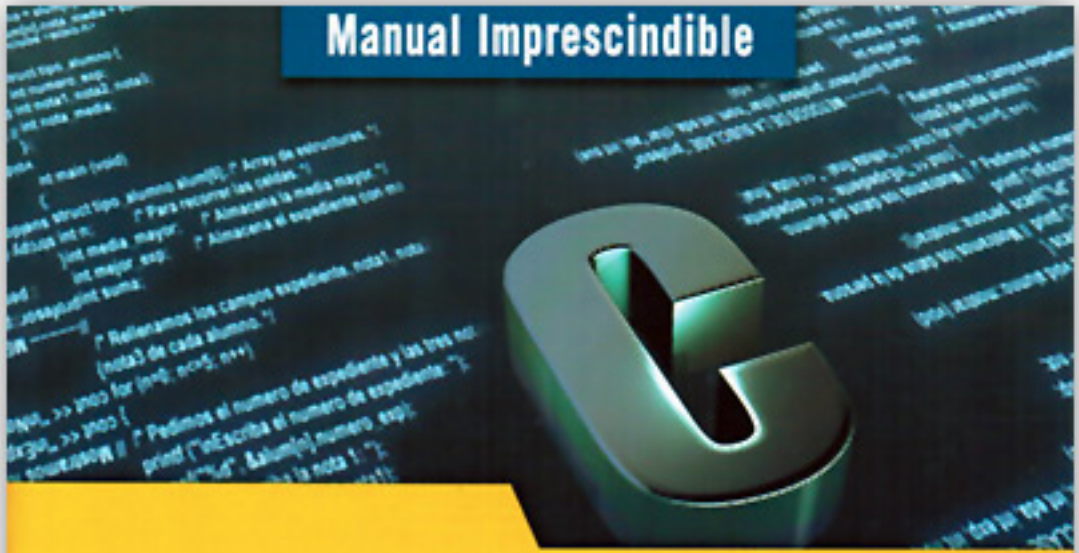


C/C++

Edición revisada y actualizada 2010

ANAYA MULTIMEDIA

Miguel Ángel Acera García

Manual Imprescindible de C/C++

Edición revisada y actualizada 2010

Miguel Ángel Acera García

ANAYA

Índice de contenidos

Introducción

El lenguaje estructurado C fue creado por Dermis Ritchie a principios de los años setenta y con él fue reescrito gran parte del sistema operativo UNIX, de ahí que C esté muy ligado a UNIX. En 1980, Bjarne Stroustrup comenzó a desarrollar el lenguaje de programación C++, que supone una evolución y refinamiento del lenguaje C y cuya principal característica es que es un lenguaje orientado a objetos. Tanto C como C++ permiten realizar todo tipo de aplicaciones: sistemas operativos, procesadores de texto, hojas de cálculo, bases de datos, etc. Esto ya da una idea de la potencia del lenguaje y del motivo por el que está de actualidad aún. Este libro es ideal para todo tipo de personas que utilicen este lenguaje: estudiantes, profesores, autodidactas y programadores en general. Los motivos son varios:

- El libro utiliza un lenguaje muy sencillo.
- Se explican detalladamente todos los aspectos básicos para conseguir una buena base.
- Se incluyen ejemplos comentados muy representativos y ejercicios para resolver.
- Contiene una gran cantidad de ilustraciones que dan claridad a las explicaciones.
- Se hace un recorrido desde C hasta el lenguaje C++, diferenciando ambos lenguajes.
- Se estudian todos los aspectos principales de estos lenguajes, que suelen ser los impartidos en las enseñanzas y los mínimos requeridos para poder programar:
  - Pseudocódigo.
  - Historia de C y C++.
  - Variables, constantes, operadores, control del flujo.
  - Entrada y salida estándar.
  - Punteros y referencias.
  - Estructuras, *arrays,* funciones, ficheros.
  - Estructuras dinámicas: listas, pilas y colas.
  - Programación orientada a objetos: sobrecarga, herencia, polimorfismo.
  - Técnicas de programación: convencional, modular, estructurada, orientada a objetos.
  - Algoritmos de ordenación y de búsqueda.
  - Bibliotecas, C y C++ en Internet.
  - Y mucho más.

Cómo usar este libro

Y este libro... ¿para quién es?

Este libro se ha escrito pensando en todas aquellas personas que sin conocimientos previos desean aprender sin esfuerzo a programar en el lenguaje C/C++, como estudiantes y autodidactas o programadores noveles. No obstante, también es de gran ayuda para aquellas que con un mayor nivel quieran comprender mejor los conceptos más básicos de este lenguaje, como profesores y programadores en general.

El libro está lleno de ejemplos e ilustraciones; ambos son los principales soportes que se utilizan para aclarar todas las explicaciones sobre los distintos aspectos de la programación en C y C++. Algunos programas de ejemplo no tienen ninguna utilidad aparente, ya que su principal objetivo es mostrar cómo se pueden aplicar los nuevos conceptos. A medida que se avanza y los nuevos conceptos se fijan, los ejemplos pasan a ser de gran utilidad incluso en la vida cotidiana.

El libro también contiene ejercicios propuestos con el fin de que el lector los realice y practique.

La estructura del libro se basa en ir retrasando aquellos conceptos que pueden resultar más complicados hasta llegar al momento en que se hacen necesarios, por lo que se comprenden mejor.

Ya desde el primer capítulo nos sumergimos en el mundo de la programación, mostrándonos algunas de las aplicaciones de la informática hoy en día y, rápidamente, enseñándonos cómo hacer nuestros primeros algoritmos.

El segundo capítulo hace un recorrido por la historia y característica del lenguaje C y C++.

Desde el capítulo 4 al capítulo 12 aprenderá a crear todo tipo de programas básicos, que posteriormente podrá mejorar empleando otros instrumentos. Las funciones constituyen la base de la programación modular; las verá en el capítulo 13.

El manejo de ficheros (crear ficheros, grabar datos en un fichero, leer datos de un fichero...) es una de las partes que siempre resulta más interesante y que encontraremos en el capítulo 14.

Las estructuras dinámicas permiten almacenar datos en memoria sin malgastarla, aprovechando mejor los recursos del ordenador. Cuáles son, en qué consisten, cómo se crean y cómo se manejan puede encontrarlo en el capítulo 15.

En el capítulo 16, de la mano de C++, entramos en la programación orientada a objetos, donde se verán los principales aspectos de ésta: clases, herencia, polimorfismo, etc.

En el capítulo 17 se recogen algunas técnicas de programación, que son de gran importancia para el lector, pues le mostrarán su evolución y cuáles no son apropiadas.
En el capítulo 18 se analizan los principales algoritmos de búsqueda y de ordenación.
Finalmente, encontrará varios apéndices donde se recoge información complementaria que le será de gran ayuda, como las bibliotecas de C y C++.
Es importante destacar que todo el código fuente que aparece en el libro está exento de tildes para evitar problemas que se pueden presentar con algunos entornos de desarrollo.

Novedades

En esta nueva edición actualizada se han incluido importantes novedades y mejoras, como por ejemplo la presentación de algunos de los entornos de desarrollo de programas en C/C++ más utilizados y que permitirán al lector conocer mejor las herramientas necesarias para poner en práctica todo lo aprendido.
Igualmente se ha añadido un nuevo capítulo sobre control de errores, una tabla ASCII, más funciones de C/C++, más explicaciones, nuevos ejemplos comentados y ejercicios.

Convenios utilizados en este libro

Los capítulos de este libro están estructurados de forma que facilitan su lectura y estudio.
Comienzan con una página de presentación en la que podemos ver qué se aprenderá. Le sigue una introducción al tema, el propio tema y un resumen.
Algunos capítulos incluyen también ejercicios para resolver, que suponen un refuerzo.
Cuando haya alguna información a destacar nos encontraremos con recuadros como el siguiente:

Nota:

Aquí aparecerá aquella información a destacar.

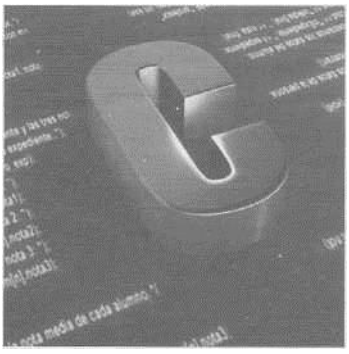
Cuando haya alguna información importante o que requiera cierta precaución, los recuadros serán:

Advertencia:

Aquí aparecerá aquella información a la que hay que prestar especial atención.

Una ayuda más: Información de soporte

En la página Web de Anaya Multimedia `http: //www. anayamultimedia.es,` encontrará material relacionado con este libro. Puede descargarlo a través del menú Soporte técnico>Complementos. A continuación introduzca el código comercial 2311197.
Además el autor deja a su disposición una dirección de correo electrónico en la que responderá a las dudas que envíe acerca de programación en C/C++ así como cualquier otro tipo de cuestión que esté relacionada con este libro:
`formacion@maacera.com`.
Si lo desea también puede visitar la sección "formación" de la página Web `(http://www.maacera.com),` en la que encontrará más códigos y explicaciones del lenguaje C/C++.



Capítulo 1

Introducción a la programación

En este capítulo aprenderá:

- Conceptos básicos de programación.
- Los pasos a seguir para resolver un problema mediante un programa.
- Las características fundamentales de los algoritmos.
- A diseñar algoritmos elementales en pseudocódigo.
- Cómo se obtiene un programa ejecutable a partir de un algoritmo.
- A aplicar la lógica de algoritmos a la vida cotidiana.

Los programadores y las aplicaciones de la informática

Hoy en día son muchas las aplicaciones de la informática. En invernaderos se utilizan ordenadores con programas que controlan aspectos como la temperatura, la humedad y el riego para conseguir mejores plantas y frutos. En aviación se utilizan simuladores de vuelo para que los pilotos entrenen y programas para controlar el tráfico aéreo. Los usuarios de ordenadores de todo el mundo manejan programas de distintos tipos: procesadores de texto, hojas de cálculo, sistemas gestores de bases de datos, sistemas operativos, navegadores de Internet, juegos, reproductores de audio, reproductores de vídeo, etc. También hay ordenadores en los edificios inteligentes, en cajeros automáticos y en muchos otros lugares.
Pues todos los programas que están en los ordenadores son hechos por programadores.

¿Por qué aprender lenguajes y técnicas de programación?

La principal razón por la que las personas aprenden lenguajes y técnicas de programación es para utilizar el ordenador como una herramienta y resolver problemas eficazmente. Para la resolución de un problema mediante un programa de ordenador se requieren al menos los siguientes pasos:

1. Definición o análisis del problema: ¿Cuál es el problema?
2. Diseño del algoritmo: ¿Qué pasos hay que seguir para resolverlo?
3. Codificación: Transformación del algoritmo en un programa.
4. Ejecución y validación del programa: ¿El programa hace lo que debería hacer?

Hasta este momento hemos utilizado algunas palabras que vendría bien aclarar y que a continuación pasamos a definir.

- Lenguaje de programación: Conjunto de palabras, símbolos y reglas utilizados para controlar las operaciones a realizar en una computadora. Ejemplos de lenguajes de programación son: C/C++, Java, PHP y C#, entre otros.
- Algoritmo: Serie de pasos a seguir para solucionar un problema.

- Programa: Secuencia de instrucciones, escritas en un lenguaje de programación, que resuelven un problema.
- Instrucción: Orden que se da al ordenador para que éste lleve a cabo una determinada operación.

Algoritmos

Un programador es una persona con capacidad para resolver problemas, por lo que para llegar a ser un programador eficaz se necesita aprender a dar soluciones a los problemas de un modo riguroso y sistemático. Esto exige el diseño previo de un algoritmo que lo resuelva de la mejor manera posible y que dará lugar, finalmente, al programa de ordenador, como se muestra en la figura 1.1.

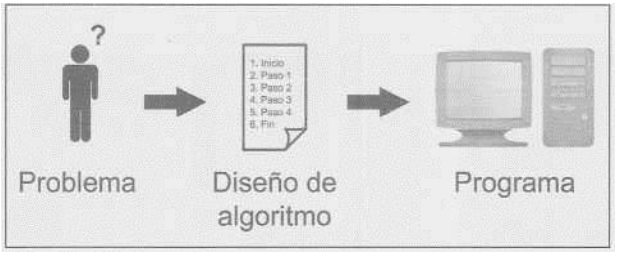


Figura 1.1. Paso del problema al programa.

Las características fundamentales que debe cumplir todo algoritmo son:

- Debe indicar exactamente el orden en el que se realiza cada paso.
- Debe obtenerse el mismo resultado cada vez que se sigan sus pasos con los mismos datos.
- Debe ser finito, es decir, si se siguen los pasos de un algoritmo éste debe finalizar en algún momento.
- Debe tener un primer y único paso que marca dónde empieza.
- Debe tener un último y único paso que marca dónde termina.

Por ahora escribiremos los algoritmos en un lenguaje natural, similar al que utilizamos normalmente. Esto es, si por ejemplo deseamos en un paso de un algoritmo mostrar en la pantalla el texto "Hola", lo único que habría que hacer es escribir eso mismo: `Mostrar en la pantalla el texto "Hola".`
A continuación analizaremos algunos algoritmos.

Un algoritmo escrito en un lenguaje natural que permita realizar la suma de dos números cualesquiera podría ser el siguiente:

```
1. Inicio.
2. Leer mediante el teclado un número y almacenarlo en una variable llamada A.
3. Leer mediante el teclado un número y almacenarlo en una variable llamada B.
4. RESULTADO = A + B.
5. Mostrar en la pantalla el valor de la variable llamada RESULTADO.
6. Fin.
```

Como puede observarse, existe un primer paso llamado `Inicio` y un último paso llamado `Fin` que nos permiten ver dónde empieza y acaba el algoritmo. Cada paso está numerado para conocer cuál va primero y cuál después. Hay que seguir ese orden.

Ahora probaremos el algoritmo. Para hacernos una idea más clara del proceso nos iremos fijando en la secuencia de pantallas de la figura 1.2, que nos mostrarán lo que podría ir ocurriendo a lo largo de la prueba si ésta se llevase a cabo en el ordenador.

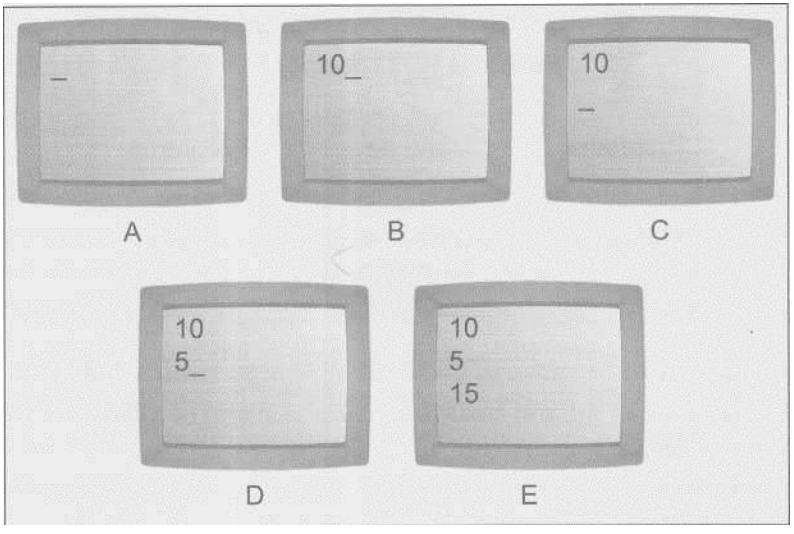


Figura 1.2. Secuencia de pantallas del algoritmo.

El primer paso nos indica el principio del algoritmo. El segundo paso permite al usuario escribir un número mediante el teclado. En este paso aparece en la pantalla un cursor, una pequeña línea horizontal o vertical parpadeando, a la espera de que el usuario teclee un número (véase la figura A).

Una vez tecleado el número, imaginemos que fue el número 10, el cual se puede ver en la pantalla (véase la figura B), sólo se puede continuar con el

siguiente paso cuando el usuario pulse la tecla Intro, señal que se da al ordenador para confirmar el número tecleado. Es en este momento cuando, automáticamente, se le da a la variable **A** el valor de dicho número, entonces **A** vale 10, y llegamos al siguiente paso. En el paso tres ocurre lo mismo (véase la figura C); podemos teclear otro número, imaginemos que fue el número 5 (véase la figura D), y tras pulsar Intro se almacena en la variable **B**, luego **B** vale 5, y se continúa con el siguiente paso. El cuarto paso es una operación aritmética en la que en primer lugar se resuelve la operación que se encuentra a la derecha **(A+B)** de la asignación (=) y que tiene por resultado 15 **(A+B** es igual a 10+5, que es igual a 15).
A continuación, este resultado se le asigna a la variable **RESULTADO**, gracias a la asignación (=), así que ahora **RESULTADO** vale 15. El paso cinco se encarga de hacer aparecer el valor de la variable **RESULTADO** en la pantalla (véase la figura E). Por último se llega al paso seis, que nos indica que hemos llegado al final del algoritmo.

Nota:

En matemáticas el símbolo igual (=) significa que lo que hay a cada lado de éste es idéntico, es decir, es un símbolo de igualdad, por ejemplo 5=5 o X=10. Sin embargo, en programación el símbolo igual (=) significa que el valor de lo que hay a su derecha se le asigna a la variable que hay a su izquierda, es decir, es un operador de asignación. Por este motivo, cuando aprendemos programación el operador de asignación también se suele representar con una flecha que apunta hacia la izquierda, lo que permite representar deforma clara y gráfica cómo funciona.

Bien, al parecer nuestro algoritmo suma correctamente los dos números tecleados por el usuario.
Cuando realizamos este proceso anterior, que consiste en probar el algoritmo con unos valores cualesquiera y observar cómo se comporta y si funciona correctamente, se dice que hacemos una traza.
La idea es que si volvemos a hacer otra traza considerando igualmente que el primer valor que el usuario teclea es 10 (para la variable **A)** y el segundo valor es 5 (para la variable **B),** el resultado debe ser el mismo que antes **y** transcurrir todos los pasos de igual manera.
Puede hacer más trazas con otros números para las variables **A** y **B.** Hágalas ayudándose con un lápiz y papel, éstos serán sus mejores amigos de aquí en adelante, además del ordenador y este libro, claro.

En cualquier caso, observando las imágenes que nos han ido indicando lo que se mostraría en pantalla, nos damos cuenta de que aparecen una serie de números que podrían inducir a errores al usuario por no saber exactamente qué significan. Esto se soluciona añadiendo al algoritmo pasos que permitan mostrar mensajes de texto como se muestra a continuación.

```
1. Inicio.
2. Mostrar en la pantalla el texto "A:".
3. Leer mediante el teclado un número y almacenarlo en la variable A.
4. Mostrar en la pantalla el texto "B:".
5. Leer mediante el teclado un número y almacenarlo en la variable B.
6. RESULTADO = A + B.
7.  Mostrar en la pantalla el texto "A+B=".
8.  Mostrar en la pantalla el valor de la variable RESULTADO.
9. Fin.
```

Repitiendo la traza anterior la secuencia de imágenes de las pantallas sería ahora más clara, ya que aparece delante del cursor el texto que hay entre las comillas dobles "**A:** " o "**B** : " para indicar al usuario que el número que va a teclear es para asignárselo a la variable **A** o **B**, respectivamente. Cuando aparece el valor del resultado mostramos delante el texto que hay entre las comillas dobles "**A+B=**", indicando así al usuario que el número que aparece a continuación es el resultado de sumar **A** más **B**. La figura 1.3 muestra la secuencia de pantallas.

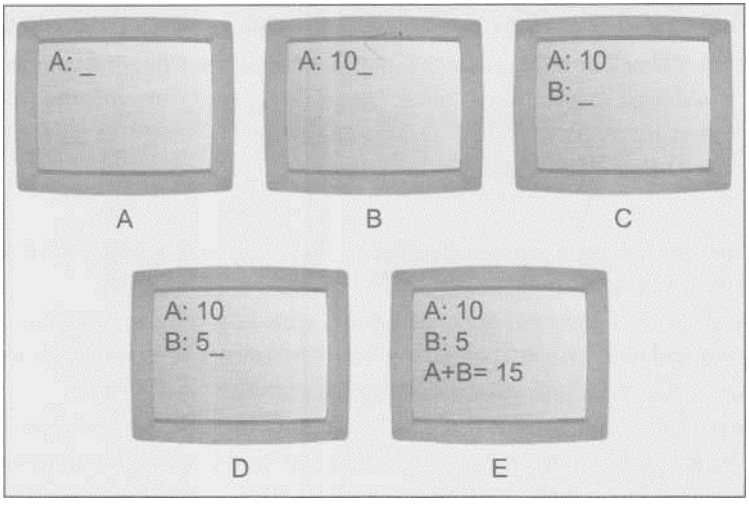


Figura 1.3. Secuencia de pantallas del algoritmo mejorado.

Mucho más claro, ¿verdad? Pues puede mejorarse añadiendo al principio del algoritmo, justo después de **Inicio**, un paso que muestre un mensaje

de texto a modo de enunciado, por ejemplo: "A continuación, se le pedirá un número A y después un número B. Seguidamente se le mostrará por pantalla el resultado de sumar A + B". Si este mensaje es lo primero que ve el usuario, éste se tranquilizará, porque sabrá qué va a ocurrir y cómo. Estas mejoras las iremos viendo y aplicando poco a poco a lo largo del libro.
Ahora supongamos que nos dicen que tenemos que diseñar un algoritmo que calcule el resto de la división entera de un número A, dado por el usuario, entre un número B, también dado por el usuario. En primer lugar tendríamos que pensar cómo afrontar el problema. Empezaremos analizando el problema, es decir, examinando cómo es una división entera. Como se indica en la figura 1.4, cuando hacemos una división entera de un número A entre un número B obtenemos un cociente, que es un número entero, y un resto.

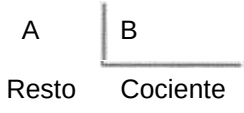


Figura 1.4. División

De manera que B por el Cociente y todo ello más el Resto da igual a A:

```
A = (B • Cociente) + Resto
```

Despejando el Resto obtenemos:

```
Resto = A - (B • Cociente)
```

Luego ya tenemos la fórmula para conseguir el resto. El usuario será quien nos dé un valor para A y otro para B, pero antes de aplicar la fórmula anterior tendremos que calcular el cociente de A entre B de la siguiente forma:

```
Cociente = A / B
```

Así, un algoritmo sencillo quedaría:

```
1. Inicio.
2. Leer mediante el teclado un número y almacenarlo en la variable A.
3. Leer mediante el teclado un número y almacenarlo en la variable B.
4. Cociente = A / B.
5. Resto = A - (B • Cociente).
6. Mostrar en la pantalla el valor de la variable Resto.
7. Fin.
```

Comprobemos si funciona el algoritmo haciendo una traza. En el primer paso encontramos el inicio del algoritmo. En el segundo paso el usuario teclea, por ejemplo, el número 5, que se almacena en la variable **A.** En el tercer paso el usuario teclea, por ejemplo, el número 2, que se almacena en la variable **B.**

En el cuarto paso se calcula `A` entre `B`, es decir, 5 / 2, y la parte entera del resultado, 2, se almacena en la variable `Cociente`. En el quinto paso se almacena en la variable `Resto` el resultado de `A - (B • Cociente)`, es decir, 5 - (2 • 2), lo que es igual a 1. En el sexto paso se muestra en la pantalla el valor de la variable `Resto`, que es 1. En el séptimo paso encontramos el indicador de fin del algoritmo.
Seguro que ya está pensando en hacer al algoritmo unas cuantas mejoras añadiendo mensajes de texto como en el anterior. Adelante, ésa es una muy buena actitud en programación.
Recuerde que cuando hacemos trazas y nos encontramos con pasos en los que se leen datos mediante el teclado nos inventamos lo que un usuario cualquiera podría haber tecleado, así que puede repetir la traza anterior con los números que quiera.

Diseño de algoritmos: Pseudocódigo y ordinogramas

Los sistemas utilizados normalmente para diseñar algoritmos son los ordinogramas y el pseudocódigo. El pseudocódigo suele tener un estilo similar al que hemos utilizado, pero con unas reglas y más parecido a un lenguaje de programación (véase la figura 1.5).

```
Inicio
   Leer del teclado A;
   Leer del teclado B;
   RESULTADO = A + B;
   Visualizar RESULTADO;
Fin
```

Figura 1.5. Pseudocódigo.

No obstante, se puede considerar perfectamente pseudocódigo a la forma que hemos visto para diseñar los algoritmos mediante un lenguaje natural. Así pues, a partir de ahora hablaremos de algoritmos en pseudocódigo.
Por otro lado, los ordinogramas consisten en una representación gráfica del algoritmo haciendo uso de unas reglas (véase la figura 1.6).
Ante estas dos formas de representación continuaremos con el pseudocódigo, ya que resulta más claro para algoritmos de tamaño medio/grande y más apropiado para la finalidad de este capítulo.

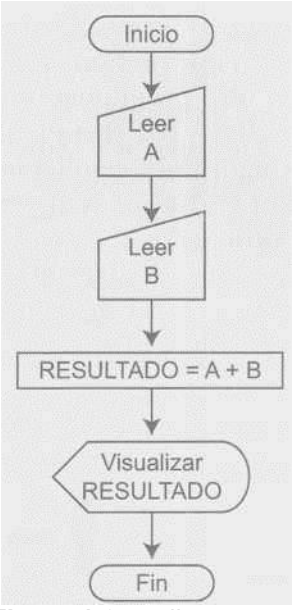


Figura 1.6. Ordinograma.

Los algoritmos que se diseñan para resolver un problema no tienen por qué escribirse primero en pseudocódigo (u ordinogramas). Hay quien directamente los escribe en su lenguaje de programación preferido, pero ésta es una mala técnica propia de programadores noveles, pues con el tiempo suele dar lugar a muchas complicaciones.

La idea de diseñar previamente un algoritmo en pseudocódigo (o mediante ordinogramas) es la de hacer un boceto del método a emplear para obtener la solución. Es este esquema sobre el que se trabaja hasta obtener un algoritmo bien definido y eficaz.

De momento, es importante practicar el diseño de algoritmos en pseudocódigo y probarlos mediante trazas, haciendo uso de lápiz y papel para anotar el valor que toma cada variable en cada paso y lo que aparecería en pantalla.

Lenguajes de programación, intérpretes y compiladores

Los algoritmos que hemos visto están escritos en un lenguaje natural (pseudocódigo), de manera que cualquier persona, informática o no, puede entenderlos. Sin embargo, el ordenador no los comprende.

No obstante, esto permite que un programador, tras leerlo, pueda reescribirlo en el lenguaje de programación que prefiera, por ejemplo C, y que sí entenderá el ordenador. Mientras, otro programador podría reescribirlo en otro lenguaje de programación distinto, por ejemplo Java, que también entenderá el ordenador. Todos esos algoritmos resultantes harían lo mismo, porque esos programadores sólo han traducido el algoritmo escrito en pseudocódigo a otro lenguaje que el ordenador sí entenderá. Al proceso de traducir el algoritmo en pseudocódigo a un lenguaje de programación se le denomina codificación (véase la figura 1.7) y al algoritmo escrito en un lenguaje de programación se le denomina código fuente.

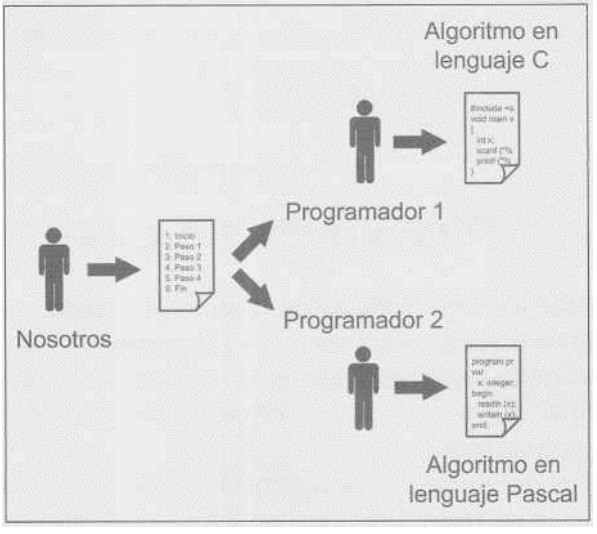


Figura 1.7. Codificación a partir de un algoritmo en pseudocódigo.

Realmente el ordenador no entiende directamente los lenguajes de programación que hemos citado con anterioridad, sino que se requiere un programa que traduzca el código fuente a otro lenguaje que sí entiende directamente, pero que resulta complicado para las personas: el lenguaje máquina. Podemos ver un ejemplo de lenguaje máquina en la figura 1.8.

```
0110  1010  0011
0011  1010  1001
1001  1100  1100
0101  0110  0011
1100  0101  0110
```

Figura 1.8. Lenguaje máquina.

Es el hecho de que el lenguaje máquina es difícil de manejar para el hombre el que lleva a utilizar los otros tipos de lenguajes de programación más fáciles y, por lo tanto, a utilizar un traductor a lenguaje máquina.
Existen dos tipos de programas que realizan esta traducción a lenguaje máquina y, según el lenguaje de programación, se utiliza uno u otro. Son los intérpretes y los compiladores. Los intérpretes, a medida que se avanza por el código fuente, convierten a lenguaje máquina cada instrucción y después la ejecutan. Los compiladores convierten todas las instrucciones en bloque a lenguaje máquina, obteniéndose el programa ejecutable, el cual se puede ejecutar después. El lenguaje C/C++ utiliza un compilador, por lo que nos centraremos en éstos.
Así pues, como se puede observar en la figura 1.9, la codificación la hace un programador y la compilación la hará un programa de ordenador: el compilador.

Figura 1.9. Proceso de transformación del algoritmo en pseudocódigo al programa ejecutable.

Nota:
En muchas ocasiones y de forma coloquial se hace referencia al código fuente con la palabra programa. Por ejemplo, si hemos hecho un código fuente en C, también podemos decir que hemos hecho un programa en C, pero tenemos que tener en cuenta que hasta que no lo compilemos no obtendremos el programa ejecutable.

De aquí deducimos que necesitaremos un compilador para conseguir nuestro programa ejecutable a partir del código fuente que hagamos. De lo contrario no podremos ejecutar y probar en el ordenador nuestros programas.
Como hemos visto hasta el momento, podemos escribir los algoritmos empleando distintos tipos de lenguajes de programación y cada uno requerirá su propio compilador (véase la figura 1.10).

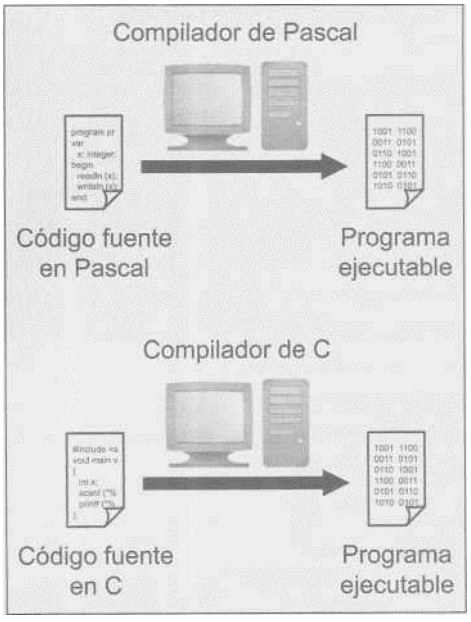


Figura 1.10. Cada lenguaje requiere su propio compilador.

De la misma manera que, valga como ejemplo, personas de distintos países con idiomas diferentes al nuestro quieren a sus propios intérpretes para traducir su idioma al nuestro.

Los algoritmos y la vida cotidiana

Llegado este punto sería una buena idea ir preparándonos para trabajar la lógica y los algoritmos, dos aspectos esenciales y que necesitaremos para poder programar correctamente. ¿Cómo? Pues haciendo simples ejercicios en los que analizamos la forma de pensar y resolver los problemas de la vida cotidiana. Aunque no lo crea, el cerebro es un puro cúmulo de algoritmos. Por ejemplo, ¿cuál sería el algoritmo en pseudocódigo que utiliza para sacar o no a la calle el paraguas? puede ser algo así:

```
1. Inicio
2. Si está lloviendo entonces: sacar el paraguas; si no: dejar el paraguas en casa.
3. Fin
```

Y ¿qué algoritmo utiliza para beber un vaso de agua? Podría ser uno como el siguiente:

```
1. Inicio
2. Coger un vaso.
3. Coger una botella de agua.
4. Echar agua de la botella en el vaso.
5. Beber del vaso.
6. Fin
```

Quizás se le hayan ocurrido más pasos u otros distintos para alguno de estos algoritmos: ¡enhorabuena! Eso es porque su mente es inquieta y busca nuevas soluciones y otras situaciones posibles. Si no es éste su caso piense, por ejemplo, en: ¿y si la botella de agua estaba vacía?

Piense en cómo son los algoritmos que utiliza para subir o bajar una persiana, para buscar una palabra en el diccionario, para cruzar una calle con un semáforo, etc.

Estos algoritmos se parecen poco a los primeros que hemos visto y que se encontraban orientados a ser utilizados finalmente en el ordenador. Sin embargo, los algoritmos de este último apartado van orientados a trabajar la lógica y tomar conciencia de que constantemente aplicamos a todo una serie de pasos para obtener unos resultados, para solucionar problemas.

Ahora intente pensar en más algoritmos de este tipo y en cuáles son sus pasos.

Resumen

En este capítulo hemos aprendido que con los programas de ordenador podemos resolver problemas, pero para ello hay que diseñar un algoritmo en pseudocódigo que, paso a paso, permita obtener la solución. Una vez que tenemos definido el algoritmo, éste puede reescribirse en un lenguaje de programación (codificación), obteniendo el código fuente, y que mediante un compilador dará lugar al programa ejecutable.

Capítulo 2

Conociendo el lenguaje C y C++

En este capítulo aprenderá:

- Los orígenes y la historia del lenguaje C y C++.
- Las características del lenguaje C.
- Las características del lenguaje C++.

Introducción

El lenguaje de programación C permite realizar todo tipo de aplicaciones: sistemas operativos, procesadores de texto, hojas de cálculo y bases de datos, entre otras. De hecho, la mayor parte del sistema operativo UNIX se encuentra desarrollado en C. Esto ya nos da una idea de la potencia del lenguaje y de que todavía está de actualidad.
El lenguaje C parte de la programación estructurada, pero con el paso de los años surge una nueva filosofía, la programación orientada a objetos, que tiene como consecuencia la evolución de C a C++.
A continuación pasaremos a ver más detenidamente la historia de C y C++, así como sus características principales.

Historia de C y C++

En 1969, Ken Thompson y Dermis Ritchie, de los Bell Labs (Laboratorios Bell), iniciaron el desarrollo del sistema operativo UNIX utilizando el lenguaje de programación B, sucesor del lenguaje BCPL. A principios de los años setenta, Dermis Ritchie desarrolló en un ordenador DEC PDP-11, usando UNIX como sistema operativo, el lenguaje de programación C, nombre que se debe a que su predecesor era el lenguaje B. Posteriormente, la gran parte del sistema operativo UNIX fue reescrito mediante el lenguaje C por Thompson y Ritchie.
Otro personaje muy asociado a los comienzos de C es Brian W. Kernighan, quien trabajó junto con Thompson y Ritchie en Bell Labs. Kernighan usó C, el que dice ser su lenguaje preferido, para desarrollar múltiples programas para UNIX, aunque es más conocido por ser coautor junto con Dermis Ritchie de un manual sobre dicho lenguaje.
Con el paso del tiempo, C, al igual que UNIX, fue ganando en popularidad y convirtiéndose en el principal lenguaje de programación para el desarrollo de software de sistemas, como sistemas operativos, generadores de programas y lenguajes de computadora. En la década de los ochenta, cuenta con un gran número de usuarios y diferentes versiones, de modo que ANSI *(American National Standards Institute,* Instituto Nacional Americano de Estándares), en 1983, estableció el comité X3J11 bajo la dirección de CBEMA con el objeto de normalizar este lenguaje de manera que un programa diseñado en ANSI C pudiera ser portable entre diferentes plataformas. En 1989, se estableció definitivamente ANSI C (estándar C89) y, en consecuencia, este estándar fue aceptado por ISO (*International*

Organization for Standardization, Organización Internacional para la Normalización) como ISO/IEC 9899-1990.

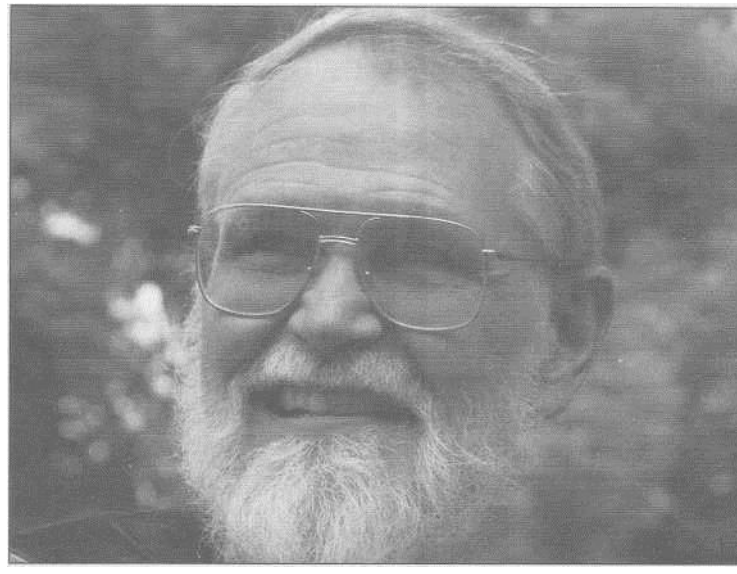


Figura 2.1. Brian W. Kernighan trabajó en el desarrollo de UNIX utilizando el lenguaje C.

En 1980, Bjarne Stroustrup, de los Bell Labs, comenzó a desarrollar el lenguaje de programación C++. El lenguaje C++ supone una evolución y refinamiento del lenguaje C, que duró hasta aproximadamente 1990. No obstante, las características del estándar ANSI de C permanecen en C++, puesto que uno de los objetivos de C++ era mantener la compatibilidad con C, con la idea de preservar los millones de líneas de código escritas y depuradas en C que existían. La principal novedad en C++ es que se trata de un lenguaje orientado a objetos.

Característica de C

El lenguaje de programación C está diseñado para poder crear programas de todo tipo. Algunas de sus características son:

- Es un lenguaje portable, esto es, permite escribir programas de forma que se pueden ejecutar en un ordenador o sistema operativo distinto a aquel en el que se crearon.

- Su número de palabras clave es reducido, sólo tiene 32 palabras clave en el estándar C89 y 37 en el estándar C99.
- Tiene estructuras de control del flujo con las que controlar el orden de ejecución de las instrucciones de un programa.
- Permite crear funciones. Estas pueden ser escritas en archivos distintos al del programa y compilarse por separado.
- C posee características de los lenguajes de nivel alto y, a su vez, de nivel bajo: acceso a bits, manejo de punteros y acceso directo a memoria.
- Es *case sensitive,* es decir, distingue entre mayúsculas y minúsculas, de tal manera que para C no es lo mismo "hola" que "Hola".
- Dispone de una biblioteca estándar que contiene numerosas funciones, además de las extensiones que proporcione cada compilador o entorno de desarrollo. Todas estas funciones permiten realizar múltiples tareas: mostrar datos en la pantalla, recoger datos tecleados, crear ficheros, guardar datos en ficheros, manipular cadenas y obtener la hora o fecha del sistema, entre otras muchas.

Todas estas características hacen de este lenguaje una potente herramienta con la que implementar programas que aprovechen bien los recursos del ordenador y le saquen un gran rendimiento.

Características de C++

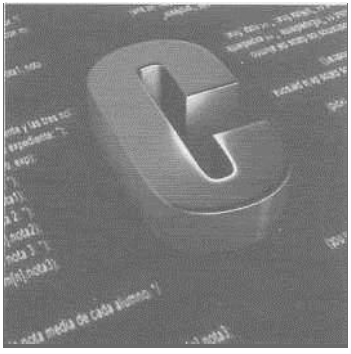
Entre las propiedades de C++ se encuentran las diversas mejora:; de C, además de las derivadas de la programación orientada a objetos, principal característica de C++.
Algunas de éstas son:

- Encapsulación.
- Herencia.
- Sobrecarga de funciones.
- Sobrecarga de operadores.
- Polimorfismo.
- Introducción de los operadores `new y delete.`
- 63 palabras clave para el estándar C++.

La mayoría de las características de C++ las analizaremos cuando tratemos la programación orientada a objetos.

Resumen

Ahora ya sabemos que Dermis Ritchie desarrolló el lenguaje estructurado C y que Bjarne Stroustrup creó el lenguaje de programación C++, una ampliación de C que se caracteriza por ser orientado a objetos. También hemos aprendido que los lenguajes C y C++ son potentes y permiten realizar todo tipo de aplicaciones, incluso sistemas operativos como UNIX.
Todos los aspectos que caracterizan a C y C++ los veremos con más detalle a medida que surjan a lo largo del libro.



Capítulo 3

Compiladores y entornos de desarrollo de C/C++

En este capítulo aprenderá:

- Qué necesitamos para hacer nuestros programas.
- Qué es un Entorno de desarrollo integrado o IDE.

Introducción

Como ya hemos explicado, para ejecutar los algoritmos que escribimos en C o C++ es necesario un compilador, es decir, un programa de ordenador que convierte nuestro código a lenguaje máquina, obteniéndose así un programa ejecutable. Pero ¿qué programa usaremos para escribir los algoritmos? ¿Qué compilador podemos utilizar? Veamos a continuación las respuestas a estas preguntas.

Entorno de desarrollo integrado

Un entorno de desarrollo integrado o IDE *(Integrated Development Environment)* es un programa formado por un conjunto de herramientas que facilitan la tarea del programador, entre las que se encuentran:

- Un editor de texto donde podemos escribir el código, modificarlo y guardarlo en un archivo igual que cuando trabajamos con un procesador de texto. Por lo tanto, también nos permite abrir los archivos de código que habíamos guardado con anterioridad.
- El compilador que, como ya sabemos, es necesario para ejecutar nuestros programas.
- El depurador, que sirve para encontrar y reparar los errores que hay en nuestro código.
- Un sistema de ayuda al programador que nos permite obtener información sobre las distintas funciones y librerías disponibles de C/C++.

Cuando accedemos a un entorno de desarrollo integrado encontramos una cierta similitud con los procesadores de texto al observar el editor de texto y las barras de herramientas.
Para acceder al compilador y al depurador existe un menú en la parte superior al estilo de cualquier programa habitual, así como accesos rápidos desde la barra de herramientas.
A continuación, vamos a dar a conocer y tratar las características principales de algunos entornos de desarrollo gratuitos, pero no debemos perder la pista a otros que pueden ir surgiendo en Internet. Para obtener información completa sobre el uso de cada uno recomendamos acudir a sus correspondientes manuales. Igualmente, no debemos olvidar los entornos de desarrollo y compiladores ofrecidos por Borland, que a pesar de ser de pago, ofrecen

muchas y potentes librerías, así como entornos de desarrollo de mayor calidad, con más funcionalidades y tutoriales, lo que hace que sean usados a nivel profesional. En Internet podemos encontrar algunas versiones gratuitas de entornos Borland.

Dev-C++

Dev-C++ es un completo entorno de desarrollo integrado para la programación del lenguaje C/C++. Disponible para Windows y para Linux, el compilador que incluye es Mingw. Aunque inicialmente se encuentra en inglés, es posible configurarlo en idioma español.
Dev-C++ está bajo licencia GPL *(General Public License)*, por lo que es completamente gratuito, siendo posible descargarlo desde la página Web de sus creadores: `www.bloodshed.net`.

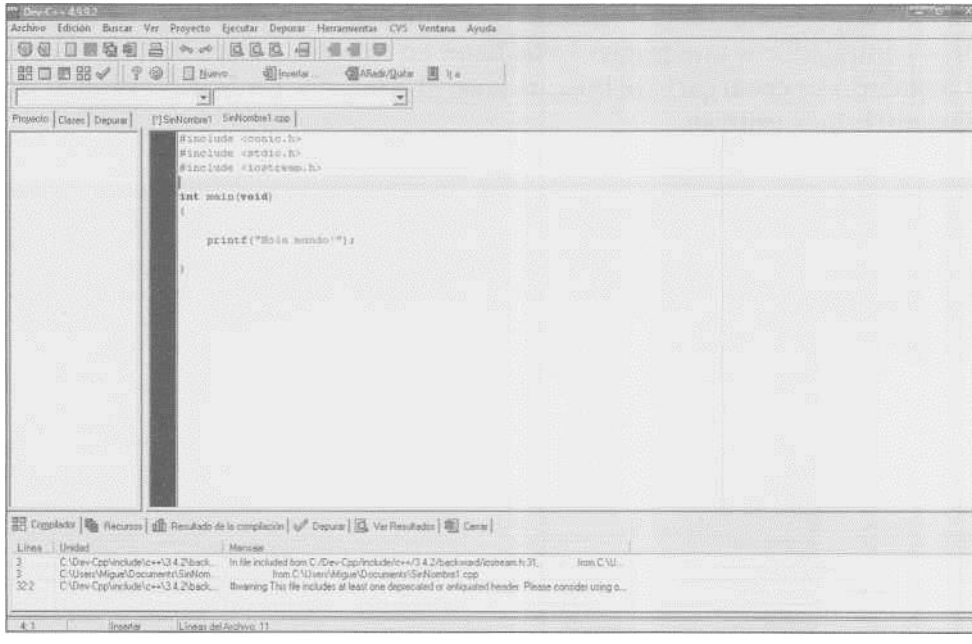


Figura 3.1. Interfaz de Dev-C++.

Cuando ordenamos la ejecución de nuestro código en Dev-C++ se abre automáticamente una ventana de MS-DOS o consola en la cual se ejecuta el programa. Tras finalizar el programa, esa ventana se cierra automáticamente. Este cierre automático supone un inconveniente si hacemos un programa que,

por ejemplo, sólo muestre en pantalla un texto, ya que su ejecución finaliza tan rápido que no nos da tiempo a ver nada: se abre la ventana, se muestra el texto, se cierra la ventana. Todo ello ocurre en décimas de segundo. No obstante, podemos emplear algunos trucos para solucionar esto y que usaremos a lo largo del libro.

Code::Blocks

Code::Blocks es otro IDE para C/C++ gratuito bajo licencia GPL. Una de sus ventajas es que además de estar disponible para Windows y Linux, también lo está para Mac.
Sin embargo, hasta el momento no es posible configurarlo en español, aunque es probable que esta opción esté disponible próximamente. Podemos descargarlo desde la Web `www.codeblocks.org` con el compilador Mingw incluido.
Sin embargo, Code::Blocks también permite ser configurado para utilizar otros compiladores que pudiéramos tener en nuestro ordenador, de hecho, al instalarlo se encargará de buscarlos en el equipo y preguntarnos cuál de ellos queremos utilizar.

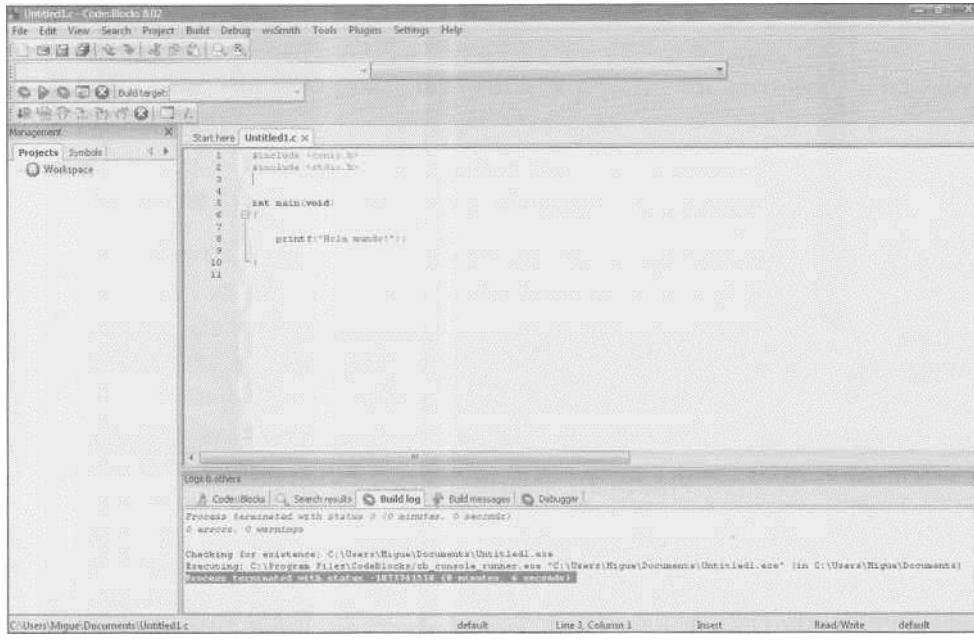
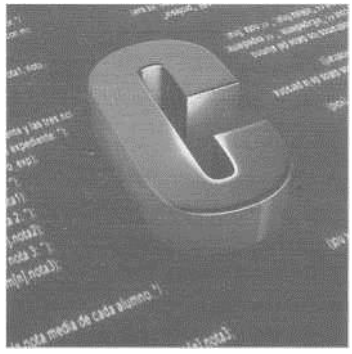


Figura 3.2. Interfaz de Code::Blocks.

Con Code:Blocks no tenemos el inconveniente que antes hemos comentado de Dev-C++ respecto a la ventana de consola, ésta no se cierra hasta que nosotros no lo hacemos manualmente, lo que nos permite ver cómodamente qué es lo último que el programa mostró.

Resumen

Ahora que ya sabemos qué es un entorno de desarrollo integrado y disponemos de uno, ya podemos hacer nuestro primer programa.
En cuanto a qué IDE utilizar, recomendamos probar los dos comentados, aunque Code::Blocks presenta algunas ventajas sobre Dev-C++. No obstante, existen otros de pago que ofrecen una mayor potencia y que son los que deberíamos utilizar si vamos a programar a nivel profesional. Todos los entornos de desarrollo disponen de tutoriales, unos mejores que otros, podemos utilizarlos para aprender a usar cada IDE.



Capítulo 4

Nuestro primer programa

En este capítulo aprenderá:

- Cómo es la estructura básica de un programa.
- Qué son y para qué sirven las bibliotecas.

Introducción

Haremos una primera aproximación a la programación en C / C++ viendo cuál es la estructura básica de un programa. Una vez comprendida esta estructura nos apoyaremos en ella para tratar los siguientes capítulos. Seguramente surjan muchas dudas, pero se verán resueltas poco a poco a lo largo del libro.

Estructura de un programa

Para empezar haremos nuestro primer programa en C, que mostrará en pantalla el mensaje "Este es mi primer programa", como se ve en la figura 4.1.

```
/* El primer programa en C */
/* A continuación indicamos las bibliotecas a usar */
#include <stdio.h>
#include <stdlib.h>

/* Ahora viene el programa principal */
int main (void)
{
   /* La siguiente linea de codigo muestra un mensaje en pantalla
   */ printf ("Este es mi primer programa");

   /* Hacemos una pausa hasta que el usuario pulse Intro */
   fflush(stdin);
   printf("\nPulse Intro para finalizar...");
   getchar ();
}
```

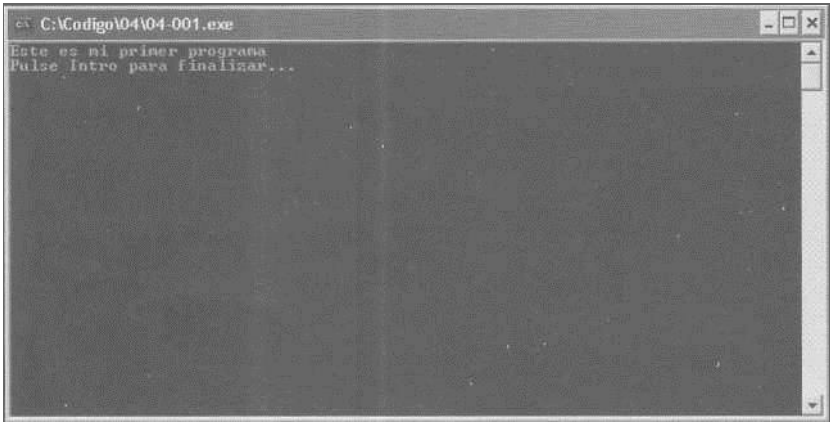


Figura 4.1. Resultado del programa en C.

Analicemos ahora algunas de las partes de este programa.

```
/* El primer programa en C */
```

La primera línea se trata de un comentario. Los comentarios se utilizan para introducir junto al código explicaciones, pero no se ejecutan. En este caso el comentario va entre los símbolos `/*` y `*/`. En este programa podemos observar más comentarios, además del que acabamos de citar. Los explicaremos más adelante en este mismo capítulo.

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

Esta línea permite hacer uso del código de la biblioteca `stdio` mediante la instrucción `#include` y la referencia al fichero `stdio.h` situado entre los signos menor que (<) `y` mayor que (>). Este tipo de fichero se denomina archivo de cabecera. Bajo esta línea de código encontramos otra que utiliza la biblioteca `stdlib`. Trataremos las bibliotecas más adelante en este mismo capítulo.

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

Estas líneas declaran la función `main` o principal.
De momento diremos que una función es un pedazo de código encerrado entre llaves al que se le da un nombre. La forma de escribir una función es: el nombre de ésta seguido de un paréntesis abierto (() y un paréntesis cerrado ()) y, a continuación, una llave abierta ({) y una llave cerrada (}). El código que se escribe entre las llaves será el elegido por nosotros para hacer aquello que queramos. Más sobre las funciones en el capítulo que las trata.
El código escrito dentro de la función cuyo nombre es `main` es el único que se ejecuta inicialmente, por lo que es obligatorio que todos los programas tengan esta función.
Anteponer a la palabra `main` la palabra `int` aumenta la portabilidad del programa. Este aspecto y por qué hemos escrito `void` entre los paréntesis lo explicaremos en el capítulo de las funciones.

Advertencia:

Los programas escritos en C y C++ tienen que tener una, y sólo una, función main, cuyo contenido es el único que se ejecuta en un principio cuando hacemos funcionar el programa. De esta manera, la función main sería igual que la puerta principal de una casa, es por donde se entra y siempre hay una. En este ejemplo la casa sería nuestro programa, como representa la figura 4.2.

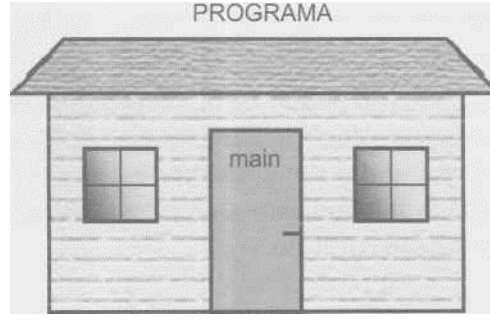


Figura 4.2. Comparación entre una casa y un programa.

Si lo que deseamos es mostrar un mensaje por pantalla escribiríamos la siguiente línea dentro de las llaves, como hemos hecho en el ejemplo que estamos analizando.

```
printf ("Este es mi primer programa");
```

Esta línea imprime en pantalla el mensaje "Este es mi primer programa" mediante la función `printf`. La función `printf` permite mostrar el texto escrito entre las comillas dobles dentro de los paréntesis que hay tras el nombre de la función. Observe que esta instrucción termina con punto y coma. Para usarla es necesario incluir al principio del programa la línea:

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

ya que la función `printf` pertenece a la biblioteca `stdio`.

```
getchar () ;
```

La línea anterior detiene la ejecución de programa hasta que pulsemos la tecla **Intro** o **Enter.** En algunos compiladores, como Dev-C++, esta línea nos permite ver la ejecución del programa antes de que se cierre rápidamente la ventana del mismo. No debemos preocuparnos si hay algo que no entendemos en este momento, poco a poco y a lo largo del libro las iremos explicando, es demasiado pronto para explicarlo todo.

Ahora haremos nuestro primer programa en C++, que volverá a mostrar el mismo resultado en pantalla que el de la figura 4.1.

```
/* El primer programa en C++ */
/* A continuación indicamos las bibliotecas a usar */
#include <iostream.h>

int main (void)
{
    /* La siguiente linea de codigo muestra un mensaje en pantalla */
    cout << "Este es mi primer programa";
```

```
    /* Hacemos una pausa hasta que el usuario pulse Intro */
    fflush(stdin);
    cout << "\nPulse Intro para finalizar...";
    cin.get () ;
}
```

En este programa la principal línea que cambia es:

```
cout << "Este es mi primer programa";
```

que realiza la misma tarea que `printf`. En este caso el texto entrecomillado se envía mediante el operador « a la pantalla `(cout)`. Observe que hay un punto y coma al final.
Para usar el objeto `cout` es necesario incluir la siguiente línea al principio del programa:

```
#include <iostream.h>
```

ya que el objeto `cout` pertenece a la biblioteca `iostream`.
También ha cambiado la forma de detener la ejecución del programa hasta que el usuario pulse la tecla **Intro** o **Enter,** siendo ahora:

```
cin.get();
```

Llegados a este punto, ya sabemos que para crear un programa se requiere una función llamada `main`, que es la que se ejecuta al hacer funcionar el programa, y que su estructura básica es:

```
int main (void)
{
    /* aqui escribimos el codigo del programa */
}
```

También sabemos mostrar un mensaje en pantalla y que según la función u objeto que empleemos para ello necesitaremos añadir la palabra `#include` con el nombre de la biblioteca correspondiente.
Con estos ejemplos podemos ir teniendo una idea de la forma que van a tener los programas escritos en C y C++.

Comentarios

Los comentarios son explicaciones del programa que se escriben junto al código, pero no se ejecutan. La finalidad de los comentarios es ayudar a entender claramente qué hace y cómo funciona el programa.
Deben ser claros y abundantes, sin dar lugar a dudas acerca del código. Esto es, deben permitir que toda persona entienda el código rápida y fácilmente.

Por ejemplo, el siguiente programa incluye comentarios que no dan mucha información sobre el programa.

```
/* El primer programa en C */

#include <stdio.h>

int main (void)
{
   printf ("Este es mi primer programa");
}
```

Pero añadiendo más comentarios queda mucho más claro. Casi no hace falta saber C para deducir qué hace el programa.

```
/* El primer programa en C */
/* A continuación indicamos las bibliotecas a usar */

#include <stdio.h>

/* Ahora viene el programa principal */
int main (void)
{
   /* La siguiente linea de codigo muestra un mensaje en pantalla */
   printf ("Este es mi primer programa");
}
```

Tengamos en cuenta que incluir comentarios o no en un programa de 10 líneas puede dar igual, pero en un programa de 1.000 líneas se hará necesario, no digamos ya en un programa de 1 millón de líneas de código, ¿verdad? Además, los comentarios facilitan realizar modificaciones futuras en el código de una manera cómoda, sin necesidad de tener que analizar todo el programa para poder encontrar y entender la parte que se quiere cambiar. Imagine un programa que hizo para una empresa hace 3 años, con 20.000 líneas de código, y ¡lo hizo sin comentarios! Ahora le piden que lo actualice adaptándolo a las nuevas necesidades de la empresa. Obviamente se le presenta un doble trabajo: adaptarlo y, previamente, analizar sus 20.000 líneas de código, que ya no le parecerán ni suyas.

Por lo tanto, incluir comentarios ha de convertirse en algo automático mientras se programa, porque si se espera a finalizar el programa para poner los comentarios será mejor aprovisionarse de café y armarse de paciencia.

Se puede comentar qué hace tal línea, qué hace una parte del programa o qué hace el programa en sí. Hay que comentar todo lo que se crea oportuno, de importancia o de interés por su complejidad.

Los comentarios en C y C++ se introducen entre los símbolos /* y */, que permiten escribir comentarios en una sola línea o bien abarcando varias. En este caso el comentario comienza con /* y finaliza con el primer */ que se encuentre a continuación (véase la figura 4.3).

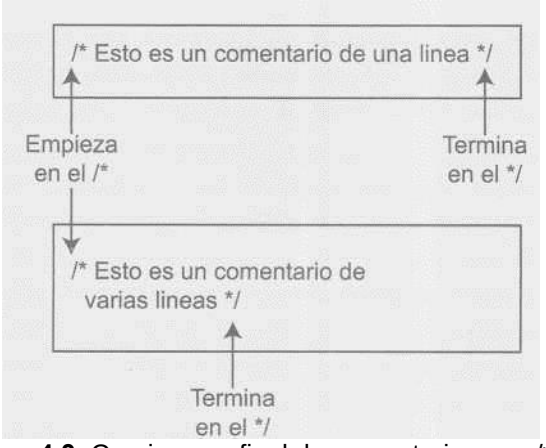


Figura 4.3. Comienzo y final de comentarios con /* y */.

Además, C++ incluye el símbolo // (dos barras), que se antepone al comentario, y permite escribirlos en una sola línea. Este tipo de comentarios comienza con dos barras y finaliza con el fin de la línea en el que se encuentra, como se indica en la figura 4.4.

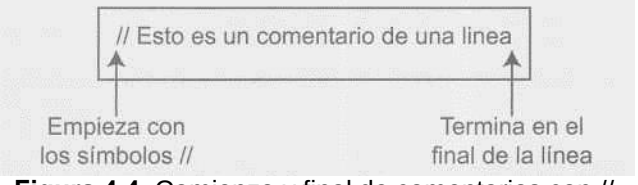


Figura 4.4. Comienzo y final de comentarios con //.

Veamos un ejemplo.

```
/* El primer programa en C++, usando distintos tipos de comentarios.
Este es de varias lineas.
*/
// A continuación indicamos las bibliotecas a usar #include <iostream.h>
int main (void) // Este es el programa principal {
    /* La siguiente linea de codigo muestra */
    /* un mensaje en pantalla */
    cout << "Este es mi primer programa";

    // Hacemos una pausa hasta que el usuario pulse Intro.
    fflush(stdin);
    cout << "\nPulse Intro para finalizar...";
    cin.get();
}
```

Recordemos que los comentarios no se ejecutan, por lo que si ponemos código dentro de comentarios éste no se ejecutará. Así, el siguiente programa no hace nada, porque la única línea de código en la función main está como comentario con / /.

```
/* Este programa no hace nada */
// A continuación indicamos las bibliotecas a usar
#include <iostream.h>

int main (void) // Este es el programa principal
{
    // La siguiente linea es un comentario y no se ejecuta
    // cout « "Este es mi primer programa";
}
```

Lo mismo ocurre en el próximo programa, pero haciendo uso de /* y */ abarcando varias líneas.

```
/* Este programa no hace nada */
// A continuación indicamos las bibliotecas a usar
#include <iostream.h>

int main (void) // Este es el programa principal
{
    /*
    La siguiente linea de código esta dentro de un comentario,
    asi que no se ejecuta:
    cout << "Este es mi primer programa";
    */
}
```

Una vez puesto el indicador de comentarios, ya sea /* o //, no es necesario dejar un espacio para empezar a escribir la explicación. Tampoco es necesario finalizar la explicación con un punto final u otro signo de puntuación, ni en el caso de los comentarios con //. En el caso de los comentarios con /* y */, éstos pueden incluir saltos de línea, como ya se ha podido observar en los ejemplos.
A lo largo de todo el libro hay numerosos ejemplos en los cuales aparecen comentarios.

Bibliotecas

Los compiladores de C y C++ suelen ir acompañados de una serie de bibliotecas, también llamadas librerías. Estas bibliotecas son colecciones de muchos trozos de código, llamados funciones, que realizan una tarea: mostrar un mensaje por pantalla, emitir un sonido por el altavoz del ordenador

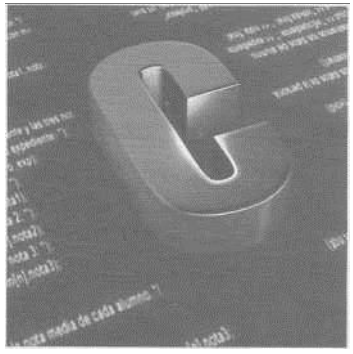
o abrir un fichero, etc. Nosotros no tenemos que preocuparnos de entender esos códigos o de cómo se hacen tales cosas, sino sólo de cómo se usan. Esto simplifica mucho la tarea al programador.
Por ejemplo, normalmente si queremos escuchar un CD utilizamos un reproductor de CD. No sabemos (o no tenemos por qué saberlo) cómo funciona internamente, con conocer qué hay que hacer para usarlo nos es suficiente. Pues el reproductor de CD es como uno de esos trozos de código.
Los trozos de código se encuentran agrupados por temas y cada grupo forma una biblioteca. Tanto los trozos de código como las bibliotecas tienen nombres. Así, los trozos de código relacionados con fechas y horas están en una biblioteca llamada `time` y los relacionados con operaciones matemáticas están en otra biblioteca llamada `math`.
Pues cada vez que queramos usar uno de esos trozos de código tenemos que indicar que vamos a hacer uso de la biblioteca en la que se encuentra. Para ello hay que escribir al principio del programa la instrucción `#include` seguida del archivo de cabecera de la biblioteca entre los signos < y >. El archivo de cabecera tiene el mismo nombre de la biblioteca y su extensión es .h. Si queremos usar la función `free` de la biblioteca `alloe`, entonces el archivo de cabecera es `alloc.h`:

```
#include <alloc.h>
```

Recuerde que en los apéndices puede encontrar un listado de las bibliotecas estándar de C y C++.

Resumen

Ahora que ya conocemos la estructura básica de un programa escrito en C/C++, incluir comentarios y qué son las bibliotecas, tenemos la base necesaria para empezar a tratar los siguientes aspectos del lenguaje ayudándonos de más ejemplos escritos en este lenguaje.



Capítulo 5

Variables y constantes

En este capítulo aprenderá:

- Qué son las variables y constantes.
- Qué nombres podemos darles.
- Cómo darles un valor inicial.

Introducción

Al igual que en matemáticas o física se trabaja con valores en la resolución de problemas, a lo largo del desarrollo de un programa también existe la necesidad de trabajar con éstos. Tendremos valores que permanecerán constantes a lo largo del programa y otros que variarán. A los primeros se les llama constantes y a los segundos variables.

Siguiendo con la referencia de las matemáticas y la física, recordemos constantes como pi (3'14) o el nº e (2'71), y variables como la típica x o `y`, que tomaban distintos valores a lo largo del problema matemático o físico. Pues, como ya hemos dicho, el concepto de constante y el de variable también existen en programación.

Así, podemos considerar que `k` representa siempre al mismo número (por ejemplo: 10) o que x representa un número en un momento (por ejemplo: 3) y otro número en otro momento (por ejemplo: 6). Entonces decimos que `k` es una constante, porque su valor siempre permanece igual, y que `x` es una variable, porque su valor cambia.

Démonos cuenta de que hemos dicho que la constante `k` y la variable `x` son números, y que hemos dado un nombre a ambas ("k" y "x"); pues bien, al nombre de la variable o constante nos referiremos como identificador y nos referiremos como tipo de dato a si será un número u otra cosa lo que almacena. De esta forma, y siguiendo los ejemplos anteriores, `k` es una constante cuyo identificador es "k" y cuyo tipo de dato es un número, mientras que x es una variable cuyo identificador es "x" y cuyo tipo es un número. Pero ¿una variable o constante puede almacenar algo distinto a un número? Sí, puede almacenar un número entero o un número real, e incluso una letra, pero esto lo veremos más adelante, en este mismo capítulo.

Para tener una idea más clara de las variables y constantes podemos representarlas gráficamente como una caja sobre la que se indica su nombre, dentro su valor y debajo su tipo de dato, como vemos en el ejemplo de la figura 5.1. Estas representaciones gráficas nos ayudarán a comprender mejor todo aquello que pueda parecer abstracto. Hay muchas a lo largo del libro, como ya habrá podido observar.

Figura 5.1. Representación gráfica de una variable o constante.

Identificadores

Los identificadores son el nombre que podemos dar a una variable, constante u otro elemento. No son válidos todos los nombres que podamos pensar, hay unas reglas. Pueden contener únicamente los siguientes símbolos: letras mayúsculas (de la 'A' a la 'Z'), letras minúsculas (de la 'a' a la 'z'), guión bajo (_) y números (del 0 al 9). Pero sólo pueden empezar con una letra mayúscula, minúscula o un guión bajo, no pueden empezar con un número. Además, no se permite el uso de la 'ñ' ó 'Ñ', ni de tildes, ni de espacios. En vez del espacio podemos emplear el guión bajo, lo cual nos será de gran ayuda para el caso de identificadores largos. Tampoco pueden coincidir con ninguna de las palabras reservadas de C y C++, que se muestran en la tabla 5.1.

Tabla 5.1. Palabras reservadas de C y C++.

and	and_eq	asm	auto
bitand	bitor	bool	break
case	catch	char	class
compl	const	const_cast	continue
default	delete	do	double
dynamic_cast	else	enum	explicit
export	extern	false	float
for	friend	goto	if
inline	int	long	mutable
namespace	new	not	not_eq
operator	or	or_eq	private
protected	public	register	reinterpret cas
return	short	signed	sizeof
static	static_cast	struct	switch
template	this	throw	true
try	typedef	typeid	typename
union	unsigned	using	virtual
void	volatile	wchar_t	while
xor	xor_eq		

Hay que tener en cuenta que el compilador empleado puede añadir más palabras clave.
Recuerde que C y C++ son *case sensitive,* por lo que las palabras reservadas deben escribirse en minúsculas.
Podemos ver algunos ejemplos de identificadores válidos en la tabla 5.2.

Tabla 5.2. Identificadores correctos.

Edad	Top10
Nombre	_temporal
mi_numero	valor_Auxiliar
FinDeSemana	Codigo
x1	cantidad_ninios

Ejemplos de identificadores no válidos y su causa son los de la tabla 5.3.

Tabla 5.3. Identificadores incorrectos.

Identificador	Error
10Top	Empieza con un número.
código	Tiene una tilde.
cantidad_niños	Tiene una "ñ".
el#contador	Tiene un carácter no válido: una almohadilla (#).
mi numero	Tiene un carácter no válido: un espacio entre "mi" y "numero".
asm	Coincide con una palabra clave de C/C++.
Litros-agua	Tiene un carácter no válido: un guión alto (-).

Por último, en la tabla 4.4 hay otras dos columnas de identificadores válidos que presentan una peculiaridad que comentamos a continuación.

Tabla 5.4. Identificadores distintos por case *sensitive.*

cochE	coche
_cantidad	_Cantidad
casa_1	CaSa_1

Aunque parece que en cada fila de la tabla está escrito lo mismo dos veces, para C/C++ no es así debido a que son *case sensitive,* es decir, no es lo mismo algo en minúsculas que en mayúsculas. De esta forma, C/C++ distinguirá entre la palabra de la primera columna y de la segunda, siendo distinto, por ejemplo, "cochE" de "coche".
Los identificadores pueden tener cualquier longitud y al menos los primeros 31 caracteres serán significativos, pero los compiladores pueden hacer variar estas propiedades.
Muy bien, ya sabemos qué identificadores no podemos usar y qué caracteres podemos emplear en nuestros identificadores de variables y constantes. Pero queda un asunto pendiente: ¿qué identificador o nombre debemos utilizar para una variable o una constante?
Las variables y constantes almacenarán valores correspondientes a alguna cosa, por ejemplo el número de lápices que tenemos en nuestro escritorio, nuestra edad o el número de ruedas de un coche. Pues el identificador debe estar relacionado con aquello que almacena, de lo contrario, nuestro programa comenzaría a ser ilegible y absurdo. Si para estos tres ejemplos anteriores tuviéramos que decidir si se trata de una variable o una constante y su identificador, una posible solución sería la siguiente. Los lápices que tenemos en nuestro escritorio pueden variar, porque se acaben o porque compremos más, luego será una variable. El nombre o identificador de esta variable podría ser `lapices_en_escritorio`, que nos indica con bastante claridad qué almacena. En el ejemplo de nuestra edad, ésta es variable, aunque algunas personas insistan en hacerla constante diciendo frases como "yo ya no cumplo más años". Es broma. Su identificador puede ser tan simple como `mi_edad`. En cuanto al ejemplo del número de ruedas de un coche, queda claro que es una constante. Son siempre cuatro, o cinco si contamos la de repuesto. Su identificador podría ser **RUEDAS_EN_COCHE**. Que el identificador de esta constante aparezca en mayúsculas no es un capricho, veamos la siguiente nota.

Nota:

Por tradición, los identificadores de las variables se escriben completamente en minúsculas y los identificadores de las constantes en mayúsculas. Esto nos ayudará a distinguir de un vistazo si un identificador corresponde a una variable o a una constante.

Una práctica común en los programadores noveles es asignar a todas las variables y constantes identificadores del tipo: **A**, **B**, **X**, **Y**, **Z**. ¿Podría alguien

decirnos qué van a almacenar? ¿El número de plantas de su jardín? O ¿cuántos libros tiene su biblioteca? Debemos huir de esta práctica que nos dificultará mucho la programación.

Nota:

Recuerde utilizar identificadores significativos para las variables y constantes, de manera que le permitan conocer rápidamente qué almacenan.

Tipos de datos

Hasta ahora hemos tratado en los ejemplos variables y constantes cuyo tipo de dato era numérico, pero ya adelantábamos que podían ser de más tipos. C y C++ proporcionan varios tipos de datos básicos. Los mostramos en la tabla 5.5.

Tabla 5.5. Tipos básicos de datos de C y C++.

Tipo de dato	Capaz de contener un
char	carácter
int	entero
float	real (punto flotante de precisión normal)
double	real (punto flotante de doble precisión)
void	sin valor
bool	booleano (exclusivo de C++)

Cada tipo de dato permite almacenar valores pertenecientes a un rango. El rango de valores de cada tipo de dato mostrado en la tabla 5.6 corresponde al mínimo garantizado según las especificaciones del estándar ANSI/ISO de C, pero los compiladores pueden superar estos rangos.
Pero ¿qué significan los números de los rangos de valores de la tabla 5.6? La mejor respuesta es con un ejemplo.
En el caso de que una variable sea de tipo int, al ser su rango de valores de -32767 a 32767, dicha variable podrá almacenar el valor 10, o el 30107, o el -15728, o el -459, o el 0, es decir, cualquier valor que esté comprendido

entre -32767 y el 32767, pasando por el 0. Sin embargo, no puede almacenar el valor 90000, ya que está fuera de su rango de valores por ser un número mayor que 32767.
Por otro lado, aunque más adelante trataremos el tipo `float y double`, adelantaremos que éstos pueden almacenar números con coma, como por ejemplo 3,14, así como números muy grandes, como pueden ser del orden de millones o miles de millones.

Tabla 5.6. Rango mínimo de valores de los tipos de datos.

Tipo	Rango de valores
char	de-127 a 127.
int	de -32767 a 32767.
float	de 1E-37 a 1E+37, con seis dígitos de precisión,
double	de 1E-37 a 1E+37, con diez dígitos de precisión.

Tipo char

Los valores del tipo `char` son caracteres representados entre comillas simples (’).
Son caracteres las letras minúsculas, las letras mayúsculas, los números y otros símbolos del juego estándar de caracteres, como los signos de puntuación. Nosotros trabajaremos con el juego de caracteres de la tabla ASCII *(American Standard Code of Information Exchange,* Código Estándar Americano para el Intercambio de Información). Ejemplos de caracteres son: 'A', '5', '-', '#', '[', 'ñ', '+'.
A cada carácter le corresponde un valor numérico o código siempre positivo, que es el de la posición que ocupa en la tabla ASCII; así, al carácter 'A' le corresponde el valor 65 y, de igual manera, pero a la inversa, al valor 65 le corresponde el carácter 'A'.

Nota:

Saber que a cada carácter de la tabla ASCII le corresponde un valor es algo a tener siempre en cuenta desde el punto de vista de un programador. Aunque de momento no le vea la utilidad a esta característica, la tendremos en cuenta para realizar algunos ejemplos y ejercicios. En los apéndices encontrará una tabla ASCII.

Existe una serie de caracteres que se escriben de una manera especial y, aunque dentro de las comillas simples hay dos caracteres, equivalen sólo a uno. Algunos de éstos se muestran en la tabla 5.7.

Tabla 5.7. Otros caracteres.

Significado	Carácter
Retroceso	'\b'
Salto de línea	'\n'
Tabulación horizontal	'\t'
Comillas dobles	'\"'
Comilla simple	'\''
Barra diagonal invertida	'\\'

Algo que debe quedar claro es que los caracteres numéricos ('1', '2', '3', '4', '5', '6', '7', '8', '9', '0') son eso, caracteres y no números. Si pretendiésemos sumar el carácter '2' y el carácter '4' no obtendríamos el carácter '6'. Esto es porque cuando escribimos los números entre comillas simples representan caracteres, símbolos, no valores. El carácter '6' no es el número 6, es un símbolo (una imagen, un dibujo, un gráfico) que representa al número 6.

Tipo int

El tipo int es un número entero, sin parte fraccionaria. Los números de tipo entero los podemos escribir en forma decimal (sin anteponer un 0 al número), octal (anteponiendo un 0 al número) o hexadecimal (anteponiendo 0x al número).
Ejemplos de números de tipo int son los mostrados en la tabla 5.8.

Tabla 5.8. Valores de tipo int.

Decimal	Hexadecimal	Octal
1	0x24	022
0	0x1	07
24	OxF	0176
4238	0x1 AC	01

Tipo float y double

El tipo `float` y `double` son números con parte entera y parte fraccionaria. Su sintaxis es:

```
[signo] [dígitos] [.] [dígitos] [exponente [signo] dígitos]
```

El signo es más (+) o menos (-); el punto (.) separa la parte entera de la fraccionaria, el exponente es la letra `e` minúscula (e) o mayúscula (E) y los dígitos son una serie de números. Los elementos entre corchetes son opcionales, pero el número no puede empezar directamente por `e` o `E`.
La sintaxis de un número de tipo `float` o `double` puede parecer compleja, pero en la práctica no tiene mayor dificultad. Observemos estos ejemplos: 0.03, -3.5, 1.0, 10e5 y 4.2E-6.
El uso de la letra `e` o `E` seguida de un número se corresponde con la notación científica y su equivalente es el número situado delante de dicha letra multiplicado por `10` elevado al número que sucede a la `e` o `E`. En la tabla `5.9` podemos ver algunos ejemplos de notación científica y su equivalencia.

Tabla 5.9. Notación científica y su equivalencia.

Notación científica	Equivalencia
380.5e7	380.5 multiplicado por: 10 elevado a 5
-7.32E26	-7.32 multiplicado por: 10 elevado a 26
0.03e-9	0.03 multiplicado por: 10 elevado a -9

Generalmente utilizaremos el tipo float para variables en la que deseemos almacenar números muy grandes y/o con decimales. De esta forma, si queremos tener una variable donde almacenar una cantidad numérica de dólares o euros, como pudiera ser 95133.45, usaremos este tipo de dato. También utilizaremos el tipo `float` para una variable en la que deseemos almacenar la estatura de una persona en metros, como pudiera ser 1.82, o su peso en kilogramos, como pudiera ser 75.325.

Nota:

Recuerde que cuando programamos, el punto que aparece en los números, como en 3.14, separa la parte entera de la fraccionaria, no lo confunda con el separador de millares que utilizamos en la vida diaria y que en realidad no existe a la hora de programar.

Por ejemplo, si en un recibo bancario vemos el número 3.758,23, a nivel de programación el número que usamos es 3758.23.

Tipo bool

El tipo de dato bool es exclusivo de C++. Puede tomar únicamente los valores true (verdadero) y false (falso).
El tipo bool es muy útil, por ejemplo, para indicar si ha ocurrido o no un evento. Para ello, podríamos utilizar una variable de este tipo que inicialmente valiese false, lo que significaría que no ha sucedido el evento en cuestión. Si el evento ocurriese haríamos que la variable tomase el valor true. A continuación, si quisiéramos saber si sucedió o no el evento sólo tendríamos que comprobar el valor de esta variable. Es como encender una bombilla cuando sucede algo. Pero esto es sólo un ejemplo.

Tipo void

El tipo void indica sin valor, es decir, nada. Este tipo resultará muy extraño ahora, pero más adelante lo usaremos.

Modificadores short y long

Los modificadores short y long se pueden aplicar precediendo al tipo int. El modificador long también puede preceder al tipo double. Estos modificadores se usan para casos especiales en los que se desea obtener un rango inferior o superior. Normalmente un short int tiene un tamaño de 16 bits, un long int de 32 bits y un long double de 64 bits. En la tabla 5.10 se muestran los rangos mínimos de valores que pueden tomar los tipos de datos, según las especificaciones del estándar ANSI/ISO de C++, cuando se emplean estos modificadores.

Tabla 5.10. Rango mínimo de valores de los tipos de datos con modificadores short y long.

Tipo	Rango de valores
short int	de -32767 a 32767
long int	de -2147483647 a 2147483647
long double	de 1E-37 a 1E+37, con diez dígitos de precisión.

Tanto en `short int` como en `long int`, la palabra `int` puede suprimirse, pudiendo utilizar simplemente `short` y `long`.
En cualquier caso:

- El rango de valores de un `short int` será menor o igual al de un `int`.
- El rango de un `int` será menor o igual al de un `long int`.
- El rango de un `double` será menor o igual al de un `long double`.

Modificadores unsigned y signed

El modificador `unsigned` se puede emplear delante de los tipos `char`, `int`, `short int` y `long int` para obtener un rango positivo de estos tipos, sin signo.
En la tabla 5.11 se muestra cómo afecta el modificador `unsigned` a los rangos mínimos de valores que pueden tomar los tipos de datos, según las especificaciones del estándar ANSI/ISO de C++.

Tabla 5.11. Rango mínimo de valores de los tipos de datos con modificador unsigned.

Tipo	Rango de valores
unsigned char	de 0 a 255
unsigned int	de 0 a 65535
unsigned short int	de 0 a 65535
unsigned long int	de 0 a 4294967295

También disponemos del modificador `signed`, el cual se utiliza igual que `unsigned`, y permite obtener para los tipos de datos el rango de valores con signo de la tabla 5.6 y 5.10, pero que por estar aplicado por defecto no suele utilizarse.

Tamaño de los tipos de datos

Los tamaños de algunos tipos dependen de la máquina y del compilador. Si la máquina es de 16 bits, el tipo `int` tendrá este tamaño, pero si es de 32 bits, tendrá este otro tamaño.
En la tabla 5.12 podemos ver los tamaños típicos de los tipos de datos en máquinas de 16 bits y de 32 bits.

Tabla 5.12. Tamaño de tipos de datos en distintas máquinas.

Tipo	Máquina de 16 bits	Máquina de 32 bits
char	8 bits	8 bits
int	16 bits	32 bits
short int	16 bits	16 bits
long int	32 bits	32 bits
unsigned char	8 bits	8 bits
unsigned int	16 bits	32 bits
unsigned short int	16 bits	16 bits
unsigned long int	32 bits	32 bits
float	32 bits	32 bits
double	32 bits	32 bits
long double	64 bits	64 bits

Declaración de variables y constantes

Declaración de variables

En C y C++, antes de usar una variable hay que declararla. Una variable se declara escribiendo su tipo de dato, un espacio, su identificador y punto y coma (;):

```
tipo identificador;
```

Imaginemos que queremos usar una variable para almacenar una edad, otra para almacenar distancias y otra para almacenar letras. Estas variables declaradas en una función `main` aparecerían de la siguiente manera.

```
int main (void)
{
   char letra;
   int edad;
   float distancia;
}
```

Pero, ¿qué ocurriría si, por ejemplo, necesitásemos una variable para almacenar el número de peras que hay en una tienda de frutas, otra variable

el número de manzanas, otra para el número de plátanos y otra para el número de naranjas? La declaración de todas estas variables sería:

```
int main (void)
{
   int peras;
   int manzanas;
   int plátanos;
   int naranjas;
}
```

Hay cuatro variables declaradas del mismo tipo en cuatro líneas. Para estos casos en los que queremos declarar más de una variable del mismo tipo está la posibilidad de hacerlo en una línea. Para ello escribimos el tipo de dato de las variables, un espacio, el identificador de una variable, una coma, un espacio, el identificador de otra variable, una coma, y así sucesivamente. Tras el último identificador debe escribirse un punto y coma (;). Veamos la sintaxis:

```
tipo identificador_1, identificador_2, ..., identificador_n;
```

El ejemplo anterior quedaría ahora:

```
int main (void)
{
   int peras, manzanas, plátanos, naranjas;
}
```

Utilizar un sistema u otro depende del estilo del programador, pero también influyen otros aspectos, como los comentarios. Añadir comentarios junto a la declaración de cada variable, para el primer ejemplo, aclara mucho el código, como puede observarse a continuación.

```
int main (void)
{
   int peras;          /* Numero de  peras que hay en la  tienda. */
   int manzanas;       /* Numero de  manzanasquehayen la tienda. */
   int plátanos;       /* Numero de  plátanosquehayen la tienda. */
   int naranjas;       /* Numero de  naranjasquehayen la tienda. */

}
```

Pero también se podrían añadir comentarios de una forma elegante a la declaración de variables en una línea:

```
int main (void)
{
  /* Las siguientes variables almacenan el número
  de frutas correspondientes que hay en la tienda */
  int peras, manzanas, plátanos, naranjas;
}
```

Así que todo dependerá del caso, pero el principio de que el código debe escribirse claro nos marcará el camino.

Declaración de constantes

Las constantes las creamos cuando vamos a utilizar un determinado valor repetidas veces y, además, sabemos que nunca cambiará. Por ejemplo, si creamos un programa de matemáticas donde utilizamos el valor del número Pi (3.141592653) multitud de veces, nos será más fácil crear una constante llamada PI a la que asignamos su valor y, a partir de ese momento, usar siempre el identificador PI en lugar de la tira de número a los que equivale. Obviamente, es más fácil recordar la palabra `PI` que el número 3.141592653. Podremos crear constantes mediante la palabra clave const y mediante la directiva #define.
Recuerde que, por estilo a la hora de programar, los identificadores de las constantes se escriben en mayúsculas, mientras que los identificadores de las variables se escriben en minúsculas.

#define

El uso de la directiva #define nos permite crear constantes, aunque su función va más allá. Para ello, escribimos #define, un espacio, el identificador de la constante, un espacio y el valor de la constante:

```
#define identificador valor
```

Pero, ¿qué tipo de valores podemos dar a estas constantes? Bien, a las constantes que creemos con # de fine les podemos dar el valor que queramos, ya sea de tipo float, int o char, lo que ocurre es que no se especifica en ningún momento.

```
#define PI 3.14
#define LADO 10
#define PRIMERA_VOCAL 'a'
```

También podemos asignar cadenas a las constantes creadas con #def ine. Si un carácter se representaba entre comillas simples y sólo podía contener un carácter (por ejemplo: 'A'), una cadena se representa entre comillas dobles y puede contener uno o más caracteres (por ejemplo: "Esto es una cadena"). Constantes cuyo valor es una cadena son:

```
#define MENSAJE_SALUDO "Hola"
#define DESPEDIDA "Hasta la próxima"
#define CODIGO "10ATO89P"
#define NOTA_1 "DO"
```

Las constantes declaradas mediante #define se suelen escribir antes de la función main y después de las directivas #include:

```
/* Aqui irían las directivas #include */
#define PI 3.14
#define LADO 10
#define PRIMERA_VOCAL 'a'
int main (void)
{
   /* Aqui iria el codigo del programa
   que utilizase las constantes
   declaradas mediante #define. */
}
```

Observe que no se escribe el punto y coma (;) al final de cada declaración, como ocurría con la palabra reservada `const`, ya que si lo hiciéramos provocaríamos errores en el programa. Esto es porque con `#define` el valor de la constante será todo aquello que se escriba tras el identificador, por lo que si ponemos un punto y coma tras el valor:

```
#define PI 3.14;
```

entonces, el valor de la constante PI incluirá también el punto y coma, siendo: `3.14;`
Pero ¿por qué ocurre esto? La solución está en cómo funciona `#define`. Supongamos que hemos escrito un programa como el siguiente, en el cual aparece en varios lugares del código la constante `SECRETO`. No importa que ahora no entienda el programa, únicamente localice los lugares donde aparece el identificador `SECRETO`.

```
#include <stdio.h>
#define SECRETO 3
int main (void)
{
   int numero;
   printf ("Escriba el numero secreto: ");
   scanf ("%d", &numero);
   fflush(stdin);
   if (numero == SECRETO)
      printf ("\nMuy bien, lo has adivinado.");
      else
      printf ("\nLo siento, no lo has adivinado. Era: %d", SECRETO);

   /* Hacemos una pausa hasta que el usuario pulse Intro */
   getchar();
}
```

Cuando compilemos este programa, el compilador detectará la declaración de una constante llamada `SECRETO` mediante la directiva `#define` y cuyo

valor es 3. A partir de la línea siguiente, cada vez que se encuentre el identificador `SECRETO` lo sustituirá por su valor.
A la vista del compilador, no a la nuestra, el programa tomaría el siguiente aspecto.

```
#include

<stdio.h>

#define SECRETO

{

int main (void)
{
   int numero;

   printf ("Escriba el número secreto: ");
   scanf ("%d", &numero);
   fflush(stdin);

   if (numero == 3)
      printf ("\nMuy bien, lo has adivinado.");
      else
      printf ("\nLo siento, no lo has adivinado. Era: %d", 3);

   /* Hacemos una pausa hasta que el usuario pulse Intro */
   getchar () ;
}
```

Observe que a partir de `#define SECRETO 3`, donde había un identificador `SECRETO` ahora aparece su valor.
Si en la declaración, después del valor de la constante, hubiéramos escrito un punto y coma (;), el programa, sólo a la vista del compilador, habría quedado así:

```
#include <stdio.h>
#define SECRETO 3;
int main (void)
{
   int numero;
   printf ("Escriba el número secreto: ");
   scanf ("%d",&numero);
   fflush(stdin);

   if (numero == 3;)
      printf ("\nMuy bien, lo has adivinado.");

   else
      printf ("\nLo siento, no lo has adivinado. Era: %d", 3;);

   /* Hacemos una pausa hasta que el usuario pulse Intro */
   getchar ();
}
```

Ahora, a los valores de **SECRETO** dentro de la función **main** los acompaña un punto y coma (;). Esto ocasionará errores cuando compilemos el programa, porque aparecen puntos y comas en lugares incorrectos, no pudiendo ejecutarse. En resumen, el código está mal escrito.

Advertencia:

En la declaración de una constante mediante la directiva `#define` *no se escribe punto y coma (;) tras el valor de ésta.*

const

La palabra reservarda `const` nos permite crear auténticas constantes con su propio tipo de dato. Para ello escribimos `const`, un espacio, el tipo de dato de la variable, un espacio, el identificador de la constante, un signo igual, el valor de la constante y punto y coma (;). Su sintaxis es:

```
const tipo identificador = valor;
```

Ejemplos de declaraciones de constantes son:

```
conts float PI = 3.14;       /*Constantedel número pi. */
conts float LADO = 10;       /*Constantedel lado de un cubo.*/
const char VOCAL_1 = 'a';    /*Constantede primera vocal. */
```

A las constantes se les asigna un valor inicial en el momento de la declaración, porque, al ser constantes, su valor será siempre el mismo desde el momento en que se creen.

Las constantes declaradas mediante `const` se suelen escribir después de las directivas `#include` y de las declaraciones con `#define` y antes de la función main:

```
#include <stdio.h>

#define SECRETO 3

const int PREMIO = 6;

int main (void)
{
    int numero;
    print*\."Escriba el número secreto: ");

    scanf ("%d', &numero) ;
    fflush(stdin);

    if (numero == SECRETO)
       printf ("\nMuy bien. Has ganado %d puntos.", PREMIO);
```

```
    else
        printf ("\nLo siento, no lo has adivinado. Era: %d", SECRETO);

    /* Hacemos una pausa hasta que el usuario pulse Intro */
    getchar();
}
```

Inicialización de variables

Cuando declaramos una variable, ésta no tiene ningún valor definido. Si utilizamos esa variable, sin asignarle ningún valor inicial, podría no funcionar correctamente nuestro programa. Para evitarlo, podemos inicializar las variables en el mismo momento de declararlas. La sintaxis es:

```
tipo identificador = valor;
```

En cualquier caso, no es obligatorio inicializar las variables.
Ejemplos de variables declaradas e inicializadas en una función main son:

```
int main (void)
{
    int mi_numero = 10; char mi_letra = ' x'; float pi = 3.1415;
}
```

De esta manera, la variable mi_numero almacena el valor 10 desde el momento en que se crea, la variable mi_letra tendrá el valor 'x' y la variable pi tendrá el valor 3.1415.

Ejercicios resueltos

Intente realizar los siguientes ejercicios. Encontrará la solución al final de este libro.

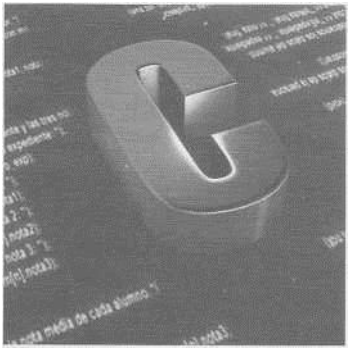
1. Supongamos que necesitamos dos variables para almacenar la posición del ratón en la coordenada X y en la coordenada Y de la pantalla. ¿Qué identificadores emplearía para cada una de estas variables?
2. Cuáles de los siguientes identificadores son incorrectos:
   - num_camiones
   - 2_registro
   - _asm
   - _código

- bandera7
- num_dia_semana
- #codigo
- año
- _pulsacion
- cod soft

1. Si tenemos una variable cuyo identificador es "numero_reg" y su valor inicial es 10, ¿cuál será el valor de la variable "numero_Reg"?
2. Cómo se declararía:
   - Una variable que permita almacenar caracteres.
   - Una constante cuyo valor sea tu nombre.
   - Una variable que permita almacenar la edad de una persona.
   - Una variable que permita almacenar un código numérico de 7 dígitos.
   - Una constante cuyo valor es el doble del número pi.

Resumen

Hemos aprendido que las variables y constantes existen también en programación, además de en materias como las matemáticas o la física. Así que hemos aprendido qué tipos de datos pueden almacenar, qué nombre (identificador) les podemos asignar, cómo se declaran y cómo se inicializan. Ahora ya estamos listos para estudiar cómo se pueden utilizar junto con operadores aritméticos, lógicos...



Capítulo 6

Operadores

En este capítulo aprenderá:

- Qué operadores posee C/ C++.
- Cómo utilizar los distintos operadores.

Introducción

Al igual que en matemáticas, en nuestros programas también podemos realizar operaciones con variables, constantes u otros valores como: sumar, restar, multiplicar o dividir; para lo cual empleamos operadores. Recordemos que esas variables, constantes u otros valores que utilizamos en una operación reciben el nombre de operandos. En la figura 6.1 podemos ver una expresión con operandos y operador.

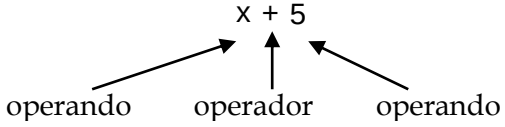


Figura 6.1. Operandos y operador en una expresión.

Una de las ventajas que tiene el lenguaje C/C++ es el gran número de operadores que posee: operadores de bits, operadores aritméticos, operadores de incremento y decremento, operadores de asignación y asignación compuesta, operadores relaciónales y lógicos.
Veremos programas de ejemplo en los que se hace uso de estos operadores. Alguno de estos programas tiene como única finalidad mostrar cómo y dónde escribir lo aprendido dentro de un programa.

Operadores aritméticos

La mayor parte de los operadores aritméticos son los comúnmente empleados en las operaciones matemáticas: la suma (+), la resta (-), la multiplicación (*) y la división (/), que devuelve la parte entera de la división de dos números. Además, encontramos otro operador, el módulo o resto (%), el cual no devuelve el resto de una división entre dos números enteros. En la tabla 6 vemos estos operadores, su significado y un ejemplo.

Tabla 6.1. Operadores aritméticos.

Operador	Nombre	Ejemplo
+	suma	a + b
-	resta	a - b

Operador	Nombre	Ejemplo
*	multiplicación	a * b
/	división	a / b
%	módulo o resto	a % b

Supongamos que tenemos dos variables: x e y, cuyos valores son 7 y 3, respectivamente. Analicemos ahora algunos ejemplos de estas dos variables y los operadores aritméticos.
Si sumamos x e y obtendremos 10: x+y es lo mismo que 7 + 3, que tiene por resultado 10.
Si a x le restamos y obtendremos 4: x-y es lo mismo que 7-3, que tiene por resultado 4.
Si multiplicamos x e y obtendremos 21: x*y es lo mismo que 7 por 3, que tiene por resultado 21.
Si dividimos x entre y obtendremos 2: x/y es lo mismo que 7 entre 3, que tiene por resultado 2 (como parte entera).
Si calculamos el resto o módulo de x entre y obtendremos 1, como se muestra en la figura 6.2.

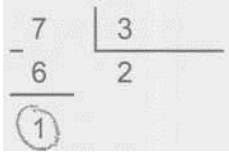


Figura 6.2. Resto de la división entera de 7 entre 3.

Luego x%y tiene por resultado 1.
El programa, que se muestra a continuación, no tiene ninguna finalidad concreta, simplemente es un ejemplo de cómo pueden escribirse las distintas operaciones aritméticas.

```
int main (void)
{
   int v, w, x, y, z;
   const int k = 10;


   v   = k+ 1;      /*  Suma de una constante y un valor. */
   w   = v* 2;      /*  Multiplicación de una variable y un valor. */
   x   = w% v;      /*  Modulo o resto de una variable entre otravariable. */
   y   = w/ k;      /*  División entera de una variable entre unaconstante. */
   z = 30 - 1;    /* Resta de dos valores. */
}
```

El símbolo de suma (+) y resta (-), además de emplearse como operadores, pueden emplearse como signos positivo y negativo, respectivamente. De esta forma, y siguiendo el ejemplo anterior de x e y, la expresión -x representaría a -7, la expresión +y representaría +3, la expresión - (-x) representaría -(-3), es decir, +3.
El siguiente programa es un ejemplo en el que aparece el uso del símbolo más (+) y menos (-) como signo de un número.

```
int main (void)
{
   int x, y;
   const int k = 10;

   x = -k; y = -x * 2;
}
```

En la primera asignación `x` recibirá el valor -10, mientras que en la segunda asignación y recibirá el valor resultante de la expresión -(-10) * 2, es decir, el valor 20.

Operadores de asignación

El operador de asignación se representa con el símbolo igual (=) y sirve para dar el resultado de una expresión a una variable. La sintaxis es:

```
variable = expresión;
```

En una asignación primero se resuelve la expresión situada a la derecha del símbolo igual (=) y el resultado se le asigna a la variable situada a su izquierda.
Un ejemplo de asignación es:

```
x = 6 + 4;
```

Donde primero se resuelve la expresión 6 + 4, cuyo resultado (10) se le asigna o da a la variable x. Ahora la variable x vale o almacena el valor 10. Estos pasos quedan representados en la figura 6.3.

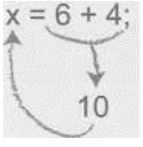


Figura 6.3. Asignación.

Es importante señalar que cuando se realiza una asignación a una variable, el valor que tuviese se pierde y se reemplaza por el nuevo valor de la asignación. Por eso es una variable, porque su valor puede variar.
El siguiente programa muestra ejemplos de asignaciones.

```
int main (void)
{
   int x, y;

   x = 0;          /* A x le damos el valor cero, asi que x vale cero. */
   x = 2;          /*  Ahora x vale 2,  y no 0. */
   y =  x + 1;     /*  A  y le damos el valor resultante  de (x+l),que es 3. */
   y =  y + 1;     /*  A  y le damos el  valor resultante  de (y+1), que es 4.*/
   y =  y + 1;     /*  A  y le damos el  valor resultante  de (y+1), que es 5.*/
}
```

Hagamos una traza para ver cómo modifican esas asignaciones el valor de x e y.
En la primera asignación (x=0;) se da a x el valor 0 (véase la figura 6.4).

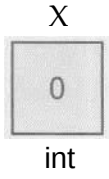


Figura 6.4. Estado de la variable x tras la primera asignación.

En la segunda asignación (x=2;) se da a x el valor 2, perdiéndose el valor que tuviera anteriormente y sustituyéndose por el nuevo, x ya no vale 0, vale 2 (véase la figura 6.5).

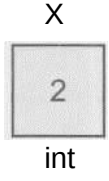


Figura 6.5. Estado de la variable x tras la segunda asignación.

En la tercera asignación (y=x+l;) se da a y el resultado de la expresión x+1, que equivale a 2 + 1, ya que como vemos en la última representación gráfica de la variable x, ésta vale 2.
El resultado de la expresión, 3, es el que se da a la variable y (véase la figura 6.6).

Figura 6.6. Estado de la variable y tras la tercera asignación.

En la cuarta asignación (y=y+l;) se asigna a `y` el valor de la expresión `y+1`. Como ya hemos dicho, en primer lugar se resuelve la expresión situada a la derecha de la asignación, `y+1`, y cuyo resultado es 4, al ser 3 el valor de `y`. Ese resultado 4 es el que se asigna a la variable situada a la izquierda de la asignación: `y` (véase la figura 6.7).

Figura 6.7. Estado de ia variable y tras la cuarta asignación.

La quinta y última asignación (y=y+l;) es igual que la anterior. Ahora y valdrá 5.
Las expresiones del tipo y=y+l sirven para incrementar el valor de una variable en una unidad. Esto es porque a una variable se le asigna ella misma más una unidad (véase figura 6.8).

Figura 6.8. Estado de la variable y tras la quinta asignación.

Y ¿qué valor almacenaría la variable de este programa tras cada asignación?

```
int main (void)
{
   char letra;

   letra = 'ñ';            /* Ahora letra almacena el carácter 'A'. */
   letra = letra + 1;      /* Ahora letra almacena el carácter 'B'. */
```

```
    letra = letra + 1;      /* Ahora letra almacena el carácter 'C'. */
    letra = letra + 1;      /* Ahora letra almacena el carácter 'D'. */
}
```

El carácter que almacena inicialmente la variable letra es 'A', gracias a la asignación:

```
letra = 1 A';
```

A continuación se produce un incremento del valor de esta variable mediante la asignación:

```
letra = letra + 1;
```

El incremento del valor de una variable que almacena un carácter tiene como resultado que la variable toma el valor del siguiente carácter de la tabla ASCII.
Así, al incrementar el valor de la variable letra ('A') en una unidad obtenemos el siguiente carácter de la tabla ASCII: 'B'.
Si repetimos esta operación obtendríamos el siguiente carácter, 'C', y así sucesivamente.
Por otro lado, en algunas ocasiones tendremos una gran cantidad de variables a las que asignar un mismo valor y en un mismo momento, por ejemplo:

```
int main (void)
{
    int a, b, x, y;

    a = 0 ;
    b = 0;
    x = 0;
    y = 0;
}
```

Para estos casos el lenguaje C nos permite usar el operador de asignación para realizar varias asignaciones en una misma línea tal como se muestra a continuación:

```
int main (void)
{
    int a, b, x, y;

    a = b = x = y= 0;
}
```

Ahora todas esas variables (a, b, x, y) valen 0.
Bien, ya podemos realizar algún programa simple. Supongamos que un profesor tiene diez alumnos y doce caramelos. Tiene que repartir los caramelos a los alumnos, pero no puede partir los caramelos, los tiene que dar enteros. ¿Cuántos le sobran?

Está claro que le sobran dos caramelos, pero lo que tendremos que hacer es calcular el resto de dividir los doce caramelos entre los diez alumnos. Sin entrar a discutir si alumnos y caramelos son variables o constantes, la parte principal de un programa que resuelve este problema podría ser:

```
int main (void)
{
    int alumnos = 10; int caramelos = 12;

    int  caramelossobran;

    caramelos_sobran = caramelos % alumnos;
}
```

Operadores de asignación compuestos

Hay casos en los que emplearemos expresiones como las siguientes para cambiar el valor de una variable:

```
/* Para incrementar el valor de resultado_1 en 2 unidades. */
resultado_1 = resultado_1 + 2;

/* Para dividir el valor de resultado_2 entre 10. */
resultado_2 = resultado_2 / 10;
```

El lenguaje C incluye una serie de operadores de asignación compuestos que nos harán la programación más cómoda, ya que permiten abreviar expresiones como las anteriores de la siguiente manera, y siendo equivalentes:

```
/* Para incrementar el valor de resultado_1 en 2 unidades. */
resultado_1 += 2;

/* Para dividir el valor de resultado_2 entre 10. */
resultado_2 /= 10;
```

Tal y como se puede observar, cuando la sintaxis de una expresión es de este tipo:

```
variable = variable operador expresión ;
```

donde la variable situada a ambos lados del símbolo de asignación (=) es la misma, el lenguaje C permite abreviar la expresión anteponiendo el operador al símbolo igual y eliminando el nombre de la variable situada tras el operador de asignación, quedando:

```
variable operador = expresión;
```

En la tabla 6.2 observamos la sintaxis básica de los operadores compuestos (aritméticos y de bits) y su forma equivalente.

Tabla 6.2. Operadores compuestos.

Forma compuesta	Equivalente
a += b; a -= b;	a = a + b; a = a - b;
a *= b;	a = a * b;
a /= b;	a = a / b;
a %= b; a «= b;	a = a % b; a = a « b;
a »= b;	a = a » b;
a &= b;	a = a & b;
a |= b;	a = a | b;
a ^= b;	a = a ^ b;

Otros ejemplos son:

```
/* Para decrementar el valor de resultado_3 en 7 unidades. */
resultado_3 -= 7;

/* Para multiplicar el valor de resultado_4 por 5. */
resultado_4 *= 5;
```

Siendo sus equivalentes:

```
/* Para decrementar el valor de resultado_3 en 7 unidades. */
resultado_3 = resultado_3 - 7;

/* Para multiplicar el valor de resultado_4 por 5. */
resultado 4 = resultado 4 * 5 ;
```

Operadores de incremento y decremento

El lenguaje C dedica operadores a una operación tan común como es la de incrementar y decrementar el valor de una variable en una unidad. Los operadores para el incremento son dos signos más seguidos, sin espacios, (++); para el decremento son dos signos menos seguidos, sin espacios, (—). Estos operadores pueden ir delante o detrás de la variable.
Así, en vez de escribir:

```
resultado = resultado + 1;
```

podemos escribir:

```
resultado++;
```

o:

```
++resultado;
```

Y en vez de escribir:

```
resultado = resultado - 1;
```

podemos escribir:

```
resultado--;
```

o:

```
--resultado;
```

Como podemos observar, las sintaxis son:

```
variable++;
++variable;
variable--;
--variable;
```

Por ejemplo, si tenemos una variable llamada edad, cuyo valor inicial es 10, y utilizamos un operador de incremento en esta variable, su valor es posteriormente 11. Su valor se habrá incrementado en una unidad. El código de este supuesto lo encontramos a continuación.

```
int main (void)
{

    int edad = 10;    /* edad vale inicialmente 10. */

    edad++;           /* Ahora edad valdrá 11. */
}
```

Y para decrementar en una unidad el valor de la variable edad tendríamos que haber empleado el operador de decremento como se muestra ahora:

```
int main (void)
{

    int edad = 10;    /* edad vale inicialmente 10. */

    edad++;           /* Ahora edad valdrá 9. */
}
```

Pero no es lo mismo poner los operadores de incremento y decremento delante o detrás de la variable.

Si el operador va delante (++variable; o —variable;), se realizará primero el incremento o decremento y después se utilizará el valor de la variable.

Si el operador va detrás (variable++; o variable—;), se usará el valor de la variable en primer lugar y después se realizará el incremento o decremento.

Veamos un ejemplo comentado de todo esto. Supongamos que declaramos dos variables de tipo entero (a y b) y las inicializamos con los valores 2 y 4, respectivamente:

```
int a = 2;
int b = 4;
```

Y a continuación escribimos:

```
a = ++b;
```

Esta última línea de código emplea un operador de incremento delante de la variable b, por lo que primero se realiza el incremento del valor de esta variable (tomando b el valor 5) y después se utiliza su valor (5) en la asignación (tomando a el valor 5).
Pero si hubiéramos escrito (en lugar de a=++b;):

```
a = b++;
```

por estar el operador de incremento después de la variable, primero se asignaría el valor de b, tomando a el valor 4. Después, se incrementaría b, tomando b el valor 5.
Y si hubiéramos escrito (en lugar de a=++b;):

```
a = --b;
```

primero se decrementaría el valor de b (tomando b el valor 3) y después se asignaría el valor de b a la variable a (tomando a el valor 3).
Y si hubiéramos escrito (en lugar de a=++b;):

```
a = b--;
```

primero se asignaría el valor de b a la variable a (tomando a el valor 4) y después se decrementaría el valor de b (tomando b el valor 3).
Estos casos sólo ocurren cuando utilizamos los incrementos (++) o los decrementos (--) en una línea de código en la que estamos haciendo más operaciones a la vez, como asignaciones.
Sin embargo, cuando usemos estos operadores solos en una línea no será necesario tener en cuenta estas consideraciones, como por ejemplo en el siguiente listado.

```
int main (void)
{
    int a = 0;     /* Ahora "a" vale 0 (inicialmente). */

    a++;           /*  Ahora "a"  vale 1 (traselincremento).   */
    ++a;           /*  Ahora "a"  vale 2 (traselincremento).   */
    a--;           /*  Ahora "a"  vale 1 (traseldecremento).   */
    --a;           /*  Ahora "a"  vale 0 (traseldecremento).    */
}
```

Operadores de bits

Los operadores de bits sólo pueden operar sobre tipos de datos char e int (enteros decimales, hexadécimales y octales). Los operadores de bits realizan operaciones sobre cada uno de los bits de un entero o char. Estos operadores son:

- AND, representado por el símbolo &.
- OR, representado por el símbolo I.
- XOR, representado por el símbolo ^.
- Complemento a uno, representado por el símbolo ~.
- Desplazamiento a la izquierda, está representado por dos signos menor que («).
- Desplazamiento a la derecha, está representado por dos signos mayor que (»).

La operación AND (&) compara bit a bit obteniendo un resultado según la tabla 6.3.

Tabla 6.3. Operación AND bit a bit.

Bit 0	Bit 1	Resultado
0	0	0
0	1	0
1	0	0
1	1	1

Así, si realizamos la operación 107 (01101011 en binario) AND 27 (00011011 en binario) se obtendría 11 (00001011 en binario).
La operación OR (|) compara bit a bit obteniendo un resultado según la tabla 6.4.

Tabla 6.4. Operación OR bit a bit.

Bit 0	Bit 1	Resultado
0	0	0
0	1	1

Bit 0	Bit 1	Resultado
1	0	1
1	1	1

Así, si realizamos la operación 107 (01101011 en binario) OR 27 (00011011 en binario) se obtendría 123 (01111011 en binario).
La operación XOR (^) compara bit a bit obteniendo un resultado según la tabla 6.5.

Tabla 6.5. Operación XOR bit a bit.

Bit 0	Bit 1	Resultado
0	0	0
0	1	1
1	0	1
1	1	0

Así, si realizamos la operación 107 (01101011 en binario) XOR 27 (00011011 en binario) se obtendría 112 (01110000 en binario).
El operador complemento a uno (~) simplemente invierte cada uno de los bits.
Así, ~ 01101011 es igual a 10010100 y ~ 00011011 es igual a 11100100.
El operador de desplazamiento a la izquierda («) desplaza los bits hacia la izquierda tantas posiciones como se indique tras el operador, colocándose en el bit menos significativo (bit más a la derecha) un cero, y el bit más significativo (bit más a la izquierda) se desecha.
Veamos un ejemplo en el que se aplica a una variable un desplazamiento a la izquierda de un bit:

```
int main (void)
{
    int izquierda = 1;
    izquierda = izquierda << 1;
}
```

En la inicialización, la variable izquierda vale 1, valor que en binario es 00000001.
Pero ¿qué sucede en la operación de desplazamiento a la izquierda? Lo podemos observar en la figura 6.9.

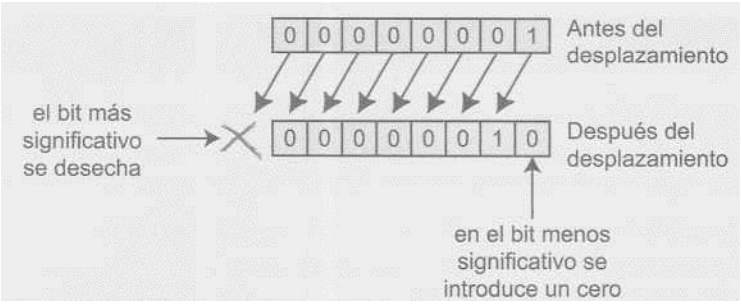


Figura 6.9. Desplazamiento a la izquierda de un número binario.

Como puede verse, el resultado del desplazamiento es 00000010 en binario, cuyo valor equivalente en decimal es 2. A continuación, este valor se le asigna a la variable izquierda.

El operador de desplazamiento a la derecha (») desplaza los bits hacia la derecha tantas posiciones como se indique tras el operador, colocándose en el bit más significativo un cero, y el bit menos significativo se desecha. Veamos ahora un ejemplo de desplazamiento a la derecha de un bit:

```
int main (void)
{
   int derecha = 1;
   derecha = derecha >> 1;
}
```

En la inicialización, la variable derecha vale 1, que en binario es 00000001. Observemos la figura 6.10 para poder comprobar cómo se realiza el desplazamiento.

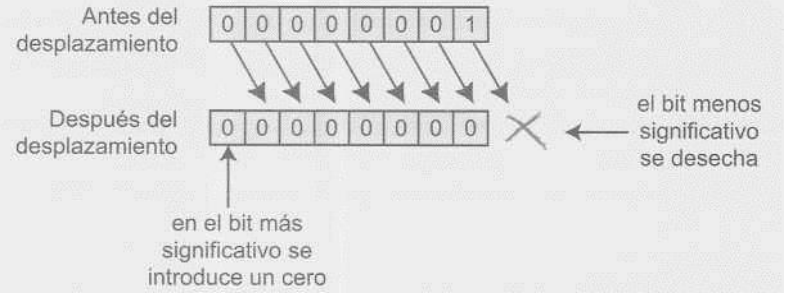


Figura 6.10. Desplazamiento a la derecha de un número binario.

Como puede verse, el resultado del desplazamiento es 00000000 en binario, cuyo valor equivalente en decimal es 0. A continuación, este valor se le asigna a la variable derecha.

Operadores relaciónales

Los operadores relaciónales se utilizan para evaluar una relación entre dos valores. La relación puede tener como resultado el valor verdadero o falso. Es decir, mediante los operadores relaciónales hacemos comparaciones entre dos valores. Estos se muestran en la tabla 6.6.

Tabla 6.6. Operadores relaciónales.

Operadores relaciónales	Significado
==	igual
!=	Distinto
>	Mayor
<	Menor
> =	Mayor o igual
< =	Menor o igual

En la tabla 6.7 encontramos algunos ejemplos simples, partiendo de que m vale 5 y n vale 8.

Tabla 6.7. Ejemplos de operadores relaciónales.

Operadores relaciónales	Se leería	Resultado
m == n	¿m es igual a n?	Falso
m != n	¿m es distinto de n?	Verdadero
m > n	¿m es mayor que n?	Falso
m < n	¿m es menor que n?	Verdadero
m >= n	¿m es mayor o igual que n?	Falso
m <= n	¿m es menor o igual a n?	Verdadero

Operadores lógicos

Los operadores lógicos evalúan de forma lógica dos valores, excepto el operador NOT, que invierte el valor lógico.

La tabla 6.8 muestra los operadores lógicos.

Tabla 6.8. Operadores lógicos.

Operadores lógicos	Significado
&&	AND
||	OR
!	NOT

El resultado de las expresiones con operadores lógicos obedece a las "tablas de verdad". De esta manera, podemos obtener el resultado de una expresión con el operador AND (&&) del tipo:

```
ValorA && ValorB
```

en la tabla 6.9.

Tabla 6.9. Tabla de verdad del operador AND (&&).

ValorA	Valor B	Resultado
Falso	Falso	Falso
Falso	Verdadero	Falso
Verdadero	Falso	Falso
Verdadero	Verdadero	Verdadero

Para el caso de las expresiones con el operador OR (| |) del tipo:

```
ValorA || ValorB
```

el resultado se obtiene según la tabla 6.10.

Tabla 6.10. Tabla de verdad del operador OR (II).

ValorA	Valor B	Resultado
Falso	Falso	Falso
Falso	Verdadero	Verdadero
Verdadero	Falso	Verdadero
Verdadero	Verdadero	Verdadero

La expresiones con el operador NOT (!) son de la forma:

```
! ValorA
```

y su resultado se obtiene según la tabla 6.11.

Tabla 6.11. Tabla de verdad del operador NOT (!).

Valor A	Resultado
Falso	Verdadero
Verdadero	Falso

Advertencia:

Es importante saber que, en C/C++, se considera falso al valor cero y verdadero a cualquier valor distinto de cero, por muy próximo que se encuentre a éste y ya sea positivo o negativo, entero o real. Por lo que incluso el valor 0.00001 es tomado como verdadero y sólo el valor cero es tomado como falso.

Veamos en la tabla 6.12 algunos ejemplos.

Tabla 6.12. Ejemplos con operadores lógicos.

Operadores lógicos	Equivalente	Resultado
27 &&-15	¿verdadero AND verdadero?	verdadero
0 && 0	¿falso AND falso?	falso
1 && 0.0001	¿verdadero AND verdadero?	verdadero
-2 || 100	¿verdadero OR verdadero?	verdadero
0 || 0	¿falso OR falso?	falso
! 0	negación de falso	verdadero
! 10	negación de verdadero	falso

Podemos también realizar otros ejemplos mezclando operadores lógicos con relaciónales:

```
(5>3) && (10<=30)
```

Esta expresión se resolvería comenzando por los paréntesis.

¿5 es mayor que 3? No, luego el resultado del primer paréntesis es falso. ¿10 es menor o igual que 30? Sí, luego el resultado del segundo paréntesis es ver dadero. Finalmente se calcularía la expresión del resultado obtenido de cada paréntesis con el operador AND (&&):

```
Falso & & Verdadero
```

que, comprobando la tabla de verdad del operador AND, el resultado sería Falso.

Orden de ejecución de los operadores

Los paréntesis permiten alterar el orden de prioridad en la resolución de una operación, resolviéndose en primer lugar el contenido de éste, pero sobre todo permiten ver claramente dicho orden de un solo vistazo. Por ejemplo, en el caso siguiente tendríamos que pararnos a analizar dicho orden:

```
x + y * 2 % 9 - z / 2
```

Mientras que en la siguiente expresión, haciendo uso de los paréntesis, veríamos el orden rápidamente.

```
( (6 * y) % 2) - ( (x/z) + 7 )
```

Así pues, ya hemos comprobado que el uso de paréntesis puede evitarnos más de un dolor de cabeza. En cualquier caso el orden de ejecución de los operadores se muestra en la tabla 6.13, donde los operadores tienen mayor prioridad de ejecución cuanto más arriba se encuentran en la tabla.

Tabla 6.13. Orden de ejecución de los operadores.

Operador	Significado	Prioridad
+ +	Incremento	Mayor
-	Decremento	
-	Signo negativo	
*	Multiplicación	
/	División	
%	Módulo o resto	
+	Suma	
-	Resta	Menor

Ejercicios resueltos

Intente realizar los siguientes ejercicios. Encontrará la solución al final de este libro.

1. Indique si son equivalentes o no las parejas de expresiones que aparecen en la tabla 6.14.

Tabla 6.14. ¿Expresiones equivalentes?

Expresión A	Expresión B	¿Son equivalentes?
++i;	i += 1;	Sí /No
b = 1 + b;	++b;	Sí /No
c = c + 1;	++c;	Sí /No
k++;	k = k -1 ;	Sí /No

2. ¿Cuál sería el valor de la variable contador después de ejecutarse el código siguiente?

```
int contador = 0;
contador++;
contador = contador + 1;
contador  =  1  + contador;
contador--;
contador  = 7 - contador;
contador--;
contador = contador - 4;
```

3. ¿Cuál sería el resultado de la siguiente expresión?

```
! ( (7<=7)  ||  (33 > 33) )
```

Resumen

Ya conocemos los operadores de C/C++. Poco a poco vamos añadiendo conocimientos con los que poder realizar programas. Ya sabemos crear un programa con variables y/o constantes y realizar operaciones simples con ellas gracias a los operadores. Pronto llegará el momento en el que sepamos manejar las herramientas suficientes para realizar programas útiles y funcionales. Paciencia.

Capítulo 7

Punteros y referencias

En este capítulo aprenderá;

- Qué son los punteros.
- Qué son las referencias.
- Cómo se utilizan.

Introducción

El tema de los punteros y referencias es uno de los que peor fama tienen entre los programadores noveles. Ese miedo a los punteros suele ser la consecuencia de malas explicaciones, pero nada como algunas aclaraciones y unos buenos esquemas para entenderlo. Animo, que es más fácil de lo que dicen. Ya lo verá.
Para explicar este tema hablaremos de la memoria del ordenador, pero a un nivel extremadamente básico y, en ocasiones, lejos de la realidad por su simplicidad, con el único fin de comprender bien el mecanismo y la esencia de los punteros y referencias.
En cualquier caso, los punteros suponen una herramienta muy potente en C/C++, pudiendo darles usos complejos, pero ésta no es nuestra intención. En este capítulo aprenderemos sólo qué son los punteros y referencias y su manejo básico; más adelante les daremos un mayor uso.

Punteros

La memoria

Partamos de la idea de que el ordenador tiene una memoria en la que se almacenan datos y que podemos representar como una serie de casillas, las cuales podrán contener datos. En la figura 7.1 podemos ver esta representación de la memoria.

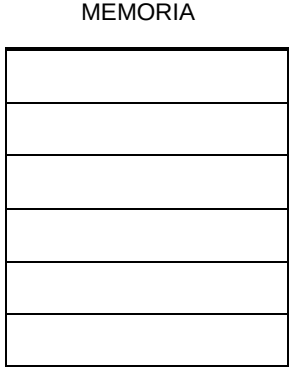


Figura 7.1. Representación de la memoria.

Pues bien, cuando en nuestros programas declaramos una variable, como por ejemplo:

```
int x;
```

ésta se almacena en una de esas casillas de la memoria, como se muestra en la figura 7.2.

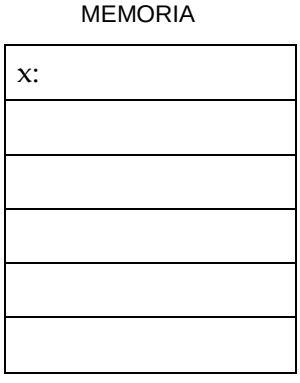


Figura 7.2. Representación de la memoria con un dato.

Si a la variable x se le asigna un valor, por ejemplo:

```
x = 10;
```

entonces, la casilla donde está la variable x en memoria almacenará el valor 10, como se muestra en la figura 7.3.

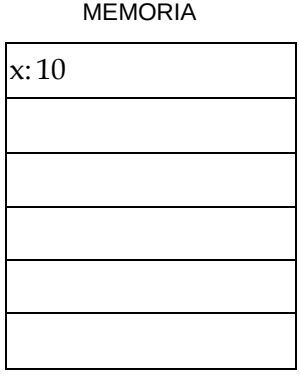


Figura 7.3. Representación de la memoria con un dato y su valor.

Si a continuación se modificase el valor de la variable x:

```
x = x + 100; //El resultado es que x vale 110.
```

el estado de la memoria quedaría como en la figura 7.4.

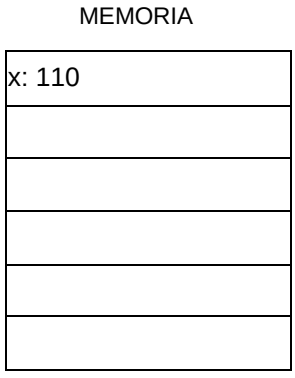


Figura 7.4. Representación de la memoria con un dato y su valor modificado.

Así que las variables y otros elementos realmente se almacenan en esa memoria, luego ¿cómo quedaría la memoria tras ejecutarse el siguiente programa si la edad del usuario fuese de 18 años y el año actual el 2548?

```
#include
<stdio.h> int
main (void)
}
   int edad, anio_actual, anio_nacimiento;

   /* Solicitamos la edad del usuario */

   printf ("Cual es su edad?: ");

   scanf ("%d", Sedad);

   fflush(stdin);

   /* Solicitamos el año actual */

   printf ("Cual es el año actual?: ");

   scanf ("%d", &anio_actual); fflush(stdin);

   /* Calculamos el año de nacimiento y lo mostramos */
   anio_nacimiento = anio_actual - edad;
   printf ("Su año de nacimiento es %d", anio_nacimiento);

   /* Hacemos una pausa hasta que el usuario pulse Intro */
   getchar();
}
```

Como en la figura 7.5.

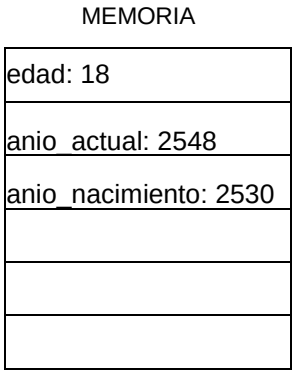


Figura 7.5. Representación de la memoria tras ejecutar el programa.

Pero se puede plantear una duda: ¿cómo sabe el ordenador dónde está cada variable dentro de la memoria? ¿Cómo las localiza cuando tiene que buscarlas para modificar su valor? El ordenador necesita saber dónde están, y el nombre de la variable o elemento no es suficiente.

Y ¿cómo podemos nosotros encontrar a un amigo en una ciudad? Fácil, las personas viven en casas y cada casa tiene una dirección. Lo único que tenemos que saber es dónde vive, cuál es su dirección.

Ésta es la solución. Cada casilla de la memoria tendrá una dirección, de manera que cuando el ordenador quiera localizar a una variable u otro elemento lo único que tiene que saber es cuál es su dirección.

Las personas tenemos direcciones como "Avenida de España, n°10" y las casillas de la memoria tienen direcciones como 8F89 o 010A:7E72. Estas direcciones (la de las casillas) no son más que simples números en formato hexadecimal. Son números muy grandes porque la memoria tiene muchas casillas. Para que nosotros podamos tener una idea más simple de todo esto utilizaremos las direcciones de memoria en formato decimal y empezando en el 1, como: 1, 2, 3... Para no confundir los números de las direcciones con cualquier otro valor les antepondremos la letra "D" de dirección, quedando: DI, D2, D3... Así evitaremos confusiones en las explicaciones y sabremos claramente cuándo nos referimos a la dirección uno (DI) o al valor uno (1). Las direcciones de cada casilla las escribiremos al lado de ésta.

Veamos entonces cómo quedaría la memoria en la figura 7.6 tras ejecutar el programa anterior.

Ahora el ordenador sabrá que la variable edad se encuentra en la dirección DI, que la variable anio actual está en la dirección D2 y que la variable anio nacimiento está en la dirección D3.

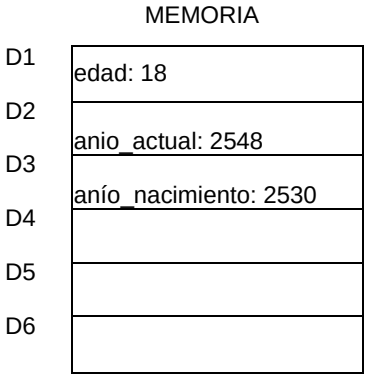


Figura 7.6. Representación de la memoria con las direcciones.

Bien, aquí terminamos la primera parte en el apartado de punteros, donde hemos aprendido que las variables y otros elementos se almacenan en unas casillas de la memoria y que cada casilla tiene una dirección. No ha sido tan difícil, ¿verdad?

Los punteros y sus operadores

Ya sabemos que existen varios tipos de datos en C/C++. El tipo char almacena caracteres (como la 'S'), el tipo int almacena números enteros (como el 6) y el tipo f loat almacena números reales (como el 3.14). Pues también existe la posibilidad de crear variables que almacenen direcciones de memoria (como la D3) y a las cuales llamamos punteros. Así que una variable puntero es una variable normal y corriente que simplemente almacena un número que se corresponde con una dirección de memoria. Debemos tener cuidado con este concepto, porque lo que almacena una variable puntero es el número de la dirección, no la dirección y su contenido.
Ya hemos visto que en una dirección de memoria, la cual corresponde a una casilla, se almacenan variables u otros elementos, que serán de un tipo de dato. En el ejemplo del programa anterior que calculaba el año de nacimiento, la variable edad se almacenaba en la dirección DI y era de tipo int. Pues cuando declaremos una variable puntero tenemos que especificar el tipo de dato que habrá en la dirección de memoria que vaya a almacenar. Pero, ¿cómo sabemos eso ahora? Un poco de paciencia.
La sintaxis para declarar una variable puntero es:

```
tipo *identificador;
```

Donde `tipo` es el tipo de dato de la variable o elemento que haya en la dirección de memoria que almacene, e `identificador` es el nombre de la variable puntero. El asterisco (*) que aparece en la declaración es necesario para crear una variable puntero.
Un ejemplo de declaración de una variable puntero sería:

```
int *mi_puntero;
```

Esta variable podrá almacenar una dirección de memoria en la cual hubiese un dato de tipo `int`. Pero, ¿qué dirección le podemos asignar a una variable puntero? Podemos asignar a una variable puntero el valor **NULL** (que equivale a 0) para indicar que no almacena ninguna dirección. Sin embargo, normalmente se asigna a una variable puntero la dirección de otra variable o elemento del mismo tipo de dato. Y ¿cómo sabemos cuál es la dirección de otra variable o elemento? Mediante el operador & (ampersand), si lo anteponemos a una variable u otro elemento se obtiene la dirección de memoria en la que se encuentra. Así, en el ejemplo del programa anterior que calculaba el año de nacimiento, si escribiésemos `Sedad` obtendríamos DI, que es la dirección de memoria donde se encontraba la variable `edad`.
Veamos qué ocurriría en el siguiente programa.

```
int main (void)
{
   int *mi_puntero;
   int edad = 18;
   mi_puntero = &edad;
}
```

La variable puntero `mi puntero` la hemos declarado de tipo `int` porque le íbamos a asignar la dirección de una variable de tipo `int: edad`, que la hemos inicializado con el valor 18. En la línea:

```
mi_puntero = &edad;
```

se obtiene mediante el operador & la dirección de la variable `edad` (D2), la cual se le asigna a la variable puntero `mi_puntero`. El estado de la memoria quedaría como en la figura 7.7.
Así que si vamos a asignar la dirección de memoria de una variable de tipo `int` a una variable puntero, ésta también será de tipo `int`. Si vamos a asignar la dirección de memoria de una variable de tipo `char` a una variable puntero, ésta también será de tipo `char`. Y así con cualquier otro tipo de dato.
Por cierto, ¿cuál sería el resultado de `&mi puntero?` Pues es la dirección de memoria en la que se encuentra la variable `mi_puntero`, que es DI. No debemos confundir la dirección en la que se encuentra la variable puntero (DI para el caso de la variable `mi_puntero)` con la dirección que almacena (D2 para el caso de la variable `mi_puntero)`.

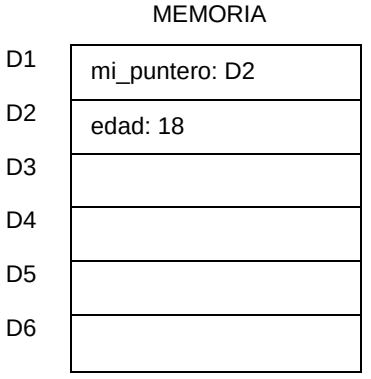


Figura 7.7. Estado de la memoria con variable puntero.

Las variables puntero también se pueden inicializar en el momento de la declaración: con la dirección de otra variable (que debe estar declarada antes que el puntero) o con el valor **NULL**. Veamos un ejemplo en el siguiente listado de código.

```
int main (void)
{
   int edad;
   int *mi_puntero1 = &edad;
   int *mi_puntero2 = NULL;
}
```

También existe otro operador: * (asterisco). Si se antepone a una variable puntero se obtiene el contenido de lo que haya en la dirección de memoria que almacena la variable puntero. Esto es, si se antepone a una dirección de memoria nos devuelve su contenido.
Luego, tomando como ejemplo el programa anterior:

- `mi_puntero` es igual a D2 (la dirección que almacena).
- `&mi_puntero` es igual a D1 (la dirección en la que se encuentra).
- `*mi_puntero` es igual a 18 (el contenido de la dirección que almacena).

En el siguiente programa se utilizan los operadores & y *.

```
int main (void)
{
   int *mi_puntero;
   int edad = 18;
   mi_puntero = &edad;

   edad = *mi_puntero + 1;
}
```

En este programa se introduce una línea nueva:

```
edad = *mi_puntero + 1;
```

En ella, `*mi_puntero` es igual al contenido que hay en la dirección que almacena la variable `mi_puntero`. Es decir, si la variable `mi_puntero` almacena la dirección D2, entonces *D2 es el contenido de la dirección D2: la variable `edad`, cuyo valor es 18. Luego escribir `*mi_puntero` es igual que escribir `edad`, siendo esa línea de código equivalente a:

```
edad = edad + 1;
```

El resultado es que el valor de la variable edad se incrementa en una unidad, tomando el valor 19.
El estado final de la memoria se muestra en la figura 7.8.

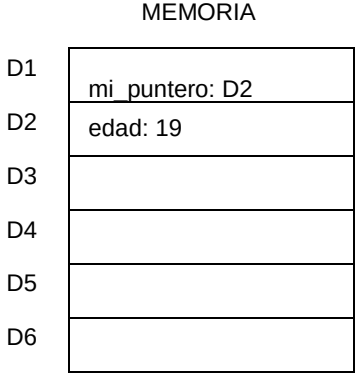


Figura 7.8. Estado de la memoria tras hacer uso de & y ***.**

La conclusión de todo esto es que cuando realicemos asignaciones del tipo:

```
mi puntero = &edad;
```

luego, cada vez que escribamos `*mi_puntero` será igual que escribir `edad`. ¿Y si añadimos al programa anterior una línea más?

```
int main (void)
{
    int *mi_puntero; int
    edad = 18; mi_puntero
    = &edad;

    edad = *mi_puntero + 1;

    *mi_puntero = 0;
}
```

El resultado es que si `mi_puntero` almacena la dirección `D2`, `*D2` hace referencia a la variable `edad`, que es lo que hay en la dirección `D2`, luego:

```
*mi_puntero=0
```

es igual que:

```
edad=0;
```

El estado resultante de la memoria lo vemos en la figura 7.9.

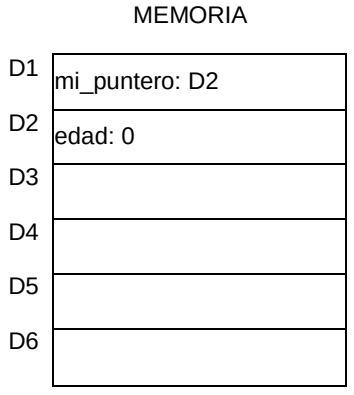


Figura 7.9. Estado de la memoria.

Como puede comprobarse, podemos llegar a trabajar con variables sin utilizar su identificador, únicamente utilizando variables puntero con sus direcciones. Veamos un último ejemplo.

```
int main (void)
{
   int lapices, plumas, total;
   int *puntero_lapices;

   int *puntero_plumas;

   int *puntero_total;

   puntero_lapices = &lapices;

   puntero_plumas = &plumas;

   puntero_total = &total;

   *puntero_lapices = 5;
   *puntero_plumas = 2;

   *puntero_total = *puntero_lapices + *puntero_plumas;
}
```

El estado final de la memoria se muestra en la figura 7.10.

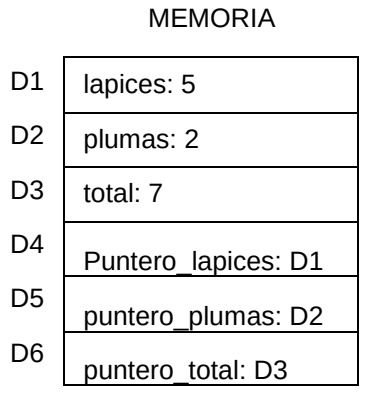


Figura 7.10. Estado final de la memoria.

Referencias

Si los punteros permitían trabajar con otras variables de forma indirecta, para lo cual utilizábamos los operadores & y *, las referencias también ofrecen esta posibilidad, y con una mayor comodidad. Sin embargo, las referencias son exclusivas de C++.
Una referencia es una especie de duplicado de una variable, de manera que cualquier operación que se realice sobre el duplicado realmente se produce sobre la variable original.
Las referencias tienen una condición: se les debe asignar la variable original en el momento de la declaración, de lo contrario se produciría un error.
La sintaxis de una referencia es:

```
tipo &identificador_referencia = identificador_original;
```

Donde tipo debe ser el mismo tipo de dato de la variable original, identif icador_referencia el nombre de la referencia e identificador_ original el nombre de la variable a la que se hace la referencia. El símbolo & (ampersand) es necesario para crear una referencia.
Veamos un ejemplo.

```
int main (void)
{
   int codigo;

   int &codigo2 =

   codigo; codigo2 = 5;
}
```

La primera línea de código dentro de la función main:

```
int codigo;
```

declara una variable de tipo int llamada código.
La siguiente línea:

```
int &codigo2 = codigo;
```

crea una referencia llamada codigo2 a la variable código. A partir de esta declaración, cualquier operación que se realice con codigo2 realmente se realiza con la variable código. Así que la asignación:

```
codigo2 = 5;
```

tiene como resultado que la variable código almacena el valor 5, ya que es equivalente a:

```
codigo = 5;
```

Podemos representar el estado de la memoria tal y como se muestra en la figura 7.11.

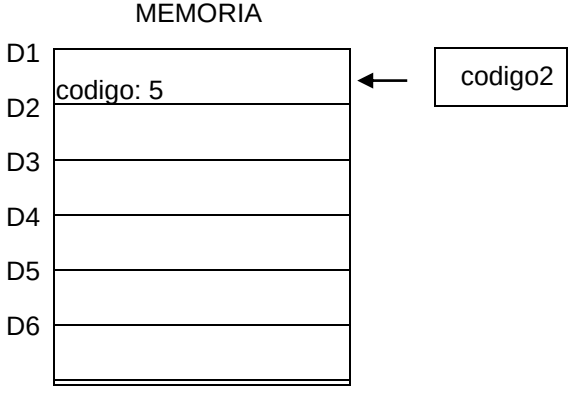


Figura 7.11. Representación de la memoria con una referencia.

En esta figura hemos representado a la referencia codigo2 dentro de una casilla que apunta con una flecha a la variable a la que hace referencia: codigo. También la hemos representado a la misma altura, pero fuera de las casillas de la memoria.
La idea es que la referencia codigo2 es equivalente a la variable codigo, hasta el punto de que si escribimos &codigo2 obtenemos DI, la dirección de la variable codigo.

Como se puede comprobar, a la hora de utilizar una referencia no se usa ningún operador, a diferencia de los punteros, con los que se utilizaban los operadores & y *. Las referencias son más simples, pero a los punteros se les puede dar más usos que a las referencias. Habrá ocasiones en las que sólo se puedan emplear punteros y otras en las que se puedan emplear tanto punteros como referencias.

Pongamos un ejemplo más.

```
int main (void)
{
   int lapices, plumas, total; int &referencia_lapices =
   lapices; int &referencia_plumas = plumas; int
   &referencia_total = total;

   referencia_lapices = 5; referencia_plumas = 2;
   referencia_total = referencia_lapices + referencia_plumas;
}
```

El resultado tras ejecutar este programa es que la variable lapices almacena el valor 5, la variable plumas el valor 2 y la variable total el valor 7. Esto es porque todas las operaciones que se hicieron con las referencias, referencia_lapices, referencia_plumas y referencia_total, realmente se estaban haciendo con las variables lapices, plumas y total. En la figura 7.12 observamos una representación de la memoria tras ejecutar el programa anterior.

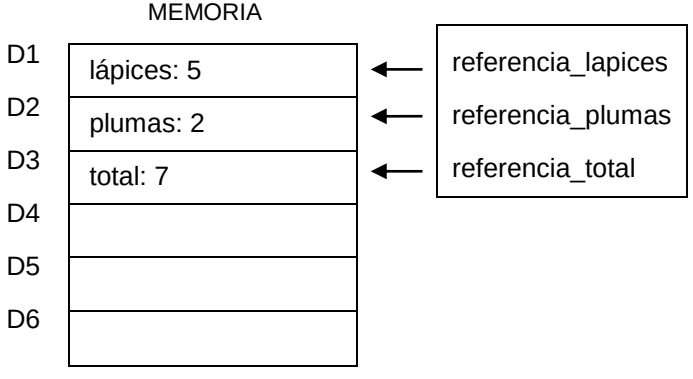


Figura 7.12. Representación de la memoria con tres referencias.

Sería interesante que comparase el código de este programa y su representación de la memoria con el caso de hacerlo con punteros y con referencias.

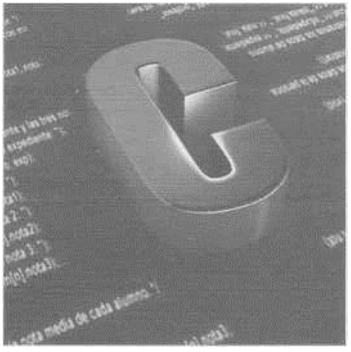
Ejercicios resueltos

Intente realizar los siguientes ejercicios. Encontrará la solución al final de este libro.

1. Realice un programa en el que almacene en una variable puntero la dirección de memoria de una variable de tipo `char` inicializada con el valor 'X'. Represente el estado de la memoria.
2. Si `mi_puntero` es una variable puntero de tipo `int`, ¿cuál de las siguiente afirmaciones es falsa?
   - Mediante `&mi_puntero` obtenemos la dirección de memoria de la variable `mi_puntero`.
   - Mediante `*mi_puntero` obtenemos el contenido que hay en la dirección de memoria que almacena la variable `mi__puntero`.
   - Mediante `mi_puntero` obtenemos el contenido que hay en la dirección de memoria que almacena la variable `mi_puntero`.
3. Realice un programa en el que se declare una variable de tipo entero y, haciendo uso de referencias, le asigne el valor cero a esta variable e incremente su valor en una unidad. Represente el estado de la memoria.

Resumen

Ya conocemos el manejo básico de punteros y referencias, los cuales nos permiten trabajar con otras variables o elementos indirectamente. Recordemos que las referencias son exclusivas de C++. Puede que ahora parezca que no sirven para nada, pero los punteros están muy ligados a C/C++, de ahí su importancia. Serán nuestros compañeros de viaje a lo largo de cualquier aventura de programación en C/C++. Pero lo importante de este capítulo es comprenderlos y saber trabajar con ellos mínimamente. La utilidad o necesidad de su uso comienza a partir del próximo capítulo.



Capítulo 8

Entrada y salida estándar

En este capítulo aprenderá:

- A mostrar mensajes en la pantalla.
- A almacenar en variables datos escritos mediante el teclado.
- A hacer programas en función de los datos escritos mediante el teclado.

Introducción

Hasta ahora hemos aprendido algunas de las herramientas mínimas para hacer programas, pero nunca se mostraba nada en la pantalla. Una de las cosas que todos ansiamos por aprender cuando empezamos a programar es mostrar datos en la pantalla y trabajar con los datos que se escriben mediante el teclado. Pues adelante.

Entrada y salida en C

Las bibliotecas de C nos ofrecen diversas funciones para mostrar datos en la pantalla y almacenar datos escritos mediante el teclado en variables. Cada función la explicaremos en primer lugar formalmente y, después, desde un punto de vista totalmente práctico y sencillo. De esta manera, las explicaciones serán aptas para todo tipo de niveles.

Salida de caracteres: putchar

Para mostrar un carácter en pantalla podemos utilizar la función `putchar` de la biblioteca `stdio. h.` Su cabecera o prototipo es:

```
int putchar (int carácter);
```

Donde el parámetro `carácter` es el código del carácter a mostrar en la pantalla. La función retorna el código del carácter a mostrar.
Después de su explicación formal pasemos a un ejemplo para demostrar su sencillez.

```
#include <stdio.h>
int main (void)
{
   /* Mostramos un carácter en la pantalla */
   putchar ('A');
   /* Hacemos una pausa hasta que el usuario pulse Intro */
   fflush(stdin);
   printf("\nPulse Intro para
   finalizar...");
   getchar();
}
```

El resultado de este programa es que la letra A aparece escrita en la pantalla, como se muestra en la figura 8.1.

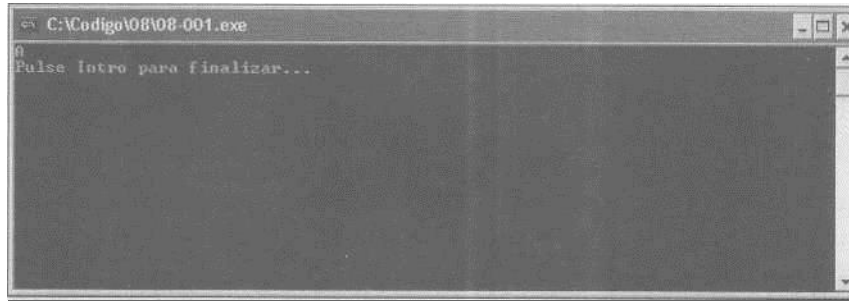


Figura 8.1. Resultado de putchar('A').

Como puede comprobarse, para mostrar un carácter en la pantalla mediante la función `putchar`, lo único que hay que hacer es escribir el carácter dentro de los paréntesis de la función:

```
putchar ('A');
```

No debemos olvidar añadir una línea al principio del programa para indicar que vamos a utilizar la biblioteca `st di o`, a la que pertenece la función `putchar`:

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

Después de todo, es mucho más sencillo de lo que parecía al principio. Pero, vamos a experimentar con esta función; probemos lo siguiente.

```
#include <stdio.h>
int main (void)
{
   char letra;
   letra = 'A';
   putchar (letra);
   /* Hacemos una pausa hasta que el usuario pulse Intro */
   fflush(stdin);
   printf("\nPulse Intro para finalizar...");
   getchar () ;
}
```

¿Se ejecutará correctamente este programa? Sí, es más, tras ejecutarlo aparecerá en pantalla lo mismo que en la figura 8.1. Analicémoslo.

```
char letra;
```

Esta línea de código declara una variable de tipo `char` llamada `letra`, que sin duda alguna almacenará un carácter, el cual se le asigna a continuación.

```
letra = 'A';
```

Ahora la variable `letra` almacena el valor `'A'`, luego:

```
putchar (letra);
```

es equivalente a:

```
putchar ('A');
```

Así que también podemos escribir dentro de los paréntesis de la función una variable que almacene un carácter, porque tendrá el mismo efecto que escribir directamente ese carácter.
Veamos el siguiente programa:

```
#include <stdio.h>

int main (void)
{
   char letra;
                                                          */
   letra = 'A';       /* Ahora letra almacena el carácter 'A'.
   putchar (letra);   /* Mostramos el valor de letra. */
   letra = letra + 1; /* Ahora letra almacena el carácter 'B'. */
   putchar (letra);   /* Mostramos el valor de letra. */
   letra = letra + 1; /* Ahora letra almacena el carácter 'C'. */
   putchar (letra);   /* Mostramos el valor de letra. */
   letra = letra + 1; /* Ahora letra almacena el carácter 'D'. */
   putchar (letra);   /* Mostramos el valor de letra. */

   /* Hacemos una pausa hasta que el usuario pulse Intro
   */ fflush(stdin);
   printf("\nPulse Intro para
   finalizar...");
   getchar ();
}
```

Ahora, gracias a la función `putchar` podemos ver en la pantalla el resultado de ejecutar este programa (véase la figura 8.2).

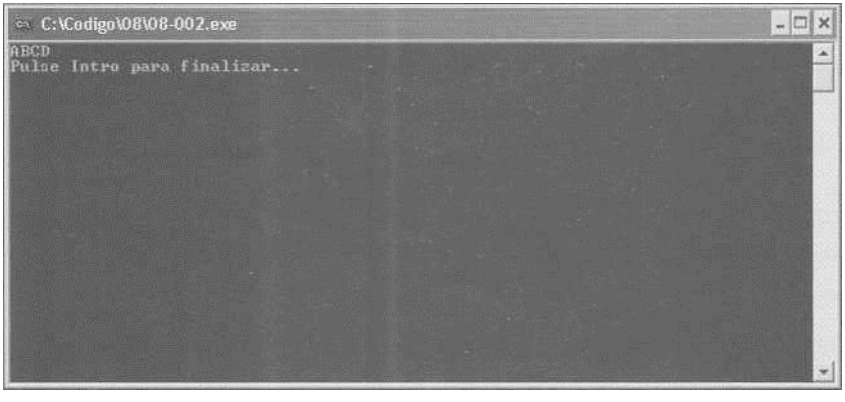


Figura 8.2. Resultado de mostrar en pantalla el valor de la variable letra.

Entrada de caracteres: getchar, getch, getche

Para poder explicar las funciones de entrada de caracteres, `getchar`, `getch` y `getche`, utilizaremos el mismo programa de ejemplo con cada una de las funciones, para así comprobar claramente la diferencia entre ellas.

getchar

Para almacenar en una variable un carácter escrito mediante el teclado (o, como se suele decir, leer del teclado un carácter) podemos utilizar la función `getchar` de la biblioteca `stdio`. Su cabecera o prototipo es:

```
int getchar (void);
```

Retorna el código del carácter escrito mediante el teclado. Esta función espera a que se pulse la tecla **Intro** para continuar con la ejecución del programa ("sí espera Intro") y muestra en pantalla el carácter escrito ("con eco"). Un ejemplo:

```
#include <stdio.h>

int main (void)
{
   char letra;
   /* Obtenemos un carácter del teclado y lo mostramos */
   letra = getchar();

   putchar(letra);

   /* Hacemos una pausa hasta que el usuario pulse Intro */
   fflush(stdin);
   printf("\nPulse Intro para finalizar...");
   getchar();
}
```

Este programa declara una variable llamada `letra` la cual almacenará un carácter:

```
char letra;
```

En la siguiente línea de código:

```
letra = getchar ();
```

primero se ejecuta lo que se encuentra a la derecha de la asignación, que es la llamada a la función `getchar`. Entonces, aparecerá en pantalla el cursor, una raya parpadeando. Esto indica que el programa está a la espera de que escribamos algún carácter, por ejemplo la 'H'. El carácter escrito ('El') aparece

parpadeando. Hasta que no pulsemos la tecla **Intro** el programa no continuará, esto es lo que significa que la función "sí espera Intro". Pulsamos **Intro.** A continuación, la llamada a la función toma el valor del carácter escrito, es decir, la línea de código anterior sería equivalente a:

```
letra = 'H';
```

por lo que la variable `letra` almacenará el valor 'H'. Con:

```
putchar(letra);
```

se muestra en pantalla el valor de la variable `letra`, que es 'H'.
No olvidemos que la línea:

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

es necesaria para utilizar la función `putchar` y `getchar`, puesto que pertenecen a la biblioteca estándar stdio.
El resultado de este programa en la pantalla lo vemos en la figura 8.3.

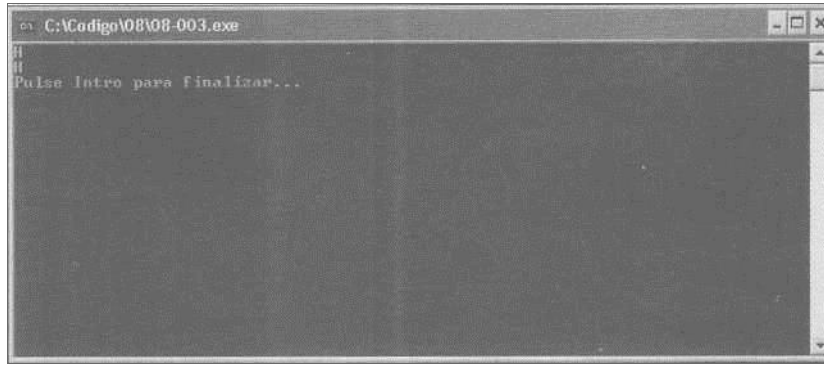


Figura 8.3. Resultado del programa con getchar.

getch

Otra función para leer un carácter del teclado es `getch` de la biblioteca `conio`. Es importante señalar que la biblioteca `conio` no es estándar, pero es muy solicitada por programadores acostumbrados a Borland, por lo que Dev-C++ también la ha incluido. Su cabecera es:

```
int getch (void);
```

Esta función retorna el código del carácter escrito. Es "sin eco" y "no espera Intro", esto es, el carácter escrito no se muestra en la pantalla y en cuanto se escriba éste la ejecución del programa continúa, sin tener que pulsar la tecla **Intro.**

El programa anterior, sustituyendo la función `getchar` por `getch`, sería el siguiente.

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

int main (void)
{
   char letra;

   /* Obtenemos un carácter del teclado y lo mostramos */

   letra = getch ();

   putchar(letra);

   /* Hacemos una pausa hasta que el usuario pulse Intro */

   fflush(stdin);
   printf("\nPulse Intro para finalizar...");
   getchar();
}
```

Tras ejecutarlo, el resultado es prácticamente el mismo. La línea:

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

es necesaria para utilizar la función `putchar`, perteneciente a la biblioteca `stdio`, y la línea:

```
#include <conio.h>
```

es necesaria para utilizar la función `getch`, perteneciente a la biblioteca `conio`.

En la línea de código:

```
letra = getch ();
```

primero se ejecuta lo que se encuentra a la derecha de la asignación, que es la llamada a la función `getch`. Entonces, aparecerá en pantalla el cursor. El programa está a la espera de que escribamos algún carácter, por ejemplo la 'M'. El carácter escrito ('M') NO aparece en pantalla, esto es lo que significa que es "sin eco", y el programa continúa la ejecución inmediatamente, sin tener que pulsar la tecla Intro, esto es lo que significa que la función "no espera Intro". Así que la llamada a la función toma el valor del carácter escrito, es decir, la línea de código anterior sería equivalente a:

```
letra = 'M';
```

por lo que la variable `letra` almacenará el valor 'M'. Con:

```
putchar(letra);
```

se muestra en pantalla el valor de la variable `letra`, que es 'M'.

El resultado de este programa en la pantalla lo vemos en la figura 8.4.

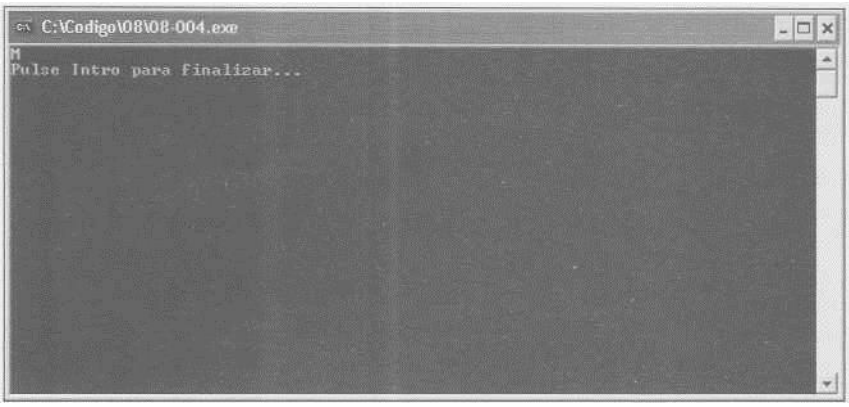


Figura 8.4. Resultado del programa con getch.

Las siguientes dos líneas de código pueden sernos muy útiles. Pruebe a incluirlas en algún programa.

```
printf ("Pulse una tecla para continuar...");
getch();
```

getche

La función para leer un carácter del teclado `getche` pertenece a la biblioteca `conio`. Recordemos que la biblioteca `conio`, aunque no es estándar, sí está disponible en Borland y en Dev-C++. Su prototipo es:

```
int getche (void);
```

Esta función retorna el código del carácter escrito. Es "con eco" y "no espera Intro".
Analicemos el mismo programa con la función `getche`.

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

int main (void)

{
   char letra;

   /* Obtenemos un carácter del teclado y lo mostramos */

   letra = getche ();

   putchar(letra);

   /* Hacemos una pausa hasta que el usuario pulse Intro */
   fflush(stdin);

   printf("\nPulse Intro para finalizar...");

   getchar();

}
```

En la línea de código:

```
letra = getche();
```

primero se ejecuta lo que se encuentra a la derecha de la asignación, que es la llamada a la función `getche`. Entonces, aparecerá en pantalla el cursor, una raya parpadeando. Esto indica que el programa está a la espera de que escribamos algún carácter, por ejemplo la 'S'. El carácter escrito ('S') aparece en pantalla, esto es lo que significa que es "con eco", y el programa continúa la ejecución inmediatamente, sin tener que pulsar la tecla Intro, y esto es lo que significa que la función "no espera Intro". Así que la llamada a la función toma el valor del carácter escrito, es decir, la línea de código anterior sería equivalente a:

```
letra = 'S';
```

por lo que la variable `letra` almacenará el valor `'S'`.

Con:

```
putchar(letra);
```

se muestra en pantalla el valor de la variable `letra`, que es `'S'`.

La línea:

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

es necesaria para utilizar la función `putchar`, perteneciente a la biblioteca `stdio`, y la línea:

```
#include <conio.h>
```

es necesaria para utilizar la función `getche`, perteneciente a la biblioteca `conio`.

El resultado de este programa en la pantalla lo vemos en la figura 8.5.

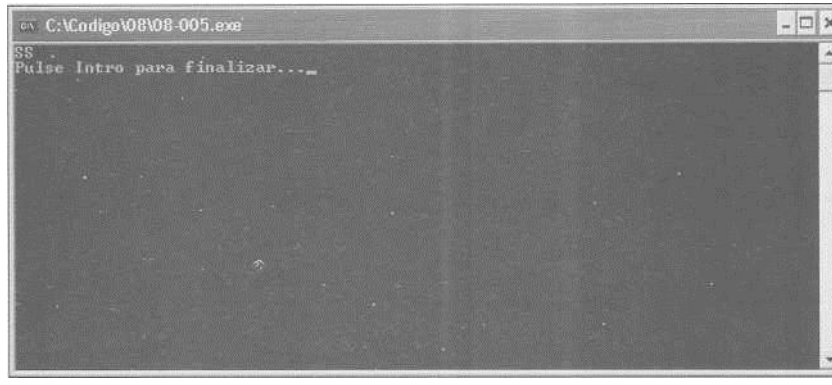


Figura 8.5. Resultado del programa con getchar.

Entrada y salida formateada

printf

La función `printf`, que pertenece a la biblioteca `stdio`, nos permite mostrar en la pantalla todo tipo de datos: valores de variable de tipo `int`, valores de constantes, cadenas de caracteres. Normalmente, la forma de utilizar esta función es:

```
printf ("cadena_con_formato" [, lista_de_argumentos]);
```

Para explicar qué es `cadena_con_formato` y `lista_de_argumentos`, lo mejor será apoyarnos en un ejemplo muy simple en el que se muestra un mensaje en pantalla. Unicamente hay decir que `lista_de_argumentos` aparece entre corchetes porque es opcional.

```
#include

<stdio.h> int

main (void)
{
   printf ("Este texto sale en pantalla");

   /* Hacemos una pausa hasta que el usuario pulse Intro */
   fflush(stdin);
   printf("\nPulse Intro para finalizar...");
   getchar();
}
```

Como la función `printf` pertenece a la biblioteca `stdio`, escribimos la línea:

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

El resultado de ejecutar la llamada a la función `printf`:

```
printf ("Este texto sale en la pantalla");
```

es que en pantalla aparece: "Este texto saldrá en la pantalla", como puede observarse en la figura 8.6.

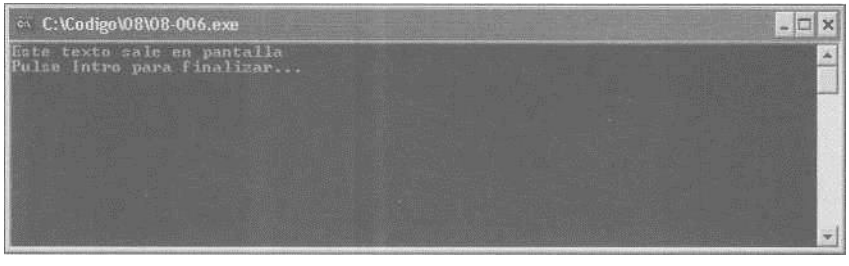


Figura 8.6. Mensaje mostrado mediante un printf.

Recordemos que llamábamos cadena de caracteres a una serie de caracteres escritos entre comillas dobles, luego "Este texto sale en la pantalla" es una cadena de caracteres. En esta cadena podemos incluir los caracteres especiales, como los recogidos en la tabla 8.1.

Tabla 8.1. Caracteres especiales.

Retroceso	'\b'
Salto de línea	'\n'
Tabulación horizontal	'\t'
Comillas dobles	'\"'
Comilla simple	'\''
Barra diagonal invertida	'\\'

En el siguiente programa encontramos algunos ejemplos y en la figura 8.7 el resultado en pantalla.

```
#include <stdio.h>

int main (void)
{
   printf ("\tEste texto se muestra tabulado."); printf ("\nEste
   texto se muestra en otra linea.\n") ; printf ("V'Este texto
   se muestra entre comillas dobles\""); printf ("\n\tEsta es la
   letra \'A\'.");

   /* Hacemos una pausa hasta que el usuario pulse Intro */
   fflush(stdin);
   printf("\nPulse Intro para finalizar...");
   getchar();
}
```

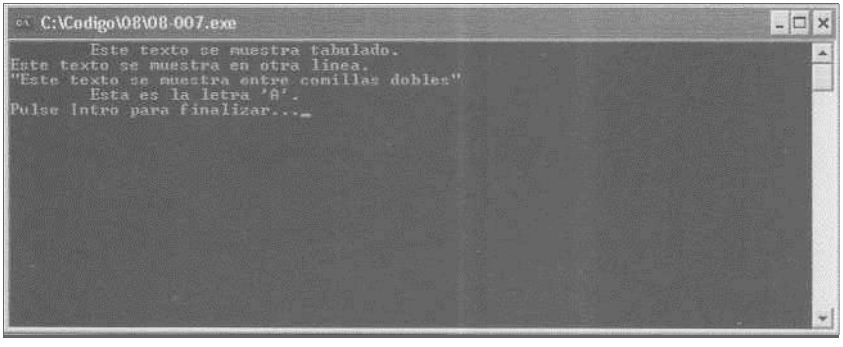


Figura 8.7. Resultado tras usar caracteres especiales en printf.

La `cadena_con_formato` es una cadena de caracteres que puede incluir determinados comandos, los cuales nos permiten, por ejemplo, mostrar en pantalla el valor de una variable, siendo necesario utilizar la `lista_de_argumentos` (que en los programas anteriores no se ha utilizado). Analicemos el siguiente programa.

```
#include <stdio.h>

int main (void)
{
   int x = 10;

   printf ("El valor de la variable x es: %d", x);

   /* Hacemos una pausa hasta que el usuario pulse Intro */
   fflush(stdin);
   printf("\nPulse Intro para finalizar...");

   getchar ();
}
```

Donde:

```
int x = 10;
```

declara una variable llamada `x` de tipo `int` con el valor inicial `10`.
En este caso, la cadena con formato de la función `printf`:

```
"El valor de la variable x es: %d"
```

contiene un comando: `%d`. Estos dos caracteres (%d) serán sustituidos por el valor de la variable x de la lista de argumentos antes de mostrar en pantalla la cadena con formato. Así que la cadena con formato sería equivalente a la siguiente:

```
"El valor de la variable x es: 10"
```

que es el mensaje que aparece en la pantalla (véase la figura 8.8).

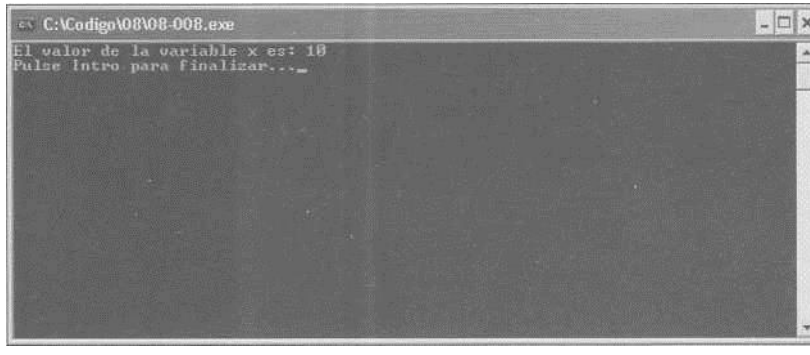


Figura 8.8. Mensaje mostrado mediante un printf con un argumento.

Éste es el mecanismo para mostrar por pantalla el valor de una variable mediante la función `printf`. Los comandos para mostrar valores de variables u otros elementos tienen la siguiente estructura:

```
%[anchura].[precisión] caracter_de_tipo
```

En donde `anchura` es el número mínimo de caracteres que ocupará el dato en pantalla (que es opcional). Para mostrar los números reales es muy útil `precisión` (que es opcional), que permite especificar el número de dígitos tras el punto. Si lo que queremos mostrar es un dato de tipo `int, char` u otro, lo tenemos que indicar mediante el `caracter_de_tipo`. Los principales caracteres de tipo se recogen en la tabla 8.2.

Tabla 8.2. Principales caracteres de tipo.

Carácter de tipo	Significado
c	carácter
s	cadena de caracteres
d	número entero
o	número octal
x	número hexadecimal
f	número real
p	dirección de memoria

Veamos unos cuantos ejemplos.

```
#include <stdio.h>

int main (void)
{
   int entero = 10;
   float real = 2.5614;
   char carácter = 'F';

   /* Muestra el valor de la variable carácter. */
   printf ("\n %c", carácter);

   /* Muestra el valor de la variable entero. */
   printf ("\n %d", entero);

   /* Muestra el valor de la variable real. */
   printf ("\n %f", real);

   /* Muestra el valor de la variable real con dos digitos de precisión. */
   printf ("\n %.2f", real);
```

```
    /* Muestra el valor de la variable entero con una anchura de 10 dígitos. */
    printf ("\n %10d", entero);

    /* Muestra el valor de la variable real con una anchura de 10 dígitos. */

    printf ("\n %10f", real);

    /* Muestra la dirección de memoria de la variable carácter. */
    printf ("\n %p", &caracter);

    /* Hacemos una pausa hasta que el usuario pulse Intro */
    fflush(stdin);
    printf("\nPulse Intro para finalizar...");
    getchar() ;
}
```

En la figura 8.9 mostramos la salida por pantalla.

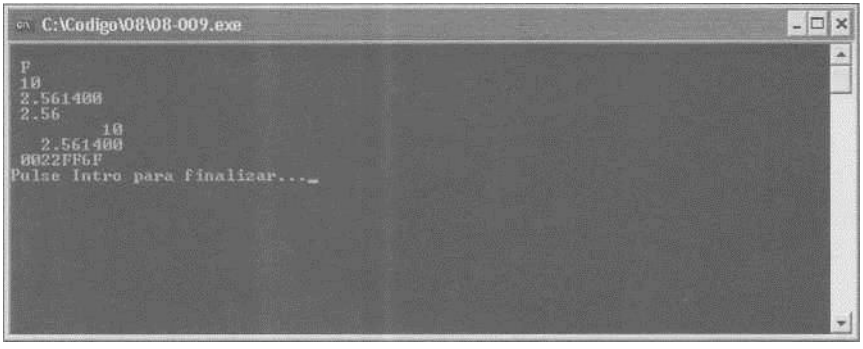


Figura 8.9. Resultado de la ejecución del programa.

Éstos son algunos de los usos más simples. El carácter de tipo 's' y las cadenas de caracteres las dejaremos para más adelante.
Pero qué ocurre si ejecutamos el siguiente programa.

```
#include <stdio.h>

int main (void)
{
    char carácter = 'A';

    printf ("%d", carácter);
    /* Hacemos una pausa hasta que el usuario pulse Intro */
    fflush(stdin);
    printf("\nPulse Intro para finalizar...");
    getchar();
}
```

En la pantalla aparece el número 65. Lo que ha sucedido es que se ha mostrado el valor de la variable carácter como un número. Ya dijimos que a

cada carácter le corresponde un código en la tabla ASCII y al carácter 'A' le corresponde el 65. Este es el valor que se obtiene al convertir el carácter 'A' a número. Y ¿se puede hacer a la inversa? Sí.

```
#include <stdio.h>

int main (void)
{
   int numero = 66;
    printf ("%c", numero);

   /* Hacemos una pausa hasta que el usuario pulse Intro */
   fflush(stdin);
   printf("\nPulse Intro para finalizar...");
   getchar();
}
```

En la pantalla aparece el carácter 'B'. El valor de la variable `numero` se ha mostrado como un carácter. Al número 66 (o código 66) le corresponde el carácter 'B', según la tabla ASCII.
Este tipo de conversiones lo podemos realizar entre distintos tipos. Pruebe lo siguiente.

```
#include <stdio.h>

int main (void)
{
   int numero = 500;

   /* Mostramos numero en formato octal: 764 */

   printf ("\n %o", numero);
   /* Mostramos numero en formato hexadecimal: 1F4 */

   printf ("\n %x", numero);
   /* Mostramos numero en formato real: 500.00 */

   printf ("\n %.2f", numero);

   /* Hacemos una pausa hasta que el usuario pulse Intro */
   fflush(stdin);
   printf ("\nPulse Intro para finalizar...");

   getchar();
}
```

En la lista de argumentos, donde escribimos el nombre de las variables u otros elementos, también podemos escribir expresiones:

```
printf ("%d", 3*2+5);
```

Para no tener que escribir tantas veces `printf`, esta función nos permite mostrar varios valores de variables u otros elementos en una misma llamada a `printf`:

```
int numero = 10;
char carácter = 'A';
printf ("numero vale %d y carácter vale %c", numero, carácter);
```

Los comandos se sustituyen en el siguiente orden: el primer comando por el valor de la primera variable, el segundo comando por el valor de la segunda variable y así sucesivamente. La anterior llamada a la función printf tiene la siguiente salida por pantalla:

```
numero vale 10 y carácter vale A
```

Después de aprender tantas cosas sobre printf vamos a hacer un programa útil y sencillo. Consiste en pedir al usuario un carácter y mostrar a continuación su código ASCII.

```
#include <stdio.h>

int main (void)
{
   char carácter;

   /* Pedimos un carácter al usuario y lo almacenamos en la variable
   carácter. */
   printf ("Escriba un carácter y pulse Intro: ")
    carácter = getchar();

   /* Mostramos el carácter y su codigo ASCII. */
   printf ("\nEl codigo ASCII de %c es: %d", carácter, carácter);

   /* Hacemos una pausa hasta que el usuario pulse Intro */
   fflush(stdin);
   printf("\nPulse Intro para finalizar...");

   getchar ();
}
```

Pruébelo con distintos caracteres, podrá saber su código. En la figura 8.10 mostramos la salida por pantalla si escribimos el carácter 'A'.

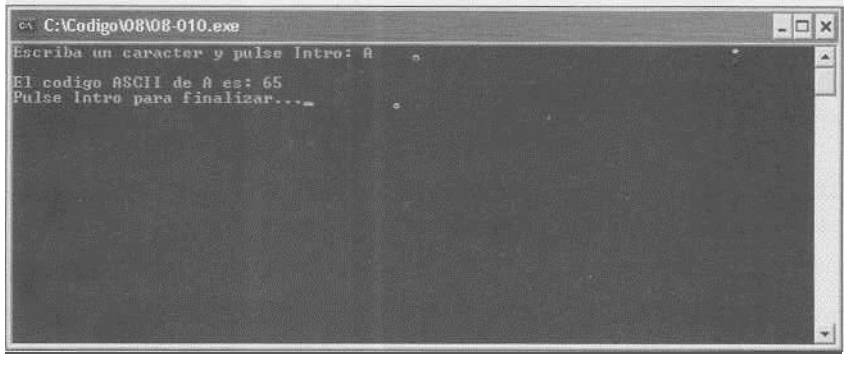


Figura 8.10. Cálculo del código ASCII de un carácter.

scanf

La función scanf, que pertenece a la biblioteca stdio, nos permite almacenar en una variable datos (escritos mediante el teclado) de diversos tipos: int, char o f loat, entre otros. Normalmente, la forma de utilizar esta función es:

```
scanf ("comando", &variable);
```

Donde variable es el identificador de la variable en la que queremos almacenar el dato que se escriba mediante el teclado. Como al especificar la variable (en el segundo parámetro de la función) hay que indicar su dirección de memoria, debemos anteponer el símbolo & (ampersand) al identificador. El comando indica qué tipo de dato vamos a leer del teclado. Su estructura es:

```
% [anchura] caracter_de_tipo
```

Los principales caracteres de tipo se indicaron anteriormente en la tabla 8.1. La función scanf recoge en la variable especificada todos los datos escritos hasta encontrar un carácter en blanco, un tabulador, hasta que se pulse **Intro,** o hasta alcanzar el número de caracteres indicados en anchura.

Al ejecutar cualquier llamada a la función scanf en la pantalla aparece el cursor parpadeando. Esto indica que el programa está a la espera de que escribamos algún dato. El dato escrito aparece en pantalla, luego es "con eco", y el cursor sigue parpadeando. Hasta que no pulsemos la tecla **Intro** el programa no continuará, luego "sí espera Intro". Pulsamos **Intro.** A continuación, la variable especificada en la función toma el valor del dato escrito. Mostramos un programa con algunos ejemplos.

```
#include <stdio.h>
int main (void)
{
   int numero;
   char carácter;
   /* Pedimos un numero de 3 cifras al usuario. */
   printf ("Escriba un numero de máximo 3 cifras: ");
   /* Lee del teclado un número de 3 digitos como
   máximo y lo almacena en la variable numero. */
   scanf ("%3d", &numero);
   /* Limpiamos el buffer del teclado.
   */ fflush (stdin);
   /* Mostramos el valor de la variable numero. */
   printf ("El valor de numero es: %d \n", numero);
```

```
  /* Pedimos un carácter al usuario. */
  printf ("Escriba un carácter: ");

  /* Lee del teclado un carácter y lo almacena en la variable
  carácter. */ scanf ("%c", &caracter);

  /* Mostramos el valor de la variable carácter. */ printf
  ("El valor de carácter es: %c", carácter);

  /* Hacemos una pausa hasta que el usuario pulse Intro */
  fflush(stdin) ;
  printf ("\nPulse Intro para finalizar...");
  getchar ();
}
```

Si escribimos el número 1234 y el carácter 'V' el resultado en pantalla es igual al de la figura 8.11.

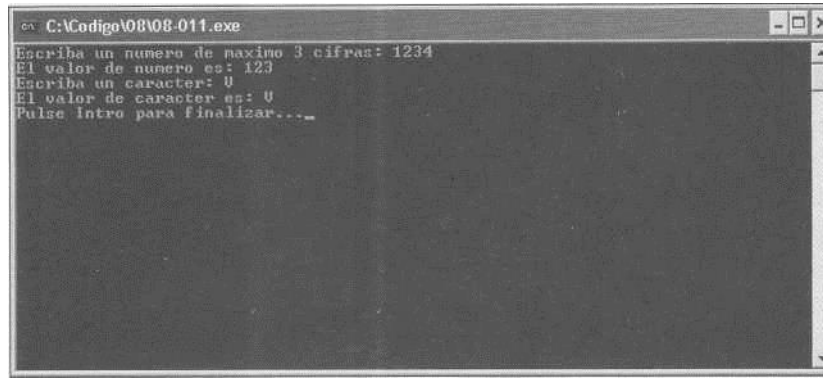


Figura 8.11. Resultado del programa con scanf.

Tal como puede observarse el número se trunca por tener más de 3 dígitos. En el caso de que la entrada de datos no sea válida en la función scanf, el *buffer* del teclado podría seguir conteniendo datos, lo que ocasiona el salto de la siguiente llamada a la función scanf. Para evitarlo, borramos el *buffer* del teclado con la llamada a la función:

```
fflush (stdin);
```

tras la llamada a la función scanf que nos ocasiona el problema, aunque para más seguridad se puede incluir tras cada scanf. El argumento stdin hace referencia al teclado, la entrada estándar. La función fflush pertenece a la biblioteca stdio.

La función scanf admite más posibilidades, pero de momento tenemos todo lo necesario.

Advertencia:

En algunos casos, tras ejecutar una llamada a la función scanf o getchar, el buffer *del teclado sigue contiendo información. Esto produce que la próxima vez que se ejecute una de estas funciones se recojan esos datos del* buffer, *algo que normalmente no se desea. Si en el* buffer *del teclado quedase almacenada la pulsación de la tecla* Intro, *la próxima llamada a scanf o getchar se saltaría automáticamente, ya que el ordenador interpretará que nosotros hemos pulsado dicha tecla. Para evitar estos casos, podemos limpiar el* buffer *llamando a* `fflush(stdin)` *tras* `scanf` *o* `getchar`.

El siguiente programa pide al usuario que escriba un número y, tras pulsar **Intro,** se muestra su doble.

```
#include <iostream.h>

int main (void)
{
    int numero;

    /* Pedimos al usuario que escriba un mensaje. */
    printf ("Escriba un numero: ");

    /* El numero que escriba el usuario lo almacenamos en la
    variable numero. */

    scanf ("%d", (&numero) ;

    /* Mostramos un mensaje indicando cual es su doble. */
    printf ("\nEl doble de %d es %d", numero, numero*2);

    /* Hacemos una pausa hasta que el usuario pulse Intro */
    fflush(stdin);
    printf("\nPulse Intro para finalizar...");
    getchar () ;
}
```

En la figura 8.12 se muestra la salida por pantalla.

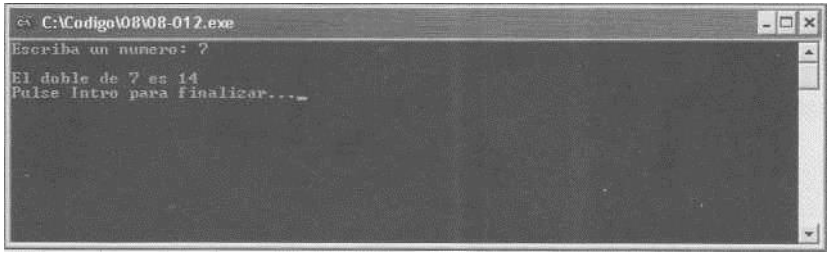


Figura 8.12. Cálculo del doble de un número.

Entrada y salida en C++

Salida en C++

Mediante el operador << podemos enviar datos a cout, la salida estándar (la pantalla). Para ello hay que hacer uso de la biblioteca iostream. La sintaxis básica es:

```
cout << elemento;
```

Donde elemento puede ser una variable, una constante, una cadena o una expresión que queremos enviar a la pantalla. Veamos un ejemplo.

```
#include <iostream.h>
#define PI 3.14

int main (void)
{
   int numero = 10;
   /* Mostramos el valor de la variable numero en la pantalla. */
   cout << numero;

   /* Hacemos un salto de linea. */

   cout << '\n';

   /* Mostramos el valor de un carácter en la pantalla. */

   cout << 'K';

   /* Hacemos un salto de linea. */

   cout << '\n';

   /* Mostramos el valor de la constante PI en la pantalla. */

   cout << PI;

   /* Hacemos un salto de linea. */

   cout << '\n';

   /* Mostramos el valor resultante de una expresión en la pantalla. */
   cout << ((3+7)*2);

   /* Hacemos un salto de linea. */ cout << '\n';

   /* Mostramos una cadena de caracteres en la pantalla. */

   cout << "Hola a todos";

   /* Hacemos una pausa hasta que el usuario pulse Intro */
   fflush(stdin);

   cout << "\nPulse Intro para finalizar...";
   cin.get () ;
```

El resultado de este programa lo encontramos en la figura 8.13, y la verdad es que nos es muy claro.

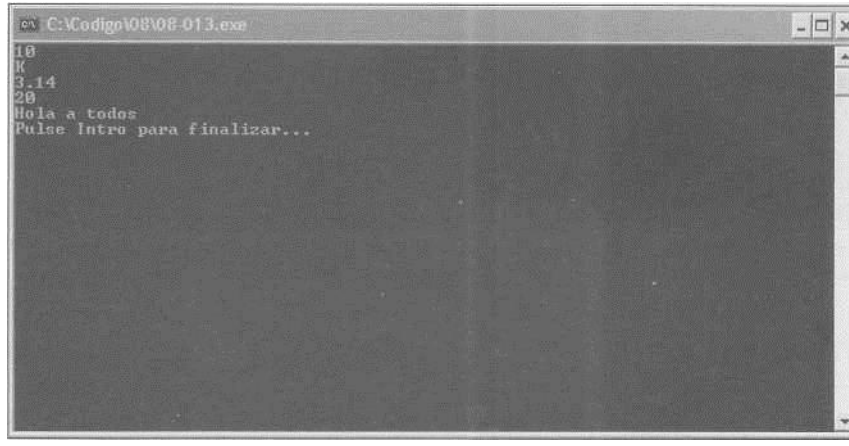


Figura 8.13. Resultado al utilizar cout.

Podemos mejorarlo a la vez que practicamos con el envío encadenado de elementos a la pantalla, que nos ahorrará escribir tantas veces cout «. La sintaxis para enviar elementos encadenados es:

```
cout << elemento_1 « elemento_2 << ... << elemento_n;
```

A continuación, el mismo ejemplo que el anterior, pero con una mejor presentación en la pantalla.

```
#include <iostream.h>
#define PI 3.14

int main (void)
{
    int numero = 10;

    cout << "La variable numero almacena: " << numero << "\n";
    cout << "La primera letra de KILO es: " << ' K' << "\n"; cout
    << PI << " es el valor de la constante pi.\n"; cout « "((3+7)*2)
    = " « ((3+7)*2) << "\n"; cout << "Hola a todos\n";

    /* Hacemos una pausa hasta que el usuario pulse Intro */
    fflush(stdin);
    cout « "\nPulse Intro para finalizar...";
    cin.get();
{
```

En la figura 8.14 observamos el resultado de este programa.

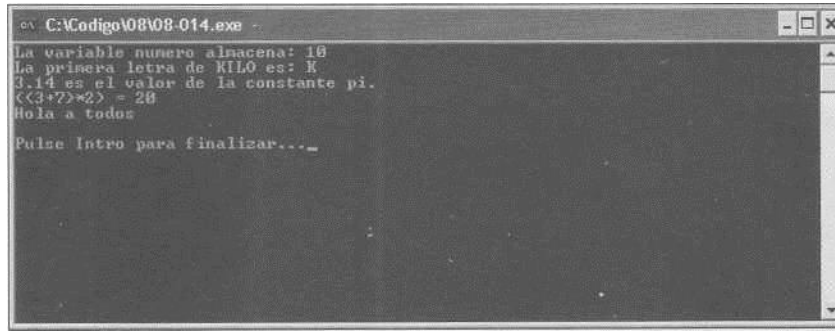


Figura 8.14. Resultado tras utilizar cout con elementos encadenados.

Entrada en C++

Mediante el operador >> podemos recibir datos de cin, la entrada estándar (el teclado). Para ello hay que hacer uso de la biblioteca iostream. La sintaxis básica es:

```
cin >> variable;
```

Donde variable es el identificador de la variable en la que se almacena lo escrito mediante el teclado. Veamos un ejemplo.

```
#include <iostream.h>

int main (void)
{
   int numero;

   /* Pedimos al usuario que escriba un mensaje. */
   cout << "Escriba un numero;
   /* El numero que escriba el usuario lo almacenamos en la
    variable numero. */
    cin >> numero;

   /* Mostramos un mensaje indicando cual es su doble. */
   cout << "\nEl doble de " << numero << " es " << (numero*2);

   /* Hacemos una pausa hasta que el usuario pulse Intro */
   fflush(stdin);
   cout << "\nPulse Intro para finalizar...";
   cin.get() ;
}
```

Cuando la ejecución llega a la línea:

```
cin >> numero;
```

aparece en pantalla el cursor, una raya parpadeando. Esto indica que el programa está a la espera de que escribamos algún número, por ejemplo el 10.

El número escrito (10) aparece en pantalla, luego es "con eco", y el cursor sigue parpadeando. Hasta que no pulsemos la tecla **Intro** el programa no continuará, luego "sí espera Intro". Pulsamos **Intro.** A continuación, la variable numero toma el valor del número escrito, es decir, la variable numero almacena el valor 10. La línea:

```
cout << "\nEl doble de " << numero << " es " << (numero*2);
```

mostrará en pantalla el mensaje: "El doble de 10 es 20". El resultado en la pantalla es igual al de la anterior figura 8.12.
También se pueden leer datos del teclado de manera encadena, pero esto es una mala práctica. La idea es que antes de cada lectura del teclado exista un cout que muestre en pantalla información acerca de la lectura. Lo siguiente sería correcto:

```
cout << "Escriba un numero par: ";
cin >> numero_par;
cout << "Escriba un carácter: ";
cin >> letra;
```

Lo siguiente no sería muy correcto:

```
cout << "Escriba un numero par y un carácter: ";
cin >> numero_par;
cin >> letra;
```

La versión poco correcta no es aconsejable, porque en la práctica el usuario puede llegar a tener la sensación de que el programa falla. ¿Por qué? Pruebe con un programa que contenga la versión poco correcta y deje que lo ejecute otra persona sin decirle nada.

Advertencia:

En C++ *también puede ocurrimos que tras ejecutar una lectura del teclado con cin, el* buffer *del teclado siga contiendo información. Esto produce que la próxima vez que se ejecute cin se recojan esos datos del* buffer, *algo que normalmente no se desea. Si en el* buffer *del teclado quedase almacenada la pulsación de la tecla* Intro, *el próximo cin se saltaría automáticamente, ya que el ordenador interpretará que nosotros hemos pidsado dicha tecla. Para evitar estos casos, podemos limpiar el* buffer *llamando a* `fflush(stdin)` *; tras el uso de cin.*

Hagamos un último programa que consista en pedir al usuario un número A y un número B, y muestre el resultado de la suma en pantalla.

```
#include <iostream.h>

int main (void)
{
   int num_A, num_B;

   /* Pedimos al usuario que escriba el numero A. */
   cout << "Escriba un numero: " ;
   /* Almacenamos el numero en la variable num_A. */
   cin >> num_A;
   /* Pedimos al usuario que escriba el numero B. */
   cout « "Escriba otro numero: ";
   /* Almacenamos el numero en la variable num_B. */
   cin >> num_B;
   /* Mostramos el resultado de sumar los dos números. */
   cout << "\n" << num_A << " + " << num_B << " = " << (num_A + num_B);

   /* Hacemos una pausa hasta que el usuario pulse Intro */
   fflush(stdin);
   cout << "\nPulse Intro para finalizar...";
   cin.get();
}
```

En la figura 8.15 se muestra el resultado del programa para el caso de escribir los números 7 y 3.

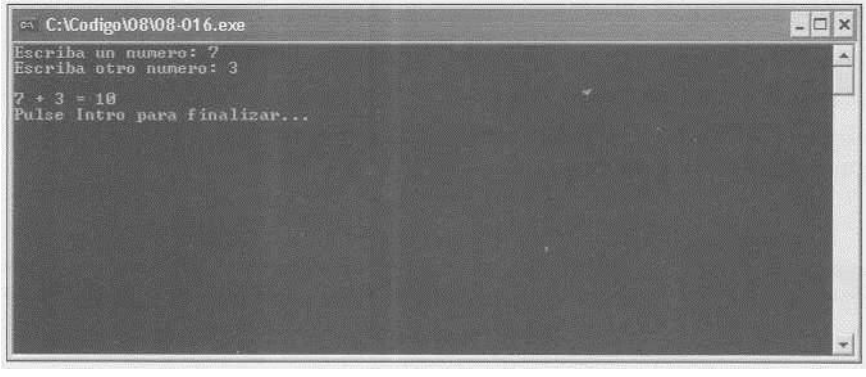


Figura 8.15. Resultado del programa que suma dos números.

Ejercicios resueltos

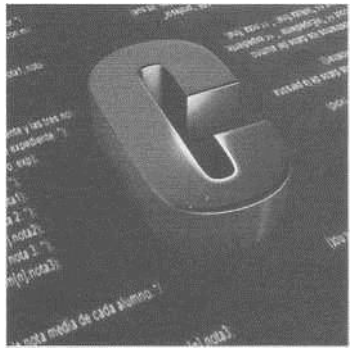
Intente realizar los siguientes ejercicios. Encontrará la solución al final de este libro.

1. Realice un programa que pida una letra al usuario y muestre el carácter siguiente en la tabla ASCII.

2. Escriba un programa que pida un número al usuario y muestre su cuadrado.
3. Escriba un programa que pida dos números al usuario y muestre el resto de dividir uno entre el otro.

Resumen

Ahora que sabemos mostrar datos en la pantalla podemos hacer nuestros programas más vistosos. Recoger en variables datos escritos mediante el teclado nos da la posibilidad de hacer programas en función de esos datos.



Capítulo 9

Control del flujo

En este capítulo aprenderá:

- Las sentencias cíe control de flujo de C/C++.
- A ejecutar un código u otro según una condición.
- A ejecutar un bloque de código un número de veces
- A elegir la sentencia de control de flujo más adecuada.

Introducción

Una de las mayores necesidades que se nos plantea ahora es la de controlar el flujo del programa. Al "yo quiero que se ejecute este bloque de código sólo cuando ocurra tal cosa", o al "yo quiero que se ejecute este otro bloque de código mientras no se cumpla tal condición", les daremos solución en este capítulo. Pero ¿qué es un bloque de código? Un bloque de código puede estar formado por una única sentencia escrita entre llaves o no, como:

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

o:

```
{
    x = x + 1;
}
```

Un bloque de código también puede estar formado por varias sentencias, en cuyo caso van siempre entre llaves:

```
{
    y = y +3;
    printf ("El valor de y es: %d", y);
}
```

Nota:

A la hora de escribir un bloque, ante la duda de si poner o no las llaves, es siempre aconsejable ponerlas, pues en cualquier caso siempre aportarán claridad al código por delimitar visualmente los bloques.

Las sentencias de control de flujo nos ofrecen una gran cantidad de posibilidades para realizar programas. Pero no nos demoremos más y comencemos.

Sentencias condicionales

Las sentencias condicionales son las que nos permitirán ejecutar un bloque de código o no, o ejecutar un bloque de código u otro. Planteemos un ejemplo de la vida cotidiana. ¿Todos los días salimos a la calle con el paraguas? No. Y ¿qué algoritmo utilizamos para coger el paraguas o no? Podría ser éste:

```
Si está lloviendo, entonces
cogemos el paraguas,
no,
    no cogemos el paraguas.
```

Considerando que por defecto no tenemos el paraguas en nuestra mano, sino en el paragüero, podría simplificarse:

```
Si está lloviendo, entonces
   cogemos el paraguas.
```

Pero ¿cómo podemos hacer estas condiciones en nuestros programas? Pues con las sentencias condicionales: `if, if-else, switch.`

Sentencia if

La sentencia condicional if (o "si") nos permite ejecutar un bloque de código sólo si se cumple una condición. Su sintaxis es:

```
if (condición)
   bloque_de_codigo
```

Donde `condición` es una expresión lógica que obligatoriamente debe escribirse entre paréntesis y `bloque_de_codigo` puede ser una única sentencia o varias, como ya hemos explicado. Si la condición tiene como resultado el valor verdadero, entonces se ejecuta el bloque de código y, a continuación, el programa continúa con el código que hubiese debajo de la sentencia `if`. Si la condición tiene como resultado el valor falso, el bloque de código no se ejecuta y el programa continuará con el código que hubiese debajo de la sentencia `if`.
En la figura 9.1 podemos ver un ordinograma que representa a la sentencia `if` y que nos puede ayudar a comprender mejor su funcionamiento.

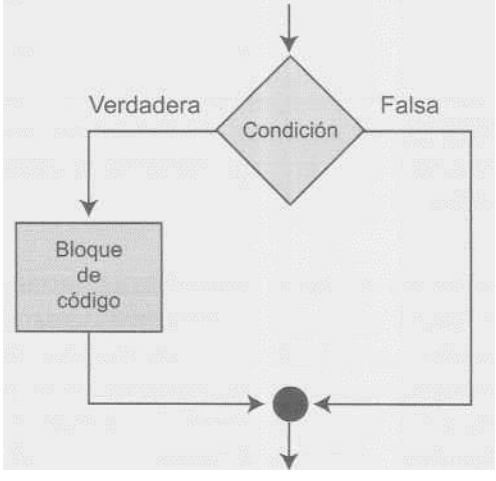


Figura 9.1. Ordinograma de la sentencia if.

Las flechas del ordinograma representan el flujo del programa. Cuando se llega a la condición el flujo se divide en dos ramas, pero sólo se puede tomar una de las dos, aquella cuyo título (Verdadera/Falsa) coincida con el resultado de la condición. Como puede observarse, sólo cuando la condición es verdadera se toma el camino que nos lleva al bloque de código. Cuando la condición se evalúa como falsa se toma un camino en el que no hay nada. Tanto si la condición es verdadera como si es falsa, se llega a un mismo punto: el final de la sentencia `if` y el comienzo de otra línea de código.
Una sentencia `if` la podemos leer así: si la condición es verdadera, entonces se ejecuta el bloque de código.
Lo mejor será poner algunos ejemplos y comentarlos.

```
#include <stdio.h>
int main (void)
{
   int numero;
   printf("Escriba un numero: ");
   scanf("%d",&numero);
   if (numero >= 0)
     printf ("El numero escrito es positivo");
   /* Esta linea ya esta fuera de la sentencia if.*/
   printf ("\nAdios");
   /* Hacemos una pausa hasta que el usuario pulse Intro */
   fflush(stdin);
   printf("\nPulse Intro para finalizar...");
   getchar();
}
```

Nota:

Tras cada sentencia añadimos un comentario que indica que ya se está fuera de la sentencia con la intención de quedar claro dónde termina y evitar confusiones.

Hagamos una traza para comprobar qué ocurre cuando se ejecuta este programa. Lo primero con lo que nos encontramos dentro de la función `main` es la declaración de una variable de tipo entero llamada `numero`. Seguidamente, mediante la función `printf`, se muestra en la pantalla un mensaje que invita al usuario a escribir un número. En la siguiente línea, la función `scanf` se encarga de almacenar en la variable `numero` el número escrito por el usuario. Si suponemos que escribió 50, ahora la variable `numero` almacena

el valor 50. Llegamos a la sentencia `if`. En primer lugar se evalúa la condición: ¿el valor de la variable `numero` es mayor o igual que cero?, es decir, ¿50 es mayor o igual que 0? Sí. El resultado es verdadero, luego se ejecuta el bloque de código de la sentencia `if`, que contiene una única llamada a la función: `printf`. Así que se muestra en pantalla un mensaje que indica que el número que escribió el usuario es positivo, lo cual es cierto. Por último, y fuera de la sentencia `if`, se ejecutaría la llamada a la función `printf` que muestra un mensaje de despedida.
¿Y si el número que el usuario escribió hubiese sido el -10? Pues al llegar a la sentencia `if` y evaluar la condición el resultado habría sido falso, porque ¿-10 es mayor o igual que 0? No. Luego, al ser evaluada la condición como falsa no se ejecutaría el bloque de código de la sentencia `if`. La ejecución del programa continuaría con las líneas de código que hubiese tras la sentencia `if`, que en este caso es la llamada a la función `printf` que muestra un mensaje de despedida.
La condición del ejemplo anterior es muy simple, pero se pueden hacer condiciones con subcondiciones:

```
#include <stdio.h>

int main (void)
{
   int numero;

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

   if ( (numero >= 0) && (numero <= 9) )
      printf ("El numero escrito tiene un solo digito");

   /* Esta linea ya esta fuera de la sentencia if. */
   printf ("\nAdios");
   /* Hacemos una pausa hasta que el usuario pulse Intro */
   fflush(stdin);
   printf("\nPulse Intro para finalizar...");
   getchar();
}
```

Este programa solicita al usuario un número y mediante una condición se evalúa si se trata de un número "mayor o igual que" 0 y "menor o igual que" 9. Tan sólo en el caso de que la condición se evalúe como verdadera se muestra un mensaje en la pantalla que indica que el número que escribió el usuario sólo tiene un dígito.
Estudiemos la condición, que tiene dos subcondiciones. Supongamos que el usuario escribió el número 10. Primero se resuelven las condiciones más

internas indicadas por los paréntesis. ¿El valor de la variable `numero` es mayor o igual que 0?, es decir, ¿10 es mayor o igual que 0? Sí, luego la primera subcondición es verdadera. ¿10 es menor o igual que 9? No, luego la segunda subcondición es falsa. Según las tablas de verdad, verdadero AND (&&) falso es igual a falso. El resultado de la condición, para este ejemplo, es falso, por lo que no se ejecutaría el bloque de código de la sentencia `if`.
Si el usuario hubiese escrito el número 5, el resultado de la condición hubiese sido verdadero, ejecutándose el bloque de código de la sentencia

Sentencia if-else

La sentencia `if`, realmente, es la misma que la sentencia `if-else` (o bien "si- si no"), pero las explicamos por separado para introducir los conceptos poco a poco y cuando nos hacen falta. En los ejemplos de la sentencia if cuando la condición era verdadera se ejecutaba el bloque de código, pero cuando era falsa no se ejecutaba ningún otro código alternativo. Esto es, en el ejemplo que pedía un número al usuario para indicar si era positivo (>=0), cuando se tecleaba un número negativo no se mostraba ningún mensaje en la pantalla. Realmente, el programa estaría más completo si en el caso de que el número no fuese positivo mostrase un mensaje indicando que es negativo. Para ello utilizamos la sentencia `if-else`. Su sintaxis es:

```
if (condición)
   bloque_de_codigo_1
else
   bloque_de_codigo_2
```

Si la condición es verdadera se ejecuta el `bloque_de_codigo_1`. Si la condición es falsa se ejecuta el `bloque_de_codigo_2`. Siempre se ejecutará uno de los dos bloques, pero no los dos a la vez: o se ejecuta un bloque, o se ejecuta el otro. En la figura 9.2 vemos el ordinograma que representa a la sentencia `if-else`.
La explicación del ordinograma de la sentencia `if-else` es la misma que la de la sentencia `i f`, a diferencia de que en el camino que corresponde a la condición evaluada como falsa hay otro bloque de código. Recuerde que sólo se puede tomar un camino en el flujo del programa: el de la condición verdadera o el de la condición falsa; aunque en cualquier caso finalizan en el mismo punto.
Una sentencia `if-else` la podemos leer así: si la condición es verdadera entonces se ejecuta el bloque de código 1, si no, se ejecuta el bloque de código 2.

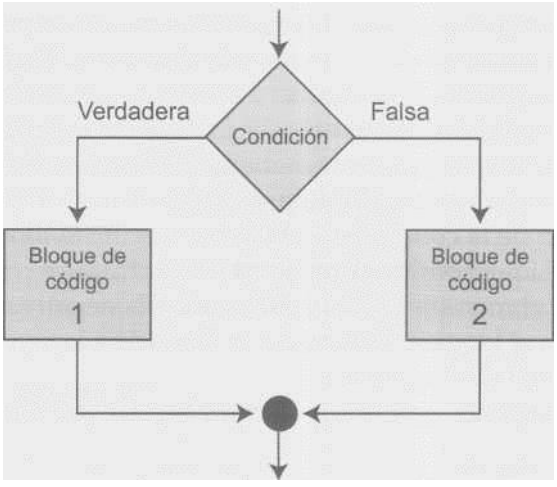


Figura 9.2. Ordinograma de la sentencia if-else.

Así, el programa que acabamos de comentar quedaría:

```
#include <stdio.h>

int main (void)
{
    int numero;

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

    if (numero >= 0)
       printf ("El numero escrito es positivo");
    else
       printf ("El numero escrito es negativo");

    /* Esta linea ya esta fuera de la sentencia if-else. */
    printf ("\nAdios");

    /* Hacemos una pausa hasta que el usuario pulse Intro */
    fflush(stdin);
    printf("\nPulse Intro para finalizar...");
    getchar();
}
```

Escriba el número que escriba el usuario el programa mostrará un mensaje indicando si es positivo o negativo. Es muy importante informar correctamente y siempre que sea necesario al usuario. Hagamos una traza a partir de la condición. Supongamos que el usuario escribe el número 10, así que la variable numero almacena el valor 10. ¿El valor de la variable numero es mayor o igual que 0?, es decir, ¿10 es mayor o igual que 0? Sí, luego el

resultado de la condición es verdadero, se ejecuta el bloque de código de la sentencia `if,` que contiene una llamada a la función `printf` que nos muestra en la pantalla el mensaje "El numero escrito es positivo" y, como no hay más sentencias en el bloque, la sentencia `i f` finaliza y llegamos a la llamada a la función `printf` que muestra un mensaje de despedida. Ahora, supongamos que el usuario escribe el número -5. ¿-5 es mayor o igual que 0? No, luego el resultado de la condición es falso, se ejecuta el bloque de código de la sentencia `el se,` que contiene una llamada a la función `printf` que muestra en la pantalla el mensaje "El numero escrito es negativo" y, como no hay más sentencias en el bloque, llegamos a la llamada a la función `printf` que muestra un mensaje de despedida.
Vamos ahora a analizar otro ejemplo con bloques que tengan más de una sentencia.

```
#include <stdio.h>

int main (void)
{
   int clave;

   printf("Escriba el numero de la contraseña: ");
   scanf("%d", &clave);

   if (clave == 123)
   {
      printf ("El numero de la contraseña es correcto.\n");
      printf ("Buenos dias.");
   }
   else
   {
      printf ("El numero de la contraseña es incorrecto.\n");
   }

   /* Esta linea ya esta fuera de la sentencia if-else. */

   /* Hacemos una pausa hasta que el usuario pulse Intro */
   fflush(stdin);
   printf ("\nPulse Intro para finalizar...");
   getchar ();
}
```

Este programa comienza solicitando al usuario que escriba un número que se corresponde con una contraseña. Supongamos que el usuario escribe el número 123, así que la variable `clave` almacena el valor 123. Llegamos a la condición. ¿El valor de la variable `clave` es igual a 123?, es decir, ¿123 es igual a 123? Sí, luego la condición es verdadera. Se ejecuta cada una de las dos llamadas a la función `printf` del bloque de la sentencia `if.` Posteriormente se ejecuta el código que haya tras la sentencia `if-else,` pero como

no hay nada más el programa finaliza. Ahora, supongamos que el usuario escribe el número 10, así que la variable `clave` almacena el valor 10. Llegamos a la condición. ¿El valor de la variable `clave` es igual a 123?, es decir, ¿10 es igual a 123? No, luego la condición es falsa. Se ejecuta la única llamada a la función `printf` del bloque de la sentencia `else`. Finalmente, se ejecuta el código que haya tras la sentencia `if-else`, pero como no hay nada más el programa finaliza.
En este programa hemos incluido un ejemplo de un bloque con una sola sentencia y con llaves, las cuales se podrían haber suprimido.

Sentencia switch

La sentencia `switch` nos permite, según el valor de una variable, ejecutar un código, u otro, ..., u otro, u otro por defecto, de una manera muy clara. Su sintaxis es:

```
switch (variable)
{
   case valor_1:  bloque_de_codigo_1
         break;

   case valor_2:  bloque_de_codigo_2
         break;

   case valor_n:  bloque_de_codigo_n
         break;


   default: bloque_de_codigo_por_defecto
}
```

Donde `variable` es la variable cuyo valor queremos evaluar y es obligatorio escribirla entre paréntesis. `valor_1, valor_2,..., valoran` son los valores que puede tomar la variable y según los cuales se ejecuta un bloque de código u otro. Estos valores son los que nosotros queramos indicar.
El funcionamiento de la sentencia `switch` consiste en comparar el valor de la variable con los especificados tras la palabra reservada `case`. En caso de que sea igual a uno de ellos se ejecuta el bloque de código situado después los dos puntos, tras lo cual se ejecuta la instrucción `break`, que hace finalizar la sentencia `switch`, continuando el flujo de datos con el código que haya tras esta sentencia. Si el valor que almacena la variable no coincide con ninguno de los valores que se especifican tras la palabra reservada `case`, entonces se ejecuta el bloque de código por defecto, que es el situado a continuación de la palabra reservada `default` y, tras el cual, la sentencia `switch` finaliza. La cláusula `default` es opcional, por lo que si no quere-

mos que se ejecute ningún código para el caso de que el valor de la variable no coincida con ninguno de los indicados tras las palabras reservadas `case`, podemos no escribir la palabra `def ault` (ni su bloque de código por defecto, claro).
En la figura 9.3 se muestra el ordinograma correspondiente a la sentencia

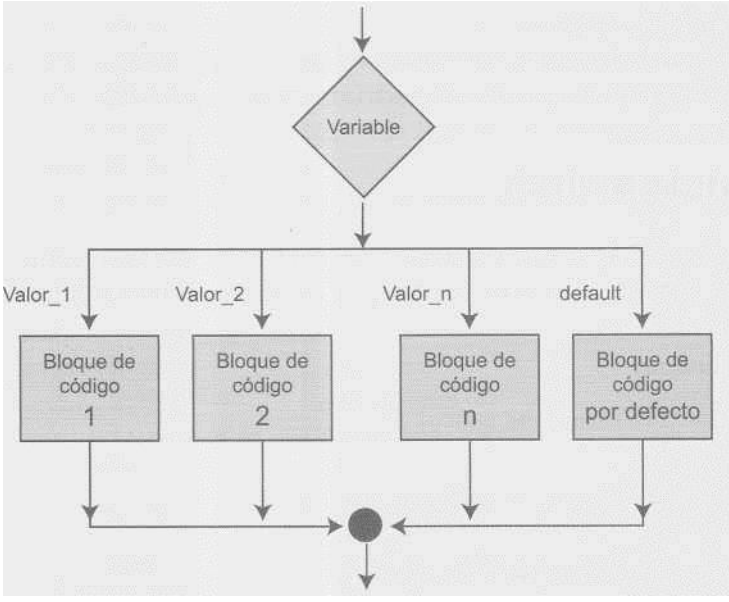


Figura 9.3. Ordinograma de la sentencia switch.

Una sentencia `switch` la podemos leer de la siguiente forma: en el caso de que la variable almacene el `valor_1`, entonces se ejecuta el `bloque_de_codigo_1;` en el caso de que la variable valga `valor_2`, entonces se ejecuta el `bloque_de_codigo_2;` en el caso de que la variable almacene el `valor_n`, entonces se ejecuta el `bloque_de_codigo_n;` en el caso de que el valor de la variable no coincida con ninguno de los anteriores, entonces se ejecuta el `bloque_de_codigo_por_defecto.`
Veamos un ejemplo.

```
#include <stdio.h>

int main (void)
{
   char vocal;

   /* Pedimos una vocal al usuario. */
   printf ("Escriba en minúscula una vocal: ");
   scanf ("%c", &vocal);
```

```
    /* Según la vocal mostramos un mensaje u otro. */
    switch (vocal)
    {
       case 'a':   printf ("Ha escrito la vocal: a.");
                      break;
       case 'e':   printf ("Ha escrito la vocal: e.");
                      break;
       case 'i':   printf ("Ha escrito la vocal: i.");
                      break;
       case 'o':   printf ("Ha escrito la vocal: o.");
                      break;
       case 'u':   printf ("Ha escrito la vocal: u.");
                      break;
       default: printf ("No ha escrito una vocal minúscula.");
    }
    /* Esta linea ya esta fuera de la sentencia switch. */

    /* Hacemos una pausa hasta que el usuario pulse Intro */
    fflush(stdin);
    printf("\nPulse Intro para finalizar...");
    getchar ();
}
```

Este programa solicita al usuario que escriba una vocal en minúscula y, tras almacenarla en la variable vocal mediante la función scanf, llegamos a la sentencia switch. La sentencia switch compara el valor de la variable vocal con los que se indican tras la palabra reservada case. Supongamos que el usuario escribió la letra 'i'. ¿Se hace referencia al valor 'i' tras alguna de las palabras reservadas case? Sí; entonces se ejecuta el código que se encuentra tras los dos puntos de case ' i ':

```
printf ("Ha escrito la vocal: i.");
break;
```

Cuando se ejecuta la instrucción break el flujo del programa sale inmediatamente de la sentencia switch, pero como no hay nada más el programa finaliza.
Ahora supongamos que el usuario hubiese escrito la letra 'X'. ¿Se hace referencia al valor 'X' tras alguna de las palabras reservadas case? No, luego se ejecuta el código que hay tras los dos puntos de default:

```
printf ("No ha escrito una vocal minúscula.");
```

Después, se sale de la sentencia switch.
La sentencia switch suele usarse habitualmente para gestionar menús en los que se muestran las distintas opciones de las que disponen, se pide al usuario que elija una y se ejecuta el código correspondiente a esa opción. En la figura 9.4 podemos ver el aspecto que toma en pantalla un programa de este tipo cuando se ejecuta.

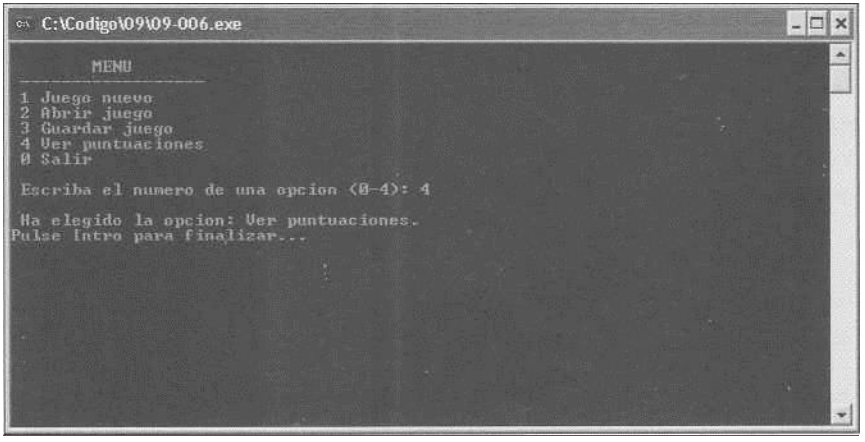


Figura 9.4. Ejemplo de programa con menú.

El código de este programa es el siguiente.

```
#include <stdio.h>

int main (void)
{
   int opcion;

   printf("\n\tMENU\n");
   printf ( " --------------- \n" ) ;
   printf(" 1 Juego nuevo\n");
   printf(" 2 Abrir juego\n");
   printf(" 3 Guardar juego\n") ;
   printf(" 4 Ver puntuaciones\n");
   printf(" 0 Salir\n");
   printf("\n");
   printf(" Escriba el numero de una opcion (0-4): ") ;
   scanf("%d", &opcion);

   switch (opcion)
   {
      case 1:  printf("\n Ha elegido la opcion: Juego nuevo.");
         break;

      case 2:  printf("\n Ha elegido la opcion: Abrir juego.");
         break;

      case 3:  printf("\n Ha elegido la opcion: Guardar juego.");
         break;

      case 4:  printf("\n Ha elegido la opcion: Ver puntuaciones.");
         break;

      case 0:  printf("\n Ha elegido la opcion: Salir.");
         break;
```

```
      default: printf("\n Ha elegido una opcion no valida.");
   }

   /* Esta linea ya esta fuera de la sentencia switch. */

   /* Hacemos una pausa hasta que el usuario pulse Intro */
   fflush(stdin);
   printf ("\nPulse Intro para finalizar...");
   getchar ();
}
```

Dentro de la función `main`, lo primero que encontramos es la declaración de una variable de tipo entero llamada `opcion`, que almacenará el número de la opción elegida del menú. A continuación, una serie de llamadas a la función `printf` permiten mostrar en pantalla el menú que veíamos en la figura 9.4 y pedir al usuario que escriba el número de la opción que desea elegir. Mediante la función `scanf` almacenamos en la variable `opcion` el número de la opción escrita. La sentencia `switch` se encarga de mostrar por pantalla un mensaje u otro en función del valor de la variable `opcion`. Si el usuario escribió el número 4 la variable `opcion` almacena este valor. La sentencia `switch` comprueba si se ha indicado este valor tras alguna de las palabras `case`. Al ser así, se ejecuta el código que haya tras *`case 4`*:

```
printf("\n Ha elegido la opcion: Ver puntuaciones.");
break;
```

Gracias a la instrucción `break` la sentencia `switch` finaliza y tras ella el programa.

¿Y si el usuario hubiese escrito el numero de una opción que no existe, por ejemplo el 10? Pues, al llegar la ejecución del programa a la sentencia `switch` y comprobar que no se ha especificado el valor 10 tras ninguna palabra reservada `case`, se ejecutaría el código de la cláusula `default`:

```
printf("\n Ha elegido una opcion no valida.");
```

A continuación la sentencia `switch` finalizaría y tras ella el programa. Puede intentar ejecutar este mismo código sin la cláusula `default` (y sin su bloque de código, claro). El resultado sería para el caso de introducir un número de una opción no especificada que el programa finalizaría sin mostrar ningún mensaje. La figura 9.5 nos muestra el resultado de ejecutar el programa anterior escribiendo el número de una opción inválida y con la cláusula `default`.

La figura 9.6 muestra el mismo caso, pero sin la cláusula default.

Estamos ante un mismo programa que tras realizar un operación errónea en un caso se informa de ella al usuario (figura 9.5) y en otro no (figura 9.6). Al informar al usuario de los errores conseguimos que éste sepa qué no ha

hecho bien y cómo remediarlo. Si no le informamos pensará que el programa ha sufrido un fallo, que es de muy mala calidad y lo tirará a la papelera (y con él nuestra fama como programador). Así que lo correcto en este caso es incluir la cláusula `default`.

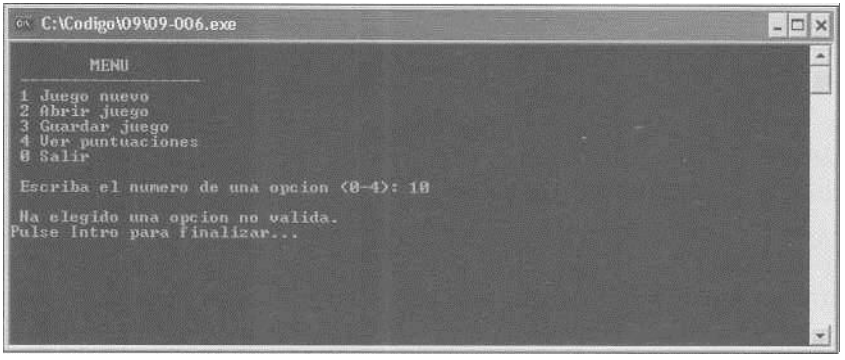


Figura 9.5. Resultado con la cláusula default.

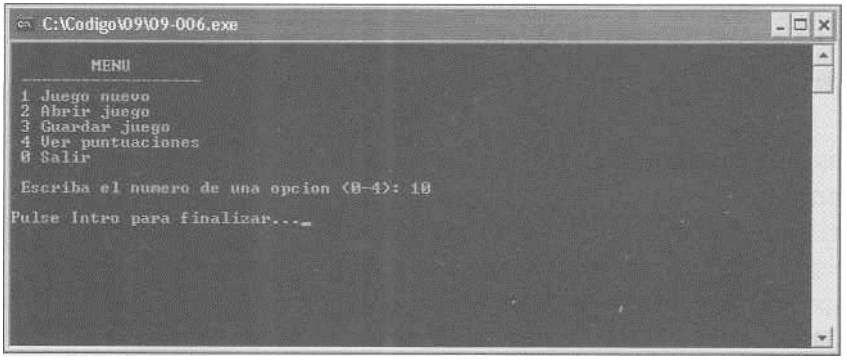


Figura 9.6. Resultado sin la cláusula default.

Sentencias repetitivas

En muchas ocasiones, querremos que se repita un bloque de código mientras se den una serie de circunstancias. A cada repetición se suele llamar iteración. Es el caso de un programa que pide una clave al usuario. Mientras no se introduzca la clave correcta el programa no deja de ejecutar una y otra vez el bloque de código encargado de solicitar la clave. O imaginemos que tenemos que hacer un programa que muestre en pantalla cien veces la palabra "Hola", ¿escribimos cien **printf** ("**Hola**") ; ? No, usamos las sentencias repetitivas,

que son: while, do-while y f or, y a las cuales también nos referimos como bucles, porque permite repetir una y otra vez un bloque de código.

Sentencia while

La sentencia while (o "mientras") permite ejecutar un bloque de código mientras se cumpla una condición. Su sintaxis es:

```
while (condición)
   bloque_de_codigo
```

Donde condición es una expresión lógica que tendrá como resultado verdadero o falso. La sentencia while se leería así: mientras la condición sea verdadera se ejecuta el bloque de código.
El funcionamiento de la sentencia while consiste en evaluar en primer lugar la condición. Si el resultado de ésta es verdadero se ejecuta el bloque de código. A continuación se vuelve a evaluar la condición. Si vuelve a ser verdadera se ejecuta nuevamente el bloque de código y seguidamente se evalúa otra vez la condición, y así sucesivamente. Cuando el resultado de la condición es falso no se ejecuta el bloque de código y la sentencia while finaliza, continuando la ejecución con el código que hubiese después de esta sentencia. Es decir, mientras la condición sea evaluada como verdadera el bloque de código se ejecuta una y otra vez. Podemos ver esta explicación representada mediante el ordinograma de la figura 9.7.

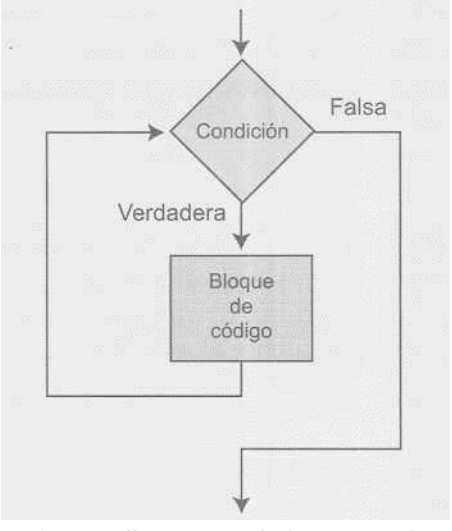


Figura 9.7. Ordinograma de la sentencia while.

Si la primera vez que se evalúa la condición resulta falsa, el bloque de código no se ejecutará ninguna vez.
La sentencia while se utiliza cuando antes de ejecutar un bloque de código (quizás ninguna vez, quizás sólo una vez, quizás varias veces) hay que realizar algún tipo de comprobación mediante la condición.
Veamos un ejemplo. Supongamos que queremos mostrar por pantalla los números del 1 al 6 (usando una variable que vaya tomando dichos valores). Hasta este momento lo habríamos hecho así:

```
#include <stdio.h>

int main (void)
{

   int numero;

   numero = 1;
   printf ("%d\n", numero);      /* Muestra el 1. */
   numero++;                     /* o: numero = numero + 1; */
   printf ("%d\n", numero);      /* Muestra el 2. */
   numero++;
   printf ("%d\n", numero);      /* Muestra el 3. */
   numero++;
   printf ("%d\n", numero);      /* Muestra el 4. */
   numero++;
   printf ("%d\n", numero);      /* Muestra el 5. */
   numero++;
   printf ("%d\n", numero);      /* Muestra el 6. */


   /* Hacemos una pausa hasta que el usuario pulse Intro */
   fflush(stdin);
   printf("\nPulse Intro para finalizar...");
   getchar ();
}
```

Si nos fijamos, hay dos líneas de código que continuamente se están repitiendo, son éstas:

```
printf ("%d\n", numero);
numero++;
```

Estas dos líneas las podemos poner dentro de una sentencia while para que se repitan automáticamente mientras el valor de la variable numero sea menor o igual que 6, que es el último número que queremos mostrar por pantalla. El siguiente programa es equivalente al anterior.

```
#include <stdio.h>

int main (void)
{
   int numero;
   numero = 1;
   while
   (numero<=6
   )
```

```
    {
       printf ("%d\n", numero);
       numero++;
    }

    /* Esta linea ya esta fuera de la sentencia while. */

    /* Hacemos una pausa hasta que el usuario pulse Intro */
    fflush(stdin);
    printf("\nPulse Intro para finalizar...");
    getchar();
}
```

Vamos a analizarlo.

```
int numero;
```

Esta primera línea de código, dentro de la función `main`, crea una variable de tipo entero llamada `numero`. Será la que vaya tomando los valores del 1 al 6.

```
numero = 1;
```

La asignación del valor 1 a la variable `numero` nos sitúa al principio de la cuenta hasta el 6.

```
while (numero<=6)
{
   printf ("%d\n", numero); numero++;
}
```

En la sentencia `while` se evalúa la condición en primer lugar. ¿El valor de la variable `numero` es menor o igual que 6?, es decir, ¿1 es menor o igual que 6? Sí, luego se ejecuta el bloque de código. Se muestra el valor de la variable `numero` mediante la función `printf`, tras lo cual se incrementa el valor de `numero`. Ahora `numero` almacena el valor 2. Volvemos a la condición. ¿El valor de la variable `numero` es menor o igual que 6?, es decir, ¿2 es menor o igual que 6? Sí, luego... (y así sucesivamente).

De esta manera llegamos al momento en el que la variable `numero` almacena el valor 5, que se muestra en pantalla y, a continuación, se incrementa su valor, siendo ahora 6. Volvemos a la condición. ¿6 es menor o igual que 6? Sí. Se muestra en pantalla el numero 6 y se incrementa el valor de la variable `numero`, siendo ahora 7. Volvemos a la condición. ¿7 es menor o igual que 6? No, luego salimos de la sentencia `while`, porque la condición es falsa. El programa finaliza.

No nos debemos preocupar porque la variable `numero` haya tomado el valor 7, como se puede comprobar en la figura 9.8 no se muestra este número en pantalla.

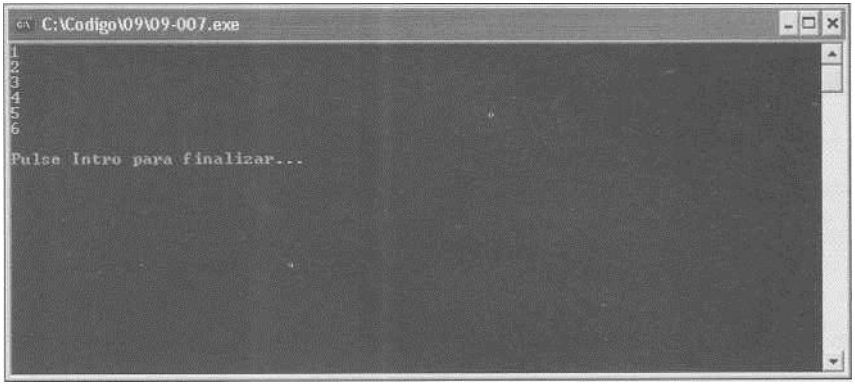


Figura 9.8. Resultado de mostrar los números del 1 al 6.

También podemos utilizar la sentencia while para mostrar los números impares del 1 al 10. Para ello incrementaremos el valor de la variable numero de dos en dos y la condición de la sentencia while será: numero<=10.

```
#include <stdio.h>

int main (void)
{
    int numero;

    numero = 1;
    while (numero<=10)
    {
       printf ("%d\n", numero);
       numero = numero + 2;
    }

    /* Esta linea ya esta fuera de la sentencia while. */

    /* Hacemos una pausa hasta que el usuario pulse Intro */
    fflush(stdin);
    printf("\nPulse Intro para finalizar...");
    getchar();
}
```

Ya hemos hecho referencia en algún momento a un programa que solicita una clave. La idea es pedir una clave numérica al usuario, y si no es igual que la especificada en nuestro programa se la vuelve a pedir. El programa no finaliza hasta que se acierta la contraseña, que es 123. El código sería éste:

```
#include <stdio.h>
#define CLAVE 123
int main (void)
{
   int numero;
```

```
    printf ("Escriba la contraseña: ") ;
    scanf ("%d", ¿¿numero);

    while (numero != CLAVE)
    {
       printf ("La contraseña es incorrecta.\n");
       printf ("Escriba la contraseña: ");
       scanf ("%d", &numero);

    }

    /* Esta linea ya esta fuera de la sentencia while. */

    printf ("La contraseña es correcta");
    /* Hacemos una pausa hasta que el usuario pulse Intro */
    fflush(stdin);
    printf ("\nPulse Intro para finalizar...");
    getchar ();
}
```

La línea:

```
#define CLAVE 123
```

declara la constante que representa a la contraseña que el usuario debe adivinar (123). Recordemos que también podríamos haber declarado esta constante mediante la palabra reservada `const`.

```
int numero;
```

Esta línea declara la variable de tipo entero, llamada `numero`, que almacenará la contraseña que el usuario introduzca.

```
printf ("Escriba la contraseña: ")
scanf ("%d", &numero) ;
```

Estas dos líneas se encargan de pedir una contraseña al usuario, la cual se almacena en la variable `numero`.

```
while (numero != CLAVE)
```

La condición de la sentencia `while` comprueba si la contraseña que escribió el usuario (almacenada en la variable `numero`) y la contraseña correcta (**CLAVE**, cuyo valor es 123) son distintas. Si sí son distintas entonces la condición se evalúa como verdadera y se ejecuta el bloque de código.

```
printf ("La contraseña es incorrecta.\n")
printf ("Escriba la contraseña: ");
scanf ("%d", &numero) ;
```

Este bloque muestra en pantalla un mensaje indicando que la contraseña es incorrecta y se pide nuevamente la contraseña al usuario. Después, se vuelve a evaluar la condición.

Si esta vez la contraseña que escribió el usuario es igual a 123, la condición `(numero!= CLAVE)` sería equivalente a ¿123 es distinto de 123? No, luego la condición es falsa y la sentencia `while` finalizaría, ejecutándose la siguiente línea de código que hubiese tras la sentencia `while`:

```
printf ("La contraseña es correcta");
```

la cual se encarga de mostrar un mensaje en la pantalla que nos informa del acierto.

Sentencia do-while

La sentencia **do-while** (o "hacer-mientras") también permite ejecutar un bloque de código mientras se cumpla una condición, pero su funcionamiento y sintaxis difieren un poco de la del **while:**

```
do
   bloque_de_codigo
while (condición);
```

Advertencia:

Observe que en la sentencia `do-while` se escribe un punto y coma (;) tras la condición, pero que en la sentencia `while` este punto y coma no se escribe.

La sentencia `do-while` se leería así: hacer (o ejecutar) el bloque de código mientras la condición sea verdadera.
El funcionamiento de la sentencia **do-while** consiste en ejecutar en primer lugar el bloque de código. A continuación, se evalúa la condición. Si el resultado de la condición es verdadero se vuelve a ejecutar el bloque de código y, después, otra vez la condición. Si vuelve a ser verdadera se ejecuta nuevamente el bloque de código y seguidamente se evalúa otra vez la condición. Cuando el resultado de la condición es falso no se ejecuta el bloque de código y la sentencia **do-while** finaliza, continuando la ejecución del programa con el código que hubiese después de esta sentencia.
En un principio puede parecer muy similar a la sentencia **while**, pero su principal diferencia es que si en la sentencia **do-while** la primera vez que se evalúa la condición resulta ser falsa, el bloque de código se habrá ejecutado una vez, porque en la sentencia **do-while** primero se ejecuta el bloque y después la condición. Mientras que si en la sentencia **while** la condición es falsa la primera vez, el bloque de código no se ejecutará ninguna vez, por-

que en la sentencia `while` primero se evalúa la condición. Es decir, el bloque de código de la sentencia `do-while` se ejecuta al menos una vez y en la sentencia `while` puede no ejecutarse nunca.
El ordinograma de la figura 9.9 representa a la sentencia `do-while`.

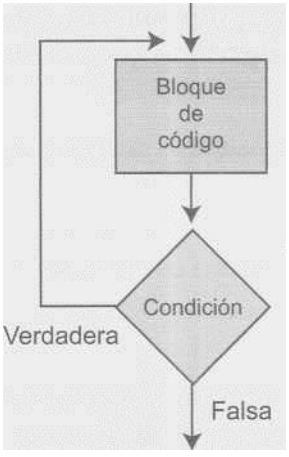


Figura 9.9. Ordinograma de la sentencia do-while.

La sentencia `do-while` se usa cuando primero debe ejecutarse un bloque de código (al menos una vez, quizás varias veces) y después hay que realizar algún tipo de comprobación mediante la condición.
Hemos visto con la sentencia `while` un programa que pide una clave al usuario y que hasta que no se acierta el programa no finaliza. Hagámoslo ahora con la sentencia `do-while`.

```
#include <stdio.h>

const int CLAVE = 123;

int main (void)
{
    int numero;

    do
    {
       printf ("Escriba la contraseña: ");
       scanf ("%d", &numero);
    }

    while (numero != CLAVE);

    /* Esta linea ya esta fuera de la sentencia do-while. */

    printf ("La contraseña es correcta");
```

```
   /* Hacemos una pausa hasta que el usuario pulse Intro */
   fflush(stdin);
   printf("\nPulse Intro para finalizar...");
   getchar();
}
```

¡Vaya, el mismo programa ocupa menos líneas de código! Esto es porque la sentencia `do-while` es más apropiada para este programa. Pensemos cómo es normalmente cualquier algoritmo que solicita una clave:

1. Se pide la contraseña.
2. Se comprueba si la contraseña es correcta: si no lo es se continúa, si sí lo es se termina.
3. Se pide la contraseña.
4. Se comprueba si la contraseña es correcta: si no lo es se continúa, si sí lo es se termina.
5. ... (y así sucesivamente).

Queda claro que este algoritmo comienza solicitando la clave, que es el bloque de código que se repite, y posteriormente se comprueba (mediante la condición). La sentencia `do-while` se adapta perfectamente a este algoritmo: se ejecuta la solicitud de contraseña mientras no sea la correcta. Sin embargo, en la sentencia `while` primero aparece la comprobación y después el bloque de código que se repite. Así que, para que quede el mismo algoritmo, tenemos que volver a escribir las líneas de código que solicitan la clave antes de la comprobación.

La sentencia `do-while` es, por el mismo motivo, la más apropiada para utilizar en los menús. La idea es mostrar en la pantalla un menú, solicitar la elección de una opción y mientras el usuario no elija la opción para salir se vuelve a repetir.

```
#include <stdio.h>

int main (void)

{

   int opcion;

   do

   {

      printf("\n\tMENU\n");

      printf (" ---------------- \n") ;

      printf(" 1 Juego nuevo\n");
      printf(" 2 Abrir juego\n");
      printf(" 3 Guardar juego\n");
      printf(" 4 Ver puntuaciones\n");
      printf(" 0 Salir\n");
      printf("\n");
```

```
      printf(" Escriba el numero de una opcion (0-4): ");
      scanf("%d", &opcion); fflush(stdin);

      switch (opcion)
      {
         case 1:  printf("\n Ha elegido la opcion: Juego nuevo.");
            break;

         case 2:  printf("\n Ha elegido la opcion: Abrir juego.");
            break;

         case 3:  printf("\n Ha elegido la opcion: Guardar juego.");
            break;

         case 4:  printf("\n Ha elegido la opcion: Ver puntuaciones.");
            break;

         case 0:  printf("\n Ha elegido la opcion: Salir.");
            break;

         default: printf("\n Ha elegido una opcion no valida.");
      }
      /* Esta linea ya esta fuera de la sentencia switch. */

      printf ("\nPulse Intro para continuar...");

      getchar();

   }
   while (opcion!=0);

   /* Esta linea ya esta fuera de la sentencia do-while. */
}
```

Las líneas:

```
printf ("\nPulse Intro para continuar...");
getchar();
```

consiguen hacer una pausa tras elegir una opción y finaliza cuando el usuario pulsa la tecla Intro.

Puede probar a suprimir la cláusula default. ¿Le gusta el efecto resultante o cree que es mejor dejarlo como está?

La sentencia do-while también la podemos utilizar en la repetición de un algoritmo mientras el usuario lo desee. El algoritmo podría ser: calcular y mostrar en la pantalla el doble de un número indicado por el usuario.

```
#include <stdio.h>

int main (void)

{

   int numero, doble;

   char letra;
```

```
   do
   {
      printf ("\nEscriba un numero: ");
      scanf ("%d",&numero);
      fflush(stdin);

      doble = numero * 2;
      printf ("\nEl doble de %d es: %d.", numero, doble);
      printf ("\n\n¿Desea calcular el doble de otro numero? (S/N): ");
      scanf ("%c", &letra);
      fflush(stdin);
   }
   while ((letra=='s') || (letra=='S'));
   /* Esta linea ya esta fuera de la sentencia do-while. */
}
```

Para controlar que el código que está dentro de la sentencia `do-while` se repita si el usuario lo desea escribimos al final del bloque de código:

```
printf ("\n\n¿Desea calcular el doble de otro numero? (S/N): ");
scanf ("%c",&letra);
```

Estas dos líneas se encargan de preguntar al usuario si desea calcular el doble de otro número y almacenar en la variable `letra` correspondiente a la respuesta. La condición:

```
while ((letra=='s') || (letra=='S'));
```

tiene dos subcondiciones para obtener la pregunta: ¿el valor de la variable `letra` es igual al carácter 's' o al carácter 'S? La sentencia se repetirá si se escribe la letra 's' (en minúscula) o 'S' (en mayúscula). Si se escribe un carácter distinto, como 'n' o 'N', la sentencia finaliza.

Sentencia for

La sentencia `for` (o "para") también permite repetir un bloque de código, pero su sintaxis es bastante distinta a la de la sentencia `while` y `do-while`:

```
for (inicializacion; condición; incremento)
   bloque_de_codigo
```

Donde `inicializacion` suele ser una asignación a una variable, `condición` es una expresión lógica que determina si se sigue repitiendo el bloque de código (cuando la condición es verdadera) o no (cuando la condición es falsa), e `incremento` es una expresión que suele modificar el valor de la variable. El funcionamiento de la sentencia `for` se puede definir en una serie de pasos:

1. Ejecución de la inicialización.
2. Evaluación de la condición: si es verdadera se continúa con el paso 3, si no se salta al paso 6.
3. Ejecución del bloque_de_codigo.
4. Ejecución del incremento.
5. Salto al paso 2.
6. Fin de la sentencia f or.

El ordinograma de la figura 9.10 representa a la sentencia f or.

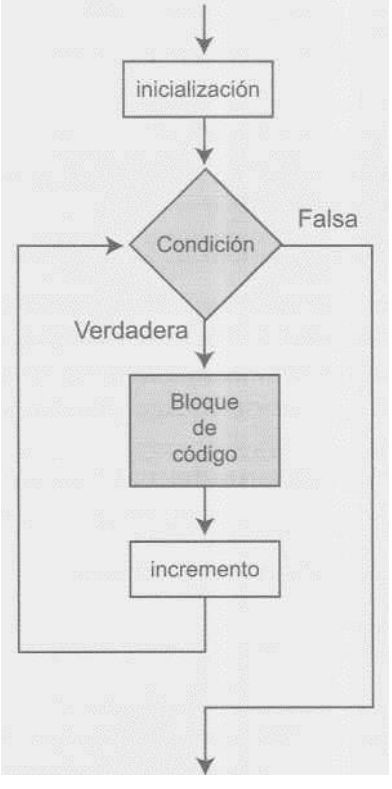


Figura 9.10. Ordinograma de la sentencia for.

Advertencia:

La repetición del bloque de código de la sentencia for no es "hasta que se cumpla la condición ", sino que es "mientras se cumpla la condición".

La sentencia `for` suele usarse para casos similares al que vimos anteriormente, en los que, por ejemplo, queremos mostrar los número del 1 a 6 en pantalla, o repetir un bloque de código un número determinado de veces. El programa que mostraba los número del 1 al 6 ocupa menos líneas de código con la sentencia `for`:

```
#include <stdio.h>
int main (void)
{
   int numero;
   for (numero=1; numero<=6; numero++)
   {
      printf ("%d\n", numero);
   }
   /* Esta linea ya esta fuera de la sentencia for. */
   /* Hacemos una pausa hasta que el usuario pulse Intro */
   fflush(stdin);
   printf ("\nPulse Intro para finalizar...");
   getchar();
}
```

Si lo comparamos con el que hicimos con la sentencia while, nos daremos cuenta de que las instrucciones son las mismas y que únicamente han cambiado de sitio, pero el orden de ejecución de las instrucciones es realmente el mismo. Las llaves del bloque de código pueden suprimirse, porque sólo contiene una instrucción.

¿Y si en vez de mostrar los números del 1 al 6 quisiéramos hacer un programa que mostrase los números del 1 a aquel que indicase el usuario?

```
#include <stdio.h>
int main (void)
{
   int numero, ultimo;
   printf ("Escriba el ultimo numero a mostrar; ");
   scanf ("%d", Sultimo);
   for (numero=1; numero<=ultimo; numero++)
   printf ("%d\n", numero);
   /* Esta linea ya esta fuera de la sentencia for. */
   /* Hacemos una pausa hasta que el usuario pulse Intro */
   fflush(stdin);
   printf("\nPulse Intro para finalizar...");
   getchar();
}
```

La variable `ultimo` será la encargada de almacenar el último número a mostrar en la pantalla. Dicho número será el que el usuario escriba gracias a las líneas:

```
printf ("Escriba el ultimo numero a mostrar: "); scanf ("%d", ¿¡ultimo) ;
```

A continuación, llegamos a la sentencia `for`.

```
for (numero=l; numero<=ultimo; numero++) printf ("%d\n", numero);
```

Vamos a suponer que el usuario escribió el número 3, por lo que la variable `ultimo` almacena el valor 3. El orden de ejecución de la sentencia `for` es:

1. `numero=l,` ahora la variable `numero` almacena el valor **1**.
2. `numero<=ultimo,` es decir, ¿1 es menor o igual que 3? Sí, luego continuamos.
3. `printf ("%d\n", numero),` se muestra en pantalla el número 1.
4. `numero++,` entonces se incrementa el valor de la variable `numero,` que ahora es 2.
5. `numero<=ultimo,` es decir, ¿2 es menor o igual que 3? Sí, luego continuamos.
6. `printf ("%d\n", numero),` se muestra en pantalla el número 2.
7. `numero++,` entonces se incrementa el valor de la variable `numero,` que ahora es 3.
8. `numero<=ultimo,` es decir, ¿3 es menor o igual que 3? Sí, luego continuamos.
9. `printf ("%d\n", numero),` se muestra en pantalla el número 3.
10. `numero++,` entonces se incrementa el valor de la variable `numero,` que ahora es 4.
11. `numero<=ultimo,` es decir, ¿4 es menor o igual que 3? No, luego finaliza la sentencia `for`.

Otro programa que podríamos hacer consistiría en pedir un número al usuario y, a continuación, mostrar su tabla de multiplicar.

```
#include <stdio.h>
int main (void)
{
    int numero, n, resultado;
    printf ("Escriba un numero: ");

    scanf ("%d", &numero);
```

```
   printf("\nLa tabla de multiplicar del numero %d es:\n", numero);
   for (n=1; n<=10; n++)
   {
      resultado = n * numero;
      printf ("\n\t%d x %d = %d", n, numero, resultado);
   }
   /* Hacemos una pausa hasta que el usuario pulse Intro */
   fflush(stdin);
   printf("\nPulse Intro para finalizar...");
   getchar();
}
```

La variable numero almacenará el número que el usuario escriba mediante las líneas:

```
printf ("Escriba un numero: "); scanf ("%d", &numero);
```

La variable n será la que tome los valores de 1 a 10 gracias a la sentencia for:

```
for (n=1; n<=10; n++)
```

y que multiplicará al número del usuario para obtener el resultado de la multiplicación, que se almacenará en la variable resultado:

```
resultado = n * numero;
printf ("\n%d x %d = %d", n, numero, resultado);
```

El resultado de ejecutar este programa para el caso de que el usuario escriba el número 2 se muestra en la figura 9.11.

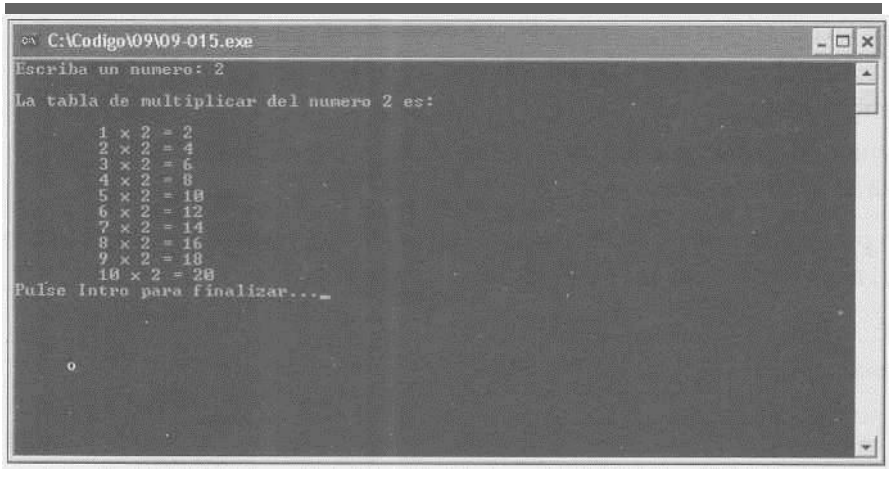


Figura 9.11. Resultado de la tabla de multiplicar del número 2.

Bucles infinitos y otros errores

Es importante que en todas las sentencias repetitivas exista alguna instrucción que haga falsa la condición en algún momento. En caso contrario, la condición siempre sería verdadera y nunca finalizaría la sentencia repetitiva. Estaríamos ante un bucle infinito.
En el siguiente programa hay intención de mostrar los números del 1 a 10, pero ¿dónde se incrementa el valor de la variable `numero`? Siempre almacena el mismo valor y la condición siempre es verdadera.

```
#include <stdio.h>

int main (void)

{
   int numero = 1;

   while (numero<=10)

      printf ("\n%d", numero);

   /* Hacemos una pausa hasta que el usuario pulse Intro */
   fflush(stdin);
   printf("\nPulse Intro para finalizar...");

   getchar();
}
```

Tenemos que añadir una instrucción que incremente el valor de la variable numero, pero ¿sería correcto lo siguiente?

```
#include <stdio.h>

int main (void)

{

   int numero = 1;

   while (numero<=10)

      printf ("\n%d", numero);
      numero++;

   /* Hacemos una pausa hasta que el usuario pulse Intro */
   fflush(stdin);

   printf("\nPulse Intro para finalizar...");

   getchar();

}
```

No. La instrucción:

```
numero++;
```

realmente se encuentra fuera de la sentencia while, por lo que no modifica el valor de la variable numero. Por tabular más o menos las líneas de código no pasan a estar dentro o fuera de una sentencia.

Como no se han utilizado las llaves, la sentencia while sólo afecta a una instrucción: la llamada a la función printf. Así que volvemos a estar ante un bucle infinito.
Pongamos las llaves para corregir el error. ¿Estaría ya bien?

```
#include <stdio.h>
int main (void)
{
   int numero = 1;
   while (numero<=10);
   {
      printf ("\n%d", numero);
      numero++;
   }
   /* Hacemos una pausa hasta que el usuario pulse Intro */
   fflush(stdin);
   printf ("\nPulse Intro para finalizar...");
   getchar();
}
```

Tampoco. Al volver a escribir el código hemos añadido un punto y coma tras la condición. El resultado es que se considerará el bloque de código de la sentencia while a lo que hay entre la condición y el punto y coma: nada. Así que estamos ante un bucle while vacío e infinito. Este mismo error nos puede ocurrir con la sentencia f or.

```
#include <stdio.h>
int main (void)
{
   int numero;
   for (numero=l; numero<=10; numero++);
   printf ("\n%d",numero);
   /* Hacemos una pausa hasta que el usuario pulse Intro */
   fflush(stdin);
   printf("\nPulse Intro para finalizar...");
   getchar () ;
}
```

Observe que al final de la línea del for hay un punto y coma. En este caso el bucle no es infinito. Lo que ocurriría es que la variable tomaría los valores del 1 al 10 en cada iteración, pero sin mostrarse nada en la pantalla, puesto que se considerará el bloque de código de la sentencia for lo que haya entre el paréntesis cerrado y el punto y coma: nada. La llamada a la función `printf` se ejecutará una sola vez.

Otro tipo de error que nos puede causar un bucle infinito es que la condición esté mal planteada, como a continuación.

```
#include <stdio.h>

int main (void)

{

    int numero = 1;

    while (numero>0)

    {

       printf ("\n%d", numero);
       numero++;

   /* Hacemos una pausa hasta que el usuario pulse Intro */
   fflush(stdin);

   printf("\nPulse Intro para finalizar...");
   getchar();
}
```

La variable numero comienza tomando el valor 1 y dentro de la sentencia while su valor se incrementa, así que nunca podrá tomar un valor menor que 1. Sin embargo, la condición para repetir el bucle es que el valor de la variable número sea mayor que 0, y esto siempre será así, nunca se dará el caso contrario, por lo que la condición nunca será falsa: el bucle es infinito. Las condiciones formadas por subcondiciones suelen causar muchos dolores de cabeza. El siguiente bucle no es infinito, pero no emplear correctamente los operadores lógicos puede ocasionar bucles infinitos. En cualquier caso, este programa no es correcto.

```
#include <stdio.h>

int main (void)
{

   char letra; do

   }

      printf ("¿Quiere mostrar esta pregunta otra vez? (S/N)");

      scanf ("%c", &letra);

   }

   while ((letra=='s') && (letra=='S'));

   /* Hacemos una pausa hasta que el usuario pulse Intro */
   fflush(stdin);

   printf("\nPulse Intro para finalizar...");

   getchar();

}
```

¿O podría decir cómo puede darse el caso de que la variable letra tome el valor 's' y 'S' a la vez?

Todos estos errores suelen ser los más comunes en lo que se refiere a los bucles. Téngalos presentes. Siempre que tenga algún tipo de problema con las sentencias repetitivas debería repasar este apartado, puede que en él encuentre la solución a su problema.

Sentencias anidadas

Cualquiera de las sentencias de este capítulo puede escribirse dentro del bloque de código de cualquier otra sentencia. A escribir unas sentencias dentro de otras se le llama anidar sentencias.
Anteriormente vimos un programa que mostraba los números impares del 1 al 10. Este también lo podemos hacer así:

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

   int numero; numero=1;
   while (numero<=10)
   {
      if (numero%2 != 0)
      {
      numero++;
   }
   /* Hacemos una pausa hasta que el usuario pulse Intro */
   fflush(stdin);
   printf("\nPulse Intro para finalizar...");
   getchar ();
}
```

Lo que hacemos es que la variable numero tome realmente todos los valores del 1 al 10, pero lo mostramos solamente cuando es impar gracia a esta condición:

```
if (numero%2 != 0)
{
   printf ("\n%d", numero);
}
```

Recordemos que si el resto o módulo de un número entre 2 es igual a cero, entonces ese número es par; en caso contrario es impar.

```
#include <stdio.h>
int main (void)
{
   int numero;
   printf ("Escriba un numero: ");
   scanf ("%d", &numero);
   if (numero >= 1)
   {
      if (numero <=10)
      {
         printf ("%d esta entre 1 y 10.", numero);
      }
   }
   /* Hacemos una pausa hasta que el usuario pulse Intro */
   fflush(stdin);
   printf("\nPulse Intro para finalizar...");
   getchar();
}
```

Las anteriores líneas de código comprueban si el valor de la variable `numero` es mayor o igual que 1, en cuyo caso se comprueba si también es menor o igual que 10, en cuyo caso se muestra un mensaje indicándolo. Estos if anidados equivalen al siguiente.

```
if ((numero>=1) && (numero<=10))
   printf ("%d esta entre 1 y 10.", numero);
```

El siguiente código se encargaría de mostrar en la pantalla la tabla de multiplicar de los números del 1 al 10.

```
(#include <stdio.h> #include <conio.h>
int main (void)
{
   int numero, n, resultado;
   for (numero=1; numero<=10; numero++)
   {
      printf ("\nTabla de multiplicar del %d:", numero);
      for (n=1; n<=10; n++)
      (
         resultado = n * numero;
         printf ("\n\t%d x %d = %d", n, numero, resultado);
      }
      printf ("\nPulse una tecla para continuar...");
      getch();
   }
}
```

Para cada iteración o repetición del primer bucle la variable `numero` toma un valor del 1 al 10 y se produce una ejecución completa de la sentencia `for` anidada. El resultado es que para cada valor que toma la variable `numero`, la variable `n` toma todos los valores del 1 al 10, y al multiplicar `n` por `numero` se obtiene la tabla de multiplicar de cada valor de `numero`.

Nota:

A la hora de anidar sentencias es muy aconsejable, sobre todo al principio, escribir las llaves que encierran al bloque de código de cada una de las sentencia, aunque no sea necesario, con el fin de ver claramente cuál está dentro de cuál y qué instrucciones están dentro de cada sentencia.

Ejercicios resueltos

Intente realizar los siguientes ejercicios. Encontrará la solución al final de este libro.

1. Realice un programa que pida un número al usuario y, a continuación, muestre un mensaje en pantalla que indique si dicho número es mayor que 10 o es menor que 10.
2. Realice un programa que, mientras el usuario lo desee, pida números e indique mediante un mensaje en pantalla si es par o impar.
3. Realice un programa que muestre en la pantalla los números del 1 al 100 y de diez en diez.

Resumen

En este capítulo hemos aprendido a controlar el flujo de ejecución de un programa, a indicar qué código debe ejecutarse, bajo qué condición y cuántas veces. Todo ello gracias a las sentencias `if`, `if-else`, `switch`, `while`, `do-while`, `for`. Pero también hemos analizado programas con código que nos puede ser muy útil para hacer menús, ejecutar un bloque de código siempre que el usuario lo desee, hacer que una variable tome una serie de valores sucesivos o no, etc.

Capítulo 10

Arrays

En este capítulo aprenderá:

- Qué son los *arrays.*
- A manejar *arrays* unidimensionales y bidimensionales.
- A trabajar con cadenas de caracteres.

Introducción

A la hora de realizar algunos programas surge la necesidad de almacenar varios datos, para lo cual deberíamos utilizar varias variables.
Un programa que almacene 4 edades escritas por el usuario mediante el teclado y después muestre la media lo podríamos hacer de la forma que se describe a continuación:

```
#include <stdio.h>

int main (void)

{
   int edad1, edad2, edad3, edad4; int media;

   /* Pedimos la edad de la persona 1. */
   printf ("Introduzca la edad de la persona 1: ");
   scanf ("%d", Sedadl);

   /* Pedimos la edad de la persona 2. */
   printf ("Introduzca la edad de la persona 2: ");
   scanf ("%d", &edad2);

   /* Pedimos la edad de la persona 3. */
   printf ("Introduzca la edad de la persona 3: ");
   scanf ("%d", &edad3);

   /* Pedimos la edad de la persona 4. */
   printf ("Introduzca la edad de la persona 4: ");
   scanf ("%d", &edad4);

   /* Calculamos la media y la mostramos. */
   media = (edadl + edad2 + edad3 + edad4) / 4;
   printf ("La media es; %d", media);
}
```

Pero si nos dicen que calculemos la media de edad de 20 ó 50 personas... ¡tendríamos que crear 20 ó 50 variables "edad"!, por no hablar de la expresión que calcula la media, que sería enorme. Realmente estamos ante un problema y la solución está en los *arrays*.
Un *array* se define como una colección indexada de variables del mismo tipo, a las que nos referimos con el nombre del *array* junto con el índice.
Los *arrays* pueden ser unidimensionales o multidimensionales (como las bidimensionales o las tridimensionales), pero nosotros nos centraremos en los unidimensionales y en los bidimensionales.
Para poder entender esta definición vamos a pasar ahora a los *arrays* unidimensionales.

Arrays unidimensionales

Un *array* unidimensional se suele representar como un número determinado de celdas, o un conjunto de celdas, unas seguidas de otras, de ahí que hablemos de colección.
En cada una de esas celdas podemos almacenar datos de un tipo determinado y también podemos cambiar esos datos, por lo que cada una de las celdas se comporta como una variable de un tipo determinado. Todas las celdas pueden almacenar datos, pero sólo del mismo tipo.
Para que podamos referirnos a una celda determinada tenemos que indicar su posición.
Para ello, a cada celda le corresponde un número, un índice.
La primera celda se corresponde con el número o la posición 0, la siguiente celda se corresponde con el número o la posición 1, y así sucesivamente.
Recuerde que el índice siempre empieza en 0 y no en 1.
Como esta colección de celdas tiene un índice (0,1,...) decimos entonces que es indexada.
Para poder referirnos a la colección de celdas le damos un nombre, el identificador del *array*.
En la figura 10.1 se muestra una representación gráfica de un *array* unidimensional y se indican sus partes.

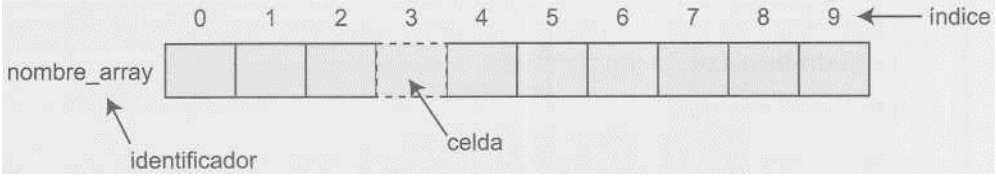


Figura 10.1. Representación gráfica de un *array* unidimensional.

En la vida cotidiana nos podemos encontrar con ejemplos de *arrays* en muchos lugares.
Imaginemos un gimnasio en el que hay 3 grupos de taquillas. Cada grupo, formado por 5 taquillas, tiene un nombre: Grupo A, Grupo B y Grupo C, para que los deportistas sepan en qué grupo está su taquilla. Dentro de cada grupo las taquillas siempre están enumeradas del 0 al 4, así los deportistas pueden saber cuál es su taquilla dentro de su grupo. Está claro que el contenido de cada taquilla puede cambiar.
Estamos ante un ejemplo real de 3 *arrays* unidimensionales, como muestra la figura 10.2.

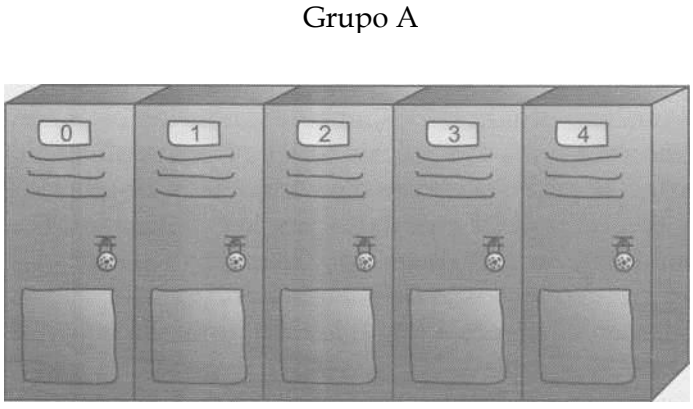


Figura 10.2. Un grupo de taquillas es similar a un *array* unidimensional.

Declaración

La sintaxis para declarar un *array* unidimensional es:

```
tipo identificador[tamaño];
```

Donde `tipo` es el tipo de dato que van a almacenar las celdas (y que es el mismo para todas), `identif icador` es el nombre del *array* unidimensional, y `tamaño` es el número de celdas que tiene.
La siguiente línea de código declara un *array* unidimensional que se llama `mi_array,` de tipo `int` y con 5 celdas.

```
int mi_array[5];
```

Su representación gráfica se muestra en la figura 10.3.

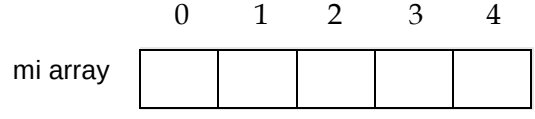


Figura 10.3. Representación gráfica de mi_array.

Observe en qué poco espacio podemos declarar tantas variables, teniendo en cuenta que cada celda del *array* se comporta como una variable. Piense que declarar `500` variables de tipo `int` es tan sencillo como escribir: `int variables [ 500] ;.`
En cada una de las celdas de `mi_array` podemos almacenar datos de tipo `int,` pero ¿cómo? Continuemos.

Acceso a elementos del array

Para acceder a una celda del *array* unidimensional, tanto para obtener su contenido como para asignarle un valor, escribimos el identificador del *array* (para indicar con qué *array* trabajaremos) seguido del índice o posición de la celda entre corchetes (para indicar con qué celda del *array* vamos a trabajar):

```
identificador[indice]
```

Veamos unos ejemplos.

1. Si queremos asignar el valor `10` a la celda `0` de `mi_array` escribiremos:

   ```
   mi_array[0] = 10;
   ```

2. Si queremos mostrar el valor de la celda 0 en la pantalla (en la que aparecerá el número 10) escribiremos:

   ```
   printf ("%d", mi_array[0]); o cout « mi_array[0];
   ```

3. Si queremos asignar a la celda 3 el valor de la celda 0 más el valor 7 escribiremos:

   ```
   mi_array[3] = mi_array[0] +7;
   ```

4. Si queremos almacenar en la celda `4` un número introducido por teclado escribiremos:

   ```
   scanf ("%d", &mi_array[4]); o cin » mi_array[4];
   ```

Si en el cuarto ejemplo el usuario escribe el número `30` mediante el teclado, el aspecto de `mi_array` sería el de la figura `10.4`.

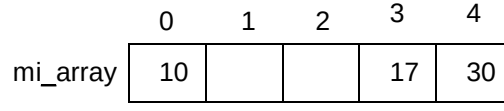


Figura 10.4. Aspecto de mi_arrray tras los ejemplos.

Inicialización del array

En el mismo momento de la declaración del *array* podemos asignar unos valores iniciales a sus celdas. La sintaxis es:

```
tipo identificador[tamaño] = {lista_de_valores};
```

Donde `lista_de_valores` son los distintos valores de las celdas separados por comas: `valor_1, valor_2, ..., valoran`. Estos valores se asignan a las celdas siguiendo el orden: de la celda 0 a la última.

Así, si escribimos la siguiente declaración:

```
int mi_array[5] = {1, 500, 0, 33, 156};
```

el *array* mi_array quedará como se muestra en la figura 10.5.

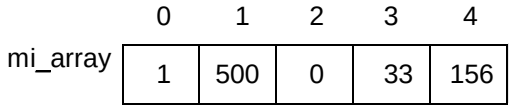


Figura 10.5. Aspecto del *array* mi_arrray tras inicializarlo.

Y si escribimos la siguiente declaración:

```
char vocales[5] = {'a',' e','i','o',' u'};
```

el *array* `vocales` quedará como se muestra en la figura 10.6.

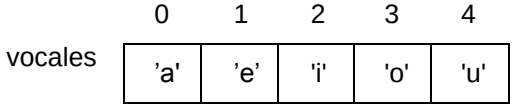


Figura 10.6. Aspecto del *array* vocales tras inicializarlo.

Inicialización de un array recorriéndolo

Si queremos inicializar todo el *array* con mismo valor, por ejemplo el 0, podemos recorrerlo con una sentencia repetitiva e ir asignando el valor 0 a cada una de las celdas. A continuación mostramos el mismo ejemplo con las sentencias `for`, `while` y `do-while`. Es muy sencillo y contiene únicamente el código principal para inicializar un *array* recorriéndolo.
En el siguiente ejemplo se realiza esta operación con un *array* de 10 celdas de tipo `int` y con la sentencia `for`.

```
int números[10];
int celda;

for (celda=0; celda<10; celda++)
    números[celda]=0;
```

En cada iteración o repetición del bucle `for` la variable `celda` toma un valor del 0 al 9. El resultado es que el bucle es equivalente al siguiente listado:

```
números[0]=0;     /* Cuando celda toma el valor 0(primera iteración). */
números[1]=0;     /* Cuando celda toma el valor 1(segunda iteración). */
números[2]=0;     /* Cuando celda toma el valor 2(tercera iteración). */
números[3]=0;     /* Cuando celda toma el valor 3 (cuarta iteración). */
números[4]=0;     /* Cuando celda toma el valor 4 (quinta iteración). */
números[5]=0;     /* Cuando celda toma el valor 5  (sexta iteración). */
```

```
números[6]=0;     /* Cuando celda toma el valor 6  (séptima iteración). */
números[7]=0;     /* Cuando celda toma el valor 7  (octava iteración).*/
números[8]=0;     /* Cuando celda toma el valor 8  (novena iteración).*/
números[9]=0;     /* Cuando celda toma el valor 9  (decima iteración).*/
```

Tras lo cual el *array* almacena el valor 0 en todas y cada una de sus celdas.
El mismo ejemplo con la sentencia `while`:

```
int números[10];
int celda;
celda=0;
while (celda<10)
{
   números[celda]=0;
   celda++;
}
```

El mismo ejemplo con la sentencia `do-while`:

```
int números[10];
int celda;
celda=0;
do
 {
   números[celda]=0;
   celda++;
 } while (celda<10);
```

¿Cómo rellenar un array con datos introducidos por teclado?

Si queremos almacenar en el *array* distintos valores introducidos por teclado podemos recorrerlo e ir asignando a cada una de las celdas un valor mediante cualquier función de lectura del teclado. A continuación mostramos el mismo ejemplo con las tres sentencias repetitivas y únicamente con el código principal.
En el siguiente ejemplo se realiza esta operación con un *array* de 10 celdas de tipo `int`, la sentencia `for` y la función `scanf`.

```
int números[10];
int celda;
for (celda=0; celda<10; celda++)
scanf ("%d", &numeros[celda]);
```

En cada iteración o repetición del bucle `for` la variable `celda` toma un valor del 0 al 9. El resultado es que el bucle es equivalente al siguiente listado:

```
scanf  ("%d",&numeros [0] ) ;/* Primera  iteración. */
scanf  ("%d",&numeros [1] ) ;/* Segunda  iteración. */
scanf  ("%d",&numeros [2] ) ;/* Tercera  iteración. */
```

```
scanf  ("%d",&numeros [3]);   /*  Cuarta iteración. */
scanf  ("%d",&numeros[4]);    /*  Quinta iteración. */
scanf  ("%d",&numeros [5]);   /*  Sexta iteración. */
scanf  ("%d”,&numeros[6]);    /*  Séptima iteración. */
scanf  ("%d",&numeros[7]);    /*  Octava iteración. */
scanf  ("%d",&numeros [8]);   /*  Novena iteración. */
scanf  ("%d",&numeros[9]);    /*  Decima iteración. */
```

Si va a utilizar este código como ejemplo para hacer otro programa recuerde que delante de cada lectura del teclado `(scanf)` debe mostrar algún tipo de mensaje al usuario (mediante `printf,` por ejemplo). Recuerde también que puede ser necesario utilizar la función `f f lush` tras la llamada a la función `scanf.`

El mismo ejemplo con la sentencia `while:`

```
int números[10];
int celda;
celda=0;
while (celda<10)
{
```

El mismo ejemplo con la sentencia `do-while:`

```
int números[10];
int celda;
celda=0;
do {
   scanf ("%d", &numeros[celda]);
   celda++;
} while (celda<10);
```

¿Cómo mostrar en pantalla el contenido de un array?

Si lo que queremos es mostrar el contenido del *array* podemos recorrerlo e ir mostrando el contenido de cada una de las celdas. A continuación mostramos el mismo ejemplo con las tres sentencias repetitivas y únicamente con el código principal.

En el siguiente ejemplo se realiza esta operación con un *array* de 10 celdas de tipo int, la sentencia for y la función printf.

```
int números[10];
int celda;
for (celda=0; celda<10; celda++)
printf ("%d", números[celda]);
```

En cada iteración o repetición del bucle f or la variable celda toma un valor del 0 al 9. El resultado es que el bucle es equivalente al siguiente listado:

```
printf  ("%dⁿ,números[0]);    /*Primera iteración. */
printf  ("%d",números[1]);    /*Segunda iteración. */
printf  ("%dⁿ,números[2]);    /*Tercera iteración. */
printf  ("%d",números[3]);    /* Cuarta iteración. */
printf  ("%dⁿ,números[4]);    /* Quinta iteración. */
printf  ("%d",números[5]);    /*  Sexta iteración. */
printf  ("%d",números[6]);    /*Séptima iteración. */
printf  ("%d",números [7]);   /* Octava iteración. */
printf  ("%dⁿ,números[8]);    /* Novena iteración. */
printf  ("%d",números[9]);    /* Decima iteración. */
```

En la pantalla aparecerán todos los valores almacenados en el *array,* uno seguido de otro. Habrá que añadir algunos detalles a la cadena de la función printf para mejorar la presentación en pantalla.
El mismo ejemplo con la sentencia while:

```
int números[10];
int celda;
celda=0;
while (celda<10)
{
   printf("%d", números[celda]);
   celda++;
}
```

El mismo ejemplo con la sentencia do-while:

```
int números[10];
int celda;
celda=0;
do
{
   printf("%d", números[celda]);
   celda++;
} while (celda<10);
```

Ejemplo

Tras aprender la forma principal de trabajar con *arrays* unidimensionales vamos a estudiar un programa que utiliza un *array.* Este consiste en pedir al usuario 10 caracteres y almacenarlos en el *array.* A continuación, hay que buscar en el *array* el carácter 'a', o 'A', y mostrar en pantalla el número de veces que aparece.
Por ejemplo, si después de introducir los caracteres en el *array* éste tuviese el aspecto de la figura 10.7 el resultado sería un mensaje en pantalla del tipo: "Hay 3 letras 'a' o 'A'.".

```
#include <stdio.h>
int main (void)
{
   char letras[10];      /*  Para almacenar  los caracteres.    */
   int celda;            /*  Para almacenar el numero de la celda. */
   int contador;        /* Para contar las 'a' o 'A'. */

   /* Pedimos caracteres al usuario y los almacenamos en las
   celdas del         array  letras. */
   for (celda=0; celda<10;  celda++)
   }
      printf ("Escriba un carácter para la celda %d: ", celda);
      scanf ("%c", &letras[celda]);
      fflush (stdin);
   }

   /* Antes de empezar a contar las 'a' o 'A' ponemos la
   variable contado con el valor 0. */ contador = 0;

   /* Recorremos el array en busca de 'a' o 'A'. Si las
   encontramos aumentamos el valor de la variable contador,
   que es la que lleva la cuenta de cuantas hay. */
   for (celda=0; celda<10; celda++)
   {
      if ((letras[celda] == 'a') || (letras[celda] == 'A'))
         contador++;
   }

   /* Mostramos un mensaje indicando el numero de
   'a' o 'A' que se han encontrado. */

   printf ("\nHay %d \'a\' o \'A\'.", contador);

   /* Hacemos una pausa hasta que el usuario pulse Intro */
   fflush(stdin);
   printf("\nPulse Intro para finalizar...");
   getchar();
}
```

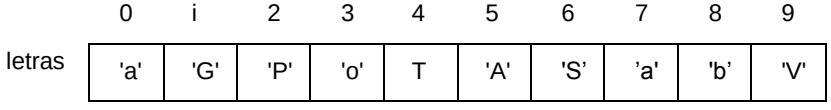


Figura 10.7. Ejemplo del *array* de caracteres.

Lo más destacable para comentar de este programa es el bucle for que cuenta el número de 'a' o 'A' que hay en el *array*:

```
for (celda=0; celda<10; celda++)
{
   if ((letras[celda] == 'a') || (letras[celda] == 'A'))
      contador++;
}
```

En cada repetición del bucle la variable celda toma un valor del 0 al 9. De esta manera, en la primera iteración la condición es equivalente a:

```
if ( (letras[0] == 'a') || (letras[0] == 'A'))
    contador++;
```

Si partimos del supuesto de que el *array* almacena los mismos datos de la figura 10.7, entonces la condición es verdadera, porque en la celda 0 hay una 'a', y el valor de la variable contador se incrementa en una unidad. En la siguiente iteración celda vale 1 y la condición es equivalente a:

```
if ( (letras[1] == 'a')  || (letras[l] == 'A'))
    contador++;
```

En este caso, la condición es falsa, porque en la celda 1 hay una 'G', y el valor de la variable contador no se incrementa, contador++ no se ejecuta.
Y así sucesivamente.
El resultado en la pantalla se muestra en la figura 10.8.

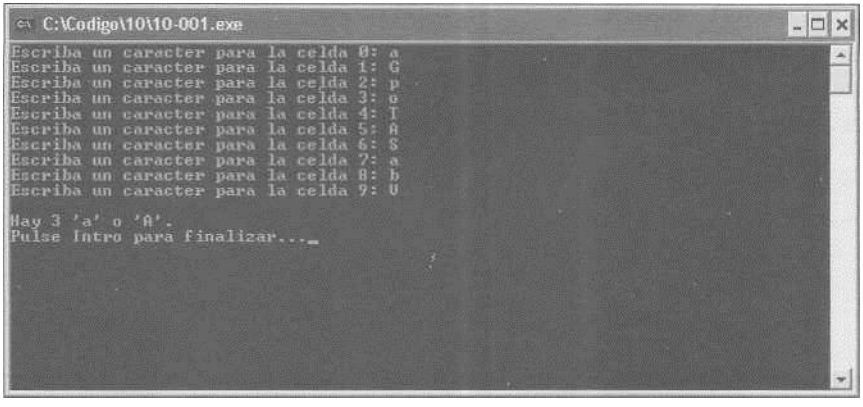


Figura 10.8. Resultado de contar el número de 'a' y 'A'.

Arrays bidimensionales

Si la forma de representar un *array* unidimensional es como se muestra en la figura 10.9.

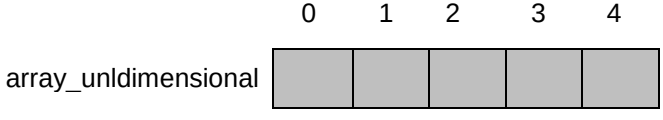


Figura 10.9. Representación de un *array* unidimensional.

Un *array* bidimensional se representa como en la figura 10.10.

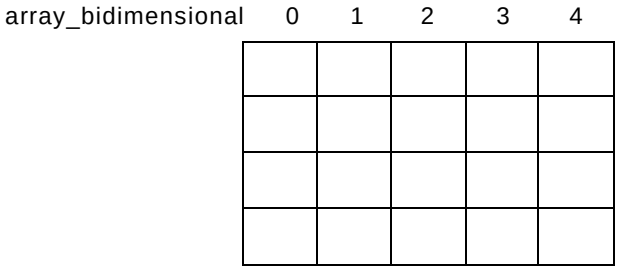


Figura 10.10. Representación de un *array* bidimensional.

Podemos observar que un *array* bidimensional es como un tablero con filas y columnas. Puede parecer obvio qué son las filas y las columnas, pero en muchas ocasiones se presentan dudas. En la figura 10.11 se indican las filas y las columnas.
En los *arrays* bidimensionales las filas y las columnas también comienzan a enumerarse en el 0.
Para acceder a una celda del *array* bidimensional tenemos que indicar su fila y su columna. Por lo demás, es igual que un *array* unidimensional.

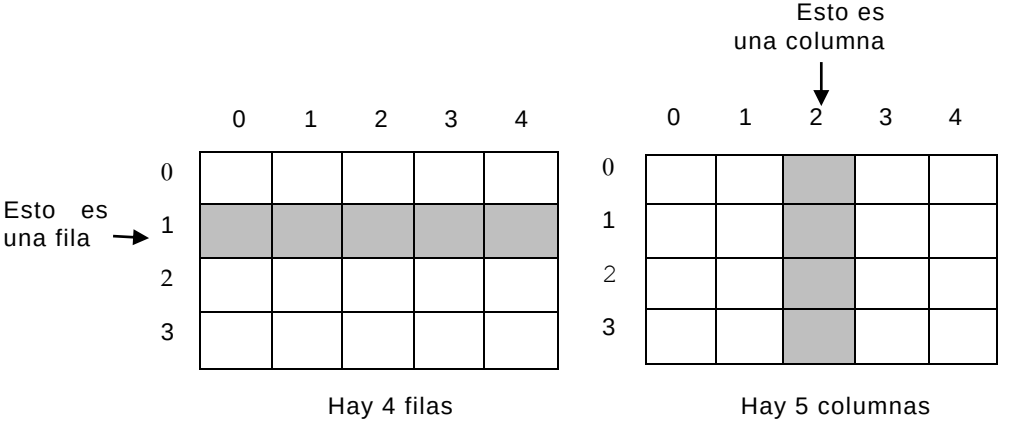


Figura 10.11. Filas y columnas en un *array* bidimensional.

Declaración

La sintaxis para declarar un *array* bidimensional es:

```
tipo identificador[numero_filas][numero_columnas];
```

Donde tipo es el tipo de dato que van a almacenar las celdas (y que es el mismo para todas), `identificador` es el nombre del *array* bidimensional, `numero_filas` es el número de filas del *array* y `numero_columas` es el número de columnas del *array*.
La siguiente línea de código declara un *array* bidimensional que se llama `array_bi`, de tipo `int` y con 4 filas y 6 columnas.

```
int array_bi[4][6];
```

Su representación gráfica se muestra en la figura 10.12.

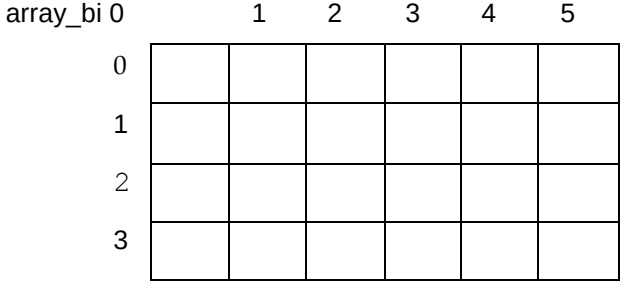


Figura 10.12. Representación gráfica de array_bi.

En cada una de las celdas de array_bi podemos almacenar datos que sean de tipo int.

Acceso a elementos del array

Para acceder a una celda del *array* bidimensional, tanto para obtener su contenido como para asignar un valor a ésta, escribimos el identificador del *array* seguido del índice de la fila entre corchetes y del índice de la columna entre corchetes:

```
identificador[indice_fila][indice_columna]
```

Veamos unos ejemplos.

1. Si queremos asignar el valor 8 a la celda situada en la fila 2 y columna 4 de array_bi escribimos:

   ```
   array_bi[2][4] = 8;
   ```

2. Si queremos mostrar el valor de la celda situada en la fila 2 y columna 4 en pantalla (donde aparecerá el número 8) escribimos:

   ```
   printf ("%d", array_bi[2][4]); o cout << array_bi[2][4];
   ```

1. Si queremos asignar a la celda situada en la fila 3 y columna 0 el valor de la celda situada en la fila 2 y columna 4, más 3, escribimos:
```
array_bi[3][0] = array_bi[2][4] + 3;
```
2. Si queremos almacenar en la celda situada en la fila 0 y columna 0 un número escrito mediante el teclado escribimos:
```
scanf ("%d", &array_bi[0][0]); o cin » array_bi[0] [0];
```

Si en el cuarto ejemplo el usuario escribe el número 701 mediante el teclado, el aspecto de `array_bi` sería el de la figura 10.13.

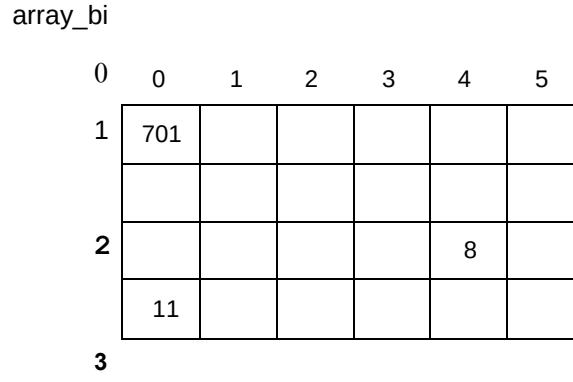


Figura 10.13. Aspecto de arrray_bi tras los ejemplos.

Inicialización del array

En el mismo momento de la declaración del *array* podemos asignar unos valores iniciales a sus celdas. La sintaxis es:

```
tipo identificador[numero_filas][numero_columnas] = {lista_de_valores};
```

Donde `lista_de_valores` son los distintos valores de las celdas separados por comas: `valor_1, valor_2,..., valor_n.` Estos valores se asignan a las celdas siguiendo el orden: de la fila 0 a la última y de la columna 0 a la última; gráficamente: de izquierda a derecha y de arriba abajo.

Así, si escribimos la siguiente declaración:

```
int tabla[31[4] =
    {
      1, 2, 3, 4,
      12, 100, 7, -2,
      27, 29, 501, 12
    } ;
```

el *array* `tabla` quedará como se muestra en la figura 10.14.

No es necesario escribir la inicialización colocada de tal manera, pero así ayuda a ver claramente qué valor va en cada celda.

tabla	0	1	2	3
0	1	2	3	4
1	12	100	7	-2
2	27	29	501	12

Figura 10.14. Aspecto del *array* tabla tras

Y si escribimos la siguiente declaración:

```
char letras[5] [2] = í
       'm', 'p',
       's', 'o',
       `a',  'u',
       't', 'r',
        `w', 'q'
    } ;
```

el *array* letras quedará como se muestra en la figura 10.15.

letras	0	1
0	'm'	'p'
1	's'	'o'
2	'a'	'u'
3	't'	'r'
4	'w'	'q'

Figura 10.15. Aspecto del *array* letras tras inicializarlo.

Inicialización de un array recorriéndolo

Si queremos inicializar todo el *array* con el mismo valor, por ejemplo el 0, podemos recorrerlo con una sentencia repetitiva e ir asignando el valor 0 a cada una de las celdas. A continuación mostramos el mismo ejemplo con las sentencias f or, while y do-while. Es muy sencillo y contiene únicamente el código principal para inicializar un *array* bidimensional recorriéndolo.
En el siguiente ejemplo se realiza esta operación con un *array* de 10 filas y 5 columnas de tipo int y con la sentencia f or.

```
int números[10][5];
int fila, columna;
for (fila=0; fila<10; fila++)
    for (columna=0; columna<5; columna++)
       números[fila][columna]=0;
```

Para cada iteración del primer bucle `for` la variable `fila` toma un valor de 0 a 9 y se produce un ejecución completa de la segunda sentencia for, que está anidada. En cada iteración del segundo `for` la variable `columna` toma un valor de 0 a 4. De esta manera, la ejecución completa de la primera iteración del primer `for` (cuando `fila` toma el valor 0) es equivalente a:

```
números[0][0]=0;
números[0][1]=0;
números[0][2]=0;
números[0][3]=0;
números[0][4]-0;
```

La ejecución completa de la segunda iteración del primer `for` (cuando `fila` toma el valor 1) es equivalente a:

```
números[1][0]=0;
números[1][1]=0;
números[1][2]=0;
números[1][3]=0;
números[1][4]=0;
```

Y así sucesivamente. Como puede observarse, la forma que se ha utilizado para acceder al *array* consiste en seleccionar una fila y recorrer sus columnas, seleccionar otra fila y recorrer sus columnas.

Tras la ejecución completa, el *array* almacena en todas sus celdas el valor 0.

El mismo ejemplo con la sentencia `while`:

```
int números[10][5]
; int fila, columna;
fila=0;
while (fila<10)
{
    columna=0; while (columna<5)
    {
       números[fila][columna]=0
       ; columna++;
    }
    fila++;
}
```

El mismo ejemplo con la sentencia `do-while`:

```
int números[10][5];
int fila, columna;
fila=0;
do
```

```
{
     columna=0;
     do
     {
        numeros[fila][columna]=0
        ; columna++;
     } while (columna<5);
     fila++;
} while (fila<10);
```

¿Cómo rellenar un array con datos introducidos por teclado?

Si queremos almacenar en el *array* bidimensional distintos valores introducidos por teclado podemos recorrerlo e ir asignando a cada una de las celdas un valor mediante cualquier función de lectura del teclado. A continuación mostramos el mismo ejemplo con las tres sentencias repetitivas y únicamente con el código principal.

En el siguiente ejemplo se realiza esta operación con un *array* de 10 filas y 5 columnas de tipo `int`, la sentencia `f or` y la función `scanf.`

```
int números[10][5];
int fila, columna;
for (fila=0; fila<10; fila++)
     for (columna=0; columna<5; columna++)
        scanf("%d",
        ínumeros[fila][columna]);
```

Para cada iteración del primer bucle `for` la variable `fila` toma un valor de 0 a 9 y se produce un ejecución completa de la segunda sentencia for, que está anidada. En cada iteración del segundo `for` la variable `columna` toma un valor de 0 a 4.

De esta manera, la ejecución completa de la primera iteración del primer for (cuando `fila` toma el valor 0) es equivalente a:

```
  scanf ("%d", &numeros[0][0]);
  scanf ("%d", &numeros[0][1]);
  scanf ("%d", &numeros[0][2]);
  scanf ("%d", &numeros[0][3]);
  scanf ("%d", &numeros[0][4]);
```

La ejecución completa de la segunda iteración del primer `for` (cuando `fila` toma el valor 1) es equivalente a:

```
  scanf ("%d", &numeros[1][0]);
  scanf ("%d", &numeros[1][1]);
  scanf ("%d", &numeros[1][2]);
  scanf ("%d", &numeros[1][3]);
  scanf ("%d", &numeros[1][4]);
```

Y así sucesivamente.
Si va a utilizar este código como ejemplo para hacer otro programa recuerde que delante de cada lectura del teclado (scanf) debe mostrar algún tipo de mensaje al usuario (mediante printf, por ejemplo).
Recuerde también que puede ser necesario usar la función fflush tras la llamada a la función scanf.
El mismo ejemplo con la sentencia while:

```
int números[10][5];
int fila, columna;
fila=0;
while (fila<10)
{
   columna=0; while
   (columna<5)
   {
      scanf("%d", &numeros[fila][columna]);
      columna++;
   }
   fila++;
}
```

El mismo ejemplo con la sentencia do-while:

```
int números[10][5];
int fila, columna;
fila=0; do {
   columna=0; do {
      scanf("%d", &numeros[fila][columna]);
      columna++;
   } while (columna<5);
   fila++;
} while (fila<10) ;
```

¿Cómo mostrar en pantalla el contenido de un array?

Si lo que queremos es mostrar el contenido del *array* bidimensional podemos recorrerlo e ir mostrando el contenido de cada una de las celdas. A continuación mostramos el mismo ejemplo con las tres sentencias repetitivas y únicamente con el código principal.
En el siguiente ejemplo se realiza esta operación con un *array* de 10 filas y 5 columnas de tipo int, la sentencia for y la función printf.

```
int números[10] [5];
int fila, columna;
for (fila=0; fila<10; fila++)
    for (columna=0; columna<5; columna++)
       printf("%d", &numeros[fila][columna]);
```

Para cada iteración del primer bucle for la variable fila toma un valor de 0 a 9 y se produce un ejecución completa de la segunda sentencia for, que está anidada. En cada iteración del segundo for la variable columna toma un valor de 0 a 4. De esta manera, la ejecución completa de la primera iteración del primer for (cuando fila toma el valor 0) es equivalente a:

```
scanf ("%d", números[0][0]) ;
printf ("%d", números[0][1]);
printf ("%d", números[0][2]);
printf ("%dn, números[0][3]);
printf ("%d", números[0][4]);
```

La ejecución completa de la segunda iteración del primer for (cuando fila toma el valor 1) es equivalente a:

```
printf ("%d", números[1][0]);
printf ("%dM, números[1][1]);
printf ("%d", números[1][2]);
printf (n%dM, números[1][3]);
printf ("%d", números[1][4]);
```

Y así sucesivamente. En la pantalla aparecerán todos los valores almacenados en el *array,* uno seguido de otro. Habrá que añadir algunos detalles a la cadena de la función printf para mejorar la presentación en pantalla.
El mismo ejemplo con la sentencia while:

```
int números[10][5];
int fila, columna;
fila=0;

while (fila<10)
{

    columna=0; while (columna<5)

    {
       printf("%d", números[fila][columna]);
       columna++;

    }

    fila++;
}
```

El mismo ejemplo con la sentencia do-while:

```
int números[10][5];
int fila, columna;
fila=0;
do
{
```

```
   columna=0;
   do
   {
      printf("%d",
      números[fila][columna]);
      columna++;
   } while (columna<5);
   fila++;
} while (fila<10);
```

Ejemplo

El siguiente programa crea un *array* bidimensional de 5 filas y 6 columnas. La primera celda de cada columna almacenará el número de expediente de los alumnos de una clase (hay 6 alumnos). Las tres celdas que hay por debajo de la primera almacenarán las notas de tres asignaturas. La última celda de la columna almacenará la nota media de las tres notas del alumno de dicha columna. Todos los valores son números enteros, luego el *array* es de tipo int. En la figura 10.16 se muestra un esquema.

	Alumno 1	Alumno 2	Alumno 3	Alumno 4	Alumno 5	Alumno 6
N° Expediente:	301	302	303	304	305	306
Nota 1 :	7	5	8	6	6	7
Nota 2:	10	6	9	5	10	8
Nota 3:	5	5	7	5	5	6
Nota Media:	7	5	8	5	7	7

Figura 10.16. Esquema.

El programa rellena el *array* bidimensional pidiendo al usuario en primer lugar el número de expediente del alumno *y,* en segundo lugar, sus tres notas. Esto lo hará para los seis alumnos. Después, se calcula la nota media de las tres notas de cada alumno y la almacena en la celda correspondiente. Por último, se muestra el número de expediente del alumno con la nota media mayor y dicha nota.

```
#include <stdio.h>

int main (void)
{
```

```
int tabla [5][6];    /*  Array bidimensional. */
int alumno;          /*  Para recorrer los alumnos (columnas).*/
int nota;            /*  Para recorrer las tres notas (filas).*/
int suma;            /*  Para la suma de las tres notas. */
int media_mayor;     /*  Almacena la media mayor. */
int mejor_exp;       /* Almacena el expediente con mayor media. */
/* Almacenamos en el array el numero de expediente y
las tres notas de cada alumno. */
for (alumno=0; alumno<=5; alumno++)
{
   /* Pedimos el numero de expediente, que se almacena
   en la fila 0. */ printf (M\nEscriba el numero de
   expediente: "); scanf("%d", &tabla[0][alumno]);
   /* Pedimos las tres notas, que se almacenan en las
   filas 1, 2 y 3. */ for (nota=1; nota<=3; nota++)
   {
      printf (M\nEscriba la nota %d: ", nota);
      scanf (n%d", &tabla[nota][alumno]);
   }
}
/* Calculamos la nota media de cada alumno. */
for (alumno=0; alumno<=5; alumno++)
{
   /* Obtenemos la suma de las tres notas. */
   suma=0;
   for (nota=1; nota<=3; nota++)
   {
      suma = suma + tabla[nota][alumno];
   }
   /* Almacenamos en la celda correspondiente la media
   de las tres notas. */
   tabla[4][alumno] = suma / 3;
}
/* Calculamos la nota media mayor y obtenemos dicha nota.
Partimos de la idea de que la mayor nota es 0. */
media_mayor=0;
for (alumno=0; alumno<=5; alumno++)
{
   /* Si estamos ante una nota mayor que la almacenada en
   media_mayor, entonces esta nueva nota pasa a ser la media mayor
   y almacenamos el numero de expediente de ese alumno. */
   if (tabla[4][alumno] > media_mayor)
   {
      media_mayor = tabla[4][alumno];
      mejor_exp = tabla[0][alumno];
   }
}
```

```
   /* Mostramos en la pantalla unos mensajes que indican el numero
   de expediente del alumno con la nota media mayor y dicha nota. */
   printf ("\n\nla mejor media:");
   printf ("\n\tExpediente: %d", mejor_exp);
   printf ("\n\tNota Media: %d\n", media_mayor);

   /* Hacemos una pausa hasta que el usuario pulse Intro */
   fflush(stdin);
   printf("\nPulse Intro para finalizar...");
   getchar();
}
```

En la figura 10.17 se muestra el resultado de este programa si introducimos los mismos datos del esquema de la figura 10.16.

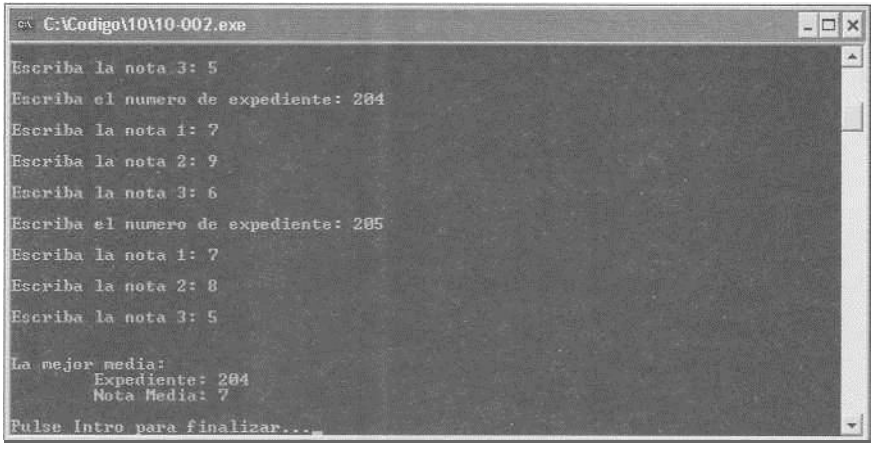


Figura 10.17. Resultado del programa de las notas.

Ejercicios resueltos

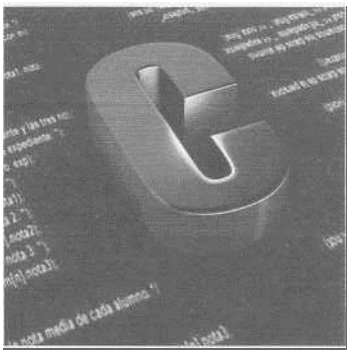
Intente realizar los siguientes ejercicios. Encontrará la solución al final de este libro.

1. Realice un programa que almacene en un *array* unidimensional 10 letras escritas por el usuario. A continuación, se debe mostrar un mensaje en pantalla indicando cuántas vocales se escribieron.
2. Realice un programa que pida al usuario 5 números enteros y los almacene en un *array* unidimensional. Posteriormente, se debe mostrar un mensaje en pantalla indicando la media de esos números, el mayor de ellos y el menor.

1. Realice un programa que pida al usuario 6 números enteros y los almacene en un *array* unidimensional. Después, se deben mostrar los mismos números pero en orden inverso al que los escribió el usuario.

Resumen

Una de las principales ventajas que nos ofrecen los *arrays* es la posibilidad de crear muchas variables (cada una de sus celdas es una variable) bajo un mismo identificador, el del *array*. Otra gran ventaja que poseen es que las celdas están indexadas, por lo que podemos acceder a cada una de ellas mediante bucles, utilizando pocas líneas de código. Con los *arrays* bidimensionales podemos almacenar datos que requieran estar ordenados en forma de tabla. En conclusión, los *arrays* nos resuelven muchos problemas, pero también tienen uno muy importante: todas las celdas deben ser del mismo tipo. Más adelante encontraremos alguna solución a este inconveniente. Paciencia.



Capítulo 11

Cadenas

En este capítulo aprenderá:

- Qué son las cadenas de caracteres.
- Cómo almacenar las cadenas en variables.
- Cómo mostrar las cadenas en pantalla.
- Cómo realizar diversas operaciones con cadenas.

Introducción

Hasta el momento hemos utilizado en nuestros programas variables que almacenan números y caracteres, pero nunca cadenas de caracteres como "Esto es una cadena". En C las variables que permiten almacenar cadenas son realmente *arrays* de tipo `char`. Con las cadenas podemos realizar gran cantidad de operaciones: fusionar dos cadenas en una, almacenar en una variable el nombre del usuario, pedir al usuario una contraseña que contenga números y letras, y muchas más. No se han tratado hasta este momento porque conviene conocer los punteros y los *arrays*.
Como seguramente esté ya impaciente por comenzar a hacer sus primeros programas con cadenas empecemos ya.

Declaración de una cadena

En C las variables que permiten almacenar cadenas se declaran igual que un *array* unidimensional de caracteres, donde el tamaño (número de celdas) se corresponde con el número máximo de caracteres de la cadena que puede almacenar. Cada carácter de la cadena quedará almacenado en una celda del *array*. La siguiente declaración puede almacenar una cadena de, como máximo, 9 caracteres (¿sólo 9 caracteres? Sí, continúe leyendo).

```
char mi_cadena[10];
```

¡Cuidado! Al final de cada cadena siempre se añade un carácter más, el carácter de fin de cadena '\0', que señala dónde termina la cadena. De esta manera, si almacenamos en `mi_cadena` la cadena "Hola" realmente se ocuparán 4 celdas (por los cuatro caracteres de "Hola") más 1 celda (por el carácter '\0'), que son un total de 5 celdas (véase la figura 11.1).

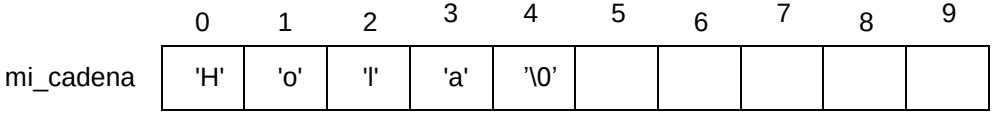


Figura 11.1. Estado de mpcadena tras almacenar "Hola".

Entonces, si tenemos pensado almacenar en `mi_cadena` una cadena de 10 caracteres (por ejemplo: "abcdefghij") tendremos que declararla con una celda más para el carácter de fin de cadena ('\0'):

```
char mi cadena[11];
```

La figura 11.2 muestra el resultado.

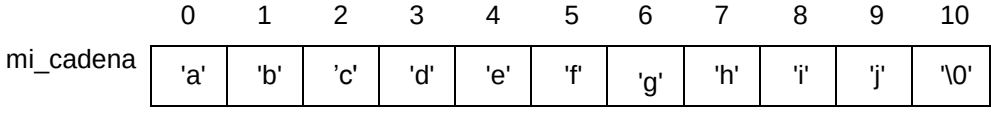


Figura 11.2. Estado de mi_cadena tras almacenar "abcdefghij".

Y ¿qué ocurriría si almacenamos una cadena mayor, por ejemplo "abcdefghijklm"? La variable `mi_cadena` tomaría el aspecto de la figura 11.3.

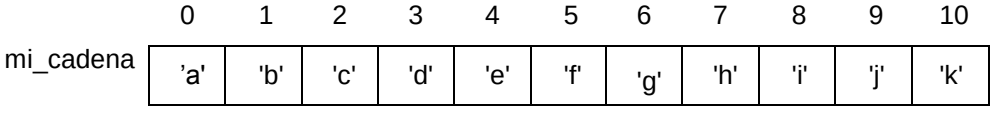


Figura 11.3. Estado de mi_cadena tras almacenar "abcdefghljklm".

Sin entrar en muchos detalles, en `mi_cadena` solamente se almacena la parte de la cadena que quepa: "abcdefghijk". Pero ¿qué ocurre con el resto de la cadena ("lm") y el carácter de fin de cadena ('\0')? Recordemos que las variables se almacenan en la memoria del ordenador, así que tras la variable `mi_cadena` hay más memoria y, por lo tanto, más lugares donde almacenar datos. Pues el resto de la cadena y el carácter de fin de cadena se han almacenado en esas otras posiciones de la memoria (sustituyendo a los datos que allí hubiese), pero fuera de la variable `mi_cadena` (véase la figura 11.4).

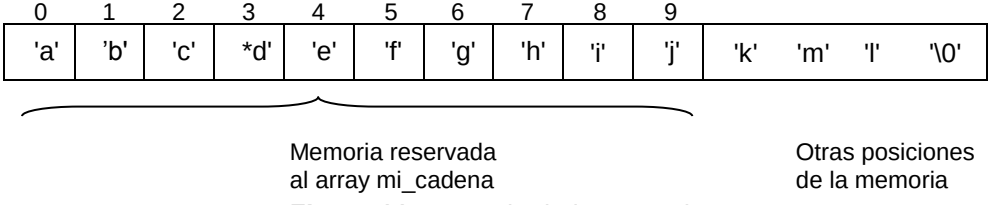


Figura 11.4. Estado de la memoria.

La consecuencia de este tipo de situaciones es un funcionamiento incorrecto del programa. Tenga en cuenta que en esas otras posiciones de memoria podría haber otros datos importantes para el programa y modificarlos podría ocasionar errores graves.

De momento, es preferible declarar los *arrays* para almacenar cadenas con un número de celdas que nos dé seguridad, por ejemplo:

```
char mi_cadena[100];
```

que nos permite almacenar una cadena con un máximo de hasta 99 caracteres, más el carácter de fin de cadena.
En cualquier caso, pronto empezaremos a estudiar mecanismos para que no nos sucedan este tipo de problemas.
Lo que debe quedar claro es que:

- Las variables que almacenan cadenas son simples *arrays* unidimensionales de tipo char.
- El número de celdas del *array* indica el tamaño máximo en caracteres de la cadena que puede almacenar.
- Cada carácter de la cadena se almacena en una celda del *array.*
- Tras el último carácter de la cadena se añade y almacena el carácter de fin de cadena: '\0'.

Lectura de cadenas por teclado

Para almacenar en variables cadenas escritas mediante el teclado disponemos de dos funciones en C: scanf y gets.

scanf

La sintaxis general de la función scanf para almacenar o asignar a un *array* una cadena escrita mediante el teclado es:

```
scanf ("%s", &identificador);
```

Donde "%s" es el comando necesario para leer cadenas del teclado, e iden- tif icador es el nombre del *array* de tipo char que almacenará la cadena. Las siguientes líneas muestran un ejemplo en el que se pide una palabra al usuario y éste se almacena en el *array* palabra:

```
char palabra[10];

printf ("Escriba una palabra: ");
scanf ("%s", Apalabra);
```

La forma de almacenarse la cadena escrita (supongamos que es "silla") en el *array,* tras pulsar **Intro,** es: el primer carácter de la cadena se almacena en la primera celda, el segundo carácter de la cadena en la segunda celda, y así con todos y cada uno de los caracteres de la cadena escrita. El último carácter que se almacenará es el de fin de cadena: '\0'. El *array* tendrá el aspecto de la figura 11.5.

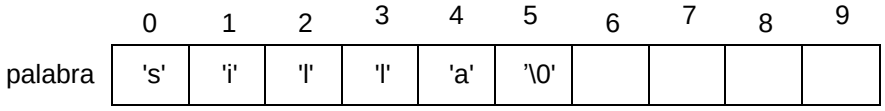


Figura 11.5. Estado de palabra tras almacenar "silla".

La función `scanf` presenta un inconveniente a la hora de leer del teclado cadenas que contengan espacios, y es que sólo almacena hasta el primer espacio que se encuentre. Si ejecutamos el siguiente listado:

```
char frase[100];

printf ("Escriba una frase: ");
scanf ("%s", &frase);
```

y cuando nos pida una frase escribimos "Esto es un ejemplo", en el *array* `frase` sólo se almacena "Esto".

gets

La función `gets`, de la biblioteca `stdio`, realiza la misma tarea que la función `scanf` con el comando "%s" para leer cadenas. La ventaja de `gets` es que permite almacenar espacios.
La sintaxis general de `gets` es más sencilla:

```
gets(identificador);
```

Donde `identif icador` es el nombre del *array* de tipo `char` que almacenará la cadena.
Si ejecutamos el anterior listado con la función `gets`:

```
char frase[100];

printf ("Escriba una frase: ");
gets(frase);
```

y volvemos a escribir "Esto es un ejemplo", en el *array* `frase` se almacena "Esto es un ejemplo".

Escritura de cadenas en pantalla

Para mostrar por pantalla una cadena almacenada en un *array* mediante la función `printf`, la sintaxis general es:

```
printf("%s", identificador);
```

Donde "%s" es el comando necesario para mostrar una cadena, e identi- f icador es el nombre del *array* que contiene la cadena.
La función printf mostrará en pantalla todos los caracteres que haya a partir de la primera celda del *array* hasta llegar al primer carácter de fin de cadena ('\0') que encuentre. El carácter '\0' no se muestra en la pantalla.
Ahora que ya sabemos mostrar variables de cadenas en pantalla podemos comprobar la diferencia entre scanf y gets cuando se introducen cadenas con espacios. Veamos qué ocurre al ejecutar el siguiente listado con scanf.

```
#include <stdio.h>

int main (void)
{
   /* Declaramos un array para almacenar la cadena.
   */ char frase [100];

   /* Pedimos una cadena al usuario y la almacenamos
   con scanf. */
   printf ("Escriba una frase (con scanf):
   "); scanf ("%s", Sfrase);

   /* Mostramos la cadena almacenada en el array. */
   printf ("\nContenido de frase: %s", frase);

   /* Hacemos una pausa hasta que el usuario pulse Intro
   */ fflush(stdin);
   printf("\n\nPulse Intro para
   finalizar..."); getchar();
}
```

Si el usuario escribe la frase "Esto es un ejemplo", el resultado en pantalla es el de la figura 11.6.

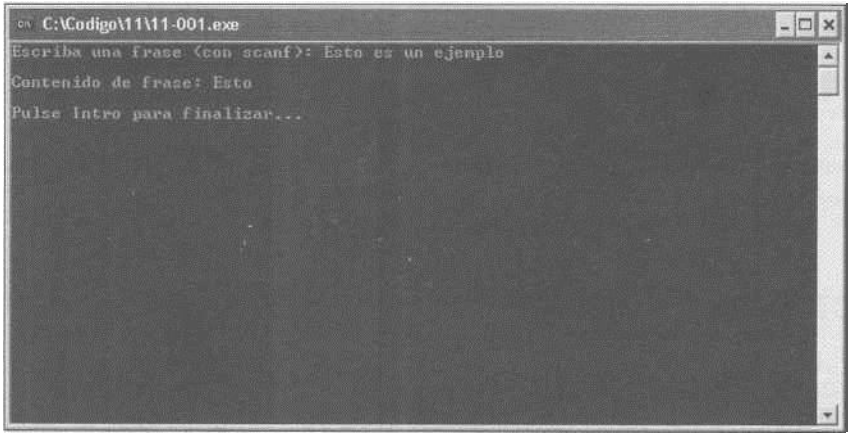


Figura 11.6. Resultado con scanf.

El mismo listado con gets.

```
#include <stdio.h>

int main (void)
{
   /* Declaramos un array para almacenar la cadena.
   */ char frase[100];

   /* Pedimos una cadena al usuario y la
   almacenamos con gets. */
   printf ("Escriba una frase (con gets):
   "); gets (frase);

   /* Mostramos la cadena almacenada en el array. */
   printf ("\nContenido de frase: %s", frase);

   /* Hacemos una pausa hasta que el usuario pulse Intro
   */ fflush(stdin);
   printf("\n\nPulse Intro para
   finalizar..."); getchar();
}
```

Si el usuario vuelve a escribir la misma frase el resultado en pantalla es el de la figura 11.7.

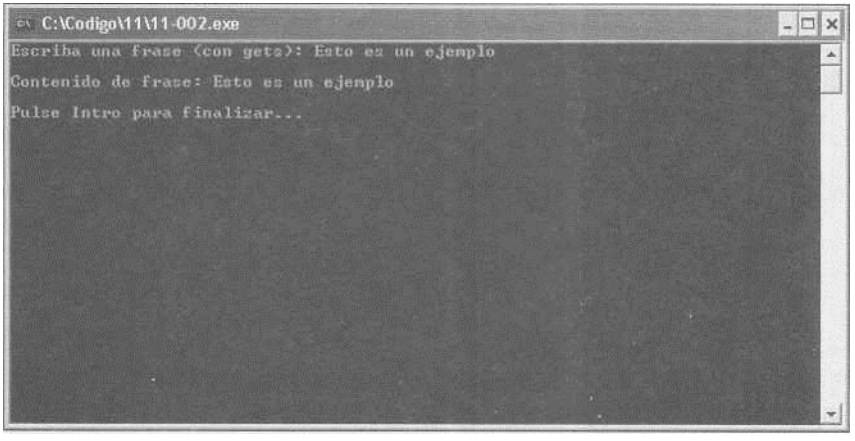


Figura 11.7. Resultado con gets.

Ejemplo

Antes de continuar vamos a realizar un par de ejemplos.
El primero consiste en pedir una frase al usuario y mostrar en la pantalla cuántas veces aparece la letra 'a' o 'A' en la cadena.

Lo que haremos será pedir una frase al usuario y almacenarla en un *array*. Después recorreremos el *array* desde la primera celda hasta que encontremos el carácter de fin de cadena ('\0'). Mientras lo recorremos comprobamos si en la celda hay un carácter 'a' o A', en cuyo caso incrementaremos un contador, que inicialmente valía cero.

```
#include <stdio.h>

int main (void)
{
   char frase[100];
   int celda;
   int contador = 0;

   /* Pedimos una frase al usuario y la almacenamos en
   el array frase. */
   printf ("Escriba una frase: ");
   gets (frase);

   /* Para recorrer el array empezamos en la celda 0. */
   celda=0;

   /* Recorremos el array mientras el contenido de la celda
   sea distinto del carácter de fin de cadena, es decir,
   mientras no lleguemos al final de la cadena. */ while
   (frase[celda]!='\0')
   {
      /* Si en la celda encontramos el carácter 'a' o 'A'
      incrementamos el valor del contador. */
      if ((frase[celda]=='a')   || (frase[celda]=='A'))
      {
         contador++;
      }

      /* Aumentamos el numero de la celda.
      */ celda++;
   }

   /* Mostramos el numero de caracteres 'a' o 'A' encontrados.
   */ printf ("\nSe han encontrado %d 'a' o 'A'.", contador);

   /* Hacemos una pausa hasta que el usuario pulse Intro */
   fflush(stdin);
   printf("\n\nPulse Intro para finalizar..."); getchar();
}
```

Estamos hablando de recorrer el *array* hasta encontrar el carácter de fin de cadena, así que lo que realmente estamos haciendo es recorrer la cadena almacenada en el *array*.

El segundo ejemplo es mucho más sencillo. El programa pide el nombre del usuario y muestra un saludo con su nombre.

```
#include <stdio.h>

int main (void)
{
    /* Declaramos el array que almacena el nombre del usuario. */ char
    nombre[70];

    /* Pedimos al usuario que escriba su nombre y lo almacenamos
    en el array nombre. */ printf ("Escriba su nombre: "); gets
    (nombre);

    /* Mostramos un saludo con el nombre del usuario. */ printf
    ("Hola, %s !", nombre);

    /* Hacemos una pausa hasta que el usuario pulse Intro */
    fflush(stdin);
    printf("\n\nPulse Intro para finalizar...");
    getchar();
}
```

Funciones de cadenas

La biblioteca string nos proporciona bastantes funciones para trabajar con cadenas, pero sólo estudiaremos algunas de las más necesarias.

Copiar cadenas: strcpy

Si quisiéramos asignar a un *array* una cadena seguramente haríamos lo siguiente:

```
char cadena[10]; cadena = "Hola";
```

Esto no es correcto. Para realizar esta operación usamos la función strcpy, que nos permite copiar una cadena en un *array*. La sintaxis general es:

```
strcpy (destino, origen)
```

Donde origen es una cadena o el identificador de un *array* con una cadena, y destino es el identificador del *array* que almacenará la cadena origen. La función strcpy copia la cadena origen en el *array* destino y añade el carácter '\0' al final de la cadena destino después de hacer la copia. Si el *array* destino almacenaba una cadena antes de una copia, dicha cadena es sustituida por la cadena origen tras la copia.

Habrá que prestar atención a que el *array* destino tenga espacio suficiente para almacenar la cadena origen.

El siguiente programa copia la cadena "Hola" en un *array* de tipo `char` llamado `saludo y` muestra en pantalla su contenido.

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

int main (void)
{
   char saludo[10];

   strcpy(saludo,
   "Hola"); printf ("%s",
   saludo);

   /* Hacemos una pausa hasta que el usuario pulse Intro */
   fflush(stdin);
   printf("\n\nPulse Intro para finalizar..."); getchar();
}
```

En la pantalla aparecerá "Hola". En la figura 11.8 se muestra el estado del *array* `saludo` después de su declaración (A) y después de `strcpy` (B).

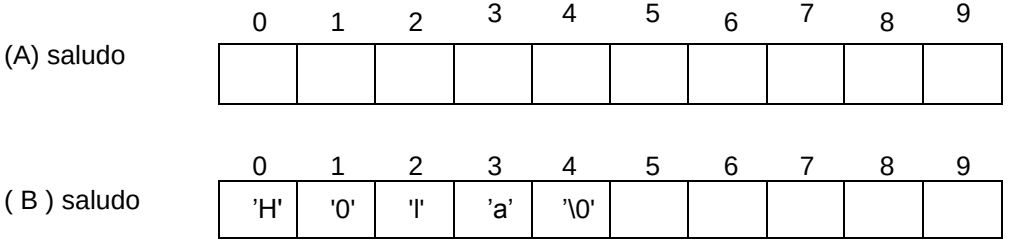


Figura 11.8. *Array* saludo.

En las siguientes líneas de código se pide una palabra al usuario (supongamos que escribe "libro") `y` se almacena en el *array* `palabra.`
La cadena almacenada en `palabra` se copia en otro *array* de tipo `char` llamado `copia.` Después, se muestra en pantalla la cadena almacenada en `copia` (que es "libro").

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

int main (void)
{
   char palabra[10]; char copia[10];

   printf ("Escriba una palabra: "); gets (palabra); strcpy
   (copia, palabra); printf ("%s", copia);
```

```
    /* Hacemos una pausa hasta que el usuario pulse Intro */
    fflush(stdin);

    printf("\n\nPulse Intro para finalizar...");
    getchar();
}
```

En la figura 11.9 se muestra el *array* copia tras su declaración (A), el *array* palabra tras almacenar con gets la palabra escrita por el usuario (B), y el *array* copia después de asignarle con strcpy la cadena almacenada en el

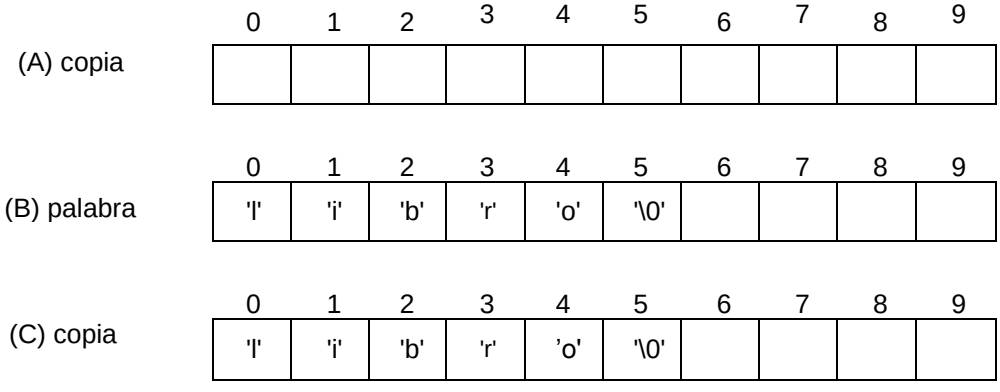


Figura 11.9. *Array* copia y palabra.

Concatenar cadenas: strcat

Al concatenar fusionamos dos cadenas; por ejemplo, partiendo de una cadena "Ho" y otra cadena "la" podemos obtener "Hola".
Para ello vamos a utilizar la función strcat, siendo su sintaxis general la siguiente:

```
strcat (destino, origen)
```

Donde origen es una cadena o el identificador de un *array* con una cadena, y destino es el identificador de un *array* con otra cadena. La cadena origen se añade a la cadena destino a partir de su carácter '\0', así que la cadena del *array* destino es modificada o ampliada. Al final de la cadena destino seguirá existiendo un carácter de fin de cadena.
Debemos prestar atención a que el *array* destino tenga espacio suficiente para añadirle la cadena origen.
Veamos un programa que concatene la cadena "Ho" almacenada en un *array* y la cadena "la" almacenada en otro.

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

int main (void)
{
   char cadenal [6];
   char cadena2 [6];

   strcpy (cadenal, "Ho");
   strcpy (cadena2, "la");
   strcat (cadenal, cadena2);
   printf ("%s", cadenal);

   /* Hacemos una pausa hasta que el usuario pulse Intro */
   fflush(stdin);
   printf("\n\nPulse Intro para finalizar...");
   getchar();
}
```

Este programa mostrará en la pantalla la cadena "Hola". En la figura 11.10 se muestra el *array* cadenal tras copiar en él "Ho" (A), el *array* cadena2 tras copiar en él la cadena "la" (B), y el *array* cadenal después de concatenarle cadena2 (C).

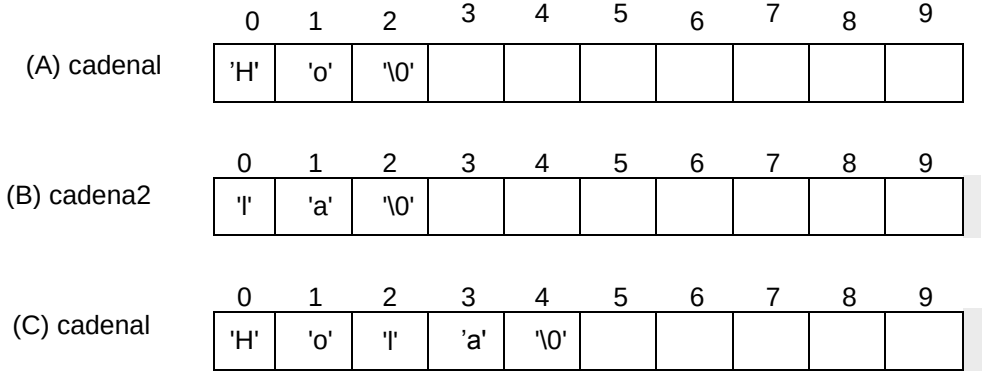


Figura 11.10./Array cadenal y cadena2.

Realicemos ahora un programa que pida el nombre al usuario (supongamos que es "Miguel") y construya en un *array* un mensaje de saludo con su nombre, que después será mostrado por pantalla.

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

int main (void)
{
   char nombre [10];
   char saludo [12] ;
```

```
    printf ("Escriba su nombre: ");
    gets (nombre); strcpy (saludo,
    "Hola "); strcat (saludo, nombre);
    printf ("%s", saludo);

    /* Hacemos una pausa hasta que el usuario pulse Intro */
    fflush(stdin);
    printf("\n\nPulse Intro para finalizar...");
    getchar ();
}
```

En la figura 11.11 se muestra el *array* nombre tras asignarle con gets el nombre del usuario (A), el *array* saludo después de copiar en él "Hola " (B), y el *array* saludo tras concatenarle nombre (C).

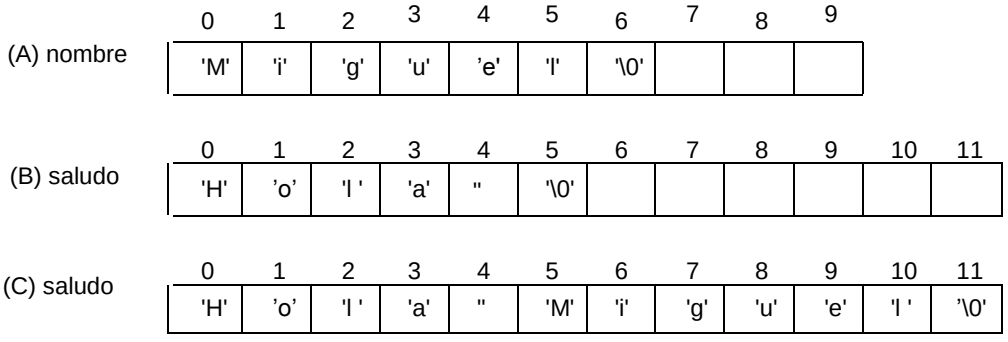


Figura 11.11. *Array* nombre y saludo.

Observe que tras la palabra "Hola " se ha dejado un espacio ('') con el fin de que al concatenar la palabra "Hola" y "Miguel" estén separadas.
En este programa, el tamaño del *array* saludo es muy justo. Esto nunca debería hacerse, porque si el nombre del usuario es mayor se saldría del *array* saludo tras la concatenación.
Dé un tamaño a sus *arrays* que le permitan despreocuparse de estos problemas, pero sin excederse; recuerde que las variables ocupan memoria y ésta no es infinita.

Tamaño de cadenas: strlen

La función `strlen` nos permite conocer el tamaño de una cadena. Dicho tamaño se expresa como el número de caracteres que contiene, sin contar el de fin de cadena. La sintaxis general es:

```
strlen (cadena)
```

Donde `cadena` es el nombre del *array* que almacena la cadena. La idea es que `strlen` empieza a contar el número de caracteres que hay desde la primera celda del *array* hasta que se encuentra el carácter '\0'. El argumento `cadena` también puede ser una cadena constante. Una vez obtenido el tamaño, lo retorna, es decir, toda la llamada a la función toma ese valor. En el siguiente listado:

```
char palabra[20];
int tamanio;

tamanio = strlen("silla");
```

`strlen` obtiene el tamaño de la cadena "silla", que es 5 por tener 5 caracteres. La llamada a esta función toma el valor que devuelve: 5. Así que la línea:

```
tamanio = strlen("silla");
```

es equivalente a:

```
tamanio = 5;
```

por lo que la variable `tamanio` toma el valor 5.
Veamos algunos ejemplos.
Si queremos mostrar en pantalla el tamaño de la cadena "Hola mundo" podemos hacerlo de dos formas:

```
/* Forma 1: usando una variable. */
int tamanio;

tamanio = strlen ("Hola mundo");   /* tamanio valdra 10. */

printf ("Tamaño: %d", tamanio);    /* En pantalla aparece: "Tamaño: 10" */

/* Forma 2: directamente. */

printf ("Tamaño: %d", strlen("Hola mundo"));
```

La segunda forma será equivalente a:

```
printf ("Tamaño: %d", 10);
```

En el siguiente ejemplo se pide una frase al usuario, se calcula su tamaño y se muestra por pantalla.

```
char frase[50];
int tamanio;

printf ("Escriba un frase: ") ;
gets (frase);

tamanio = strlen(frase);
printf("Tamaño: %d", tamanio)
```

Si el usuario hubiese escrito "Esto es un ejemplo", en pantalla se mostraría "Tamaño: 18".

Comparación de cadenas: strcmp

Con la función `strcmp` podemos comparar dos cadenas para comprobar si son iguales, distintas, o si una es mayor que otra.
Cuando decimos "si una es mayor que otra" no nos referimos a su tamaño en caracteres, sino al orden alfabético, por ejemplo, la 'A' es menor que la Z', porque el código ASCII de la 'A' es 65 y el de la 'Z' es 90, y 65 es menor que 90.
La sintaxis general es:

```
strcmp(cadena1, cadena2)
```

Donde `cadena1 y cadena2` son las dos cadenas a comparar.
La comparación entre las dos cadenas se hace carácter a carácter y comienza en el primer carácter de cada una de las cadenas y continúa con los siguientes, hasta llegar a una diferencia entre caracteres o al final de la cadena. La comparación entre caracteres se realiza teniendo en cuenta su valor o código numérico de la tabla ASCII. De esta manera, y como ya hemos explicado antes, el carácter 'A' (código 65) es menor que el carácter 'a' (código 97). Esto significa que distingue entre mayúsculas y minúsculas, porque distingue entre todos los caracteres de la tabla ASCII, incluidos números y signos de puntuación.
La función `strcmp,` tras ser ejecutada, devuelve un valor:

- Menor que cero `(<0)` si `cadena1` es menor que `cadena2.`
- Igual a cero `(==0)` si `cadena1` es igual que `cadena2.`
- Mayor que cero `(>0)` si `cadena1` es mayor que `cadena2.`

Como `strcmp` permite comparar dos cadenas se utiliza principalmente en condiciones y de la siguiente manera:

```
/* Para comprobar si cadena1 es menor que cadena2. */
if (strcmp(cadena1, cadena2) < 0)
/* Para comprobar si cadena1 es igual que cadena2. */
if (strcmp(cadena1, cadena2) == 0)
/* Para comprobar si cadena1 es mayor que cadena2. */
if (strcmp(cadena1, cadena2) > 0)
```

En el siguiente programa se piden dos palabras al usuario y se indica en pantalla cuál es la mayor, cuál es la menor y si son iguales o distintas.

```
#include <stdio.h>
#include <string.h>
int main (void)
```

```
{
   /* Declaramos un array para cada cadena. */
   char palabral [20] ; char palabra2[20];

   /* Pedimos la palabral al usuario. */ printf
   ("Escriba la palabral: "); gets (palabral);

   /* Pedimos la palabra2 al usuario. */ printf
   ("Escriba la palabra2: "); gets (palabra2);

   /* Comprobamos si la palabral es menor que la palabra2. */
   if (strcmp(palabral, palabra2) < 0)

      printf ("\nLa palabral es menor que la palabra2."); else

      printf ("\nLa palabra2 es menor que la palabral.");

   /* Comprobamos si la palabral es mayor que la palabra2. */
   if (strcmp(palabral, palabra2) > 0)

      printf ("\nLa palabral es mayor que la palabra2."); else

      printf ("\nLa palabra2 es mayor que la palabral.");

   /* Comprobamos si la palabral es igual que la palabra2. */
   if (strcmp(palabral, palabra2) == 0)

      printf ("\nLa palabral es igual que la palabra2."); else

      printf ("\nLa palabral es distinta de la palabra2.");

   /* Hacemos una pausa hasta que el usuario pulse Intro */
   fflush(stdin);

   printf("\n\nPulse Intro para finalizar..."); getchar();
}
```

Ahora, hagamos un programa que, dada la contraseña "HoLa", pida al usuario constantemente dicha contraseña hasta que la acierte, es decir, que sea igual que "HoLa".

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

int main (void)
{
   char clave[20];

   do
   {

      printf ("Contraseña: "); gets (clave);

   }

   while (strcmp(clave, "HoLa") != 0);
```

```
   printf ("\nAcceso concedido.");

   /* Hacemos una pausa hasta que el usuario pulse Intro */
   fflush(stdin);
   printf("\n\nPulse Intro para finalizar..."); getchar();
}
```

Buscar una cadena en otra cadena: strstr

Mediante la función `strstr` podemos buscar una cadena (aguja) dentro de otra (pajar). La sintaxis general es:

```
strstr(cadenal, cadena2)
```

Donde `cadena2` es la cadena buscada (aguja) en la `cadenal` (pajar). Si no se encuentra, la función retorna **NULL**, en caso contrario, retorna un puntero a la posición en la que ha encontrado `cadena2` dentro de `cadenal`.
En el siguiente ejemplo buscamos la aparición de la cadena "mensaje" en la cadena "Este es un mensaje de prueba". En lugar de cadenas estáticas podemos utilizar variables.

```
if (strstr("Este es un mensaje de prueba", "mensaje")!=NULL)
{
   printf ("Encontrado!");
}
else
{
   printf ("No encontrado.");
}
```

La función `strstr` es *case seiisitive,* es decir, distingue entre mayúsculas y minúsculas. De esta forma, si escribimos:

```
if (strstr("Este es un mensaje de prueba", "mensaje")!=NULL)
```

no obtendremos el mismo resultado que si escribimos:

```
if (strstr("Este es un mensaje de prueba", "Mensaje")!=NULL)
```

Convertir una cadena en minúsculas: strlwr

En algunas ocasiones es posible que deseemos convertir todos los caracteres de la A a la Z en minúsculas (de la a a la z). Para ello utilizaremos la función `strlwr,` no perteneciente a ANSI C, pero disponible en diversos compiladores, como Dev-C++. Su sintaxis general es:

```
strlwr(cadena)
```

En el siguiente ejemplo vamos a almacenar en la variable `frase` la cadena "AbCdEfl23" mediante la función `strcpy`. Seguidamente, aplicamos la función `strlwr` a `frase` para convertir la cadena almacenada a minúsculas. El resultado en pantalla es "abcdefl23". Como podemos comprobar, sólo los caracteres de la A a la Z son pasados a minúscula, ningún otro carácter.

```
#include <string.h>
#include <stdio.h>
int main(void)
{
   char frase[256];
   strcpy(frase,
   "AbCdEf123");
   strlwr(frase);
   printf("%s", frase);
   /* Hacemos una pausa hasta que el usuario pulse Intro */
   fflush(stdin);
   printf("\n\nPulse Intro para finalizar..."); getchar();
}
```

Convertir una cadena en mayúsculas: strupr

La función `strupr` es exacta a `strlwr`, la única diferencia es que convierte una cadena a mayúsculas. Su sintaxis general es:

```
strupr(cadena)
```

En el ejemplo anterior, si en lugar de `strlwr` utilizamos `strupr`:

```
strupr(frase);
```

el resultado en pantalla sería "ABCDEF123".

Trocear una cadena: strtok

La función strtok, del estándar ANSI C, nos permite obtener fragmentos de una cadena limitados por una determinada marca o carácter.
Su sintaxis general es:

```
strtok(cadena, marca)
```

La función `strtok` retorna un puntero al fragmento de cadena en el que aparece la marca. Pero mejor veamos cómo funciona y se usa a través de un ejemplo explicado.

```
#include <string.h>
#include <stdio.h>
int main(void)
{
   char palabras[250];
   char *p;
   /* Solicitamos al usuario una serie de palabras separadas
   por comas y que se almacenaran en la cadena 'palabras' */
   printf("Escribe varias palabras separadas por comas: ");
   gets(palabras) ;
   /* Obtenemos un puntero al primer fragemento de la cadena
   'palabras' que contiene el carácter coma. Si el puntero es
   distinto de NDLL, muestra la primera palabra en pantalla */
   p = strtok(palabras, ","); if (P)
   {
      printf("\n%s", p);
   }
   /* Mientras encontremos mas fragmentos con coma en la misma
   cadena, continuamos mostrándolos */ while (p = strtok (NULL,
   {
      printf("\n%s", p);
   }
   /* Hacemos una pausa hasta que el usuario pulse Intro */
   fflush(stdin);
   printf("\n\nPulse Intro para finalizar..."); getchar () ;
}
```

En primer lugar, se solicita al usuario que esciba una serie de palabras separadas por comas.

```
printf("Escribe varias palabras separadas por comas: "); gets(palabras);
```

Vamos a suponer que el usuario escribe mediante el teclado la cadena:

```
hierro,oro,mercurio
```

la cual es almacenada en la cadena palabras.
A continuación, se utiliza por primera vez la función strtok:

```
p = strtok(palabras, ",");
```

En esta primera vez, debemos pasar en el primer parámetro la cadena que queremos trocear, en nuestro caso palabras. En el segundo parámetro incluimos una cadena con el carácter que deseemos y que es empleado para

separar los fragmentos de la cadena, que en este ejemplo es el símbolo de la coma. Tras ejecutar la función, esta retorna un puntero al principio del primer fragmento de cadena hasta la marca, que se corresponde con la palabra "hierro". Dado que el resultado es distinto de **NULL**, se cumple la condición de la sentencia **if** y se ejecuta:

```
printf("\n%s", p);
```

que muestra en pantalla dicha palabra.

```
while (p = strtok(NULL, ","))
```

A continuación, se ejecuta la sentencia **while**, cuya condición vuelve a usar la función **strtok**, pero con una diferencia, ahora, en lugar de una cadena se especifica NULL. Esto se hace para indicar que no queremos volver a trocear la misma cadena desde el principio, sino que queremos usar la misma cadena y continuar troceándola desde el punto donde lo dejamos anteriormente.

La primera vez que se utiliza **strtok** en la condición del bucle **while** se obtiene un puntero al segundo fragmento de la cadena **palabras** que contiene una coma. En esta ocasión será un puntero al fragmento "oro". Como la obtención del fragmento ha tenido éxito, el valor retornado por **strtok** es disinto de NULL **y** la condición el **while** es positiva, ejecutándose el bloque de código de éste:

```
printf ("\n%s", p);
```

Una vez más, se muestra en pantalla la siguiente palabra de las escritas por el usuario. En la siguiente iteración del bucle **while**, **strtok** obtiene el último fragmento, "mercurio", también mostrado en pantalla. Una última iteración no tiene éxito, puesto que **strtok** no encuentra el símbolo coma: el programa finaliza.

No es necesario que siempre utilicemos el mismo símbolo en las sucesivas llamadas a la función **strtok**, cada vez podemos indicar uno diferente.

Convertir una cadena a número: atoi

En algunas ocasiones, nos encontraremos con números almacenados en variables tipo cadena y con los cuales queremos realizar operaciones matemáticas. En estos casos, podemos usar la función **atoi**, de la librería **stdlib** y del estándar ANSI C. Su sintaxis general es:

```
atoi(cadena)
```

La función **atoi** recibe por parámetro la cadena con números y retorna los números en formato **int**.

Veamos un ejemplo:

```
#include <string.h>

#include <stdio.h>

#include <stdlib.h>

int main(void)
{
    char cadena[20];
    int numero;

    /* Mostramos una cadena formada
    por números */ strcpy(cadena,
    "146"); printf ("\n%s", cadena);

    /* Convertimos la cadena formada por
    números a un numero entero y lo mostramos
    */ numero = atoi(cadena); printf("\n%d",
    numero);

    /* Operamos con la variable numérica
    y mostramos el resultado en pantalla
    */ numero = numero + 100;
    printf("\n%d", numero);

    /* Hacemos una pausa hasta que el usuario pulse Intro */
    fflush(stdin);
    printf("\n\nPulse Intro para finalizar...");
    getchar();
}
```

Una vez obtenemos mediante atoi el número almacenado en cadena y lo almacenamos en la variable numero, ya podemos realizar operaciones matemáticas con él o tratarlo como un número, algo que no se puede hacer con la variable cadena, puesto que es un array de caracteres, es texto, no es un número.

Ejercicios resueltos

Intente realizar los siguientes ejercicios. Encontrará la solución al final de este libro.

1. Realice un programa que pida primero una frase al usuario y después un carácter. El programa debe mostrar en la pantalla el número de veces que aparece el carácter indicado por el usuario en la cadena.
2. Realice un programa que pida dos palabras al usuario y, a continuación, las vuelva a mostrar en orden alfabético.

3. Realice un programa que pida una frase al usuario. Después deben mostrarse los caracteres de dicha cadena en orden inverso. Por ejemplo, si el usuario escribió "Hola mundo", el programa debe mostrar posteriormente "odnum aloH".

Resumen

En este capítulo hemos aprendido a manejar las cadenas con algunas de las principales funciones proporcionadas por la biblioteca **string**. No obstante, en esta biblioteca podemos encontrar otras muchas funciones de gran utilidad.

El hecho de que en C las cadenas se creen mediante *arrays* de tipo **char** nos permite realizar un gran número de operaciones con ellas si lo deseamos, pero también podemos trabajar con cadenas sin pensar que son *arrays,* sino simples variables que almacenan cadenas de caracteres.

Capítulo 12

Estructuras

En este capítulo aprenderá:

- Qué son las estructuras.
- Cómo manejár las estructuras.
- Cómo tener distintos tipos de datos en un mismo *array*.

Introducción

Los *arrays* nos permiten almacenar distintos datos, pero todos del mismo tipo. Recordemos aquel programa del capítulo de *arrays* en el que teníamos un *array* bidimensional con los números de expediente, tres notas y la nota media de seis alumnos. Todos estos datos eran del mismo tipo. Si hubiésemos querido añadir un dato más como el nombre del alumno no hubiera sido posible, porque el nombre es un *array* de tipo `char` y los otros datos son de tipo `int`.

Por otro lado, almacenar los datos de los alumnos en un *array* bidimensional no está mal, pero sería todavía mejor almacenarlos de alguna forma más ordenada.

En la vida cotidiana, y recurriendo al papel, cuando queremos almacenar los datos de un alumno utilizamos una ficha, en la que incluso se puede poner una foto, como la de la figura 12.1.

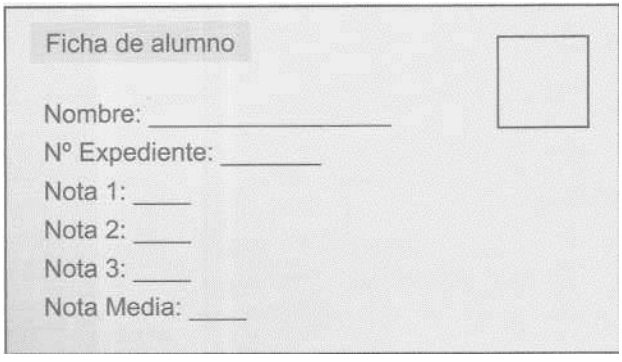


Figura 12.1. Ficha en papel de un alumno.

A "Nombre", "N° Expediente", "Nota 1", "Nota 2", "Nota 3" y "Nota Media" nos referimos como campos. La idea es crear y utilizar esas fichas en nuestros programas, pero sin fotos, claro. En programación, esas fichas son las estructuras y los campos de las fichas son variables declaradas dentro de una estructura.

Las estructuras admiten variables de distinto tipo y, también, podemos crear *arrays* unidimensionales de una estructura, es decir, cada celda almacenará una estructura. De esta manera conseguimos juntar variables de distintos tipos en un *array*. ¡Hemos resuelto el problema!

En la figura 12.2 se muestra una representación gráfica de un *array* de estructuras con cuatro celdas.

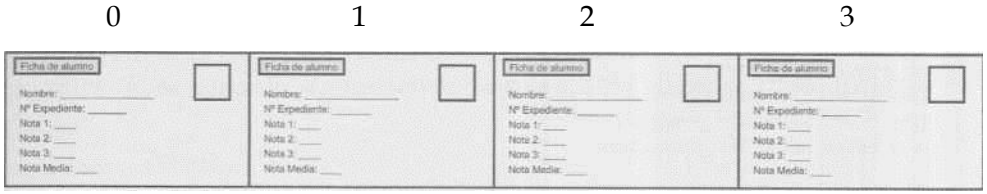


Figura 12.2. Representación gráfica de un array de estructuras.

Todo es muy parecido a la vida cotidiana. Un *array* unidimensional de estructuras con cuatro celdas es igual que tener un pequeño montón con cuatro fichas. Si queremos buscar la ficha de un alumno para ver su nota media comprobamos ficha a ficha su nombre hasta encontrarle y, después, consultamos su nota media. En un *array* de estructuras comprobaríamos celda a celda hasta encontrar el nombre del alumno buscado y, entonces, consultaríamos su nota media.

Declaración

Para utilizar una estructura primero hay que crearla, que es como crear un molde de imprenta para una ficha, pero que en principio no sirve para almacenar datos. Posteriormente, declararemos variables de esa estructura, que es como obtener un ejemplar de una ficha a partir del molde. Esa variable de la estructura contiene los campos donde ya podemos almacenar los datos. En la figura 12.3 podemos ver una representación de lo que haremos en código.

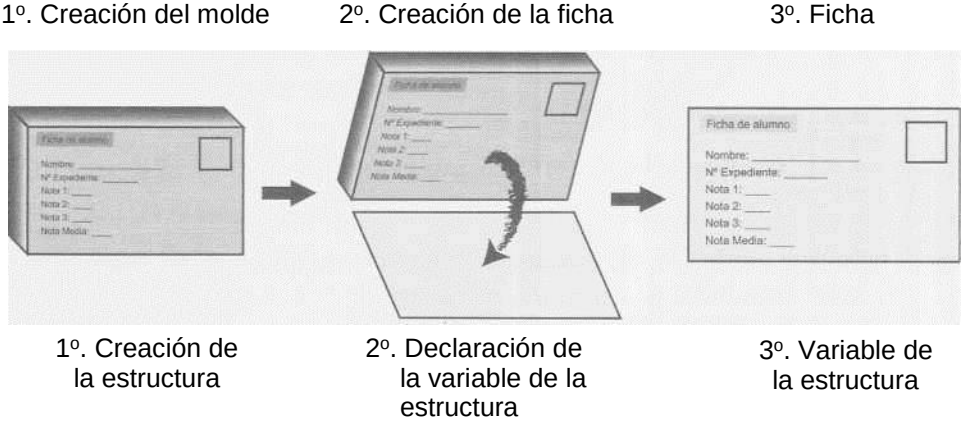


Figura 12.3. Paso de la creación de la estructura a la variable de la estructura.

La forma general de crear estructuras es:

```
struct identificador_s {
    tipo identificador_campo1;
    tipo identificador_campo2;
    tipo identificador_campoN;
};
```

Donde `struct` es una palabra reservada y necesaria para crear la estructura, e `identif icador_s` es el nombre que daremos a la estructura, que sería igual al nombre del molde. A continuación, y entre llaves, se declaran las variables, que se corresponden con los campos de la ficha. Observe que al final de la definición de la estructura hay punto y coma (;).
A las estructuras les damos un nombre porque podemos tener más de una y necesitaremos identificarlas de alguna manera. Para crear una estructura equivalente a la ficha de la figura 12.1 escribimos:

```
struct Tficha {

    char nombre[30];
    int numero_exp;
    int nota_1,
    nota_2, nota_3;
    int nota_media

} ;
```

El nombre de esta estructura es `Tficha` y contiene estos 6 campos: `nombre, numero_exp, nota_1, nota_2, nota_3` y `nota_media.` Como puede ver, las variables se declaran como siempre.
No olvide el punto y coma tras la definición.
Las definiciones de estructuras se pueden escribir dentro de la función `main,` pero es conveniente escribirlas encima de ésta para que puedan ser accesibles por todo el programa:

```
/* Aqui van los #include */

/* Aqui van los #define */

struct Tficha {

    char nombre[30]; int numero_exp; int nota_1,
    nota_2, nota_3; int nota_media

} ;

int main (void)

{

    /* Aqui va el codigo del programa principal */

}
```

Ya tenemos la estructura. Para declarar una variable de una estructura la

```
struct identificador s variable s;
```

Donde `identificador_s` es el nombre que le dimos a la estructura y `variable_s` es el identificador de la variable que vamos a crear de esa estructura. Esta sintaxis sigue siendo la misma de siempre para declarar variables:

```
tipo identificador;
```

pero en este caso `struct identificador_s` actúa como el tipo de dato de la variable. Para declarar una variable de la estructura anterior escribimos:

```
struct Tficha {
    char nombre[30];
    int numero_exp;
    int nota_1, nota_2, nota_3;
    int nota_media

};

int main (void)
{
    struct Tficha mi_ficha;
}
```

Donde **mi_ficha** es una variable de la estructura **Tficha**. También podemos declarar variables de estructuras de la forma:

```
struct Tficha mi_ficha1, mi_ficha2, mi_ficha3;
```

Nota:

Como en la declaración de una variable de una estructura aparecen tres palabras (struct, identificador de la estructura e identificador de la variable de la estructura), para no confundir el identificador de la estructura con el de la variable, al nombre de la estructura le podemos añadir una letra "T" o la palabra "tipo" de alguna manera, por aquello de que junto con la palabra reservada struct actúa como un tipo de dato.

Entonces, si la sintaxis para declarar un *array* unidimensional es:

```
tipo identificador[Tamaño];
```

y queremos declarar un *array* unidimensional de 10 celdas con estructuras **Tficha**, como **struct Tficha** actúa como un tipo de dato, tenemos que escribir:

```
struct Tficha mi_array[10];
```

El resultado es un *array* llamado **mi_array** de 10 celdas, y en cada celda hay una estructura **Tficha**.

También se pueden declarar variables de una estructura a la vez que se define. La sintaxis es:

```
struct identificador_s {
    tipo identificador_campo1;
    tipo identificador_campo2;
    tipo identificador_campoN;
} variable_s;
```

Donde `variable s` es el identificador de la variable de la estructura llamada `identificador_s.`

Ejemplo:

```
struct Tficha {
    char nombre[30];
    int numero_exp;
    int nota_1, nota_2, nota_3;
    int nota_media
} mi_ficha;
```

Acceso a los campos

Ahora nos falta saber cómo acceder a los campos de las variables de las estructuras. Pues bien, es tan fácil como escribir la siguiente sintaxis:

```
variable_s.identificador_campo
```

Donde variable_s es el identificador de la variable de la estructura, e identif icador_campo es el nombre de la variable declarada en la estructura a la que queremos acceder. Observe que entre los dos identificadores hay un punto. Así, para acceder al campo nota_l de la variable mi_ficha (que es una estructura Tficha) escribimos primero el nombre de la variable de la estructura, un punto y el nombre del campo:

```
mi_ficha.nota_1
```

La idea es que cuando accedemos a un campo con lo que estamos trabajando realmente es con una variable, una simple variable de las de siempre, pero que está dentro de una estructura. Y como es una simple variable podemos hacer todas las siguientes operaciones y más, pero siempre indicando a qué variable de estructura pertenece:

```
mi_ficha.nota_l = 6;              /* Asignarle un valor. */
printf ("%d", mi_ficha.nota_l);   /* Mostrar su valor en pantalla. */
scanf ("%d", &mi_ficha.nota_l);   /* Asignarle un valor por teclado. */
mi_ficha.nota_l = mi_ficha.nota_l + 2; /* Incrementar su valor. */
mi_ficha.nota_1—;                 /* Decrementar su valor. */
```

Veamos el ejemplo de un programa en el que se crea una estructura con tres campos: nombre, edad y teléfono. El usuario debe rellenar estos datos por teclado. Posteriormente se mostrarán los datos de la estructura.

```
#include <stdio.h>

/* Creamos la estructura"tipo_ficha".*/
   struct tipo_ficha {
   char nombre[50];
   int edad;
   char telefono[20] ;
};

int main (void)
{
    /* Declaramos un variable de la estructura "tipo_ficha". */
    struct tipo_ficha persona;

    /* Pedimos los datos al usuario y los almacenamos en cada
    uno de los campos de la estructura. */
    printf ("Escriba su nombre: ");
    gets (persona.nombre);
    printf ("Escriba su edad: ");
    scanf ("%d", spersona.edad);
    fflush (stdin);
    printf ("Escriba su telefono: ");
    gets (persona.telefono);

    /* Mostramos los datos almacenados en la estructura. */

    printf ("\nLa edad de %s es %d", persona.nombre, persona.edad);
    printf ("\ny su telefono es %s.", persona.telefono);

    /* Hacemos una pausa hasta que el usuario pulse Intro */
    fflush(stdin);
    printf("\n\nPulse Intro para finalizar...");
    getchar();
}
```

Si el nombre del usuario es Ana, su edad 10 y su teléfono 000-000-000, el estado de la variable persona sería el de la figura 12.4.

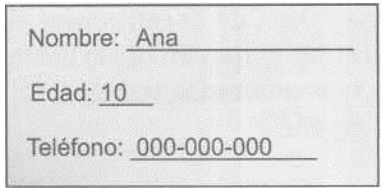


Figura 12.4. Estado de la variable persona.

El resultado en pantalla del programa es el de la figura 12.5.

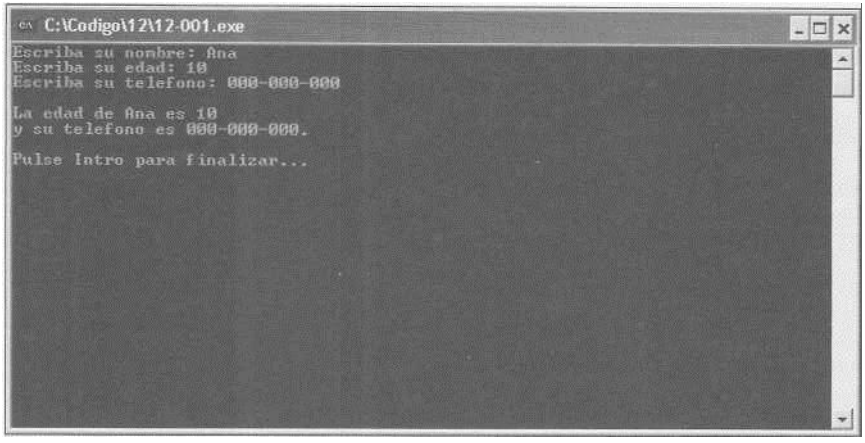


Figura 12.5. Resultado del programa.

Estructuras y arrays

Ya hemos dicho en el apartado "Declaración" cómo se crean *arrays* de estructuras. Pero ¿cómo se manejan?
Si tenemos la siguiente estructura:

```
struct tipo_libro {
   char
   titulo[30];
   int paginas;
};
```

para registrar libros y su número de páginas, también necesitaremos un *array* de esta estructura, porque tendremos varios libros. Si suponemos que tenemos 10 libros la declaración es:

```
struct tipo_libros libros[10];
```

Donde `libros` es el nombre del *array* y `struct tipo_libros` el tipo de dato.
Para acceder al campo `paginas` de la estructura almacenada en la celda 0 escribimos el nombre del *array*, el número de la celda (entre corchetes) donde está la estructura con la que queremos trabajar, un punto y el nombre del campo al que queremos acceder:

```
libros[0].paginas
```

Así es como se accede a cualquier campo de las estructuras de un *array*. Si queremos rellenar por teclado el campo `tituloypaginas` de la celda 0, escribimos:

```
printf ("Escriba el titulo del libro: ") ; gets
(libros[0].titulo);
printf ("Escriba el numero de paginas del libro: "); scanf
("%d", Slibros[0].titulo);
```

Y si queremos almacenar el título y el número de páginas de 10 libros podemos hacerlo con un bucle f or:

```
for (celda=0; celda<=9; celda++)
{
    printf ("\nEscriba el titulo del libro %d: ", celda+1);
    gets (libros[celda].titulo);
    printf ("Escriba el numero de paginas del libro %d: ", celda+1);
    scanf ("%d", Slibros[celda].titulo);
}
```

Hagamos ahora, utilizando las estructuras, el programa del capítulo de *arrays* en el que teníamos un *array* bidimensional con los números de expediente, tres notas y la nota media de seis alumnos. Necesitaremos una estructura con cinco campos: número de expediente, nota 1, nota 2, nota 3 y nota media. Si tenemos 6 alumnos necesitaremos un *array* unidimensional con 6 celdas de estas estructuras (véase la figura 12.6).

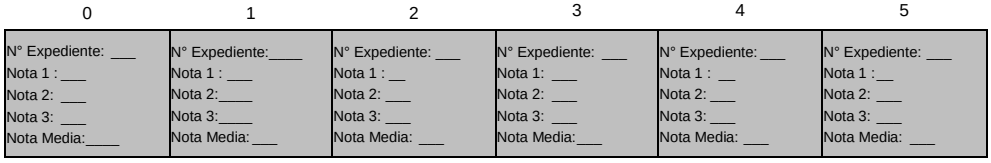


Figura 12.6. Array de estructuras.

El programa pide al usuario que rellene el número de expediente de cada alumno y sus tres notas. Posteriormente, se calcula la nota media de cada alumno y se muestra el número de expediente del alumno con la mayor nota media y dicha nota.

```
#include <stdio.h>
struct tipo_alumno {
    int numero_exp;
    int nota1, nota2, nota3;
    int nota_media;
};
int main (void)
{
    struct tipo_alumno alum[6]; /* Array de estructuras. */
    int n;                      /* Para recorrer las celdas. */
    int media_mayor;            /* Almacena la media mayor. */
    int mejor_exp;              /* Almacena el expediente con mayor media. */
```

```
    /* Rellenamos los campos expediente, notal, nota2 y nota3 de cada
    alumno. */ for (n=0; n<=5; n++)
    {
       /* Pedimos el numero de expediente y las tres notas. */
       printf ("\nEscriba el numero de expediente: ");
       scanf("%d", &alum[n].numero_exp);
       printf ("\nEscriba la nota 1: ");
       scanf ("%d", &alum[n].notal);
       printf ("\nEscriba la nota 2: ");
       scanf ("%d", &alum[n],nota2);
       printf ("\nEscriba la nota 3: ");
       scanf ("%d", &alum[n].nota3);
    }
    /* Calculamos la nota media de cada alumno. */ for (n=0; n<=5; n++)
    {
       suma = alum[n].notal + alum[n].nota2 + alum[n].nota3; alum[n].nota_media
       = suma / 3;
    }
    /* Calculamos la nota media mayor y obtenemos dicha nota.
    Partimos de la idea de que la mayor nota es 0. */
    media_mayor=0;
    for (n=0; n<=5; n++)
    {
       /* Si estamos ante una nota mayor que la almacenada en
       media_mayor, entonces esta nueva nota pasa a ser la media
       mayor y almacenamos el numero de expediente de ese alumno.
       */ if (alum[n].nota_media > media_mayor)
       {
          media_mayor = alum[n].nota_media;
          mejor_exp = alumfn].numero_exp;
       }
    }
    /* Mostramos en la pantalla unos mensajes que indican el numero
    de expediente del alumno con la nota media mayor y dicha nota. */
    printf ("\n\nLa mejor media:");
    printf ("\n\tExpediente: %d", mejor_exp);
    printf ("\n\tNota Media: %d\n", media_mayor);
    /* Hacemos una pausa hasta que el usuario pulse Intro */
    fflush(stdin);
    printf ("\n\nPulse Intro para finalizar...");
    getchar ();
}
```

El resultado final de este programa es exacto al del capítulo de *arrays,* pero ya no tenemos que preocuparnos de las filas y columnas en el *array,* y si queremos podemos añadir un campo que sea el nombre del alumno. Más

Ejercicios resueltos

Intente realizar los siguientes ejercicios. Encontrará la solución al final de este libro.

1. Escriba la definición y dibuje las siguientes estructuras:
   - Una estructura que permita almacenar una fecha. Tendrá 3 campos: día, mes y año.
   - Una estructura que sirva para catalogar los libros de una biblioteca. Tendrá los siguientes campos: título, autor, tema, ISBN y número de páginas. El ISBN es un número de identificación de 10 dígitos.
2. Realice un programa que almacene el nombre de seis jugadores de videojuegos y la puntuación máxima de cada uno. Posteriormente, debe mostrarse en pantalla el nombre del jugador con el mayor número de puntos y el del jugador con el menor número de puntos.

Resumen

Una de las ventajas que ofrecen las estructuras es la de agrupar una serie de datos bajo un nombre. Esto nos permite tener una visión más estructurada y lógica del programa y sus datos.
Las estructuras también nos han resuelto el problema de almacenar datos de distintos tipos en un mismo *array*.
Puede parecer que no tienen una gran utilidad, pero todas las características que hemos mencionado son muy valiosas. En programación el orden, la claridad y la lógica son piezas clave.

Capítulo 13

Funciones

En este capítulo aprenderá:

- Qué son las funci ones.
- A crear sus propias funciones.
- A llamar a las funciones.
- Cómoretornar datos con las funciones.
- El paso de parámetros por valor y por referencia.
- Qué es el ámbito local y global.
- Qué son las funciones recursivas.

Introducción: Divide y vencerás

En la vida cotidiana las tareas suelen estar repartidas. Por ejemplo, en una fábrica de bombones como la mostrada en la figura 13.1 existen distintos departamentos: uno para hacer los bombones, otro para hacer los envoltorios y el último, que pone los bombones en los envoltorios.

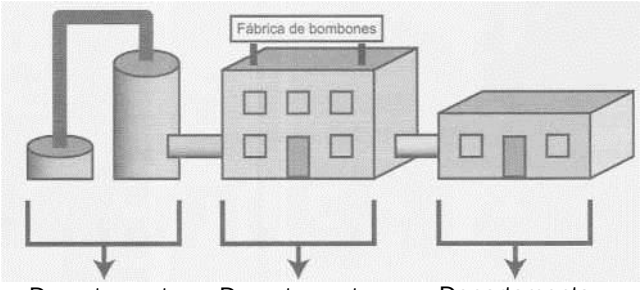


Figura 13.1. Departamento de una fábrica de bombones.

Si la fábrica de bombones no estuviese dividida en departamentos, sino que las distintas tareas estuviesen entremezcladas, en el caso de que ocurriese un error sería muy difícil determinar exactamente la causa. Pero con la fábrica dividida en tareas es muy fácil localizar fallos, con comprobar el estado en el que salen los bombones o los envoltorios de sus respectivos departamentos podemos saber en qué departamento está la avería y resolver el problema.
Otro ejemplo. Piense en las desventajas que tiene un ordenador portátil frente a un PC en caso de avería de alguno de sus componentes. ¿Cuánto nos puede llevar sustituir cualquier componente de nuestro PC? Lo que tardemos en ir a nuestra tienda de informática, comprar otro y sustituirlo nosotros mismos. Esto es gracias a que los componentes del PC, como el monitor, el teclado y el ratón, están divididos.
Así que el éxito parece estar en dividir un problema grande en problemas menores y éstos, a su vez, en otros menores, así hasta llegar a un nivel en el que los problemas sean fáciles de resolver.
La programación modular sigue esta estrategia, y nosotros también la seguiremos dividiendo el programa principal en subprogramas. Los subprogramas también reciben otros nombres, como subrutinas o módulos. En C/C++ reciben el nombre de funciones.

Las ventajas de la programación modular son:

- Nos permite descomponer el programa en subprogramas.
- La verificación de los subprogramas se puede hacer por separado: para comprobar un subprograma no es necesario tener que probar el resto del programa.
- Los programas son más claros.
- Los programas se pueden crear entre varias personas: cada persona puede crear un subprograma y posteriormente juntarlos en un programa.

¿Y cómo dividimos un programa principal en subprogramas? Pues observamos el programa principal para comprobar si contiene trozos de código que realizan tareas determinadas: ordenar un *array* unidimensional, rellenar un *array,* controlar el acceso al programa mediante la petición de una contraseña, etc. Esos trozos de código los extraemos del programa principal y les damos un nombre identificativo, obteniendo las funciones (o subprogramas). Para ejecutarlos escribimos sus nombres en el programa principal. En la figura 13.2 vemos un esquema de este proceso.

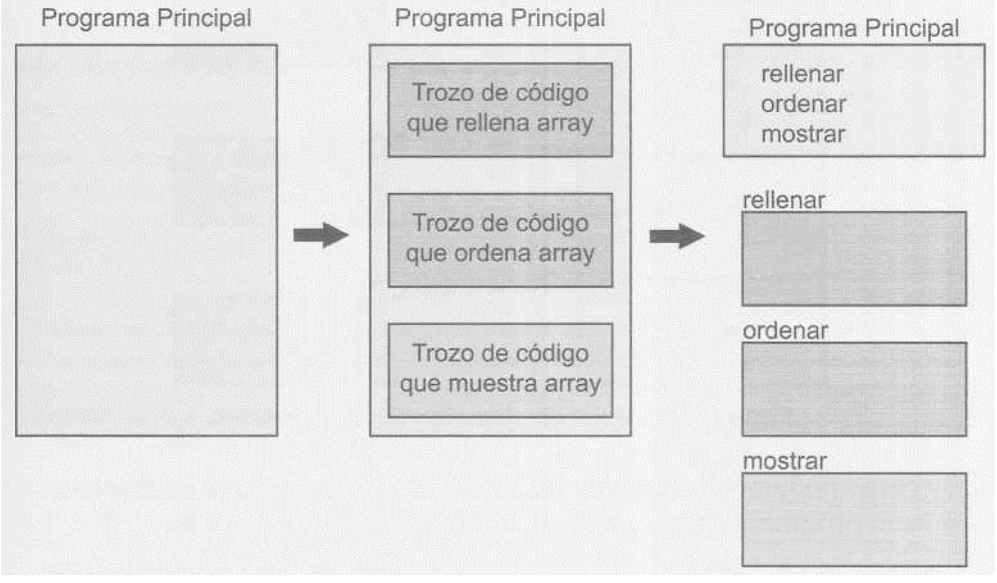


Figura 13.2. Paso del programa principal a la obtención de funciones.

En este esquema se muestra un programa principal, en el que tras ser observado se aprecian tres trozos de código que realizan tareas muy determinadas: rellenar un *array,* ordenar un *array* y mostrar un *array.*

Posteriormente, esos trozos de código se extraen del programa principal y se les da un nombre: rellenar, ordenar y mostrar, obteniendo tres funciones o subprogramas. En el lugar donde se encontraban esos trozos de código en el programa principal se escriben los nombres de las funciones.
En ese programa se seguirá ejecutando en primer lugar el programa principal (que en C/C++ es la función `main`) .
Cuando se encuentra el nombre de la función `rellenar` se ejecuta todo el código de ésta y, después, se continúa ejecutando la línea de código siguiente del programa principal.
Al llegar al nombre de la función `ordenar` se ejecuta todo el código de ésta y, a continuación, se continúa ejecutando la línea de código siguiente del programa principal. Por último, al llegar al nombre de la función `mostrar` se ejecuta todo el código de ésta y, después, se continúa ejecutando la línea de código siguiente del programa principal, hasta que finaliza.
Podemos ver este proceso en la figura 13.3, donde las flechas representan el hilo de ejecución.

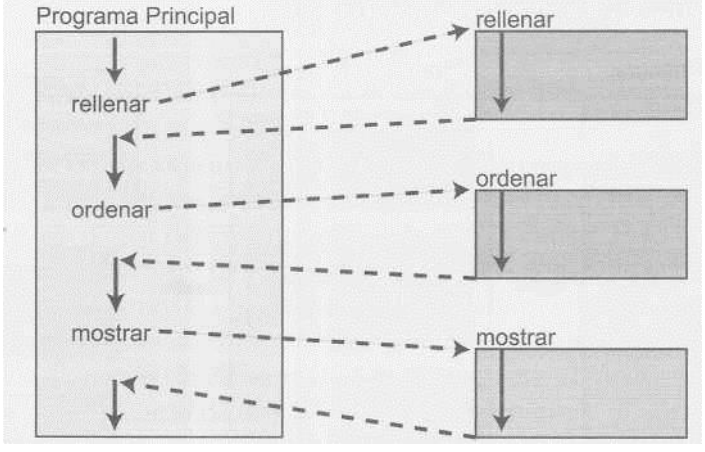


Figura 13.3. Modo de ejecución de las funciones.

Tal y como podemos comprobar, el resultado de la ejecución es equivalente al mismo programa cuando todo el código se encontraba junto.

Estructura de una función

Vamos a partir de una idea que sea muy simple de función y, poco a poco, vamos a ampliarla.

En principio, la sintaxis de una función es:

```
identificador ()

{
}
```

Donde identif icador es el nombre de la función, le siguen dos paréntesis y, por último, una llave abierta y otra cerrada. El código de la función se escribe entre las llaves. Observe que detrás de los paréntesis no se escribe punto y coma.
¡Vaya!, a simple vista es muy parecido a la función main.
En el siguiente ejemplo hemos escrito o bien definido una función llamada saludo que muestra un mensaje de bienvenida.

```
saludo ()
{
    printf ("Hola a todos");
}
```

Para utilizarla en un programa tenemos que declararla antes de la función main, para ello escribimos su prototipo seguido de un punto y coma:

```
saludo () ;
```

El prototipo de una función es el nombre de ésta más los paréntesis; bueno, sólo esto de momento.

```
#include <stdio.h>

/* Declaramos la función saludo. */

void saludo ();

/* Programa principal */

int main (void)

{

    /* Llamamos a la función saludo. */
    saludo();

    /* Mostramos mensaje de despedida. */
    printf ("\nAdios");

    /* Hacemos una pausa hasta que el usuario pulse Intro */
    fflush(stdin);

    printf("\n\nPulse Intro para finalizar...");

    getchar () ;

}

/* Función saludo */ void saludo ()

{

    printf ("Hola a todos");

}
```

La primera línea de código:

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

tenemos que escribirla porque en el programa (que es todo el listado) se hace uso de la función `printf`, que pertenece a la biblioteca `stdio`.
A continuación, está la declaración de la función `saludo`:

```
saludo ();
```

que indica que en algún lugar del programa existe una función que se llama `saludo`.
Dentro de la función `main` aparece la llamada a la función `saludo`:

```
saludo ();
```

que consta del nombre de la función y de los paréntesis. El punto y coma se escribe para indicar el final de la sentencia.
Cuando la ejecución encuentra esta línea se ejecuta el contenido de la función `saludo`:

```
printf ("Hola a todos");
```

mostrándose en pantalla el mensaje "Hola a todos". Tras finalizar el código de la función `saludo`, la ejecución continúa con la línea siguiente tras la llamada a la función:

```
printf ("\nAdios");
```

mostrándose en pantalla el mensaje "Adiós". Finalmente, el programa acaba.

Nota:

Quizás se pueda plantear una duda respecto a la declaración defunciones: ¿por qué no se declara la función main? La declaración de una función sirve para indicar que una función, que no tenía por qué existir, existe. La función main es la función principal, es como la puerta principal de un edificio. ¿Alguien ha visto alguna vez en un edificio un cartel que diga "Este edificio tiene una puerta principal"? No, porque se supone que dehe tenerla, o ¿por dónde entran las personas? ¿por las ventanas? Lo mismo ocurre con la función main, no es necesario declararla porque el compilador sabe que tiene que existir, porque es el lugar por donde comienza la ejecución del programa. No ocurre lo mismo con el resto defunciones que hagamos. ¿Alguien presupone que dentro de un edificio hay oficinas? No, luego habrá que poner carteles que lo indiquen. Igualmente, las funciones que hagamos tenemos que declararlas, indicando que existen.

Ahora que conocemos el orden de ejecución de un programa seguro que sabría decir qué hace el siguiente `y` qué aparecería en pantalla.

```
#include <stdio.h>

/* Declaramos la función tabla_multiplicar. */
void tabla_multiplicar ();

/* Programa principal */

int main (void)
{
   /* Mostramos mensaje de bienvenida. */
   printf ("\nHola");

   /* Llamamos a la función tabla_multiplicar. */
   tabla_multiplicar ();

   /* Mostramos mensaje de despedida. */
   printf ("\nAdios");

   /* Hacemos una pausa hasta que el usuario pulse Intro */
   fflush(stdin);

   printf("\n\nPulse Intro para finalizar...");
   getchar();
}
/* Definición de la función tabla_multiplicar */
void tabla_multiplicar ()
{
   int n;           /* Almacena los números que multiplican a 2. */

   int resultado;   /* Almacena el resultado de la multiplicación. */

   /* Mostramos en pantalla la tabla de multiplicar

   del numero 2. */

   for (n=l; n<=10; n++)

   {
      /* Obtenemos el resultado de la multiplicación. */
      resultado = 2 * n;

      /* Mostramos el resultado. */

      printf ("\n2 x %d = %d", n, resultado);

   }
}
```

Una "pequeña" pista. La ejecución comienza por el código de la función `main`, como siempre. Se muestra en pantalla un mensaje de bienvenida, se ejecuta el código de la función `tabla_multiplicar` y se muestra en pantalla un mensaje de despedida. ¿Ya?

Por cierto, a riesgo de parecer pesados, vuelva a observar que el código de las funciones que hemos hecho no tiene nada de especial. También podemos definir la función `tabla multiplicar` de manera que dentro de ésta se pregunte al usuario por el número del que quiere mostrar su tabla:

```
/* Definición de la función tabla_multiplicar */
void tabla_multiplicar ()
{
   int numero;       /*  Almacena el numero dado por el usuario. */
   int n;            /*  Almacena los números que multiplican a "numero". */
   int resultado;    /*  Almacena el resultado de la multiplicación.*/
   /* Pedimos el numero  al usuario. */
   printf ("\nEscriba un numero: ");
   scanf ("%d", &numero);
   /* Mostramos en pantalla la tabla de multiplicar
   del numero "numero". */
   for (n=1; n<=10; n++)
   {
      /* Obtenemos el resultado de la multiplicación. */
      resultado = numero * n;
      /* Mostramos el resultado. */
      printf ("\n%d x %d = %d", numero, n, resultado);
   }
}
```

Paso de parámetros por valor

Para explicar los próximos conceptos representaremos a las funciones como una caja que realiza una tarea gracias al código que contiene. También les daremos el nombre de la función para identificarlas. Así, una representación gráfica de la última función que hemos hecho, `tabla_multiplicar`, la podemos ver en la figura 13.4.

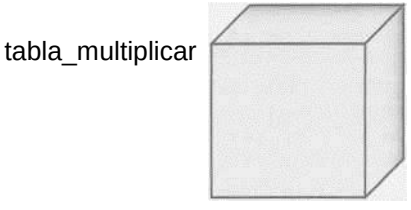


Figura 13.4. Representación de la función tabla_multiplicar.

La caja de la función `tabla_multiplicar`, cuando se ejecuta, muestra en la pantalla la tabla de multiplicar de un número dado por el usuario. Pero, ¿podríamos introducir de alguna manera en esa caja el número en vez de pedírselo al usuario? Sí, gráficamente crearíamos una ranura en la caja por la que introducir el número, por ejemplo el 4. A esa ranura hay que darle un nombre que escribimos junto a ella, por ejemplo "A", ya que podrían existir

varias ranuras y el código necesitaría hacer referencia a una u otra de alguna manera. El código de la caja espera que por la ranura entre un número y no otra cosa, como un carácter. Así que junto a la ranura también tenemos que indicar el tipo de dato que hay que introducir, en este caso int. Una vez introducido el número, el código de la caja muestra en pantalla su tabla de multiplicar. Véase la figura 13.5.

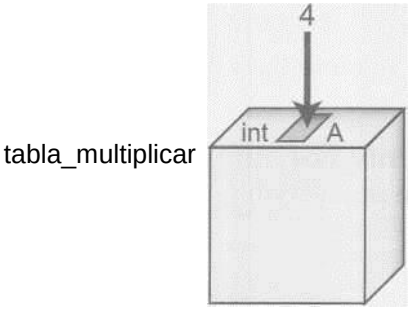


Figura 13.5. Creamos una ranura en la caja para introducir un número.

En C/C++ estas ranuras reciben el nombre de parámetros y se crean entre los paréntesis que hay tras el nombre de la función.
Los parámetros también deben tener un tipo de dato y un nombre, que se escriben de la misma manera que se declara una variable. El prototipo de la función **tabla_multiplicar** con el parámetro **A** sería:

```
tabla_multiplicar (int A)
```

El equivalente a introducir un valor por una ranura es pasar un valor o dato por parámetro.
Pero, ¿cómo sería el código de la función **tabla_multiplicar** para que calcule la tabla del número pasado por parámetro? Cuando pasemos por parámetro un valor a la función, dicho valor se le asignará al parámetro **A**, que realmente es una variable normal, de las que hemos utilizado siempre, por lo que la podemos tratar como tal. Así, si pasamos el valor 4 por parámetro a la función, **A** tomará el valor 4. Luego el código tendrá que calcular la tabla de multiplicar del valor de la variable **A**, en este caso del 4.

```
/* Definición de la función tabla_multiplicar */
void tabla_multiplicar (int A)
{
   int n;           /* Almacena los números que multiplican a A. */
   int resultado;   /* Almacena el resultado de la multiplicación. */
   /* Mostramos en pantalla la tabla de multiplicar del numero A. */
   for (n=1; n<=10; n++)
```

```
   {
      /* Obtenemos el resultado de la multiplicación. */
      resultado = A * n;
      /* Mostramos el resultado. */
      printf ("\n%d x %d = %d", A, n, resultado);
   }
}
```

Cuando llamemos a la función, como tiene un parámetro, tenemos que pasarle un valor. Para pasar un valor por parámetro escribimos el nombre de la función seguido de los paréntesis, y dentro de éstos el valor. Por ejemplo:

```
tabla_multiplicar (4);
```

Cualquier valor que escribamos entre los paréntesis se le asignará al parámetro **A**, ya sea el valor de una variable:

```
int main (void)
{
   int numero;
   numero = 10;
   /* Pasamos por parametro el valor de "numero",
   que es 10. El parametro "A" tomara el valor 10.
   */ tabla_multiplicar (numero);
}
```

o el valor de una constante:

```
#define CONSTANTE 6
int main (void)
{
   /* Pasamos por parametro el valor de CONSTANTE, que
   es 6.
   El parametro "A" tomara el valor 6. */
   tabla_multiplicar (CONSTANTE);
}
```

o el resultado de una expresión:

```
int main (void)
{
   /* Pasamos por parametro el valor resultante de
   (20+1)*2, que es 42.
   El parametro "A" tomara el valor 42. */
   tabla_multiplicar ( (20+1)*2 );
}
```

o el valor de una variable cuyo valor le fue asignado por teclado:

```
int main (void)
{
   int numero;
```

```
    /* Pedimos un numero al usuario. */
    printf ("\nEscriba un numero: ");
    scanf ("%d", &numero) ;

    /* Pasamos por parametro el valor de "numero", que es
    escrito por el usuario.
    El parametro "A" tomara el valor de "numero". */
    tabla_multiplicar (numero);
}
```

Veamos un programa completo que muestra la tabla de multiplicar de un número escrito por el usuario.

```
#include <stdio.h>

/* Declaramos la función tabla_multiplicar */
void tabla_multiplicar (int A);


int main (void)

{
    int numero;

    /* Pedimos un numero al usuario. */
    printf ("\nEscriba un numero: "); scanf
    ("%d", &numero);

    /* Pasamos por parametro el valor de "numero",
    que es escrito por el usuario. */
    tabla_multiplicar (numero);

    /* Hacemos una pausa hasta que el usuario pulse Intro */
    fflush(stdin);
    printf("\n\nPulse Intro para finalizar...");
    getchar () ;
}


/* Definición de la función tabla_multiplicar */ void tabla_multiplicar (int A)

{
    int n;              /* Almacena los números que multiplican a A. */
    int resultado;      /* Almacena el resultado de la multiplicación. */

    /* Mostramos en pantalla la tabla de multiplicar
    del numero A. */
    for (n=1; n<=10; n++)
    {
       /* Obtenemos el resultado de la multiplicación. */
       resultado = A * n;

       /* Mostramos el resultado. */
       printf ("\n%d x %d = %d", A, n, resultado);
    }
}
```

Las funciones pueden tener más de un parámetro.
Si diseñamos una función llamada sumar, mediante el sistema de representación gráfica de las cajas, que muestre en pantalla el resultado de sumar dos números introducidos por ranuras obtendríamos el resultado de la figura 13.6.

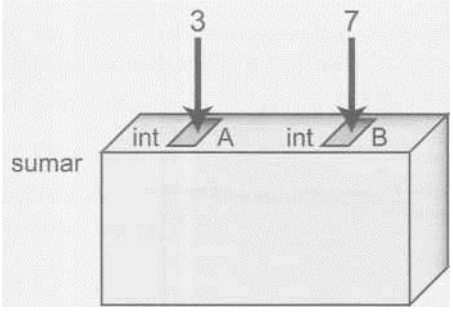


Figura 13.6. Representación de la función sumar.

En esta figura se ha nombrado a cada ranura como "A" y "B", y se han introducido los valores 3 y 7, respectivamente. Dentro de la caja se debería realizar una operación del tipo A+B.
En C/C++, los parámetros se separan por comas. La función sumar tendría el siguiente aspecto.

```
void sumar (int A, int B)
{
   int resultado;

   /* Obtenemos el resultado de sumar A y B. */
   resultado = A + B;

   /* Mostramos el resultado. */ printf ("%d",
   resultado);
}
```

Cuando llamemos a esta función y le pasemos valores por parámetro, éstos los escribimos separados por comas. Un programa que utilice esta función tendría este otro aspecto:

```
#include <stdio.h>

/* Declaramos la función sumar */
void sumar (int A, int B);

/* Programa principal. */
int main (void)
{
   int numero1, numero2;
```

```
    /* Pedimos un numero al usuario y lo
    almacenamos en "numero1". */
    printf ("\nEscriba un numero: ");
    scanf ("%d", {¡numero1) ;
    /* Pedimos otro numero al usuario y
    lo almacenamos en "numero2". */
    printf ("\nEscriba otro numero: ");
    scanf ("%d", &numero2);
    /* Pasamos por parametro el valor de
    "numero1" y "numero2" indicados por
    el usuario. */
    sumar (numero1, numero2);
    /* Hacemos una pausa hasta que el
    usuario pulse Intro */
    fflush(stdin);
    printf("\n\nPulse Intro para
    finalizar...");
    getchar () ;
}

/* Definimos la función sumar. */
void sumar (int A, int B)
{
    int resultado; /* Almacena el resultado de A + B. */
    /* Obtenemos el resultado de sumar A y B. */
    resultado = A + B;

    /* Mostramos el resultado. */
    printf ("El resultado de la suma es: %d", resultado);
}
```

También es posible que hagamos funciones que no tengan ningún parámetro. En estos casos, lo más elegante es escribir la palabra reservada void dentro de los paréntesis, lugar destinado a la declaración de parámetros, como en el siguiente ejemplo.

```
void saludo (void)
{
    printf ("Hola a todos");
}
```

Esto no es obligatorio, pero de esta manera indicamos claramente que la función no tiene ningún parámetro. Es como poner en la caja un cartel de "No tiene ranuras". En la llamada a la función no hay que escribir nada entre los paréntesis, sólo se escribe void en la declaración y en la definición:

```
#include <stdio.h>

/* Declaramos la función saludo. */

void saludo (void);

/* Programa principal */

int main (void)
```

```
{
   /* Llamamos a la función saludo. */
   saludo () ;

   /* Hacemos una pausa hasta que el usuario pulse Intro */
   fflush(stdin);
   printf("\n\nPulse Intro para finalizar...");
   getchar () ;
}

/* Función saludo */
void saludo (void)
{
   printf ("Hola a todos");
}
```

Las funciones retornan datos

En algunas ocasiones nos interesa que el resultado de una función no se muestre en la pantalla, sino obtenerlo de alguna manera para que podamos utilizarlo como queramos. La forma de obtener este resultado es que la función nos lo devuelva o retorne.

Siguiendo como ejemplo la función que suma dos números pasados por parámetro, ésta nos puede devolver o retornar el resultado de la suma a través de otra ranura.

En esta ranura sólo hay que indicar el tipo de dato del valor que se retorna y no hay que darle ningún nombre, porque sólo hay una ranura de este tipo para devolver datos. Como el resultado de la suma de los dos números es otro número de tipo int, éste será el tipo de dato que indiquemos junto a la ranura.

Veamos en la figura 13.7 cómo se representa la función sumar cuando devuelve el resultado.

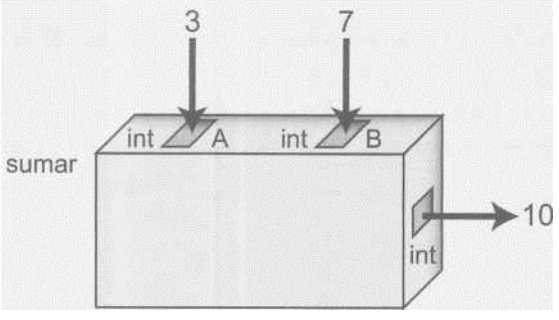


Figura 13.7. Representación de función que retorna un valor.

En C/ C++ para que una función retorne un valor hay que escribir dentro de ésta la instrucción:

```
return expresión;
```

donde `expresión` es la expresión que contiene el valor a retornar. También hay que anteponer al nombre de la función el tipo de dato del valor que se retorna.

Cuando se llame a la función, dicha llamada tomará entonces el valor que se retorne.

Para que la función `sumar` devuelva el resultado de la suma, en vez de mostrarlo en la pantalla, la definición sería:

```
int sumar (int A, int B)
{
   int resultado; /* Almacena el resultado de A + B. */
   /* Obtenemos el resultado de sumar A y B. */
   resultado = A + B;
   /* Devolvemos o retornamos el resultado. */
   return resultado;
}
```

En esta función queremos devolver el valor de la variable resultado, que almacena la suma de las variables (o parámetros) **A y B.** Para ello primero obtenemos la suma:

```
resultado = A + B;
```

y, a continuación, retornamos el valor de resultado:

```
return resultado;
```

Como la variable resultado es de tipo int, delante del nombre de la función escribimos el tipo de dato de esta variable:

```
int sumar (int A, int B)
```

Para explicar cómo se utiliza la llamada a esta función veremos un programa completo.

```
#include <stdio.h>
/* Declaramos la función sumar */ int sumar (int A, int B);
/* Programa principal. */
int main (void)
{
   int numero1, numero2;
   int resultado suma;
```

```
    /* Pedimos un numero al usuario y lo almacenamos
    en "numero1". */
    printf ("\nEscriba un numero: ");
    scanf ("%d", &numero1);
    /* Pedimos otro numero al usuario y lo
    almacenamos en "numero2". */
    printf ("\nEscriba otro numero: ");
    scanf ("%d", &numero2);

    /* Pasamos por parametro el valor de "numero1"
    y "numero2" indicados por el usuario, y el valor
    retornado lo almacenamos en la variable
    "resultado_suma". */
    resultado_suma = sumar (numero1, numero2);
    printf ("\nEl resultado de la suma es: %d",
    resultado_suma);

    /* Hacemos una pausa hasta que el usuario pulse Intro */
    fflush(stdin);
    printf("\n\nPulse Intro para finalizar...");
    getchar () ;
}


/* Definimos la función sumar. */
int sumar (int A, int B)
{
    int resultado; /* Almacena el resultado de A + B. */

    /* Obtenemos el resultado de sumar A y B. */
    resultado = A + B;

    /* Devolvemos o retornamos el resultado. */
    return resultado;
}
```

Ahora la declaración de la función incluye el tipo de dato que devuelve:

```
int sumar (int A, int B);
```

En la línea de código:

```
resultado_suma = sumar (numero1, numero2);
```

primero se ejecuta la expresión situada a la derecha de la asignación. Así que se llama a la función `sumar` y se pasan por parámetro los valores de las variables `numero1` (supongamos que es `3`) y `numero2` (supongamos que es 7). Entonces, ya en la función, el parámetro `A` toma el valor `3` y el parámetro `B` toma el valor 7. La variable `resultado` toma el valor 10 mediante la asignación:

```
resultado = A + B;
```

El valor de `resultado`, que es 10, se retorna:

```
return resultado;
```

Como la función `sumar` se ha terminado, la ejecución vuelve al programa principal. Toda la llamada a la función toma el valor que retornó: 10. Luego la asignación es equivalente a:

```
resultado_suma = 10;
```

y la variable `resultado_suma` toma el valor 10.
También es posible que hagamos funciones que no retornen ningún valor. En estos casos, lo más elegante es escribir la palabra reservada `void` delante del nombre de la función, lugar destinado para el tipo de dato que se retorna, como en el siguiente ejemplo.

```
void saludo (void)
{
    printf ("Hola a todos");
}
```

Esto no es obligatorio, pero de esta manera indicamos claramente que la función no devuelve ningún valor. Como esta función tampoco tiene parámetros también escribimos `void` entre los paréntesis.

Nota:

En la función main estamos indicando siempre que se retorna un valor de tipo int porque esto permite aumentar la portabilidad del programa. Si la fundó?: main retorna el valor cero es que el programa finalizó correctamente, pero si devuelve un valor distinto de cero es que se produjo algún error. Sin embargo no escribimos ninguna instrucción return, ya que por defecto se retorna cero, pero esto sólo ocurre en la función main. En el resto defunciones es necesario devolver un valor deforma explícita con la instrucción return.

Paso de parámetros por referencia

Supongamos que queremos hacer una función muy simple que modifique el valor de una variable numérica pasada por parámetro incrementándola en una unidad. La siguiente función `incrementar` no cumple tal fin, aunque lo pueda parecer.

```
#include <stdio.h> void

incrementar (int n);

int main (void)
```

```
{
    int numero =1;    /* "numero" vale 1. */

    /* Intentamos incrementar el valor de "numero". */
    incrementar (numero);

    /* Mostramos el valor de "numero",
    que sigue siendo 1. */
    printf ("%d", numero);

    /* Hacemos una pausa hasta que el usuario pulse Intro */
    fflush(stdin);
    printf("\n\nPulse Intro para finalizar...");
    getchar();
}


void incrementar (int n)
{
    n = n + 1;  /* Incrementamos el valor de "n". */
}
```

Las variables declaradas dentro de cada función, inclusive los parámetros, sólo existen dentro de éstas y son independientes. Así, una variable llamada "X" dentro de una función es distinta a otra variable llamada "X" dentro de otra función.
Cuando se llama a la función incrementar y se le pasa por parámetro el valor de la variable numero, que es 1:

```
incrementar (numero);
```

lo que ocurre es que el valor de numero se le asigna al parámetro n, tomando la variable n el valor 1. Posteriormente, se incrementa el valor de n, tomando el valor 2:

```
n = n + 1;
```

De vuelta en el programa principal, se ejecuta la línea:

```
printf ("%d", numero);
```

que muestra el valor de la variable numero, que es 1, puesto que en ningún momento hemos hablado de que la variable numero se haya incrementado. Sólo hemos incrementado la variable n.
Esto ocurre porque estamos pasando a la función el valor de la variable, y no la misma variable para modificar su contenido. Recuerde que las variables declaradas dentro de cada función, inclusive los parámetros, sólo existen dentro de éstas y son independientes. De forma que una función no puede saber lo que hay en otras a no ser que se pase por parámetro. En la figura 13.8 se representa la independencia de las funciones main e incrementar y de las variables, así como el paso por valor.

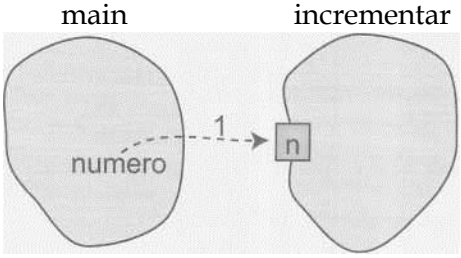


Figura 13.8. Independencia de funciones y paso por valor.

Tal como puede observarse la función main, mediante la llamada a la función incrementar, le envía el valor 1 al parámetro n. Y ese valor 1 es lo único que conoce la función incrementar de la función main.
Para que la función incrementar pueda modificar el valor de la variable numero la solución es el paso por referencia. Para ello utilizaremos los punteros. Ya vimos que con los punteros podemos acceder a una variable a partir de su dirección de memoria y ésta será la estrategia a seguir. Lo que vamos a hacer es enviar a la función incrementar la dirección de memoria de la variable numero, que obtendremos mediante el operador & (ampersand). Ya en la función, y con la dirección de numero, accederemos al contenido de esta variable mediante el operador *, luego el parámetro n será un puntero. En la figura 13.9 vemos un esquema junto con la representación del estado de la memoria tras la llamada a la función.

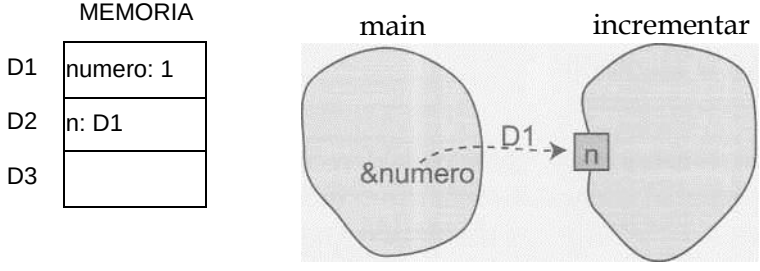


Figura 13.9. Independencia de funciones y paso por referencia.

El programa quedaría así:

```
#include <stdio.h>

void incrementar (int *n);

int main (void)
{
   int numero = 1;    /* "numero" vale 1. */
```

```
    /* Incrementar el valor de "numero". */
    incrementar (&numero);

    /* Mostramos el valor de "numero". */
    printf ("%d", numero);

    /* Hacemos una pausa hasta que el usuario pulse Intro */
    fflush(stdin);
    printf("\n\nPulse Intro para finalizar...");
    getchar () ;
}

void incrementar (int *n)
{
    *n = *n + 1;       /* Incrementamos el valor de "*n". */
}
```

Inicialmente la variable numero vale 1 y se almacena en la dirección de memoria DI. Al llamar a la función incrementar le pasamos la dirección de memoria de la variable numero mediante el operador &:

```
incrementar (&numero);
```

Ya en la función incrementar, el parámetro n recibe la dirección de memoria de la variable numero, que es de tipo int. Así que n se declara como un puntero a int y se recibe el valor DI.

```
void incrementar (int *n)
{
    *n = *n + 1;
{
```

Hasta el momento el estado de la memoria es el de la anterior figura.
Llegamos a la asignación:

```
*n = *n + 1;
```

Primero se resuelve la expresión de la derecha de la asignación. Como n vale DI, *n es igual al contenido de la dirección DI, que es la variable numero, cuyo valor es 1. Así que esta asignación es equivalente a:

```
*n = numero + 1;
```

que es igual a:

```
*n = 2;
```

El resultado de la derecha de la asignación se almacena en el lugar indicado a la izquierda, que es la variable numero, puesto que *n hace referencia a esta variable, como acabamos de explicar. Luego la variable numero almacena ahora el valor 2.
El estado de la memoria es el de la figura 13.10.

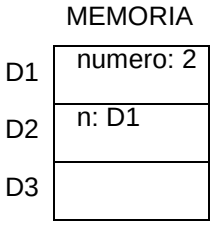


Figura 13.10. Estado de la memoria tras la asignación.

La función `incrementar` finaliza `y` la ejecución continúa en el programa principal (o main) con la siguiente línea de código después de la llamada a la función.

```
printf ("%d", numero);
```

Esta vez, al mostrar en pantalla el valor de la variable `numero` aparecerá un 2. ¡Hemos logrado que la función `incrementar` modifique el valor de la variable `numero`!

Así que si queremos que una función modifique el valor de una variable de otra función tenemos que utilizar los punteros como acabamos de explicar. Si estamos trabajando en C++, en vez de utilizar este sistema podemos usar las referencias (&), como vimos en el capítulo de punteros `y` referencias. En este caso el programa anterior quedaría así:

```
#include <stdio.h>

void incrementar (int &n);

int main (void)
{
    int numero =1;        /* "numero" vale 1. */

    /* Incrementar el valor de "numero". */
    incrementar (numero);

    /* Mostramos el valor de "numero". */
    printf ("%d", numero);

    /* Hacemos una pausa hasta que el usuario pulse Intro */
    fflush(stdin);
    printf("\n\nPulse Intro para finalizar...");
    getchar ();
}


void incrementar (int &n)
{
    n = n + 1;         /* Incrementamos el valor de "numero". */
}
```

Recordemos que a la vez que se crean las referencias hay que asignarles la variable a la que se hace dicha referencia. Como en el momento en que se llama a la función incrementar se asigna numero a n, entonces n pasa a ser una referencia a la variable numero. Dentro de la función incrementar, todo lo que hagamos con la referencia n realmente se lo estamos haciendo a la variable numero. Así que todo es mucho más sencillo, no tenemos que pensar en direcciones de memoria, pero recuerde que las referencias son exclusivas de C++.
En la figura 13.11 vemos el estado de la memoria justo antes del incremento n=n+l y después del incremento.

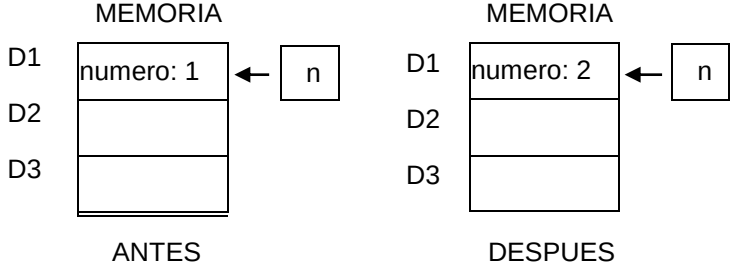


Figura 13.11. Estado de la memoria antes y después del incremento.

El paso de parámetros por referencia también puede ser usado para devolver valores, como hacíamos al usar return, pero con la ventaja de que podemos devolver tantos como queramos. Recordemos que usando return sólo se podía devolver un valor.

Ámbito de las variables locales y globales

Ya hemos comentado que las variables, otros elementos y los parámetros que se declaran o definen en una función sólo existen dentro de ésta, de manera que las variables de una función son independientes y desconocidas por las otras funciones. Esto nos llevaba a utilizar el paso por parámetro para comunicar unas funciones con otras, como se mostraba en las figuras 13.9 y 13.10. A estas variables se les denomina locales.
En algunas ocasiones tendremos variables u otros elementos que necesitamos que puedan se utilizados en todo o gran parte del programa y en cualquier momento. Entonces creamos variables, u otros elementos, globales.

Para ello se declara o define la variable u otro elemento fuera de toda función y al principio del programa, como en el siguiente programa.

```
#include <stdio.h>

int global; /* Variable global. */

void prueba (void);   /* Declaración de la función. */
/* Programa principal. */
void main (void)
{
    int local1; /* Variable local. */

    /* Asignamos el valor 1 a la variable global. */
    global = 1;

    /* Llamamos a la función prueba. */
    prueba();

    /* Mostramos el valor de la variable global. */
    printf ("\n%d", global);

    /* Hacemos una pausa hasta que el usuario pulse Intro */
    fflush(stdin);
    printf("\n\nPulse Intro para finalizar...");
    getchar();
}

/* Definición de la función prueba */ void
prueba (void)
{
    int local2; /* Variable local. */

    /* Mostramos el valor de la variable global. */
    printf ("\n%d", global);

    /* Incrementamos el valor de la variable global. */
    global = global +1;
}
```

Como vemos, la variable `global` es utilizada por todas las funciones sin necesidad de paso por parámetro. Pero si intentamos usar la variable `local2` en la función `main`, o la variable `local1` en la función `prueba`, se producirá un error cuando compilemos. En la figura 13.12 se muestra una representación con estas variables y el ámbito de las variables locales y globales.

Por eso, las definiciones de estructuras, las constantes y las declaraciones de funciones se declaran fuera de toda función y al principio del programa, para hacerlas globales y poder usarlas en cualquier momento. Ya tiene una buena cantidad de ejemplos de elementos globales. Aparte de estos elementos, declarar variables globales para evitar el paso por parámetro o porque

sí está muy mal visto y es propio de los programadores noveles. Utilice variables globales sólo cuando tenga una razón de peso, el paso por parámetro da mayor claridad al programa.

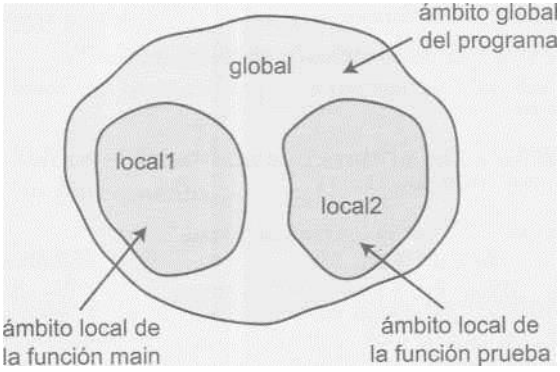


Figura 13.12. Ámbito de las variables locales y globales.

Las variables globales comienzan su existencia en el punto en el que se declaran y dejan de existir cuando el programa finaliza. Las variables locales en una función comienzan su existencia en el punto en el que se declaran y dejan de existir cuando finaliza la función.
Si la variable local pertenece a un bloque de código, ésta deja de existir cuando el bloque finaliza.

Recursividad

La recursividad es un mecanismo para resolver determinados problemas y que consiste en una función que se llama a sí misma. Suele ser bastante costosa de entender, puesto que se puede llegar a necesitar una gran capacidad de abstracción en algunos casos.
Una función recursiva debe contemplar dos casos:

1. El caso en el que se llama a sí misma.
2. El caso en el que no se llama a sí misma.

El primero hace que sea una función recursiva y el segundo impide que se esté llamando a sí misma siempre, evitando una recursividad infinita. Seguidamente analizaremos paso a paso la función recursiva factorial, una de las funciones recursivas más utilizadas en el aprendizaje de este tema.

Función recursiva factorial

El factorial de un número n es igual a:

```
n • (n-1)• (n-2)• (n-3)•...· 1
```

Por ejemplo, el factorial de 6 es 720:

```
6 - 5 - 4 - 3 - 2 - 1  = 720
```

Y el factorial de 3 es 6:

```
3 • 2 • 1 = 6
```

A continuación, veremos un ejemplo sobre cómo se desarrolla la ejecución de la función recursiva que resuelve el factorial de un número pasado por parámetro. Es importante leerlo despacio, con calma y, si es necesario, con ayuda de papel y lápiz para hacer algunos esquemas o apuntes, pues resolver mentalmente una función recursiva puede resultar complicado al principio.

```
int factorial (int numero)
{
   int resultado; if (numero==1)
   {
      /* Caso no recursivo*/ resultado = 1;
   }
   else
   {
      /* Caso recursivo */
      resultado = numero * factorial(numero - 1);
   }
   return resultado;
}
```

Partiremos de la idea de que queremos obtener el factorial de 3. La función main podría ser como la siguiente.

```
int main (void)
{
   int x;
   x = factorial(3);
   printf ("El factorial de 3 es; %d", x) ;

   /* Hacemos una pausa hasta que el usuario pulse Intro
   */ fflush(stdin);
   printf("\n\nPulse Intro para finalizar...");
   getchar();
}
```

La variable x será la encargada de almacenar el resultado de calcular el factorial de 3 mediante la función recursiva factorial.

Una vez obtenido el resultado y almacenado en la variable x:

```
x = factorial (3);
```

se mostrará en pantalla su valor:

```
printf ("El factorial de 3 es: %d", x);
```

Ahora que sabemos que una función recursiva se usa como cualquier otra, veamos cómo funciona.

La primera llamada a la función factorial

En la línea de código de la función main:

```
x = factorial(3);
```

es donde se llama por primera vez a la función recursiva factorial, pasándole por parámetro el valor 3. Si observamos la función factorial, el valor pasado por parámetro, en este caso 3, se almacena en la variable numero:

```
int factorial (int numero)
```

De este modo la variable numero toma el valor 3. Tras la declaración de la variable resultado, encontramos una sentencia condicional if:

```
if (numero==1)
{
    /* Caso no recursivo*/ resultado = 1;
}
else
{
    /* Caso recursivo */
    resultado = numero * factorial(numero - 1);
}
```

Como la variable numero almacena el valor 3, la condición (numero==1) es falsa, por lo que se ejecuta el bloque else:

```
resultado = numero * factorial(numero - 1);
```

Esta línea de código es entonces equivalente a:

```
resultado = 3 * factorial(3 - 1);
```

esto es:

```
resultado = 3 * factorial(2);
```

De esta manera, la variable resultado tomará el valor de 3 multiplicado por el resultado de la llamada afactorial (2). Por último, la función factorial devolverá el valor de la variable resultado:

```
return resultado;
```

La segunda llamada a la función factorial

Pero ¿cuál es el resultado de la llamada a `factorial (2)` ? y ¿qué ocurre con la llamada `afactorial(2)?La` respuesta es que se vuelve a realizar otra ejecución de la función `factorial,` pero esta vez el valor pasado por parámetro es 2, así que en esta nueva llamada la variable `numero` toma el valor 2. Tras la declaración de la variable `resultado,` encontramos la sentencia condicional `if:`

```
if (numero==l)
{
   /* Caso no recursivo*/ resultado = 1;
}
else
{
   /* Caso recursivo */
   resultado = numero * factorial(numero - 1);
}
```

Como la variable `numero` almacena el valor 2, la condición `(numero==l)` es falsa, por lo que se ejecuta el bloque `else:`

```
resultado = numero * factorial(numero - 1);
```

Esta línea de código es entonces equivalente a:

```
resultado = 2 * factorial(2 - 1);
```

esto es:

```
resultado = 2 * factorial(1);
```

De esta manera, la variable `resultado` tomará el valor de 2 multiplicado por el resultado de la llamada a `factorial` (1). Por último, la función `factorial` devolverá el valor de la variable resultado:

```
return resultado;
```

La tercera llamada a la función factorial

Y ahora ¿cuál es el resultado de la llamada `afactorial(l)?y` ¿qué ocurre con la llamada `afactorial(l)?La` respuesta es que se vuelve a realizar otra ejecución de la función `factorial,` pero esta vez el valor pasado por parámetro es 1, así que en esta nueva llamada la variable `numero` toma el valor 1. Tras la declaración de la variable `resultado,` encontramos la sentencia condicional `if`:

```
if (numero==l)
{
```

```
    /* Caso no recursivo*/ resultado = 1;
}
else
{
    /* Caso recursivo */
    resultado = numero * factorial(numero - 1);
}
```

Como la variable `numero` almacena el valor 1, la condición `(numero==1)` es verdadera, por lo que se ejecuta el bloque:

```
resultado = 1;
```

La variable `resultado` toma el valor 1 `y` la función `factorial` devuelve el valor de la variable `resultado`, que es 1:

```
return resultado;
```

Resolviendo las llamadas a la función factorial

Así que la llamada a la función `factorial (1)`, devuelve el valor 1, como acabamos de comprobar. Si recordamos, en la segunda llamada a la función `factorial, factorial (2)`, llegamos a la línea de código:

```
resultado = 2 * factorial(1);
```

pero no sabíamos cuál era el valor devuelto por `factorial (1)`, ahora sabemos que es 1:

```
resultado = 2 * 1 ;
```

La variable `resultado` toma el valor `2 y` la segunda llamada a la función devuelve el valor de la variable `resultado`, que es `2`:

```
return resultado;
```

Por tanto, la llamada a la función `factorial (2)`, devuelve el valor 2, como acabamos de comprobar. Haciendo nuevamente memoria, en la primera llamada a la función `factorial, factorial (3)`, en la función `main`, llegamos a a línea de código:

```
resultado = 3 * factorial(2);
```

Pero no sabíamos cuál era el valor devuelto por `factorial(2)`, ahora sabemos que es 2:

```
resultado = 3 * 2 ;
```

La variable `resultado` toma el valor `6 y` la primera llamada a la función devuelve el valor de la variable `resultado`, que es `6`:

```
return resultado;
```

Por tanto, la llamada a la función `factorial (3),` devuelve el valor `6,` como acabamos de comprobar. De vuelta a la función `main,` teníamos la línea:

```
x = factorial(3);
```

Donde la variable `x` toma el valor devuelto por `factorial (3),` que ahora sabemos que es 6. Finalmente, gracias a la siguiente línea de código:

```
printf ("El factorial de 3 es: %d", x);
```

se muestra en pantalla un mensaje informando sobre cuál es el factorial de 3:

```
El factorial de 3 es 6
```

Recursividad infinita

Es muy importante que toda función recursiva tenga un caso en el que no se llame a sí misma, o las llamadas serían infinitas y el programa no tendría fin. En el ejemplo de la función factorial, si no existiese el caso:

```
if (numero==1)
{
    /* Caso no recursivo*/
    resultado = 1;
}
```

en el cual no se vuelve a hacer la llamada a la función `factorial,` estaríamos ante una función con recursividad infinita, puesto que sólo se ejecutaría la línea:

```
resultado = numero * factorial(numero - 1);
```

y las llamadas a:

```
factorial(numero - 1);
```

se repetirían una y otra vez. Siguiendo el ejemplo de factorial de 3, si no existiese el caso que impide la recursividad infinita, las llamadas a factorial se harían de la forma:

```
factorial   (3);
factorial   (2);
factorial   (1);
factorial (- 1);
factorial (- 2);
factorial (- 3);
factorial (- 4);
factorial (- 5);
```

y así sucesivamente. No tendría fin. Gráficamente, es como si en el esquema de la figura 13.13 no existiesen las flechas de retorno.

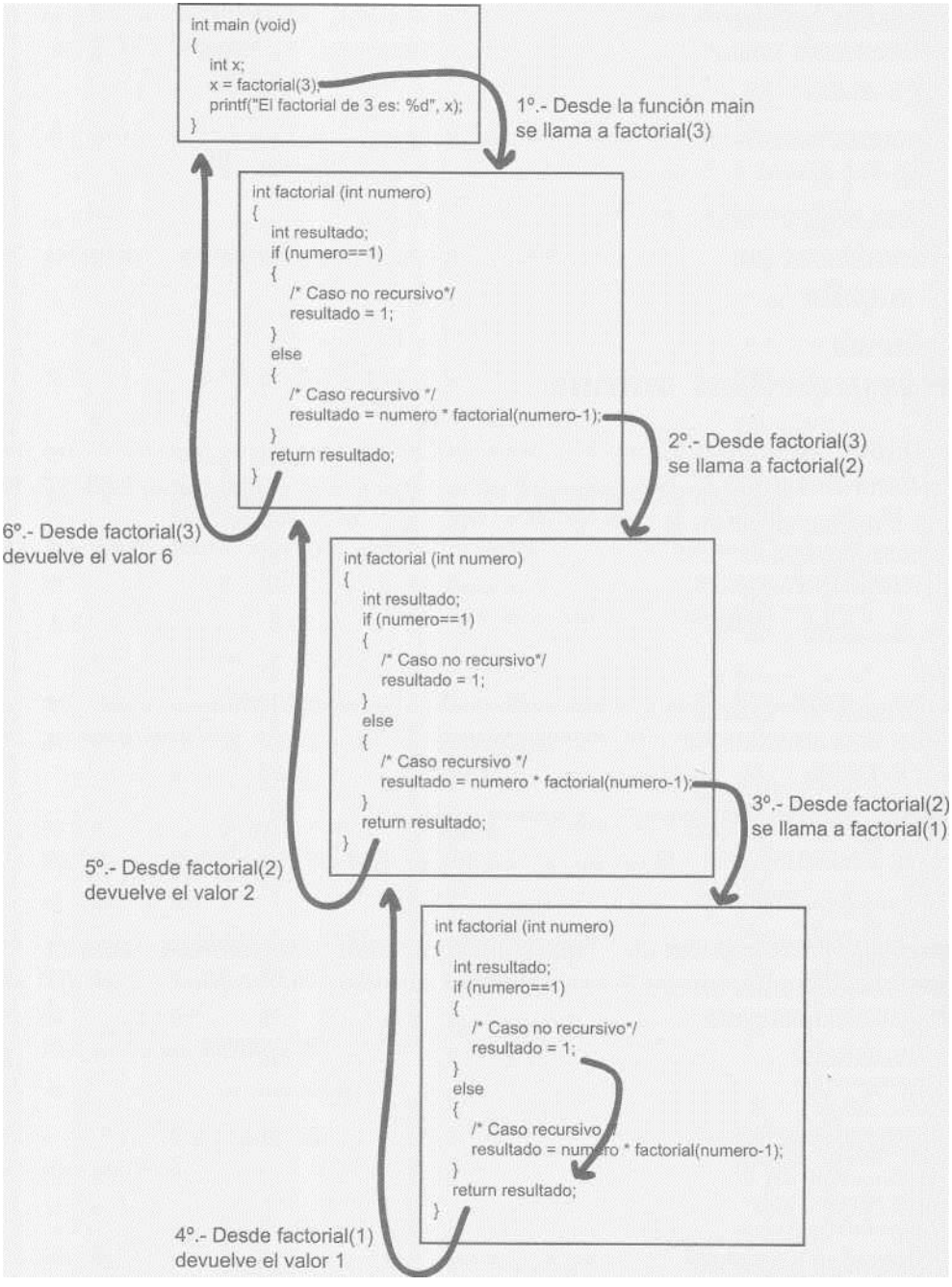


Figura 13.13. Representación gráfica de las llamadas recursivas de la función factorial.

Ejercicios resueltos

Intente realizar los siguientes ejercicios. Encontrará la solución al final de este libro.

1. Escriba un programa que utilice una función que pida una clave al usuario. Hasta que no acierte la clave (que por ejemplo es 123) la función no debe finalizar. En el programa principal, debe mostrarse un mensaje de saludo antes de la llamada a la función y otro mensaje de despedida después de la llamada a la función.
2. Realice un programa que utilice una función a la que se le pasa un número por parámetro. La función debe mostrar en la pantalla un mensaje indicando si el número es par o impar.
3. Escriba una función que modifique el valor de una variable numérica pasada por parámetro. La función debe pedir un número al usuario y asignárselo a la variable.

Resumen

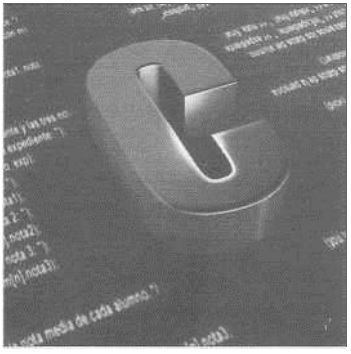
En este capítulo hemos aprendido muchos conceptos nuevos, pero la idea principal es que podemos dividir los programas en funciones. Imagine un programa de 100.000 líneas de código y todas ellas en la función `main`. Si el programa tiene un error, ¿por dónde empezamos a chequearlo? Con las funciones es más fácil encontrar los errores y corregirlos. Podemos afrontar el problema por partes.

También hemos aprendido que a las funciones les podemos pasar valores por parámetro, que pueden retornar valores, que pueden modificar los valores de las variables y que pueden llamarse a sí mismas (recursivas).

Conocer ambos ámbitos nos permite entender muchos aspectos, como por qué la definición de una estructura se escribe fuera de toda función y al principio de un programa.

En conclusión, la programación modular nos ofrece una gran cantidad de ventajas, entre ellas conseguir programas más claros y estructurados.

Lo más importante y el principal objetivo de este capítulo es aprender a crear y utilizar las funciones correctamente, por eso hemos usado como ejemplos programas muy sencillos.



Capítulo 14

Ficheros

En este capítulo aprenderá:

- Qué son los ficheros.
- Cómo crear Tin fichero.
- Cómo guardar datos en un fichero.
- Cómo recuperar el contenido de un fichero.

Introducción

En todos los programas que hemos hecho hasta el momento, cuando su ejecución finalizaba los valores de las variables, *arrays* y estructuras se perdían. Es decir, los datos que introdujo el usuario en el programa ya no están. Para evitar que cada vez que ejecutemos el programa tengamos que volver a escribir de nuevo todos los datos, éstos los guardamos en ficheros.
Un fichero es un conjunto de datos que se almacenan en un soporte como el disco duro, un disquete, un CD-ROM o un DVD, y se les da un nombre para poder identificarlos. Los datos de un fichero no se pierden ni al finalizar el programa ni al apagar el ordenador.
Podemos guardar en ellos números, caracteres, cadenas, estructuras y *arrays,* entre otros.
En los ficheros podemos realizar distintas operaciones, como: crearlos, abrirlos, ver su contenido, añadirles datos y cerrarlos.
En C, todas las funciones que utilicemos para trabajar con ficheros pertenecen a la biblioteca `stdio`.

Apertura de un fichero

Ya hemos dicho que los ficheros tienen un nombre y que se almacenan en soportes como, por ejemplo, el disco duro. El disco duro es muy grande, así que para que el ordenador sepa dónde está cada fichero dicho soporte tiene una tabla en la que se relaciona el nombre de un fichero y la dirección (o lugar) en la que se encuentra dentro del disco. Cuando abramos un fichero obtendremos esa dirección y será con ella con la trabajemos para realizar las distintas operaciones.

fopen

Para crear un fichero o abrir uno ya existente utilizamos la función `fopen`, que devuelve la dirección del fichero, un puntero a **FILE**. Su sintaxis general es:

```
fopen (nombre, modo)
```

Donde `nombre` es una cadena con el nombre del fichero y, opcionalmente, su ruta. Por ejemplo: "c:/textos/prueba.txt" o "datos.bin". Si no especificamos la ruta del fichero se supondrá que se encuentra en el directorio actual de trabajo.

El parámetro `modo` es una cadena que contiene una serie de caracteres que configuran cómo queremos que se abra o se cree el fichero. En la tabla 14.1, se indican las distintas posibilidades seguidas de su significado.

Tabla 14.1. Modos de apertura de ficheros.

Modo	Descripción
"r"	(*Read*, Leer) Si el fichero existe se abre en modo sólo lectura y, por defecto, en modo texto. Si el fichero no existe la función devuelve el valor NULL. Se utiliza para leer los datos de un fichero en modo texto.
"w"	(*Write*, Escribir) Si el fichero no existe lo crea para escribir en él y lo deja abierto, por defecto, en modo texto, y si ya existe lo sobrescribe. Todas las operaciones de escritura con el modo "w" se realizan al final del fichero. Se utiliza para crear ficheros y escribir en él en modo texto.
"a"	(*Append*, Añadir) Si el fichero no existe lo crea para escribir en él y lo deja abierto, por defecto, en modo texto, y si el fichero ya existe permite añadir más datos al final de éste, respetando sus datos anteriores. Se utiliza para añadir datos en modo texto a un fichero.
"r+"	Igual que "r", pero también permite escribir en cualquier punto del fichero. Se utiliza para leer y modificar los datos de un fichero en modo texto.
"w+"	Igual que "w", pero también permite leer del fichero. Se utiliza para crear un fichero en modo texto y poder realizar operaciones de lectura.
"a+"	Igual que "a", pero también permite leer del fichero. Se utiliza para añadir datos a un fichero en modo texto y poder realizar operaciones de lectura.
"rt"	Igual que "r". La 't' es de Texto.
"wt"	Igual que "w". La 't' es de Texto.
"at"	Igual que "a". La 't' es de Texto.
"rt+"	Igual que "r+".
"wt+"	Igual que "w+".
"at+"	Igual que "a+".
"rb"	Igual que "r", pero en modo binario, en vez de en modo texto. La 'b' es de Binario.
"wb"	Igual que "w", pero en modo binarlo, en vez de en modo texto. La 'b' es de Binarlo.

Modo	Descripción
"ab":	Igual que "a", pero en modo binario, en vez de en modo texto. La 'b' es de Binario.
"rb+":	Igual que "r+", pero en modo binario, en vez de en modo texto.
"wb+":	Igual que "w+", pero en modo binario, en vez de en modotexto.
"ab+":	Igual que "a+", pero en modo binario, en vez de en modotexto.

Entonces, si queremos abrir un fichero de texto llamado "leeme.txt" para ver su contenido escribimos:

```
fopen ("leeme.txt", "rt");
```

Como ya hemos dicho, esta función nos devuelve la dirección del fichero, que es la que utilizaremos para realizar las distintas operaciones en éste. Así que tendremos que almacenarla en una variable puntero a FILE a la que llamaremos, por ejemplo, fichero:

```
#include <stdio.h>

int main (void)

{

    /* Variable para almacenar la dirección del fichero. */

    FILE *fichero;

    /* Abrimos el fichero "leme.txt" */

    fichero = fopen ("leeme.txt", "rt");

    /* Hacemos una pausa hasta que el usuario pulse Intro */
    fflush(stdin);

    printf("\n\nPulse Intro para finalizar...");

    getchar();

}
```

Tras utilizar fopen en el código anterior, cualquier operación que hubiésemos realizado con la variable fichero la estaríamos realizando con el fichero "leeme.txt".

Si queremos crear un fichero binario llamado "libros.dat" escribimos:

```
#include <stdio.h>

int main (void)

{

    /* Variable para almacenar la dirección del fichero. */

    FILE *fichero;

    /* Abrimos el fichero "libros.dat" */
    fichero = fopen ("libros.dat", "wb");
```

```
   /* Hacemos una pausa hasta que el usuario pulse Intro */
   fflush(stdin);
   printf("\n\nPulse Intro para finalizar...");
   getchar();
}
```

Cierre de un fichero

fclose

Debemos cerrar los ficheros en cuanto dejemos de usarlos; de lo contrario, podemos alterar sus datos por error, lo cual no nos hará ninguna gracia. Para ello utilizamos la función `fclose`, siendo su sintaxis general:

```
fclose (puntero)
```

Donde `puntero` es el identificador de la variable puntero a `FILE`, que almacena la dirección del fichero abierto. Cerramos el fichero de texto anterior:

```
#include <stdio.h>

int main (void)
{
   /* Variable para almacenar la dirección del fichero. */
   FILE *fichero;

   /* Abrimos el fichero "leeme.txt" */

   fichero = fopen ("leeme.txt", "rt");

   /* Cerramos el fichero "leeme.txt" */

   fclose (fichero);

   /* Hacemos una pausa hasta que el usuario pulse Intro
   */ fflush(stdin);
   printf("\n\nPulse Intro para finalizar...");
   getchar();
}
```

Si el fichero "leeme.txt" no existe, como ya se ha explicado, la función `fopen` devuelve `NULL`, luego la variable `fichero` tomaría este valor. No tiene mucho sentido cerrar el fichero "leeme.txt" si no se ha podido abrir antes. Así que vamos a añadir un control que detecte si el fichero se ha abierto (`fichero` es distinto de `NULL`) o no (`fichero` es igual a `NULL`). Sólo en el caso de que se abra vamos a cerrarlo, lo cual es lo lógico, ¿verdad?

```
#include <stdio.h>

int main (void)
{
```

```
   /* Variable para almacenar la dirección del fichero. */
   FILE *fichero;
   /* Abrimos el fichero "leeme.txt" */ fichero = fopen
   ("leeme.txt", "rt");
   /* Controlamos si el fichero "leeme.txt" se ha abierto.
   */ if (fichero == NOLL)
   {
      printf ("Error: El fichero leeme.txt no se ha abierto.");
   }
   else
   {
      /* Cerramos el fichero "leeme.txt" */
      fclose (fichero);
   }
   /* Hacemos una pausa hasta que el usuario pulse Intro */
   fflush(stdin);
   printf("\n\nPulse Intro para finalizar...");
   getchar();
}
```

Para que el programa quede más completo hemos añadido un mensaje de error para el caso de que el fichero no exista. De esta manera mantenemos siempre informado al usuario, un aspecto muy importante.

Escritura de un fichero

fputc

En un fichero de texto podemos escribir caracteres, pero de uno en uno, con la función `fputc`. La sintaxis general es:

```
fputc (carácter, puntero);
```

Donde `carácter` es el carácter que queremos escribir en el fichero indicado por la variable `puntero`. Ejemplo:

```
fputc ('A', fichero);
```

fputs

También podemos escribir en un fichero de texto una cadena utilizando la función `fputs`. La sintaxis general es:

```
fputs (cadena, puntero);
```

Donde `cadena` es la cadena de caracteres que queremos escribir en el fichero indicado por la variable `puntero`.
Ejemplo:

```
fputs ("Esto es una cadena", fichero);
```

fwrite

Sin embargo, la función `fwrite` nos permite trabajar tanto con ficheros de texto como binarios y escribir todo tipo de datos. Como esta función nos permite hacer de todo nos detendremos más en ella.
La sintaxis general es:

```
fwrite (direccion_dato, tamaño, veces, puntero);
```

Donde `direccion_dato` es la dirección de la variable o elemento que queremos escribir en el fichero indicado por la variable `puntero`. En `tamaño` debemos indicar el tamaño en bytes de la variable o elemento que queremos escribir en el fichero, y en `veces` indicamos cuántos elementos de tamaño `tamaño` vamos a escribir.
¡Cuidado!, `veces` no es el números de veces que queremos que se escriba repetidamente el dato indicado en `direccion_dato`.
Para obtener el tamaño de una variable podemos usar el operador `sizeof` y para obtener el tamaño de una cadena la función `strlen`. Pero veamos unos ejemplos.

```
#include <stdio.h>

int main (void)
{
    FILE *fichero;
    char carácter = 'A';

    fichero = fopen ("prueba.txt", "wt");

    if (fichero == NULL)
       printf ("Error: No se ha podido crear el fichero prueba.txt.");
    else {
       fwrite (&caracter, sizeof(carácter), 1, fichero);
       fclose (fichero);
    }

    /* Hacemos una pausa hasta que el usuario pulse Intro */
    fflush(stdin);
    printf("\n\nPulse Intro para finalizar...");
    getchar();
}
```

En la línea:

```
fwrite (&caracter, sizeof(carácter), 1, fichero);
```

hemos escrito en el fichero "prueba.txt" el contenido de la variable `carácter`, que es 'A'. Para ello indicamos en el primer parámetro la dirección de memoria de la variable `carácter`, cuyo contenido queremos escribir en el fichero. En el segundo parámetro escribimos el tamaño en bytes de la variable `carácter`. Utilizamos el operador `sizeof` para que lo calcule. El tercer parámetro indica cuántos elementos de tamaño `sizeof (carácter)` hay en la variable `carácter`: 1. Este parámetro generalmente será 1, aunque al escribir *arrays* en los ficheros puede variar. El cuarto parámetro es la variable puntero al fichero abierto.
Si en el mismo fichero queremos escribir una cadena almacenada en la variable declarada como:

```
char nombre[30];
```

escribiremos:

```
fwrite (&nombre, strien(nombre), 1, fichero);
```

Para almacenar en un fichero binario el contenido de una variable estructura llamada `ficha` escribimos:

```
fwrite (&ficha, sizeof(ficha), 1, fichero);
```

En el siguiente programa se escribe en el fichero "prueba.txt" una frase pedida al usuario.

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

int main (void)

{

    FILE *fichero;
    char frase[100];

    fichero = fopen ("prueba.txt", "wt");

    if (fichero == NÜLL)

       printf ("Error: No se ha podido crear el fichero prueba.txt.");

    else

    {

       printf ("Escriba una frase: ");

       gets (frase);

       fwrite (Sfrase, strlen(frase), 1, fichero); fclose (fichero);

    }


    /* Hacemos una pausa hasta que el usuario pulse Intro */
    fflush(stdin);
```

```
    printf("\n\nPulse Intro para finalizar...");
    getchar ();
    }
```

Supongamos que queremos crear un fichero binario, llamado "datos.dat", en el que almacenemos tres estructuras con la siguiente definición:

```
Struct t_ficha {
    char
    nombre[30];
    int edad;
};
```

El programa sería el siguiente:

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

struct t_ficha {
    char
    nombre[30];
    int edad;
} ;

int main (void)

{

    FILE *fichero;
    struct t_ficha persona;

    int n;

    fichero = fopen ("datos.dat", "wb");

    if (fichero == NULL)
       printf ("Error: No se ha podido abrir el fichero datos.dat.");
    else {
       /* Solicitamos al usuario un nombre y una edad, que almacenamos en
       una estructura. Posteriormente, se graba la estructura en el fichero.
       Esto se ejecuta 3 veces. */
       for (n=1; n<=3; n++)
       {
          printf ("\n\nNombre: "); gets (persona.nombre);
          printf ("\nEdad: "); scanf ("%d", &persona.edad) ;
          fwrite(¿¿persona, sizeof(persona), 1, fichero);
       }
       fclose (fichero);
    }

    /* Hacemos una pausa hasta que el usuario pulse Intro */
    fflush(stdin);
    printf("\n\nPulse Intro para finalizar...");
    getchar ();
}
```

Lectura de un fichero

Claro está que si ya sabemos escribir datos en los ficheros ahora queremos abrirlos y leerlos para ver sus contenidos.
El mecanismo de lectura consiste en lo siguiente. Cuando se abre un fichero el puntero se encuentra al principio de éste. Si hacemos una operación de lectura se lee el elemento sobre el que está el puntero y, a continuación, el puntero avanza al siguiente elemento, pero hasta que no se haga otra operación de lectura no se lee.

fgetc

Para leer un carácter de un fichero de texto podemos utilizar la función fgetc, que devuelve el carácter leído. La sintaxis general es:

```
fgetc (puntero);
```

Donde **puntero** es el identificador de la variable puntero a **FILE** correspondiente a un fichero de texto abierto. Como la función devuelve el carácter leído, un ejemplo de su uso es:

```
#include <stdio.h>

int main (void)
{
   FILE *fichero;
   char carácter;

   fichero = fopen ("prueba.txt", "rt");

   if (fichero == NULL)
      printf ("Error: No se ha podido abrir el fichero prueba.txt.");
   else
   {
      carácter = fgetc (fichero);
      printf ("El carácter leido es: %c", carácter); fclose (fichero);
   }

   /* Hacemos una pausa hasta que el usuario pulse Intro */
   fflush(stdin);
   printf("\n\nPulse Intro para finalizar...");
   getchar();
}
```

Tras leer del fichero un carácter éste se almacena en la variable **carácter**, que posteriormente se muestra en pantalla.

feof

Si quisiéramos mostrar en pantalla todos los caracteres que hay en el fichero "prueba.txt", o en cualquier otro, tendríamos que detectar el final del fichero para no salimos de éste. Para ello utilizamos la función feof. La sintaxis general es:

```
feof (puntero)
```

Donde puntero es el identificador de la variable puntero al fichero. Esta función retorna el valor 0 si no se ha llegado al final del fichero y retorna un valor distinto de 0 si se llega al final.
El siguiente programa muestra todo el contenido de "prueba.txt".

```
#include <stdio.h>

int main (void)

   FILE *fichero;
   char carácter;

   fichero = fopen ("prueba.txt", "rt");

   if (fichero == NULL)
      printf ("Error: No se ha podido abrir el fichero
   prueba.txt.");
      else {
      carácter = fgetc (fichero);
      while (feof(fichero)==0)
      {
         printf ("%c", carácter);
         carácter = fgetc (fichero);
      }
      fclose (fichero);
   }
    /* Hacemos una pausa hasta que el usuario pulse Intro */
    fflush(stdin);
    printf("\n\nPulse Intro para finalizar...");
    getchar ();
}
```

La idea consiste en leer un carácter y, si no hemos llegado al final del fichero, lo mostramos en pantalla. Así, hasta llegar al final del fichero.

fgets

También podemos leer una cadena de caracteres mediante la función fgets. La sintaxis general es:

```
fgets (cadena, tamaño, puntero)
```

Donde `cadena` es la variable que va a almacenar la cadena que se lea del fichero, `tamaño` es la cantidad máxima de caracteres que vamos a leer del fichero y almacenar en `cadena` y `puntero` es el identificador de la variable puntero al fichero. Decimos que `tamaño` es la cantidad máxima de caracteres que vamos a leer del fichero porque la lectura se puede detener antes si se encuentra un salto de línea.
Esta función devuelve la cadena de caracteres leída o el valor **NULL** si hubiese un error o se llegase al final del fichero.
El ejemplo anterior con fgets quedaría así:

```
#include <stdio.h>

int main (void)
{
    FILE *fichero;
    char
    cadena[256];
    char *resultado;

    fichero = fopen ("prueba.txt", "rt")

    ; if (fichero == NULL)
       printf ("Error: No se ha podido abrir el fichero prueba.txt.");
    else {
       resultado = fgets (cadena, 256, fichero); while (resultado!=NULL)
       {
          printf ("%s", cadena);
          resultado = fgets (cadena, 256, fichero);
       }
       fclose (fichero);
    }

    /* Hacemos una pausa hasta que el usuario pulse Intro */
    fflush(stdin);
    printf("\n\nPulse Intro para finalizar...");
    getchar();
}
```

fread

Para leer datos de distintos tipos podemos utilizar la función `fread`, que ofrece la misma potencia que `fwrite`, pero para leer. Además, la sintaxis es idéntica:

```
fread (direccion_dato, tamaño, veces, puntero);
```

Donde `direccion_dato` es la dirección de la variable o elemento donde queremos almacenar los datos leídos del fichero indicado por la variable

puntero. En tamaño debemos indicar el tamaño en bytes que queremos leer el fichero, y en veces indicamos cuántos elementos de tamaño tamaño vamos a leer.
Se nos puede plantear la duda de qué valores indicamos en tamaño y veces, pero, como lo normal es que el fichero que vayamos a leer también lo hayamos creado nosotros, conoceremos estos valores, que los habremos utilizado previamente con fwrite.
Si anteriormente hemos creado un fichero binario, llamado "datos.dat", en el que almacenamos estructuras con la siguiente definición:

```
struct t_ficha { char nombre[30]; int edad;
};
```

el programa que muestra los campos de todas las estructuras almacenadas en dicho fichero sería el siguiente:

```
#include <stdio.h>

struct t_ficha { char nombre[30]; int edad;
} ;

int main (void)

{
   FILE *fichero;
   struct t_ficha persona;

   fichero = fopen ("datos.dat", "rb");

   if (fichero == NULL)
      printf ("Error: No se ha podido abrir el fichero datos.dat.");
   else {
      fread(Spersona, sizeof(persona), 1 , fichero);
      while (feof (fichero)==0)
      {
         printf ("\n\nNombre: %s", persona.nombre);
         printf ("\nEdad: %d", persona.edad);
         fread(&persona, sizeof(persona), 1 , fichero);
      }
      fclose (fichero);
   }

   /* Hacemos una pausa hasta que el usuario pulse Intro */
   fflush(stdin);
   printf("\n\nPulse Intro para finalizar...");
   getchar();
}
```

También podemos crear un programa que solicite al usuario una edad, busque en el fichero todas las personas con dicha edad y muestre en pantalla sus nombres:

```
#include <stdio.h>

struct t_ficha {
   char
   nombre[30]; int
   edad;
} ;


int main (void)
{
   FILE *fichero;
   struct t_ficha persona;
   int edadbuscada;

   /* existeedad: almacena el valor 0 si no encuentra ninguna
   persona con la edad buscada, y el valor 1 en caso contrario.
   */ int existeedad = 0;

   /* Pedimos al usuario una edad a buscar */

   printf ("\nEscriba una edad: ", edadbuscada);

   scanf("%d", &edadbuscada);

   fichero = fopen ("datos.dat", "rb");

   if (fichero == NULL)
      printf ("\nError: No se ha podido abrir el fichero datos.dat.");
   else
   {
      fread(Spersona, sizeof(persona), 1 , fichero);
      while (feof(fichero)==0)
      {
         if (persona.edad == edadbuscada)
         {
            printf ("\n\nNombre: %s", persona.nombre);
            existeedad=1;
         }

         fread(Spersona, sizeof (persona), 1, fichero);
      }

      fclose (fichero);

      if (existeedad==0)
         printf("\n\nNo existe ninguna persona con esa edad");
   }

   /* Hacemos una pausa hasta que el usuario pulse Intro */
   fflush(stdin);
   printf ("\n\nPulse Intro para finalizar...");
   getchar();
}
```

Acceso directo a un registro: fseek

En algunas ocasiones necesitaremos acceder a un determinado registro, por ejemplo el situado en tercer lugar, para leerlo o para modificarlo. La función fseek, del estándar ANSI C, nos permite situar directamente el puntero del fichero en cualquier registro o, dicho de otra forma, saltar desde el principio del fichero a una posición determinada de éste. Su sintaxis general es:

```
fseek (puntero, tamano_salto, desde);
```

Donde puntero es la varióle puntero del fichero con el que estemos trabajando, tamano_salto es el tamaño en bytes del salto realizado desde la posición desde. El parámetro desde puede tomar tres valores posibles definidos como constantes (ver tabla 14.2).

Tabla 14.2. Posibles valores del parámetro "desde" de la función fseek.

Constante	Valor	Descripción
SEEK SET	0	Salta desde el principio del fichero.
SEEK CUR	1	Salta desde la posición actual.
SEEK END	2	Salta desde el final del fichero.

Quedará más claro con un ejemplo. Supongamos que tenemos un fichero llamado "usuarios.dat" y que a priori sabemos que contiene cinco estructuras de tipo:

```
struct t_ficha {
  char nombre[30];
  char clave[30];
} ;
```

Ahora, imaginemos que queremos hacer un pequeño programa que muestre en pantalla los datos del registro situado en tercer lugar, podría ser similar al mostrado a continuación.

```
#include <stdio.h>

struct t_ficha {
   char nombre [30];
   char clave[30];
} ;

int main (void)
{
```

```
    FILE *fichero;
    struct t_ficha usuario;

    fichero = fopen ("usuarios.dat", "rb");

    if (fichero == NULL)
       printf ("Error: No se ha podido abrir el fichero usuarios.dat.");
    else {
       fseek(fichero, 2*sizeof (struct t_ficha), SEEK_SET);
       fread(Susuario, sizeof(struct t_ficha), 1, fichero);
       printf ("\n\nNombre: %s", usuario.nombre);
       printf ("\nClave: %s", usuario.clave);
       fread (¡¿usuario, sizeof (usuario) , 1, fichero);
       fclose (fichero);
    }

    /* Hacemos una pausa hasta que el usuario pulse Intro */
    fflush(stdin);
    printf("\n\nPulse Intro para finalizar...");
    getchar () ;
}
```

La principal novedad es el uso de la función fseek:

```
fseek(fichero, 2*sizeof(struct t_ficha), SEEK_SET);
```

El primer parámetro indica el puntero del archivo con el que vamos a trabajar, en este caso es fichero, que está asociado al archivo "usuarios.dat", previamente abierto mediante la función fopen.
El tercer parámetro, SEEK_SET, especifica que vamos a mover o bien a desplazar el puntero del archivo una determinada distancia desde el principio del archivo.
¿Y cuánto vamos a desplazar el puntero al archivo? Pues si queremos leer el tercer registro, debemos situar el puntero del archivo al principio de éste, por lo que tendremos que saltar los dos primeros registros. ¿Y cómo sabemos cuál es la "distancia a saltar" o tamaño en bytes de estos dos registros? Mediante la función sizeof (struct t_ficha) obtenemos el tamaño de la estructura, por lo que si lo multiplicamos por 2, obtendremos el tamaño de dos registros, que es lo que especificamos en el segundo parámetro. Resumiendo, la función fseek anterior nos permite saltar dos (2) registros (sizeof (struct t_ficha)) desde el principio (SEEK_SET) del archivo (fichero).
Una vez ejecutada la llamada a la función fseek, el puntero fichero se encuentra al principio del tercer registro, por lo que la llamada a la función fread lo lee directamente.
Si por ejemplo quisiéramos leer un registro sí otro no hasta terminar el fichero, utilizaríamos el valor constante SEEK CUR tras cada fread:

```
#include <stdio.h>

struct t_ficha { char nombre[30]; char clave[30];
} ;

int main (void)
{
    FILE *fichero; struct
    t_ficha usuario;
    fichero = fopen ("usuarios.dat", "rb") ;

    if (fichero == NULL)
       printf ("Error: No se ha podido abrir el fichero
    usuarios.dat."); else {
       /* Leemos el primer registro */
       fread(&usuario, sizeof(struct t_ficha), 1, fichero);
       /* Si no hemos llegado al final del archivo continuamos
       */ while (! feof(fichero))
       {
          printf ("\n\nNombre: %s", usuario.nombre);
          printf ("\nClave: %s", usuario.clave);
          /* Saltamos 1 registro desde la posición en la que nos
          encontramos */
          fseek(fichero, l*sizeof(struct t_ficha), SEEK_CUR);
          /* Leemos el siguiente registro */
          fread(Susuario, sizeof(usuario), 1, fichero);
       }
       fclose (fichero);
    }

    /* Hacemos una pausa hasta que el usuario pulse Intro */
    fflush(stdin);
    printf("\n\nPulse Intro para finalizar...");
    getchar () ;
}
```

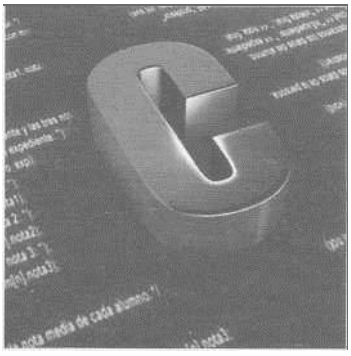
Ejercicios resueltos

Intente realizar los siguientes ejercicios. Encontrará la solución al final de este libro.

1. Realice un programa que pida al usuario el nombre de un fichero de texto *y,* a continuación, muestre todo su contenido en la pantalla. Controle la posibilidad de que el fichero no exista.
2. Escriba un programa que permita almacenar en un fichero de texto tantas frases como el usuario desee. Puede comprobar si se han almacenado correctamente con el programa del ejercicio anterior.

Resumen

Almacenar los datos en los ficheros tiene muchas ventajas, puesto que dichos datos no se borran cuando finaliza el programa o se apaga el ordenador. Además, pueden ser llevados de un ordenador a otro. Pensemos en los documentos que se escriben en el ordenador, las partidas guardadas de los juegos, e, incluso, los programas que hacemos. Todos ellos se pueden guardar o almacenar en un soporte como el disco duro. Pero además podemos recuperarlos, modificarlos y borrarlos. Si no pudiésemos almacenar los datos en los ficheros habría que reescribirlos todos cada vez que ejecutásemos un programa.



Capítulo 15

Estructuras dinámicas

En este capítulo aprenderá:

- Qué son las estructuras dinámicas.
- Cómo reservar memoria y liberarla
- A implementar listas, pilas y colas.
- A trabajar con listas, pilas y colas.

Introducción

El *array es* una estructura de datos estática, es decir, su tamaño es fijo (invariable), porque en el momento en el que se compila su declaración se reserva la memoria para éste. Supongamos que en un programa hemos declarado un *array* para almacenar hasta cinco números:

```
int serie[5];
```

Si, por alguna circunstancia, sólo almacenamos un número en la primera celda del *array*, entonces las otras cuatro celdas estarán desocupadas, desperdiciando memoria. Si por cualquier otra circunstancia queremos almacenar siete números en el *array*, entonces no hay celdas suficientes.
Este problema se soluciona con las estructuras dinámicas, que permiten modificar su tamaño mediante la reserva y liberación de memoria durante la ejecución del programa, adaptándose a las necesidades y optimizando el uso de la memoria. Si sólo quiero almacenar un dato reservamos memoria sólo para éste; si quiero almacenar siete datos reservamos memoria para esos siete. En la figura 15.1 se muestra la diferencia entre las estructuras estáticas y dinámicas siguiendo el ejemplo de almacenar uno o siete números.
Antes de comenzar con las estructuras dinámicas (de las cuales estudiaremos las listas, las pilas y las colas) vamos a aprender a reservar y liberar memoria en C y C++.
Cuando reservamos una cantidad de memoria para almacenar en ella algún dato, ésta es para nuestro uso y ningún otro programa puede utilizarla. Cuando ya no nos haga falta dicha memoria reservada debemos liberarla, es decir, dejarla disponible para otros programas, o para el nuestro. Es muy importante liberarla, porque la memoria no es infinita.

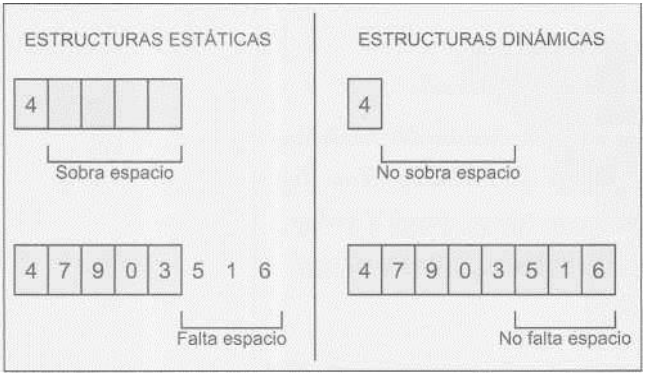


Figura 15.1. Diferencia entre estructuras estáticas y dinámicas.

Reserva y liberación de memoria en C

Para reservar memoria C nos proporciona la función `mal loe,` de la biblioteca `stdlib,` a la que debemos pasar por parámetro el número de bytes que queremos reservar.
La función nos devolverá la dirección de memoria en la que empieza la zona reservada o **NULL** si no hay memoria suficiente. La forma general de utilizar esta función es:

```
puntero = (tipo *) malloc (bytes);
```

En donde `puntero` es el identificador de la variable puntero que almacena la dirección de memoria de la zona reservada y que es del mismo tipo que el dato que vamos a almacenar en ésta, `tipo` es el tipo del dato que vamos a almacenar en la memoria y `bytes` es la cantidad de memoria a reservar expresada en número de bytes.
Una vez que hemos reservado la memoria utilizamos la variable puntero y los operadores de punteros para trabajar con los datos almacenados en dicha memoria.
Para liberar la memoria reservada utilizamos la función `f ree,` de la biblioteca `stdlib,` de la forma:

```
free (puntero);
```

Donde `puntero` es el identificador de la variable puntero que almacena la dirección de memoria de la zona reservada, que habremos usado previamente junto con la función `malloc.`
En el siguiente programa reservamos memoria para una estructura, rellenamos sus campos, los mostramos y liberamos la memoria reservada. Tras el listado pasamos a comentar el programa.

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

struct t_ficha {
   int numero;
   char palabra[20];
};

int main (void)
{
   struct t_ficha *mi_ficha;

   mi_ficha = (struct t_ficha *) malloc (sizeof(struct t_ficha));

   printf ("Escriba un numero: ");
   scant ("%d", &mi_ficha->numero);
```

```
    fflush(stdin);
    printf ("Escriba una palabra: ");
    gets(mi_ficha->palabra);

    printf ("\nNumero: %d", mi_ficha->numero);
    printf ("\nPalabra: %s", mi_ficha->palabra);

    free (mi_ficha) ;

    /* Hacemos una pausa hasta que el usuario pulse Intro */
    fflush(stdin);
    printf("\n\nPulse Intro para finalizar...");
    getchar () ;
}
```

Hemos definido una estructura llamada `t_f i cha` con dos campos: uno numérico y otro una cadena.

```
struct t_ficha {
    int numero;
    char palabra[20];
} ;
```

La primera línea de código dentro de la función main declara un puntero a una estructura `t_ficha.`

```
struct t_ficha *mi_ficha;
```

Esta variable puntero no es una estructura, sólo almacenará la dirección de memoria en la que se encuentre la estructura.
En la siguiente línea:

```
mi_ficha = (struct t_ficha *) malloc (sizeof(struct t_ficha));
```

se reserva memoria para una estructura. La dirección en la que comienza la zona reservada la almacenamos en la variable `mi_f icha.`
Como queríamos reservar memoria para una estructura debíamos pasar a la función `malloc` el tamaño de la estructura, para lo cual hemos utilizado el operador `sizeof.`
Recuerde que si indicamos entre paréntesis una variable o un tipo de dato tras `sizeof` se obtiene el número de bytes que ocupa.
Tras esta línea la variable `mi_ficha` tiene la dirección de memoria de la nueva estructura.
Para acceder a los campos de la estructura utilizamos los caracteres en vez del punto, ya que `mi_f icha` no es una estructura, sino un puntero a una estructura.

```
printf ("Escriba un numero: ");
scanf ("%d", &mi_ficha->numero);
printf ("Escriba una palabra: ");
gets(mi ficha->palabra);
```

Una vez que el usuario ha almacenado algún dato en los campos de la estructura los mostramos para comprobar si todo ha ido bien.

```
printf ("\nNumero: %d", mi_ficha->numero);

printf ("\nPalabra: %s", mi_ficha->palabra);
```

Como ya no queremos hacer nada más con la estructura liberamos la memoria reservada antes de finalizar el programa:

```
free (mi_ficha);
```

Reserva y liberación de memoria en C++

C++ proporciona el operador **new** para reservar memoria. La forma general de utilizarlo es:

```
puntero = new tipo;
```

En donde **puntero** es el identificador de la variable puntero que almacena la dirección de memoria de la zona reservada y que es del mismo tipo que el dato que vamos a almacenar en ésta, **tipo** es el tipo del dato que vamos a almacenar en la memoria.

Una vez que hemos reservado la memoria utilizamos la variable puntero y los operadores de punteros para trabajar con los datos almacenados en dicha memoria.

Para liberar la memoria reservada utilizamos el operador **delete** de la forma siguiente:

```
delete puntero;
```

Donde **puntero** es el identificador de la variable puntero que almacena la dirección de memoria de la zona reservada, que habremos utilizado previamente junto con la función **new**.

El siguiente programa es el mismo que el anterior, pero utilizando los operadores **new** y **delete** en vez de las funciones **malloc** y **free**.

```
#include <iostream.h>

struct t_ficha {
   int numero;
   char palabra[20];
};


int main (void) {

   struct t ficha *mi ficha;
```

```
    mi_ficha = new struct t_ficha;

    cout << "Escriba un numero:
    cin >> mi_ficha->numero;
    cout << "Escriba una palabra:
    cin >> mi_ficha->palabra;

    cout << "XnNumero: " << mi_ficha->numero;
    cout << "\nPalabra: " << mi_ficha->palabra;

    delete mi_ficha;

    /* Hacemos una pausa hasta que el usuario pulse Intro */
    fflush(stdin);
    cout << "\nPulse Intro para finalizar...";
    cin.get();
}
```

Listas

Una lista es una colección de elementos y es algo muy normal en la vida cotidiana: la lista de la compra, la lista de los alumnos de una clase, la lista de productos a comprar al distribuidor.
En las estructuras dinámicas, cuando queremos añadir un elemento se reserva memoria para éste y cuando queremos eliminarlo liberamos su memoria. Como cada elemento de la lista puede encontrarse en una zona de memoria distinta necesitamos algún mecanismo que nos permita "encadenar" todos los elementos como si de un *array* se tratase.
La solución es que cada elemento de la lista, al que se denomina nodo, tenga dos partes: una donde almacenar los datos y otra (una variable puntero a la que llamaremos **siguiente)** donde almacenar la dirección del siguiente nodo. Pero ¿dónde se almacena la dirección del primer nodo? Pues la almacenaremos en una variable puntero (a la que llamaremos **principio),** que será la que apunte al principio de la lista. Cuando la lista esté vacía **principio** almacenará el valor **NULL.**
En la figura 15.2 podemos ver una representación gráfica de una lista vacía y una lista con tres nodos, donde se muestra una supuesta dirección de memoria (D1, D4...) en la que se encuentra cada nodo.
Observe que el último nodo almacena el valor **NULL** en el lugar de la dirección del siguiente nodo, ya que no hay un siguiente nodo.
Existen varios tipos de listas (como doblemente enlazada y circulares), pero trataremos únicamente las simples, que son las que han sido comentadas hasta el momento.

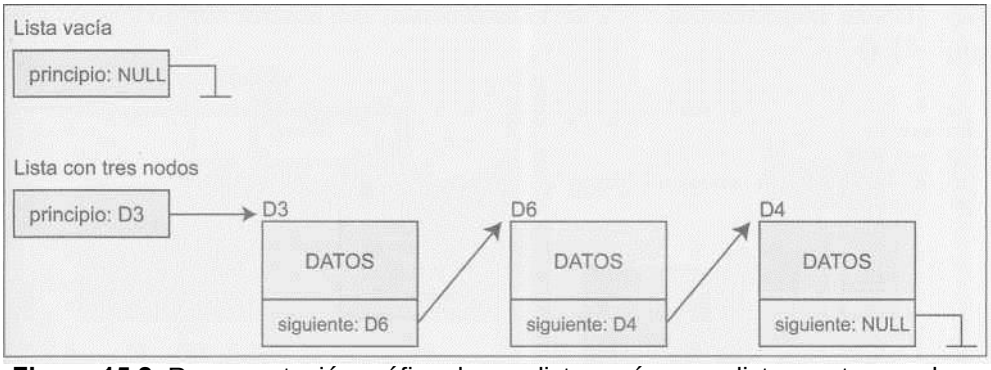


Figura 15.2. Representación gráfica de una lista vacía y una lista con tres nodos.

Operaciones básicas de una lista

Las operaciones básicas, de las cuales vamos a explicar las estrategias, son: insertar un nodo al principio de la lista, insertar un nodo de forma ordenada por los datos que contiene, insertar un nodo al final de la lista y borrar un nodo.

Insertar al principio

Para insertar un nodo al principio de la lista, en primer lugar, vamos a crear el nodo y almacenamos en él los datos. A continuación, si la lista está vacía, porque **principio** almacena el valor **NULL**, hacemos que **principio** apunte al nuevo nodo. Al campo **siguiente** del nuevo nodo le asignamos **NULL** (véase la figura 15.3).

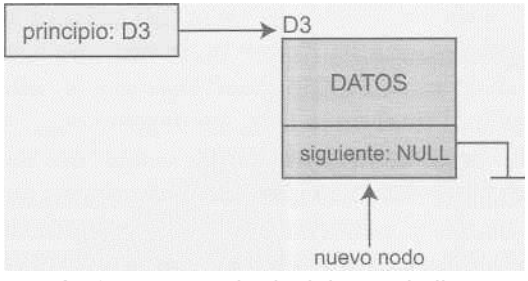


Figura 15.3. Insertar al principio con la lista vacía.

Si la lista no está vacía asignamos al campo **siguiente** del nuevo nodo la dirección del primer nodo de la lista, que obtenemos de la variable

Por último, asignamos a principio la dirección del nuevo nodo (véase la figura 15.4).

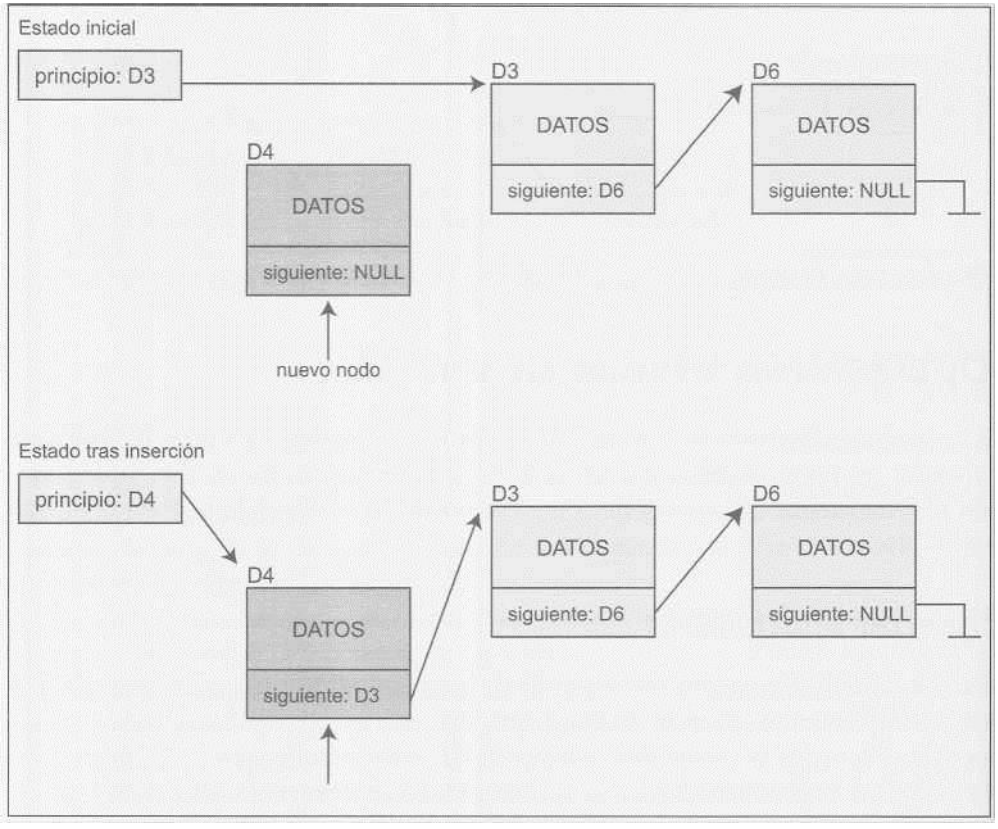


Figura 15.4. Insertar a! principio con la lista no vacía.

Insertar al final

Para insertar un nodo al final de la lista, en primer lugar, creamos el nodo y almacenamos en él los datos.
A continuación, si la lista está vacía hacemos que **principio** apunte al nuevo nodo. Al campo **siguiente** del nuevo nodo le asignamos **NULL** (el resultado es el mismo de la anterior figura 15.3). Si la lista no está vacía asignamos al campo **siguiente** del nuevo nodo el valor **NULL**.
Por último, recorremos la lista hasta llegar al nodo con el campo **siguiente** a **NULL** (que es el último) **y** le asignamos la dirección del nuevo nodo (véase la figura 15.5).

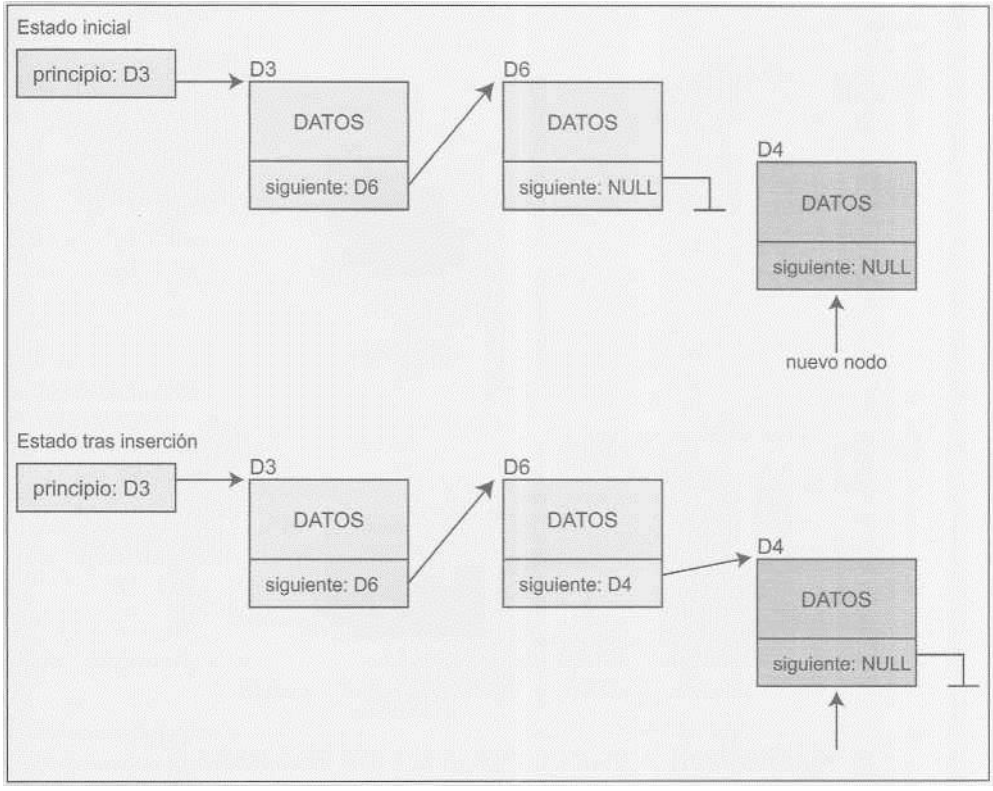


Figura 15.5. Insertar al final de la lista no vacía.

Insertar ordenado

Los nodos también pueden ser insertados en orden en función de los datos que almacena y en función de un criterio de ordenación.
De esta manera, una vez que se ha creado el nodo y han sido almacenados en él los datos, se recorre la lista hasta encontrar la posición donde debería insertarse.
De esta manera, el nodo podría quedar insertado al principio (que ya hemos analizado), al final (que también hemos analizado) o entre otros dos nodos (que analizaremos ahora).
Para insertar un nodo entre otros dos de la lista, en primer lugar asignamos al campo **siguiente** del nuevo nodo la dirección del nodo que quedará a la derecha.
Posteriormente, asignamos al campo **siguiente** del nodo que quedará a la izquierda la dirección del nuevo nodo (véase la figura 15.6).

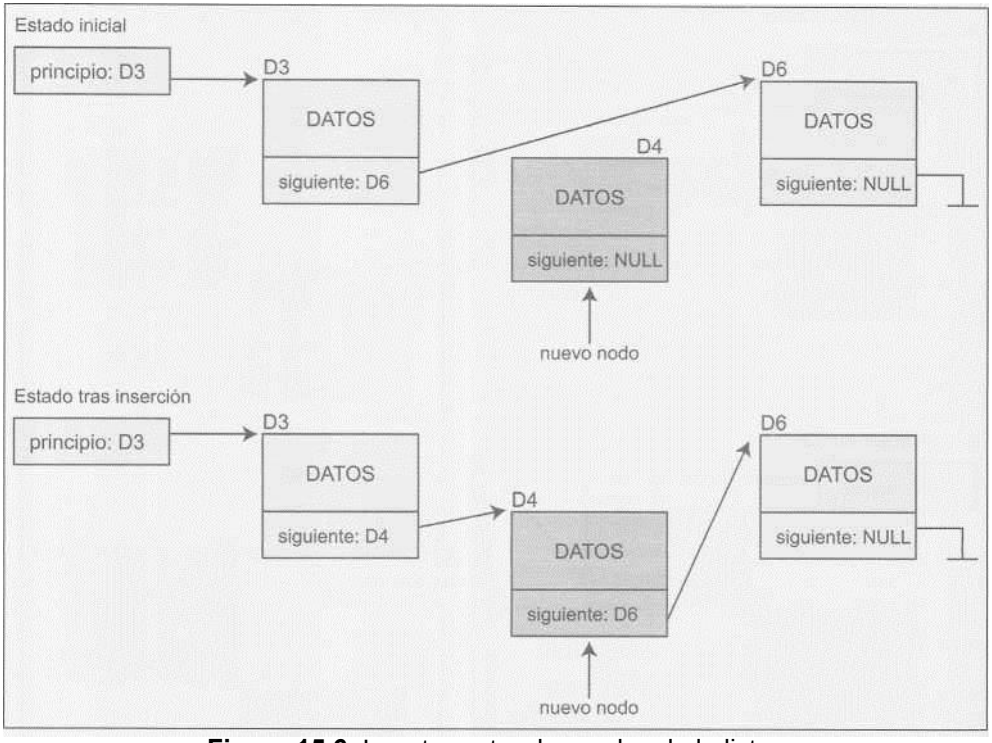


Figura 15.6. Insertar entre dos nodos de la lista.

Borrar

Para borrar un nodo, en primer lugar, recorremos la lista hasta encontrar el nodo a eliminar. Luego, asignamos el campo `siguiente` del nodo a borrar al campo `siguiente` del nodo situado a su izquierda (o a `principio` si es el primer nodo de la lista). Por último, asignamos al campo `siguiente` del nodo a borrar el valor `NULL` y se elimina el nodo. En la figura 15.7 vemos la representación gráfica del borrado de un nodo situado entre otros dos.

Implementación de una lista

Los nodos de la lista tendrán como datos un simple número entero, y la definiremos de la siguiente forma:

```
struct t_nodo {
   int numero;
   t_nodo *siguiente;
};
```

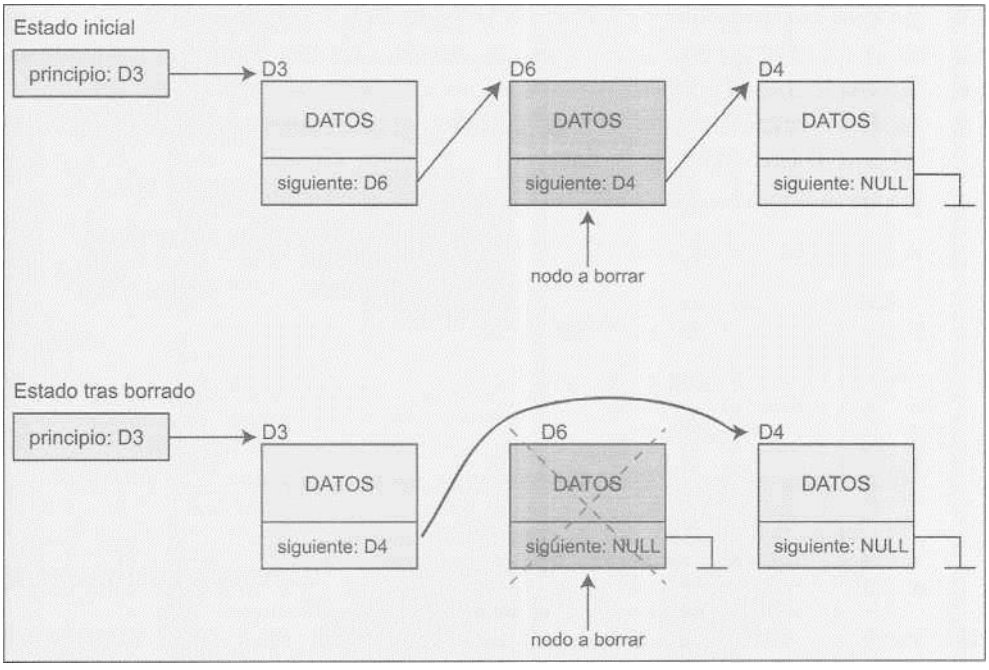


Figura 15.7. Borrado de un nodo de la lista.

Donde **numero** es la variable que almacena los datos del nodo, y **siguiente** es una variable puntero a otra estructura **t_nodo**.
También declararemos la variable **principio**, el puntero al primer nodo de la lista, que inicializaremos a **NULL** para indicar que la lista está vacía:

```
struct t_nodo *principio = NULL;
```

A continuación, implementamos algunas de las operaciones básicas de las listas: insertar al principio, insertar al final, insertar ordenado, borrar, buscar, mostrar y borrar toda la lista. También desarrollaremos una función **main** en la que se utiliza una lista.

Insertar al principio

La función **insertar_al_principio** tiene un parámetro llamado **p** que debe recibir la dirección del principio de la lista, es decir, el valor de la variable **principio**. Este parámetro se ha declarado como una referencia (&) a la variable **principio**, que es un puntero a la estructura `t_nodo (struct t_nodo*)`. El prototipo de esta función es:

```
void insertar al principio (struct t nodo *&p)
```

De esta manera podremos utilizar en la función la variable p como si se tratase de la variable principio, pero recuerde, una vez más, que las referencias son exclusivas de C++.
Dentro de la función se crea el nuevo nodo, se rellenan sus datos por teclado y se inserta al principio de la lista.

```
void insertar_al_principio (struct t_nodo *&p)
{
    struct t_nodo *nuevo_nodo;

    /* Creamos el nuevo nodo */

    nuevo_nodo = new struct t_nodo;

    /* Rellenamos los campos de datos */

    cout << "\nEscriba un numero: ";

    cin >> nuevo_nodo->numero;

    /* Insertamos el nuevo nodo al principio de la lista */

    /* El campo 'siguiente' del nuevo nodo apunta al mismo nodo
    que 'p', es decir, que 'principio'. */
    nuevo_nodo->siguiente = p;
    /* 'p', es decir, 'principio', apunta al nuevo nodo. */
    p = nuevo_nodo;
}
```

Insertar al final

La función insertar_al_f inal pide un número al usuario y lo inserta al final de la lista.

```
void insertar_al_final (struct t_nodo *&p)
{
    struct t_nodo *nuevo_nodo;

    /* 'aux' sirve para recorrer la lista */

    struct t_nodo *aux;

    /* Creamos el nuevo nodo */

    nuevo_nodo = new struct t_nodo;

    /* Rellenamos los campos de datos */
    cout << "\nEscriba un numero: ";
    cin >> nuevo_nodo->numero;
    /* El campo siguiente es igual a NDLL porque va a ser el ultimo
    nodo de la lista. */

    nuevo_nodo->siguiente = NULL;

    /* Insertamos el nuevo nodo al final de la lista */

    /* Si 'p' es distinto de NDLL es que la lista no esta vacia. */
    if (p!=NULL)
```

```
    {
       /* con `aux' recorremos la lista desde 'p', es decir, desde el 'principio',
       en busca del ultimo nodo, que tiene el campo siguiente a NULL. */
       aux = p;
       while (aux->siguiente != NULL)
       {
          aux = aux->siguiente;
       }
       /* El campo siguiente del ultimo nodo apunta al nuevo nodo */
       aux->siguiente = nuevo_nodo;
    }
    else
    {
       /* Si la lista esta vacia 'p', es decir,
       'principio' apunta al nuevo nodo. */
       p = nuevo_nodo;
    }
}
```

Insertar ordenado

La función insertar ordenado pide un número al usuario y lo inserta en la lista de forma ordenada, de menor a mayor.

```
void insertar_ordenado (struct t_nodo *&p)
{
    struct t_nodo *nuevo_nodo;

    /* 'anterior' y 'actual' se utilizan para recorrer la lista y
    encontrar la posición donde insertar el nuevo nodo. */

    struct t_nodo *anterior;

    struct t_nodo *actual;

    /* Creamos el nuevo nodo */ nuevo_nodo = new struct t_nodo;

    /* Rellenamos los campos de datos */

    cout << "\nEscriba un numero:

    cin >> nuevo_nodo->numero;

    /* Insertamos el nuevo nodo al final de la lista */

    /* 'actual' apunta inicialmente a 'p', que es 'principio' */

    actual = p;

    /* Mientras no se llegue al final de la lista (actual!=NULL) y el
    numero del nodo en el que nos encontramos sea menor que el numero
    del nodo nuevo (actual->numero < nuevo_nodo->numero) avanzamos
    por la lista. */
    while ((actual!=NULL) && (actual->numero < nuevo_nodo->numero))
```

```
   {
      /* 'anterior' apunta al mismo nodo que 'actual' */
      anterior = actual;
      /* Hacemos que 'actual' apunte al siguiente nodo */ actual
      = actual->siguiente;
   }

   /* Si 'actual' es igual a 'p' es que hay que insertarlo
   al principio. */ if (actual==p)
   {
      nuevo_nodo->siguiente = p;
      p = nuevo_nodo;
   }
   else
   {
      /* Si no se inserta al principio, se inserta entre
      'anterior' y 'actual', de forma que el siguiente a
      'anterior' es el nuevo nodo y el siguiente al nuevo nodo
      es 'actual'. */ anterior->siguiente = nuevo_nodo;
      nuevo_nodo->siguiente = actual;
   }
}
```

Borrar

La función borrar pide un número al usuario, lo busca en la lista y entonces lo elimina.

```
void borrar (struct t_nodo *&p)
{
   /* 'anterior' y 'actual' se utilizan para recorrer la
   lista y encontrar el nodo a eliminar. */

   struct t_nodo *anterior;
   struct t_nodo *actual;

   int numero;

   /* Pedimos el numero a borrar de la lista */

   cout << "\nEscriba un numero: ";
   cin >> numero;

   /* Borramos el nodo correspondiente al numero indicado
   por el usuario. */

   /* 'actual' apunta inicialmente a 'p', que es 'principio' */
   actual = p;

   /* Mientras no se llegue al final de la lista (actual!=NULL) y el
   numero del nodo en el que nos encontramos sea distinto del numero
   buscado (actual->numero != numero) avanzamos por la lista */
```

```
    while ((actual!=NULL) && (actual->numero != numero))
    {
       /* 'anterior' apunta al mismo nodo que 'actual' */
       anterior = actual;
       /* Hacemos que 'actual' apunte al siguiente nodo */
       actual = actual->siguiente;
    }

    /* Si 'actual' es distinto de NULL es que el numero
    buscado esta en la lista y 'actual' apunta a su nodo */
    if (actual!=NULL)
    {
       /* Si 'actual' es igual a 'p' es que el nodo a borrar
       es el primero. */
       if (actual==p)
       !
          /* Hacemos que 'p' apunte al siguiente nodo
          a borrar y el campo siguiente lo ponemos a
          NULL. Después borramos el nodo liberando su
          memoria. */
          p = actual->siguiente;
          actual->siguiente = NULL;
          delete actual;
       }
       else
       {
          /* Hacemos que en nodo anterior al que queremos
          borrar apunte al siguiente del que queremos
          borrar. El campo siguiente el nodo a borrar lo
          ponemos a NULL. Después borramos el nodo liberando
          su memoria. */ anterior->siguiente =
          actual->siguiente; actual->siguiente = NULL;
          delete actual;
       }
    }
}
```

Buscar

La función buscar pide un número al usuario y lo busca en la lista. A continuación, se muestra un mensaje indicando si está o no en la lista.

```
void buscar (struct t_nodo *&p)
{
   /* 'actual' se utiliza para recorrer la lista */
   struct t_nodo *actual;

   /* 'encontrado' indica si se ha encontrado el
   numero (encontrado vale 1) o no (encontrado vale
   0). Partimos del supuesto de que no se ha
   encontrado. */
```

```
   int encontrado = 0;

   int numero;
   /* Pedimos el numero a buscar en la lista */
   cout << "\nEscriba un numero: ";

   cin >> numero;

   /* Buscamos el nodo correspondiente al numero indicado
   por el usuario. */

   /* 'actual' apunta inicialmente a 'p', que es
   'principio' */ actual = p;

   /* Mientras no se llegue al final de la lista
   (actual!=NULL) y no se encuentre el numero buscado
   (encontrado == 0) seguimos recorriendo las lista. */
   while ( (actual!=NULL) && (encontrado == 0))
   {
    /* Si encontramos el numero damos el valor 1 a
    encontrado. */

    if (actual->numero == numero) encontrado = 1;

    /* Hacemos que 'actual' apunte al siguiente nodo */
    actual = actual->siguiente;
   }

   /* Mostramos un mensaje indicando si el numero buscado
   esta o no en la lista. */
   if (encontrado == 1)
      cout << "\nEl numero esta en la lista";
   else
      cout << "\nEl numero no esta en la lista";
}
```

Mostrar

La función mostrar muestra todos los números de la lista.

```
void mostrar (struct t_nodo *&p)
{
   /* 'actual' se utiliza para recorrer la lista */

   struct t_nodo *actual;

   /* 'actual' apunta inicialmente a 'p', que es
   'principio' */ actual = p;

   /* Si 'p' es igual a NULL es que la lista esta vacia */
   if (p==NULL)
```

```
    {
       cout << "\nLa lista esta vacia.
    {
    else
    {
       /* Si la lista no esta vacia la recorremos
       y mostramos el campo numero. */
       while (actual!=NULL)
       {
          cout << actual->numero << " ";
          actual = actual->siguiente;
       }
    }

}
```

Borrar todo

La función borrar_todo elimina todos los nodos de la lista, liberando toda la memoria reservada.

```
void borrar_todo (struct t_nodo *&p)
{
    /* 'nodo_a_borrar' se apunta al nodo a borrar */
    struct t_nodo *nodo_a_borrar;

    while (p!=NULL)
    {
       /* Hacemos que el 'nodo_a_borrar' apunte al
       primer nodo */ nodo_a_borrar = p;

       /* Hacemos que ' p' apunte al siguiente nodo a
       borrar y el campo siguiente lo ponemos a NULL.
       Después borramos el nodo liberando su memoria.
       */ p = p->siguiente; nodo_a_borrar->siguiente =
       NULL; delete nodo_a_borrar;
    }
}
```

Ejemplo

La siguiente función main contiene un ejemplo del uso de estas funciones.

```
int main (void)
{
    struct t_nodo *principio = NULL;

    cout << "\n -- Insertar al principio ----";
    insertar_al_principio (principio);
```

```
   cout << "\n--- Insertar al final ---
   insertar_al_final (principio);
   cout << "\n--- Insertar ordenado ---
   insertar_ordenado (principio);
   cout << "\n--- Borrar ---
   borrar (principio);
   cout << "\n--- Buscar ---
   buscar (principio);
   cout << "\n--- Mostrar --
   mostrar (principio);

   borrar_todo (principio);

   /* Hacemos una pausa hasta que el usuario pulse Intro */
   fflush(stdin);
   cout << "\nPulse Intro para finalizar...";
   cin.get();
}
```

Pilas

Las pilas son una variante de las listas que tienen como requisito que las inserciones son siempre al principio y que las consultas y borrados son siempre del primer nodo.

Implementación de una pila

Ahora mostraremos la implementación de la función para insertar al principio, para borrar el primer nodo y para mostrar los datos del primer nodo.

Insertar

La función `insertar` pide un número al usuario y lo inserta en la cima de la pila.

```
void insertar (struct t_nodo *&p)
{
   struct t_nodo *nuevo_nodo;

   /* Creamos el nuevo nodo */
   nuevo_nodo = new struct t_nodo;

   /* Rellenamos los campos de datos */
   cout << "\nEscriba un numero: ";
   cin >> nuevo nodo->numero;


  /* Insertamos el nuevo nodo al principio de la lista */
```

```
    /* El campo 'siguiente' del nuevo nodo apunta al mismo nodo
    que 'p', es decir, que 'principio'. */ nuevo_nodo->siguiente
    = p;
    /* 'p', es decir, 'principio', apunta al nuevo nodo. */
    p = nuevo_nodo;
}
```

Borrar

La función borrar elimina el número situado en la cima de la pila.

```
void borrar (struct t_nodo *&p)
{
    /* 'actual' se utilizan para borrar el primer nodo. */
    struct t_nodo *actual;

    /* Borramos el primer nodo. */

    /* 'actual' apunta inicialmente a 'p', que es 'principio' */
    actual = p;

    /* Si 'p' es distinto de NULL es que la pila no esta vacia */
    if (p!=NULL)
    {
       /* Hacemos que 'p' apunte al siguiente nodo
       a borrar y el campo siguiente lo ponemos a
       NULL. Después borramos el nodo liberando su
       memoria. */
       p = actual->siguiente;
       actual->siguiente = NULL;
       delete actual;
    }
}
```

Mostrar

La función mostrar muestra el número situado en la cima de la pila.

```
void mostrar (struct t_nodo *&p)
{
    /* 'actual' se utiliza para mostrar
    los datos del primer nodo */
    struct t_nodo *actual;

    /* 'actual' apunta inicialmente a 'p', que es 'principio' */
    actual = p;


   /* Si 'p' es igual a NULL es que la pila esta vacia */
```

```
    if (p==NULL)
    {
       cout « "\nLa lista esta vacia.";
    }
    else
    {
       /* Si la pila no esta vacia mostramos
       el campo numero. */
       cout « actual->numero « "
    }


}
```

Borrar todo

La función borrar_todo elimina todos los nodos de la pila, liberando toda la memoria reservada. Esta función es la misma que la de las listas.

Colas

Las colas son otra variante de las listas que tienen como requisito que las inserciones son siempre al final y que las consultas y borrados son siempre del primer nodo.
La idea es, por ejemplo, imitar el comportamiento de una cola de personas que esperan para entrar en el cine. El primero que entra en el cine es el primero que llegó y la última persona que llegue al lugar se incorpora al final de la cola.

Implementación de una cola

Las funciones de las colas son iguales que algunas de las listas o pilas.

Insertar

La función insertar es igual que la función insertar_al_f inal de las listas, aunque normalmente las colas suelen tener otro puntero al final de ésta, que hace menos costosa la inserción.

Borrar

La función borrar de las colas es igual que la de las pilas.

Mostrar

La función `mostrar` de las colas es igual que la de las pilas.

Borrar todo

La función `borrar todo` elimina todos los nodos de la cola, liberando toda la memoria reservada. Esta función es la misma que la de las listas.

Resumen

Las estructuras dinámicas son un mecanismo idóneo para almacenar datos en memoria sin malgastarla, como puede ocurrir con los *arrays*. Recuerde que algunas veces hemos aconsejado declarar *arrays* con un tamaño más grande de lo necesario para que no falte espado, pero lo que conseguimos así es desperdiciar memoria.

Puede que se plantee cuál de las tres estructuras dinámicas aprendidas (listas, colas y pilas) debería utilizar para resolver un determinado problema, entonces piense en cuál es el comportamiento de ese problema y a qué mecanismo de cada estructura se asemeja más.

Las funciones para manejar las listas, pilas y colas que hemos visto pueden mejorarse mucho, ya que éstas se han implementado de una forma muy simple para facilitar las explicaciones.

Estas estructuras dinámicas son las más elementales, pero puede que oiga hablar de otras, como los árboles y los grafos. Como puede comprobar siempre hay más.

Capítulo 16

Programación orientada a objetos (POO)

En este capítulo aprenderá:

- Qué es la programación orientada a objetos.
- Los conceptos básicos de la programación orientada a objetos.
- A definir clases, crear objetos y trabajar con ellos.
- Qué son los constructores y destructores.
- Qué es la sobrecarga, la herencia y el polimorfismo.

Introducción: Conceptos básicos

La vida cotidiana está llena de objetos (la televisión, la radio, el ordenador, un reloj, un coche, etc.) con los que podemos interactuar (a través de sus botones, sus palancas, etc.). Estamos totalmente acostumbrados a ellos y realmente percibimos todas las cosas como objetos. Así que ¿por qué no llevar esta filosofía a la programación? Esto es lo que hace C++, que es un lenguaje orientado a objetos. Pero vayamos poco a poco.
Ya que la programación orientada a objetos (POO) pretende llevar a la programación la filosofía de los objetos de la vida cotidiana, para explicar algunos conceptos partiremos de un objeto real: una radio digital como la mostrada en la figura 16.1.

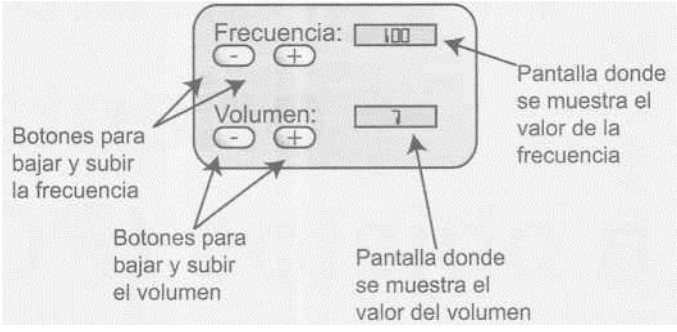


Figura 16.1. Radio digital y sus partes.

Esta radio digital tiene dos propiedades: la frecuencia (que en la figura 16.1 tiene un valor de 100) y el volumen (que tiene un valor de 7 en la misma figura). Mediante los botones "+" y de frecuencia y de volumen podemos cambiar los valores de estas dos propiedades. En la POO se denomina atributos a las propiedades de los objetos.
La forma de comunicarnos con este objeto es a través de sus botones. Cuando queremos bajar el volumen pulsamos el botón del volumen. En POO se diría que al objeto radio le enviamos un mensaje: bajar el volumen. Al pulsar ese botón se pone en funcionamiento el circuito para disminuir el volumen. En POO se diría que al enviar el mensaje "bajar el volumen" al objeto se ejecuta el método que se encarga de disminuir el valor del volumen.
Esta radio podemos utilizarla tranquilamente sin necesidad de saber cómo funciona o cómo está hecha, y sólo podemos cambiar el volumen y la frecuencia a través de los botones correspondientes que hay en la misma radio. En POO, a esta característica que consiste en que los objetos son como cajas

negras a cuyas propiedades sólo se puede acceder a través de esos botones se le denomina encapsulación.
Así que un objeto tiene, básicamente, unas propiedades (como el volumen y la frecuencia de la radio digital) y un circuito (por ejemplo, para aumentar y disminuir la frecuencia y el volumen de la radio digital). En POO, un objeto tiene, básicamente, atributos y métodos.
Pero ¿cómo podemos crear un objeto radio en C++?

Clases y objetos

En la vida cotidiana, antes de crear un objeto, por ejemplo una radio digital, tenemos que: crear los planos del circuito, diseñar cuántos botones va a tener y cuál será la función de cada uno, especificar cuántas propiedades necesita, etc. Una vez que el diseño está completo ya podemos fabricar las radios. En POO, a ese diseño se le denomina clase y a lo que se "fabrica" a partir de la clase se le denomina objeto.
De momento vamos a ver una sintaxis simplificada de la definición de una clase:

```
class nombre
{

   declaracion_de_atributos

   declaracion_de_metodos
};
```

Donde `nombre` es el identificador que daremos a la clase, en `declaracion_ de_atributos` declararemos los atributos y en `declaracion_de_ métodos` declararemos los métodos de la clase. Debe quedar claro que los atributos son simples variables (de tipo entero, estructuras, *arrays,* etc.) y que los métodos son simples funciones que trabajan con los atributos.
Por otro lado, en la radio hay partes que no son accesibles directamente desde el exterior, como la propiedad frecuencia y volumen, y otras que sí, como los botones. El hecho de que algo sea o no accesible directamente se puede conseguir en una clase mediante los modificadores `public` (público), que permite que algo sea accesible desde el exterior del objeto, y `prívate` (privado), que permite que algo sea accesible sólo desde el interior del objeto. Lo normal es que los atributos sean `prívate`, por aquello de la encapsulación, mientras que aquellos métodos que queramos que sean accesibles desde el exterior del objeto serán `public`. Si no queremos que algún método sea accesible desde el exterior lo haremos `prívate`.

Veamos ahora la definición de la clase radio y el uso de los modificadores. Esta definición es muy simple, para poder captar bien las ideas, pero le animamos a que haga todas las mejoras que considere sobre ella.

```
class radio
{
   prívate:
      int frecuencia;
      int volumen;
   public:
      void inicializar (void);
      void subir_volumen (void);
      void bajar_volumen (void);
      void subir_frecuencia (void);
      void bajar_frecuencia (void);
      void mostrar_volumen (void);
      void mostrar_frecuencia (void);
} ;
```

Podemos ver dentro de la clase radio dos zonas: `private` y `public`. Dentro de la zona privada hemos declarado los atributos: una variable para almacenar el valor de la frecuencia y otra para el valor del volumen. Esta zona va desde el modificador `private` hasta el modificador `public`. Por defecto todo es privado, por lo que no sería necesario haber escrito `private`.

En la zona pública hemos declarado los métodos de la clase: cuatro funciones que permitirán aumentar o bien disminuir el valor de las variables `frecuencia y volumen`, más una quinta función llamada `inicializar`, que se encargará de dar un valor inicial a las variables `frecuencia y volumen`. La zona pública va desde el modificador `public` hasta el final de la definición de la clase.

Tras definir la clase tenemos que definir los métodos, ya que sólo los hemos declarado. La definición de un método es igual que la de cualquier otra función, pero delante del nombre del método debemos escribir el nombre de la clase, dos puntos y otros dos puntos:

```
void radio::inicializar (void)
{
    // Da un valor inicial a los atributos volumen y frecuencia,
    frecuencia=100; volumen=7;
}

void radio::subir_volumen (void)
{
    // Incrementa en una unidad el valor del atributo volumen.
    volumen++;
}

void radio::bajar volumen (void)
```

```
{
    // Décrémenta en una unidad el valor del atributo volumen,
    volumen--;
}

void radio::subir_frecuencia (void)
{
    // Incrementa en una unidad el valor del atributo frecuencia,
    frecuencia++;
}

void radio::bajar_frecuencia (void)
{
    // Décrémenta en una unidad el valor del atributo frecuencia,
    frecuencia--;
}


void radio::mostrar_volumen (void)
{
    // Muestra en pantalla el valor del atributo volumen, cout <<
    "\nVolumen: " << volumen;
}

void radio::mostrar_frecuencia (void)
{
    // Muestra en pantalla el valor del atributo frecuencia, cout <<
    "\nFrecuencia: " << frecuencia;
}
```

Los métodos de una clase pueden acceder de manera directa a sus atributos, por lo que no hay que hacer ningún tipo de paso por parámetro para manipularlos.

Bueno, pues ya tenemos definida completamente la clase. Todo está listo para declarar (crear) un objeto de la clase radio. En POO, cuando se crea un objeto de una clase se dice que se crea una instancia de la clase, por lo que con la palabra instancia se hace referencia al objeto en sí mismo. La sintaxis para crear una instancia de una clase es:

```
nombre_clase nombre_objeto;
```

Donde `nombre_clase` es el identificador de la clase de la cual queremos crear un objeto (por ejemplo: radio) y `nombre obj eto` es el identificador de dicho objeto. Por ejemplo:

```
int main (void)
{
    radio mi_radio_digital;
}
```

Una vez que tenemos el objeto querremos utilizar sus métodos. Para ello, escribimos el nombre del objeto, un punto y el método:

```
int main (void)
{
   radio mi_radio digital;

   mi_radio__digital. inicializar () ;
   mi_radio_digital.mostrar_frecuencia() ;
   mi_radio_digital.mostrar_volumen();
   mi_radio_digital.baj ar_frecuencia();
   mi_radio_digital.subir_volumen();
   mi_radio_digital.mostrar_frecuencia() ;
   mi_radio_digital.mostrar_volumen();
}
```

Tras ejecutar este código aparecerá en la pantalla:

```
Frecuencia: 100

Volumen: 7

Frecuencia: 99

Volumen: 8
```

Para probar este programa escriba estas partes en el orden que se indica:

- Primero, los `#include` que hagan falta, como por ejemplo: `#include ciostream.h>.`
- Después, la definición de la clase radio.
- En tercer lugar, la declaración de los métodos de la clase radio.
- Y en último lugar, la función principal `main.`

Métodos, parámetros y return

Claro está que los métodos pueden tener parámetros de igual manera que los estudiados hasta el momento. Por ejemplo, el anterior método `inicializar` lo podemos definir y declarar de forma que nos permita indicarle por parámetro el valor inicial de la frecuencia y el volumen. La definición sería del tipo:

```
void inicializar (int fre, int vol);
```

Luego la declaración sería:

```
void radio::inicializar (int fre, int vol)
{
   frecuencia = fre; volumen = vol;
}
```

En donde el valor pasado por el parámetro `fre` se le asigna al atributo `frecuencia` y el valor pasado por el parámetro `vol` se le asigna al atributo `volumen.` Un ejemplo de su uso es el siguiente:

```
int main (void)
{
    radio mi_radio_digital;

    mi_radio_digital.inicializar(200, 14) ;
    mi_radio_digital.mostrar_frecuencia();
    mi_radio_digital.mostrar_volumen();
}
```

Tras ejecutar este código aparecerá en la pantalla:

```
Frecuencia: 200
Volumen: 14
```

También podría ocurrir que nos interesase (a nosotros como programadores) conocer el valor de un atributo, por ejemplo la frecuencia, sin que se mostrase por pantalla. Entonces, crearíamos un método que retornase dicho atributo. Su declaración sería del tipo:

```
int devolver_frecuencia (void);
```

Y su declaración:

```
int radio::devolver_frecuencia (void)
{
    return frecuencia;
}
```

Sencillo, ¿verdad?

Punteros a objetos

En algunas ocasiones, en POO, es necesario trabajar con punteros a objetos. La sintaxis es:

```
nombre_clase *puntero_objeto; puntero_objeto = new nombre_clase;
```

Donde nombre_clase es el identificador de la clase de la que vamos a crear un objeto y puntero_objeto es el identificador de la variable puntero al objeto que creamos. La primera línea declara sólo una variable puntero a la clase. En la siguiente línea, mediante el operador new, se reserva memoria para el objeto y se crea la instancia a la clase. A la variable puntero_ob j eto le asignamos la dirección de memoria donde se encuentra el objeto creado, es decir, la variable puntero_ob j eto apunta al objeto creado. Ejemplo:

```
radio *mi_radio_digital;
mi_radio_digital = new radio;
```

Ahora, mi_radio_digital no es un objeto, sino un puntero al objeto. También se puede crear un puntero a un objeto en una sola línea.

Lo siguiente es equivalente a la anterior declaración:

```
radio *mi_radio_digital = new radio;
```

Antes de finalizar el programa debemos asegurarnos de liberar la memoria reservada con el operador `new`, para lo cual utilizamos el operador `delete` seguido de un espacio, el identificador del puntero a la zona de memoria reservada y un punto y coma:

```
delete mi_radio_digital;
```

La sintaxis para acceder a los métodos de un objeto mediante punteros es:

```
puntero_objeto->metodo(parametros)
```

Donde `método` es el nombre del método a usar y `parametros` son los parámetros que le tengamos que pasar, si es que los tiene. Veamos un ejemplo completo en la función main, ya que la definición de la clase y la declaración de sus métodos no cambian:

```
int main (void)
{
   // Reservamos memoria para el objeto y lo creamos,
   radio *mi_radio_digital = new radio;

   mi_radio_digital->inicializar(50, 10) ;
   mi_radio_digital->mostrar_frecuencia();
   mi_radio_digital->mostrar_volumen();

   // Liberamos la memoria reservada con new.
   delete mi_radio_digital;
}
```

Tras ejecutar este código aparecerá en la pantalla:

```
Frecuencia: 50
Volumen: 10
```

Constructores y destructores

Un objeto tiene siempre un momento en el que comienza a existir (cuando se crea) y un momento en el que deja de existir, al igual que cualquier variable. Si no recuerda esto repase el ámbito de las variables locales y globales en este mismo libro. Pues en el momento en el que se crea un objeto se invoca automáticamente al método constructor, y en el que deja de existir se invoca al método destructor. El método constructor se utiliza normalmente para inicializar los atributos, mientras que el método destructor se suele utilizar para liberar la memoria que haya podido ser reservada.

El método constructor no retorna nada, ni void, pero sí puede tener parámetros. Además podemos sobrecargarlo creando más de un método constructor, todos con el mismo nombre, pero cada uno con parámetros distintos en tipo o número. Cuando se cree el objeto se ejecutará el constructor que coincida en el tipo o número de parámetros pasados. Al igual que podemos sobrecargar el método constructor también podemos sobrecargar otros métodos.
Si no creamos un constructor se ejecutará uno por defecto y vacío.
El método destructor también tiene el mismo nombre de la clase, pero se le antepone el símbolo ~. No retorna nada, ni void, ni acepta parámetros, lo que implica que sólo puede haber un método destructor.
Este método no puede ser llamado explícitamente, ya que se ejecuta de forma automática.
Con los constructores ya no haría falta el método inicializar de la anterior clase radio. La definición de la clase quedaría de la siguiente manera con dos constructores y un destructor:

```
class radio
{
    prívate:
       int frecuencia;
    int volumen;
    public:
       radio (void);
       radio (int fre, int vol);
       void subir_volumen (void);
       void bajar_volumen (void);
       void subir_frecuencia (void);
       void bajar_frecuencia (void);
       void mostrar_volumen (void);
       void mostrar_frecuencia (void);
       ~radio();
};
```

Se han incluido dos constructores: uno sin parámetros, que pone los atributos a cero, y otro con parámetros, que inicializa los atributos con los valores pasados por parámetro. Veamos las declaraciones de los constructores y el destructor.

```
radio::radio (void)
{
    frecuencia = 0;
    volumen = 0;
}

radio::radio (int fre, int vol)
{
    frecuencia = fre;
    volumen = vol;
}
```

```
radio::~radio()
{
}
```

El destructor lo hemos declarado vacío, puesto que no tenemos que liberar ninguna memoria ni realizar ninguna otra operación cuando el objeto se destruya.
En la siguiente función `main` se crean dos objetos invocando a cada constructor. En la primera declaración se ejecuta el constructor sin parámetros, quedando los atributos inicializados a 0. En la segunda declaración se ejecuta el constructor con parámetros.

```
int main (void)
{

   radio mi_radio_digital_1;
   radio mi_radio_digital_2 (90, 10);

   cout << "\nRadio 1: ";
   mi_radio_digital_1.mostrar_frecuencia();
   mi_radio_digital_1.mostrar_volumen();
   cout << "\nRadio 2:";
   mi_radio_digital_2.mostrar_frecuencia();
   mi_radio_digital_2.mostrar_volumen();
   cout << "\nRadio 3:";
   mi__radio_digital_3 .mostrar_frecuencia () ;
   mi_radio_digital_3.mostrar_volumen();
}
```

Tras ejecutar este código aparecerá en la pantalla:

```
-- Radiol ---
Frecuencia: 0
Volumen: 0
-- Radio 2 --
Frecuencia: 90
Volumen: 10
```

Sobrecarga de métodos

Al hablar de los constructores ya hemos adelantado que la sobrecarga de métodos consiste en crear más de un método con el mismo identificador y distintos parámetros en tipo o número. Por ejemplo, en la siguiente clase hay varios métodos sobrecargados:

```
class sumar
{
   public:
      int suma (int a, int b);
      int suma (int a, int b, int c) ;
```

```
        float suma (float a, float b);
        float suma (float a, float b, float c);
}
```

Si en un programa llamamos al método sumar se ejecutará aquel cuyos parámetros coincidan en tipo y número con los valores pasados. Ejemplo:

```
int main (void)
{
    sumar mi_suma;

    // Ejecuta el primer método definido en la clase,
    cout << "\n" « mi_suma (1, 2);

    // Ejecuta el segundo método definido en la clase,
    cout << "\n" << mi_suma (1, 2, 3);

    // Ejecuta el tercer método definido en la clase,
    cout << "\n" « mi_suma (3.1, 2.7);

    // Ejecuta el cuarto método definido en la clase,
    cout << "\n" « mi_suma (3.1, 2.7, 10.2);
}
```

La sobrecarga nos permite, utilizando un mismo identificador, ejecutar el método más adecuado a los parámetros pasados. Trabajar con un objeto con métodos sobrecargados es muy cómodo.

Herencia

La herencia es una de las características principales de la programación orientada a objetos. Permite que una clase herede atributos y métodos de otra, y añadir otros nuevos.

A la clase que hereda de otra se le denomina subclase, clase hija o clase derivada, mientras que la clase de la que hereda la subclase recibe el nombre de superclase, clase padre o clase base. Una subclase sólo puede heredar de una superclase, pero una superclase puede tener varias subclases (véase la figura 16.2).

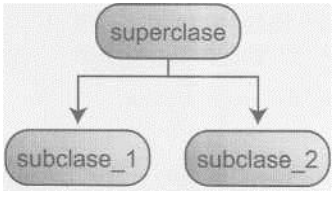


Figura 16.2. La herencia de una superclase a una subclase.

Definición de una jerarquía de clases

Por ejemplo, es obvio que tanto un alumno como un profesor son personas, por lo que tienen en común atributos como el nombre y la edad. Pero un alumno tiene atributos que un profesor no tiene, y viceversa. De esta manera, podemos tener una superclase persona, con unos atributos generales como el nombre y la edad, y dos subclases: alumnos y profesores. Los alumnos pueden tener como atributos particulares el número de expediente y la nota final. Los profesores pueden tener como atributos particulares el nombre de la asignatura que imparten y el sueldo. Cada clase tendrá los métodos correspondientes para trabajar con sus atributos. En la figura 16.3 vemos una representación de esta jerarquía, donde junto a cada clase se muestran sus atributos.

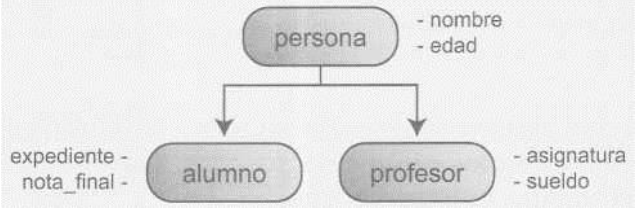


Figura 16.3. Jerarquía de clases: persona, alumno, profesor.

Aunque exista una herencia, las subclases sólo podrán acceder a los atributos de la superclase persona a través de los métodos que ésta proporcione y que sí heredarán. Puede parecer extraño hablar de herencia y que los atributos no se hereden, pero ésta es la forma de mantener la encapsulación.

¿Y cómo indicamos qué se hereda y qué no? Lo indicamos con los modificadores `public`, `private` y `protected`. El modificador `public` (público) permite el acceso desde el exterior del objeto y desde las subclases (hereda). El modificador `private` (privado) no permite el acceso desde el exterior del objeto y tampoco permite el acceso desde las subclases (no hereda). El modificador `protected` (protegido) no permite el acceso desde el exterior del objeto, pero sí permite el acceso desde las subclases (hereda).

Una subclase se define igual que una clase, pero tras su identificador se escriben dos puntos, `public` y el nombre de la superclase:

```
class nombre_subclase : public nombre_superclase {
};
```

Para comprender bien la herencia podemos definir la anterior jerarquía de clases de una manera muy simple:

```
class persona (
       // Por defecto es privado.
       char nombre[60];
       int edad;
   public:
       persona(char *nom, int eda) ;
       void mostrar_persona (void);
};

class alumno : public persona {
       // Por defecto es privado,
    int expediente; int nota_final;
    public:
       alumno (char *nom, int eda, int exp, int not) ;
       void mostrar_alumno (void) ;
}   ;

class profesor : public persona {
       // Por defecto es privado.
       char asignatura[30];
       int sueldo;
       public:
       profesor (char *nom, int eda, char *asi, int sue);
       void mostrar_profesor (void);
} ;
```

Accesibilidad a atributos y métodos

Siguiendo con la definición de la jerarquía anterior, las subclases alumno y profesor heredarán de la superclase persona el método:

```
void mostrar_persona (void);
```

Veamos la implementación de los métodos mostrar.

```
void persona::mostrar_persona (void)
{
    // Mostramos los datos de la persona,
    cout << "\nNombre: " << nombre;
    cout << "\nEdad: " << edad;
}

void alumno::mostrar_alumno (void)
{
    // Mostramos los datos de la persona.
    mostrar_persona();

    // Mostramos los datos del alumno,
    cout << "\nExpediente: " « expediente;
    cout << "\nNota final: " « nota;
}
```

```
void profesor::mostrar_profesor (void)
{
   // Mostramos los datos de la persona.
   mostrar_persona();

   // Mostramos los datos del profesor, cout «
   "\nAsignatura: " « asignatura; cout << "\nSueldo: "
   << sueldo;
}
```

Las subclases pueden utilizar los miembros públicos (también los protegidos si los hubiese) de la superclase como si estuviesen implementadas en las mismas subclases.

De esa manera, los métodos mostrar de las subclases pueden llamar al método mostrar de la superclase, como puede observarse, y conseguir mostrar los datos de la persona. Ésta es la única manera de acceder a los atributos de la superclase: a través de sus métodos. El siguiente ejemplo sería incorrecto y produciría un error, ya que desde una subclase se está intentando acceder a los atributos privados de la superclase:

```
void profesor:;mostrar_profesor (void)
{
   // Error de acceso a atributos privados de superclase.
   cout << "\nNombre: " << nombre;
   cout << "\nEdad: " << edad;

   // Mostramos los datos del profesor,
   cout << "\nAsignatura: " << asignatura;
   cout << "\nSueldo: " << sueldo;
}
```

Los constructores en la herencia

Cuando se llama al constructor de una subclase también se invoca al constructor por defecto (vacío) de la superclase. Pero, si el constructor de la superclase necesita parámetros, debemos llamarle explícitamente en el constructor de la subclase. En este caso, primero se inicializan los atributos de la superclase y después los atributos de la subclase. La sintaxis de la declaración del constructor de la subclase es del tipo:

```
nombre_subclase (decl_param_super, decl_param_sub) ;
```

Mientras que la sintaxis en la declaración es del tipo:

```
nombre_subclase: : nombre__subclase (decl_param_super, decl_param_sub)
        : nombre_superclase (param__super)
{
   // Inicilizacion de atributos con parametros de subclase.
}
```

Donde decl_param_super es la declaración de los parámetros de la superclase, decl_param_sub es la declaración de los parámetros de la subclase y param_super son los parámetros de la llamada al constructor de la superclase, no su declaración.
La declaración de los constructores de la anterior jerarquía de clases ya se mostraba en la definición de las clases.
Sus declaraciones son las siguientes:

```
persona::persona(char *nom, int eda)
{
     strcpy (nombre, nom);
     edad = eda;
}

alumno::alumno (char *nom, int eda, int exp, int not)
            : persona(nom, eda)
{
     expediente = exp;
     nota_final = not;
}

profesor:¡profesor (char *nom, int eda, char *asi, int sue)
            : persona(nom, eda) í
     strcpy (asignatura, asi);
     sueldo = sue;
}
```

En la herencia, en primer lugar se construye la superclase y, después, la subclase. En la destrucción ocurre lo contrario, primero se destruye la subclase y después la superclase.

Instancias de subclases

Siguiendo la anterior definición de la jerarquía de clases y la implementación de sus métodos veamos cómo se crea una instancia de una subclase utilizando los constructores con parámetros y cómo se trabaja con el objeto y sus métodos.

```
int main (void)

     alumno estudiante_l ("Michel", 26, 301, 7);
     alumno estudiante_2 ("Anne", '26, 125, 8);
     profesor profe_l ("Cris", 20, "Music", 1500);

     estudiante_l.mostrar_alumno();
     estudiante_2.mostrar_alumno();
     profe_1.mostrar_profesor();

     }
```

¿Alguna diferencia notable hasta este momento en el uso de los objetos de subclases? No, ¿verdad? Eso es así porque la diferencia real se encuentra en cómo está organizada la información en las distintas clases que forman la jerarquía.

Polimorfismo

Se podría definir el polimorfismo como la habilidad de un objeto para cambiar de forma. Esto permite tener en un mismo *array* distintos objetos, como alumnos y profesores. Para conseguir el polimorfismo hay que trabajar con punteros a los objetos y con métodos redefinidos. Se dice que un método está redefinido cuando en las distintas clases de la jerarquía existe un método con el mismo nombre y los mismos parámetros, pero con implementación distinta. En el polimorfismo, estos métodos deben llevar el modificador `virtual` en la función redefinida que se declara por primera vez, es decir, la de la superclase.
La anterior jerarquía de clases preparada para el polimorfismo:

```
class persona {
       // Por defecto es privado,
    char nombre[60];
       int edad;
       public:
       persona(char *nom, int eda) ;
       virtual void mostrar (void);
};


class alumno : public persona {
       // Por defecto es privado,
    int expediente;
       int nota_final;
       public:
       alumno (char *nom, int eda, int exp, int not);
       void mostrar (void);


class profesor : public persona {
       // Por defecto es privado,
    char asignatura[30];
       int sueldo;
       public:
       profesor (char *nom, int eda, char *asi, int sue);
       void mostrar (void);
};
```

Ahora, el método `mostrar` está redefinido (el prototipo es el mismo) y es virtual.

Sus declaraciones son:

```
void persona:¡mostrar (void)
{
    // Mostramos los datos de la persona, cout
    << "\nNombre: " << nombre;
    cout << "\nEdad: " << edad;
}

void alumno:¡mostrar (void)
{
    // Mostramos los datos de la persona,
    persona:¡mostrar();

    // Mostramos los datos del alumno,
    cout << "\nExpediente: " << expediente;
    cout << "\nNota final: " << nota;
}

void profesor::mostrar (void)
{
    // Mostramos los datos de la persona,
    persona::mostrar();

    // Mostramos los datos del profesor,
    cout << "\nAsignatura: " << asignatura;
    cout << M\nSueldo: " << sueldo;
}
```

Los métodos mostrar de las subclases tienen una llamada al método mostrar de la superclase, para lo que hay que escribir: el nombre de la superclase, dos puntos, dos puntos y el nombre del método de la superclase.
Esto es necesario porque el método mostrar está redefinido y si escribimos únicamente:

```
mostrar () ;
```

se ejecutará el método más próximo con ese nombre, es decir, él mismo, obteniendo una función recursiva y sin fin.
Ya hemos dicho que para que haya polimorfismo debe existir redefinición, métodos virtuales y punteros a los objetos.
Faltan los punteros, que los vamos a ver en el programa que se muestra a continuación:

```
int main (void)
{
    persona *vector[3];

    // Creamos los objetos.
    vector[0] = new alumno ("Michel", 26, 301, 7);
    vector[1] = new alumno ("Anne", 26, 125, 8);
    vector[2] = new profesor ("Cris", 20, "Music", 1500);
```

```
    // Mostramos los objetos,
    vector[0]->mostrar();
    vector[1]->mostrar();
    vector[2]->mostrar();
}
```

En línea:

```
persona *vector[3];
```

Declaramos un *array* de tres celdas con punteros a la clase `persona`. A continuación, se asigna a cada celda una referencia a la subclase `alumno` o bien `profesor`. ¡Hemos conseguido un *array* con distintos objetos, esto es el polimorfismo! Y lo mejor viene ahora.

Cuando llamemos al método `mostrar` de cada objeto (recuerde que como son punteros a los objetos utilizamos "->" en vez de un punto para acceder a los métodos) se ejecutará el correspondiente al alumno o al profesor, lo cual se consigue gracias a que son métodos virtuales.

Ejercicios resueltos

Intente realizar los siguientes ejercicios. Encontrará la solución al final de este libro.

1. Defina la clase y declare sus métodos para el siguiente objeto: una nave de un videojuego. Se sabe que la nave se puede situar en unas coordenadas (x,y) de la pantalla y que tiene un máximo de tres vidas. Además, la nave puede moverse en la pantalla (subir, bajar, ir a la derecha e ir a la izquierda) y ser destruida (no más de tres veces).
2. Defina una jerarquía de clases y sus métodos para los siguientes objetos: vehículo, coche, moto. Tanto el coche como la moto son vehículos. Ambos vehículos tienen un número de ruedas y una velocidad máxima. Por otro lado, el coche tiene una característica que las motos no tienen: el número de puertas. Mientras que las motos tienen la característica de admitir una o dos plazas.

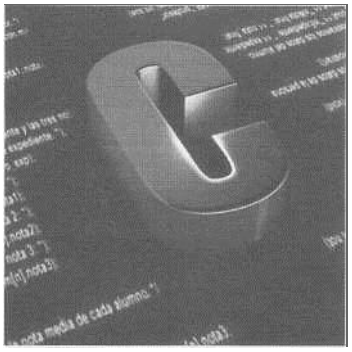
Resumen

La programación orientada a objetos cuenta con una gran cantidad de ventajas, de ahí la gran acogida que tiene. Por ejemplo, la POO favorece el diseño de software complejo, pues su mecanismo permite utilizar diversos

objetos que constituyan finalmente una aplicación. Piense que si un objeto está bien diseñado puede ser utilizado por cualquier persona en sus programas, dado que son módulos totalmente independientes.

Otra ventaja es que una clase se escribe una vez y a partir de ahí se pueden crear tantos objetos como se deseen de dicha clase. Las características de cada objeto dependerán de cómo se le trate. Piense en lo bueno que es esto a la hora de diseñar videojuegos en los que hay una gran cantidad de elementos, como naves, personas u otros seres.

En este capítulo se ha hecho una simple introducción a la programación orientada a objetos, pues explicar todas sus características y posibilidades en detalle está fuera del propósito de este libro, por extensión y complejidad.



Capítulo 17

Técnicas de programación

En este capítulo aprenderá:

- Qué es la programación convencional.
- Qué es la programación modular.
- Qué es la programación estructurada.
- Qué es la programación orientada a objetos.

Introducción

En la historia de la programación se han buscado técnicas que hicieran más fácil no sólo la implementación del código sino también su corrección y posibles modificaciones.
Esto ha llevado a una evolución en la forma de programar que ha aumentado la productividad, minimizando la complejidad y facilitando la corrección de errores.
Algunas de estas técnicas ya han sido explicadas en este libro, por lo que haremos un rápido recorrido desde la programación convencional hasta la programación orientada a objetos.

Programación convencional

Se podría decir que su único objetivo es que el programa funcione. El resultado es un programa con un bloque de código indiviso que utiliza saltos incondicionales y rutinas entremezcladas. De esta forma se genera un código muy poco claro y con un coste muy elevado en recursos. Además, añade una gran dificultad a la hora de depurar errores, mantenerlo y realizar futuras modificaciones.
Como los programas son cada vez más grandes, y hay mayores necesidades de modificarlos, esta técnica se hace insostenible: los costes de mantenimiento y producción se elevan demasiado.
Suele ser una de las prácticas de los programadores noveles sin disciplina, pero hay que evitarla a toda costa.

Programación estructurada

Como solución a los problemas que presentaba la programación convencional surge la programación estructurada. Consiste en la utilización de estructuras del control básicas:

- Secuencial.
- Alternativa o condicional `(if, if-else, switch)`.
- Iterativa `(while, do-while, for)`.

También se prohíbe el salto incondicional `(goto)` y se propone la elección de estructuras de datos adecuadas.

De esta manera, los programas tienen claramente un único inicio y un único final. El flujo del programa es único: de arriba abajo.
Aunque el programa es ya bastante claro para el programador, sigue existiendo un problema: los programas grandes siguen siendo muy difíciles de mantener.

Programación modular

La programación modular, con la idea "divide y vencerás", propone dividir el programa en otros más sencillos a los que se les denomina módulos. Cada módulo lleva a cabo una tarea específica y debe existir obligatoriamente un módulo principal.
Algunas de las ventajas de la programación modular son:

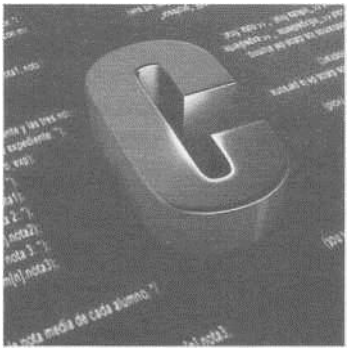
- Nos permite descomponer el programa en subprogramas.
- Se facilita la corrección, modificación y actualización del código.
- La verificación de los subprogramas puede hacerse por separado: para comprobar un subprograma no es necesario probar el resto del programa.
- Los programas son más claros.
- Los programas se pueden crear entre varias personas: cada persona puede crear un subprograma y posteriormente juntarlos en un programa.

Programación orientada a objetos

Basada en la idea natural de un mundo lleno de objetos, este tipo de programación se ha impuesto, ya que se asemeja a la forma de interactuar con los objetos cotidianos. La idea es que para interactuar con unos datos debemos hacerlo a través de unas funciones concretas creadas para tal fin, de la misma manera que para interactuar con un televisor lo hacemos a través de unos botones determinados.
El programador implementa objetos, o simplemente los usa, o aprovecha parte de su comportamiento.
Para usar un objeto no es necesario conocer cómo está implementado, sólo debemos utilizar los métodos que nos ofrece. A esta idea se la conoce como encapsulación.
Otras características de la programación orientada a objetos son la herencia y el polimorfismo.

Resumen

Queda claro que las técnicas de programación no se inventaron por diversión, sino para crear programas fáciles de mantener y modificar, sin olvidar que permitiesen detectar errores rápidamente. No tenga pereza a una buena técnica, con el tiempo lo agradecerá.



Capítulo 18

Algoritmos de ordenación y búsqueda

En este capítulo aprenderá:

- A ordenar un *afray* y a buscar en él.
- El método de la burbuja.
- El método de selección directa.
- El método de búsqueda secuencial.
- El método de búsqueda binaria.

Introducción

En este capítulo vamos a aprender algunos de los algoritmos para ordenar un *array* y realizar búsquedas en él. Aunque el tipo de elementos que almacena el *array* puede hacer variar levemente algunas partes de los algoritmos, la estrategia de éstos no cambia.
Utilizaremos para explicar los algoritmos un *array* de números enteros con cuatro celdas y con valores almacenados en todas sus celdas:

```
int vector[4] = {3, 2, 4, 1};
```

También utilizaremos otras variables en los algoritmos:

```
int i, j, k, aux;
```

Son bastantes los algoritmos existentes, pero veremos tan sólo algunos de los más sencillos.

Algoritmos de ordenación

Algunos de los métodos de ordenación son: burbuja, inserción, selección y quicksort.
Analizaremos el método de la burbuja y de selección directa.

Método de la burbuja

Este método de ordenación consiste en comparar los elementos adyacentes de un *array* e intercambiarlos. Para el caso de la ordenación ascendente, dicho intercambio se realiza cuando un elemento situado a la izquierda es mayor que otro situado a su derecha.

```
for (i=0; i<=N-1; i++)
    for (j=N; j>=i+1; j—)
       if (vector[j-1] > vector [j])
       {
          aux = vector[j];
          vector[j] = vector[j-1];
          vector[j — 1] = aux;
       }
```

Donde `N` es el índice de la última celda. Si hace una traza se dará cuenta de que el elemento menor "sube" como una burbuja hasta el principio del *array* (véase la figura 18.1).

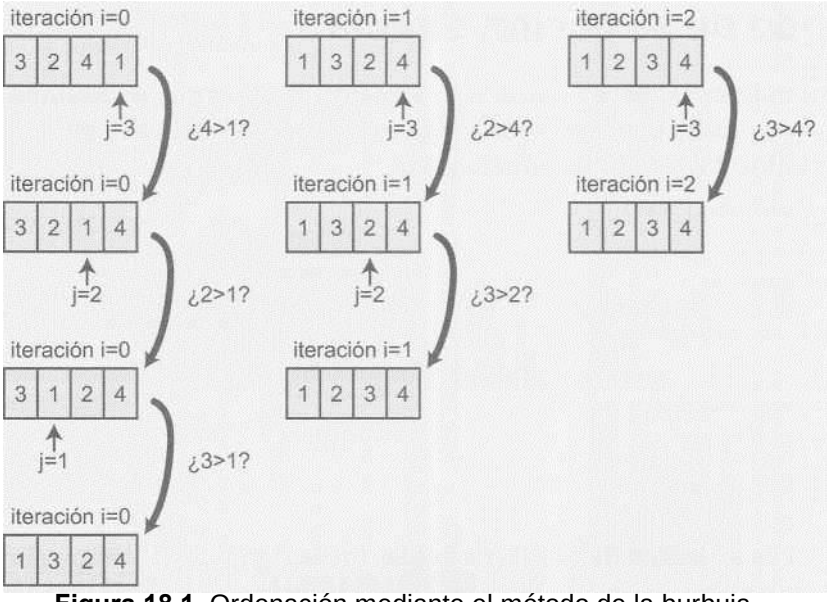


Figura 18.1. Ordenación mediante el método de la burbuja.

Este listado ordena por el método de la burbuja el *array* de números.

```
#include <stdio.h>

int main (void)
{
    /* Declaramos el array de números
    */ int vector[4] = {3, 2, 4, 1};
    int i, j, aux;

    /* Ordenamos el array de números */
    for (i=0; i<=2; i++)
       for (j =3; j>=i+1; j--)
          if (vector[j-1] > vector [j])
          {
             aux = vector[j];
             vector[j] = vector[j-1];
             vector[j-1] = aux;
          }

    /* Mostramos el array de números */
    for (i=0; i<=3; i++)
       printf ("%d ", vector[i]);

    /* Hacemos una pausa hasta que el usuario pulse Intro */
    fflush(stdin);
    printf("\n\nPulse Intro para finalizar...");
    getchar ();
    }
```

Método de selección directa

Este método consiste en tomar un elemento k del *array,* a continuación se busca entre todas las celdas que tenga a su derecha, inclusive ella misma, el menor valor y después los intercambia.

```
for (k=0; k<=N-1; k++)
{
    i=k;
    aux=vector[k];
    for (j=k+1; j<=N; j++)
    if (vector[j]<aux)
    {
        i=j;
        aux=vector[i];
    }
    vector [i] = vector[k];
    vector [k] = aux;
    }
```

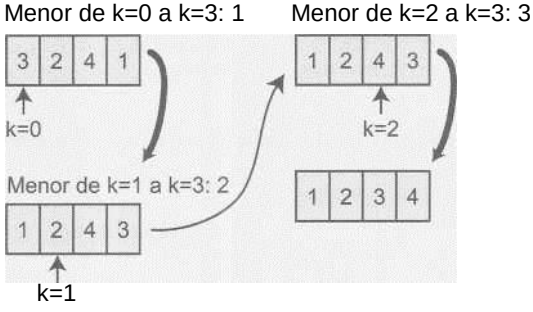


Figura 18.2. Ordenación mediante el método de selección directa.

Este listado ordena por el método de selección directa el *array* de números.

```
#include <stdio.h>

int main (void)
{
    /* Declaramos el array de números */

    int vector[4] = {3, 2, 4, 1};

    int i, j, aux;

    /* Ordenamos el array de números */
    for (k=0; k<=2; k++)
    {
       i=k;
       aux=vector[k];
```

```
      for (j=k+1; j<=3; j++)
         if (vector[j]<aux)
         {
            i=j ;
            aux=vector[i];
         }
      vector[i] = vector[k] ;
      vector[k] = aux;
   }

   /* Mostramos el array de números */
   for (i=0; i<=3; i++)
      printf ("%d ", vector[i]);

   /* Hacemos una pausa hasta que el usuario pulse Intro */
   fflush(stdin);
   printf("\n\nPulse Intro para finalizar...");
   getchar();
}
```

Algoritmos de búsqueda

Buscar un elemento en un *array* es una operación muy típica. Analizaremos el método de la búsqueda secuencial y de la búsqueda binaria.

Método de búsqueda secuencial

Si vamos a realizar la búsqueda en un *array* que no está ordenado debemos realizar este tipo de búsqueda. Consiste en recorrer el *array* hasta encontrar el elemento o llegar al final de éste.

```
i=0 ;
while ((i<=N) && (vector[i]!=elem))
{
   i++;
}

if (i>N)
{
   /* Codigo para el caso de no encontrar el elemento */
}
else
{
   /* Codigo para el caso de encontrar el elemento en la celda i */
}
```

En donde **N** es el índice de la última celda **y elem** es el valor del elemento buscado.

Esta búsqueda también se puede utilizar en un *array* que esté ordenado, aunque esto no sería lo más adecuado.
El siguiente listado busca mediante el método de búsqueda secuencial un número indicado por el usuario en el *array* de números.

```
#include <stdio.h>

int main (void)
{
   /* Declaramos el array de números
   */ int vector[4] = {3, 2, 4, 1);
   int i, elem;

   /* Pedimos el numero al usuario */
   printf ("¿Que numero quiere buscar? ");
   scanf ("%d", Selem);

   /* Buscamos el numero */
   i=0;
   while ((i<=3) && (vector[i]!=elem))
   {
      i++;
   }

   /* Comprobamos si se ha encontrado el numero */
   if (i>3)
      printf("\nEl numero %d no esta en el array.", elem);
   else
      printf("\nEl numero %d si esta en el array.", elem);

   /* Hacemos una pausa hasta que el usuario pulse Intro */
   fflush(stdin);
   printf("\n\nPulse Intro para finalizar...");
   getchar();
}
```

Método de búsqueda binaria

Sólo se puede utilizar en *arrays* que ya estén ordenados, en cuyo caso es más óptimo que la búsqueda secuencial.
Consiste en ir dividiendo el espacio de búsqueda. Se consulta si el valor buscado está en la mitad derecha o en la mitad izquierda del vector. Se toma la mitad en la que se encuentra y se consulta si el valor buscado está en su mitad derecha o en su mitad izquierda. Y así sucesivamente hasta encontrar el elemento.

```
i=0
;
j=N
; do
```

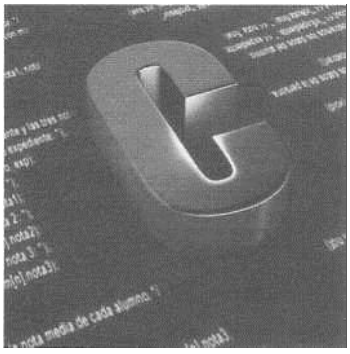
```
{
   medio = (i+j) / 2;
   if (elem < vector[medio])
      j=medio-1;else
      i=medio+1;
} while ((vector[medio]!=elem) && (i<=j));

if ((vector[medio]!=elem)
{
   /* Codigo para el caso de no encontrar el elemento */
}
else
{
   /* Codigo para el caso de encontrar el elemento en la celda i */
}
```

El siguiente listado busca mediante el método de búsqueda binaria un número indicado por el usuario en el *array* de números.

```
#include <stdio.h>

int main (void)


{
   /* Declaramos el array de números */

   int vector[4] = [3, 2, 4, 1};

   int i, j, medio, elem;

   /* Pedimos el numero al usuario */

   printf ("¿Que numero quiere buscar? ");

   scanf ("%d", Selem);

   /* Buscamos el numero */ i=0; j=3; do
   {
      medio = (i+j) / 2;
      if (elem < vector[medio])
         j=medio-l;
      else
         i=medio+l;
   } while ((vector[medio]!=elem) && (i<=j));

   /* Comprobamos si se ha encontrado el numero */
   if (vector[medio]!=elem)
      printf("\nEl numero %d no esta en el array.", elem);
   else
      printf("\nEl numero %d si esta en el array.", elem);

   /* Hacemos una pausa hasta que el usuario pulse Intro */
   fflush(stdin);
   printf("\n\nPulse Intro para finalizar...");
   getchar ();
}
```

Resumen

Tras este capítulo ya podemos abordar programas que requieran realizar búsquedas en *arrays* y ordenaciones.
Es conveniente que estudie minuciosamente estos algoritmos para comprenderlos perfectamente. De esta manera podrá modificarlos para adaptarlos a sus necesidades.



Capítulo 19

Control de errores y validación de datos

En este capítulo aprenderá:

- Algoritmos para validar datos.
- Algoritmos para tranformar datos.

Introducción

En muchas ocasiones será nececesario controlar si los datos que el usuario introduce en el programa son los esperados o son correctos, en otras ocasiones habrá que comprobar si el resultado de una operación es válido, o transformar un dato a un determinado formato, etc.
En este capítulo analizaremos algunas funciones y algoritmos que permiten llevar a cabo estas tareas y que pueden ser de gran utilidad.

Controlar datos incorrectos

Si pedimos un determinado dato al usuario y éste lo escribe mal, debemos comprobarlo y volver a pedírselo hasta que sea correcto. Veamos un ejemplo muy sencillo.

```
#include <stdio.h>

int main (void)
{
    int edad;

    do
    {
       printf("\nEscribe tu edad: ");
       scanf("%d", &edad);
       fflush(stdin);

       if ( (edad<0) || (edad>120))
          printf("\nEl numero indicado no es valido.");

    } while((edad<0) || (edad>120));

    /* Hacemos una pausa hasta que el usuario pulse Intro */
    fflush(stdin);
    printf("\n\nPulse Intro para finalizar...");
    getchar();
}
```

En el algoritmo anterior, se solicita la edad al usuario, pero si escribe un número menor que 0 o mayor que 120, se mostrará un mensaje en pantalla avisando de que el número no es válido y se volverá a pedir el número gracias al bucle do-while. Así, no permitimos que el programa continúe con un valor irreal almacenado en una variable. Esta filosofía de programación debemos aplicarla en todo momento.
A continuación, trataremos algunas funciones que pueden ser de ayuda para controlar datos.

Contar palabras de una cadena

La siguiente función cuenta y retorna el número de palabras de una cadena pasada por parámetro. La estrategia es recorrer la cadena y contar el número de espacios, ya que tras cada palabra siempre se escribe un espacio Para recorrer la cadena emplearemos la variable posición y para contabilizar el número de palabras la variable numpalabras.

```
int contar_palabras(char *cadena)
{
   int posición = 0;
   int numpalabras = 0;

   /* Si el tamaño de la cadena es 0 entonces no tiene palabras,
   en caso contrario, recorre la cadena y cuenta las palabras. */
   if (strlen(cadena)==0)
   {
      numpalabras=0;
   }
   else
   {
      /* Recorremos la cadena y por cada espacio contamos una
      palabra. */
      for (posicion=0; posicion<strlen(cadena); posicion++)
      {
         if (cadena[posición]==' ')
         {
            Numpalabras++;
         }
      )
      /* La ultima palabra no esta seguida de ningún espacio
      asi que incrementamos incondicionamente en 1 el numero
      de palabras */ numpalabras++;
   }

   /* Retorna el numero de palabras. */
   return numpalabras;
}
```

Veamos un pequeño ejemplo de uso de la función `contar_palabras`.

```
int n;

n = contar_palabras("Esta es una frase de prueba");
printf("Hay %d palabras", n);
```

Otro ejemplo de uso de la función contar_palabras:

```
char frase[50];
int n;

printf("Escribe una frase: ");
```

```
gets(frase)
n = contar_palabras(frase) ;
printf("Hay %d palabras", n) ;
```

Eliminar espacios innecesarios de una cadena

La siguiente función elimina los espacios situados al principio y al final de la cadena pasada por parámetro, así como la secuencia de varios espacios.

```
void eliminar_espacios(char *cadena)
{
   char
   cadena_temp[256];
   int i = 0;
   int j = 0;
   int espacio = 0;

   /* Recorre la cadena pasada por parametro,
   ignorando los espacios no validos y copiando los
   caracteres validos en cadena_temp. */
   while(i<strlen(cadena))
   {
      if (cadena[i]!=' ')
      {
         cadena_temp[j] = cadena[i];
         j++;
         espacio=1;
      }
      else
      if (espacio>0)
      {
         cadena_temp[j] = cadena[i];
         j++;
         espacio = 0;
      }
      i++;
   }

   /* Si el ultimo carácter es un espacio, lo
   sustituye por el carácter de fin de cadena,
   si no, añade el carácter de fin de cadena. */
   if (cadena_temp[j-1]==' ')
      cadena_temp[j-1] = 1\0';
   else
      cadena_temp[j] = '\0';

   /* Copia la cadena sin espacios en la
   cadena pasada por parametro. */
   strcpy(cadena, cadena_temp);
}
```

Esta función, que requiere el uso de la librería `string`, utiliza de forma auxiliar la variable `cadena_temp` para almacenar la cadena pasada por parámetro sin los espacios innecesarios. La variable `i` es utilizada para recorrer `cadena`, mientras que la variable `j` es empleada para indicar la posición en la que debe almacenarse el carácter válido de `cadena` en `cadena_temp`. La variable `espacio` se utiliza para indicar si el siguiente espacio que encontremos en `cadena` debe ser ignorado (valor 0) o no (valor 1). La idea es que tras cada carácter distinto de espacio `(cadena [ i ]! =' ')` se permite un espacio `(espacio=1;)`.
Una vez se detecta un espacio, si `espacio>0`, dicho espacio es válido y se almacena en `cadena_temp`, tras lo cual, la variable `espacio` toma el valor 0, no permitiéndose almacenar más espacios hasta que no se encuentre otro carácter distinto. Cada vez que se almacena un carácter en `cadena_temp`, la variable j aumenta en uño para que la próxima vez se almacene en la siguiente posición. Finalmente, una vez se ha recorrido toda la cadena pasada por parámetro, `cadena__temp` es copiada en `cadena`.
La función `eliminar_espacios` puede mejorarse de forma que no sea necesario utilizar ninguna cadena auxiliar, como es `cadena_temp`. ¿Se atreve a intentar hacerla?
A continuación, vamos a ver un ejemplo de uso de la función `eliminar_espacios`.

```
int main (void)
{
   char frase[256];

   strcpy (frase, " Hola a todos ");
   eliminar_espacios(frase) ;
   printf("La cadena sin espacios: -%s-", frase);

   printf("\n\nPulse la tecla Intro para finalizar...");
   getchar ();
}
```

Eliminar todos los espacios de una cadena

Una modificación en la función anterior nos permitirá crear una función que elimine todos los espacios de la cadena pasada por parámetro.

```
void sin_espacios(char *cadena)
{
   char cadena_temp[256];
   int i = 0;
   int j = 0;
```

```
   /* Recorre la cadena pasada por parametro,
   ignorando los espacios. */
   while(i<strlen(cadena))
   {
      if (cadena[i]!=' ')
      {
         cadena_temp[j] = cadena[i];
         j++;
      }

      i++;
   }

   /* Inserta carácter de fin de cadena. */
   cadena_temp[j] = ' \ 0 ';

   /* Copia la cadena sin espacios en la
   cadena pasada por parametro. */
   strcpy(cadena, cadena_temp);
}
```

Comprobar extensión de un fichero

En algunas ocasiones se solicita al usuario el nombre de un archivo con una extensión concreta.
La siguiente función permite pasarle por parámetro el nombre del archivo y una extensión válida. Retorna 1 si el archivo tiene una extensión válida, o 0 en caso contrario.

```
int comprobar_extension(char *archivo, char *extensión)
{
   int resultado = 0;

   if (strstr(archivo, extensión)!=NULL)
      resultado = 1;

   return resultado;
}
```

A continuación, vemos un pequeño ejemplo de su uso.

```
if (comprobar_extension(nombre_archivo, "txt")==1)
{
   printf("Archivo con extensión valida.");
}
```

Donde nombre_archivo es una variable que almacena el nombre de un fichero dado por el usuario. El uso de esta función requiere incluir la librería string.

Comprobar formato fecha

Si solicitamos al usuario que escriba una fecha con un formato específico, por ejemplo dd-mm-aaaa, será necesario comprobar que aquello que escribe es correcto. Podemos utilizar la siquiente función, que retorna el valor 1 cuando el formato es correcto y 0 en caso contrario. Requiere el uso de las librerías string y stdlib.

```
int formato_fecha(char *fecha)
{
    char *p;
    int dia, mes, anio;

    /* Partimos de la premisa de que el resultado es
    invalido, asi que debemos demostrar lo contrario*/
    int resultado = 0;

    /* Antes de nada, comprobamos que el tamaño sea
    valido, "dd-mm-aaaa" tiene 10 caracteres */
    if (strlen(fecha)!=10)
    {
       /* Obtenemos el dia y comprobamos
       si su valor es valido */
       p = strtok(fecha, "-");
       if (p)
       {
          dia = atoi(p); if ((dia>=1)
          && (dia<=31))
          {
             /* Obtenemos el mes y comprobamos
             si su valor es valido */
             p = strtok(NULL, "-");
             if (p)
             {
                mes = atoi(p);
                if ((mes>=1) && (mes<=12))
                {
                   /* Obtenemos el año y comprobamos
                   si su valor es valido */
                   p = strtok(NULL, "-")
                   if (p)
                   {
                      anio = atoi(p);
                      if (anio>0)
                         resultado = 1;
                   }
                }
             }
          }
       }
    }
    return resultado;
}
```

La estrategia es la siguiente. En primer lugar comprueba que el tamaño de la cadena sea 10, que es lo que se corresponde con el formato "dd-mm-aaaa". Seguidamente, obtiene el día y comprueba que su valor sea correcto, en caso afirmativo, realiza la misma operación con el mes, tras el cual, si también es correcto, comprueba el año.
Haciendo estas comprobaciones de forma anidada conseguimos que si alguna de ellas falla, no se hagan más comprobaciones.
Como seguramente esté pensando, esta función puede ser mejorada de forma importante: teniendo en cuenta que no todos los meses tienen 31 días, los años bisiestos, etc. También puede intentar realizar una función similar para comprobar el formato hora: "hh:mm:ss".

Comprobar año bisiesto

La siguiente función permite saber si un año es bisiesto (con 366 días) o no (con 365 días), retornando el valor 1 o 0, respectivamente.
El año bisiesto requiere que las dos últimas cifras del número del año sean divisibles por cuatro, pero que, además, el año no termine en dos ceros; en este caso sólo se considerará bisiesto si es divisible por 400.
Algunos ejemplos son: el año 2044 es bisiesto porque 44 es divisible entre cuatro; el año 2001 no es bisiesto puesto que 01 no es divisible entre cuatro; el 1900 no es año bisiesto porque termina en dos ceros y al dividir este número entre 400 no da un número exacto; el 2000 es año bisiesto porque, aunque termina en dos ceros, sí es divisible entre 400.

```
int bisiesto (int anio)
{
    /* bisiesto: almacena 1 si es bisiesto, 0 en caso contrario */

    int es_bisiesto;

    /* dos_ultimas_cifras : almacena las dos ultimas cifras del año */ int
    dos_u1timasteifras;

    /* Partimos de la idea de que el año no es bisiesto */

    es_bisiesto=0 ;

    /* Obtenemos las dos ultimas cifras del año pasado por parametro. */
    dos_ultimas_cifras = anio % 100;

    /* Si se cumplen las condiciones para ser año bisiesto se asigna a la variable
    "es_bisiesto" el valor 1.*/
    if (dos_ultimas__cifras != 0)
    {
       if (dos_ultimas_cifras%4==0)
          es_bisiesto = 1;

    }
```

```
    else
    {
       if (anio%400 == 0)
           es_bisiesto = 1;
    }

    /* Retorna el valor de la variable "es_bisiesto". */
    return es_bisiesto;
}
```

A continuación, mostramos un ejemplo de uso de la función:

```
int main(void)
{
    /* resultado: almacena el resultado de la función "bisiesto" */

    int resultado;

    /* x: almacena el numero del año dado por el usuario */

    int x;

    /* Pedimos un año al usuairo y lo almacenamos en la variable "x". */

    printf("\n Escriba el año a comprobar si es bisiesto: ");

    scanf("%d", Sx);

    /* Llamamos a la función "bisiesto" pasándole "x" por parametro. El resultado
    lo almacenamos en la variable "resultado". */ resultado = bisiesto(x);

    /* Según el valor de "resultado" mostramos un mensaje u otro indicando si el
    año es bisiesto. */ if (resultado==1)
       printf("\n Es bisiesto."); else
       printf("\n No es bisiesto.");

    /* Hacemos una pausa hasta que el usuario pulse Intro */
    fflush(stdin);
    printf("\n\nPulse Intro para finalizar...");
    getchar();
}
```

Comprobar DNI

En España todas las personas tienen asignado un número de identificación único: el número del DNI (Documento Nacional de Identidad). Dicho número está formado por un número de varias cifras seguido de una letra de control, es decir, según los números habrá una letra u otra. Esto es un sencillo sistema de control que permite comprobar que el número es válido. La letra correspondiente a un número de DNI se calcula mediante el siguiente sistema. Se obtiene el resto de dividir el número de DNI entre 23.

El número resultante nos indica la posición de la letra, correspondiente a ese DNI, en la siguiente cadena: "TRWAGMYFPDXBNJZSQVHLCKET".
Se debe tener en cuenta que el resto 0 se corresponde con la T, el resto 1 con la 'R', el resto 2 con la letra 'W', y así sucesivamente. La siguiente función retorna la letra correspondiente al número de DNI pasado por parámetro.

```
char letra_dni (long int numero)
{
    /* resto: almacena el resto de DNI/23 */

    int resto;

    /* letra: almacena la letra del DNI */

    char letra;

    /* cadena: almacena la cadena de letras */

    char cadena[25];

    /* Almacenamos la cadena de letras de DNI */
    strcpy(cadena, "TRWAGMYFPDXBNJZSQVHLCKET");

    /* Calculamos la letra y la retornamos */
    resto = numero % 23;
    letra = cadena[resto];
    return letra;
}
```

El siguiente código muestra un ejemplo de uso de la función anterior. Se solicita al usuario un número de DNI y letra de DNI. Posteriormente, se comprueba si la letra dada por el usuario es la misma que la obtenida con la función letra_dni a partir del número. Si son iguales se muestra un mensaje y finaliza el programa, en caso contrario, se muestra un mensaje de error y se vuelven a pedir los datos al usuario.

```
int main (void)
{
    /* dni: almacena el numero de DNI */

    long int dni;

    /* letra: almacena la letra del DNI */

    char letra;

    do
    {
       /* Pedimos al usuario el numero de DNI */
       printf("\n\nEscribe tu numero de DNI: ");
       scanf("%lu", &dni);

       fflush(stdin);

       /* Pedimos al usuario el numero de DNI */
       printf("Escribe la letra de tu DNI: ");
       scanf("%c", sletra);
       fflush(stdin);
```

```
        /* Comprobamos si la letra es válida */
        if (letra == letra_dni(dni))
        {
           printf("\nEl DNI es correcto.");
        }
        else
        {
           printf("\nEl DNI no es valido.");
           printf("\nLa letra correspondiente deberia ser: %c", letra_dni(dni)) ;
        }
    }while(letra != letra_dni(dni));

    /* Hacemos una pausa hasta que el usuario pulse Intro */
    fflush(stdin);
    printf("\n\nPulse Intro para finalizar...");
    getchar();
}
```

Resumen

Las funciones y algoritmos que hemos presentado y explicado en este capítulo nos permiten comprobar la validez de determinados datos, sin embargo, podemos crear otras muchas funciones según nuestras necesidades. La idea principal con la que debemos quedarnos es que es imprescindible controlar todos los datos que el usuario introduce en el programa, en ningún momento debemos suponer que el usuario va a escribir exactamente lo que esperamos, o que nunca se va a equivocar. En caso de no controlar estos aspectos el programa fallará con total seguridad y, obviamente, no deseamos que esto ocurra bajo ningún concepto. Los ejemplos que hemos analizado son muy sencillos, pero podemos complicarlos o usarlos en programas más complejos.

Apéndice A

Bibliotecas estándar de C

Bibliotecas estándar de C

A continuación, describiremos algunas de las bibliotecas y funciones de C. Tenga en cuenta que los compiladores suelen añadir más bibliotecas y funciones, por lo que hemos incluido algunas que pertenecen a éstos. Debería comprobar de cuáles dispone su compilador.
Muchas de las funciones han sido explicadas ya en este libro. En cualquier caso, puede obtener el prototipo de cada función en la ayuda de su compilador, que también suele incluir una descripción amplia y ejemplos de su uso. Recuerde que para utilizar una función de una biblioteca debe incluir en el programa el archivo cabecera, que tiene extensión ".h". Así, si la biblioteca que desea incluir es `stdio`, debe incluir el archivo cabecera `stdio.h`:

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

Funciones de Entrada/Salida

En la biblioteca conio encontramos una gran variedad de funciones: para leer caracteres del teclado, manipular el cursor o borrar la pantalla, entre otras. En la tabla A.l se describen algunas de estas funciones.

Tabla A.1. Funciones de la biblioteca conio.

Función	Descripción
clrscr	Limpia la pantalla.
cprintf	Escribe una cadena formateada en la pantalla.
cscanf	Lee una entrada formateada del teclado.
getch	Lee un carácter del teclado sin eco en pantalla.
getche	Lee un carácter del teclado con eco en pantalla.
gotoxy	Coloca el cursor en una posición de la pantalla.
kbhit	Comprueba si hay alguna tecla en el *buffer* del teclado.
textbackg round	Establece el color del fondo.
textcolor	Establece el color del texto.
textmode	Cambia el modo de pantalla en modo texto.
wherex	Obtiene la posición horizontal del cursor en la pantalla.

Función	Descripción
wherey	Obtiene la posición vertical del cursor en la pantalla. Define una
window	ventana en modo texto.

Funciones de caracteres

La mayoría de las funciones pertenecientes a la biblioteca ctype, del estándar ANSI, se encargan de comprobar si un carácter es de puntuación, es un espacio en blanco, es alfanumérico, es numérico, etc. También hay funciones para convertir caracteres a mayúsculas o minúsculas.
En la tabla A.2 se describen algunas de estas funciones.

Tabla A.2. Funciones de ia biblioteca ctype.

Función	Descripción
isalnum	Comprobación de carácter alfanumérico.
isalpha	Comprobación de carácter alfabético.
isascii	Comprobación de carácter ASCII.
iscntrl	Comprobación de carácter de control.
isdigit	Comprobación de dígito decimal (0-9). Comprobación de
¡slower	carácter en minúsculas.
ispunct	Comprobación de carácter de puntuación.
isspace	Comprobación de carácter de espacio en blanco.
isupper	Comprobación de carácter en mayúsculas.
to lower	Transforma un carácter a minúsculas si estaba en mayúsculas.
toupper	Transforma un carácter a mayúsculas si estaba en minúsculas.

Funciones matemáticas

Las funciones de la biblioteca math, del estándar ANSI, nos permiten realizar una gran cantidad de operaciones matemáticas.
En la tabla A.3 se describen algunas de estas funciones.

Tabla A.3. Funciones de la biblioteca math.

Function	Descripción
abs	Valor absoluto de un número.
acos	Arco coseno.
asin	Arco seno.
atari	Arco tangente.
cos	Coseno.
cosh	Coseno hiperbólico.
exp	Valor exponencial.
tabs	Valor absoluto.
fmod	Módulo de un complejo.
hypot	Hipotenusa.
Idexp	Valor multiplicado por 2 elevado a un exponente.
log	Logaritmo neperiano.
log 10	Logaritmo en base 10.
pow	Valor elevado a otro valor.
pow10	10 elevado a un valor.
sin	Seno.
sinh	Seno hiperbólico.
sqrt	Raíz cuadrada.
tan	Tangente.
tanh	Tangente hiperbólica.

Funciones de la entrada/salida estándar

En la biblioteca stdio, del estándar ANSI, encontramos funciones para manejar la entrada/salida estándar. Con ellas podremos mostrar datos en la pantalla, leer del teclado y realizar operaciones con ficheros, entre otras cosas. En la tabla A.4 se describen algunas de estas funciones.

Tabla A.4. Funciones de la biblioteca stdio.

Función	Descripción
tclose	Cierra un fichero.
fcloseall	Cierra todos los ficheros abiertos.
fdopen	Abre un fichero asociado a un manejador.
feof	Comprueba si se ha llegado al final del fichero.
terror	Detecta si ha ocurrido un error en el fichero.
fflush	Vacía un canal.
fgetc	Lee un carácter de un fichero.
fgets	Lee una cadena de caracteres de un fichero.
topen	Abre un fichero asociado a un puntero.
fprintf	Escribe una cadena formateada en un fichero.
fpute	Escribe un carácter en un fichero.
fputs	Escribe una cadena de caracteres en un fichero.
tread	Lee datos de un fichero.
fscanf	Lee una serie de campos de una vez de un fichero.
fseek	Posiciona el puntero en un lugar del fichero.
ftell	Obtiene la posición del puntero del fichero.
fwrite	Escribe datos en un fichero.
getchar	Lee un carácter del teclado.
gets	Lee una cadena del teclado.
printf	Escribe una cadena con formato en pantalla.
putehar	Escribe un carácter en pantalla.
puts	Escribe una cadena de caracteres en pantalla.
remove	Borra un fichero.
rename	Renombra un fichero.
rewind	Sitúa el puntero de un fichero al principio.
scant	Lee una serie de campos de una vez del teclado.

Funciones de la biblioteca estándar

En la biblioteca stdlib, del estándar ANSI, encontramos una gran variedad de funciones, teniendo la mayoría como propósito hacer conversiones. También hay funciones para realizar divisiones enteras, hacer búsquedas, reservar memoria y obtener números aleatorios, entre otras.
En la tabla A.5 se describen algunas de estas funciones.

Tabla A.5. Funciones de la biblioteca stdlib.

Función	Descripción
atof	Convierte una cadena en un `f loat`.
atoi	Convierte una cadena en un `int`.
atol	Convierte una cadena en un `long`.
bsearch	Realiza un búsqueda binarla en un *array.*
calloc	Asigna memoria principal.
div	División entera.
ectv	Convierte un `float` en una cadena.
getenv	Consigue una cadena del entorno. Convierte un entero en una
itoa	cadena.
labs	Obtiene el valor absoluto de un `long`.
ldiv	División entera de dos enteros de tipo `long`.
lfind	Realiza una búsqueda lineal.
lsearch	Realiza una búsqueda lineal.
ltoa	Convierte un valor `long` en una cadena.
malloc	Asigna memoria.
qsort	Realiza una ordenación rápida. Genera números aleatorios.
rand	
randomize	Inicializa el generador de números aleatorios.
realloc	Reasigna memoria principal.
strtod	Convierte una cadena en un `double`.
strtol	Convierte una cadena en un `long`.

Función	Descripción
stroul	Convierte una cadena en un `long` sin signo. Recurre al archivo del DOS command.com.
system	
ultoa	Convierte un valor `long` sin signo en una cadena.

Funciones de cadenas de caracteres y memoria

Las funciones de la biblioteca string, del estándar ANSI, permiten operar con cadenas: copiar una cadena en otra, concatenar, comparar, obtener su tamaño, etc. También permiten operar con posiciones de memoria.
En la tabla A.6 se describen algunas de estas funciones.

Tabla A.6. Funciones de la biblioteca string.

Función	Descripción
memcmp	Compara n caracteres entre el *buffer* 1 y el *buffer* 2.
memcpy	Copla n caracteres desde el buffer fuente al buffer destino.
strcat	Añade una cadena de caracteres a otra.
strchr	Localiza la primera aparición de un carácter en una cadena de caracteres.
strcmp	Compara dos cadenas de caracteres *(case sensitive).*
strcmpi	Compara dos cadenas de caracteres (no *case sensitive).*
strcpy	Copia una cadena de caracteres en otra.
strcspn	Localiza la primera aparición de un carácter, o de un conjunto de caracteres, dado en una cadena.
strlen	Longitud de la cadena de caracteres.
strlwr	Convierte la cadena de caracteres a minúsculas.
strrchr	Localiza la última aparición de un carácter en una cadena.
strrev	Invierte los caracteres de una cadena.
strset	Sustituye todos los caracteres de una cadena por un carácter determinado.

Función	Descripción
strstr	Localiza una cadena dentro de otra cadena,
strtok	Localiza un símbolo dentro de una cadena,
strupr	Transforma una cadena a mayúsculas.

Funciones de tiempo

Las funciones de la biblioteca `time,` del estándar ANSI, permiten obtener la hora `y` fecha del sistema, así como modificarlas.
En la tabla A.7 se describen algunas de estas funciones.

Tabla A.7. Funciones de la biblioteca time.

Función	Descripción
asctime	Transforma la fecha y la hora a una cadena de caracteres ASCII.
dock	Determina la hora del reloj del microprocesador.
ctime	Transforma la hora y la fecha en una cadena de caracteres.
difftime	Calcula la diferencia entre dos horas.
gmtime	Transforma la fecha y la hora a GMT.
stime	Establece la fecha y la hora para el sistema.
striti me	Permite formatear la salida de la fecha y la hora.
time	Obtiene la hora actual del sistema.

Apéndice B

Bibliotecas estándar de C++

Bibliotecas estándar de C++

En C++ no es necesario añadir la extensión ".h" al archivo cabecera, pero si esto no funcionase se la podemos añadir. Así, si la biblioteca que desea incluir es iostream, debe incluir el archivo cabecera de la forma:

```
#include <iostream>
```

o:

```
#include <iostream.h>
```

Además, encontramos para cada biblioteca <B. h> de C una equivalente en C++ que comienza por c: <cB>.
A continuación, describiremos algunas de las bibliotecas de C++ agrupadas por el tema que tratan. Estamos en C++, así que estas bibliotecas suelen contener plantillas o clases para crear objetos como listas y colas.

Contenedores

Muchas veces necesitamos crear conjuntos de datos o almacenarlos en listas o colas. C++ nos proporciona una serie de bibliotecas para crear listas, colas o *arrays* a partir de unas plantillas. Así que no es necesario que implemente una cola o una pila, créela a partir de las bibliotecas de C++ (véase la tabla B.l).

Tabla B.1. Contenedores.

Biblioteca	Descripción
vector	Array unidimensional de T.
list	Lista doblemente enlazada de T.
deque	Cola de doble extremo de T.
queue	Cola de T.
stack	Pila de T.
set	Conjunto de T.

Cadenas

En la tabla B.2 se describen algunas de las bibliotecas para poder trabajar con cadenas.

Tabla B.2. Cadenas

Biblioteca	Descripción
string	Cadena de T.
cctype	Clasificación de caracteres.
Cstring	Equivalente a `<string.h>` de C.
cstdlib	Equivalente a `<stdlib.h>` de C.

Entrada/salida

En la tabla B.3 se describen algunas de las bibliotecas que permiten manejar la entrada/salida en C++.

Tabla B.3. Entrada/salida.

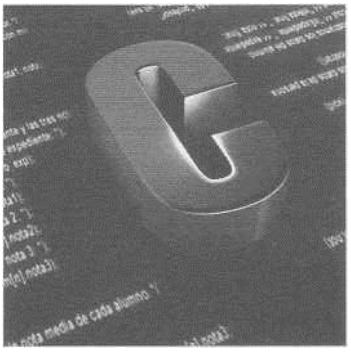
Biblioteca	Descripción
iostream	Objetos y operaciones de entrada/salida estándar. Plantilla
istream	de flujo de entrada.
ostream	Plantilla de flujo de salida.
cstdio	Equivalente a `<stdio. h>` de C.

Números

En la tabla B.4 se describen algunas de las funciones para realizar operaciones matemáticas.

Tabla B.4. Números.

Biblioteca	Descripción
complex	
valarray	Números complejos y operaciones. Vectores numéricos y
numeric	operaciones. Operaciones numéricas generalizadas.
cmath	Funciones matemáticas estándar.



Apéndice C

El lenguaje C/C++ en Internet

El lenguaje C/C++ en Internet

En Internet podemos encontrar mucha información adicional sobre el lenguaje de programación C y C++. A continuación hacemos referencia a dos de los sitios más interesantes que podemos encontrar: los de los creadores de C y C++. Tenga en cuenta que el continuo cambio típico de Internet puede ocasionar la desaparición de alguno de ellos.

Dennis Ritchie

En http : //www. es . bell- labs . com/who/dmr/ se aloja el sitio de Dennis Ritchie, quien desarrolló el lenguaje C. Actualmente trabaja en el Computing Sciences Research Center de los Bell Labs. En su página principal mantiene una foto suya. Entre sus enlaces podemos llegar a su biografía o a información sobre sus trabajos.

Bjarne Stroustrup

En http: //www. research. att. com/~bs/homepage . html encontramos el sitio del creador del lenguaje C++, Bjarne Stroustrup. En él hay información sobre la biografía de este profesor de informática en Texas A&M University, entrevistas y publicaciones, entre otros muchos datos. También es probable que vea alguna foto suya en la página principal.

Apéndice D

Solución de los ejercicios

Solución de los ejercicios

En este apéndice veremos las soluciones a los ejercicios propuestos a lo largo del libro, que se presentan agrupadas por capítulos.
Algunas soluciones se realizan en C, otras en C++ y otras en ambos, de esta manera podemos familiarizarnos con los dos lenguajes. En cualquier caso, es muy importante tener presente que las soluciones a los ejercicios que consisten en realizar código son simplemente una posible solución: los identificadores pueden cambiarse, pueden utilizarse otras sentencias de control de flujo, pueden incluirse más mensajes de información al usuario, etc. Algunas soluciones a los ejercicios se acompañan de notas que se deben tener en consideración.

Variables y constantes

Ejercicio 1

Enunciado

Supongamos que necesitamos dos variables para almacenar la posición del ratón en la coordenada X y en la coordenada Y de la pantalla. ¿Qué identificadores emplearía para cada una de estas variables?

Solución

Podríamos utilizar como identificadores para dos variables que almacenan la posición del ratón en las coordenadas (X, Y) de la pantalla:

- posicion_x
- posicion_y

Ambos identificadores son significativos y rápidamente nos indican qué almacenan. Haber elegido como identificadores x e y no habría sido una buena idea, pues son nombres demasiados generales.

Nota:

No olvide que es muy iimportante utilizar identificadores significativos y que nos ayuden a saber rápidamente el propósito de una

variable, constante n otro elemento. Además, nos permite realizar códigos más legibles y fáciles de comprender.

Ejercicio 2

Enunciado

Cuáles de los siguientes identificadores son incorrectos:

- num_camiones
- 2_registro
- _asm
- _código
- bandera7
- num_dia_semana
- #codigo
- año
- _pulsacion
- cod soft

Solución

Los identificadores incorrectos son:

- 2_registro, porque empieza con un número (2).
- _código, porque tiene un carácter no válido, la tilde ('ó').
- #codigo, porque tiene un carácter no válido, la almohadilla ('#').
- año, porque tiene un carácter no válido, la eñe ('ñ').
- cod soft, porque tiene un carácter no válido, el espacio (' ').

Nota:

Recuerde que los identificadores sólo pueden tener los caracteres: letras mayúsculas (de la 'A' a la 'Z '), letras minúsculas (de la 'a' a la 'z'), guión bajo (_) y números (del 0 al 9). Pero sólo pueden empezar con una letra mayúscula, minúscula o un guión bajo, no pueden empezar con un número. Además, no se permite el uso de la 'n ó 'Ñ', ni de tildes, ni de espacios.

Ejercicio 3

Enunciado

Si tenemos una variable cuyo identificador es "numero_reg" y su valor inicial es 10, ¿cuál será el valor de la variable "numero_Reg"?

Solución

No podemos saber cuál es el valor de la variable "numero_Reg", ya que no es igual que la variable "numero_reg". Debemos recordar que C/C++ es *case sensitive,* es decir, distinguen entre mayúsculas y minúsculas, considerando ambas variables distintas.

Nota:

Tenga siempre presente que C y C++ *son* case sensitive. *No abuse de utilizar identificadores que mezclen minúsculas con mayúsculas, pues es fácil confundirse, teniendo que realizar más revisiones para comprobar si tal letra estaba en mayúsculas o en minúsculas. Se recomienda utilizar siempre identificadores en minúsculas, a excepción de las constantes, que se escriben completamente en mayúsculas.*

Ejercicio 4

Enunciado

Cómo se declararía:

- Una variable que permita almacenar caracteres.
- Una constante cuyo valor sea tu nombre.
- Una variable que permita almacenar la edad de una persona.
- Una variable que permita almacenar un código numérico de 7 dígitos.
- Una constante cuyo valor es el doble del número pi.

Solución

Veamos cómo se podría declarar:

- Una variable que permita almacenar caracteres:
  ```
  char carácter;
  ```
- Una constante cuyo valor sea tu nombre:
  ```
  #define NOMBRE "Alberto"
  ```
- Una variable que permita almacenar la edad de una persona:
  ```
  int edad;
  ```
- Una variable que permita almacenar un código numérico de 7 dígitos:
  ```
  long int codigo;
  ```
- Una constante cuyo valor es el doble del número pi:
  ```
  const float DOBLE PI = 6.28;
  ```

Operadores

Ejercicio 1

Enunciado

Indique si son equivalentes o no las parejas de expresiones que aparecen en la tabla D.l.

Tabla D.1. ¿Expresiones equivalentes?

Expresión A	Expresión B	¿Son equivalentes?
++i;	i += 1;	Sí /No
b = 1 + b;	++b;	Sí/No
c = c + 1;	++c;	Sí /No
k++;	k = k- 1;	Sí/No

Solución

En la tabla D.2 comprobamos si son equivalentes o no las parejas de expresiones que se muestran.

Tabla D.2. Solución.

Expresión A	Expresión B	¿Son equivalentes?
++i;	i+= 1 ;	Sí
b = 1 + b;	++b;	Sí
c = c + 1;	++c;	Sí
k++;	k = k -1;	No

Nota:

No olvide que no es lo mismo utilizar los operadores de incremento y decremento delante o detrás de una variable cuando se utilizan en una expresión. Pero si se utilizan solos en una línea el efecto es igual.

Ejercicio 2

Enunciado

¿Cuál sería el valor de la variable contador después de ejecutarse el siguiente código?

```
int contador = 0;

contador++;
contador = contador + 1;
contador = 1 + contador;
contador==;
contador = 7 - contador;
contador--;
contador = contador - 4;
```

Solución

El valor de la variable contador tras ejecutarse cada línea de código se muestra en la tabla D.3.

Tabla D.3. Valor de la variable contador.

Línea de código	Valor de contador
int contador = 0;	contador=0;
contador++;	contador=1;

Línea de código	Valor de contador
contador = contador + 1;	contador=2;
contador = 1 + contador;	contador=3;
contador--;	contador=2;
contador = 7 - contador;	contador=5;
contador-;	contador=4;
contador = contador - 4;	contador=0;

Tras todas estas líneas de código, finalmente, la variable **contador** vale 0.

Nota:

En el ejercicio anterior se comprueba línea a línea qué ocurre, esto es una traza. Las trazas le ayudarán a examinar el comportamiento de su programa y a detectar posibles errores. Además, haciendo trazas podrá aprender cómo se comportan los programas y las distintas sentencias.

Ejercicio 3

Enunciado

¿Cuál sería el resultado de la siguiente expresión?

```
! ( (7<=7) || (33 > 33) )
```

Solución

El resultado de la expresión:

```
!((7<=7)  || (33>33))
```

es falso, o lo que es lo mismo 0. A continuación vemos la resolución del ejercicio por partes.

- (7<=7) es verdadero, porque 7 sí es menor o igual que 7.
- (33>33) es falso, porque 33 no es mayor que 33.

- (7<=7) || (33>33) es verdadero, puesto que verdadero OR falso es verdadero.
- !((7<=7) || (33>33)) es falso, porque la negación de verdadero es falso.

Nota:

Cuando desee hacer condiciones con subcondiciones es preferible abusar de los paréntesis. Los paréntesis le ayudan a determinar claramente la preferencia de ejecución y qué operadores afectan a cada expresión.

Punteros y referencias

Ejercicio 1

Enunciado

Realice un programa que almacene en una variable puntero la dirección de memoria de una variable de tipo char inicializada con el valor 'X'. Represente el estado de la memoria.

Solución

```
int main (void)
{
   char variable = 'X';
   char *mi_puntero;

   /* Asignamos a la variable "mi_puntero" la dirección de
   memoria en la que se encuentra la variable llamada
   "variable". */
   mi_puntero = &variable;
}
```

En la figura D.l se muestra la representación del estado de la memoria.

Nota:

No olvide que una variable puntero es una simple variable más, pero que el dato que almacena es un número que se corresponde con una dirección de memoria.

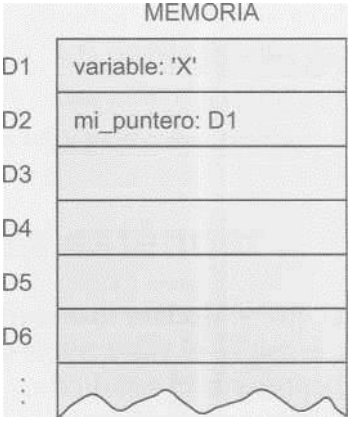


Figura D.1. Estado de la memoria del ejercicio 1.

Ejercicio 2

Enunciado

Si `mi_puntero` es una variable puntero de tipo `int`, ¿cuál de las siguiente afirmaciones es falsa?

- Mediante `&mi__puntero` obtenemos la dirección de memoria de la variable `mi puntero.`
- Mediante `*m±_puntero` obtenemos el contenido que hay en la dirección de memoria que almacena la variable `mi_puntero.`
- Mediante `mi_puntero` obtenemos el contenido que hay en la dirección de memoria que almacena la variable `mi_puntero.`

Solución

La tercera afirmación ("Mediante `mi_puntero` obtenemos el contenido que hay en la dirección de memoria que almacena la variable `mi_puntero"`) es falsa, porque mediante `mi_puntero` obtenemos el valor que almacena esta variable, que es una dirección de memoria.

Nota:

*Recuerde que una variable puntero almacena una dirección de memoria, que el operador * delante de una variable puntero obtiene el*

contenido de dicha dirección de memoria, y que el operador & delante de cualquier variable obtiene la dirección de dicha variable en memoria.

Ejercicio 3

Enunciado

Realice un programa en el que se declare una variable de tipo entero y, haciendo uso de referencias, le asigne el valor cero a esta variable e incremente su valor en una unidad. Represente el estado de la memoria.

Solución

```
int main (void)

{
   int edad;
   int &referencia_edad = edad;

   /* Asignamos a "referencia_edad" ei valor cero.
   El resultado es que realmente asignamos a la variable
   "edad" el valor cero. */

   referencia_edad = 0;

   /* Mediante "referencia_edad" incrementamos el valor
   de la variable "edad" en una unidad. */
   referencia_edad = referencia_edad + 1;
}
```

En la figura D.2 se muestra la representación del estado de la memoria.

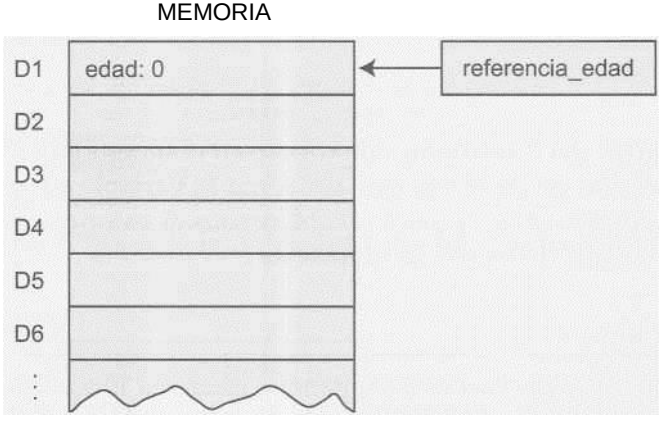


Figura D.2. Estado de la memoria del ejercicio 2.

Nota:

Aunque las referencias son mucho más fáciles de usar que los punteros recuerde que son exclusivas de C++. *No obstante, se pueden utilizar referencias y punteros a la vez.*

Entrada y salida estándar

Ejercicio 1

Enunciado

Realice un programa que pida una letra al usuario y muestre el carácter siguiente en la tabla ASCII.

Solución

```
#include <stdio.h>

int main (void)
{
    char carácter;

    /* Pedimos un carácter al usuario y lo almacenamos en la variable carácter. */
    printf ("Escriba un carácter y pulse Intro: ");

    carácter = getchar();

    /* Incrementamos el valor de carácter en una unidad y mostramos su valor.
    */ carácter = carácter + 1;
    printf ("\nEl carácter siguiente al introducido es %c ", carácter);

    /* Hacemos una pausa hasta que el usuario pulse Intro */
    fflush(stdin);
    printf("\n\nPulse Intro para finalizar...");
    getchar();
}
```

Ejercicio 2

Enunciado

Escriba un programa que pida un número al usuario y después muestre su cuadrado.

Solución

```
#include <iostream.h>
int main (void)
{
   int numero;

   /* Pedimos un numero al usuario y lo almacenamos
   en la variable "numero". */
   cout << "Introduzca un numero: ";
   cin >> numero;

   /* Mostramos el cuadrado del numero. */
   cout << " El cuadrado de "<< numero <<" es "<< (numero*numero);

   /* Hacemos una pausa hasta que el usuario pulse Intro */
   fflush(stdin);
   cout << "\nPulse Intro para finalizar...";
   cin.get ();
}
```

Ejercicio 3

Enunciado

Escriba un programa que pida dos números al usuario y muestre el resto de dividir uno entre el otro.

Solución

```
#include <stdio.h> int main (void)
{
   int dividendo, divisor; int resto;

   /* Pedimos al usuario el dividendo. */

   printf ("Introduzca el dividendo: ") ;

   scanf("%d", ¿¿dividendo);

   /* Pedimos al usuario el divisor. */

   printf ("Introduzca el divisor: ");

   scanf ("%d", ¿¿divisor);

   /* Calculamos el resto de la división. */

   resto = dividendo % divisor;

   /* Mostramos el resto de dividir el dividendo entre el divisor */
   printf ("\nEl resto de dividir %d entre %d es %d", dividendo, divisor, resto);
}
```

Control del flujo

Ejercicio 1

Enunciado

Realice un programa que pida un número al usuario y, a continuación, muestre un mensaje en pantalla que indique si dicho número es mayor que 10 o es menor que 10.

Solución

```
#include <stdio.h>

int main (void)
{
    int numero;

    /* Pedimos un numero al usuario. */
    printf("Escriba un numero: ");
    scanf("%d", &numero);

    /* Comprobamos si el numero es mayor o
    menor que 10 y mostramos un mensaje
    indicándolo. */
    if (numero < 10)
    {
       printf("\n%d es menor que 10", numero);
    }
    else
    {
       printf("\n%d es mayor que 10", numero);
    }

    /* Hacemos una pausa hasta que el usuario pulse Intro */
    fflush(stdin);
    printf("\n\nPulse Intro para finalizar...");
    getchar();
}
```

Ejercicio 2

Enunciado

Realice un programa que, mientras el usuario lo desee, pida números e indique mediante un mensaje en pantalla si es par o impar.

Solución

```
#include
<stdio.h>

int main (void)
{
   int numero; char
   continuar;

   /* El bucle do-while se repite mientras el
   usuario lo desee, es decir, mientras la
   variable "continuar" valga 's' o 'S'. */

   do
   {
      /* Pedimos un numero al usuario. */
      printf("Escriba un numero: ");
      scanf("%d", &numero);

      fflush(stdin);

      /* Comprobamos si el numero es par o
      impar y mostramos un mensaje
      indicándolo. */
      if (numero%2 == 0)
      {
         printf("\n%d es par.", numero);
      }
      else
      {
         printf("\n%d es impar.", numero);
      }


      /* Preguntamos al usuario si desea continuar. */

      printf ("\nDesea escribir otro numero? (S/N): ");
      continuar = getchar();

   }while ((continuar==¹s')   || (continuar=='S'));

   /* Hacemos una pausa hasta que el usuario pulse Intro */
   fflush(stdin);
   printf("\n\nPulse Intro para finalizar..."); getchar ();
}
```

Nota:

Es importante mantener informado correctamente al usuario de todo lo que sucede en el programa, incluso si se produce un error. Esto permite que el usuario conozca qué ha hecho mal y pueda poner una solución. En caso contrario puede ocurrir que el usuario piense que él lo ha hecho todo bien y que es el programa el que funciona mal.

Ejercicio 3

Enunciado

Realice un programa que muestre en la pantalla los números del 1 al 100 y de diez en diez.

Solución

```
#include <stdio.h>

int main (void)
{
    int numero;

    /* Empezamos por el numero 1. */
    numero = 1;

    /* Mientras el numero sea menor o igual a 100
    mostramos el numero y lo incrementamos de 10 en 10
    unidades. */

    while (numero<=100)
    {
       printf ("%d ", numero);
       numero = numero + 10;
    }

    /* Hacemos una pausa hasta que el usuario pulse
    Intro */
    fflush(stdin);
    printf("\n\nPulse Intro para finalizar...");
    getchar();
}
```

Arrays Ejercicio 1 Enunciado

Realice un programa que almacene en un *array* unidimensional de 10 letras escritas por el usuario.
A continuación, se debe mostrar un mensaje en pantalla indicando cuántas vocales se escribieron.

Solución

```
#include<stdio.h>

int main(void)
{
   char letras[10]; int i ;
   int contador = 0;

   /* Pedimos al usuario 10 letras y las almacenamos en el array. */
   for (i=0; i<=9; i++)
   {
      printf("(%d/10) Escriba una letra: ", i+1) ;
      scanf("%c", &letras[i]);
      fflush(stdin);
   }

   /* Recorremos el array y comparamos cada celda con
   las vocales. Si el contenido de una celda es igual a una
   vocal aumentamos el valor de la variable "contador" */

   for (i=0; i<=9; i++)
   {
      if ( (letras[i]=='a')   || (letras[i]=='e') ||
                (letras[i]=='i') II (letras[i]=='o') ||
                (letras[i]=='u') |¡ (letras[i]=='A')
                                 ||
                (letras[i]=='E') II (letras[i]=='I') ||
                (letras[i]=='0') II (letras[i]=='U') )
      {
         contador = contador + 1;
      }

   }

   /* Mostramos el numero de vocales contadas. */

   printf("\nHa escrito %d vocales.", contador);

   /* Hacemos una pausa hasta que el usuario pulse Intro */
   fflush(stdin);
   printf("\n\nPulse Intro para finalizar...");
   getchar () ;
}
```

Ejercicio 2

Enunciado

Realice un programa que pida al usuario 5 números enteros y los almacene en un *array* unidimensional. Posteriormente, se debe mostrar un mensaje en pantalla indicando la media de esos números, el mayor de ellos y el menor.

Solución

```
#include<stdio.h>

int main(void)
{
   char números[5];
   int i, media, mayor, menor;
   int suma = 0;

   /* Pedimos al usuario 5 números y los
   almacenamos en el array. */
   for (i=0; i<=4; i++)
   {
      printf("(%d/5) Escriba un numero: ", i+1);
      scanf("%d", &numeros[i]);
      fflush(stdin);
   }

   /* Calculamos la media de los 5 números. Para ello, sumamos
   los cinco números en la variable "suma". Después dividimos
   la suma entre cinco y ya tenemos la media. */
   for (i=0; i<=4; i++)
   {
      suma = suma + números[i];
   }
   media = suma / 5;

   /* Obtenemos el numero mayor y el menor.
   Partimos de la idea de que el primer numero es el mayor y
   el menor, inicialmente. */ menor = números[0];

   mayor = números[0];

   /* A continuación, comparamos el resto de los valores
   almacenados con el mayor y el menor encontrado hasta el
   momento. */
   for (i=l; i<=4; i++)
   {
      if (números[i] < menor)
      {
         menor = números[i];
      }

      if (números[i] > mayor)
      {
         mayor = números[i];
      }

   }

   /* Mostramos la media, el numero mayor y el menor. */
   printf("\nMedia: %d.", media);
```

```
    printf("\nNumero mayor: %d.", mayor);
    printf("\nNumero menor: %d.", menor);

    /* Hacemos una pausa hasta que el usuario pulse Intro */
    fflush(stdin);
    printf("\n\nPulse Intro para finalizar...");
    getchar();
}
```

Ejercicio 3

Enunciado

Realice un programa que pida al usuario 6 números enteros y los almacene en un *array* unidimensional. Después, se deben mostrar los mismos números pero en orden inverso al que los escribió el usuario.

Solución

```
#include<stdio.h>

int main(void)
{
    char números[6]; int i;

   /* Pedimos al usuario 6 números y los almacenamos en el
   array. */
   for (i=0; i<=5; i++)
   {
      printf("(%d/6) Escriba un numero: ", i+1);
      scanf("%d", &numeros[i]);
      fflush(stdin);
   }

   /* Mostramos los 6 números en orden inverso, recorriendo
   el vector desde el final hasta el principio. */
   printf("\nLos 6 números en orden inverso: ");
   for (i=5; i>=0; i--)
   {
      printf("%d ", números[i]);
   }

   /* Hacemos una pausa hasta que el usuario pulse Intro */
   fflush(stdin);
   printf("\n\nPulse Intro para finalizar...");
   getchar();
}
```

Cadenas

Ejercicio 1

Enunciado

Realice un programa que pida primero una frase al usuario y luego un carácter. El programa debe mostrar en la pantalla el número de veces que aparece el carácter indicado por el usuario en la cadena.

Solución

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

int main (void)
{
    char frase[50];
    char carácter;
    int i;
    int veces = 0;

    /* Pedimos una frase al usuario. */
    printf("Escriba una frase; ");
    gets(frase);

    /* Pedimos un carácter al usuario. */
    printf("Escriba un carácter: ");
    scanf("%c", Scaracter);

    /* Buscamos en la frase el carácter, y cada vez que lo
    encontramos aumentamos el contador "veces". */
    for (i=0; i<strlen(frase); i++)
    {
        if (frase[i]==caracter)
        {
            veces = veces + 1;
        }

    }

    /* Mostramos el numero de veces que aparece el carácter en la frase. */

    printf ("\nEl carácter '%c' aparece %d veces en la frase.", carácter, veces);

    /* Hacemos una pausa hasta que el usuario pulse Intro */
    fflush(stdin);
    printf("\n\nPulse Intro para finalizar...");
    getchar () ;
}
```

Ejercicio 2

Enunciado

Realice un programa que pida dos palabras al usuario y, a continuación, las vuelva a mostrar en orden alfabético.

Solución

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

int main (void)
{
   char
   palabral[20];
   char
   palabra2[20];

   /* Pedimos dos palabras al usuario.
   */ printf("Escriba una palabra: ");
   gets(palabral);
   printf("Escriba otra palabra: ");
   gets(palabra2);

   /* Mostramos las palabras ordenadas
   alfabeticamente. */
   if (strcmp(palabra1, palabra2)<0)
   {
      printf("%s\n",
      palabra1);
      printf("%s\n",
      palabra2);
   }
   else
   {
      printf("%s\n",
      palabra2);
      printf("%s\n",
      palabra1);
   }
```

Ejercicio 3

Enunciado

Realice un programa que pida una frase al usuario. Luego deben mostrarse los caracteres de esa cadena en orden inverso. Por ejemplo, si el usuario escribió "Hola mundo", el programa debe mostrar posteriormente "odnum aloH".

Solución

```
#include<stdio.h>
#include<string.h> int main (void)
{
    char frase[50];
    int i ;

    /* Pedimos una frase al usuario. */
    printf("Escriba una frase: ");
    gets(frase);

    /* Mostramos la frase invertida recorriendo el
    array desde el final hasta el principio. */
    printf("XnFrase invertida: ");
    for (i=strlen(frase)-1; i>=0; i--)
    {
       printf("%c", frase[i]);
    }

    /* Hacemos una pausa hasta que el usuario pulse Intro */
    fflush(stdin);
    printf("\n\nPulse Intro para finalizar...");
    getchar();
}
```

Estructuras Ejercicio 1

Enunciado

Escriba la definición y dibuje las siguientes estructuras:

- Una estructura que permita almacenar una fecha. Tendrá tres campos: día, mes y año.
- Una estructura que sirva para catalogar los libros de una biblioteca. Tendrá los siguientes campos: título, autor, tema, ISBN y número de páginas. El ISBN es un número de identificación de 10 dígitos.

Solución a la estructura de la fecha

```
struct fecha {
int dia;
int mes;
int anio;
};
```

En la siguiente figura D.3 se puede observar la representación de la estructura `fecha`.

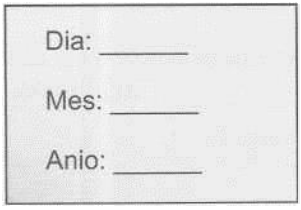


Figura D.3. Estructura fecha.

Solución a la estructura de los libros

```
struct datos_libro {
   char titulo[50];
   char autor[60];
   char tema[20];
   long int isbn; int
   paginas;
};
```

En la siguiente figura D.4 se puede observar la representación de la estructura datos libro.

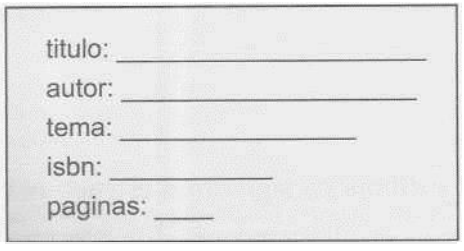


Figura D.4. Estructura datos_fecha.

Ejercicio 2

Enunciado

Realice un programa que almacene el nombre de seis jugadores de videojuegos y la puntuación máxima de cada uno. Posteriormente, debe mostrarse en pantalla el nombre del jugador con el mayor número de puntos y el del jugador con el menor número de puntos.

Solución

Antes de ver el código vamos a comentar para qué se usa cada variable:

- int i: se emplea para recorrer el *array*.
- int max_puntos: almacena la mayor puntuación.
- int min_puntos: almacena la menor puntuación.
- char max_jugador[50]: almacena el nombre del jugador con más puntos.
- char minjugador [50]: almacena el nombre del jugador con menos puntos.
- struct registro lista[6]: almacena las estructuras de los seis jugadores.

La solución sería la siguiente.

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

/* Definimos la estructura del array. */
struct registro { char jugador[50]; int
puntos;
};

int main (void)

{
   struct registro lista[6];
   int i;
   int max_puntos; int
   min_puntos; char max_j
   ugador[50]; char
   min_jugador[50];

   /* ----- Rellenamos la lista de seis jugadores ------ */

   /* Pedimos al usuario el nombre y la puntuación de seis
   jugadores y los almacenamos en el array. */
   for (i=0; i<=5; i++)
   {
      printf("\nEscriba el nombre del jugador %d: ", i+1) ;
      gets (lista [i] .jugador);
      printf("Escriba su puntuación de: ");
      scanf("%d", Slista[i].puntos);
      fflush(stdin);
   }


   /*  ---- Mostramos el nombre del jugador con mas
   puntos y el nombre del jugador con menos ------  */

   /* Tomamos la puntuación del primer jugador de la lista,
   y su nombre, como la maxima y la minima. */
```

```
    max_puntos=lista[0].puntos;
    strcpy(max_jugador, lista[0].jugador);
    min_puntos=lista[0].puntos;
    strcpy(min_jugador, lista[0].jugador);

    /* Recorresmos el array desde la siguiente celda. */
    for (i=1; i<=5; i++)
    {
       /* Comprobamos si el jugador actual
       tiene el máximo de puntos. */
       if (lista[i].puntos>max_puntos)
       {
          max_puntos=lista[i].puntos;
          strcpy (max_jugador, lista [i].jugador);
       }

       /* Comprobamos si el jugador actual
       tiene el minimo de puntos. */
       if (lista[i].puntos<min_puntos)
       (
          min_puntos=lista[i].puntos;
          strcpy(min_jugador, lista[i].jugador);
       }
    }
    /* Mostramos el nombre del jugador con mas puntos y el nombre
    del jugador con menos puntos. */

    printf ("\nJugador con mas puntos: %s", max_jugador) ;
    printf("\nJugador con menos puntos: %s", min_jugador);

    /* Hacemos una pausa hasta que el usuario pulse Intro */
    fflush(stdin);
    printf("\n\nPulse Intro para finalizar...");
    getchar();
}
```

Funciones

Ejercicio 1

Enunciado

Escriba un programa que utilice una función que pida una clave al usuario. Hasta que no acierte la clave (que por ejemplo es 123) la función no debe finalizar. En el programa principal, debe mostrarse un mensaje de saludo antes de la llamada a la función y otro mensaje de despedida después de la llamada a la función.

Solución

```
#include <stdio.h>

/* Definimos la clave, que es 123. */
#define PASSWORD 123

/* Declaramos la función que pide la clave. */

void control__de_acceso (void) ;

int main (void)
{
    /* Mostramos el mensaje de saludo. */
    printf("Bienvenido!\n");

    /* Llamamos a la función que pide la clave.*/
    control_de_acceso();

    /* Mostramos el mensaje de despedida. */
    printf("\nHasta pronto!");

    /* Hacemos una pausa hasta que el usuario pulse Intro */
    fflush(stdin);
    printf("\n\nPulse Intro para finalizar...");
    getchar();
}

void control_de_acceso(void)
{
    int clave;

    /* Pedimos la clave al usuario una y otra vez mientras no la acierte. */

    do
    {
       /* Pedimos la clave al usuario. */
       printf("\nEscriba la clave de acceso: ");
       scanf("%d", Sclave); fflush(stdin);

       /* Si la clave es incorrecta mostramos un mensaje. */
       if (clave!=PASSWORD)
       {
          printf("\nClave incorrecta!\n");
       }

    }while (clave!=PASSWORD);
}
```

Nota:

Recuerde que debe declarar las funciones que cree, pero que no hay que declarar la función principal o main.

Ejercicio 2

Enunciado

Realice un programa que utilice una función a la que se le pasa un número por parámetro. La función debe mostrar en la pantalla un mensaje indicando si el número es par o impar.

Solución

```
#include <stdio.h>

/* Declaramos la función par_impar. */

void par_impar(int numero);

int main (void)
{
    int mi_numero;

    /* Pedimos un numero al usuario sobre el que averiguar si es par
    o impar. */
    printf("Escribe un numero para saber si es par o impar: ");
    scanf("%d", &mi_numero);

    /* Llamamos a la función par_impar y le pasamos por parametro el
    numero dado por el usuario. */ par_impar(mi_numero);

    /* Hacemos una pausa hasta que el usuario pulse Intro */
    fflush(stdin);
    printf("\n\nPulse Intro para finalizar...");
    getchar();
}

void par_impar(int numero)

{
    /* Mostramos un mensaje en la pantalla indicando si el numero
    pasado por paramentro es par o impar. */
    if (numero%2==0)
       printf("El numero es par!"};
       else
       printf("El numero es impar!");
}
```

Nota:

No olvide que el paso de parámetros por valor no permite modificar el valor de una variable pasada por parámetro.

Ejercicio 3

Enunciado

Escriba una función que modifique el valor de una variable numérica pasada por parámetro. La función debe pedir un número al usuario y asignárselo a la variable.

Solución con punteros (válido para C/C++)

```
#include <stdio.h>

/* Declaramos la función asignar. */

void asignar(int *numero);

int main (void)
{
    int mi_numero;

    /* Llamamos a la función asignar, en la que el usuario da un
    valor a la variable pasada por parametro. */
    asignar(&mi_numero);

    /* Mostramos el valor actual de "mi_numero". */

    printf("\nEl valor de mi_numero es %d.", mi_numero);

    /* Hacemos una pausa hasta que el usuario pulse Intro */
    fflush(stdin);
    printf("\n\nPulse Intro para finalizar...");
    getchar();
}

void asignar(int 'numero)
{
    /* Pedimos un numero al usuario y se lo asignamos a la variable
    pasada por parametro. */
    printf("Escriba un numero: ");
    scanf("%d", numero);
}
```

Solución con referencias (válido sólo para C++)

```
#include <stdio.h>

/* Declaramos la función asignar. */

void asignar(int &numero);

int main (void)
{
    int mi_numero;
```

```
   /* Llamamos a la función asignar, en la que el usuario da un
   valor a la variable pasada por parametro. */
   asignar(mi_numero);

   /* Mostramos el valor actual de "mi_numero". */

   printf("\nEl valor de mi_numero es %d.", mi_numero);

   /* Hacemos una pausa hasta que el usuario pulse Intro */
   fflush(stdin);
   printf("\n\nPulse Intro para finalizar...");
   getchar ();
}
void asignar(int &numero)
{
   /* Pedimos un numero al usuario y se lo asignamos a la variable
   pasada por parametro. */
   printf("Escriba un numero: ");
   scanf("%d", &numero);
}
```

Nota:

Para que una función pueda modificar una variable pasada por parámetro debe utilizar punteros (válido para C y C++) o referencias (válido sólo para C++). En este caso hablamos de paso de parámetros por referencia.

Ficheros

Ejercicio 1

Enunciado

Realice un programa que pida al usuario el nombre de un fichero de texto y, a continuación, muestre todo su contenido en la pantalla. Controle la posibilidad de que el fichero no exista.

Solución

```
#include<stdio.h>
int main (void)
{
   FILE *fichero;
   char nombre_fichero[256];
   char carácter;
```

```
    /* Pedimos al usuario el nombre del fichero y lo almacenamos en
    la variable "nombre_fichero". */
    printf("\nEscriba el nombre del fichero que desea visualizar:");
    gets(nombre_fichero);

    /* Abrimos el fichero. */
    fichero = fopen(nombre_fichero, "rt");

    /* Si "fichero" es igual a NULL, el fichero no se ha abierto.
    Entonces mostramos un mensaje de error. */
    if (fichero==NULL)
    {
       printf("Error: No se ha podido abrir el fichero %s", nombre fichero);
    }
    else
    {
       /* Si el fichero se ha abierto, leemos su contenido carácter a carácter
       y lo mostramos en la pantalla, mientras no se llegue al final del fichero.
       Después cerramos el fichero. */
       carácter = fgetc (fichero);
       while (feof (fichero)==0)
       {
          printf ("%c", carácter);
          carácter = fgetc (fichero);
       }

       fclose(fichero);
    }

    /* Hacemos una pausa hasta que el usuario pulse Intro */
    fflush(stdin);
    printf("\n\nPulse Intro para finalizar...");
    getchar();
}
```

Nota:

Realice siempre la comprobación de si el fichero se ha abierto o no, e informe de ello al usuario para el caso de que no sea posible. De esta manera evitamos realizar operaciones de escritura o lectura en un fichero que realmente no se ha abierto y la aparición de errores no deseados durante la ejecución.

Ejercicio 2

Enunciado

Escriba un programa que permita almacenar en un fichero de texto tantas frases como el usuario desee.

Puede comprobarse si se han almacenado correctamente con el programa anterior.

Solución

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

int main (void)
{
   FILE *fichero;

   char nombre_fichero[256];

   char frase[256];

   char continuar;

   /* Creamos el fichero donde almacenar las frases. */

   fichero = fopen("frases.txt", "wt");

   /* Si "fichero" es igual a NULL, el fichero no se ha creado. Entonces
   mostramos un mensaje de error. */
   if (fichero==NULL)
   {
      printf("Error: No se ha podido crear el fichero frases.txt");
   }
  else
  {
     /* Si el fichero se ha creado, pedimos frases al usuario mientras
     lo desee y las almacenamos en el fichero. Antes de almacenarlas en
     el fichero les concatenamos un salto de linea ("\n") para que cada
     frase quede en una linea distinta del fichero.
     Después cerramos el fichero. */
     do {
        printf("\nEscriba una frase: ");
        gets(frase);
        strcat(frase, "\n");
        fwrite (Sfrase, strlen(frase), 1, fichero);

        printf("\nDesea almacenar otra frase? (S/N): ");
        continuar=getchar();
        fflush(stdin);

     }while ((continuar=='s')   || (continuar=='S'));

     fclose(fichero);
  )

  /* Hacemos una pausa hasta que el usuario pulse Intro */
  fflush(stdin);
  printf("\n\nPulse Intro para finalizar...");
  getchar () ;
}
```

Programación orientada a objetos (POO)

Ejercicio 1

Enunciado

Defina la clase y declare sus métodos para el siguiente objeto: una nave de un videojuego. Se sabe que la nave se puede situar en unas coordenadas (x, y) de la pantalla y que tiene un máximo de 3 vidas. Además, la nave puede moverse en la pantalla (subir, bajar, ir a la derecha e ir a la izquierda) y ser destruida (no más de 3 veces).

Solución

La implementación de la clase `nave` y sus métodos la hemos acompañado al final con una función main para probarla, por eso aparece al principio del siguiente listado un `inelude`.

```
#include <iostream.h>

/*---- Definición de la clase nave -*/

class nave {
      int
      posicion_x;
      int
      posicion_y;
      int vidas;

   public:
      nave ();
      nave (int x, int y);
      void posicionar(int x, int y);
      void subir (void);
      void bajar (void);
      void derecha (void);
      void izquierda (void);
      void destruir (void);
      void mostrar_estado (void);
};

/* ---- Métodos de la clase nave---- */

nave::nave()
{
    /* Inicializamos los atributos posicion_x y posicion_y con el valor
    cero e inicializamos el atributo vidas con el valor 3. */
    posicion_x = 0;
    posicion_y = 0;
```

```
    vidas = 3;
}

nave::nave(int x, int y)
{
    /* Asignamos a los atributos posicion_x y
    posicion_y los valores pasados por parametro
    al constructor e inicializamos el atributo vidas
    con el valor 3. */
    posicion_x = x;
    posicion_y = y;
    vidas = 3;
}

void nave::posicionar(int x, int y)
{
    /* Asignamos a los atributos posicion_x y posicion_y los
    valores pasados por parametro al constructor. */
    posición x = x; posicion_y = y;
}

void nave:¡subir (void)
{
    /* Decrementamos el valor del atributo posicion_y para
    mover la nave hacia arriba.*/
    posicion_y = posicion_y + 1;
}

void nave:¡bajar (void)
{
    /* Decrementamos el valor del atributo posicion_y para
    mover la nave hacia abajo.*/
    posicion_y — posición y - 1;
}


void nave::derecha (void)
{
    /* Incrementamos el valor del atributo posición x para
    mover la nave a la derecha.*/
    posicion_x = posicion_x + 1;
}

void nave::izquierda (void)
{
    /* Decrementamos el valor del atributo posicion_x para
    mover la nave a la izquierda.*/
    posicion_x = posicion_x - 1;
}


void nave::destruir (void)
{
    /* Al destruir la nave le queda una vida menos, pero si
    tiene cero vidas no se le restan mas. */
```

```
    if (vidas>0)
    {
       vidas = vidas - 1;
    }

}

void nave::mostrar_estado (void)
{
    /* Mostramos los atributos de la clase nave. */
    cout << "posicion_x: " << posicion_x << "\n";
    cout << "posicion_y: " << posicion_y << "\n";
    cout << "vidas : " << vidas << "\n";
}

/*----- Probamos las clases en una función main -------*/

int main (void)
{
    /* Creamos un objeto nave llamado "enterprise" que ejecuta
    el constructor sin parámetros. */
    nave enterprise;

    /* Creamos un objeto nave llamado "x_wing" que ejecuta el
    constructor sin parámetros. */
    nave x_wing;

    /* Creamos un objeto nave llamado "tie_fighter" que ejecuta
    el constructor con parámetros, situando la nave en las
    coordenadas 100, 200. */
    nave tie_fighter(100, 200);

    /* Mostramos el estado inicial de las tres naves. */
    cout << "---Enterprise --  (Estado inicial)\n";
    enterprise.mostrar_estado();
    cout << "----- X_Wing ----  (Estado inicial) \n";
    x_wing .mostrar_estado () ;
    cout << "---Tie Fighter— (Estado inicial)\n";
    tie_fighter.mostrar_estado();

    /* Damos una nueva posición a la nave enterprise. */
    enterprise.posicionar(30, 50);

    /* Movemos la nave x_wing arriba y a la derecha. */
    x_wing.subir(); x_wing.derecha();

    /* Destruimos una vez la nave tie_fighter. */
    tie_fighter.destruir() ;

    /* Mostramos el estado final de las tres naves. */
    cout << "---Enterprise --  (Estado final)\n";
    enterprise.mostrar_estado();
    cout << "----- X_Wing ----  (Estado final) \n";
    x_wing.mostrar_estado();
```

```
    cout << " --Tie Fighter— (Estado final)\n";
    tie_fighter.mostrar_estado();

    /* Hacemos una pausa hasta que el usuario pulse Intro */
    fflush(stdin);
    cout << "\nPulse Intro para finalizar...";
    cin.get();
}
```

Ejercicio 2

Enunciado

Defina una jerarquía de clases y sus métodos para estos objetos: vehículo, coche, moto. Tanto el coche como la moto son vehículos. Ambos vehículos tienen un número de ruedas y una velocidad máxima. Por otro lado, el coche tiene una característica que las motos no tienen: el número de puertas. Mientras que las motos tienen la característica de admitir una o dos plazas.

Solución

La implementación de las clases y sus métodos la hemos acompañado al final con una función main que utiliza las clases moto y coche, y sus métodos, por eso aparece al principio del siguiente listado un inelude.

```
#include <iostream.h>

/*----- Definición de la superclase vehiculo ------ */

class vehiculo {
       int numero_ruedas;
       int velocidad_maxima;

    public;
       vehiculo (int ruedas, int velocidad);
       void mostrar (void);

} ;

/*----- Definición de la subclase coche ------ */

class coche:public vehiculo (

int numero_puertas; public:
       coche (int ruedas, int velocidad, int puertas);
       void mostrar (void);
};
```

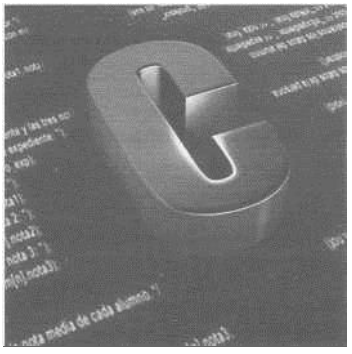
```
/*------Definición de la subclase moto------ */

class moto:public vehiculo {

char biplaza;

public:
       moto (int ruedas, int velocidad, char bi);
       void mostrar (void);
};

/*------Métodos de la superclase vehiculo------ */

vehiculo::vehiculo (int ruedas, int velocidad)
{
   /* Asignamos a los atributos de la superclase vehiculo los
   valores pasados por parametro al constructor. */
   numero_ruedas = ruedas; velocidad_maxima = velocidad;
}

void vehiculo:¡mostrar (void)
{
   /* Mostramos los atributos de la superclase vehiculo. */
   cout << "Numero de ruedas : " << numero_ruedas << "\n";
   cout << "Velocidad maxima : " « velocidad_maxima << "\n";
}

/* ---- Métodos de la subclase coche----*/

coche::coche(int ruedas, int velocidad, int puertas)
                   :vehiculo(ruedas, velocidad)
{
   /* Asignamos al atributo numero_puertas el valor
   pasado por el tercer parametro del constructor. */
   numero_puertas = puertas;
}



void coche:¡mostrar (void)
{
   /* Llamamos al método mostrar de la superclase
   para mostrar sus atributos. */
   vehiculo:¡mostrar();

   /* Mostramos los atributos de la subclase coche. */
   cout << "Numero de puertas: " << numero_puertas << "\n";
}

/* ---- Métodos de la subclase moto-----*/

moto::moto (int ruedas, int velocidad, char bi)
                   :vehiculo(ruedas, velocidad)
{
   /* Asignamos atributo biplaza eL valor pasado
   por el tercer parametro del constructor. */
```

```
    biplaza = bi;
}

void moto:¡mostrar (void)
{
    /* Llamamos al método mostrar de la superclase para mostrar sus
    atributos. */

    vehiculo:¡mostrar ();

    /* Mostramos los atributos de la subclase moto. */
    cout << "Biplaza       : " << biplaza << "\n";
}

/*----- Probamos las clases en una función main -------*/

int main (void)
{
    /* Creamos un objeto moto. */

    moto mi_moto(2, 80, 'N');

    /* Creamos un objeto coche. */

    coche mi_coche(4, 220, 5);

    /* Mostramos los datos del coche. */
    cout << " --  DATOS DEL COCHE  ----\n";
    mi_coche.mostrar();

    /* Mostramos los datos de la moto. */
    cout << " --  DATOS DE LA MOTO  ---\n";
    mi_moto.mostrar();

    /* Hacemos una pausa hasta que el usuario pulse Intro */
    fflush(stdin);
    cout << "\nPulse Intro para finalizar...";
    cin.get();
}
```

Los métodos mostrar de la clases coche y moto contienen esta línea:

```
vehiculo::mostrar();
```

que se encarga de llamar al método mostrar de la superclase vehiculo. En este ejemplo debemos indicar explícitamente el nombre de la superclase, cuyo método mostrar queremos ejecutar, ya que todas las clases tienen un método con el mismo nombre. Si no especificamos la superclase se ejecutaría el método mostrar de la misma subclase, obteniendo una llamada recursiva infinita.



Apéndice E

Tabla de caracteres ASCII

Tabla de caracteres ASCII

El código ASCII, del inglés *American Standard Codefor Information Interchange* (Código Estándar Americano para el Intercambio de Información), es un estándar compuesto por los 128 caracteres más usados por los países occidentales (numerados del 0 al 127), incluyendo letras, símbolos de puntuación, números y símbolos o valores empleados en comunicación. El código ASCII extendido incluye otros 128 caracteres más (numerados del 128 al 255), pero no está estandarizado y puede variar de una plataforma a otra.
Debemos tener en cuenta que los ordenadores tan sólo entienden los números, concretamente binario, y que toda la información que almacenamos en ellos es en forma de números, aunque nosotros veamos letras, números y otros símbolos. El código ASCII establece una relación entre los caracteres y el código numérico equivalente.

Tabla E.1. Tabla de caracteres ASCII.

Carácter/Significado	Código ASCII
NUL (Carácter nulo)	0
SOH (Inicio de encabezado)	1
STX (Inicio de texto)	2
ETX (Fin de texto)	3
EOT (Fin de transmisión)	4
ENQ (Enquiry)	5
ACK (Acknowledgement)	6
BEL (Timbre)	7
BS (Retroceso)	8
TAB (Tabulación horizontal)	9
LF (Salto de línea)	10
VT (Tabulación vertical)	11
FF (Form feed)	12
CR (Retorno de carro)	13
SO (Shift out)	14
SI (Shift in)	15

Carácter/Significado	Código ASCII
DLE (Data link escape)	16
DC1 (Device control 1)	17
DC2 (Device control 2)	18
DC3 (Device control 3)	19
DC4 (Device control 4)	20
NAK (Negative acknowledgement)	21
SYN (Synchronous idle)	22
ETB (Fin de bloque de transmisión)	23
CAN (Cancelar)	24
EM (End of medium, fin de medio)	25
SUB (Sustituto)	26
ESC (Escape)	27
FS (Separador de archivo)	28
GS (Separador de grupo)	29
RS (Separador de registro)	30
US (Separador de unidad)	31
SP (Espacio)	32
!	33
"	34
#	35
$	36
%	37
&	38
'	39
(	40
)	41
*	42
+	43

Carácter/Significado	Código ASCII
,	44
-	45
.	46
/	47
0	48
1	49
2	50
3	51
4	52
5	53
6	54
7	55
8	56
9	57
:	58
;	59
<	60
=	61
>	62
?	63
@	64
A	65
B	66
C	67
D	68
E	69
F	70
G	71

Carácter/Significado	Código ASCII
H	72
I	73
J	74
K	75
L	76
M	77
N	78
0	79
P	80
Q	81
R	82
S	83
T	84
U	85
V	86
W	87
X	88
Y	89
Z	90
[	91
\	92
]	93
^	94
_	95
`	96
a	97
b	98
c	99

Carácter/Significado	Código ASCII
d	100
e	101
f	102
g	103
h	104
i	105
j	106
k	107
l	108
m	109
n	110
o	111
p	112
q	113
r	114
s	115
t	116
u	117
v	118
w	119
X	120
y	121
z	122
{	123
|	124
}	125
~	126
DEL (Borrar)	127

Una forma de ver todos los caracteres de la tabla ASCII de nuestro ordenador es mediante un pequeño programa de C que muestra en pantalla cada carácter seguido de su correspondiente código encerrado entre paréntesis.

```
#include <stdio.h>

int main (void)
{
    int codigo;

    /* Mostramos la tabla ASCII completa*/
    printf("Tabla ASCII\n");
    for (codigo=0; codigo<=255; codigo++)
    {
       printf("%c (%d)\t", codigo, codigo);
    }

    /* Hacemos una pausa hasta que el usuario pulse Intro*/
    getchar();
}
```

En este programa utilizamos una variable de tipo entero llamada `codigo` que tomará los valores de 0 a 255. Mientras, en la función `printf` forzamos la representación de este número al carácter equivalente en la tabla ASCII mediante el uso de `%c`.

También podemos hacer un programa que solicite al usuario un número entre 0 y 255 y muestre en pantalla el carácter ASCII equivalente. El código sería el siguiente.

```
#include <stdio.h>

int main (void)
{
    int codigo;

    /* Pedimos el codigo al usuario */

    printf("Escriba un codigo ASCII (0-255): ");

    scanf("%d", &codigo);

    fflush(stdin);

    /* Mostramos el carácter equivalente */
    printf("El carácter equivalente es: %c", codigo);

    /* Hacemos una pausa hasta que el usuario pulse Intro*/
    getchar();
}
```

Ya solamente nos falta hacer un programa que solicite al usuario un carácter y muestre en pantalla el código ASCII equivalente. A continuación, vemos el código.

```
#include <stdio.h>

int main (void)
{
   int carácter;

   /* Pedimos el codigo al usuario */
   printf ("Escriba un carácter y después pulse Intro: ");
   scant("%c", &caracter); fflush(stdin);

   /* Mostramos el carácter equivalente */
   printf("El codigo ASCII equivalente es: %d", carácter);

   /* Hacemos una pausa hasta que el usuario pulse Intro*/
   getchar();
}
```

Obviamente, estos pequeños programa son simplemente para comprender un poco más cómo podemos manejar los caracteres y sus códigos ASCII en C/C++. Si lo desea puede añadir mejoras a estos códigos.