Introducción a la programación en C

12

En esta Unidad aprenderás a…

1. **Conocer como es un programa en C. Desde el código fuente hasta el fichero ejecutable en varios entornos típicos de PC.**
2. **El concepto de variable y constante. Los diferentes tipos de variables básicas en C.**
3. **Hacer programas sencillos en C.**
4. **Detectar los errores más frecuentes en programación y como evitarlos.**

Introducción

En estos capítulos vamos a estudiar los rudimentos de un lenguaje de programación. Como ya sabemos, un lenguaje de programación es una herramienta que nos permite elaborar programas (ficheros ejecutables) de forma más o menos fácil. Los procesadores tan solo son capaces de ejecutar instrucciones muy sencillas, tales como la suma de dos números, mover un valor del procesador a la memoria, etc. Además, para entender este tipo de instrucciones, el procesador debe recibirlas codificadas en sistema binario, es decir, como secuencia de unos y ceros. Para entendernos, si el procesador recibe un 1000 0111, por ejemplo, entenderá que debe sumar dos cifras (y otra secuencia de unos y ceros le dirá cuáles), y si recibe un 1000 0100, en cambio, lo que hará es restarlas. Es a base de dichas secuencias binarias y de muchas instrucciones que TODOS los procesadores consiguen ejecutar cualquier programa (incluidos, por ejemplo, los juegos más complejos y con mejores gráficos). Este idioma binario es lo que se conoce como lenguaje máquina.

Aunque se podría crear un programa simplemente escribiendo los cientos de millones de unos y ceros que representan las órdenes del procesador (y en los años sesenta, con los primeros ordenadores, así se hacía), este sistema resulta muy complicado. Por ese motivo, se decidió simplificar un poco la tarea y se crearon los lenguajes de programación. Estos lenguajes son simplemente programas que traducen las órdenes que nosotros les escribimos siguiendo unas reglas sintácticas (por eso se llaman lenguajes) al idioma de unos y ceros, el único que el procesador es capaz de entender. Según lo cerca o lo lejos que esté el lenguaje de programación del idioma binario del ordenador, se clasifica en tres grupos:

- **Lenguajes de bajo nivel.** El lenguaje de bajo nivel por excelencia es el ensamblador. Este lenguaje es simplemente una traducción directa de cada orden del procesador a una palabra que explique qué hace esa orden (la palabra, evidentemente, está en inglés). En el ejemplo anterior, en vez de escribir 1000 0111, escribiríamos ADD (*suma* en inglés). No facilita mucho las cosas, porque sigue siendo necesario escribir muchísimas ordenes para elaborar programas muy simples, pero ayuda porque es más sencillo recordar que para sumar hay que escribir ADD que acordarse de escribir 1000 0111.

- **Lenguajes de alto nivel.** Los lenguajes de alto nivel, entre los que se encuentra el C, son la base de la programación. En estos lenguajes, en vez de escribir ADD, por ejemplo, podemos escribir instrucciones del tipo "C=A+B" (es decir, C es igual a la suma de A más B), mucho más cercanas a nuestro idioma natural. Existen muchísimos lenguajes de alto nivel: Basic, Pascal, Fortran, Cobol, C... pero todos son muy parecidos, de forma que una vez aprendido uno, es muy fácil aprender el resto. Además, C es uno de los más usados en el mundo (por no decir el que más).

- **Lenguajes orientados a objetos.** Estos lenguajes son de nivel todavía más alto que los anteriores, aunque se basan en ellos (por ejemplo C++ y Java, dos de los mas comunes, son extensiones de C). Al contrario de lo que ocurre con los lenguajes de alto nivel y el ensamblador, no es posible aprender un lenguaje orientado a objetos sin conocer antes un lenguaje de alto nivel. El objetivo de los lenguajes orientados a objetos es, sobre todo, crear programas, no desde cero, sino juntando piezas programadas por otras personas.

El lenguaje que vamos a aprender aquí es el C. Se trata seguramente del lenguaje de programación de alto nivel más usado en el mundo. Su origen se remonta a los años 1969-1973, cuando Dennis Ritchie generó un nuevo lenguaje a partir del lenguaje "B" creado por Ken Thompson. Lógicamente, lo llamó C. Cuenta la leyenda que Ritchie en realidad quería ser más eficiente jugando al juego Space Travel (http://cm.bell-labs.com/cm/cs /who/dmr/spacetravel.html, ojo, está en inglés), necesitaba un sistema operativo nuevo y decidió crearlo. Sin embargo, escribir un sistema operativo en ensamblador era una tarea bastante pesada, así que prefirió crear un lenguaje de programación nuevo que le facilitara la tarea. Por cierto: el sistema operativo que creó con C fue ni más ni menos que Unix. Los gastos los pagaron los laboratorios Bell de AT&T, y Ritchie justificó la compra de un PDP-11 (un ordenador de la época, bastante caro por cierto) diciendo que iba a crear un sistema para cumplimentar de forma automatizada las patentes.

En 1978, Ritchie y Brian Kernighan publicaron el primer libro de C (*The C Programming Language*) y, a partir ese momento, C se convirtió en un referente de la programación. Actualmente, existen muchas pequeñas variantes. Son tantas que se ha creado un estándar, el ANSI C, compatible con prácticamente todos los compiladores de C. Este estándar es el que vamos a trabajar en el libro. Para terminar, un último detalle sobre los distintos lenguajes de programación. Hemos hablado de compiladores de C sin explicar qué son. Existen dos formas de obtener un código ejecutable a partir de un programa fuente: los intérpretes y los compiladores:

- **Los intérpretes** son lenguajes que leen las instrucciones en alto nivel y las traducen en tiempo real a lenguaje máquina.

- **Los compiladores,** en cambio, leen todo el fichero con el programa y lo traducen a otro fichero en lenguaje máquina, que es el que podemos ejecutar. Existen, pues, dos ficheros en vez de uno, pero la gran ventaja es que una vez compilado el programa, el compilador ya no es necesario y, además, el lenguaje compilado se ejecuta mucho más rápido que el interpretado.

C pertenece a este último tipo de lenguajes. En los siguientes apartados veremos cómo realizar todo el proceso para conseguir un programa ejecutable.

12.1 Un primer programa en C

Casos prácticos

1 // Todas las líneas que empiezan por // son comentarios que sirven para explicar cómo es el

// programa. Se puede escribir lo que se quiera.

/* Si se quieren hacer comentarios de más de una línea se puede escribir todo el comentario entre los símbolos /* (inicio de comentario) y fin de comentario: */

> Una **palabra reservada** es una palabra que tiene un significado especial en el lenguaje y sólo se puede usar para eso en concreto.

/* Al principio del texto es útil poner comentarios que expliquen cosas del programa:*/

// Programa: Bienvenida.c

// Utilidad: Imprime en pantalla un mensaje de bienvenida.

// Programador: Yo mismo.

// Fecha: La de hoy.

// Versión: 1.0

/* Una vez se han escrito todos los comentarios que parezcan oportunos, al principio del texto hay que poner todas aquellas indicaciones generales que necesite el programa. Por ejemplo: */

#include <stdio.h>

/* En la línea anterior, *#include* es una **palabra reservada.** En C, *#include* significa que a continuación se va a poner entre los signos < y > el nombre de un fichero ya programado. Ese fichero contiene órdenes del lenguaje que se van a usar en nuestro programa.

Concretamente, la línea anterior dice que hay que incluir el fichero *stdio.h*, ya que más adelante en el programa se va a usar una orden que hay programada dentro de él. Estos ficheros con órdenes ya programadas son las **librerías.***/

> Una **librería** es un conjunto, entre otras cosas, de **subrutinas** de uso habitual ya programadas. Resulta muy útil ya que ahorra trabajo a la hora de crear programas, porque no hay que volver a programarlas.

/*Las órdenes concretas que se ejecutan al empezar el programa deben ir justo a continuación de la palabra reservada *main* (principal). *main* es una palabra reservada que sirve para indicar el inicio del programa propiamente dicho y siempre debe ir seguida de "()". Más adelante ya se explicará por qué.*/

main()

{

/*Los símbolos *{* y *}* sirven para indicar el principio y el final de las órdenes a ejecutar. Nuestro programa sólo va a imprimir en pantalla. Para imprimir en pantalla hay una orden ya programada en el fichero *stdio.h* (por eso se incluyó antes) que se llama *printf*. Esta orden es lo que se llama una **subrutina**.*/

printf("Hola, soy un programa muy educado.\n");

> Una **subrutina** es un conjunto de órdenes más simples agrupadas bajo un nombre.

/* La subrutina printf, imprime en pantalla lo que se le indica a continuación entre paréntesis. Esta información, que se envía a una subrutina, se llama parámetro y en este caso es un mensaje entre comillas. El ; de final indica que se ha acabado la orden.*/

}

> Un **parámetro** es una información que se envía a una **subrutina** y que habitualmente modifica su funcionamiento.

Un programa en C es un texto plano (como el que se obtiene con el bloc de notas o con los editores emacs o vi) escrito usando una serie de reglas. Estas reglas sirven para mandar al ordenador la ejecución de una serie de tareas y dependen del lenguaje. Concretamente, en C, un programa tiene la estructura que se puede ver en el caso práctico 1. El programa de ejemplo parece bastante largo, pero si se limita al texto básico, queda reducido a lo que se ve en el caso práctico 2:

Casos prácticos

2 // Programa: Bienvenida.c
// Utilidad: Imprime en pantalla un mensaje de bienvenida.
// Programador: Mismamente yo.
// Fecha: La de hoy.
// Versión: 1.0

```
#include <stdio.h>

main()

{
   printf("Hola, soy un programa muy educado.\n");
}
```

Se llama **cadena** a una sucesión de caracteres. Habitualmente se escribe entre comillas dobles y suele contener una palabra o frase.

Se observa claramente que el programa simplemente imprime una **cadena** en pantalla.

12.2 Cómo crear un programa ejecutable

Imaginemos ahora que ya tenemos el texto anterior escrito en un fichero de texto, que hemos llamado *Bienvenida.c*. ¿Cómo se consigue que ese texto se transforme en algo que el ordenador pueda entender? Para ello hay que realizar dos operaciones: **compilarlo** y **linkarlo.**

- **Compilar** un programa significa transformar el texto plano en un código que el procesador pueda entender. De esta tarea se encarga un programa llamado compilador.

- **Linkar** un programa significa juntarlo con otros trozos de programa. En nuestro ejemplo debemos juntar lo que nosotros hemos hecho con todo lo que hay programado en la librería *stdio*, concretamente con el código de la subrutina *printf*. Esta tarea la realiza un programa denominado linkador.

Actualmente, muchos compiladores de C efectúan los dos pasos a la vez y se dice, simplemente, que se compila el programa.

Así pues, para obtener un programa ejecutable, en primer lugar debe escribirse su código fuente en un fichero de texto. A continuación, hay que compilar el programa, para obtener el fichero ejecutable. El proceso es básicamente el mismo en todos los sistemas operativos, pero la forma de trabajar no es la misma en Windows que en Linux, así que veremos ambos casos por separado.

LINUX

En primer lugar debemos escribir el texto del programa. Para ello ejecutamos cualquier procesador de texto (vi, vim, emacs, kedit, nedit...) y creamos el fichero *bienvenida.c* con el texto que hemos visto en el ejemplo anterior. Guardamos el fichero.

A continuación, compilamos (y linkamos en el mismo paso) el programa con el compilador de C de GNU: *gcc*.

Para hacerlo, abrimos una consola y desde el directorio donde hayamos guardado el fichero *bienvenida.c* escribimos la orden:

 # gcc bienvenida.c -o bienvenida

Esta orden crea el fichero ejecutable *bienvenida*, que podemos ejecutar simplemente escribiendo (Figura 12.1):

 # bienvenida

WINDOWS

En Windows se suele trabajar directamente con un entorno de programación que agrupa, entre otros elementos, el editor de texto y el compilador. Algunos de los entornos de programación más usuales en Windows son Dev-C++ (incluido en el CD), Visual Studio o Turbo C. Además, con Windows también se puede proceder en modo consola, siguiendo los mismos pasos que hemos visto para Linux.

Aunque los principios básicos para conseguir un programa ejecutable son siempre los mismos, vamos a describir ahora las particularidades de los entornos más usuales.

Dev-C++:

Dev-C++ es un entorno de desarrollo de C con licencia libre. Se puede conseguir la última versión en http://www.bloodshed.net (también hay una copia en el CD). Una vez instalado el software, hay que ejecutar el entorno Dev-C++ (Figura 12.2). A continuación, debemos escribir el programa. Para ello, creamos un *Nuevo Código Fuente*. Esto abrirá una nueva ventana en blanco, donde hay que escribir el código del ejemplo anterior. A continuación, lo guardamos con el nombre *Bienvenida.c* y como tipo *Código Fuente de C* (extensión *.c*, Figura 12.3).

Una vez tenemos el código escrito podemos compilarlo dentro del mismo entorno. Para ello basta con pulsar el botón adecuado (arriba a la izquierda), hacer clic con el ratón en la primera opción del menú *Ejecutar* o pulsar *Ctrl+F9*. Si la compilación no da ningún error, podemos *ejecutar* ya el programa (*Ctrl+F10*).

Lo más probable es que después de seguir los pasos anteriores no veamos nada. Esto es debido a que el programa abre una consola (una ventana de texto), imprime en ella lo que le hemos dicho y a continuación la ventana se cierra antes de que podamos ver nada. Si queremos ver algo, debemos conseguir que el programa se pare antes de acabar. Así pues, debemos utilizar la subrutina *system*, que envía una orden al sistema operativo: detenerse. El programa, entonces, queda así:

```
#include <stdio.h>
#include <stdlib.h> /*Esta es la librería donde está la orden system.*/

main()
{
  printf("Hola, soy un programa muy educado.\n");
   system("PAUSE"); /*Esta orden sólo hace que el programa se detenga y podamos ver lo que escribe.*/
}
```

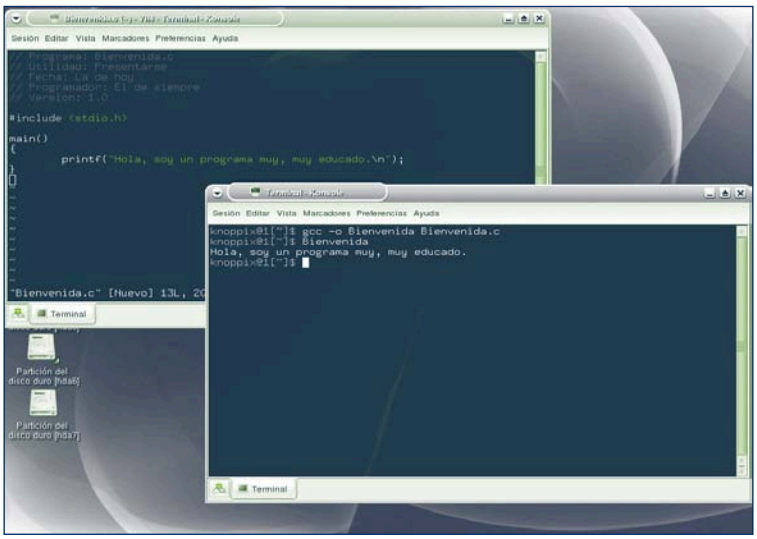

Fig. 12.1. *El entorno Linux. Una ventana con un editor del programa y otra con la consola y las órdenes de compilación y ejecución, y el resultado.*

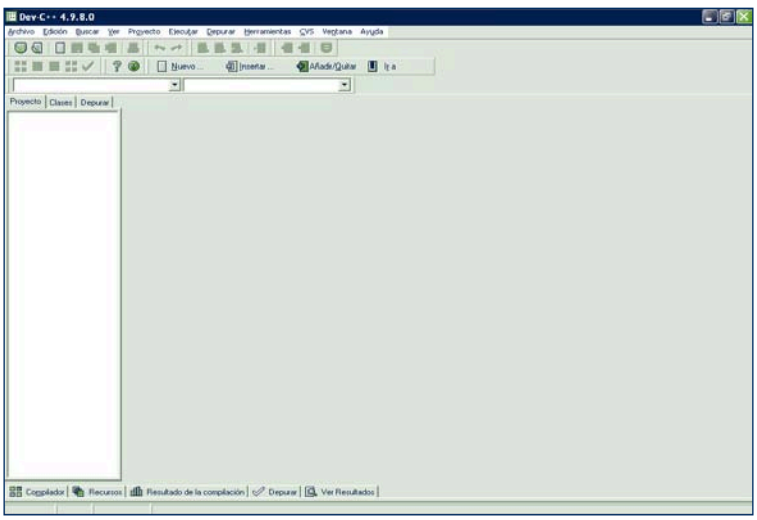

Fig. 12.2. *Entorno Dev-C++ con el editor en blanco.*

Una vez modificado el código del programa, deberemos compilarlo y ejecutarlo de nuevo. Podemos hacerlo en un solo paso pulsando directamente *F9*.

Para verificar que realmente hemos creado un programa, basta con cerrar el entorno e ir a la carpeta donde guardamos el fichero *Bienvenida.c*. Veremos que ha aparecido un nuevo fichero llamado *Bienvenida.exe*. Si hacemos doble clic sobre este fichero, se ejecutará el programa.

MS Visual Studio:

Éste es el entorno desarrollado por Microsoft para la creación de aplicaciones. Sólo puede conseguirse pagando la licencia, aunque quizás no sea el más adecuado para generar programas simples en C, debido a su complejidad y a su tendencia a crear ejecutables muy grandes. Sin embargo, su uso está bastante extendido. Sólo funciona en sistemas Windows.

Par empezar, hay que ejecutar el entorno Microsoft Visual C++. A continuación, debemos escribir el programa. Para ello vamos al menú *File* y escogemos la pestaña *Files*. Dentro de esta pestaña hay muchas clases de ficheros, pero elegimos *C++ Source File* (Figura 12.4). Antes de aceptar la selección debemos especificar en los cuadros de texto de la derecha el lugar donde lo guardaremos: *c:\Mis Documentos\ejemplo* (*Location*) y el nombre que le queremos dar: *Bienvenida.c* (*File name*). Esto hará que el área de edición, la zona grande que estaba en gris a la derecha, pase a estar en blanco. Haciendo clic con el ratón sobre ella veremos que

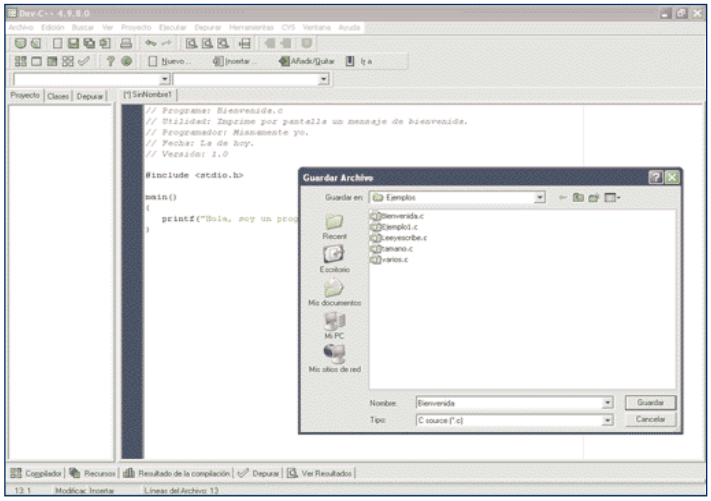

Fig. 12.3. *Guardar el programa, nombre y tipo.*

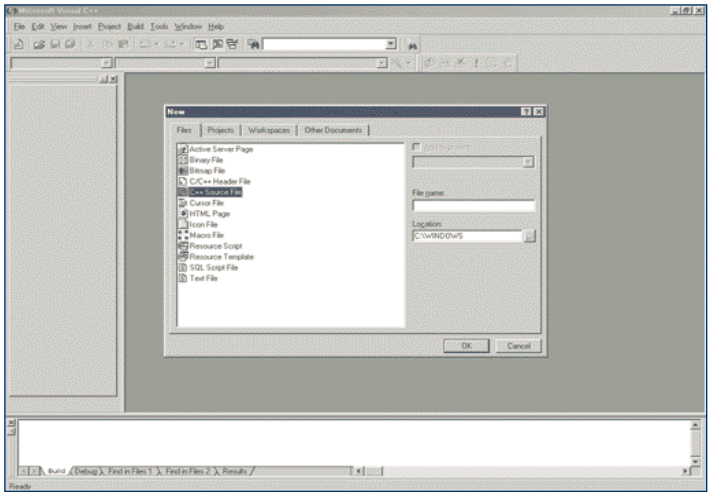

Fig. 12.4. *Entorno Microsoft Visual Studio C++, código de C.*

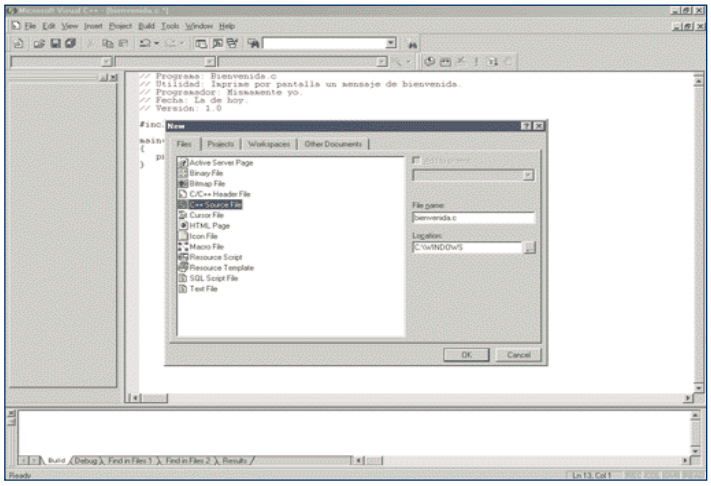

Fig. 12.5. *Edición del programa.*

lo que escribamos en el teclado aparecerá en la pantalla. Debemos escribir el código del ejemplo anterior. Observamos que determinadas palabras del código aparecen coloreadas. En este entorno, el color verde se utiliza para el código que el compilador interpreta como comentario, y el azul para las palabras reservadas.

Una vez escrito el código, podemos compilarlo dentro del mismo entorno. Basta con pulsar el botón adecuado (arriba a la derecha), hacer clic con el ratón en la opción del menú *Build* llamada *Compile* o pulsar *Ctrl+F7*. Aparece un mensaje avisándonos de la necesidad de crear un espacio de trabajo activo (*active project workspace*). Aceptamos, y si la compilación no da ningún error, podemos pasar a *linkar* el programa (opción *build* del menú Build o *F7*). Por último, si tampoco han surgido problemas al enlazar (*linkar, build*) podemos ejecutarlo mediante la opción *Ctrl+F5* o seleccionando en el menú la opción *Execute*. Si todo va bien, el entorno abre una ventana con el resultado de la ejecución y el mensaje final *Press any key to continue* ("Pulse cualquier tecla para continuar."). Este mensaje es similar al obtenido en el entorno Dev-C++ con la instrucción *system("pause")*. Hay que recalcar que este entorno ofrece más posibilidades. Así que en caso de problemas, es posible volver a un punto conocido cerrando el programa y arrancándolo de nuevo.

En este entorno, conviene tener en cuenta los demás ficheros generados en el directorio de la compilación. Si miramos el contenido de dicho directorio (lo escogimos antes) vemos que, además del fichero con el código fuente, incluye otra serie de ficheros con información auxiliar, generados por el programa. Entre ellos cabe destacar el que tiene la terminación *.dsw*, que contiene la información necesaria para que el entorno sepa dónde finalizó la última vez, qué ficheros contiene el proyecto y otras informaciones. Si cerramos el entorno y queremos abrirlo de nuevo, basta con hacer doble clic en este fichero, y recuperamos el trabajo en el punto en el que lo dejamos. Otro fichero que debemos saber dónde está es nuestro programa ejecutable. Dentro del directorio del código fuente figura el directorio *debug* o *release*, que contiene nuestro fichero ejecutable. Que aparezca en un directorio u otro depende de cómo tengamos configurado el entorno. Lo más habitual es configurarlo o tenerlo configurado para que produzca la opción *debug*, que genera ejecutables más grandes y permite la depuración del código (de esta cuestión hablaremos más adelante).

Si ejecutamos el programa que se encuentra en la carpeta *debug*, por ejemplo, observamos que se ejecuta, pero no se para ni muestra el mensaje *Press any key to continue* ("Pulse cualquier tecla para continuar."). Si queremos que se detenga al final en la ejecución fuera del entorno, debemos incluir una instrucción en el código del programa que detenga dicha ejecución hasta que se pulse una tecla. Podemos copiar el ejemplo con esta orden (*system("pause")*) de la explicación del entorno Dev-C++.

Turbo C 2.0

Este software creado por la casa Borland fue un estándar de desarrollo en la década de los 80. Actualmente, Borland no le proporciona asistencia y permite su descarga gratuita desde http://community.borland.com/museum. A pesar de su antigüedad, se trata de un software muy fiable y de uso todavía muy extendido. Funciona en cualquier sistema MS-DOS o superior (Windows). Curiosamente, cuando se trata de realizar programas sencillos resulta mucho más eficiente que otros entornos más complicados, como Visual Studio.

Fig 12.6. *Entorno de Turbo C 2.0.*

Una vez bajado el programa, basta con instalarlo (http://www.elricondelc.com/comos/comotc.php incluye una ayuda) y ejecutar el programa *tc.exe*. En ese momento, se puede entrar en la ventana de edición y crear el programa. Una vez guardado (mediante la opción *Save* del menú *File*) hay que compilarlo (menú *Compile*, opción *Make EXE File*) y ejecutarlo (menú *Run*, opción *Run* o *Ctrl+F9*). La tecla *F10* cambia el cursor entre la ventana de edición y los menús (Figura 12.6).

Podemos observar que, si ejecutamos el programa, no se ve nada, porque el compilador alterna entre las vistas de ejecución (donde se ve el resultado de nuestro programa) y de edición (que es donde lo escribimos). Podemos cambiar entre las vistas con *Alt+F5* o la opción *User Screen* del menú *Run*.

Si queremos que el programa haga una pausa y espere hasta que se pulse una tecla para acabar (tanto dentro del entorno como si ejecutamos el fichero *.exe* que genera el compilador), debemos copiar el código de la explicación del entorno Dev-C++.

En este momento ya estamos en condiciones de intentar hacer las actividades 1 y 2. Ánimo.

12.3 Los primeros errores

Al escribir un programa es muy fácil cometer errores. En programación, los errores pueden ser de varios tipos: sintácticos, de linkado, de ejecución o de algoritmo.

Los errores sintácticos son los más frecuentes y, por suerte, también los más fáciles de corregir. Se producen al escribir una sentencia que el compilador no entiende, es decir, cuando cometemos un error en la escritura. Por ejemplo, el compilador espera que todas las sentencias acaben con un punto y coma. Si no lo encuentra, indica un error y es incapaz de producir el fichero ejecutable (Figura 12.7). Para experimentar con los primeros errores, podemos realizar la actividad 3.

Como acabamos de ver, los compiladores son muy sensibles a los errores sintácticos. En general, en sus mensajes, todos los compiladores intentan señalar cuál ha sido el origen del error y en qué punto se ha producido, pero en algunos casos (como ocurre con las comillas) no lo hacen demasiado bien y, además, nos dan una multitud de mensajes por un solo error. Esto indica que, siempre que sea posible, los errores deben corregirse por orden: empezando por el primero, volviendo a compilar, y así sucesivamente hasta corregirlos todos. A medida que nos acostumbremos a encontrar errores en nuestros programas, iremos entendiendo los mensajes de error y mejorando nuestra habilidad para corregirlos.

Los errores de linkado se producen cuando el compilador no encuentra la librería adecuada para utilizar una subrutina o interpreta algún nombre como una función desconocida. Encontraremos un ejemplo en la actividad 4.

En los dos casos anteriores, el compilador no llega a generar un fichero ejecutable (si vemos que existe, probablemente lo hayamos generado en una compilación anterior). Los errores de ejecución se producen cuando el código fuente genera un programa (fichero ejecutable) que al iniciarse no llega a ejecutarse completamente por alguna causa. En el apartado siguiente analizaremos un error de esta clase.

El último tipo de error, el de algoritmo, se produce cuando no escribimos en el programa las instrucciones adecuadas o en el orden adecuado. No produce ningún tipo de mensaje de error pero se aprecia porque el programa no hace lo que debería hacer. A veces, darse cuenta de este tipo de errores es difícil en programas grandes que se ejecutan de muchas maneras distintas. Para verlos, hay que probar el programa.

¿Cómo se prueba un programa? Muy fácil, ejecutándolo. ¿Ha funcionado bien? En programas sencillos, la respuesta a esta pregunta es un simple sí o no. Sin embargo, en programas grandes, la respuesta

a esta pregunta podemos formularla de una manera no tan trivial: con los datos de entrada introducidos en el programa ¿es correcto el resultado obtenido? Si la respuesta a esta última pregunta es **sí, con cualquiera de los posibles datos de entrada, la salida siempre es correcta,** nuestro programa funciona. Pero, pensemos en la frase puesta en negrita. ¿Qué ocurre si los posibles datos de entrada son cientos de miles? ¿Hemos de ejecutar el programa cientos de miles de veces para comprobarlo?

La mayoría de las veces, basta con unas pocas ejecuciones para saber si funcionará siempre, pero escogiendo los datos de entrada de forma que prueben todos los casos posibles distintos. En muchas ocasiones, sin embargo, la tarea termina no siendo tan simple. Tal y como se demuestra con la experiencia, muchos programas importantes presentan nuevos errores cada poco tiempo (seguro que a todos se nos ocurren ejemplos sin necesidad de citar aquí uno). En cualquier caso, conviene tener en cuenta que la detección y corrección de errores es una de las partes más costosas de un programa. Así pues, resulta muy importante planearlos bien e intentar programarlos de la forma más clara posible. Seguiremos insistiendo en esta cuestión más adelante.

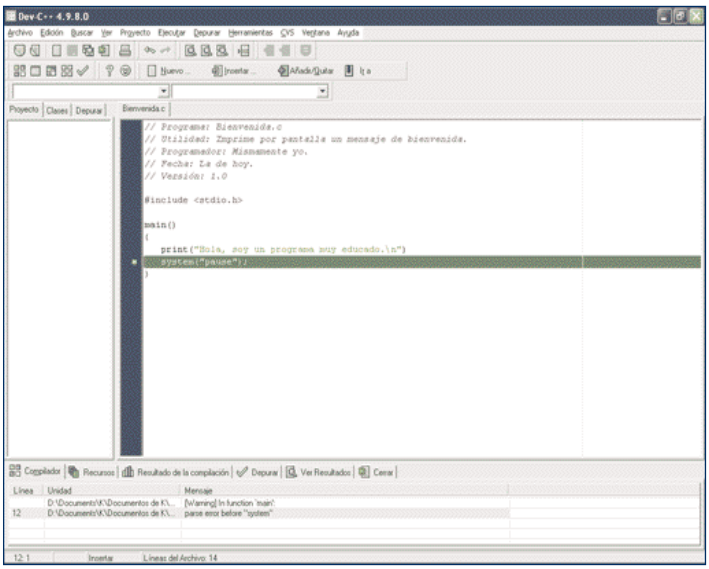

Fig. 12.7. *Errores de compilación en Dev-C++ cuando falta un punto y coma.*

12.4 Variables enteras

Si nos interesa crear un programa algo más complejo que el del ejemplo del punto anterior, nos convendrá seguramente guardar datos para poder usarlos más adelante. Imaginemos por ejemplo un programa que pida un número por teclado y a continuación lo imprima en pantalla. En el tiempo que transcurre entre que lo leemos y lo escribimos, tenemos que guardarlo en algún lugar. Esta función es la que cumplen las variables. Las **variables** son un espacio de la memoria del ordenador donde nuestro programa almacena datos útiles. Para poder trabajar con ellas debemos asignarles un **nombre** y un **tipo**. El nombre es, simplemente, la forma de llamarlas para poder usarlas, mientras que el tipo especifica qué guarda la variable. Una variable de tipo entero, por ejemplo, puede guardar un número entero, es decir, 0, 23 o -1456. Un detalle a tener en cuenta es que los nombres de variable en C diferencian entre minúsculas y mayúsculas, así que hay que tener cuidado de no equivocarse. Una buena práctica es escribirlas siempre de la misma forma (por ejemplo, todas en minúsculas). Veamos un programa que lee e imprime un número entero en el caso

Casos prácticos

3

```
// Programa: Leeyescribe.c
/* Utilidad: Lee un número entero de teclado y lo imprime en
   pantalla.*/
// Programador: Yo mismo.
// Fecha: Hoy mismo.
// Versión: 1.0

#include <stdio.h>

main()
{
   int numero; // Declaramos que vamos a usar una variable que se llama numero y tiene tipo int (entero).
```

Si queremos que *printf* imprima % tendremos que escribir %%.

Casos prácticos

```
printf("Hola, soy un programa muy educado.\n");
printf("Por favor, introduce un número y pulsa enter: ");
scanf("%d", &numero); // scanf es una subrutina que lee de teclado. También está programada en la librería
                      // stdio y tiene dos parámetros, separados por una coma. En el primero ("%d") le decimos
                      // que lea un entero y, en el segundo (&numero), que lo guarde en la variable numero.
printf("El numero es: %d\n", numero);
                      // printf, además de imprimir cadenas fijas, también puede imprimir el contenido de
                      // variables. En este caso le pasamos dos parámetros. En el primero le decimos qué debe
                      // imprimir, incluyendo un %d para ordenar que además imprima una variable entera, y en el
                      // segundo, que la variable que tiene que imprimir es la que se llama numero.
}
```

práctico 3:

Como se observa en el caso práctico 3, es necesario declarar siempre una variable antes de utilizarla. La declaración no es más que especificar qué variables vamos a usar, dándoles un nombre y un tipo. Evidentemente, el nombre debe ser único y, además, no puede coincidir con el de ninguna palabra reservada del lenguaje. El tipo tiene que ser uno de los que permite el lenguaje (de momento sólo se ha visto el tipo *int*). Hay que tener en cuenta, también, que todas las declaraciones de variables deben estar al principio, detrás de *{* (apertura de llave) y antes de las instrucciones propiamente dichas, aunque hay una excepción, las variables globales.

En el ejemplo hemos introducido una nueva forma de trabajar con *printf*, esto es, escribiendo, además de una cadena, el contenido de una variable. Para ello hemos utilizado, dentro de la cadena, una secuencia de control, *%d*, que indica que en un segundo parámetro especificaremos el nombre de la variable entera a imprimir. Además, hemos incluido también la subrutina *scanf* que lee de teclado. El primer parámetro contiene una cadena con una indicación de lo que vamos a leer (y cuyo formato coincide con el de *printf*, así que *%d* indica que vamos a leer un entero) y el segundo parámetro es la variable en la que vamos a guardar el valor leído. Antes del nombre de variable aparece el signo *&*. Este signo es necesario porque *scanf*, a diferencia de *printf*, va a modificar el contenido de la variable y, por lo tanto, no debe conocer su valor, sino su dirección en la memoria. Veremos este punto con más detalle en el apartado de subrutinas; de momento, lo importante es recordar que un *&* delante de un nombre significa dirección y que para que *scanf* pueda guardar los valores que lee en las variables que le digamos debe conocer la dirección de las mismas.

> Si queremos imprimir acentos con printf, deberemos averiguar el código ASCII de la vocal acentuada e imprimirlo directamente como \xNN, donde NN es el código ASCII en hexadecimal del carácter a imprimir. Ej: printf("\xA0"); imprime una á.

> Se dice que una subrutina **devuelve** un valor cuando su resultado puede asignarse a una variable o bien imprimirse directamente. Ej: a=sizeof(b); a valdría 2, 4 u 8 según el procesador. Las subrutinas que devuelven un valor se llaman **funciones**.

Casos prácticos

4
```
// Programa: Sumar.c
// Utilidad: Lee dos números enteros de teclado e imprime la suma por pantalla.
// Programador: Yo mismo.
// Fecha: Hoy mismo.
// Versión: 1.0

#include <stdio.h>

main()
{
  int operando1; // Podemos declarar tantas variables como queramos, una por línea.
  int operando2, resultado; // O bien podemos declarar varias del mismo tipo en la misma línea separadas por comas.

  printf("Hola, soy un programa muy educado.\n");
  printf("Por favor, introduce un numero y pulsa enter: ");
  scanf("%d", &operando1);
  printf("Por favor, introduce otro numero y pulsa enter: ");
  scanf("%d", &operando2);

  resultado=operando1+operando2; // Asignamos a la variable resultado la suma de operando1 y operando2.
            // Fijémonos que C efectúa la operación tal y como nos parece lógico. Primero suma los operandos y a
            // continuación asigna el resultado a la variable que hay a la izquierda del igual. Esto permite usar
            // expresiones como resultado=resultado+operando1.

  printf("La suma de %d y %d da como resultado %d.\n", operando1, operando2, resultado);
                          // printf puede imprimir tantas variables como queramos de una vez.
}
```

Casos prácticos

5
```
// Programa: Tamano.c
/* Utilidad: Imprime el tamaño de una variable entera
en pantalla.*/
// Programador: Yo mismo.
// Fecha: Hoy mismo.
// Versión: 1.0

#include <stdio.h>

main()
{
  int variableentera;

  printf("El tamaño de una variable entera es:  %d\n",
sizeof(variableentera));
     /* Fijémonos que, en vez de asignarlo a una varia-
ble, se puede utilizar el valor devuelto por una subru-
tina como parámetro para otra.*/
}
```

Nota: la cadena de la instrucción printf (la que va entre comillas) no se puede dividir en varias líneas. En los casos prácticos, por cuestiones de edición, pueden aparecer divididas, pero no es una sintaxis correcta. Debemos recordar corregirlo en nuestro programa.

Operación	Signo	Ejemplo de uso	Valor inicial de a	Valor de b	Valor final de a
Asignación	=	a=10;	Cualquiera	–	10
Suma	+	a=a+b;	5	3	8
Resta	–	a=a-b;	5	3	2
Multiplicación	*	a=a*b;	5	3	15
División	/	a=a/b;	5	3	1
Resto	%	a=a%b;	5	3	2
Incremento	++	a++;	5	–	6
Decremento	—	a—;	5	–	4

Tabla 12.1. *Operaciones enteras y ejemplos.*

Tamaño en bytes	Valor mínimo	Valor máximo
1	$-128\,(-2^{(8-1)})$	$127\ (2^{(8-1)}-1)$
2	$-32\,768\,(-2^{(16-1)})$	$32767(2^{(16-1)}-1)$
4	$-2\,147\,483\,648\,(-2^{(32-1)})$	$2147483647(2^{(32-1)}-1)$
8	$(-2^{(64-1)})$	$(2^{(64-1)}-1)$

Tabla 12.2. *Posibles valores enteros según su tamaño.*

Modificador	Ejemplo	Efecto
Short	short int a; short a;	La variable se hace "corta", es decir, es un entero de solo 2 bytes.
Long	long int a; long a;	La variable se hace "larga", es decir, es un entero de 8 bytes (en función del procesador, seguirá siendo de 4).
Unsigned	unsigned int a; unsigned a;	La variable no tiene signo, es decir, es un número natural (entero no negativo).
Signed	signed int a;	La variable tiene signo. En realidad no tiene ningún efecto porque es la opción por defecto.

Tabla 12.3. *Modificadores de tamaño de las variables enteras.*

Un programa, además de leer y escribir variables, también puede realizar operaciones con ellas. Por ejemplo, puede sumar dos variables enteras y guardar el resultado en una tercera variable.
Las variables enteras admiten varias operaciones además de la suma: la resta (-), la multiplicación (*), la división (/) y el resto (%). La tabla 1 muestra un resumen y varios ejemplos. Vamos a resolver las actividades 5 y 6 y, además, para ver un ejemplo de error en ejecución haremos la actividad 7.

Otra característica que hay que tener en cuenta en el caso de las variables es el rango de representación. Como hemos señalado, las variables no son más que un espacio de memoria donde se almacena un dato. Como el espacio de memoria es limitado, el tamaño del dato que se puede representar también lo será. Por ejemplo, si el tamaño de la memoria es de 1 byte, como mucho podremos representar un número entero que quepa en un byte, es decir, un número entre -128 y 127 (lo cual es bastante poco). Actualmente, la mayoría de ordenadores usan variables enteras de 4 u 8 bytes, aunque todavía pueden encontrarse algunos que las usan de 2.
Una forma fácil de averiguar el tamaño de una variable es ejecutando la subrutina *sizeof()* (tamaño de, en inglés). La subrutina *sizeof* **devuelve** el tamaño en bytes de su parámetro. En la tabla 2 puede verse una correspondencia entre el tamaño de una variable entera y los valores que puede almacenar, y en el caso práctico 5 un ejemplo de uso.
Las variables enteras pueden incorporar modificadores que varíen su tipo. Estos modificadores se ponen en la declaración delante del tipo (es decir, delante de la palabra *int*) y se resumen en la tabla 12.3 (página 348). También podemos comprobarlo resolviendo la actividad 8.
¿Qué sucede cuando intentamos dar a una variable un valor mayor que el que puede aceptar por su tamaño? En general, la variable acepta el valor y lo guarda, pero reduciéndolo a un valor que pueda almacenar. Para reducirlo, cuando llega al máximo, vuelve a empezar por abajo y da tantas vueltas como sea necesario. Veamos un ejemplo: si el valor máximo es 127 y el mínimo -128 e intentamos que almacene un 128 (uno más del máximo), almacenará el -128 (el primero por abajo). Si intentamos que almacene el 129, guardará el -127, etc. En realidad, lo que hace es truncar el valor que intentamos que almacene en la cantidad de bits que puede almacenar. Ocurre lo mismo si el número mayor del rango es el resultado de una operación; entonces, sólo se

Casos prácticos

6

```
// Programa: Precedencias.c
// Utilidad: Ejemplifica el orden en el que se realizan las operaciones.
// Programador: Yo mismo.
// Fecha: Hoy mismo.
// Versión: 1.0

#include <stdio.h>

main()
{
  int a=5, b=3, c=2; // C permite inicializar las variables en la declaración, simplemente asignándoles un valor.

  printf("El resultado de a+b*c es:  %d\n", a+b*c);  // La multiplicación se hace antes que la suma...
  printf("El resultado de (a+b)*c es: %d\n", (a+b)*c); // a no ser que se pongan paréntesis.

  printf("El resultado de a*b/c es: %d\n", a*b/c);
          // Cuando todos los operadores son iguales, C opera de izquierda a derecha.
  printf("El resultado de a*(b/c) es: %d\n", a*(b/c));
          // Además, como la división es entera, los decimales se pierden al dividir.

          // Otro caso curioso es el de los operadores ++ y --
  printf("Si a vale %d y hacemos a++ (%d), a valdra %d despues de asignar.\n", a, a++, a);
    printf("En cambio si a vale %d y hacemos --a (%d), a valdra %d antes de asignar.\n", a, --a, a);
     }
```

Inicializar es asignar un primer valor a una variable.

Precedencia de operadores	Asociatividad (orden de la operación)
(,)	De izquierda a derecha
-, ++, —	De derecha a izquierda
*, /, %	De izquierda a derecha
+, -	De izquierda a derecha

Tabla 12.4. *Procedencia de operadores y asociatividad.*

Operación	Equivale a:
a=a+b++;	a=a+b; b=b+1;
a+=b;	a=a+b;
a*=a;	a=a*a;
a-=b;	a=a-b;
a/=b—;	a=a/b; b=b-1
a+=++b;	b=b+1; a=a+b;

Tabla 12.5. *Ejemplos de operaciones complejas.*

12.5 Variables de caracteres

guardará la parte que cabe. Este efecto se llama desbordamiento y puede ser muy importante. Para verlo, vamos a resolver la actividad 9.

Más sobre operaciones aritméticas

Lo que hemos visto hasta ahora es suficiente para realizar cual-

0	nulo	32	espacio	64	@	96	`	128	Ç	160	á	192	└	224	α
1	principio de encabezado	33	!	65	A	97	a	129	ü	161	í	193	┴	225	ß
2	inicio de texto	34	"	66	B	98	b	130	é	162	ó	194	┬	226	Γ
3	fin del texto	35	#	67	C	99	c	131	â	163	ú	195	├	227	π
4	fin de transmisión	36	$	68	D	100	d	132	ä	164	ñ	196	─	228	Σ
5	consulta	37	%	69	E	101	e	133	à	165	Ñ	197	┼	229	σ
6	reconocimiento	38	&	70	F	102	f	134	å	166	ª	198	╞	230	µ
7	campana	39	'	71	G	103	g	135	ç	167	º	199	╟	231	τ
8	retroceso	40	(	72	H	104	h	136	ê	168	¿	200	╚	232	Φ
9	tabulación horizontal	41	)	73	I	105	i	137	ë	169	⌐	201	╔	233	Θ
10	avance de línea/nueva línea	42	*	74	J	106	j	138	è	170	¬	202	╩	234	Ω
11	tabulación vertical	43	+	75	K	107	k	139	ï	171	½	203	╦	235	δ
12	avance de página/nueva página	44	,	76	L	108	l	140	î	172	¼	204	╠	236	∞
13	retorno de carro	45	-	77	M	109	m	141	ì	173	¡	205	═	237	φ
14	desplazamiento hacia fuera	46	.	78	N	110	n	142	Ä	174	«	206	╬	238	ε
15	desplazamiento hacia dentro	47	/	79	O	111	o	143	Å	175	»	207	╧	239	∩
16	escape de vínculo de datos	48	0	80	P	112	p	144	É	176	▒	208	╨	240	≡
17	control de dispositivo 1	49	1	81	Q	113	q	145	æ	177	▓	209	╤	241	±
18	control de dispositivo 2	50	2	82	R	114	r	146	Æ	178	▒	210	╥	242	≥
19	control de dispositivo 3	51	3	83	S	115	s	147	ô	179	▒	211	╙	243	≤
20	control de dispositivo 4	52	4	84	T	116	t	148	ö	180	▒	212	╘	244	⌠
21	confirmación negativa	53	5	85	U	117	u	149	ò	181	▒	213	╒	245	⌡
22	inactividad síncrona	54	6	86	V	118	v	150	û	182	▒	214	╓	246	÷
23	fin del bloque de transmisión	55	7	87	W	119	w	151	ù	183	▒	215	╫	247	≈
24	cancelar	56	8	88	X	120	x	152	ÿ	184	▒	216	╪	248	°
25	fin del medio	57	9	89	Y	121	y	153	Ö	185	▒	217	┘	249	∙
26	sustitución	58	:	90	Z	122	z	154	Ü	186	▒	218	┌	250	·
27	escape	59	;	91	[	123	{	155	¢	187	▒	219	▒	251	√
28	separador de archivos	60	<	92	\	124	|	156	£	188	▒	220	▒	252	n
29	separador de grupos	61	=	93	]	125	}	157	¥	189	▒	221	▒	253	²
30	separador de registros	62	>	94	^	126	~	158	Pts	190	▒	222	▒	254	■
31	separador de unidades	63	?	95	_	127	SUPR	159	ƒ	191	▒	223	▒	255	

Fig. 12.8. *Tabla ASCII.*

quier operación con números enteros. Sin embargo, en muchos casos interesa hacer operaciones compuestas por dos o más operaciones simples (por ejemplo, sumar dos números y multiplicarlos por un tercero). En estos casos, basta con escribir todas las operaciones en una misma línea, pero para hacerlo correctamente debemos tener en cuenta el orden de precedencia, es decir, qué operaciones se ejecutan antes y cuáles después. La tabla 12.4 contiene el orden de precedencia de las operaciones en C, pero el efecto se ve mejor con un sencillo programa. Después de ver el caso práctico 6 intentaremos resolver la actividad 10.

Por último, la tabla 5 contiene algunos ejemplos de formas curiosas de indicar determinadas operaciones en C. Como se aprecia, en C hay muchas maneras de obtener los mismos resultados. Existe una cierta tendencia a "reinventar la rueda", aunque por claridad lo mejor suele ser usar la forma más clásica (y clara) de hacer las cosas. Contra lo que mucha gente suele pensar, los programas NO son más rápidos si se escriben de forma más críptica. Otra razón por la que resulta importante escribir los programas de forma clara y ordenada es la facilidad para detectar los errores. Un programa en el que resulte difícil encontrar errores se considera peor escrito que otro que hace lo mismo pero es más fácil de corregir.

Casos prácticos

7
```
// Programa: Caracteres.c
// Utilidad: Imprime el tamaño y contenido de un carácter.
// Programador: Yo mismo.
// Fecha: Hoy mismo.
// Versión: 1.0

#include <stdio.h>

main()
{
  char car='A';

   printf("El tamanyo de una variable caracter es: %d\n", sizeof(car));
  printf("El contenido de car como numero es: %d\n", car);
  printf("Pero el contenido de car como caracter es: %c\n", car);
    /* Una variable de tipo char puede contener un carácter o un entero de 1 byte, todo depende de cómo la usemos.*/
}
```

Casos prácticos

8
```
// Programa: Leeyescribecar.c
// Utilidad: Lee un carácter de teclado y lo imprime por pantalla.
// Programador: Yo mismo.
// Fecha: Hoy mismo.
// Versión: 1.0

#include <stdio.h>

main()
{
  char letra;

   printf("Hola, sigo siendo un programa muy educado.\nPor favor, introduce una letra y pulsa enter: ");
  scanf("%c", &letra);
  printf("La letra es '%c' y su codigo ASCII es %d\n", letra, letra);
   /* Dentro de una cadena, podemos poner comillas simples (') directamente, pero si quisiéramos imprimir comillas dobles (") deberíamos escribir \".*/
}
```

12.6 Variables reales

Tipo	Valor mínimo	Valor máximo	Menor valor positivo	Mayor valor negativo
float	–3.4028232E+38	+3.4028232E+38	+1.1754945E–38	–1.1754945E–38
double	–1.79769313E+308	+1.79769313E+308	+2.2250738585E–308	–2.2250738585E–308

Tabla 12.6. *Rango de las variables reales.*

Otras variables muy utilizadas en C son las de tipo carácter. Estas variables, como hemos visto en el tema 1, contienen un valor numérico entre 0 y 255 (ocupan un solo byte) y son útiles porque, como se observa a través de la tabla ASCII (Figura 12.8), cada valor corresponde a un carácter alfanumérico. Estas variables se declaran de tipo *char*. El caso práctico 7 muestra cómo trabajan los caracteres en C. También podemos hacer la actividad 11.

Fijémonos en el caso práctico 7. En la segunda línea, *printf*

Casos prácticos

9
```
// Programa: Sumarreales.c
// Utilidad: Lee dos números reales de teclado e imprime la suma en pantalla.
// Programador: Yo mismo.
// Fecha: Hoy mismo.
// Versión: 1.0

#include <stdio.h>

main()
{
  float operando1, operando2;

  printf("Hola, soy un programa muy educado.\nPor favor, introduce un numero y pulsa enter: ");
  scanf("%f", &operando1);
  printf("Por favor, introduce otro numero y pulsa enter: ");
  scanf("%f", &operando2);

  printf("La suma de %f y %f da como resultado %f\n", operando1, operando2, operando1+operando2);
}
```

12.7 Conversión de tipos

imprime un número, el 65, que es el valor de la "A" de la tabla ASCII, mientras que en la tercera línea imprime el carácter "A". Sin embargo, en ambas instrucciones le decimos a *printf* que imprima el contenido de la variable *a*. ¿Cuál es entonces la diferencia? La diferencia está en la secuencia de control dentro del primer parámetro.

La primera secuencia, *%d*, es la que ya conocemos del apartado anterior e indica que hay que imprimir un entero. La segunda secuencia de control, *%c*, indica que hay que imprimir un carácter, así que eso es lo que se imprime. Igualmente, si quisiéramos crear un programa que leyera un carácter de teclado, deberíamos indicar *%c* en la secuencia de control de *scanf*, tal y como sucede en el caso práctico 8.

Casos prácticos

10
```
// Programa: Dividecondecimal.c
// Utilidad: Lee dos números enteros y muestra el resultado de su división con decimales.
// Programador: Yo mismo.
// Fecha: Hoy mismo.
// Versión: 1.0

#include <stdio.h>

main()
{
  int operando1=5, operando2=2;
```

```
    float resultado;

    resultado=operando1/operando2; // Si hacemos la operación así, C hará la división entera.
    printf("La division entera de %d y %d da como resultado %f\n", operando1, operando2, resultado);

    resultado=(float)operando1/(float)operando2;
    printf("La division real de %d y %d da como resultado %f\n", operando1, operando2, resultado);
}
```

El tercer tipo de variables más usado son las variables reales. Como su mismo nombre indica, las variables reales almacenan un valor real, es decir, con decimales. Existen dos tipos de variables reales, las de simple precisión (llamadas *float*), que ocupan 4 bytes, y las de doble precisión (llamadas *double*), que ocupan 8 bytes. Como se observa en la tabla 6, las variables reales admiten un rango de representación muy amplio. Además de las dos columnas típicas (valor mínimo y máximo) también hemos incluido el mayor valor negativo y el menor valor positivo. Esto es así porque como las variables reales representan números reales, además de estar limitadas por arriba y por abajo, también lo están en el valor más pequeño que pueden guardar sin llegar a 0. Es decir, en una variable real simple (*float*) no podríamos almacenar, por ejemplo, el número $1*10^{-40}$.

Las variables reales admiten los mismos tipos de operaciones que las variables enteras, menos la operación resto, claro, aunque su resultado será real y, por lo tanto, debe ser asignado a una variable real. Si queremos trabajar con variables reales y las subrutinas *printf* y *scanf* debemos tener en cuenta que un *float* se lee y escribe con *%f*, mientras que un *double* se lee y escribe con *%lf*. También es importante tener en cuenta que el separador de decimales es el punto (.). Después de mirar el caso práctico 9 vamos a resolver la actividad 12.

Casos prácticos

11
```
// Programa: Tiposytipos.c
// Utilidad: Da tres resultados diferentes de la misma operación (a/b*c).
// Programador: Yo mismo.
// Fecha: Hoy mismo.
// Versión: 1.0

#include <stdio.h>

main()
{
  int a,b,c;
  int d;
  float e;
  double f;

  printf("Hola, sigo siendo un programa muy educado.\n");
  printf("Por favor, introduce un numero y pulsa enter: ");
  scanf("%d", &a);
  printf("Por favor, introduce otro numero y pulsa enter: ");
   scanf("%d", &b);
      printf("Por favor, introduce un tercer numero y pulsa enter: ");
```

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

    d=a/b;
    printf("La operacion con enteros da %d\n",d*c);
    e=(float)a/(float)b;
    printf("La operacion con reales en simple precision da %.20f\n",e*(float)c);
    f=(double)a/(double)b;
    printf("La operacion con reales en doble precision da %.20lf\n",f*(double)c);
}
```

Con lo que hemos visto hasta el momento podemos crear programas que ejecuten operaciones con cualquiera de los tipos básicos de lenguaje C. Sin embargo, hasta ahora, todos los programas que hemos analizado realizan operaciones entre operandos del mismo tipo. A veces sin embargo, es necesario realizar operaciones entre operandos de distinto tipo, o efectuar un cálculo y guardar el

12.8 Constantes

resultado en un tipo diferente al original. Por ejemplo, podemos tener un par de números enteros y querer averiguar el resultado de su división con decimales. En estos casos se efectúa lo que se conoce como conversión de tipos (o *casting*, en argot). El *casting* simplemente consiste en especificar al compilador que, después de leer el contenido de una variable, debe transformarlo en un nuevo tipo. En el programa, se indica anteponiendo a la variable el tipo al que hay que convertirla, entre paréntesis. Se puede ver un ejemplo del resultado obtenido y sus particularidades en el caso práctico 10.

Casos prácticos

12
```
// Programa: Convierteaeuros.c
/* Utilidad: Lee una cantidad de pesetas y la convierte
a euros.*/
// Programador: Yo mismo.
// Fecha: Mismamente hoy.
// Versión: 1.0

#include <stdio.h>

#define EUROENPESETAS 166.66
        /* Se pueden definir constantes antes o des-
pués de los include, según convenga. */

main()
{
  int pesetas;

  printf("Escribe una cantidad de pesetas: ");
  scanf("%d",&pesetas);
   printf("%d pesetas son %.2f euros.\n",pesetas,
(float)pesetas/EUROENPESETAS);
 /* El .2 del %f significa que queremos que la cantidad
se imprima con 2 decimales (céntimos).*/
}
```

Como se puede ver en el ejemplo anterior, en el primer caso no hemos

Ejercicios propuestos

1 Para obtener un fichero fuente necesitas una aplicación:
- a) De tipo Word (editor de textos de Microsoft con muchas funciones de edición).
- b) De tipo editor de texto plano, como el bloc de notas de Windows.
- c) De tipo como el compilador de C que incluye MS Visual Studio, no el editor.
- d) Como el editor de textos del OpenOffice (editor de textos de software libre con muchas funciones de edición).

2 El fichero fuente contiene:
- a) Instrucciones en lenguaje máquina.
- b) Las instrucciones que vienen con el compilador y que se incluyen en forma de librerías.
- c) Las instrucciones del programa en lenguaje de alto nivel siguiendo la sintaxis del lenguaje.
- d) Las instrucciones de utilización del programa una vez compilado.

3 Puedes tener un programa sin ninguna variable:
- a) Cierto.
- b) Falso.

4 Los errores sintácticos:
- a) No permiten que se genere el fichero objeto.
- b) No permiten que se enlacen (linken) los programas.
- c) No permiten que se ejecuten los programas.
- d) Ninguna de las anteriores.

5 Los errores sintácticos:
- a) No tienen ninguna indicación.
- b) Aparecen como mensajes en el linkado sin referencias claras al lugar donde se producen.
- c) Contienen referencias al lugar donde el compilador lo encuentra y el tipo de error que es.
- d) Todas las anteriores.

6 Cuando has de corregir errores en un fichero fuente de un programa:
- a) Lo mejor es empezar por el último porque así no afecta a los demás.
- b) Da lo mismo por cual empieces porque al final no han de quedar errores.
- c) Lo más importante es ir reduciendo el número de errores en cada corrección. Si cambias una cosa y salen más errores lo has hecho mal seguro.
- d) Lo mejor es empezar por el primero y así los errores que dependan de este se corrigen o aparecen.

7 Las variables enteras:
- a) Sólo se pueden declarar una por línea:
- b) Se pueden declarar una o varias por línea.
- c) Sólo se pueden declarar en una línea.
- d) Se pueden declarar en varias líneas pero sólo una por línea.

8 Los identificadores de una variable:
- a) Han de ser únicos.
- b) No pueden coincidir con las palabras reservadas.
- c) Conviene que reflejen la función de la variable.
- d) Todas las anteriores.

9 Los identificadores de variables distinguen entre letras mayúsculas y minúsculas
- a) Cierto.
- b) Falso.

10 El código ASCII:
- a) Representa con un byte (8 bits) 256 caracteres alfanuméricos.
- b) Representa las letras y los números.
- c) Representa sólo las variables que contienen caracteres. Para los números utilizamos los int y los float.
- d) Es un código interno del sistema informático empleado exclusivamente para la transmisión de información por el puerto serie.

11 Si tienes que guardar la información del saldo de una cuenta bancaria en euros el tipo de variable más adecuado es:
- a) int
- b) char
- c) long int
- d) double

12 Si has guardado un valor de 2.6 en una variable float y lo asignas a una variable de tipo entero el valor guardado es:
- a) 2
- b) 2.6
- c) 3
- d) 26

13 Si tienes dos variables de tipo entero que contiene los valores 5 y 2, si las divides entre ellas y el resultado lo asignas a una variable de tipo float obtienes:
- a) 2
- b) 2.5
- c) 3
- d) Un error. No se pueden asignar de forma automática diferentes tipos.

14 Al declarar una constante en un programa:
a) Su valor depende del momento de la ejecución.
b) No cambia mientras no le asignamos un nuevo valor en el programa. De ahí el nombre.
c) No cambia.
d) Al declararlo fuera del main no puede utilizarse en el programa. Sirve sólo como referencia.

15 Si tienes una constante entera, ¿a que tipo de variable la puedes asignar?
a) A todas, es un número.
b) A carácthereres solo si el código ASCII coincide con una letra.
c) A reales solo añadiendole ".0" en la asignación.
d) Solo la podemos asignar a variables enteras.

Actividades

1 Escribe, compila y ejecuta en tu entorno habitual de trabajo el programa del caso práctico 1.

2 Crea un programa (o modifica el de la actividad anterior) que se presente e imprima tu nombre. Intenta que lo escriba todo en la misma línea o en líneas diferentes. ¿Para qué sirve el *\n* que hay al final de la cadena que enviamos a *printf*?

3 Ahora que el programa de la actividad 1 ya funciona, vuelve a abrir el fichero que lo contiene y elimina el punto y coma (*;*) del final de la sentencia *printf*. ¿Qué error indica el compilador? Vuelve a escribir el punto y coma (*;*) y borra las segundas comillas (") de dentro de los paréntesis de la función *printf*. ¿El error obtenido es el mismo?

4 A partir del primer programa, una vez funcione, cambia *printf* por *print*. Fíjate que, aunque sea un error de escritura, como en el caso anterior, ahora se produce en otro momento del proceso y es indicado por el linkador. Lee el mensaje de vuestro sistema e intenta comprenderlo. ¿Ha señalado el error en el punto en el que has alterado el programa?

5 Escribe el programa del caso práctico *Leeyescribe.c* y ejecútalo. Si estás trabajando en Windows acuérdate de incluir la orden *system("PAUSE")* para poder ver el resultado.

6 Escribe un programa que lea dos números de teclado e imprima en pantalla su suma, resta, multiplicación, división y resto. Intenta hacerlo usando sólo tres variables enteras.

7 Vamos a ver ahora un error de ejecución. Una vez te funcione el programa de la actividad anterior, ejecútalo e introduce los valores 7 y 0. Tu programa no acabará sino que dará un error, debido a la división por 0. Se trata de un error de ejecución y la única manera de evitarlo es comprobar, antes de dividir, que el segundo número introducido no sea 0. Más adelante ya veremos cómo hacerlo.

8 Escribe el programa *tamano.c* y modifícalo para saber la longitud, en tu procesador, de una variable *short* y de una variable *long*.

9 Prueba ahora de nuevo el código de la actividad 5. ¿Qué sucede si escribes el número 2147483647? ¿Y si escribes el 2147483648? ¿Y si sigues aumentando la cifra? ¿A qué se debe?

10 ¿Qué resultados crees que escribirá el programa del caso práctico *Precedencias.c*? Pruébalo. ¿Has acertado? Intenta explicar los fallos que has observado.

11 Escribe el programa del caso práctico *Caracteres.c* y prueba a asignar otros valores iniciales a la variable *car* (por ejemplo, del 160 al 165). ¿Qué letras imprime el programa? ¿Por qué?

12 Modifica el ejercicio del caso práctico *Sumarreales.c* para que funcione con números de doble precisión.

13 Crea un programa que divida y multiplique 3 variables enteras (a, b y c) leídas de teclado. La operación que debe realizar es: a/b*c. El programa debe dar 2 resultados: el resultado de las operaciones enteras y el resultado operando con variables reales. Prueba con los valores a=10, b=3 y c=3. Explica la diferencia entre los dos resultados. ¿Cómo explicas que el segundo resultado no sea 9.999999?

14 Escribe el programa del caso práctico *Tiposytipos.c* y pruébalo con los mismos valores de la actividad 11. ¿Qué valores da esta vez? ¿A qué se debe la diferencia?

15 Elabora un programa que transforme euros en pesetas. Ten en cuenta que los euros pueden tener decimales (los céntimos) y las pesetas no.