leen de un fichero de datos, en nuestro caso del fichero estándar de entrada **stdin**.

Esto puede realizarse fácilmente distribuyendo los nodos, según se leen, equitativamente a la izquierda y a la derecha de cada nodo. El proceso recursivo que se indica a continuación, es la mejor forma de realizar esta distribución. Para un número dado *n* de nodos y siendo *ni* (nodos a la izquierda) y *nd* (nodos a la derecha) dos enteros, el proceso es el siguiente:

1. Utilizar un nodo para la raíz.
2. Generar el subárbol izquierdo con *ni* = *n/2* nodos utilizando la misma regla.
3. Generar el subárbol derecho con *nd* = *n-ni-1* nodos utilizando la misma regla.

Cada nodo del árbol consta de los siguientes miembros: *clave*, puntero al subárbol izquierdo y puntero al subárbol derecho.

```
typedef struct datos nodo;                          /* tipo nodo */
struct datos                  /* estructura de un nodo del árbol */
{
  int clave;
  nodo *izdo;       /* puntero a la raíz del subárbol izquierdo */
  nodo *dcho;         /* puntero a la raíz del subárbol derecho */
};
```

El proceso de construcción lo lleva a cabo una función recursiva denominada *construir_arbol*, la cual construye un árbol de *n* nodos. El prototipo de esta función es:

```
nodo *construir_arbol(int);
```

Esta función tiene un parámetro entero que se corresponde con el número de nodos del árbol y devuelve un puntero a la raíz del árbol construido. En realidad diremos que devuelve un puntero a cada subárbol construido lo que permite realizar los enlaces entre nodos. Observe que para cada nodo se ejecutan las dos sentencias siguientes:

```
q->izdo = construir_arbol(ni);
q->dcho = construir_arbol(nd);
```

que asignan a los miembros *izdo* y *dcho* de cada nodo, las direcciones de sus subárboles izquierdo y derecho, respectivamente.

Además de esta función, añadiremos al programa, las funciones *visualizar_arbol* y *borrar_arbol* vistas en el programa anterior.

El programa completo se muestra a continuación.

```
/*************** Árbol perfectamente equilibrado ***************/
/* arboleq.c
 */
#include <stdio.h>
#include <stdlib.h>

typedef struct datos nodo;                          /* tipo nodo */
struct datos                  /* estructura de un nodo del árbol */
{
  int clave;
  nodo *izdo;       /* puntero a la raíz del subárbol izquierdo */
  nodo *dcho;         /* puntero a la raíz del subárbol derecho */
};

/* Funciones */
```

```
void error(void)
{
  perror("error: insuficiente espacio de memoria");
  exit(1);
}

nodo *NuevoNodo()
{
  nodo *q = (nodo *)malloc(sizeof(nodo));
  if (!q) error();
  return (q);
}

nodo *construir_arbol(int);
void visualizar_arbol(nodo *, int);
void borrar_arbol(nodo *a);

void main ()                                  /* función principal */
{
  nodo *raiz;                        /* apunta a la raíz del árbol */
  int n;

  printf ("Número de nodos: "); scanf("%d", &n);
  printf ("Introducir claves:\n\n");
  raiz = construir_arbol(n);      /* construir árbol de n nodos */
  visualizar_arbol(raiz, 0);
  borrar_arbol(raiz);
}

/*****************************************************************
                    Función construir árbol
*****************************************************************/
/* Construir un árbol de n nodos perfectamente equilibrado
 */
nodo *construir_arbol(int n)
{
  nodo *q;
  int ni, nd;

  if (n == 0)
    return (NULL);
  else
  {
    ni = n / 2;                  /* nodos del subárbol izquierdo */
    nd = n - ni - 1;               /* nodos del subárbol derecho */
    q = NuevoNodo();
    printf("clave:  "); scanf("%d", &q->clave);
    q->izdo = construir_arbol(ni);
    q->dcho = construir_arbol(nd);
    return (q);
  }
}

/*****************************************************************
                      Visualizar el árbol
*****************************************************************/
/* Visualizar el árbol direccionado por a. n es el nivel de cada
```

```
 * nodo que se visualiza. El árbol se recorre en inorden.
 */
void visualizar_arbol(nodo *a, int n)
{
  int i;

  if (a != NULL) /* si el árbol no está vacío ... */
  {
    visualizar_arbol(a->izdo, n+1);
    for (i = 1; i <= n; i++)
      printf("          ");
    printf("%d\n", a->clave);
    visualizar_arbol(a->dcho, n+1);
  }
}

/*****************************************************************
                        Borrar el árbol
*****************************************************************/
/* Liberar la memoria asignada a cada uno de los nodos del árbol
 * direccionado por a. Se recorre el árbol en postorden.
 */
void borrar_arbol(nodo *a)
{
  if (a != NULL)
  {
    borrar_arbol(a->izdo);
    borrar_arbol(a->dcho);
    free(a);
  }
}
```

EJERCICIOS RESUELTOS

1. Supongamos un polinomio con términos de la forma cx^Ay^B. Por ejemplo:

   $2x^2y - xy + 5$

   La estructura correspondiente a cada término será de la siguiente forma:

```
typedef struct datos elemento;
struct datos
{
  float coeficiente;
  int ab;
  elemento *siguiente;
};
```

   donde *ab* es un número entero resultado de juntar el exponente *A* y el exponente *B*; suponemos ambos comprendidos en el intervalo 0 a 9.

   Se pide:

a) Escribir una función que cree el elemento de cabecera para una lista circular simplemente enlazada con elementos del tipo *elemento*. La función prototipo será de la forma:

```
elemento *crear_cabecera();
```

b) Escribir una función que permita insertar un término de un polinomio, en una lista circular con un elemento de cabecera. La inserción se realizará de forma que los elementos de la lista a partir del elemento cabecera queden clasificados en orden decreciente del campo *ab*. La función prototipo será de la forma:

```
void insertar(elemento *cab, float coef, int ab);
```

donde *cab* es un puntero al elemento de cabecera y *coef* y *ab* son el coeficiente y los exponentes de *x* e *y* respectivamente.

c) Escribir una función que visualice el polinomio en orden decreciente del campo *ab* como muestra el ejemplo siguiente:

```
2x^2y - xy + 5
```

La función prototipo será de la forma:

```
void visualizar(elemento *cab);
```

donde *cab* es un puntero al elemento de cabecera.

Escribir una función principal **main** que lea el *coeficiente* y los exponentes *ab* de cada término de un determinado polinomio, e inserte dichos términos en la lista llamando a la función *insertar*. Así mismo, cuando la lista esté construida, llamará a la función *visualizar* para presentar el polinomio en pantalla.

El programa completo se muestra a continuación.

```
/************************ Polinomios ************************/
/* polinomi.c
 */
#include <stdio.h>
#include <stdlib.h>

typedef struct datos elemento;
struct datos
{
  float coeficiente;
  int ab;
  elemento *siguiente;
};
```

```
elemento *crear_elemento()
{
  /* Crear un nuevo elemento */
  elemento *e = (elemento *)malloc(sizeof(elemento));

  if (e == NULL)
  {
    perror("crear_elemento");
    exit(-1);
  }
  e->coeficiente = 0;
  e->ab = 0;
  e->siguiente = NULL;
  return e;
}

elemento *crear_cabecera()
{
  /* Crear el elemento de cabecera */
  elemento *cabecera = crear_elemento();

  cabecera->siguiente = cabecera;
  return cabecera;
}

void insertar(elemento *cab, float coef, int ab)
{
  /* Crear un nuevo elemento */
  elemento *nuevo = crear_elemento();
  elemento *anterior = cab, *siguiente = anterior->siguiente;

  nuevo->coeficiente = coef;
  nuevo->ab = ab;

  /* Insertar el elemento en la lista ordenado por ab */
  while (siguiente != cab && nuevo->ab < siguiente->ab)
  {
    anterior = siguiente;
    siguiente = anterior->siguiente;
  }
  anterior->siguiente = nuevo;
  nuevo->siguiente = siguiente;
}

void visualizar(elemento *cab)
{
  #define abs(a) ((a)>0 ? (a) : (-(a)))
  int a, b;
  elemento *q = cab->siguiente;

  /* Visualizar el polinomio */
  while (q != cab)
  {
    /* Visualizar el término apuntado por q */
    a = q->ab / 10;
    b = q->ab % 10;
    if (q->coeficiente < 0)
```

```
      printf(" - ");
    else if (q->coeficiente > 0 && q != cab->siguiente)
      printf(" + ");
    if (abs(q->coeficiente) != 1 || q->ab == 0)
      printf("%g", abs(q->coeficiente));
    if (a > 0)
      printf("x");
    if (a > 1)
      printf("^%d", a);
    if (b > 0)
      printf("y");
    if (b > 1)
      printf("^%d", b);
    q = q->siguiente;
  }
}

void borrar_lista(elemento *cab)
{
  elemento *p, *q = cab->siguiente;
  /* Liberar la memoria asignada al polinomio */
  while (q != cab)
  {
    p = q->siguiente;
    free(q);
    q = p;
  }
  free(q); /* eliminar el elemento de cabecera */
}

void main()
{
  elemento *cab = crear_cabecera();
  int a, b, r;
  float coef;

  printf("Introducir por cada término el coeficiente, a entre\n"
         "0 y 9 y b entre 0 y 9. Por ejemplo: -3 0 2\n"
         "Finalice la entrada con 0 0 0\n\n");
  printf("término (coef a b): ");

  while ((r = scanf("%g %d %d", &coef, &a, &b)) == 3 &&
         (a>=0 && a<=9 && b>=0 && b<=9))
  {
    if ( coef == 0 && a == 0 && b == 0 ) break;
    insertar(cab, coef, a*10+b);
    printf("término (coef a b): ");
  }

  if ( r == 3 && a>=0 && a<=9 && b>=0 && b<=9 )
  {
    printf("Polinomio: ");
    visualizar(cab);
  }
  else
    printf("Error en la entrada de datos. "
           "Ejecute de nuevo el programa.\n");
```

```
  printf("\n");
  borrar_lista(cab);
}
```

2. Se quiere escribir un programa para ordenar líneas de caracteres por el método de la burbuja, pero en lugar de utilizar arrays, vamos a realizar la ordenación utilizando listas doblemente enlazadas. Las líneas de texto procederán de un fichero existente en el disco. Cada elemento de la lista tiene la forma siguiente:

```
struct elemento
{
  char *cadena; /* puntero a una línea de texto */
  struct elemento *anterior, *siguiente;
};
```

Para acceder a la lista vamos a declarar una estructura que defina el comienzo y el final de la misma:

```
struct control_lista
{
  struct elemento *inicio; /* puntero al primer elemento de la lista */
  struct elemento *fin;    /* puntero al último elemento de la lista */
};
```

Se pide lo siguiente:

a) Escribir una función *leer_fichero* que tras abrir el fichero, lo lea y forme una lista doblemente enlazada en la que cada elemento contenga una línea de dicho fichero. Tras formar la lista, se devolverá la estructura de control de la misma.

```
struct control_lista leer_fichero(char *fichero);
```

b) Escribir la función *ordenar_lista* que ordene por el método de la burbuja todas las líneas de una determinada lista pasada como argumento. La ordenación se hará sobre la propia lista; esto es, sin utilizar arrays adicionales. Las líneas se deben ordenar en orden alfabético ascendente.

```
void ordenar_lista(struct control_lista lista);
```

c) Escribir la función *visualizar_lista* para que visualice todas las líneas que hay en la lista pasada como argumento.

```
void visualizar_lista(struct control_lista lista);
```

d) Escribir la función *borrar_lista* para que libere la memoria ocupada por todos los elementos de la lista. Tenga presente que también tiene que eliminar la memoria que fue asignada para almacenar las líneas de texto.

```
void borrar_lista(struct control_lista lista);
```

Utilizando las funciones anteriores, escribir un programa que reciba el nombre de un fichero de texto a través de la línea de órdenes del sistema operativo y tras leer sus líneas, las presente por pantalla en orden alfabético ascendente.

El programa completo se muestra a continuación.

```
/****************** Lista doblemente enlazada ******************/
/* ordenar.c
 */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

struct elemento
{
  char *cadena; /* puntero a una línea de texto */
  struct elemento *anterior, *siguiente;
};

struct control_lista
{
  struct elemento *inicio; /* puntero al primer elemento de la lista */
  struct elemento *fin;    /* puntero al último elemento de la lista */
};

struct elemento *crear_elemento(char *linea)
{
  struct elemento *q;

  /* Asignar memoria para un elemento */
  if ((q = (struct elemento *)malloc(sizeof(struct elemento))) == NULL)
  {
    printf("Error al asignar memoria.\n");
    exit(-1);
  }
  q->siguiente = q->anterior = NULL;

  /* Asignar memoria para almacenar una línea */
  if ((q->cadena = (char *)malloc(strlen(linea)+1)) == NULL)
  {
    printf("Error al asignar memoria.\n");
    exit(-1);
  }

  strcpy(q->cadena, linea);
  return (q);
}
```

```
struct control_lista leer_fichero(char *fichero)
{
  #define MAX_CHAR 256
  FILE *stream;
  struct control_lista lista;
  char linea[MAX_CHAR];

  /* Abrir el fichero para leer */
  if ((stream = fopen(fichero, "r")) == NULL)
  {
    printf("No se puede abrir el fichero: %s\n", fichero);
    exit(-1);
  }
  lista.inicio = lista.fin = NULL; /* lista vacía */

  /* Leer las líneas de texto del fichero y almacenarlas en una
   * lista doblemente enlazada.
   *
   * Primer elemento de la lista
   */
  if (fgets(linea, MAX_CHAR, stream) != NULL)
    lista.inicio = lista.fin = crear_elemento(linea);
  /* Los restantes elementos de la lista se añaden a continuación
   * del elemento inicialmente creado */
  while (fgets(linea, MAX_CHAR, stream) != NULL)
  {
    lista.fin->siguiente = crear_elemento(linea);
    lista.fin->siguiente->anterior = lista.fin;
    lista.fin = lista.fin->siguiente;
  }
  fclose(stream);
  return (lista);
}

void ordenar_lista(struct control_lista lista)
{
  char *aux;
  enum BOOLEAN {NO, SI} ordenado = NO;
  struct elemento *actual, *fin = lista.fin;

  /* Ordenar la lista utilizando el método de la burbuja */
  while (!ordenado && fin != lista.inicio)
  {
    ordenado = SI;
    for (actual = lista.inicio; actual != fin; actual = actual->siguiente)
    {
      if (strcmp(actual->cadena, actual->siguiente->cadena) > 0)
      {
        aux = actual->cadena;
        actual->cadena = actual->siguiente->cadena;
        actual->siguiente->cadena = aux;
        ordenado = NO;
      }
    }
    fin = fin->anterior;
  }
}
```

```
void visualizar_lista(struct control_lista lista)
{
  struct elemento *q = lista.inicio;

  /* Visualizar todos los elementos de la lista */
  while (q != NULL)
  {
    puts(q->cadena);
    q = q->siguiente;
  }
}

void borrar_lista(struct control_lista lista)
{
  struct elemento *q = lista.inicio;

  /* Liberar la memoria ocupada por la lista */
  while (q != NULL)
  {
    lista.inicio = lista.inicio->siguiente;
    free(q->cadena); /* liberar la memoria ocupada por la línea */
    free(q);      /* liberar la memoria ocupada por el elemento */
    q = lista.inicio;
  }
}

void main(int argc, char *argv[])
{
  struct control_lista lista;

  if (argc != 2)
  {
    printf("Sintaxis: %s nombre_fichero\n", argv[0]);
    exit(-1);
  }

  lista = leer_fichero(argv[1]);
  ordenar_lista(lista);
  visualizar_lista(lista);
  borrar_lista(lista);
}
```

3. En un fichero tenemos almacenados los nombres y las notas de los alumnos de un determinado curso. Cada registro del fichero tiene la siguiente estructura:

```
struct alumno
{
  char nombre[61];
  float nota;
} talumno;
```

Queremos leer los datos de este fichero para construir una estructura de datos en memoria que se ajuste a un árbol binario de búsqueda perfectamente equilibrado. Para ello, se pide:

a) Escribir una función que visualice la estructura en árbol.

```
void visualizar_arbol(tnodo *arbol, int nivel);
```

tnodo es el tipo de los nodos del árbol:

```
typedef struct nodo
{
  talumno alumno;
  struct nodo *izdo, *dcho;
} tnodo;
```

b) Escribir una función que a partir del árbol imprima un listado con los nombres y sus correspondientes notas, en orden ascendente.

```
void visualizar_ascen(tnodo *arbol);
```

c) Escribir una función que permita crear un nodo de tipo *tnodo*. La función devolverá un puntero al nodo creado.

```
tnodo *crear_nodo();
```

d) Escribir una función que utilizando la función *crear_nodo* permita construir un árbol de búsqueda perfectamente equilibrado, a partir de los registros de un fichero ordenado. La función devolverá la raíz del árbol construido.

```
tnodo *crear_arbol(FILE *pf, int total_nodos);
```

e) Escribir una función que libere la memoria ocupada por la estructura de datos en árbol.

```
void borrar_arbol(tnodo *a);
```

f) Escribir una función principal **main** que utilizando las funciones anteriores cree un *árbol binario de búsqueda perfectamente equilibrado* a partir de los datos de un fichero pasado como argumento en la línea de órdenes. Así mismo, visualizará la estructura en árbol y un listado en orden ascendente de los nombres de los alumnos con sus notas. Finalmente borrará el árbol.

NOTA: Antes de crear el árbol, debemos ordenar el fichero. Esto facilita la creación del árbol binario con los dos requisitos impuestos: que sea de *búsqueda* y *perfectamente equilibrado*. En este ejercicio supondremos que partimos de un fichero ordenado. El en capítulo "Algoritmos" veremos cómo se ordena un fichero.

El programa completo se muestra a continuación. No obstante, vamos a realizar antes algunas aclaraciones.

Por definición de árbol de búsqueda, para todo nodo, las claves menores que la del propio nodo forman el subárbol izquierdo y las mayores el subárbol derecho. Según esto, la clave menor se encuentra en el nodo más a la izquierda y la clave mayor en el nodo más a la derecha del árbol. Por lo tanto, para visualizar las claves en orden ascendente tendremos que recorrer el árbol en inorden.

Entonces, si pensamos en el proceso inverso, esto es, si partimos de un fichero con las claves ordenadas y construimos un árbol perfectamente equilibrado tenemos que utilizar la forma inorden para conseguir al mismo tiempo que el árbol sea de búsqueda. Es decir, en la función *crear_arbol* es obligado el orden en el que se ejecutan las siguientes sentencias:

```
q->izdo = crear_arbol(pf, ni); /* crear subárbol izquierdo */
fread(&q->alumno, sizeof(talumno), 1, pf); /* raíz */
q->dcho = crear_arbol(pf, nd); /* crear subárbol derecho */
```

Vemos que primero se construye el subárbol izquierdo, después la raíz y por último el subárbol derecho. Como las claves contenidas en el fichero están ordenadas, la clave menor se almacenará en el nodo más a la izquierda y la mayor en el nodo más a la derecha, dando así lugar a un árbol de búsqueda, además de perfectamente equilibrado. Vea a continuación el programa completo.

```
/**** Árbol binario de búsqueda y perfectamente equilibrado ****/
/* arbolbeq.c
 */
#include <stdio.h>
#include <stdlib.h>

#define MAX 61

typedef struct
{
  char nombre[MAX];
  float nota;
} talumno;

typedef struct nodo
{
  talumno alumno;
  struct nodo *izdo, *dcho;
} tnodo;

void visualizar_arbol(tnodo *arbol, int nivel)
{
  /* Imprimir los nodos de un árbol binario */
  int i;
  if (arbol != NULL)
  {
    visualizar_arbol(arbol->izdo, nivel+1);
    for (i = 0; i < nivel; i++)
      printf("\t");
```

```
    printf("%s (%g)\n", arbol->alumno.nombre, arbol->alumno.nota);
    visualizar_arbol(arbol->dcho, nivel+1);
  }
}

void visualizar_ascen(tnodo *arbol)
{
  /* Listado alfabético */
  if (arbol != NULL)
  {
    visualizar_ascen(arbol->izdo);
    printf("%s (%g)\n", arbol->alumno.nombre, arbol->alumno.nota);
    visualizar_ascen(arbol->dcho);
  }
}

tnodo *crear_nodo()
{
  /* Crear un nodo del árbol binario */
  tnodo *aux;

  if ((aux = (tnodo *)malloc(sizeof(tnodo))) == NULL)
  {
    perror("crear_nodo");
    exit(-1);
  }
  aux->izdo = aux->dcho = NULL;
  return aux;
}

tnodo *crear_arbol(FILE *pf, int total_nodos)
{
  /* Crear un árbol binario de búsqueda perfectamente
   * equilibrado partiendo de unos datos ordenados.
   */
  int ni, nd;
  tnodo *q;

  if (total_nodos == 0)
    return NULL;

  ni = total_nodos / 2;
  nd = total_nodos - ni - 1;
  q = crear_nodo();
  q->izdo = crear_arbol(pf, ni);
  fread(&q->alumno, sizeof(talumno), 1, pf);
  q->dcho = crear_arbol(pf, nd);

  return q;
}

void borrar_arbol(tnodo *a)
{
  /* Liberar la memoria asignada a cada uno de los nodos del árbol
   * direccionado por a. Se recorre el árbol en postorden.
   */
```

```
  if (a != NULL)
  {
    borrar_arbol(a->izdo);
    borrar_arbol(a->dcho);
    free(a);
  }
}

void main(int argc, char *argv[])
{
  FILE *pf;
  tnodo *raiz;
  int total;

  /* Verificar el número de argumentos pasados en la línea
   * de órdenes
   */
  if (argc != 2)
  {
    fprintf(stderr, "Sintaxis: %s nombre_fichero.\n",argv[0]);
    exit(-1);
  }

  /* Abrir el fichero. */
  if ((pf = fopen(argv[1], "r+b")) == NULL)
  {
    perror(argv[1]);
    exit(-1);
  }

  /* Cálculo del total de registros del fichero. */
  fseek(pf, 0L, SEEK_END);
  total = (int)ftell(pf) / sizeof(talumno);
  rewind(pf);

  /* Si el fichero no está ordenado, invocar a la función
   * correspondiente para ordenarlo. En este caso suponemos
   * que está ordenado.
   */
  raiz = crear_arbol(pf, total);
  fclose(pf);
  visualizar_arbol(raiz, 0);
  visualizar_ascen(raiz);

  borrar_arbol(raiz);
}
```

4. Escribir un programa que permita calcular la frecuencia con la que aparecen las palabras en un fichero de texto. La forma de invocar al programa será:

```
palabras nombre_fichero↵
```

donde *nombre_fichero* es el nombre del fichero de texto del cual queremos obtener la estadística.

La contabilidad de las palabras que aparezcan en el texto la vamos a realizar de la siguiente forma:

a) Leemos una palabra del fichero. Se define una palabra como una secuencia de caracteres alfabéticos. Dos palabras consecutivas estarán separadas por espacios en blanco, tabuladores, signos de puntuación, etc.

b) La palabra leída deberá insertarse por orden alfabético ascendente, en el nodo adecuado de una estructura de datos cuyo primer elemento tiene la forma:

```
typedef struct nodo
{
  char *palabra;
  int contador;
  struct nodo *izdo;
  struct nodo *dcho;
} tnodo;
```

donde *palabra* es un puntero a la palabra leída, *contador* indica el total de veces que la palabra aparece en el texto, *izdo* es un puntero a una estructura tipo *tnodo* que contiene una palabra que está alfabéticamente antes que la actual y *dcho* es un puntero a una estructura de tipo *tnodo* que contiene una palabra que está alfabéticamente después de la actual.

c) Una vez que hemos leído todo el fichero, se presentará por pantalla una estadística con el siguiente formato:

```
TOTAL PALABRAS = 66
TOTAL PALABRAS DIFERENTES = 32
a = 3
actual = 2
alfabéticamente = 2
antes = 1
aparece = 1
contador = 1
contiene = 2
de = 3
derecho = 1
después = 1
...
```

El programa incluye un fichero de cabecera *palabras.h* que aporta las definiciones y declaraciones necesarias para el mismo.

```
/* palabras.h */
/* Estructura de un nodo del árbol */
typedef struct nodo
{
  char *palabra;
  int contador;
  struct nodo *izdo;
  struct nodo *dcho;
} tnodo;

/* Variables globales. */
int total_palabras = 0, total_palabras_diferentes = 0;

/* Prototipos */
tnodo *crear_nodo(char *palabra);
int leer_palabra(FILE *pf, char *palabra);
tnodo *insertar_nodo(tnodo *palabras, char *palabra);
void imprimir_estadistica(tnodo *palabras);
void borrar_arbol(tnodo *a);
```

La función *crear_nodo* crea un nodo que almacena la palabra pasada como argumento.

La función *leer_palabra* lee una palabra del fichero referenciado por *pf*. La palabra queda apuntada por el parámetro *palabra*. Esta función devuelve 1 si lee una palabra y 0 en otro caso.

La función recursiva *insertar_nodo* inserta un nodo que almacena la palabra referenciada por *palabra*, en un árbol binario de búsqueda. El parámetro *palabras* es la raíz del árbol. La función devuelve la raíz del árbol (subárbol construido) para permitir realizar los enlaces entre nodos.

La función *imprimir_estadistica* visualiza en orden alfabético las palabras y la frecuencia con que aparecen.

La función *borrar_arbol* libera la memoria ocupada por la estructura en árbol.

El programa completo se muestra a continuación.

```
/*********** Estadística de las palabras de un texto ***********/
/* palabras.c
 */
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include <string.h>
#include "palabras.h"

void main(int argc, char **argv)
{
  char palabra[256];
```

```
  FILE *pf;
  tnodo *palabras = NULL;

  /* Analizar la línea de órdenes y abrir el fichero */
  if (argc < 2)
    pf = stdin;
  else if ((pf = fopen(argv[1], "r")) == NULL)
  {
    perror(argv[1]);
    exit(-1);
  }

  /* Leer las palabras del fichero y construir el árbol binario de
     búsqueda */
  while (leer_palabra(pf, palabra))
    palabras = insertar_nodo(palabras, palabra);

  /* Impresión de los resultados. */
  printf("TOTAL PALABRAS = %d\n", total_palabras);
  printf("TOTAL PALABRAS DIFERENTES = %d\n", total_palabras_diferentes);
  imprimir_estadistica(palabras);
  borrar_arbol(palabras);
}

tnodo *crear_nodo(char *palabra)
{
  /* "palabra" referencia la palabra que queremos almacenar */
  tnodo *nodo;

  /* Reservar memoria para un nodo del árbol */
  if ((nodo = (tnodo *)malloc(sizeof(tnodo))) == NULL)
  {
    perror("crear_nodo");
    exit(-1);
  }
  /* Reservar memoria para almacenar la palabra en el nodo */
  if ((nodo->palabra=(char *)malloc(strlen(palabra)+1)) == NULL)
  {
    perror("crear_nodo");
    free(nodo);
    exit(-1);
  }
  /* Almacenar la palabra en el nodo */
  strcpy(nodo->palabra, palabra);
  nodo->contador = 1;
  nodo->izdo = nodo->dcho = NULL;
  /* Incrementar el contador de palabras diferentes */
  total_palabras_diferentes++;
  return nodo;
}

int leer_palabra(FILE *pf, char *palabra)
{
  int c;
  /* Leer una palabra del fichero referenciado por pf. La palabra
   * queda apuntada por "palabra"
   */
```

```
    *palabra = '\0';

    /* Eliminar los caracteres que no forman parte de la palabra */
    while ((c = fgetc(pf)) != EOF && !isalpha(c) && !strchr("áéíóú", c));
    if (c == EOF) return 0;

    /* Leer una palabra */
    *palabra++ = tolower(c);
    while ((c = fgetc(pf)) != EOF && isalpha(c) || strchr("áéíóú", c))
    *palabra++ = tolower(c);
    *palabra = '\0';

    /* Incrementar el contador de palabras leídas */
    total_palabras++;
    return 1;
}

tnodo *insertar_nodo(tnodo *palabras, char *palabra)
{
    /* Insertar un nodo que almacena "palabra" en un árbol binario
     * de búsqueda. "palabras" es la raíz del árbol.
     */
    int cmp;

    /* Si el árbol está vacío, creamos el nodo raíz */
    if (palabras == NULL)
      return (crear_nodo(palabra));

    cmp = strcmp(palabra, palabras->palabra);
    if (cmp == 0)
      palabras->contador++;
    else if (cmp < 0)
      palabras->izdo=insertar_nodo(palabras->izdo, palabra);
    else
      palabras->dcho = insertar_nodo(palabras->dcho, palabra);

    return palabras;
}

void imprimir_estadistica(tnodo *palabras)
{
    /* Recorrer el árbol en inorden para imprimir por orden
     * alfabético las palabras y su frecuencia de aparición
     */
    if (palabras != NULL)
    {
      imprimir_estadistica(palabras->izdo);
      printf("%s = %d\n", palabras->palabra, palabras->contador);
      imprimir_estadistica(palabras->dcho);
    }
}

void borrar_arbol(tnodo *a)
{
    /* Liberar la memoria asignada a cada uno de los nodos del árbol
     * direccionado por a. Se recorre el árbol en postorden.
     */
```

```
  if (a != NULL)
  {
    borrar_arbol(a->izdo);
    borrar_arbol(a->dcho);
    free(a);
  }
}
```

EJERCICIOS PROPUESTOS

1. Realizar un programa que permita crear una lista lineal clasificada ascendentemente, en la que cada elemento sea del tipo *datos* que se especifica a continuación:

```
typedef struct alumno
{
  char nombre[60];
  float nota;
} talumno;

typedef struct datos
{
  talumno alumno;
  struct datos *siguiente;
} telemento;
```

El programa incluirá las siguientes funciones:

a) Añadir un elemento. Esta función comprenderá dos casos: insertar un elemento al principio de la lista o insertar un elemento después de otro.

```
void anadir(telemento **, talumno);
```

La función *añadir* recibirá como parámetros la dirección del primer elemento de la lista, parámetro que será pasado por referencia puesto que puede variar cuando se inserte un elemento al principio, y los datos del alumno a insertar.

b) Borrar un elemento de la lista. Esta función comprenderá dos casos: borrar el elemento del principio de la lista o borrar otro elemento cualquiera.

```
void borrar(telemento **, talumno);
```

La función *borrar* recibirá como parámetros la dirección del primer elemento de la lista, parámetro que será pasado por referencia puesto que puede variar cuando se borre el primer elemento, y los datos del alumno a borrar.

c) Buscar un elemento en la lista.

```
telemento *buscar(telemento *, talumno);
```

La función *buscar* recibirá como parámetros la dirección del primer elemento de la lista y los datos del alumno a buscar y devolverá como resultado la dirección del elemento si se encuentra, o un valor **NULL** si no se encuentra.

d) Visualizar el contenido de la lista.

```
void visualizar(telemento *);
```

La función *visualizar* recibirá como parámetros la dirección del primer elemento de la lista.

e) Presentar un menú con cada una de las operaciones anteriores.

```
int menu(void);
```

La función *menu* presentará un menú con las opciones: *añadir*, *borrar*, *buscar*, *visualizar* y *salir* y devolverá como resultado un entero correspondiente a la opción elegida.

f) Borrar la lista.

```
void borrar_lista(telemento *);
```

La función *borrar_lista* liberará la memoria ocupada por todos los elementos de la lista.

2. Supongamos la siguientes declaraciones:

```
typedef struct term *polinomio;
struct term
{
  float coeficiente;
  int grado;
  polinomio siguiente;
};
```

Con las variables de tipo *polinomio* se formarán listas dinámicas en las que cada uno de los elementos se corresponderá con un término en *x* del polinomio (*coeficiente* y *grado*). Se pide:

a) Realizar una función *leer* que permita leer un polinomio introducido por el teclado de la forma: *5 x4 −1 x1 +1 x0 +3 x2*. Como vemos el polino-

mio puede introducirse no ordenado y entre cada coeficiente y la *x* hay uno o más espacios en blanco; lo mismo ocurre entre dos términos en *x* consecutivos.

```
polinomio leer(void);
```

b) Realizar una función *ordenar* para poner los términos del polinomio en orden descendente. Por ejemplo, de acuerdo con la entrada anterior, el polinomio quedaría así: *5 x4 +3 x2 -1 x1 +1 x0.*

```
void ordenar(polinomio);
```

c) Realizar una función *completar* para introducir los términos de coeficiente 0 que faltan para obtener un polinomio completo. Por ejemplo, siguiendo con el polinomio anterior, el resultado sería el siguiente:

5 x4 +0 x3 +3 x2 -1 x1 +1 x0.

```
void completar(polinomio);
```

3. En un fichero en disco disponemos del *nombre* y del *dni* de un conjunto de alumnos. La estructura de cada registro del fichero es así:

```
typedef struct
{
  char nombre[60];
  unsigned long dni;
} alumno;
```

Se desea escribir un programa para visualizar los registros del fichero ordenados por el miembro *dni*. Para ello leeremos los registros del fichero y los almacenaremos en un árbol binario de búsqueda ordenado por el *dni*. Cada nodo del árbol será de la forma siguiente:

```
typedef struct elem
{
  alumno datos;       /* datos del nodo */
  struct elem *izdo; /* puntero al subárbol izquierdo */
  struct elem *dcho; /* puntero al subárbol derecho */
} nodo;
```

Se pide:

a) Escribir una función *insertar* que permita añadir nodos a una estructura en árbol binario de búsqueda. Los nodos estarán ordenados por el miembro *dni*.

```
nodo *insertar(nodo *raiz, alumno a);
```

El parámetro *raíz* es la raíz del árbol y *a* es el registro leído del fichero que hay que añadir al árbol.

b) Escribir una función *visu_ascen* para que recorra el árbol referenciado por *raíz* y visualice los datos en orden ascendente del miembro *dni*.

```
void visu_ascen(nodo *raiz);
```

c) Escribir una función *visu_descen* para que recorra el árbol referenciado por *raíz* y visualice los datos en orden descendente del miembro *dni*.

```
void visu_descen(nodo *raiz);
```

Utilizando las funciones anteriores, escribir un programa *listar* que reciba a través de la línea de órdenes el nombre de un fichero y el orden de presentación, y visualice los registros del fichero en el orden especificado:

```
listar -a fichero
listar -d fichero
```

donde *fichero* es el nombre del fichero cuyos registros queremos visualizar, *a* significa ascendentemente y *d* significa descendentemente.

4. El filtro *sort* lee líneas de texto del fichero estándar de entrada y las presenta en orden alfabético en el fichero estándar de salida. El ejemplo siguiente muestra la forma de utilizar *sort*:

```
sort↵
lo que puede hacerse
en cualquier momento
no se hará
en ningún momento.
(eof)
en cualquier momento
en ningún momento.
lo que puede hacerse
no se hará
```

Se desea escribir un programa de nombre *ordenar*, que actúe como el filtro *sort*. Para ordenar las distintas líneas vamos a ir insertándolas en un árbol binario de búsqueda, de tal forma que al recorrerlo podamos presentar las líneas en orden alfabético. Cada nodo del árbol se ajusta a la definición siguiente:

```
typedef struct datos
{
  char *linea; /* puntero a una línea de texto */
```

```
  struct datos *izq, *der;
} nodo;
```

Para realizar esta aplicación se pide escribir al menos las funciones siguientes:

a) Una función que lea líneas del fichero estándar de entrada y genere un árbol binario de búsqueda. El prototipo de esta función será así:

```
nodo *crear_arbol(void);
```

La función devolverá un puntero al nodo *raíz* del árbol creado.

b) Una función que recorra un árbol de las características anteriores y presente las líneas de texto que referencian sus nodos. El prototipo de esta función será:

```
void imprimir_arbol(nodo *a, char orden);
```

Los valores posibles del parámetro *orden* son: *a*, presentar las líneas en orden alfabético ascendente, y *b*, presentar las líneas en orden alfabético descendente.

Escribir un programa de nombre *ordenar* que responda a la funcionalidad siguiente:

```
ordenar↵
```

Lee líneas del fichero estándar de entrada y las presenta en el fichero estándar de salida en orden alfabético ascendente.

```
ordenar -r↵
```

Lee líneas del fichero estándar de entrada y las presenta en el fichero estándar de salida en orden alfabético descendente.

CAPÍTULO 12

ALGORITMOS

En este capítulo vamos a exponer cómo resolver problemas muy comunes en programación. El primer problema que nos vamos a plantear es la recursión; estos son problemas cuyo planteamiento forma parte de su solución. El segundo problema que vamos a abordar es la ordenación de objetos en general; este es un problema tan común que no necesita explicación. Algo tan cotidiano como una guía telefónica, es un ejemplo de una lista clasificada. El localizar un determinado teléfono exige una búsqueda por algún método. El problema de búsqueda será el último que resolveremos.

RECURSIVIDAD

Se dice que un proceso es recursivo si forma parte de sí mismo o sea que se define en función de sí mismo. La recursión aparece en la vida diaria, en problemas matemáticos, en estructuras de datos y en muchos otros problemas.

La recursión es un proceso extremadamente potente, por lo que la analizaremos detenidamente para saber cuándo y cómo aplicarla. Esto quiere decir que aunque un problema por definición sea recursivo, no siempre este será el método de solución más adecuado.

En las aplicaciones prácticas, antes de poner en marcha un proceso recursivo es necesario demostrar que el nivel máximo de recursión, esto es, el número de veces que se va a llamar a sí mismo, es no solo finito, sino realmente pequeño. La razón es que se necesita cierta cantidad de memoria para almacenar el estado del proceso cada vez que se abandona temporalmente, debido a una llamada para ejecutar un proceso que es él mismo. El estado del proceso de cálculo en curso hay que almacenarlo para recuperarlo cuando se acabe la nueva ejecución del proceso y haya que reanudar la antigua.

En términos de un lenguaje de programación, una función es recursiva cuando se llama a sí misma.

Un ejemplo es la función de Ackerman *A*, la cual está definida para todos los valores enteros no negativos *m* y *n* de la forma siguiente:

```
A(0,n) = n+1
A(m,0) = A(m-1,1)                (m > 0)
A(m,n) = A(m-1,A(m,n-1))         (m,n > 0)
```

El seudocódigo que especifica cómo solucionar este problema aplicando la recursión, es el siguiente:

```
<función A(m,n)>
 IF (m es igual a 0) THEN
  devolver como resultado n+1
 ELSE IF (n es igual a 0) THEN
  devolver como resultado A(m-1,1)
 ELSE
  devolver como resultado A(m-1,A(m,n-1))
 ENDIF
END <función A(m,n)>
```

A continuación presentamos el programa correspondiente a este problema.

```
/*************** FUNCIÓN DE ACKERMAN RECURSIVA ***************/
/* ackerman.c
 */
#include <stdio.h>

int Ackerman(int, int);

/********************** función principal **********************/
void main()
{
  int m, n, a;
  printf("Cálculo de A(m,n)=A(m-1,A(m,n-1))\n\n");
  printf("Valores de m y n : ");
  scanf("%d %d", &m, &n);
  a = Ackerman(m,n);
  printf("\n\nA(%d,%d) = %d\n",m,n,a);
}

/************************** Ackerman **************************/
int Ackerman(int m, int n)
{
  if (m == 0)
    return n+1;
  else if (n == 0)
    return Ackerman(m-1, 1);
```

```
  else
    return Ackerman(m-1, Ackerman(m,n-1));
}
```

Supongamos ahora que nos planteamos el problema de resolver la función de Ackerman, pero sin aplicar la recursión. Esto nos exigirá salvar las variables necesarias del proceso en curso, cada vez que la función se llame a sí misma, con el fin de poder reanudarlo cuando finalice el nuevo proceso invocado.

La mejor forma de hacer esto es utilizar una pila, con el fin de almacenar los valores *m* y *n* cada vez que se invoque la función para una nueva ejecución y tomar estos valores de la cima de la pila, cuando esta nueva ejecución finalice, con el fin de reanudar la antigua.

El seudocódigo para este programa puede ser el siguiente:

```
<función A(m,n)>
 Declarar un array para utilizarlo como una pila, con el fin de
 almacenar los valores de: m,n
 Inicializar la pila con los valores m,n
 DO
  Tomar los datos de la parte superior de la pila
  IF (m es igual a 0) THEN
   Amn = n+1
   IF (pila no vacía)
    sacar de la pila los valores: m, n
    meter en la pila los valores: m, Amn
   ELSE
    devolver como resultado Amn
   ENDIF
  ELSE IF (n es igual a 0) THEN
   meter en la pila los valores: m-1,1
  ELSE
   meter en la pila los valores: m-1, Amn
   meter en la pila los valores: m,n-1
  ENDIF
 WHILE (1)
END <función A(m,n)>
```

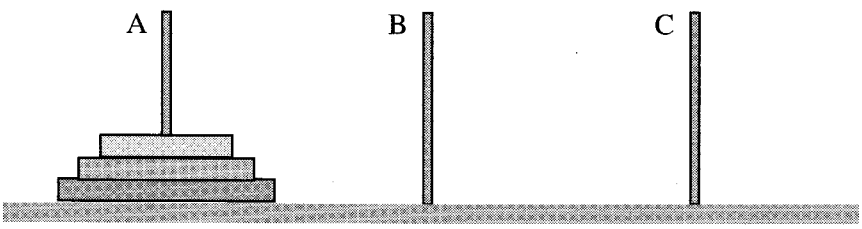
A continuación presentamos el programa correspondiente a este problema.

```
/************** FUNCIÓN DE ACKERMAN NO RECURSIVA **************/
/* acker_nr.c
 */
#include <stdio.h>
#include <stdlib.h>
```

```
typedef struct datos elemento;
typedef elemento *pelemento;
struct datos
{
  int m,n;
  pelemento siguiente;
};

void error(void)
{
  perror("error: no hay suficiente espacio en la pila\n\n");
  exit(1);
}

pelemento NuevoElemento()
{
  pelemento q = (pelemento)malloc(sizeof(elemento));
  if (!q) error();
  return (q);
}

int Ackerman(int, int);
void mete_pila(pelemento *, int, int);
void saca_pila(pelemento *, int *, int *);

/********************** función principal **********************/
void main()
{
  int m, n, a;
  printf("Cálculo de A(m,n)=A(m-1,A(m,n-1))\n\n");
  printf("Valores de m y n : ");
  scanf("%d %d", &m, &n);
  a = Ackerman(m,n);
  printf("\n\nA(%d,%d) = %d\n",m,n,a);
}

/************************** Ackerman **************************/
int Ackerman(int m, int n)
{
  pelemento pila = NULL;          /* pila de elementos (m,n) */
  int Ackerman_m_n = 0;

  mete_pila(&pila, m, n);

  while (1)
  {
    /* tomar los datos de la cima de la pila */
    saca_pila(&pila, &m, &n);
    if (m == 0) /* resultado para un elemento A(m,n) calculado */
    {
      Ackerman_m_n = n+1;
      if (pila)
      {
        saca_pila(&pila, &m, &n);
        mete_pila(&pila, m, Ackerman_m_n);
      }
      else
```

```
        return (Ackerman_m_n);
    }
    else if (n == 0)
      mete_pila(&pila, m-1, 1);
    else
    {
      mete_pila(&pila,m-1,Ackerman_m_n); /* n=Ackerman(m,n-1) */
      mete_pila(&pila, m, n-1);
    }
  }
}

/********************** meter en la pila **********************/
void mete_pila(pelemento *p, int m, int n)
{
  pelemento q;

  q = NuevoElemento();
  q->m = m, q->n = n;
  q->siguiente = *p;
  *p = q;
}

/********************** sacar de la pila **********************/
void saca_pila(pelemento *p, int *pm, int *pn)
{
  pelemento q = *p;                    /* cabecera de la pila */

  if (q == NULL)
  {
    printf("\nPila vacía\n");
    exit(2);
  }
  else
  {
    *pm = q->m, *pn = q->n;
    *p = q->siguiente;
    free(q);
  }
}
```

Un proceso en el cual es realmente eficaz aplicar la recursión es el problema de las *Torres de Hanoi*. Este problema consiste en tres barras verticales *A*, *B* y *C* y *n* discos, de diferentes tamaños, apilados inicialmente sobre la barra *A*, en orden de tamaño decreciente.



El objetivo es mover los discos desde la barra *A* a la barra *C*, bajo las siguientes reglas:

1. Se moverá un solo disco cada vez.
2. Un disco no puede situarse sobre otro más pequeño.
3. Se utilizará la barra *B* como pila auxiliar.

Una posible solución, es el algoritmo recursivo que se muestra a continuación:

1. Mover *n–1* discos de la barra *A* a la *B*.
2. Mover el disco *n* de la barra *A* a la *C*, y
3. Mover los *n–1* discos de la barra *B* a la *C*.

Resumiendo estas condiciones en un cuadro obtenemos:

	nº discos	*origen*	*otra torre*	*destino*
inicialmente	n	A	B	C
1	n–1	**A**	C	**B**
2	1	**A**	B	**C**
3	n–1	**B**	A	**C**

La función a realizar será mover *n* discos de origen a destino:

```
mover(n_discos, origen, otratorre, destino);
```

El seudocódigo para este programa puede ser el siguiente:

```
<función mover(n_discos, A, B, C)>
 IF (n_discos es mayor que 0) THEN
   mover(n_discos-1, A, C, B)
   mover(disco_n, A, B, C)
   mover(n_discos-1, B, A, C)
 ENDIF
END <función mover>
```

A continuación presentamos el programa correspondiente a este problema. El resultado será los movimientos realizados y el número total de movimientos.

```
/********************** Torres de Hanoi **********************/
/* hanoi.c
 */
#include <stdio.h>

int mover(int, char, char, char);

/*************************** main ****************************/
void main()
{
  int n_discos, movimientos;

  printf("Nº de discos : ");
  scanf("%d", &n_discos);
  movimientos = mover(n_discos, 'A', 'B', 'C');
  printf("\nmovimientos efectuados: %d\n", movimientos);
}

/*************************** mover ***************************/
int mover(int n_discos, char a, char b, char c)
{
  static int movimientos = 0;

  if (n_discos > 0)
  {
    mover(n_discos-1, a, c, b);
    printf("mover disco de  %c  a  %c\n", a, c);
    movimientos++;
    mover(n_discos-1, b, a, c);
  }
  return movimientos;
}
```

Si ejecuta este programa para *n_discos* = *3*, el resultado será el siguiente:

```
Nº de discos : 3
mover disco de  A  a  C
mover disco de  A  a  B
mover disco de  C  a  B
mover disco de  A  a  C
mover disco de  B  a  A
mover disco de  B  a  C
mover disco de  A  a  C

movimientos efectuados: 7
```

Como ejercicio se propone realizar este mismo problema, pero sin utilizar recursión.

CLASIFICACIÓN DE DATOS

Uno de los procedimientos más comunes y útiles en el procesamiento de datos, es la clasificación u ordenación de los mismos. Se considera ordenar al proceso de

reorganizar un conjunto dado de objetos en una secuencia determinada. El objetivo de este proceso generalmente es facilitar la búsqueda de uno o más elementos pertenecientes a un conjunto. Como, ejemplo considérense las listas de los alumnos matriculados en una cierta asignatura, las listas del censo, los índices alfabéticos de los libros, las guías telefónicas, etc. Esto quiere decir que muchos problemas están relacionados de alguna forma con el proceso de ordenación. Es por lo que la ordenación es un problema importante a considerar.

La ordenación, tanto numérica como alfanumérica, sigue las mismas reglas que empleamos nosotros en la vida normal. Esto es, un dato numérico es mayor que otro, cuando su valor es más grande, y una cadena de caracteres es mayor que otra, cuando está después por orden alfabético.

Podemos agrupar los métodos de ordenación en dos categorías: ordenación de arrays u ordenación interna, cuando los datos se guardan en memoria interna, y ordenación de ficheros u ordenación externa, cuando los datos se guardan en memoria externa, generalmente discos.

En este apartado no se trata de analizar exhaustivamente todos los métodos de ordenación y ver sus prestaciones de eficiencia, rapidez, etc. sino que simplemente analizamos desde el punto de vista práctico los métodos más comunes para ordenación de arrays y de ficheros.

Método de la burbuja

Hay muchas formas de clasificar datos y una de las más conocidas es la clasificación por el *método de la burbuja.*

Veamos a continuación el algoritmo correspondiente, para ordenar una lista de menor a mayor, partiendo de que los datos a ordenar están en una lista de *n* elementos:

1. Comparamos el primer elemento con el segundo, el segundo con el tercero, el tercero con el cuarto, etc. Cuando el resultado de una comparación sea "mayor que", se intercambian los valores de los elementos comparados. Con esto conseguimos llevar el valor mayor a la posición *n*.

2. Repetimos el punto 1, ahora para los *n–1* primeros elementos de la lista. Con esto conseguimos llevar el valor mayor de éstos a la posición *n–1*.

3. Repetimos el punto 1, ahora para los *n–2* primeros elementos de la lista y así sucesivamente.

4. El proceso termina después de repetir el proceso descrito, *n–1* veces, o cuando al finalizar la ejecución del *iésimo* proceso no haya habido ningún cambio.

El seudocódigo para este algoritmo puede ser el siguiente:

```
<función clasificar(array "a" de "n" elementos)>
["a" es un array cuyos elementos son a0, a1, ..., an-1]
 n = n-1
 DO WHILE ("a" no esté ordenado y n > 0 )
  i = 1
  DO WHILE ( i <= n )
   IF ( a[i-1] > a[i] ) THEN
    permutar a[i-1] con a[i]
   ENDIF
   i = i+1
  ENDDO
  n = n-1
 ENDDO
END <clasificar>
```

El siguiente ejemplo presenta la programación de este algoritmo para el caso concreto de ordenar alfabéticamente una lista de cadenas de caracteres.

```
/****************** Clasificar líneas de texto *****************/
/* burbuja.c
 */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define MAXC 81      /* número máximo de caracteres por línea */

int LeerLineas(char **, int);
void Clasificar(char **, int);
void EscribirLineas(char **, int);

void main()
{
  char **lineas;  /* puntero al array que contiene las líneas */
  int lineasmax;          /* número máximo de líneas a ordenar */
  int nlineas;                    /* número de líneas leídas */

  printf("nº máximo de líneas a ordenar: ");
  scanf("%d", &lineasmax);
  fflush(stdin);
  /* Asignación de memoria para lineas[lineasmax][MAXC] */
  if (!(lineas = (char **)malloc(lineasmax * sizeof(char *))))
  {
    perror("Insuficiente espacio de memoria\n");
    exit(1);
  }
```

```
  for (i = 0; i < lineasmax; i++)
  {
    if (!(lineas[i] = (char *)malloc(MAXC)))
    {
      perror("Insuficiente espacio de memoria\n");
      exit(1);
    }

  }

  printf("Proceso de clasificación de líneas de caracteres.\n");
  printf("Cada línea finaliza con Enter.\n\n");
  printf("Entrada de líneas; eof para finalizar.\n");

  if ((nlineas = LeerLineas(lineas, lineasmax)) > 0)
  {
    printf("Proceso de clasificación.\n\n");
    Clasificar(lineas, nlineas);
    EscribirLineas(lineas, nlineas);
  }
  else
    printf("No hay líneas para clasificar.\n");

  /* Liberar la memoria asignada al array */
  for (i = 0; i < lineasmax; i++)
    free(lineas[i]);
  free(lineas);
}

/********************** Leer Líneas ***************************/
int LeerLineas(char **lineas, int lineasmax)
{
  int nlineas;
  char *p;

  nlineas = 0;
  /* leer n líneas */
  while ((p = gets(lineas[nlineas++])) != NULL)
  {
    if (nlineas == lineasmax)
      return (nlineas);
  }
  return (nlineas-1);              /* número de líneas leídas */
}

/************************ Clasificar ***************************/
void Clasificar(char **lineas, int numero_de_lineas)
{
  char aux[MAXC];
  int i, s;

  s = 1;
  while ((s == 1) && (--numero_de_lineas > 0))
  {
    s = 0;                                    /* no permutación */
    for (i = 1; i <= numero_de_lineas; i++)
      /* ¿ la línea (i-1) es mayor que la línea (i) ? */
```

```
      if (strcmp(lineas[i-1], lineas[i]) > 0)
      {
        /* permutar las lineas (i-1) e (i) */
        strcpy(aux, lineas[i-1]);
        strcpy(lineas[i-1], lineas[i]);
        strcpy(lineas[i], aux);
        s = 1;                                    /* permutación */
      }
  }
}

/******************** Escribir Líneas ************************/
void EscribirLineas(char **lineas, int nlineas)
{
  int i = 0;

  while (nlineas-- > 0)
    printf("%s\n", lineas[i++]);
}
```

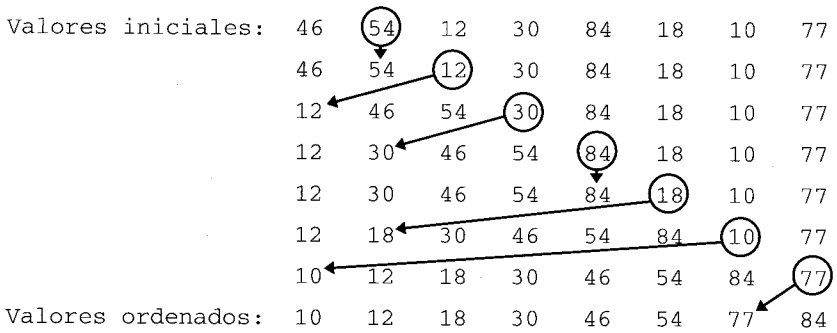
Observe que *s* inicialmente vale *0* para cada iteración, y toma el valor *1* cuando, al menos, se efectúa un cambio entre dos elementos. Si en una exploración a lo largo de la lista, no se efectúa cambio alguno, *s* permanecerá valiendo *0*, lo que indica que la lista está ordenada, terminando así el proceso.

Cuando se analiza un método de ordenación, hay que determinar cuántas comparaciones e intercambios se realizan para el caso más favorable, para el caso medio y para el caso más desfavorable.

En el método de la burbuja se realizan $(n-1)(n/2)=(n^2-n)/2$ comparaciones en el caso más desfavorable, donde n es el número de elementos a ordenar. Para el caso más favorable, la lista está ordenada, el número de intercambios es *0*. Para el caso medio es $3(n^2-n)/4$; hay tres intercambios por cada elemento desordenado. Y para el caso menos favorable, el número de intercambios es $3(n^2-n)/2$. El análisis matemático que conduce a estos valores, queda fuera del propósito de este libro. El tiempo de ejecución es un múltiplo de n^2 y está directamente relacionado con el número de comparaciones y de intercambios.

Método de inserción

El algoritmo para este método de ordenación es el siguiente: inicialmente, se ordenan los dos primeros elementos del array, luego se inserta el tercer elemento en la posición correcta con respecto a los dos primeros, a continuación se inserta el cuarto elemento en la posición correcta con respecto a los tres primeros elementos ya clasificados y así sucesivamente hasta llegar al último elemento del array. Por ejemplo,



El seudocódigo para este algoritmo puede ser el siguiente:

<función inserción(array "a" de "n" elementos)>
["a" es un array cuyos elementos son a_0, a_1, ..., a_{n-1}]
i = 1
DO WHILE (i < n)
x = a[i]
insertar x en la posición correcta entre a_0 y a_i
ENDDO
END <inserción>

La programación de este algoritmo, para el caso concreto de ordenar numéricamente una lista de números, es la siguiente:

```
/****************** Ordenación por inserción ******************/
/* inserc.c
 */
#include <stdio.h>
#include <stdlib.h>

void insercion(int [], int);

void main()
{
  static int lista[] = { 46, 54, 12, 30, 84, 18, 10, 77 };
  int n_elementos = sizeof(lista)/sizeof(int);
  register i;

  insercion(lista, n_elementos);

  printf("Lista ordenada:\n");
  for (i = 0; i < n_elementos; i++)
    printf("%6d", lista[i]);
}

/************************ inserción ****************************/
void insercion(int lista[], int n_elementos)
{
```

```
  int i, k, x;

  /* desde el segundo elemento */
  for (i = 1; i < n_elementos; i++)
  {
    x = lista[i];
    k = i-1;
    while (k >=0 && x < lista[k]) /* para k=-1, se ha alcanzado*/
    {                             /* el extremo izquierdo.     */
      lista[k+1] = lista[k];      /* hacer hueco para insertar */
      k--;
    }
    lista[k+1] = x;                  /* insertar x en su lugar */
  }
}
```

Análisis del método de inserción directa:

	comparaciones	*intercambios*
caso más favorable	n–1	2(n–1)
caso medio	$(n^2+n-2)/4$	$(n^2+9n-10)/4$
caso menos favorable	$(n^2+n)/2-1$	$(n^2+3n-4)/2$

Para el método de inserción, el tiempo de ejecución es función de n^2 y está directamente relacionado con el número de comparaciones y de intercambios.

Método quicksort

El método de ordenación *quicksort*, está generalmente considerado como el mejor algoritmo de ordenación disponible actualmente.

El algoritmo correspondiente a este método, es el siguiente:

1. Se selecciona un valor perteneciente al rango de valores del array. Este valor se puede escoger aleatoriamente, o haciendo la media de un pequeño conjunto de valores tomados del array. El valor óptimo sería la mediana (el valor que es menor o igual que los valores correspondientes a la mitad de los elementos del array y mayor o igual que los valores correspondientes a la otra mitad). No obstante, incluso en el peor de los casos (el valor escogido está en un extremo), *quicksort* funciona correctamente.

2. Se divide el array en dos partes, una con todos los elementos menores que el valor seleccionado y otra con todos los elementos mayores o iguales.

3. Se repiten los puntos 1 y 2 para cada una de las partes en la que se ha dividido el array, hasta que esté ordenado.

   El proceso descrito es esencialmente recursivo.

   El seudocódigo para este algoritmo puede ser el siguiente:

```
<función qs(array "a")>
 Se elige un valor x del array
 DO WHILE ( "a" no esté dividido en dos partes )
  [dividir "a" en dos partes: a_inf y a_sup]
  a_inf con los elementos a_i < x
  a_sup con los elementos a_i >= x
 ENDDO
 IF ( existe a_inf ) THEN
  qs( a_inf )
 ENDIF
 IF ( existe a_sup ) THEN
  qs( a_sup)
 ENDIF
END <qs>
```

A continuación se muestra una versión de este algoritmo, que selecciona el elemento medio del array para proceder a dividirlo en dos partes. Esto resulta fácil de implementar, aunque no siempre da lugar a una buena elección. A pesar de ello, funciona correctamente.

```
/*********** Ordenación quicksort. Método recursivo ***********/
/* qsort.c
 */
#include <stdio.h>
#include <stdlib.h>

void quicksort(int [], int);

void main()
{
  static int lista[] = { 10, 3, 7, 5, 12, 1, 27, 3, 8, 13 };
  int n_elementos = sizeof(lista)/sizeof(int);

  register i;

  quicksort(lista, n_elementos);

  printf("Lista ordenada:\n");
```

```
  for (i = 0; i < n_elementos; i++)
    printf("%6d", lista[i]);
}

/************************* Quicksort ************************/
void qs(int [], int, int);

void quicksort(int lista[], int n_elementos)
{
  qs(lista, 0, n_elementos - 1);
}

/* Función recursiva qs */
void qs(int lista[], int inf, int sup)
{
  register izq, der;
  int mitad, x;

  izq = inf; der = sup;
  mitad = lista[(izq+der)/2];
  do
  {
    while (lista[izq] < mitad && izq < sup) izq++;
    while (mitad < lista[der] && der > inf) der--;
    if (izq <= der)
    {
      x = lista[izq], lista[izq] = lista[der], lista[der] = x;
      izq++; der--;
    }
  }
  while (izq <= der);
  if (inf < der) qs(lista, inf, der);
  if (izq < sup) qs(lista, izq, sup);
}
```

Observamos que cuando el valor *mitad* se corresponde con uno de los valores de la lista, las condiciones *izq* < *sup* y *der* > *inf* de las sentencias

```
while (lista[izq] < mitad && izq < sup) izq++;
while (mitad < lista[der] && der > inf) der--;
```

no serían necesarias. En cambio, si el valor *mitad* es un valor no coincidente con un elemento de la lista, pero que está dentro del rango de valores al que pertenecen los elementos de la misma, esas condiciones son necesarias para evitar que se puedan sobrepasar los límites de los índices del array. Para experimentarlo, pruebe como ejemplo la lista *1 1 3 1 1* y elija *mitad* = *2* fijo.

En el método *quicksort*, en el caso más favorable, esto es, cada vez se selecciona la mediana obteniéndose dos particiones iguales, se realizan *n.log n* comparaciones y *n/6.log n* intercambios, donde *n* es el número de elementos a ordenar; en el caso medio, el rendimiento es inferior al caso óptimo en un factor de *2.log 2*; y en el caso menos favorable, esto es, cada vez se selecciona el valor mayor ob-

teniéndose una partición de $n-1$ elementos y otra de un elemento, el rendimiento es del orden de $n.n=n^2$. Con el fin de mejorar el caso menos favorable, se sugiere elegir, cada vez, un valor aleatoriamente o un valor que sea la mediana de un pequeño conjunto de valores tomados del array.

La función *qs* sin utilizar la recursión puede desarrollarse de la forma siguiente:

```
/************************** quicksort ************************/
/* qsort_nr.c
 */
void qs(int [], int, int);

void quicksort(int lista[], int n_elementos)
{
  qs(lista, 0, n_elementos - 1);
}

/* Función no recursiva qs */
void qs(int lista[], int inf, int sup)
{
  #define NE 100
  typedef struct
  {
    int inf, sup;
  } elemento_pila;

  static elemento_pila pila[NE];
  register izq, der;
  int mitad, x, p;

  /* Inicializar la pila con los valores: inf, sup */
  p = 1, pila[p].inf = inf, pila[p].sup = sup;
  do
  {
   /* tomar los datos inf, sup de la parte superior de la pila */
    inf = pila[p].inf, sup = pila[p].sup, p--;
    do
    { /* División del array en dos partes */
      izq = inf; der = sup;
      mitad = lista[(izq+der)/2];
      do
      {
      while (lista[izq] < mitad && izq < sup) izq++;
      while (mitad < lista[der] && der > inf) der--;
      if (izq <= der)
      {
        x=lista[izq], lista[izq]=lista[der], lista[der]=x;
        izq++; der--;
      }
      }
      while (izq <= der);
      if (izq < sup)
      { /* meter en la pila los valores: izq, sup */
```

```
            p++, pila[p].inf = izq, pila[p].sup = sup;
        }
        /* inf = inf; */    sup = der;
    }
    while (inf < der);
  }
  while (p);
}
```

En esta solución observamos que después de cada paso se generan dos nuevas sublistas. Una de ellas la tratamos en la siguiente iteración y la otra la posponemos, guardando sus límites *inf* y *sup* en una pila.

Comparación de los métodos expuestos

La tabla siguiente muestra los tiempos, en segundos, consumidos por los métodos de ordenación estudiados anteriormente, para realizar la ordenación de un array de 1000 elementos de tipo *float* (lógicamente estos tiempos son orientativos ya que serán menores o mayores dependiendo del hardware de su máquina).

	ordenado	*inverso*	*aleatorio*
burbuja	0.33 seg	253 seg	94 seg
inserción	0.27 seg	158 seg	0.44 seg
quicksort	1.7 seg	1.7 seg	1.7 seg

El significado de cada una de las columnas es el siguiente:

ordenado: tiempo para ordenar un array que inicialmente ya está ordenado.

inverso: tiempo para ordenar un array que inicialmente está ordenado en sentido inverso.

aleatorio: tiempo para ordenar un array que inicialmente ya está ordenado, pero, al que se le han permutado aleatoriamente dos de sus elementos.

El método de la burbuja es el peor de los métodos; el método de inserción directa mejora considerablemente y el método quicksort es el más rápido y mejor método de ordenación de arrays con diferencia.

BÚSQUEDA DE DATOS

El objetivo de ordenar un conjunto de objetos, generalmente es facilitar la búsqueda de uno o más elementos pertenecientes a un conjunto; aunque es posible realizar dicha búsqueda sin que el conjunto de objetos esté ordenado, pero esto trae como consecuencia un mayor tiempo de proceso.

Búsqueda secuencial

Este método de búsqueda, aunque válido, es el menos eficiente. Se basa en comparar el valor que se desea buscar con cada uno de los valores del array. El array no tiene por qué estar clasificado.

El seudocódigo para este método de búsqueda puede ser el siguiente:

```
<función Búsqueda_S( array a, valor que queremos buscar)>
 i = 0
 DO WHILE ( no encontrado )
  IF ( valor = a[i] )
   encontrado
  ENDIF
  i = i+1
 ENDDO
END <Búsqueda_S>
```

Como ejercicio, escribir el código correspondiente a un programa que permita buscar un valor en un array previamente leído.

Búsqueda binaria

Un método eficiente de búsqueda, que puede aplicarse a los *arrays clasificados*, es la búsqueda binaria. Partiendo de que los elementos del array están almacenados en orden ascendente, la búsqueda binaria consiste en lo siguiente:

Se selecciona el elemento del centro o aproximadamente del centro del array. Si el valor a buscar no coincide con el elemento seleccionado y es mayor que él, se continúa la búsqueda en la segunda mitad del array. Si, por el contrario, el valor a buscar es menor que el valor del elemento seleccionado, la búsqueda continúa en la primera mitad del array. En ambos casos, se halla de nuevo el elemento central, correspondiente al nuevo intervalo de búsqueda, repitiéndose el ciclo. El proceso se repite hasta que se encuentra el valor a buscar, o bien hasta que el in-

tervalo de búsqueda sea nulo, lo que querrá decir que el elemento buscado no figura en el array.

El seudocódigo para este algoritmo puede ser el siguiente:

```
<función Búsqueda_B( array a, valor que queremos buscar )>
 DO WHILE ( exista un intervalo donde buscar )
  x = elemento mitad del intervalo de búsqueda
  IF ( valor = x ) THEN
   encontrado
  ELSE
   IF ( valor > x ) THEN
    buscar "valor" en la segunda mitad del intervalo de búsqueda
   ENDIF
   IF ( valor < x ) THEN
    buscar "valor" en la primera mitad del intervalo de búsqueda
   ENDIF
  ENDIF
 ENDDO
END <Búsqueda_B>
```

A continuación se muestra el programa correspondiente a este método.

```
/********************** Búsqueda Binaria **********************/
/* busbin.c
 */
#include <stdio.h>
#include <stdlib.h>

int BusquedaB(float [], float, int, int);

void main()
{
  /* Array ordenado */
  static float lista[] = { 3, 6, 9, 12, 15, 18, 21, 24, 27, 30,
                           33, 36, 39, 42, 45, 48, 51, 54, 57, 60 };
  /* Número de elementos del array */
  const int n_elementos = sizeof(lista)/sizeof(float);
  int posicion;
  float valor;

  printf("Introducir el valor a buscar ");
  scanf("%f", &valor);
  posicion = BusquedaB(lista, valor, 0, n_elementos-1);
  if (posicion != -1)
    printf("\nLa posición de %g es %d\n", valor, posicion);
  else
    printf("\nEl valor %g no está en la lista\n", valor);
}
```

```
/************ Función recursiva búsqueda binaria ***************
 *
 * La función devuelve como resultado la posición del valor.
 * Si el valor no se localiza se devuelve un resultado -1.
 */
int BusquedaB(float lista[], float v, int inf, int sup)
{
  int mitad;

  /* Comprobar si la búsqueda falla */
  if ((inf >= sup) && (lista[inf] != v))
  {
    /* Búsqueda sin éxito */
    return -1;
  }

  /* Sigue la búsqueda */
  mitad = (inf + sup) / 2;
  if (lista[mitad] == v)
  {
    /* Búsqueda con éxito */
    /* El primer elemento ocupa la posición 0 */
    return mitad;
  }
  else
    if (v > lista[mitad])
      BusquedaB(lista, v, mitad + 1, sup);
    else
      BusquedaB(lista, v, inf, mitad - 1);
}
```

Búsqueda de cadenas

Uno de los métodos más eficientes en la búsqueda de cadenas dentro de un texto es el algoritmo Boyer y Moore. La implementación básica de este método, construye una tabla *delta* que se utilizará en la toma de decisiones durante la búsqueda de una subcadena. Dicha tabla contiene un número de entradas igual al número de caracteres del código que se esté utilizando. Por ejemplo, si se está utilizando el código de caracteres ASCII, la tabla será de 256 entradas. Cada entrada contiene el valor *delta* asociado con el carácter que representa. Por ejemplo, el valor *delta* asociado con *A* estará en la entrada 65 y el valor *delta* asociado con el espacio en blanco, en la entrada 32. El valor *delta* para un carácter, es la posición de la ocurrencia más a la izquierda de ese carácter respecto a la posición final en la cadena buscada. Las entradas correspondientes a los caracteres que no pertenecen a la cadena a buscar, tienen un valor igual a la longitud de esta cadena.

Por lo tanto, para definir la tabla *delta* para una determinada subcadena a buscar, construimos un array con todos sus elementos inicializados a la longitud de dicha cadena, y luego, asignamos el valor *delta* para cada carácter de la subcadena, así:

```
for ( i = 1; i <= longitud_cadena_patrón; i++ )
  delta[cadena_patron[i - 1]] = longitud_cadena_patron - i;
```

En el algoritmo de Boyer y Moore, la comparación se realiza de derecha a izquierda, empezando desde el principio del texto. Es decir, se empieza comparando el último carácter de la cadena que se busca con el correspondiente carácter en el texto donde se busca; si los caracteres no coinciden, la cadena que se busca se desplaza hacia la derecha un número de caracteres igual al valor indicado por la entrada en la tabla *delta* correspondiente al carácter del texto que no coincide. Si el carácter no aparece en la cadena que se busca, su valor *delta* es la longitud de la cadena que se busca.

Veamos un ejemplo. Suponga que se desea buscar la cadena "cien" en el texto "Más vale un ya que cien después se hará". La búsqueda comienza así:

```
Texto:              Más vale un ya que cien después se hará
Cadena a buscar:    cien
```

El funcionamiento del algoritmo puede comprenderse mejor situando la cadena a buscar paralela al texto. La comparación es de derecha a izquierda; por lo tanto, se compara el último carácter en la cadena a buscar (*n*) con el carácter que está justamente encima en el texto (*espacio*). Como *n* es distinto de *espacio*, la cadena que se busca debe desplazarse a la derecha un número de caracteres igual al valor indicado por la entrada en la tabla *delta* que corresponde al carácter del texto que no coincide. Para la cadena "cien",

```
delta['c'] = 3
delta['i'] = 2
delta['e'] = 1
delta['n'] = 0
```

El resto de las entradas valen 4, longitud de la cadena. Según esto, la cadena que se busca se desplaza cuatro posiciones a la derecha (el espacio en blanco no aparece en la cadena que se busca).

```
Texto:              Más vale un ya que cien después se hará
Cadena a buscar:        cien
```

Ahora, *n* no coincide con *e*; luego la cadena se desplaza una posición a la derecha (*e* tiene un valor asociado de uno).

```
Texto:              Más vale un ya que cien después se hará
Cadena a buscar:         cien
```

n no coincide con *espacio*; se desplaza la cadena cuatro posiciones a la derecha.

```
Texto:              Más vale un ya que cien después se hará
Cadena a buscar:                 cien
```

n no coincide con *y*; se desplaza la cadena cuatro posiciones a la derecha.

```
Texto:              Más vale un ya que cien después se hará
Cadena a buscar:                cien
```

n no coincide con *u*; se desplaza la cadena cuatro posiciones a la derecha.

```
Texto:              Más vale un ya que cien después se hará
Cadena a buscar:                    cien
```

n no coincide con *i*; se desplaza la cadena dos posiciones a la derecha.

```
Texto:              Más vale un ya que cien después se hará
Cadena a buscar:                      cien
```

Todos los caracteres de la cadena coinciden con los correspondientes caracteres en el texto. Para encontrar la cadena se han necesitado sólo 7 comparaciones; el algoritmo directo habría realizado 20 comparaciones.

El algoritmo Boyer-Moore es más rápido porque tiene información sobre la cadena que se busca, en la tabla *delta*. El carácter que ha causado la no coincidencia en el texto indica cómo mover la cadena respecto del texto. Si el carácter no coincidente en el texto no existe en la cadena, ésta puede moverse sin problemas a la derecha un número de caracteres igual a su longitud, pues es un gasto de tiempo comparar la cadena con un carácter que ella no contiene. Cuando el carácter no coincidente en el texto está presente en la cadena, el valor *delta* para ese carácter alinea la ocurrencia más a la derecha de ese carácter en la cadena, con el carácter en el texto.

A continuación se muestra el código correspondiente a este método. La función *BuscarCadena* es la que realiza el proceso descrito. Esta función devuelve la posición de la cadena en el texto o un -1 si la cadena no se encuentra.

```
/******************* Búsqueda de subcadenas *******************/
/* cadenas.c
 */
#include <stdio.h>
#include <string.h>

/************************ Tabla delta ************************/
void TablaDelta(int delta[256], char *cadena)
{
  int i, LongCad = strlen(cadena);

  /* Inicializar la tabla delta */
  for (i = 0; i < 256; i++)
    delta[i] = LongCad;
```

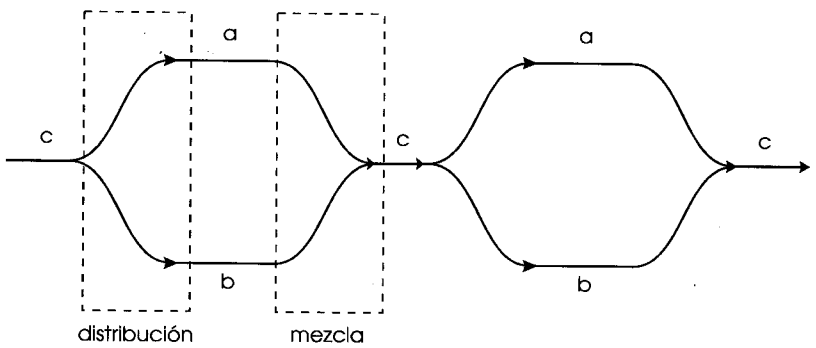
```
  /* Asignar valores a la tabla */
   for (i = 1; i <= LongCad; ++i)
      delta[cadena[i - 1]] = LongCad - i;
}

/***************** Algoritmo de Boyer y Moore *****************/
int BuscarCadena( char *texto, char *cadena)
{
  int delta[256];
  int i, j, LongCad, LongTex = strlen(texto);

  /* Longitud de la cadena a buscar */
   LongCad = strlen(cadena);

  /* i es el índice dentro del texto */
  i = LongCad;

  TablaDelta(delta, cadena);

  while (i <= LongTex)
  {
    /* j es un índice dentro de la cadena a buscar */
    j = LongCad;

    /* Mientras haya coincidencia de caracteres */
    while (cadena[j - 1] == texto[i - 1])
    {
      if ( j > 1 )
      {
        // Siguiente posición a la izquierda
        j--;
        i--;
      }
      else
      {
        /* Se llegó al principio de la cadena, luego se encontró */
        return (i - 1);
      }
    }
    /* Los caracteres no coinciden. Mover i lo que indique el
     * valor "delta" del carácter del texto que no coincide
     */
    i += delta[texto[i - 1]];
  }
  return -1;
}

void main()
{
  char *texto = "Más vale un ya que cien después se hará";
  char *cadena = "cien";

  if ( BuscarCadena(texto, cadena) != -1 )
    printf("La cadena %s está en el texto\n", cadena);
  else
    printf("La cadena %s no está en el texto\n", cadena);
}
```

ORDENACIÓN DE FICHEROS EN DISCO

Para ordenar un fichero, dependiendo del tamaño del mismo, podremos proceder de alguna de las dos formas siguientes. Si el fichero es pequeño, tiene pocos registros, se puede copiar en memoria en un array y utilizando las técnicas vistas anteriormente, ordenamos dicho array y a continuación copiamos el array ordenado de nuevo en el fichero. Sin embargo, muchos ficheros son demasiado grandes y no cabrían en un array en memoria, por lo que para ordenarlos recurriremos a técnicas que actúen sobre el propio fichero.

Ordenación de ficheros. Acceso secuencial

El siguiente programa desarrolla un algoritmo de ordenación de un fichero utilizando el acceso secuencial, denominado *mezcla natural*. La secuencia inicial de los elementos, viene dada en el fichero *c* y se utilizan dos ficheros auxiliares denominados *a* y *b*. Cada pasada consiste en una *fase de distribución* que reparte equitativamente los tramos ordenados del fichero *c* sobre los ficheros *a* y *b*, y una *fase que mezcla* los tramos de los ficheros *a* y *b* sobre el fichero *c*.



Este proceso se ilustra en el ejemplo siguiente. Partimos de un fichero *c*. Con el fin de ilustrar el método de *mezcla natural*, separaremos los tramos ordenados en los ficheros, por un guión (-).

```
fichero c:  18  32 - 10  60 - 14  42  44  68 - 12  24  30  48
```

Fase de distribución:

```
fichero a:  18  32 - 14  42  44  68
fichero b:  10  60 - 12  24  30  48
```

Fase de mezcla:

```
fichero c:  10  18  32  60 - 12  14  24  30  42  44  48  68
```

Fase de distribución:

```
fichero a:  10  18  32  60
fichero b:  12  14  24  30  42  44  48  68
```

fase de mezcla:

```
fichero c:  10  12  14  18  24  30  32  42  44  48  60  68
```

Para dejar ordenado el fichero del ejemplo hemos necesitado realizar dos pasadas. El proceso finaliza, tan pronto como el número de tramos ordenados del fichero *c*, sea *1*.

Una forma de reducir el número de pasadas es distribuir los tramos ordenados sobre más de dos ficheros.

Según lo expuesto, el algoritmo de ordenación *mezcla natural* podría ser así:

```
<función mezcla_natural()>
 DO
  [ Crear y abrir los ficheros temporales a y b]
  distribucion();
  n_tramos = 0;
  mezcla();
 WHILE (n_tramos != 1);
END <mezcla_natural()>
```

La estructura del programa está formada por las funciones siguientes:

```
void main(int argc, char *argv[]);
void mezcla_natural(FILE *f);
int distribuir(FILE *f, FILE *fa, FILE *fb);
int mezclar(FILE *fa, FILE *fb, FILE *f);
```

La función **main** analiza los parámetros en la línea de órdenes, abre el fichero a ordenar y llama a la función de *mezcla natural*.

La función *mezcla_natural* hace sucesivas llamadas a la funciones *distribuir* y *mezclar* hasta que el fichero quede ordenado.

El programa completo se muestra a continuación.

```
/* Método de ordenación mezcla natural para ficheros
 * accedidos secuencialmente.
 *
 * ord_sec.c
 *
```

```
 * El nombre del fichero a ordenar se recibe a través de la
 * línea de órdenes y debe ser un fichero de texto. La ordenación
 * se realiza por líneas con objeto de que éstas queden dispuestas
 * en orden alfabético ascendente.
 *
 */
#include <stdio.h>
#include <stdlib.h>
#define MAX 256

void mezcla_natural(FILE *f);
int distribuir(FILE *f, FILE *fa, FILE *fb);
int mezclar(FILE *fa, FILE *fb, FILE *f);

void main(int argc, char *argv[])
{
  FILE *pfichero;
  char respuesta, str[MAX];

  /* Análisis de los parámetros de la línea de órdenes. */
  if (argc != 2)
  {
    fprintf(stderr, "Sintaxis: %s fichero", argv[0]);
    exit(-1);
  }
  /* Apertura del fichero. */
  if ((pfichero = fopen(argv[1], "r+")) == NULL)
  {
    perror(argv[1]);
    exit(-1);
  }
  /* Ordenación. */
  mezcla_natural(pfichero);
  do
  {
    printf("¿Desea visualizar el fichero? (s/n)  ");
    respuesta = getchar();
    fflush(stdin);
  }
  while (tolower(respuesta) != 's' && tolower(respuesta) != 'n');
  /* Salida de datos */
  if (respuesta == 's')
  {
    rewind(pfichero);
    while (fgets(str, MAX, pfichero))
      printf("%s", str);
  }
  if (ferror(pfichero))
    perror("Error durante la lectura");

  fclose(pfichero);
}

/********************** Mezcla natural **********************/
void mezcla_natural(FILE *f)
{
  FILE *fa, *fb;
```

```
  int nro_tramos;
  do
  {
    fa = tmpfile(); /* fichero temporal */
    fb = tmpfile(); /* fichero temporal */
    rewind(f);
    nro_tramos = distribuir(f, fa, fb);
    if (nro_tramos <= 1)
    {
      rmtmp();
      return;
    }
    rewind(f),    rewind(fa), rewind(fb);
    nro_tramos = mezclar(fa, fb, f);
    /* Eliminar los ficheros temporales */
    fclose(fa);
    fclose(fb);
  }
  while (nro_tramos != 1);
} /* mezcla_natural */

/******************** Fase de distribución ********************/
int distribuir(FILE *f, FILE *fa, FILE *fb)
{
  FILE *faux = fa;
  char str[MAX];
  char str_ant[MAX];
  int nro_tramos = 1;

  if (fgets(str_ant, MAX, f))
    fputs(str_ant, fa);
  else
    return 0;

  while (fgets(str, MAX, f))
  {
    if (strcmp(str, str_ant) < 0)
    {
      /* Cambiar al otro fichero */
      faux = (faux == fa) ? fb : fa;
      ++nro_tramos;
    }
    strcpy(str_ant, str);
    fputs(str, faux);
  }
  return (nro_tramos);
} /* distribuir */

/*********************** Fase de mezcla ***********************/
int mezclar(FILE *fa, FILE *fb, FILE *f)
{
  char stra[256], strb[256], stra_ant[256], strb_ant[256];
  int nro_tramos = 1;

  /* Leemos las dos primeras cadenas */
  fgets(stra, MAX, fa);
  strcpy(stra_ant, stra);
```

```
fgets(strb, MAX, fb);
strcpy(strb_ant, strb);

/* Vamos leyendo y comparando hasta que se acabe alguno de los
 * ficheros. La fusión se realiza entre pares de tramos
 * ordenados. Un tramo de fa y otro de fb darán lugar a un
 * tramo ordenado sobre f.
 */
while (!feof(fa) && !feof(fb))
{
  if (strcmp(stra, strb) < 0)        /* 1 */
  {
    if (strcmp(stra, stra_ant) < 0)  /* 2 */
    /* Encontrado el final del tramo de A */
    {
      strcpy(stra_ant, stra);
      /* Copiamos el tramo ordenado del fichero B */
      do
      {
        fputs(strb, f);
        strcpy(strb_ant, strb);
      }
      while (fgets(strb,MAX,fb) && strcmp(strb,strb_ant) > 0);
      ++nro_tramos;
    }
    else /* 2 */
    {
      /* Copiamos la cadena leída del fichero A */
      strcpy(stra_ant, stra);
      fputs(stra, f);
      fgets(stra, MAX, fa);
    }
  }
  else /* 1 */
  {
    if (strcmp(strb, strb_ant) < 0)  /* 3 */
    /* Encontrado el final del tramo de B */
    {
      strcpy(strb_ant, strb);
      /* Copiamos el tramo ordenado del fichero A */
      do
      {
        fputs(stra, f);
        strcpy(stra_ant, stra);
      }
      while (fgets(stra,MAX,fa) && strcmp(stra,stra_ant) > 0);
      ++nro_tramos;
    }
    else /* 3 */
    {
      /* Copiamos la cadena leída del fichero B. */
      strcpy(strb_ant, strb);
      fputs(strb, f);
      fgets(strb, MAX, fb);
    }
  }
} /* while */
```

```
  /* Caso de acabarse primero el fichero B */
  if (feof(fb))
  {
    fputs(stra, f);
    while (fgets(stra, MAX, fa))
      fputs(stra, f);
  }
  /* Caso de acabarse primero el fichero A */
  else if (feof(fa))
  {
    fputs(strb, f);
    while (fgets(strb, MAX, fb))
      fputs(strb, f);
  }

  return (nro_tramos);
} /* mezclar */
```

Ordenación de ficheros. Acceso aleatorio

El acceso aleatorio a un fichero, a diferencia del secuencial, permite ordenar la información contenida en el mismo sin tener que copiarla sobre otro fichero, para lo que le aplicaremos un proceso análogo al aplicado a los arrays, lo que simplifica enormemente el proceso ordenación. Esto quiere decir que los métodos expuestos para ordenar arrays, pueden ser aplicados también para ordenar ficheros utilizando el acceso aleatorio.

El siguiente programa ordena un fichero pasado como argumento en la línea de órdenes, en el cual cada registro está formado por dos campos: *denominación* y *precio*. El desarrollo del programa variará en función de la estructura de los datos y del tipo del campo (numérico o alfanumérico) que se utilice para la ordenación del fichero. Nosotros vamos a ordenar el fichero por el campo *denominación*, de tipo alfabético, empleando el método *quicksort* explicado anteriormente en este mismo capítulo.

La estructura del programa está formada por las funciones:

```
void quicksort(FILE *pf, int n_elementos);
void permutar_registros(FILE *pf, int izq, int der);
char *campo(FILE *pf, int n);
void main(int argc, char *argv[])
```

La función *quicksort* realiza la ordenación de los *n_elementos* o registros del fichero referenciado por *pf*. Para ello *quicksort* invoca a *qs*.

La función *permutar_registros* es llamada por la función *qs* (*quicksort*) cuando hay que permutar dos registros del fichero para que queden correctamente ordenados.

La función *campo* es llamada por la función *qs* (*quicksort*) cada vez que es necesario leer un registro. Esta función devuelve el campo del registro por el que estamos realizando la ordenación.

La función **main** recibe como parámetro el nombre del fichero a ordenar, llama a la función *quicksort* para ordenar el fichero y después de ordenarlo pregunta al usuario si desea visualizar el fichero.

El programa completo se muestra a continuación.

```
/* Método de ordenación quicksort para ficheros
 * accedidos aleatoriamente.
 *
 * ord_ale.c
 */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

typedef struct datos registro;                    /* tipo registro */
struct datos                          /* definición de un registro */
{
  char referencia[20];
  long precio;
};
registro reg;                                          /* registro */

void quicksort(FILE *pf, int n_elementos);
void permutar_registros(FILE *pf, int izq, int der);
char *campo(FILE *pf, int n);

void main(int argc, char *argv[])
{
  char respuesta;
  int t_reg = sizeof(registro);          /* tamaño de un registro */
  FILE *pf;                                 /* puntero al fichero */
  int n_elementos;

  /* Comprobar el número de argumentos pasados en la línea de
   * órdenes
   */
  if (argc != 2)
  {
    printf("Sintaxis: ord_sec nombre_fichero\n");
    exit(1);
  }

  /* abrir el fichero argv[1] para leer/escribir "r+b" */
  if ((pf = fopen(argv[1], "r+b")) == NULL)
  {
    printf("El fichero %s no puede abrirse\n", argv[1]);
    exit(1);
  }
```

```
    fseek(pf, 0L, SEEK_END);
    n_elementos = (int)ftell(pf)/t_reg;
    rewind(pf);
    quicksort(pf, n_elementos);
    printf("Fichero ordenado\n");
    fclose(pf);       /* cerrar el fichero */

    do
    {
      printf("¿Desea visualizar el fichero? (s/n)  ");
      respuesta = getchar();
      fflush(stdin);
    }
    while (tolower(respuesta) != 's' && tolower(respuesta) != 'n');

    /* Salida de datos */
    if (respuesta == 's')
    {
      /* abrir el fichero argv[1] para leer "r" */
      if ((pf = fopen(argv[1], "r")) == NULL)
      {
        printf("El fichero no puede abrirse.");
        exit(1);
      }

      /* Leer el primer registro del fichero */
      fread(&reg, t_reg, 1, pf);
      while (!ferror(pf) && !feof(pf))
      {
        printf("Referencia:    %s\n", reg.referencia);
        printf("Precio:        %ld\n\n", reg.precio);

        /* Leer el siguiente registro del fichero */
        fread(&reg, t_reg, 1, pf);
      }
    }
    if (ferror(pf))
      perror("Error durante la lectura");

    fclose(pf);
}

/* Función Quicksort para ordenar un fichero */

void qs(FILE *pf, int inf, int sup);
void quicksort(FILE *pf, int n_elementos)
{
  qs(pf, 0, n_elementos - 1);
}

void qs(FILE *pf, int inf, int sup)
{
  register izq, der;
  char mitad[20];

  izq = inf; der = sup;
  /* Obtener el campo mitad por el que se va a ordenar,
```

```
    * del registro mitad
    */
   strcpy(mitad, campo(pf, (int)(izq+der)/2));
   do
   {
     while (strcmp(campo(pf,izq), mitad) < 0 && izq < sup) izq++;
     while (strcmp(mitad, campo(pf,der)) < 0 && der > inf) der--;
     if (izq <= der)
     {
       permutar_registros(pf, izq, der);
       izq++; der--;
     }
   }
   while (izq <= der);

   if (inf < der) qs(pf, inf, der);
   if (izq < sup) qs(pf, izq, sup);
}

/* Permutar los registros de las posiciones izq y der */
void permutar_registros(FILE *pf, int izq, int der)
{
   int t_reg = sizeof(registro);        /* tamaño de un registro */
   registro x, y;

   fseek(pf, (long)izq * t_reg, SEEK_SET);
   fread(&x, t_reg, 1, pf);
   fseek(pf, (long)der * t_reg, SEEK_SET);
   fread(&y, t_reg, 1, pf);

   fseek(pf, (long)izq * t_reg, SEEK_SET);
   fwrite(&y, t_reg, 1, pf);
   fseek(pf, (long)der * t_reg, SEEK_SET);
   fwrite(&x, t_reg, 1, pf);
}

/* Leer el campo utilizado para ordenar */
char *campo(FILE *pf, int n)
{
   int t_reg = sizeof(registro);        /* tamaño de un registro */

   fseek(pf, (long)n * t_reg, SEEK_SET);
   fread(&reg, t_reg, 1, pf);
   return (reg.referencia);
}
```

ALGORITMOS HASH

Los algoritmos *hash* son métodos de búsqueda, que proporcionan una *longitud de búsqueda* pequeña y una flexibilidad superior a la de otros métodos, como puede ser, el método de *búsqueda binaria* que requiere que los elementos del array estén ordenados.

Por *longitud de búsqueda* se entiende el número de accesos que es necesario efectuar sobre un array para encontrar el elemento deseado.

Este método de búsqueda permite, como operaciones básicas, además de la búsqueda de un elemento, insertar un nuevo elemento y eliminar un elemento existente.

Arrays hash

Un array producto de la aplicación de un algoritmo *hash* se denomina *array hash* y son los arrays que se utilizan con mayor frecuencia en los procesos donde se requiere un acceso rápido a los datos. Gráficamente estos arrays tienen la siguiente forma:

CLAVE	CONTENIDO
5040	
3721	
...	
6375	

El array se organiza con elementos formados por dos miembros: *clave* y *contenido*.

La *clave* constituye el medio de acceso al array. Aplicando a la *clave* una función de acceso *fa*, previamente definida, obtenemos un número entero positivo *i*, que nos da la posición del elemento correspondiente, dentro del array.

$$i = fa(clave)$$

Conociendo la posición, tenemos acceso al contenido. El miembro *contenido* puede albergar directamente la información, o bien un puntero a dicha información, cuando ésta sea muy extensa.

El acceso, tal cual lo hemos definido, recibe el nombre de *acceso directo*.

Como ejemplo, suponer que la *clave* de acceso se corresponde con el número del documento nacional de identidad (*dni*) y que el contenido son los datos correspondientes a la persona que tiene ese *dni*. Una función de acceso, *i=fa(dni)*,

que haga corresponder la posición del elemento en el array con el *dni*, es inmediata:

$$i = dni$$

la cual da lugar a un acceso directo. Esta función así definida presenta un inconveniente y es que el número de valores posibles de *i* es demasiado grande para utilizar un array de este tipo.

Para solucionar este problema, siempre es posible, dado un array de longitud *L*, crear una función de acceso, *fa*, de forma que genere un valor comprendido entre *0* y *L*, más comúnmente entre *1* y *L*.

En este caso puede suceder que dos o más claves den un mismo valor de *i*:

$$i = fa(clave_1) = fa(clave_2)$$

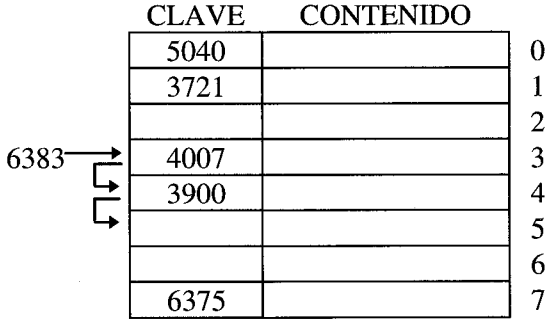
El método *hash* está basado en esta técnica; el acceso al array es directo a través del número *i* y cuando se produce una *colisión* (dos claves diferentes dan un mismo número *i*) este elemento se busca en una zona denominada *área de overflow*.

Método hash abierto

Este es uno de los métodos más utilizados. El algoritmo para acceder a un elemento del array de longitud *L*, es el siguiente:

1. Se calcula $i = fa(clave)$.

2. Si la posición *i* del array está libre, se inserta la *clave* y el *contenido*. Si no está libre y la *clave* es la misma, error: clave duplicada. Si no está libre y la *clave* es diferente, incrementamos *i* en una unidad y repetimos el proceso descrito en este punto 2. Como ejemplo, vea la tabla siguiente:



En la figura, se observa que se quiere insertar la clave 6383. Supongamos que aplicando la función de acceso, obtenemos un valor 3; esto es,

$$i = fa(6383) = 3$$

Como la posición 3 está ocupada y la clave es diferente, tenemos que incrementar *i* y volver de nuevo al punto 2 del algoritmo.

La *longitud media de búsqueda* en un *array hash abierto* viene dada por la expresión:

$$accesos = (2-k)/(2-2k)$$

siendo *k* igual al número de elementos existentes en el array, dividido por *L*.

Por ejemplo, si existen 60 elementos en un array de longitud *L=100*, el número medio de accesos para localizar un elemento será:

$$accesos = (2-60/100)/(2-2*60/100) = 1,75$$

En el método de *búsqueda binaria*, el número de accesos viene dado por el valor $log_2 N$, siendo *N* el número de elementos del array.

Para reducir al máximo el número de colisiones y, como consecuencia, obtener una *longitud media de búsqueda* baja, es importante elegir bien la función de acceso.

Una *función de acceso* o *función hash* bastante utilizada y que proporciona una distribución de las claves uniforme y aleatoria es la *función mitad del cuadrado* que dice: "dada una clave *C*, se eleva al cuadrado (C^2) y se cogen *n* bits del medio, siendo $2^n <= L$". Por ejemplo,

Supongamos: L = 256 lo que implica n = 8
C = 625
$C^2 = 390625$ ($0 <= C^2 <= 2^{32}-1$)
$390625_{10} = 00000000000001011111010111100001_2$
n bits del medio: $01011111_2 = 95_{10}$

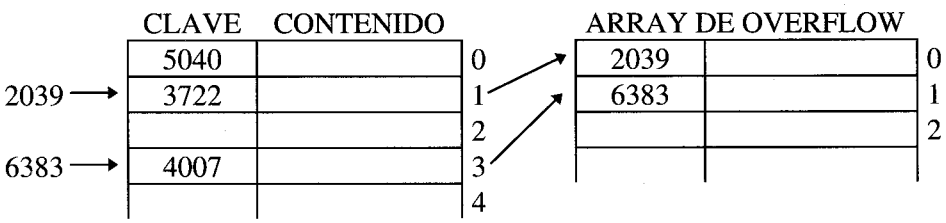
Otra función de acceso muy utilizada es la función *módulo* (resto de una división entera):

$$i = módulo(clave/L)$$

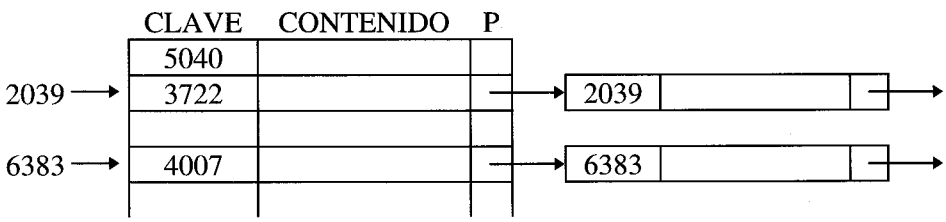
Cuando se utilice esta función es importante elegir un número primo para *L*, con la finalidad de que el número de colisiones sea pequeño. Esta función es llevada a cabo en C por medio del operador %.

Método hash con overflow

Una alternativa al método anterior es la de disponer de otro array separado, para insertar las claves que producen colisión, denominado *array de overflow*, en el cual se almacenan todas estas claves de forma consecutiva.



Otra forma alternativa más normal es organizar una lista encadenada por cada posición del array donde se produzca una colisión.



Cada elemento de esta estructura incorpora un nuevo miembro P, el cual es un puntero a la lista encadenada de overflow.

Eliminación de elementos

En el método *hash* la eliminación de un elemento no es tan simple como dejar vacío dicho elemento, ya que esto da lugar a que los elementos insertados por colisión no puedan ser accedidos. Por ello se suele utilizar un miembro (campo) complementario que sirva para poner una marca de que dicho elemento está eliminado. Esto permite acceder a otros elementos que dependen de él por colisiones, ya que la clave se conserva y también permite insertar un nuevo elemento en esa posición cuando se dé una nueva colisión.

Un ejemplo de un array hash

Crear un array *hash* de una determinada longitud *L* que permita almacenar los datos número de *matrícula* y *nombre* de los alumnos matriculados en una cierta Universidad, utilizando el método *hash abierto* y la función de acceso *módulo*.

El seudocódigo para el método *hash* abierto es el siguiente:

```
<función hash(array, n_elementos, elemento x)>
 [El array está inicializado a valor 0]
 i = matricula módulo n_elementos
 DO WHILE (no insertado y haya elementos libres)
  IF (elemento "i" está libre) THEN
   copiar elemento x en la posición i
  ELSE
   IF (clave duplicada) THEN
    error: clave duplicada
   ELSE
    [se ha producido una colisión]
    [avanzar al siguiente elemento]
    i = i+1
    IF (i = n_elemento) THEN
     i = 0
    ENDIF
   ENDIF
  ENDIF
 ENDDO
END <hash>
```

A continuación se muestra el programa correspondiente a este método.

```
/************************ Arrays Hash ************************/
/* hash.c
 */
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <string.h>

typedef struct datos elemento;          /* nuevo tipo elemento */
struct datos                          /* elemento para el array */
{
  unsigned int matricula;
  char *nombre;
};
```

```
/*************** rutina de manipulación del error **************/
void error(void)
{
  perror("error: insuficiente espacio de memoria");
  exit(1);
}

void hash(elemento *, int, elemento);
int siguiente_primo(int);

void main()
{
  elemento *a;              /* dirección de comienzo del array */
  int n_elementos;               /* nº de elementos del array */
  int i;
  char nom[81];
  elemento x;

  printf("número de elementos:  ");
  scanf("%d", &n_elementos);
  /* Hacer que la longitud del array sea un número primo */
  n_elementos = siguiente_primo(n_elementos);
  printf("Se construye un array de %d elementos\n", n_elementos);

  /* crear el array dinámico "a" */
  a = (elemento *)calloc(n_elementos, sizeof(elemento));
  if (!a) error();

    /* Inicializar el array */
  for (i=0; i < n_elementos; i++)
  {
    a[i].matricula = 0;
    a[i].nombre = NULL;
  }

  /* Introducir datos */
  printf("Introducir datos. Finalizar con matrícula = 0\n\n");
  printf("matrícula:  ");
  scanf("%u", &x.matricula); fflush(stdin);
  while (x.matricula)
  {
    printf("nombre:     "); gets(nom);
    /* asignar espacio para nombre */
    x.nombre = (char *)malloc(strlen(nom)+1);
    if (!x.nombre) error();
    strcpy (x.nombre, nom);
    hash(a, n_elementos, x);      /* llamada a la función hash */
    printf("matrícula:  ");
    scanf("%u", &x.matricula); fflush(stdin);
  }

  /* Liberar la memoria asignada dinámicamente */
  for (i=0; i < n_elementos; i++)
    if (a[i].nombre) free(a[i].nombre);
  free(a);
}
```

```
/******************** Método hash abierto ********************/
void hash(elemento *a, int n_elementos, elemento x)
{
  int i;                  /* índice para referenciar un elemento */
  int conta = 0, insertado = 0;                    /* contador */

  i = x.matricula % n_elementos;           /* función de acceso */
  while (!insertado && conta < n_elementos)
  {
    if (a[i].matricula == 0)                  /* elemento libre */
    {
      a[i] = x;
      insertado = 1;
    }
    else                                     /* clave duplicada */
      if (x.matricula == a[i].matricula)
      {
        printf("error: matrícula duplicada\n");
        insertado = 1;
      }
      else                                              /* colisión */
      {
        /* Siguiente elemento libre */
        i++, conta++;
        if (i == n_elementos) i = 0;
      }
  }
  if (conta == n_elementos)
    printf("error: array lleno\n");
}

/****** Buscar un número primo a partir de un número dado ******/
int siguiente_primo(int n)
{
  int primo = 0, i;

  if (n % 2 == 0)
    n++;
  while (!primo)
  {
    primo = 1;                                          /* primo */
    for (i = 3; i <= (int)sqrt((double)n); i += 2)
      if (n % i == 0)
        primo = 0;                                  /* no primo */
    if (!primo) n += 2;                        /* siguiente impar */
  }
  return n;
}
```

EJERCICIOS RESUELTOS

1. Un centro numérico es un número que separa una lista de números enteros (comenzando en 1) en dos grupos de números, cuyas sumas son iguales. El primer centro numérico es el 6, el cual separa la lista (1 a 8) en los grupos: (1, 2, 3, 4, 5)

y (7, 8) cuyas sumas son ambas iguales a 15. El segundo centro numérico es el 35, el cual separa la lista (1 a 49) en los grupos: (1 a 34) y (36 a 49) cuyas sumas son ambas iguales a 595. Escribir un programa que calcule los centros numéricos entre 1 y *n*.

El ejemplo *(1 a 5) 6 (7 a 8)*, donde se observa que *6* es un centro numérico, sugiere ir probando si los valores *3, 4, 5, 6, ..., cn, ..., n-1* son centros numéricos. En general *cn* es un centro numérico si la suma de todos los valores enteros desde *1* a *(cn-1)* coincide con la suma desde *(cn+1)* a *lim_sup_grupo2* (límite superior del grupo segundo de números). Para que el programa sea eficiente, buscaremos el valor *lim_sup_grupo2* entre los valores *(cn+1)* y *n-1* utilizando el método de búsqueda binaria. Recuerde que la suma de los valores enteros entre *1* y *x* viene dada por la expresión *(x * (x + 1))/2*.

El programa completo se muestra a continuación.

```
/* Calcular los centros numéricos entre 1 y n.
 *
 * cen_num.c
 */
#include <stdio.h>

unsigned long BusquedaB(unsigned long cn,
                        unsigned long suma_grupo1,
                        unsigned long inf, unsigned long sup);

void main()
{
  unsigned long n;                /* centros ncos. entre 1 y n */
  unsigned long cn;          /* posible centro numérico (c. n.) */
  unsigned long anterior;       /*nº anterior al posible c. n. */
  unsigned long siguiente;    /* nº siguiente al posible c. n. */
  unsigned long suma_grupo1;
  unsigned long lim_sup_grupo2;

  printf("Centros numéricos entre 1 y ");
  scanf("%lu", &n);

  for (cn = 3; cn <= n; cn++)
  {
    anterior = cn - 1;
    siguiente = cn + 1;
    suma_grupo1 = (anterior * (anterior + 1)) / 2;
    lim_sup_grupo2 = BusquedaB(cn,suma_grupo1,siguiente,n - 1);
    if (lim_sup_grupo2)
      printf("%lu es centro numérico de 1 a %lu y %lu a %lu\n",
              cn, anterior, siguiente, lim_sup_grupo2);
  }
}
```

```
/************ Función recursiva búsqueda binaria ***************
 *
 * (1 a anterior) cn (siguiente a lim_sup_gr2)
 * suma_grupo1 = suma de los valores desde 1 a anterior
 * suma_grupo2 = suma de los valores desde siguiente a lim_sup_gr2
 *
 * La función devuelve como resultado el valor lim_sup_gr2.
 * Si cn no es un centro numérico devuelve un valor 0.
 */
unsigned long BusquedaB(unsigned long cn,
                        unsigned long suma_grupo1,
                        unsigned long inf, unsigned long sup)
{
  unsigned long lim_sup_gr2;
  unsigned long suma_grupo2;

  suma_grupo2 = (inf * (inf + 1)) / 2 - suma_grupo1 - cn;

  /* Comprobar si la búsqueda falla */
  if ((inf >= sup) && (suma_grupo2 != suma_grupo1))
    return 0;

  /* Sigue la búsqueda */
  lim_sup_gr2 = (inf + sup) / 2;
  suma_grupo2 = (lim_sup_gr2*(lim_sup_gr2 + 1))/2-suma_grupo1-cn;
  if (suma_grupo2 == suma_grupo1)
  {
    /* Búsqueda con éxito */
    /* Se devuelve el límite superior del segundo grupo */
    return lim_sup_gr2;
  }
  else
    if (suma_grupo1 > suma_grupo2)
      BusquedaB(cn, suma_grupo1, lim_sup_gr2 + 1, sup);
    else
      BusquedaB(cn, suma_grupo1, inf, lim_sup_gr2 - 1);
}
```

EJERCICIOS PROPUESTOS

1. Realizar un programa que clasifique un fichero almacenado en el disco, utilizando el método de inserción. El proceso de clasificación se realizará directamente sobre el fichero (no utilizar arrays ni ficheros auxiliares).
2. Realizar una función que a partir de dos ficheros ordenados *a* y *b*, obtenga como resultado un fichero *c* también ordenado, que sea fusión de los dos ficheros anteriores. A continuación realizar un programa que utilizando esta función visualice los registros del fichero ordenado. Los ficheros *a*, *b* y *c* serán pasados como argumentos en la línea de órdenes.
3. Escribir una función *ordenar* que permita ordenar ascendentemente un array de dos dimensiones en el que cada elemento sea del tipo siguiente:

```
typedef struct datos elemento;
struct datos
{
  unsigned int matricula;
  char *nombre;
};
```

El prototipo de la función será así:

```
void ordenar(elemento a[][MAX], int filas, int cols);
```

Para realizar el proceso de ordenación emplee el método que quiera, pero realícela directamente sobre el array *a* y por el miembro de la estructura que se desee, seleccionado a partir de un menú que presentará la propia función.

Finalmente, escriba un programa que pueda recibir un fichero como argumento en la línea de órdenes, cuyos registros sean del tipo *elemento*, almacene los registros en un array de dos dimensiones y utilizando la función *ordenar* ordene el array y lo visualice una vez ordenado.

4. Realizar un programa que cree una lista dinámica a partir de una serie de números cualesquiera introducidos por el teclado. A continuación, ordenar la lista ascendentemente utilizando el método *quicksort*.

5. Realizar un programa que:

   a) Cree una lista de elementos del tipo:

   ```
   struct registro
   {
     char nombre[40];
     unsigned int matricula;
     unsigned int nota;
   };
   ```

   b) Clasifique la lista utilizando el método de inserción.

   c) Busque una nota por el número de matrícula utilizando el método de búsqueda binaria.

6. Realizar un programa que permita crear y operar con una tabla utilizando el método *hash abierto*. El programa tendrá al menos las funciones siguientes:

   a) Una función para leer registros de la forma:

   ```
   número del cliente
   saldo del cliente
   ```

b) Una función para insertar registros en la tabla utilizando como índice de acceso el valor *(clave módulo L)* donde la *clave* se corresponde con el *número del cliente* y *L* es la longitud de la tabla.

c) Una función que permita buscar un determinado registro. La función devolverá una referencia al registro o un valor nulo si no se encuentra.

APÉNDICE A

DEPURAR UN PROGRAMA

En este apéndice vamos a ver cómo utilizar dos depuradores bastante extendidos. Estos son, el depurador *Code View* de Microsoft y el depurador *gdb* de UNIX.

EL DEPURADOR Code View DE Microsoft

El depurador *Code View* de Microsoft, es un programa interactivo que nos ayudará a localizar errores rápidamente. *Code View* puede utilizarse solamente con ficheros *.exe*; no se puede utilizar con ficheros *.com*.

En orden a depurar un programa, es bueno tener presente la siguiente consideración: el compilador C permite escribir más de una sentencia en una misma línea. Por ejemplo:

```
car = texto[i]; if (car == '\n') ++lineas;
```

Esta forma de escribir un programa dificulta la depuración, ya que no es posible acceder a las sentencias individuales que componen la línea. Por ello, se recomienda escribir las sentencias anteriores de la siguiente forma:

```
car = texto[i];
if (car == '\n')
  ++lineas;
```

Compilar y enlazar un programa C para depurarlo

Cuando se tiene la intención de depurar un programa, en el momento de compilarlo, se debe especificar la opción */Zi*. Esta opción indica al compilador que incluya números de línea e información extra para el depurador en el fichero objeto.

Si en un programa compuesto por varios módulos, solamente deseamos depurar algunos de ellos, éstos deben ser compilados con la opción */Zi*, y el resto con la opción */Zd*. La opción */Zd* añade al fichero objeto números de línea, pero no información extra, con lo que el espacio ocupado en el disco y en la memoria es menor. También debe especificarse la opción */Od*, la cual impide la optimización, ya que ésta puede dificultar la depuración. Por ejemplo:

```
cl /Zi /Od prog01.c
```

La orden anterior compila y enlaza el fichero fuente *prog01.c*. El resultado es un fichero ejecutable con información para el depurador. El fichero objeto es enlazado automáticamente con la opción */CO* (abreviatura de */CODEVIEW*).

El siguiente ejemplo compila *modulo01.c*, produciendo un fichero *modulo01.obj* con la información total necesaria para el depurador; compila *modulo02.c* produciendo un fichero objeto con información limitada. A continuación se utiliza **cl** otra vez para enlazar los ficheros objeto resultantes. Esta vez, **cl** no compila de nuevo, ya que los argumentos no tienen extensión. La opción */Zi* en esta última orden, hace que el enlazador (*linker*) sea invocado con la opción */CO*. El resultado es un fichero ejecutable, *modulo01.exe*, que permite depurar *modulo01.c*.

```
cl /Zi /Od /c modulo01.c
cl /Zd /Od /c modulo02.c
cl /Zi modulo01 modulo02
```

INVOCAR A Code View

Una vez compilado un programa con las opciones necesarias para depurarlo, invocaremos a *Code View* para proceder a su depuración. La sintaxis es la siguiente:

cv *fichero-ejecutable*

El siguiente ejemplo invoca al depurador *Code View* provisto con Microsoft C, que carga el fichero *prog01.exe* en memoria para depurarlo.

```
cv prog01
```

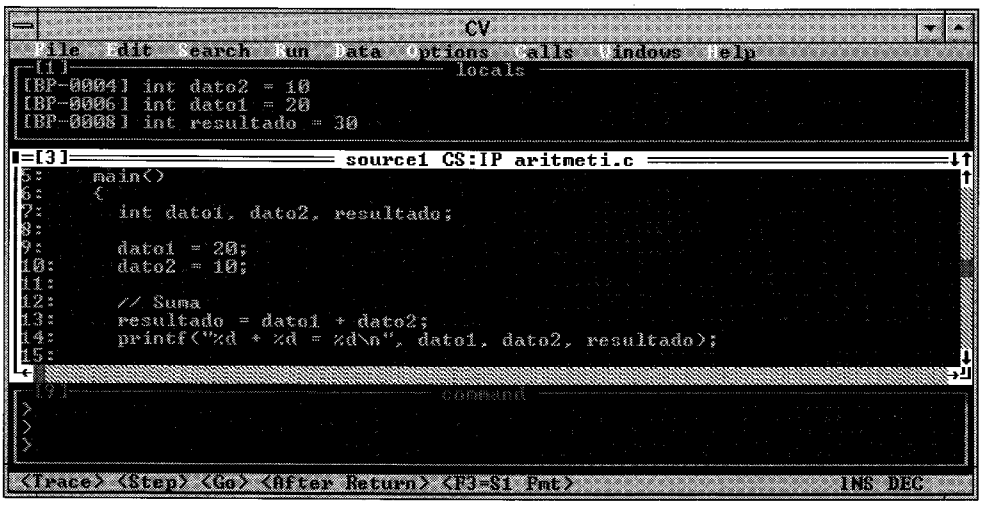
Code View puede utilizar la memoria expandida o la extendida si el gestor de memoria correspondiente está instalado.

Cuando invocamos al depurador *Code View* se visualiza una pantalla dividida en tres áreas. El área primera ocupa la línea 1, la cual visualiza la barra de menús.

El área segunda ocupa las líneas 2 a 24 (dependiendo del tipo de tarjeta de vídeo, podemos disponer de más líneas) y es utilizada para visualizar las ventanas

del depurador. Para movernos dentro de estas ventanas podemos utilizar las teclas de movimiento del punto de inserción y las teclas *RePág* (*PgUp*) y *AvPág* (*PgDn*). Para pasar de una ventana a otra se pulsa la tecla *F6* o *Shift+F6*.

El área tercera ocupa la línea 25 y es utilizada para indicar las acciones que podemos tomar.



Pantalla inicial del depurador de Microsoft C

Las operaciones mínimas que debe incluir un depurador son las siguientes:

Rastrear

Permite ver la sentencia del programa que se está ejecutando. *Code View* incluye las siguientes opciones:

1. Ejecutar una sentencia cada vez, incluidas funciones definidas por el usuario. Esta modalidad se activa y se continúa, pulsando la tecla *F8*.

2. Cuando la sentencia a ejecutar coincide con una llamada a una función definida por el usuario y no queremos que ésta se ejecute paso a paso, utilizaremos la tecla *F10* en vez de la tecla *F8*.

Punto de parada

Un punto de parada permite hacer una pausa en un lugar determinado dentro del programa para comprobar los valores de las variables en ese instante. Colocar los puntos de parada donde se sospeche que está el error.

Para poner o quitar un punto de parada, se coloca el punto de inserción en la línea donde se desea que tenga lugar la pausa y se pulsa *F9*. Se pueden poner tantos puntos de parada como necesitemos.

Ventanas de seguimiento

Las ventanas *locals* y *watch* que podemos activar o desactivar desde el menú *Windows* de *Code View*, permiten observar los valores de las variables y de las expresiones especificadas. Al ejecutar el programa paso a paso podemos observar en estas ventanas, los valores que van tomando las variables especificadas.

Ejecución controlada

Si pulsamos la tecla *F5*, la ejecución continúa hasta el final del programa o hasta el primer punto de parada, si existe.

Cuando se pulsa *F7* el programa se ejecuta desde la última sentencia ejecutada, hasta la línea donde se encuentra el punto de inserción.

Ejecución automática

Si desea procesar el programa paso a paso de una forma automática, ejecute la orden *Animate* del menú *Run*. Una vez iniciada la animación de un programa, ésta puede ser detenida pulsando cualquier tecla. La velocidad de ejecución automática se puede aumentar o disminuir mediante la orden *Preferences* del menú *Options*.

Ejecute la orden *Restart* cada vez que inicie de nuevo el proceso de depuración.

Para salir del depurador, ejecute la orden *Exit* del menú *File*.

Operaciones comunes a las ventanas locals y watch

Es posible modificar el valor de una variable o de una expresión incluida en cualquiera de estas ventanas. Esto se hace escribiendo encima del valor actual.

Cualquier elemento de datos (estructura, array, o puntero) puede ser expandido o contraído si la ventana, *locals* o *watch*, está activa. Un elemento de datos que está precedido por el signo mas (+) significa que puede ser expandido y si está precedido por el signo menos (–) significa que puede ser contraído.

Para expandir o contraer un elemento de datos, apunte al mismo con el ratón y haga doble clic con el botón izquierdo; o bien, mueva el punto de inserción al elemento de datos y pulse *Entrar*.

MENÚS DE CODE VIEW

Cuando se entra en *Code View*, mediante la orden **cv**, lo primero que aparece es la barra de menús y tres ventanas: la ventana *locals* para mostrar las variables locales, la ventana *source* para mostrar el código fuente del programa a depurar y la ventana *command* que permite emitir órdenes para el depurador.

La barra de menús consta de los siguientes menús: *File*, *Edit*, *Search*, *Run*, *Data*, *Options*, *Calls*, *Windows* y *Help*. Para seleccionar un menú, se puede optar por cualquiera de las dos formas siguientes:

1. Pulsar la tecla *Alt*, para activar la barra de menús y después la tecla correspondiente a la letra que se presenta en alto brillo o color diferente del menú deseado (es indiferente el utilizar mayúsculas o minúsculas). Por ejemplo *R* para activar el menú *Run*.

2. Pulsar la tecla *Alt*, para activar la barra de menús y utilizando las teclas de movimiento del punto de inserción, elegir el menú deseado y pulsar la tecla *Entrar*.

Para seleccionar una orden de un menú, se puede proceder de cualquiera de las dos formas siguientes:

1. Pulsar la tecla correspondiente a letra que se presenta en alto brillo o color diferente de la orden deseada (es indiferente el utilizar mayúsculas o minúsculas). Por ejemplo, si lo que se quiere es salir de *Code View*, se selecciona el menú *File* y se pulsa la tecla *x* correspondiente a la orden *Exit*.

2. Utilizando las teclas de movimiento del punto de inserción, seleccionar la orden deseada y pulsar *Entrar*.

Algunas órdenes de estos menús tienen escrito a su derecha la tecla o teclas aceleradoras que realizan la misma operación. Por ejemplo la orden *Exit* del menú *File* tiene asignadas como teclas aceleradoras *Alt+F4*. Esto significa que al pulsar las teclas *Alt+F4* se ejecuta la orden *Exit*.

Algunas órdenes abren una ventana de diálogo; por ejemplo, la orden *Find* del menú *Search*. En este caso se responde a las cuestiones planteadas y a continuación se pulsa *Entrar* (< OK >).

Para abandonar un menú, simplemente hay que pulsar la tecla *Esc*.

Para obtener ayuda sobre cualquier orden, primero se selecciona y después se pulsa la tecla *F1*.

CODE VIEW CON RATÓN

El depurador *Code View* está diseñado para utilizar un ratón de Microsoft o compatible con éste.

Para ejecutar una orden de un menú utilizando el ratón:

1. Se apunta al menú y se pulsa el botón izquierdo del ratón.

2. Se apunta a la orden deseada del menú y se pulsa el botón izquierdo del ratón.

Para desplazar el texto sobre la ventana activa, se dispone de una barra de desplazamiento vertical y otra horizontal. Ambas tienen un cuadrado de desplazamiento y en sus extremos unos botones de desplazamiento.

- Para avanzar o retroceder una línea, se apunta al botón de desplazamiento superior o inferior de la barra de desplazamiento vertical y se pulsa el botón izquierdo del ratón. Para desplazar el texto una posición a la izquierda o a la derecha se apunta al botón derecho o izquierdo de la barra de desplazamiento horizontal y se pulsa el botón izquierdo del ratón.

- Para avanzar o retroceder una página, se coloca el cursor del ratón sobre la barra de desplazamiento vertical, entre el cuadrado de desplazamiento y el botón inferior o entre el cuadrado de desplazamiento y el botón superior, y se pulsa el botón izquierdo del ratón. La operación es análoga para desplazar el texto una página hacia la izquierda o hacia la derecha; eso sí, actuando sobre la barra de desplazamiento horizontal.

- También se puede desplazar el texto, apuntando con el cursor del ratón al cuadrado de desplazamiento y arrastrándolo, moviendo el ratón en la dirección deseada con el botón izquierdo pulsado.

Si se quiere modificar el tamaño de una ventana, se apunta a la línea de separación entre ventanas y con el botón izquierdo del ratón pulsado, se arrastra en la dirección apropiada para agrandar o reducir la ventana.

Para activar una ventana, apuntar a cualquier lugar dentro de la misma y pulsar el botón izquierdo del ratón.

Para activar o desactivar cualquier opción dentro de una ventana de diálogo, apuntar al espacio entre corchetes o entre ángulos y pulsar el botón izquierdo del ratón.

EL DEPURADOR gdb DE UNIX

Cuando se tiene la intención de depurar un programa C escrito bajo UNIX, en el momento de compilarlo, se debe especificar la opción *–g*. Esta opción indica al compilador que incluya información extra para el depurador en el fichero objeto. Por ejemplo:

```
cc -g prog01.c -o prog01
```

La orden anterior compila y enlaza el fichero fuente *prog01.c*. El resultado es un fichero ejecutable con información para el depurador.

Una vez compilado un programa con las opciones necesarias para depurarlo, invocaremos a *gdb* para proceder a su depuración. La sintaxis es la siguiente:

gdb *fichero-ejecutable*

El siguiente ejemplo invoca al depurador *gdb* de UNIX, que carga el fichero *prog01* en memoria para depurarlo.

```
gdb prog01
```

Una vez que se ha invocado al depurador, desde la línea de órdenes se pueden ejecutar órdenes como las siguientes:

break [*fichero*:]*función*

Establece un punto de parada en la función indicada del fichero especificado. Por ejemplo, la siguiente orden pone un punto de parada en la función *escribir*.

```
b escribir
```

break [*fichero*:]*línea*

Establece un punto de parada en la línea indicada. Por ejemplo, la siguiente orden pone un punto de parada en la línea 10.

```
b 10
```

delete punto-de-parada

Elimina el punto de parada especificado. Por ejemplo, la siguiente orden elimina el punto de parada 1.

```
d 1
```

run [*argumentos*]

Inicia la ejecución del programa que estamos depurando. La ejecución se detiene al encontrar un punto de parada o al finalizar el programa. Por ejemplo:

```
run
```

print expresión

Visualizar el valor de una variable o de una expresión. Por ejemplo, la siguiente orden visualiza el valor de la variable *total*.

```
p total
```

next

Ejecutar la siguiente línea. Si la línea coincide con una llamada a una función definida por el usuario, no se entra a depurar la función. Por ejemplo:

```
n
```

continue

Continuar con la ejecución del programa. Por ejemplo:

```
c
```

step

Ejecutar la siguiente línea. Si la línea coincide con una llamada a una función definida por el usuario, se entra a depurar la función. Por ejemplo:

```
s
```

list

Visualizar código. Por ejemplo:

```
l
```

bt

Visualizar el estado de la pila (las llamadas en curso a funciones).

help [*orden*]

Solicitar ayuda sobre la orden especificada.

quit

Finalizar el trabajo de depuración.

APÉNDICE B

VISUAL C++

Si tiene el paquete Microsoft Visual C++ para Windows podrá, igual que con cualquier otro compilador de C/C++, crear sus aplicaciones para MS-DOS. Este apéndice le muestra la forma más rápida de utilizar el entorno de programación de Visual C++ para escribir, construir y ejecutar una aplicación MS-DOS.

APLICACIÓN QuickWin

Cuando se construye una aplicación MS-DOS con el entorno de desarrollo de Visual C++, para ejecutarla hay que salir a MS-DOS y una vez ejecutada, para continuar con la aplicación, hay que volver al entorno de desarrollo de Visual C++. Esto, además de ser un inconveniente, ralentiza el desarrollo. Visual C++ soluciona este problema permitiendo crear una aplicación *QuickWin* en lugar de una aplicación MS-DOS.

Una aplicación *QuickWin* es un programa estándar de entrada/salida que sólo puede ejecutarse en el sistema operativo Windows. Es la forma más rápida y sencilla de combinar una aplicación MS-DOS con Windows.

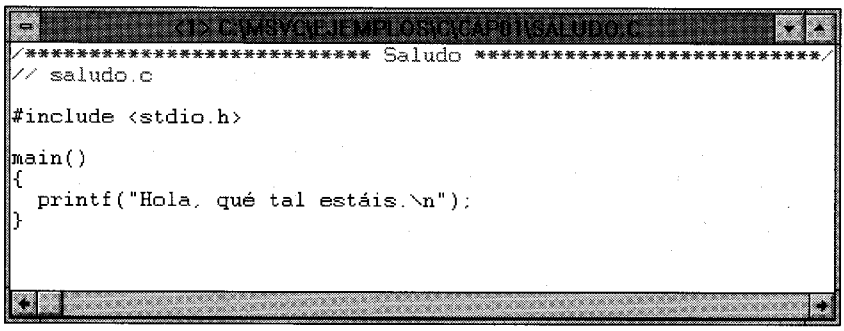
De esta forma, un usuario de Visual C++ puede escribir su programa MS-DOS, elegir como tipo de proyecto *QuickWin* y ejecutar el programa desde Windows. Cuando ejecute el programa aparecerá una aplicación Windows MDI con una ventana dedicada a todas las funciones de E/S del programa MS-DOS. Esta ventana es igual que el monitor de vídeo; el usuario puede escribir en la ventana y visualizar resultados en la misma.

Naturalmente, *QuickWin* no puede interpretar programas que utilicen funciones específicas para el hardware tales como las interrupciones de la BIOS o funciones que manipulen directamente la memoria de vídeo.

Cuando el usuario tenga a punto su programa, puede realizar una última compilación eligiendo como tipo de proyecto, *Aplicación MS-DOS (.EXE)*. De esta forma obtendrá un programa ejecutable directamente en MS-DOS.

UN EJEMPLO SIMPLE

Para ver lo fácil que es utilizar la librería *QuickWin*, pruebe a escribir el programa que se muestra a continuación:



Para ello, ejecute Visual C++ desde el administrador de programas de Windows, cierre cualquier proyecto que haya abierto ejecutando la orden *Close* del menú *Project* y cierre todas las ventanas ejecutando la orden *Close All* del menú *Windows*.

A continuación, para escribir un programa *QuickWin*, proceda así:

1. Ejecute la orden *New* del menú *File*. Se crea una ventana vacía.

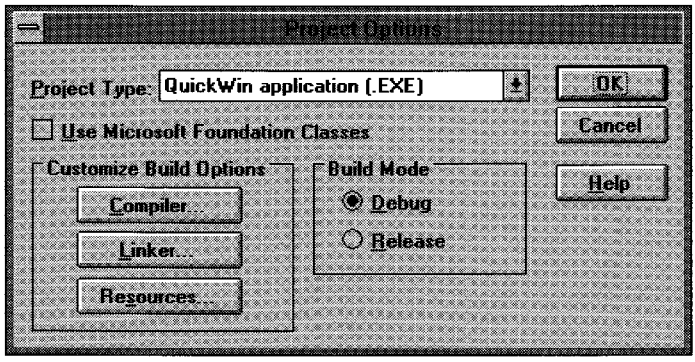
2. Escriba el código correspondiente al programa que desea realizar. Para el ejemplo que estamos haciendo, escriba el código mostrado en la figura anterior.

3. Ejecute la orden *Save As* del menú *File* para guardar el programa. Siguiendo con nuestro ejemplo, guarde el programa con el nombre *saludo.cpp* o *saludo.c*. la extensión *.cpp* la utilizará cuando se trate de un programa C++ y la extensión *.c* la utilizará cuando se trate de un programa C.

Una vez ejecutados los pasos anteriores, ya tiene escrito el programa. Para construir y ejecutar el programa *QuickWin*, proceda así:

1. Asegúrese de que la ventana activa es la que contiene su programa, haciendo clic sobre ella.

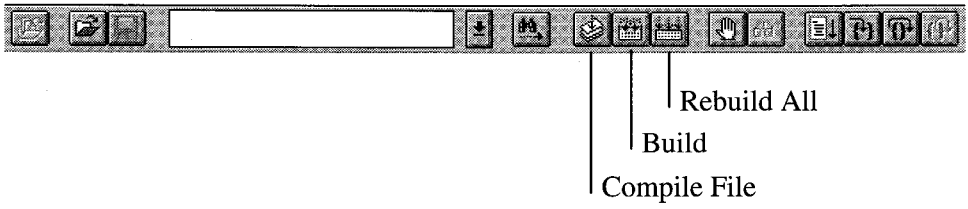
2. Ejecute la orden *Project* del menú *Options*. Se visualiza el diálogo siguiente:



3. Como puede ver en la figura, elija como tipo de proyecto *Aplicación Quick-Win (.EXE)*. Después seleccione *Debug* si tiene intención de utilizar el depurador o *Release* en caso contrario. El botón *Compiler* le muestra las opciones del compilador y el botón *Linker* las del enlazador. Por ejemplo, si no quiere que se generen ficheros para el analizador, haga clic en el botón *Compiler* y en la lista *Category* elija *Listing Files* y asegúrese de que la opción *Browser Information* no está señalada; después haga clic en el botón *OK*.

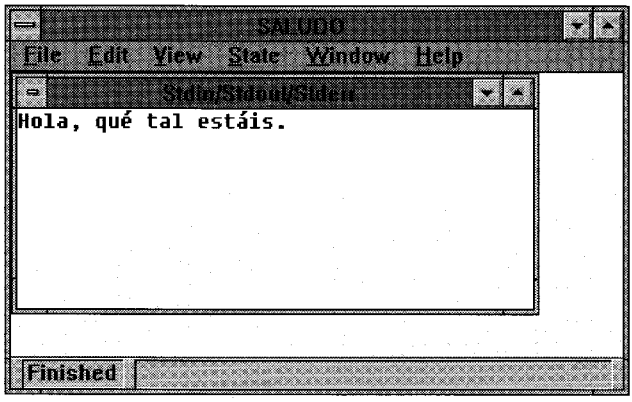
4. Haga clic en el botón *OK* de la ventana *Opciones del proyecto*.

5. Ejecute la orden *Build* del menú *Project* o haga clic en el botón correspondiente de la barra de herramientas.



Cuando finalice el proceso de construcción, aparecerá en la barra de estado *0 errores y 0 avisos (warnings)*. Si hay errores, utilice la tecla *F4* o las teclas *Shift+F4* para situarse sobre la línea que contiene el error y corríjalo. Repita este proceso hasta corregir todos los errores. Después construya de nuevo el programa ejecutable.

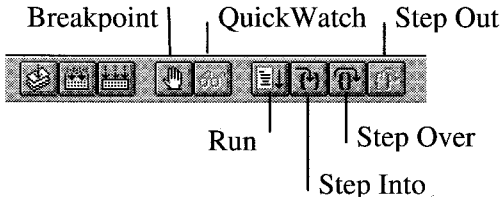
6. Ejecute la orden *Execute* del menú *Project* o pulse la teclas *Ctrl+F5* para ejecutar el programa. Se visualizan las ventanas siguientes:



Una vez que haya visto los resultados, para cerrar la aplicación *QuickWin* ejecute la orden *Exit* del menú *File* o pulse las teclas *Ctrl+C*. Esto hará que retorne al entorno de desarrollo de Visual C++.

DEPURAR LA APLICACIÓN

Una vez construida la aplicación, si entre las opciones del proyecto eligió *Debug* podrá ahora, si lo necesita, depurar la aplicación (vea el apéndice A). Para ello utilice las órdenes del menú *Debug* o los botones correspondientes de la barra de herramientas.



Las órdenes de que dispone para depurar una aplicación son las siguientes:

Go (Run) o *F5*
Comienza o continúa la ejecución del programa.

Restart o *Shift+F5*
Reinicia la ejecución del programa.

Stop Debugging o *Alt+F5*
Detiene el proceso de depuración.

Step Into o *F8*

Ejecuta el programa paso a paso. Si la línea a ejecutar coincide con una llamada a una función definida por el usuario, dicha función también se ejecuta paso a paso.

Step Over o *F10*

Ejecuta el programa paso a paso. Si la línea a ejecutar coincide con una llamada a una función definida por el usuario, dicha función no se ejecuta paso a paso, sino de una sola vez.

Step Out o *Shift+F7*

Cuando una función definida por el usuario ha sido invocada para ejecutarse paso a paso, puede ejecutarse totalmente utilizando esta orden.

Step to Cursor o *F7*

Ejecuta el código que hay entre la última línea ejecutada y la línea donde se encuentra el punto de inserción.

Show Call Stack o *Ctrl+K*

Muestra el estado de la pila; esto es, las llamadas a funciones cuya ejecución aún no ha finalizado.

Breakpoints o *F9*

Pone o quita un punto de parada en la línea sobre la que está el punto de inserción.

QuickWatch o *Shift+F9*

Visualiza el valor de la variable que está bajo el punto de inserción o el valor de la expresión seleccionada (sombreada).

APÉNDICE C

CÓDIGOS DE CARACTERES

UTILIZACIÓN DE CARACTERES ANSI CON WINDOWS

Una tabla de códigos es un juego de caracteres donde cada uno tiene asignado un número utilizado para su representación interna.

Windows utiliza el juego de caracteres ANSI. Para escribir un carácter ANSI que no esté en el teclado:

1. Localice en la tabla que se muestra en la página siguiente el carácter ANSI que necesite y observe su código numérico.
2. Pulse la tecla **Bloq Num** (Num Lock) para activar el teclado numérico.
3. Mantenga pulsada la tecla **Alt** y utilice el teclado numérico para pulsar el 0, y a continuación las teclas correspondientes al código del carácter.

Por ejemplo, para escribir el carácter ± en el entorno Windows, mantenga pulsada la tecla **Alt** mientras escribe 0177 en el teclado numérico.

JUEGO DE CARACTERES ANSI

DEC	CAR
33	!
34	"
35	#
36	$
37	%
38	&
39	'
40	(
41	)
42	*
43	+
44	,
45	-
46	.
47	/
48	0
49	1
50	2
51	3
52	4
53	5
54	6
55	7
56	8
57	9
58	:
59	;
60	<
61	=
62	>
63	?
64	@
65	A
66	B
67	C
68	D
69	E
70	F
71	G
72	H
73	I
74	J
75	K
76	L
77	M
78	N
79	O
80	P
81	Q
82	R
83	S
84	T
85	U
86	V
87	W
88	X

DEC	CAR
89	Y
90	Z
91	[
92	\
93	]
94	^
96	`
97	a
98	b
99	c
100	d
101	e
102	f
103	g
104	h
105	i
106	j
107	k
108	l
109	m
110	n
111	o
112	p
113	q
114	r
115	s
116	t
117	u
118	v
119	w
120	x
121	y
122	z
123	{
124	|
125	}
126	~
127	▒
128	▒
129	▒
130	▒
131	▒
132	▒
133	▒
134	▒
135	▒
136	▒
137	▒
138	▒
139	▒
140	▒
141	▒
142	▒
143	▒
144	▒
145	'

DEC	CAR
146	'
147	"
148	"
149	o
150	–
151	—
152	▒
153	▒
154	▒
155	▒
156	▒
157	▒
157	▒
159	▒
160	
161	¡
162	¢
163	£
164	¤
165	¥
166	|
167	§
168	¨
169	©
170	ª
171	«
172	¬
173	-
174	®
175	¯
176	°
177	±
178	²
179	³
180	´
181	µ
182	¶
183	·
184	¸
185	¹
186	º
187	»
188	¼
189	½
190	¾
191	¿
192	À
193	Á
194	Â
195	Ã
196	Ä
197	Å
198	Æ
199	Ç
200	È
201	É

DEC	CAR
202	Ê
203	Ë
204	Ì
205	Í
206	Î
207	Ï
208	Ð
209	Ñ
210	Ò
211	Ó
212	Ô
213	Õ
214	Ö
215	×
216	Ø
217	Ù
218	Ú
219	Û
220	Ü
221	Ý
222	Þ
223	ß
224	à
225	á
226	â
227	ã
228	ä
229	å
230	æ
231	ç
232	è
233	é
234	ê
235	ë
236	ì
237	í
238	î
239	ï
240	ð
241	ñ
242	ò
243	ó
244	ô
245	õ
246	ö
247	÷
248	ø
249	ù
250	ú
251	û
252	ü
253	ý
254	þ
255	ÿ

UTILIZACIÓN DE CARACTERES ASCII

En UNIX, MS-DOS y fuera del entorno Windows se utiliza el juego de caracteres ASCII. Para escribir en MS-DOS un carácter ASCII que no esté en el teclado:

1. Busque el carácter en la tabla de códigos que coincida con la tabla activa. Utilice la orden **chcp** para saber qué tabla de códigos está activa.
2. Mantenga pulsada la tecla **Alt** y utilice el teclado numérico para pulsar las teclas correspondientes al número del carácter que desee.

Por ejemplo, si está utilizando la tabla de códigos 850, para escribir el carácter π mantenga pulsada la tecla **Alt** mientras escribe 227 en el teclado numérico.

JUEGO DE CARACTERES ASCII

VALOR DECIMAL	VALOR HEXA-DECIMAL	CONTROL-CARACT	CARACT	VALOR DECIMAL	VALOR HEXA-DECIMAL	CARACT.	VALOR DECIMAL	VALOR HEXA-DECIMAL	CARACT	VALOR DECIMAL	VALOR HEXA-DECIMAL	CARACT.	VALOR DECIMAL	VALOR HEXA-DECIMAL	CARACT	VALOR DECIMAL	VALOR HEXA-DECIMAL	CARACT.
000	00	NUL		043	2B	+	086	56	V	129	81	ü	172	AC	¼	215	D7	╫
001	01	SOH	☺	044	2C	,	087	57	W	130	82	é	173	AD	¡	216	D8	╪
002	02	STX	☻	045	2D	-	088	58	X	131	83	â	174	AE	«	217	D9	┘
003	03	ETX	♥	046	2E	.	089	59	Y	132	84	ä	175	AF	»	218	DA	┌
004	04	EOT	♦	047	2F	/	090	5A	Z	133	85	à	176	B0	▒	219	DB	■
005	05	ENQ	♣	048	30	0	091	5B	[	134	86	å	177	B1	▒	220	DC	▄
006	06	ACK	♠	049	31	1	092	5C	\	135	87	ç	178	B2	▓	221	DD	▌
007	07	BEL	•	050	32	2	093	5D	]	136	88	ê	179	B3	│	222	DE	▐
008	08	BS	◘	051	33	3	094	5E	^	137	89	ë	180	B4	┤	223	DF	▀
009	09	HT	○	052	34	4	095	5F	_	138	8A	è	181	B5	╡	224	E0	α
010	0A	LF	◙	053	35	5	096	60	`	139	8B	ï	182	B6	╢	225	E1	ß
011	0B	VT	♂	054	36	6	097	61	a	140	8C	î	183	B7	╖	226	E2	Γ
012	0C	FF	♀	055	37	7	098	62	b	141	8D	ì	184	B8	╕	227	E3	π
013	0D	CR	♪	056	38	8	099	63	c	142	8E	Ä	185	B9	╣	228	E4	Σ
014	0E	SO	♫	057	39	9	100	64	d	143	8F	Å	186	BA	║	229	E5	σ
015	0F	SI	☼	058	3A	:	101	65	e	144	90	É	187	BB	╗	230	E6	µ
016	10	DLE	►	059	3B	;	102	66	f	145	91	æ	188	BC	╝	231	E7	τ
017	11	DC1	◄	060	3C	<	103	67	g	146	92	Æ	189	BD	╜	232	E8	Φ
018	12	DC2	↕	061	3D	=	104	68	h	147	93	ô	190	BE	╛	233	E9	Θ
019	13	DC3	‼	062	3E	>	105	69	i	148	94	ö	191	BF	┐	234	EA	Ω
020	14	DC4	¶	063	3F	?	106	6A	j	149	95	ò	192	C0	└	235	EB	δ
021	15	NAK	§	064	40	@	107	6B	k	150	96	û	193	C1	┴	236	EC	∞
022	16	SYN	▬	065	41	A	108	6C	l	151	97	ù	194	C2	┬	237	ED	∅
023	17	ETB	↨	066	42	B	109	6D	m	152	98	ÿ	195	C3	├	238	EE	∈
024	18	CAN	↑	067	43	C	110	6E	n	153	99	Ö	196	C4	─	239	EF	∩
025	19	EM	↓	068	44	D	111	6F	o	154	9A	Ü	197	C5	┼	240	F0	≡
026	1A	SUB	→	069	45	E	112	70	p	155	9B	¢	198	C6	╞	241	F1	±
027	1B	ESC	←	070	46	F	113	71	q	156	9C	£	199	C7	╟	242	F2	≥
028	1C	FS	∟	071	47	G	114	72	r	157	9D	¥	200	C8	╚	243	F3	≤
029	1D	GS	↔	072	48	H	115	73	s	158	9E	Pt	201	C9	╔	244	F4	⌠
030	1E	RS	▲	073	49	I	116	74	t	159	9F	ƒ	202	CA	╩	245	F5	⌡
031	1F	US	▼	074	4A	J	117	75	u	160	A0	á	203	CB	╦	246	F6	÷
032	20	SP	Space	075	4B	K	118	76	v	161	A1	í	204	CC	╠	247	F7	≈
033	21		!	076	4C	L	119	77	w	162	A2	ó	205	CD	═	248	F8	°
034	22		"	077	4D	M	120	78	x	163	A3	ú	206	CE	╬	249	F9	∙
035	23		#	078	4E	N	121	79	y	164	A4	ñ	207	CF	╧	250	FA	·
036	24		$	079	4F	O	122	7A	z	165	A5	Ñ	208	D0	╨	251	FB	√
037	25		%	080	50	P	123	7B	{	166	A6	ª	209	D1	╤	252	FC	n
038	26		&	081	51	Q	124	7C	|	167	A7	º	210	D2	╥	253	FD	²
039	27		'	082	52	R	125	7D	}	168	A8	¿	211	D3	╙	254	FE	■
040	28		(	083	53	S	126	7E	~	169	A9	⌐	212	D4	╘	255	FF	
041	29		)	084	54	T	127	7F	⌂	170	AA	¬	213	D5	╒			
042	2A		*	085	55	U	128	80	Ç	171	AB	½	214	D6	╓			

CÓDIGOS EXTENDIDOS

Segundo código	Significado
3	NUL (null character)
15	Shift Tab (—< + +)
16-25	Alt-Q/W/E/R/T/Y/U/I/O/P
30-38	Alt-A/S/D/F/G/H/I/J/K/L
44-50	Alt-Z/X/C/V/B/N/M
59-68	Keys F1-F10 (disabled as softkeys)
71	Home
72	Up arrow
73	PgUp
75	Left arrow
77	Right arrow
79	End
80	Down arrow
81	PgDn
82	Ins
83	Del
84-93	F11-F20 (Shift-F1 to Shift-F10)
94-103	F21-F30 (Ctrl-F1 through F10)
104-113	F31-F40 (Alt-F1 through F10)
114	Ctrl-PrtSc
115	Ctrl-Left arrow
116	Ctrl-Right arrow
117	Ctrl-End
118	Ctrl-PgDn
119	Ctrl-Home
120-131	Alt-1/2/3/4/5/6/7/8/9/0/—/=
132	Ctrl-PgUp
133	F11
134	F12
135	Shift-F11
136	Shift-F12
137	Ctrl-F11
138	Ctrl-F12
139	Alt-F11
140	Alt-F12

CÓDIGOS DEL TECLADO

Tecla	Código en Hex	Tecla	Código en Hex
Esc	01	*Left/Right orrow*	0F
!1	02	*Q*	10
@2	03	*W*	11
#3	04	*E*	12
$4	05	*R*	13
%5	06	*T*	14
^6	07	*Y*	15
&7	08	*U*	16
**8*	09	*I*	17
(9	0A	*O*	18
)0	0B	*P*	19
_=	0C	*{[*	1A
+=	0D	*}]*	1B
Backspace	0E	*Enter*	1C
Ctrl	1D	*|*	2B
A	1E	*Z*	2C
S	1F	*X*	2D
D	20	*C*	2E
F	21	*V*	2F
G	22	*B*	30
H	23	*N*	31
J	24	*M*	32
K	25	*<,*	33
L	26	*>.*	34
:;	27	*?/*	35
" '	28	*RightShift*	36
~ `	29	*PrtScr**	37
LeftShift	2A	*Alt*	38
Spacebar	39	*7Home*	47
Caps Lock	3A	*8Up arrow*	48
F1	3B	*9PgUp*	49
F2	3C	*—*	4A
F3	3D	*4Left arrow*	4B
F4	3E	*5*	4C
F5	3F	*6Right arrow*	4D
F6	40	*+*	4E
F7	41	*1End*	4F
F8	42	*2Down arrow*	50
F9	43	*3PgDn*	51
F10	44	*0Ins*	52
F11	D9	*Del*	53
F12	DA		
Num Lock	45		
Scroll Lock	46		

APÉNDICE D

ÍNDICE ALFABÉTICO

—#—

—A—

—F—

Del mismo autor

- Curso de programación con **PASCAL**

ISBN: 84-86381-36-3
226 págs.

- Curso de programación **GW BASIC/BASICA**

ISBN: 84-86381-87-8
320 págs.

- Manual para **TURBO BASIC** Guía del programador

ISBN: 84-86381-43-6
444 págs.

- Manual para **Quick C 2** Guía del programador

ISBN: 84-86381-65-7
540 págs.

- Manual para **Quick BASIC 4.5** Guía del programador

ISBN: 84-86381-74-6
496 págs.

- Curso de programación **Microsoft COBOL**

ISBN: 84-7897-001-0
480 págs.

- Enciclopedia del lenguaje **C**

ISBN: 84-7897-053-3
888 págs.

- Curso de programación **QBASIC y MS-DOS 5**

ISBN: 84-7897-059-2
384 págs.

- Curso de programación **RM/COBOL-85**

ISBN: 84-7897-070-3
396 págs.

- Microsoft **Visual Basic** Aplicaciones para Windows

ISBN: 84-7897-092-4
456 págs.

- El abecé de **MS-DOS 6**

ISBN: 84-7897-114-9
224 págs.

- Programación orientada a objetos **con C ++**

ISBN: 84-7897-118-1
496 págs.

- Enciclopedia de **Visual Basic**

ISBN: 84-7897-132-7
760 págs.

- Microsoft **Visual C++** Aplicaciones para Windows

ISBN: 84-7897-180-7
850 págs.

Catálogo Gratuito

Envíe hoy por correo este cupón y recibirá un catálogo completo de nuestros libros.

NOMBRE _____

CALLE _____

CIUDAD _____

CODIGO POSTAL _____ *PROVINCIA* _____

TELEFONO _____ *FAX* _____

Título del libro adquirido:

Curso de Programación C/C++

¿Qué opinión general le merece?

_____Excelente _____Bueno _____Regular _____Flojo

¿Por qué escogió este libro?

_____Me lo recomendó un amigo
_____Me lo recomendó un dependiente de librería
_____Ví publicidad suya en _____
_____La reputación del Autor
_____La reputación de Ra-Ma
_____Lo ví en un catálogo de Ra-Ma
_____Por necesidad laboral o de estudios
_____Leí una crítica suya en _____
_____Lo ví expuesto en librería _____Grandes superficies
_____Otros: _____

¿Dónde compró este libro?

_____Librería _____Librería Grandes Superficies
_____Tienda de productos informáticos
_____Por catálogo (Nombre: _____)
_____Directamente a Ra-Ma
_____Otros: _____

¿Compró este libro con cargo a?

_____Presupuesto Particular _____Presupuesto de Empresa

¿Cuántos libros sobre informática compra al año?

_____1-3 _____3-5 _____5-7 _____7-9 _____más de 10

¿Cuántos libros de Ra-Ma ha comprado?

_____1-3 _____3-5 _____5-7 _____7-9 _____más de 10

Indique el nivel de conocimientos informáticos que ha cubierto con este libro:

_____Principiante _____Medio _____Avanzado

¿Con qué clase de ordenador trabaja?

_____Apple Mac _____IBM-PC o compatible _____Otros _____

¿Qué tipo de programas informáticos utiliza con asiduidad?

_____Animación/Tratamiento de Imágenes y Presentaciones Gráficas
_____Autoedición
_____Bases de datos
_____Comunicaciones
_____Contabilidad
_____Entornos C.A.D.
_____Entornos estadísticos y matemáticos
_____Entornos integrados/Automatización de oficinas
_____Gestión económica
_____Hojas de cálculo
_____Idiomas
_____Lenguajes o técnicas de programación
_____Procesadores de Texto
_____Redes
_____Sistemas operativos
_____Utilidades en General
_____Otros: _____

¿Dónde situaría su actividad laboral?

_____Administración/Secretariado
_____Dirección
_____Gestión/Supervisión
_____Ingeniero/Técnico
_____Presidencia
_____Otros: _____

Especifique los pros y contras que ha encontrado a este libro: _____

_____

_____

_____

_____

RA-MA Editorial
Carretera de Canillas, 144
28043 MADRID
ESPAÑA

ra-ma