# Fundamentos de programación en C

Este texto es una introducción a la programación de ordenadores en lenguaje C.

Se ha revisado con la intención que su nivel sea el razonable para una asignatura de "Fundamentos de programación" o similar, aunque quizá algunos centros (especialmente universitarios) exijan un nivel más alto que el que se cubre.

Está organizado de una forma ligeramente distinta a los libros de texto "convencionales", procurando incluir ejercicios prácticos lo antes posible, para evitar que un exceso de teoría en los primeros temas haga el texto pesado de seguir.

Aun así, este texto no pretende "sustituir a un profesor", sino servir de apoyo para que los alumnos no pierdan tiempo en tomar apuntes. Pero es trabajo del profesor aclarar las dudas que surjan y proponer muchos más ejercicios que los que figuran aquí.

Este texto ha sido escrito por Nacho Cabanes. Si quiere conseguir la última versión, estará en mi página web:

www.nachocabanes.com



Para cualquier sugerencia, no dude en contactar conmigo a través de mi web. Si se trata de dudas sobre programación, suele ser fácil localizarme en los foros de

www.AprendeAProgramar.com

Revisión actual: 0.95

# Contenido

# 0. Conceptos básicos sobre programación

Un **programa** es un conjunto de órdenes para un ordenador.

Estas órdenes se le deben dar en un cierto **lenguaje,** que el ordenador sea capaz de comprender.

El problema es que los lenguajes que realmente entienden los ordenadores resultan difíciles para nosotros, porque son muy distintos de los que nosotros empleamos habitualmente para hablar. Escribir programas en el lenguaje que utiliza internamente el ordenador (llamado "**lenguaje máquina**" o "código máquina") es un trabajo duro, tanto a la hora de crear el programa como (especialmente) en el momento de corregir algún fallo o mejorar lo que se hizo.

Por eso, en la práctica se emplean lenguajes más parecidos al lenguaje humano, llamados "lenguajes de **alto nivel**". Normalmente, estos son muy parecidos al idioma inglés, aunque siguen unas reglas mucho más estrictas.

## *0.1. Lenguajes de alto nivel y de bajo nivel.*

Vamos a ver en primer lugar algún ejemplo de lenguaje de alto nivel, para después comparar con lenguajes de bajo nivel, que son los más cercanos al ordenador.

Uno de los lenguajes de alto nivel más sencillos es el lenguaje BASIC. En este lenguaje, escribir el texto Hola en pantalla, sería tan sencillo como usar la orden

```
PRINT "Hola"
```

Otros lenguajes, como Pascal, nos obligan a ser algo más estrictos, pero a cambio hacen más fácil descubrir errores:

```
program Saludo;

begin
  write('Hola');
end.
```

El equivalente en lenguaje C resulta algo más difícil de leer, por motivos que iremos comentando un poco más adelante:

```
#include <stdio.h>

int main()
{
  printf("Hola");
  return 0;
}
```

Los lenguajes de bajo nivel son más cercanos al ordenador que a los lenguajes humanos. Eso hace que sean más difíciles de aprender y también que los fallos sean más difíciles de descubrir y corregir, a cambio de que podemos optimizar al máximo la velocidad (si sabemos cómo), e incluso llegar a un nivel de control del ordenador que a veces no se puede alcanzar con otros lenguajes. Por ejemplo, escribir Hola en lenguaje ensamblador de un ordenador equipado con el sistema operativo MsDos y con un procesador de la familia Intel x86 sería algo como

```
dosseg
.model small
.stack 100h

.data
hello_message db 'Hola',0dh,0ah,'$'

.code
main  proc
      mov    ax,@data
      mov    ds,ax

      mov    ah,9
      mov    dx,offset hello_message
      int    21h

      mov    ax,4C00h
      int    21h
main  endp
end   main
```

Resulta bastante más difícil de seguir. Pero eso todavía no es lo que el ordenador entiende, aunque tiene una equivalencia casi directa. Lo que el ordenador realmente es capaz de comprender son secuencias de ceros y unos. Por ejemplo, las órdenes "mov ds, ax" y "mov ah, 9" (en cuyo significado no vamos a entrar) se convertirían a lo siguiente:

```
1000 0011 1101 1000 1011 0100 0000 1001
```

(Nota: los colores de los ejemplos anteriores son una ayuda que nos dan algunos entornos de programación, para que nos sea más fácil descubrir errores).

## *0.2. Ensambladores, compiladores e intérpretes*

Está claro entonces que las órdenes que nosotros hemos escrito (lo que se conoce como "programa **fuente**") deben convertirse a lo que el ordenador comprende (obteniendo el "programa **ejecutable**").

Si elegimos un lenguaje de bajo nivel, como el ensamblador (en inglés *Assembly*, abreviado como *Asm*), la traducción es sencilla, y de hacer esa traducción se encargan unas herramientas llamadas **ensambladores** (en inglés *Assembler*).

Cuando el lenguaje que hemos empleado es de alto nivel, la traducción es más complicada, y a veces implicará también recopilar varios fuentes distintos o incluir posibilidades que se encuentran en bibliotecas que no hemos preparado nosotros. Las herramientas encargadas de todo esto son los **compiladores**.

El programa ejecutable obtenido con el compilador o el ensamblador se podría hacer funcionar en otro ordenador similar al que habíamos utilizado para crearlo, sin necesidad de que ese otro ordenador tenga instalado el compilador o el ensamblador.

Por ejemplo, en el caso de Windows (y de MsDos), y del programa que nos saluda en lenguaje Pascal, tendríamos un fichero fuente llamado SALUDO.PAS. Este fichero no serviría de nada en un ordenador que no tuviera un compilador de Pascal. En cambio, después de compilarlo obtendríamos un fichero SALUDO.EXE, capaz de funcionar en cualquier otro ordenador que tuviera el mismo sistema operativo, aunque no tenga un compilador de Pascal instalado.

Un **intérprete** es una herramienta parecida a un compilador, con la diferencia de que en los intérpretes no se crea ningún "programa ejecutable" capaz de funcionar "por sí solo", de modo que si queremos distribuir nuestro programa a alguien, deberemos entregarle el programa fuente y también el intérprete que es capaz de entenderlo, o no le servirá de nada. Cuando ponemos el programa en funcionamiento, el intérprete de encarga de convertir el programa en lenguaje de alto nivel a código máquina, orden por orden, justo en el momento en que hay que procesar cada una de las órdenes.

Para algunos lenguajes, es frecuente encontrar compiladores pero no suele existir intérpretes. Es el caso del lenguaje C, de Pascal y de C++, por ejemplo. En cambio, para otros lenguajes, lo habitual es trabajar con intérpretes y no con compiladores, como ocurre con Python, Ruby y PHP.

Además, hoy en día existe algo que parece intermedio entre un compilador y un intérprete: Existen lenguajes que no se compilan para obtener un ejecutable para un ordenador concreto, sino un ejecutable "genérico", que es capaz de funcionar en distintos tipos de ordenadores, a condición de que en ese ordenador exista una "**máquina virtual**" capaz de entender esos ejecutables genéricos. Esta es la idea que se aplica en Java: los fuentes son ficheros de texto, con extensión ".java", que se compilan a ficheros ".class". Estos ficheros ".class" se podrían llevar a cualquier ordenador que tenga instalada una "máquina virtual Java" (las hay para la mayoría de sistemas operativos). Esta misma idea se sigue en el lenguaje C#, que se apoya en una máquina virtual llamada "Dot Net Framework" (algo así como "armazón punto net").

## *0.3. Pseudocódigo*

A pesar de que los lenguajes de alto nivel se acercan al lenguaje natural, que nosotros empleamos, es habitual no usar ningún lenguaje de programación concreto cuando queremos plantear los pasos necesarios para resolver un problema, sino emplear un lenguaje de programación ficticio, no tan estricto, muchas veces escrito incluso en español. Este lenguaje recibe el nombre de **pseudocódigo.**

Esa secuencia de pasos para resolver un problema es lo que se conoce como **algoritmo** (realmente hay alguna condición más, por ejemplo, debe ser un número finito de pasos). Por tanto, un programa de ordenador es un algoritmo expresado en un lenguaje de programación.

Por ejemplo, un algoritmo que controlase los pagos que se realizan en una tienda con tarjeta de crédito, escrito en pseudocódigo, podría ser:

```
Leer banda magnética de la tarjeta
Conectar con central de cobros
Si hay conexión y la tarjeta es correcta:
   Pedir código PIN
  Si el PIN es correcto
    Comprobar saldo_existente
    Si saldo_existente > importe_compra
      Aceptar la venta
      Descontar importe del saldo.
    Fin Si
  Fin Si
Fin Si
```

**Ejercicios propuestos**

1. (0.3.1) Localizar en Internet el intérprete de Basic llamado Bywater Basic, en su versión para el sistema operativo que se esté utilizando y probar el primer programa de ejemplo que se ha visto en el apartado 0.1.
2. (0.3.2) Localizar en Internet el compilador de Pascal llamado Free Pascal, en su versión para el sistema operativo que se esté utilizando, instalarlo y probar el segundo programa de ejemplo que se ha visto en el apartado 0.1.
3. (0.3.3) Localizar un compilador de C para el sistema operativo que se esté utilizando (si es Linux o alguna otra versión de Unix, es fácil que se encuentre ya instalado) y probar el tercer programa de ejemplo que se ha visto en el apartado 0.1.

# 1. Toma de contacto con C

Dentro de los lenguajes de programación, C es un lenguaje que tiene un cierto "**prestigio**". Esto se debe fundamentalmente a dos razones:

- Es bastante "**portable**": un programa bien hecho en C se podrá llevar a un ordenador distinto o incluso a un sistema operativo distinto (de MsDos a Windows o a Linux, por ejemplo) con muy pocos cambios o quizás incluso sin ningún cambio. El motivo es que existe un estándar: el **ANSI C**, que soportan casi todos los compiladores. Por eso, si nos ceñimos al estándar, es seguro que nuestros programas funcionarán en distintos sistemas; cuanto más nos separemos del estándar (en ocasiones podremos hacerlo), más difícil será que funcionen en otro sistema distinto.

- Permite hacer "**casi de todo**": podemos usar órdenes de alto nivel (muy cercanas al lenguaje humano), pero también de bajo nivel (más cercanas a lo que realmente entiende el ordenador). De hecho, podremos incluso incorporar órdenes en lenguaje ensamblador en medio de un programa escrito en C, aunque eso supone que ganemos en control de la máquina que estamos manejando, a costa de perder en portabilidad (el programa ya no se podrá llevar a otros ordenadores que no usen el mismo lenguaje ensamblador).

**En su contra**, el lenguaje C tiene que es más difícil de aprender que otros y que puede resultar difícil de leer (por lo que ciertos errores pueden tardar más en encontrarse).

Los **pasos** que seguiremos para crear un programa en C serán:

1. Escribir el programa en lenguaje C (**fichero fuente**), con cualquier editor de textos.
2. Compilarlo con nuestro compilador. Esto creará un **"fichero objeto"**, ya convertido a un lenguaje que el ordenador es capaz de entender.
3. Enlazarlo con otros ficheros del compilador, unas bibliotecas auxiliares que incluirán en nuestro programa aquellas posibilidades que hayamos empleado nosotros pero que realmente no sean parte del lenguaje C básico, sino ampliaciones de algún tipo. Esto dará lugar al **fichero ejecutable**, que ya podremos usar desde MsDos o el sistema operativo que estemos manejando, en nuestro ordenador o en cualquier otro, aunque ese otro ordenador no tenga el compilador que nosotros hemos utilizado.

La mayoría de los compiladores actuales permiten dar todos estos pasos desde un único **entorno**, en el que escribimos nuestros programas, los compilamos, y los depuramos en caso de que exista algún fallo.

En el siguiente apartado veremos un ejemplo de uno de estos entornos, dónde localizarlo y cómo instalarlo.

## *1.1 Escribir un texto en C*

Vamos con un primer ejemplo de programa en C, posiblemente el más sencillo de los que "hacen algo útil". Se trata de escribir un texto en pantalla. La apariencia de este programa la vimos en el tema anterior. Vamos a verlo ahora con más detalle:

```
#include <stdio.h>

int main()
{
     printf("Hola");
     return 0;
}
```

Esto escribe "Hola" en la pantalla. Pero antes de ver cómo probar este programa, hay mucho que comentar:

- Eso de "#include" nos permite incluir ciertas características para **ampliar** el lenguaje base. En este caso, el motivo es que en el lenguaje C base no hay predefinida ninguna orden para escribir en pantalla[1] (!), sino que este tipo de órdenes son una extensión, que se define en un fichero llamado "stdio.h". Esto no es un problema, vamos a encontrar ese "stdio.h" en cualquier compilador que usemos. ¿Y por qué se pone entre < y >? ¿Y por qué eso de # al principio? Esos detalles los iremos viendo un poco más adelante.
- Ya que hemos hablado de ella, "printf" es la orden que se encarga de **mostrar un texto** en pantalla. Es la responsable de que hayamos necesitado escribir ese "include" al principio del programa.
- Aun quedan cosas: ¿qué hacen esas llaves { y } ? C es un lenguaje estructurado, en el que un programa está formado por diversos "**bloques**". Todos los elementos que componen este bloque deben estar relacionados entre sí, lo que se indica encerrándolos entre llaves: { y }. Por convenio, y para facilitar la legibilidad, el contenido de cada bloque se escribe un poco más a la derecha (típicamente 4 espacios).
- ¿Por qué el "**return** 0"? Será la forma de decir "todo ha terminado sin problemas". En algunas versiones del lenguaje C se podrá omitir esa línea. Normalmente, cuando más antigua sea la versión, más tolerante será y nos permitirá eliminar algunos formalismos; las versiones más recientes serán más estrictas y nos obligarán a incluir estos formalismos, o al menos nos avisarán si no están presentes.
- Finalmente, ¿qué es eso de "**main**"? Es algo que debe existir siempre, e indica el punto en el que realmente comenzará a funcionar el programa. Después de "main" van dos llaves { y }, que delimitan el bloque más importante: el **cuerpo** del programa. ¿Y por qué tiene la palabra "int" delante? ¿Y por qué tiene un paréntesis vacío a continuación? Eso lo veremos más adelante...

Cuando vayamos avanzando un poco, iremos concretando más alguna que otra cosa de las que aquí han quedado "en el aire".

[1] Hay que recordar que un ordenador no es sólo lo que acostumbramos a tener sobre nuestra mesa, con pantalla y teclado. Existen otros muchos tipos de ordenadores que realizan tareas complejas y no necesitan una pantalla durante su funcionamiento normal, como el ordenador que controla el ABS de un coche.

**Ejercicio propuesto**: (1.1.1) Crea un programa en C que te salude por tu nombre (ej: "Hola, Nacho").

Sólo un par de cosas más antes de seguir adelante:

- Cada orden de C debe terminar con un **punto y coma** (;)
- El C es un lenguaje de **formato libre**, de modo que puede haber varias órdenes en una misma línea, u órdenes separadas por varias líneas o espacios entre medias. Lo que realmente indica dónde termina una orden y donde empieza la siguiente son los puntos y coma. Por ese motivo, el programa anterior se podría haber escrito también así (aunque no es aconsejable, porque puede resultar menos legible):

```
#include <stdio.h>
int main() {   printf("Hola");   return 0; }
```

- De hecho, hay dos formas especialmente frecuentes de colocar la llave de comienzo, y yo usaré ambas indistintamente. Una es como hemos hecho en el primer ejemplo: situar la llave de apertura en una línea, sola, y justo encima de la llave de cierre correspondiente. La segunda forma habitual es situándola a continuación del nombre del bloque que comienza (en nuestro caso, a continuación de la palabra "main"), así:

```
#include <stdio.h>

int main() {
    printf("Hola");
    return 0;
}
```

  (esta es la forma que yo emplearé en este texto cuando estemos trabajando con fuentes de mayor tamaño, para que ocupe un poco menos de espacio).

- La gran mayoría de las órdenes que encontraremos en el lenguaje C son palabras en inglés o abreviaturas de éstas. Pero hay que tener en cuenta que C **distingue entre mayúsculas** y minúsculas, por lo que "printf" es una palabra reconocida, pero "Printf", "PRINTF" o "PrintF" no lo son.

**¿Y si ese programa "me da error"?** En algunas revisiones recientes del lenguaje C (especialmente en los compiladores que siguen el estándar C99) se obliga a que aparezca la palabra "int" antes de "main", y a que la última línea sea "return 0;". Veremos los motivos más adelante, en el tema 7, pero de momento asumiremos que si queremos que nuestro programa se pueda probar con cualquier compilador, su apariencia deberá ser la que se ha visto con anterioridad, mientras que algunos compiladores más antiguos se permitirá omitir algunos detalles (int, return) y el fuente podría quedar así:

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

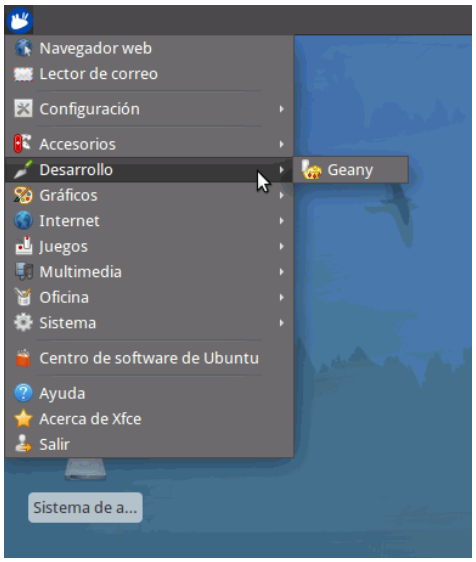
```
{
    printf("Hola");
}
```

### 1.1.1. Cómo probar este programa en Linux

Los sistemas operativos de la familia Unix, como Linux, suelen incluir un compilador de C, de modo que será fácil probar nuestros programas desde uno de estos sistemas. De hecho, usar Linux es una buena forma de practicar para alguien que empieza a programar en C… incluso si utiliza Windows normalmente, porque:

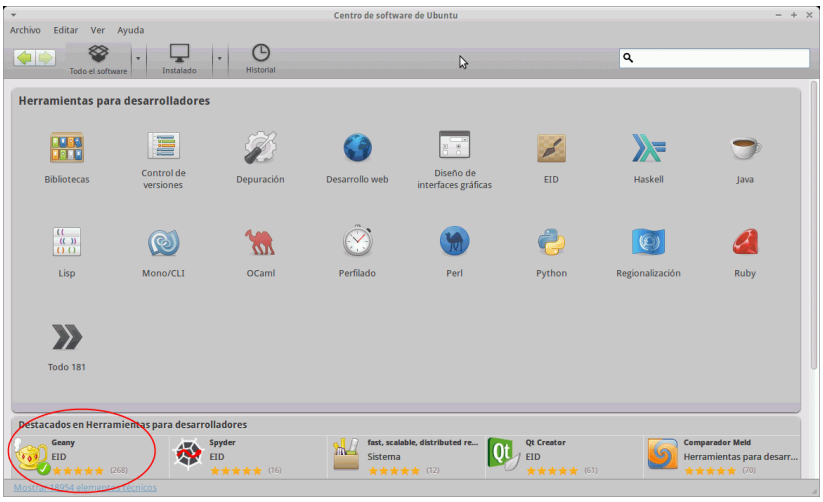
- Algunas versiones recientes de Linux, como Ubuntu, permiten instalarlo junto con Windows de forma muy sencilla, sin necesidad de crear particiones ni responder a preguntas técnicas complejas. Al encender el ordenador, se nos preguntaría si queremos arrancar en Windows o en Linux.
- Una alternativa aún más sencilla y con menos efectos colaterales, aunque algo más lenta en velocidad de ejecución, es la siguiente: Un ordenador con 2 Gb de memoria RAM o más, permite también crear una máquina virtual (usando software gratuito, como VirtualBox) e instalar en ella cualquier versión de Linux. Entonces, Linux se usaría desde dentro de Windows, como cualquier otro programa.

En los siguientes pasos, supondré que usamos una versión de Linux de los últimos años, que tenga entorno gráfico.

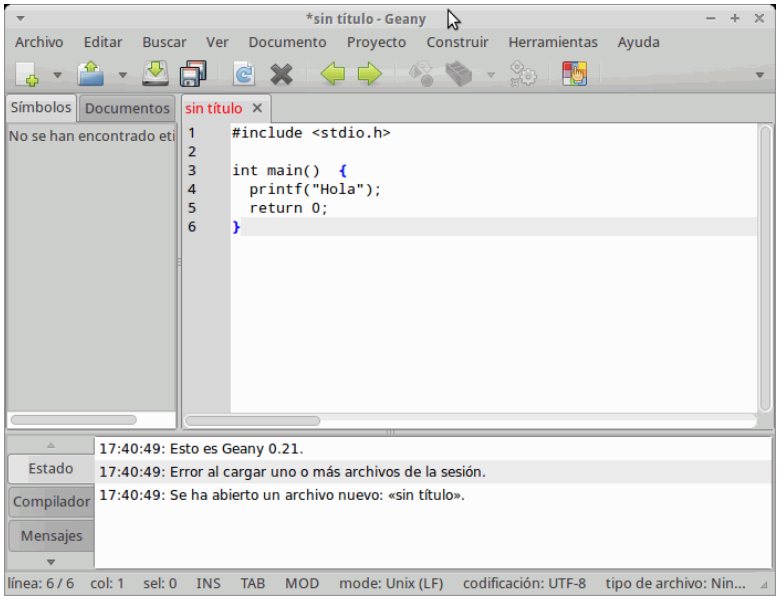



La forma más sencilla es usar algún entorno integrado, que nos permita teclear nuestros fuentes y probarlos desde él. A mí me gusta Geany, que es muy ligero y potente. Es posible que ya esté instalado en una parte del menú llamada "Desarrollo" o "Programación" (las siguientes capturas de pantalla corresponden al escritorio XFCE, de Xubuntu 12.04):

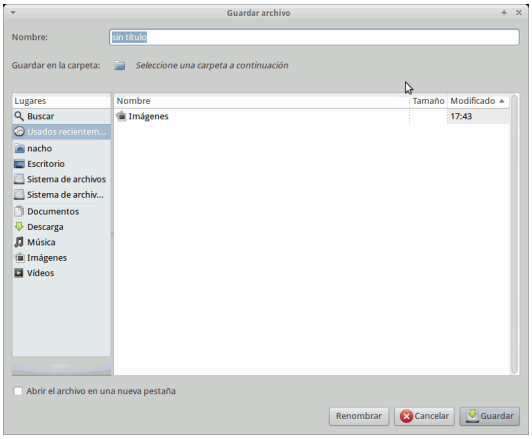
Si no estuviera instalado, añadir software en un sistema Linux suele ser muy fácil: basta abrir el gestor gráfico de paquetes de nuestro sistema, como Synaptic, Adept o el "Centro de Software de Ubuntu":



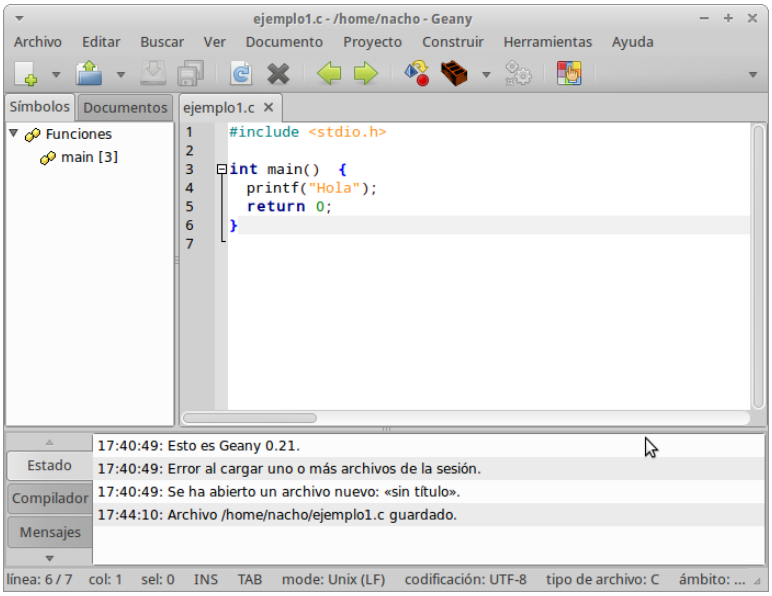

En él aparecerá una lista de software qe podemos instalar, entre la que debería aparecer Geany. Una vez instalado, entramos a él y podemos empezar a teclear nuestro programa:



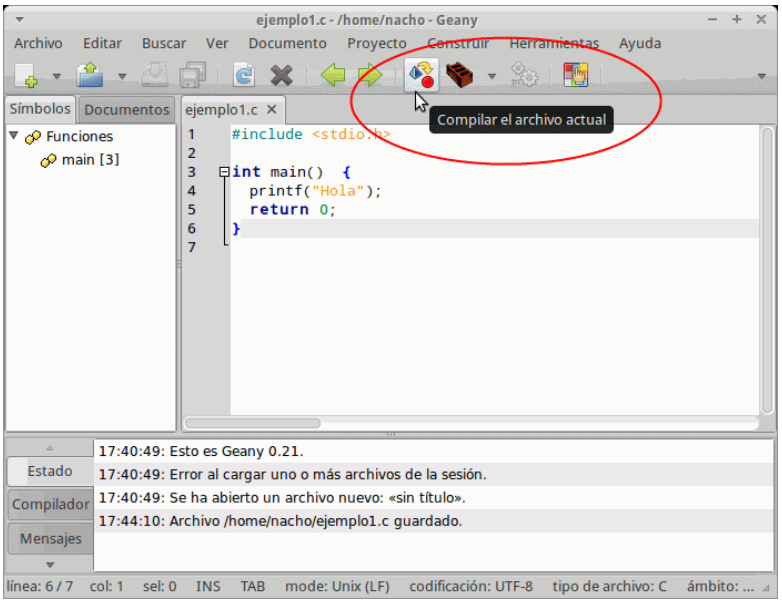

La sintaxis no se verá realzada en colores todavía, porque el editor aún no sabe que es un fuente en lenguaje C. Para eso, entramos al menú "Archivo" y escogemos la opción "Guardar como":



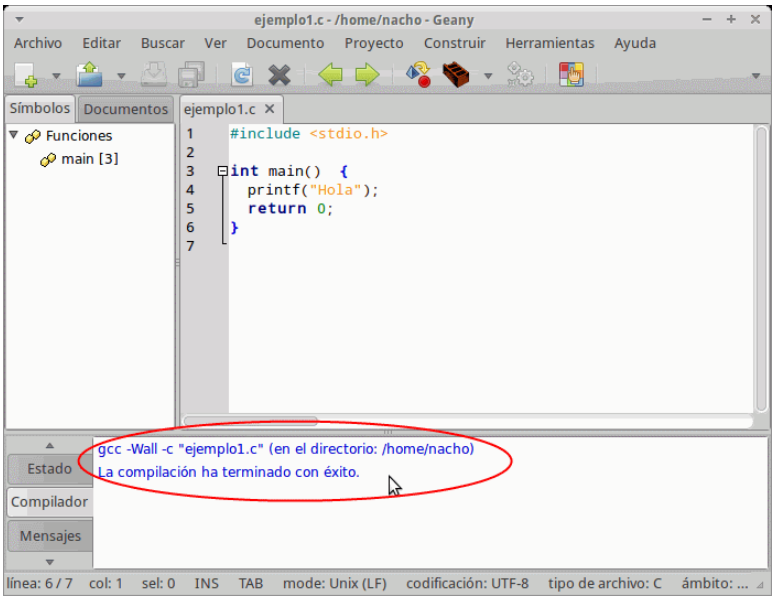

Si ponemos un nombre que termine en ".c", el editor sabrá que se trata de un fuente en este lenguaje y empezará a mostrar la sintaxis destacada en colores:



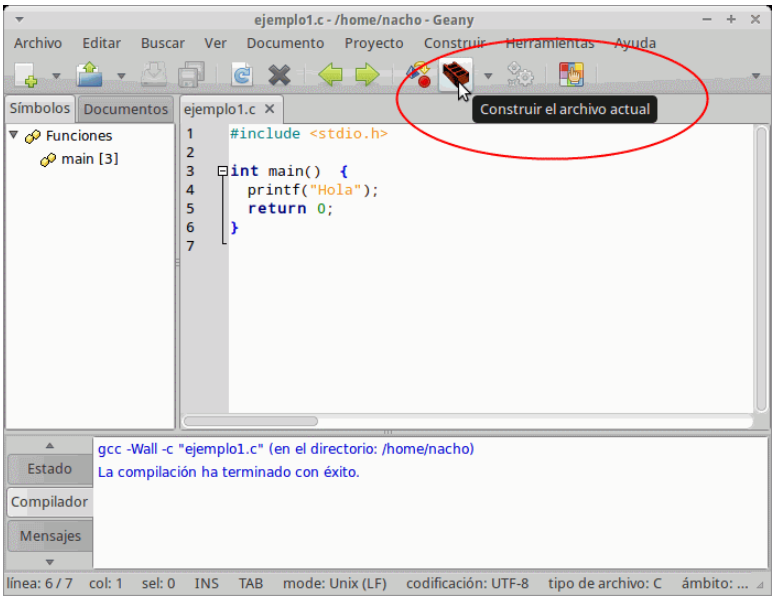

Probar el resultado de nuestro fuente desde aquí también es sencillo: tenemos un botón que nos permite compilarlo



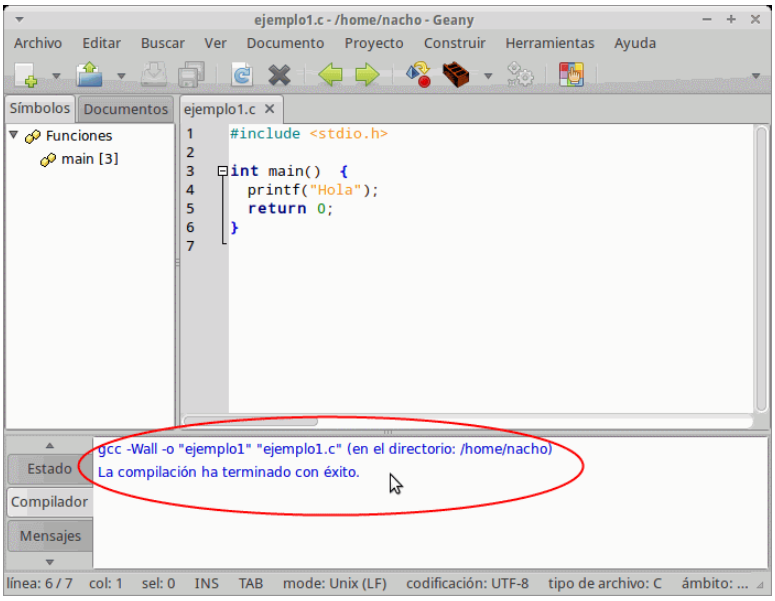

y en la parte inferior se nos dirá si todo ha sido correcto, o bien qué errores hay y en qué líneas están:



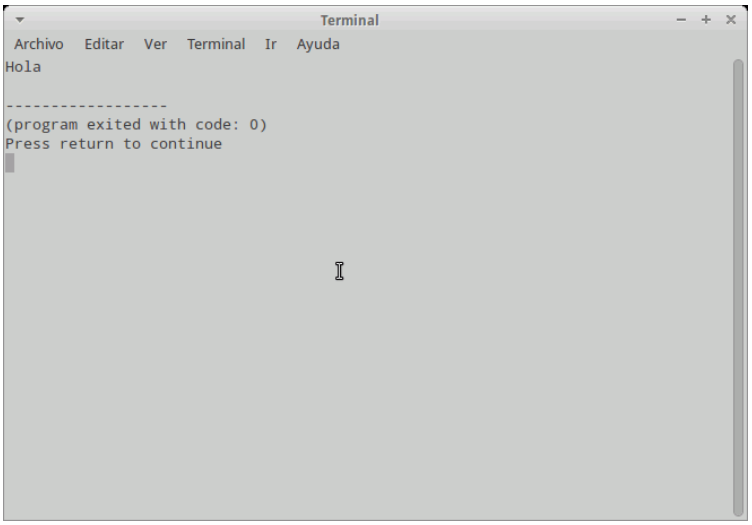

Pero esta opción de compilar no crea ningún ejecutable, así que normalmente preferiremos la de "construir", que se encuentra a su lado:



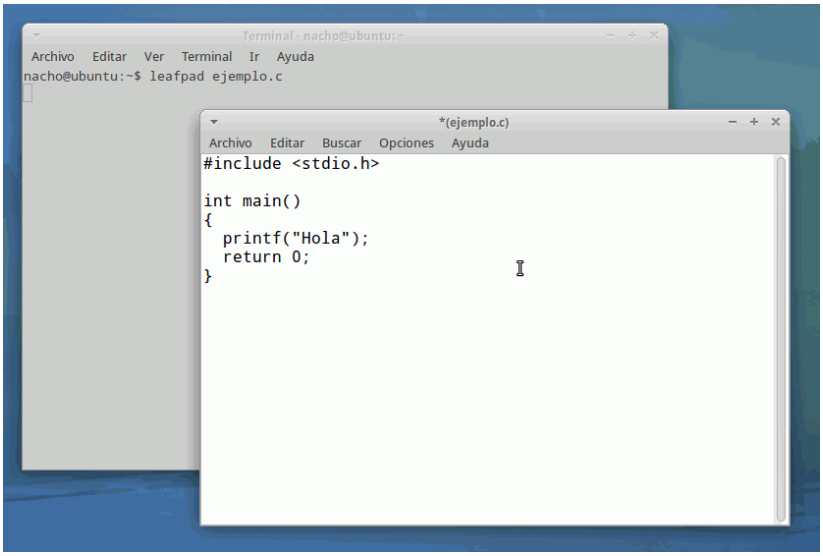

y, nuevamente, se nos informará en la parte inferior de si todo ha sido correcto:



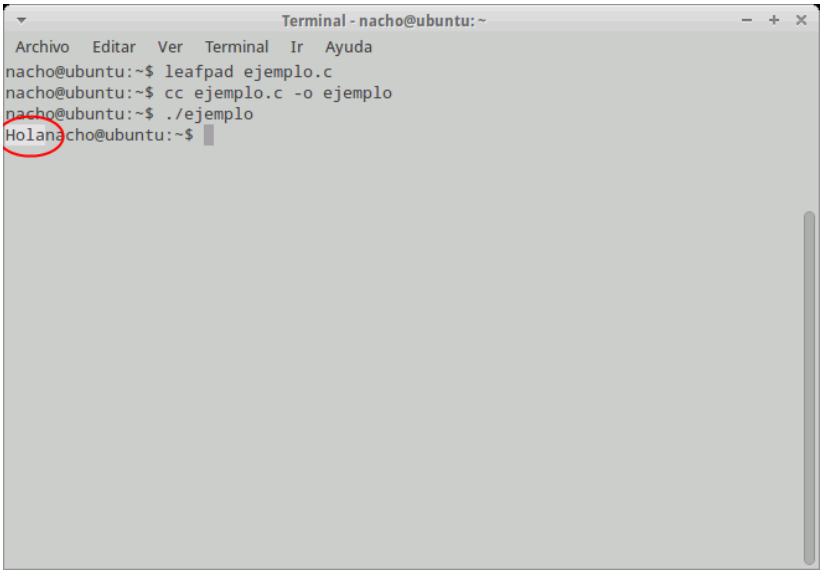

Y a su derecha tenemos otro botón que nos permite ejecutar el programa. Aparecerá una ventana de terminal, en la que podremos ver su resultado, y que se quedará "parada" hasta que pulsemos Intro, para que tengamos tiempo de ver los resultados.




Si preferimos "compilar a mano", primero tendremos que lanzar un editor para teclear nuestro fuente. El editor del sistema dependerá de qué escritorio usemos: en XFCE es Leafpad, muy simple, en Gnome es Gedit, más completo, y en KDE es Kate, también muy completo:



Una vez hemos introducido el fuente, lo compilaríamos con "cc" (y la opción "-o" para indicar el nombre del ejecutable), y lo lanzaríamos por su nombre, precedido por "./":




Es decir, para compilar nuestro fuente usaríamos:

```
cc ejemplo.c –o ejemplo
```

Donde:

- "cc" es la orden que se usa para poner el compilador en marcha.
- "ejemplo.c" es el nombre del fuente que queremos compilar.
- La opción "-o" se usa para indicar el nombre que queremos que tenga el fichero ejecutable resultante (la "o" viene de "output", salida).
- "ejemplo" es el nombre que tendrá el programa ejecutable.

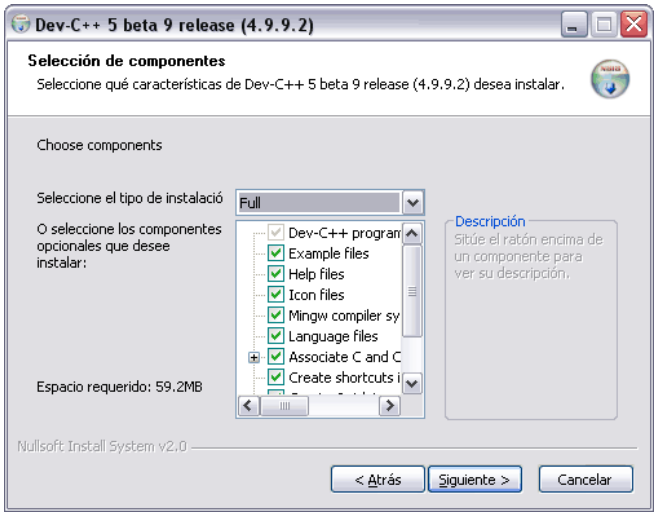
Y para probar el programa teclearíamos

```
./ejemplo
```

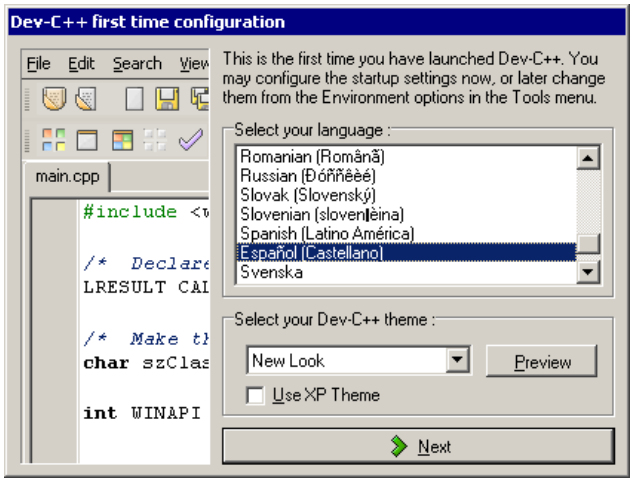
Con lo que debería aparecer escrito "Hola" en la pantalla.

### 1.1.2. Cómo probar este programa en Windows

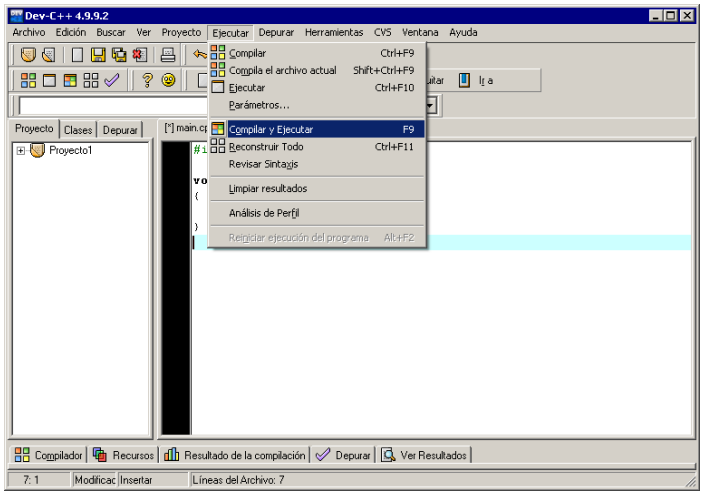
La familia de sistemas Windows no incluye ningún compilador de C, de modo que tendremos que localizar uno e instalarlo. Existen muchos gratuitos que se pueden descargar de Internet, y que incluyen un editor y otras herramientas auxiliares. Es el caso de Dev-C++, por ejemplo, que es sencillo aunque algo anticuado. Tiene su página oficial en www.bloodshed.net. La instalación es poco más que hacer doble clic en el fichero descargado, y hacer clic varias veces en el botón "Siguiente":




En el caso de Dev-C++, podemos incluso trabajar con el entorno en español:

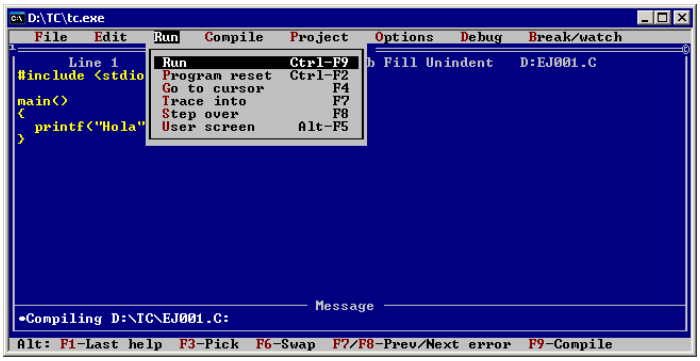



Para crear nuestro programa, en el menú "Archivo" escogemos "Nuevo / Código fuente", y nos aparece un editor vacío en el que ya podemos empezar a teclear. Si queremos nuestro programa en funcionamiento, entraríamos al menú "Ejecutar" y usaríamos la opción "Compilar y ejecutar":



(si todavía no hemos guardado los cambios en nuestro fuente, nos pediría antes que lo hiciéramos).

Puede ocurrir que se muestre el texto en pantalla, pero la ventana **desaparezca tan rápido que no tengamos tiempo de leerlo**. Si es nuestro caso, tenemos dos opciones:



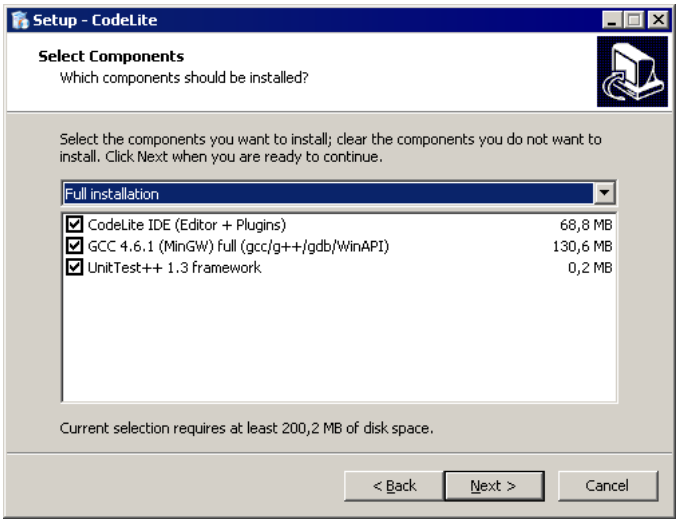

- Algunos entornos (como los de la familia Turbo C y Borland C) tienen una opción "User Screen" (pantalla de usuario) en el menú "Run" (ejecutar), que nos mostraría lo que ha aparecido en esa pantalla que no pudimos leer.
- Otros entornos, como Dev-C++, no tienen esa posibilidad, por lo que deberíamos hacer un pequeño cambio a nuestro fuente, para que espere a que pulsemos la tecla Intro antes de terminar. Una orden que nos permitiría hacer eso (y que veremos con más detalle más adelante) es "getchar()", así que nuestro fuente quedaría

```
#include <stdio.h>

int main()
{
    printf("Hola");
    getchar();
    return 0;
}
```

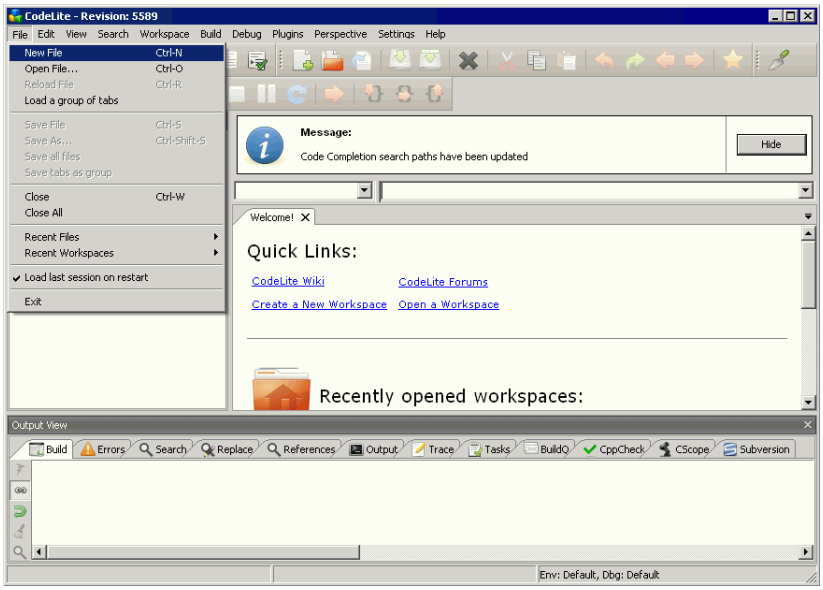
O bien, si usamos la sintaxis más abreviada:

```
#include <stdio.h>

main()
{
    printf("Hola");
    getchar();
}
```

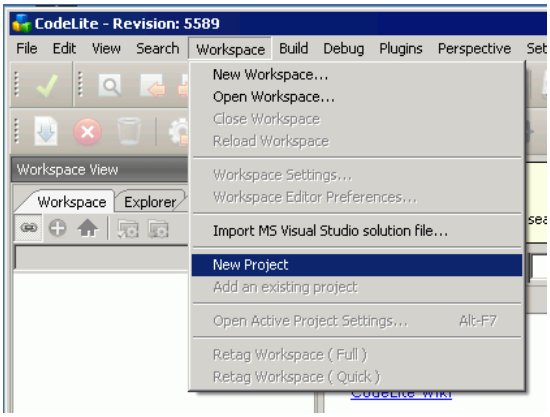
Dev-C++ incluye un compilador antiguo. Turbo C++ y Borland C++, muy extendidos, tampoco son recientes. Hay alternativas para Windows que son algo más modernas y gratuitas, como CodeLite, que se puede descargar desde www.codelite.org. Si se va a usar desde Windows, tendremos que descargar la versión que incluye el compilador MinGW. Su instalación es sencilla: poco más que hacer doble clic, comprobar que estamos instalando la versión que incluye GCC (MinGW) e ir pulsando los botones "Next":



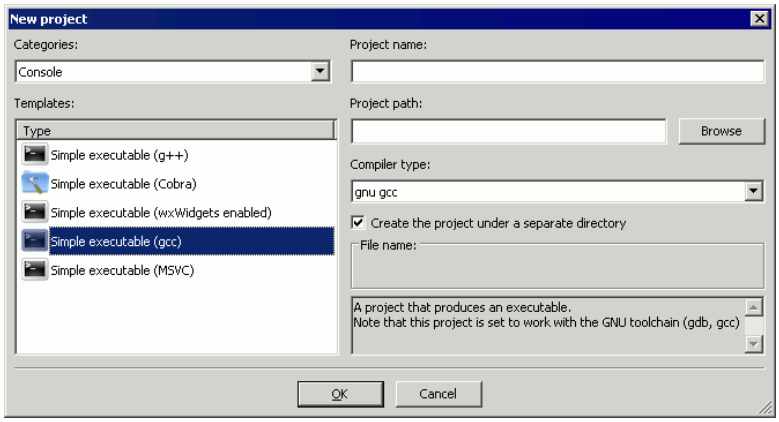

Y para poder crear un programa, no podemos empezar a teclear directamente, ni siquiera crear un nuevo fuente con la opción "New File" del menú "File", porque no se nos permitirá compilar ese fuente:



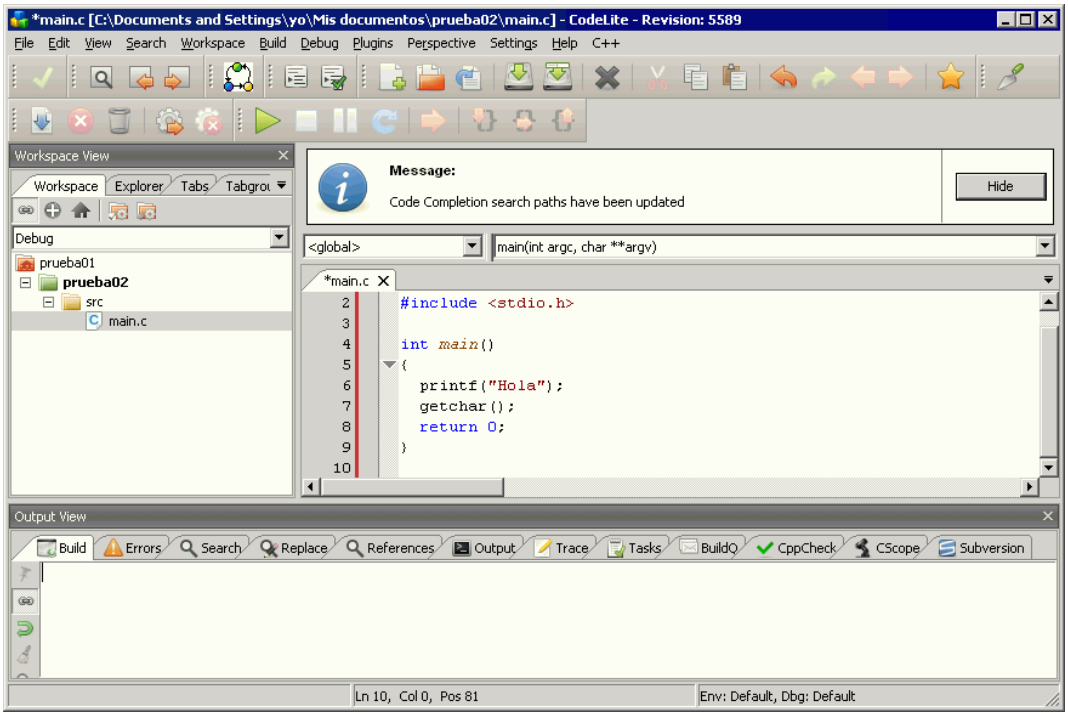

sino que debemos decir que queremos crear un nuevo proyecto ("New project"), desde el menú "Workspace" (hablaremos un poco más sobre "proyectos" en el tema 11):



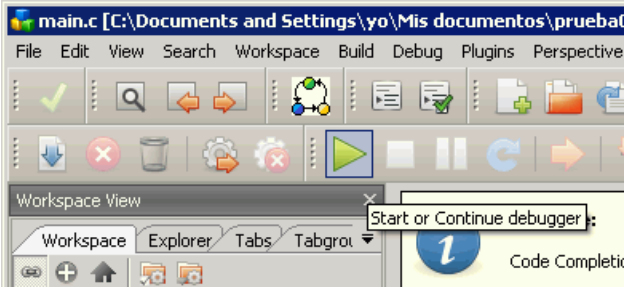
este proyecto será de "consola", para compilar con "gcc":




Deberemos darle un nombre (por ejemplo, "prueba01"), hacer clic en "Browse" para elegir una carpeta en la que guardarlo (por ejemplo, "Mis documentos") y entonces ya podremos comenzar a teclear el fuente:




Cuando esté listo, lo guardamos y pulsamos el botón de "comenzar la depuración":



Se nos preguntará si queremos construir el ejecutable antes de lanzarlo, a lo que deberemos responder que sí, y, si todo es correcto, se nos mostrará una pantalla "de consola" con los resultados:


(En este caso, al igual que con Dev-C++, mientras estemos haciendo pruebas, la línea anterior al "return 0" que completa el programa debería ser "getchar();", para que el programa se detenga y podamos leer los resultados. En un proyecto "entregable" de una universidad o centro de estudios, o bien en una aplicación de consola terminada, típicamente habría que eliminar esos "getchar").

## *1.2. Mostrar números enteros en pantalla*

Cuando queremos escribir un texto "tal cual", como en el ejemplo anterior, lo encerramos entre comillas. Pero no siempre querremos escribir textos prefijados. En muchos casos, se tratará de algo que habrá que calcular. El caso más sencillo es el de una operación matemática. En principio, podríamos pensar en intentar algo así (que está **mal**):

```
printf(3+4);
```

En muchos lenguajes de programación esto es perfectamente válido, pero no en C. La función "printf" nos obliga a que lo que escribamos en primer lugar sea un texto, indicado entre comillas. Eso que le indicamos entre comillas es realmente un **código de formato**. Dentro de ese código de formato podemos tener caracteres especiales, con los que le indicamos dónde y cómo queremos que aparezca un número (u otras cosas). Esto lo veremos con detalle un poco más adelante, pero de momento podemos anticipar que "**%d**" sirve para decir "quiero que aquí aparezca un número entero". ¿Qué número? El que le indicamos a continuación, separado por una coma:

```
printf("%d", 3+4);
```

Este ejemplo mostraría en pantalla un número entero (%d) que sea el resultado de suma 3 y 4.

Podemos escribir entre las comillas más detalles sobre lo que estamos haciendo:

```
printf("El resultado de sumar 3 y 4 es %d", 3+4);
```

El fuente completo (ejemplo 002) sería:

```
#include <stdio.h>

int main()
{
    printf("El resultado de sumar 3 y 4 es %d", 3+4);
    return 0;
}
```

(Como en los ejemplos anteriores, recuerda que con ciertos entornos de desarrollo puedes necesitar incluir "getchar();" antes de "return", para poder ver el resultado en pantalla antes de que se cierre la ventana)

**Ejercicio propuesto**: (1.2.1) Crea un programa que diga el resultado de sumar 118 y 56.

## *1.3. Operaciones aritméticas básicas*

Está claro que el símbolo de la suma será un +, y podemos esperar cual será el de la resta, pero alguna de las operaciones matemáticas habituales tiene símbolos menos intuitivos. Veamos cuales son los más importantes:

| Operador | Operación |
|---|---|
| + | Suma |
| - | Resta, negación |
| * | Multiplicación |
| / | División |
| % | Resto de la división ("módulo") |

**Ejercicio propuesto:**

- (1.3.0.1) Hacer un programa que calcule el producto de los números 12 y 13.
- (1.3.0.2) Un programa que calcule la diferencia entre 12345 y 998.
- (1.3.0.3) Un programa que calcule y muestre el resultado de dividir 3765 entre 18 y el resto de esa división.

### 1.3.1. Orden de prioridad de los operadores

Sencillo:

- En primer lugar se realizarán las operaciones indicadas entre paréntesis.
- Luego la negación.
- Después las multiplicaciones, divisiones y el resto de la división.
- Finalmente, las sumas y las restas.
- En caso de tener igual prioridad, se analizan de izquierda a derecha.

**Ejercicio propuesto:** (1.3.1.1) Calcular (a mano y después comprobar desde C) el resultado de estas operaciones:

- -2 + 3 * 5

- (20+5) % 6
- 15 + -5*6 / 10
- 2 + 10 / 5 * 2 - 7 % 1

### 1.3.2. Introducción a los problemas de desbordamiento

El espacio del que disponemos para almacenar los números es limitado. Si el resultado de una operación es un número "demasiado grande", obtendremos un resultado erróneo. Por eso en los primeros ejemplos usaremos números pequeños. Más adelante veremos a qué se debe realmente este problema y cómo evitarlo.

## *1.4. Introducción a las variables: int*

Las **variables** son algo que no contiene un valor predeterminado, un espacio de memoria al que nosotros asignamos un nombre y en el que podremos almacenar datos.

El primer ejemplo nos permitía escribir "Hola". El segundo nos permitía sumar dos números que habíamos prefijado en nuestro programa. Pero esto tampoco es "lo habitual", sino que esos números dependerán de valores que haya tecleado el usuario o de cálculos anteriores.

Por eso necesitaremos usar variables, en las que guardemos los datos con los que vamos a trabajar y también los resultados temporales. Vamos a ver como ejemplo lo que haríamos para sumar dos números enteros que fijásemos en el programa.

### 1.4.1. Definición de variables: números enteros

Para usar una cierta variable primero hay que **declararla**: indicar su nombre y el tipo de datos que querremos guardar.

El primer tipo de datos que usaremos serán números enteros (sin decimales), que se indican con "int" (abreviatura del inglés "integer"). Después de esta palabra se indica el nombre que tendrá la variable:

```
int primerNumero;
```

Esa orden reserva espacio para almacenar un número entero, que podrá tomar distintos valores, y al que nos referiremos con el nombre "primerNumero".

Si vamos a usar dos o más números enteros, podemos declararlos en la misma línea:

```
int primerNumero, segundoNumero;
```

### 1.4.2. Asignación de valores

Podemos darle un valor a esa variable durante el programa haciendo

```
    primerNumero = 234;
```

O también podemos darles un valor inicial ("inicializarlas") antes de que empiece el programa, en el mismo momento en que las definimos:

```
    int primerNumero = 234;
```

O incluso podemos definir e inicializar más de una variable a la vez

```
    int primerNumero = 234, segundoNumero = 567;
```

(esta línea reserva espacio para dos variables, que usaremos para almacenar números enteros; una de ellas se llama primerNumero y tiene como valor inicial 234 y la otra se llama segundoNumero y tiene como valor inicial 567).

Después ya podemos hacer operaciones con las variables, igual que las hacíamos con los números:

```
    suma = primerNumero + segundoNumero;
```

### 1.4.3. Mostrar el valor de una variable en pantalla

Una vez que sabemos cómo mostrar un número en pantalla, es sencillo mostrar el valor de una variable. Para un número hacíamos cosas como

```
printf("El resultado es %d", 3+4);
```

pero si se trata de una variable es idéntico:

```
printf("Su suma es %d", suma);
```

Ya sabemos todo lo suficiente para crear nuestro programa (ejemplo 003) que sume dos números usando variables:

```
#include <stdio.h>

int main()
{
    int primerNumero;
    int segundoNumero;
    int suma;

    primerNumero = 234;
    segundoNumero = 567;
    printf("Comenzamos... ");

    suma = primerNumero + segundoNumero;
    printf("Su suma es %d", suma);
```

```
    return 0;
}
```

(Recuerda que en muchos compiladores para Windows, quizá necesites incluir "getchar();" antes del "return 0" para tener tiempo de leer lo que aparece en pantalla).

Repasemos lo que hace:

- *#include <stdio.h>* dice que queremos usar funciones de entrada/salida estándar.
- *int main()* indica donde comienza en sí el cuerpo del programa.
- *{* señala el principio del cuerpo (de "main")
- *int primerNumero;* reserva espacio para guardar un número entero, al que llamaremos primerNumero.
- *int segundoNumero;* reserva espacio para guardar otro número entero, al que llamaremos segundoNumero.
- *int suma;* reserva espacio para guardar un tercer número entero, al que llamaremos suma.
- *primerNumero = 234;* da el valor del primer número que queremos sumar
- *segundoNumero = 567;* da el valor del segundo número que queremos sumar
- *suma = primerNumero + segundoNumero;* halla la suma de esos dos números y la guarda en otra variable, en vez de mostrarla directamente en pantalla.
- *printf("Su suma es %d", suma);* muestra en pantalla el resultado de esa suma.
- *return 0;* debe ser la última orden, por detalles que veremos más adelante.
- *}* señala el final del cuerpo (de "main")

Nota: las variables las podemos declarar **dentro del cuerpo** del programa (*main*) o fuera de él. En programas tan sencillos no habrá diferencia. Más adelante veremos que en ciertos casos sí se comportarán de forma distinta según donde las hayamos declarado.

Lo que sí es importante es que muchos compiladores de C obligan a **declarar las variables antes de cualquier otra orden**, de modo que el siguiente programa (ejemplo 003b)

```
#include <stdio.h>

int main()
{
    int primerNumero;
    int segundoNumero;

    primerNumero = 234;
    segundoNumero = 567;
    printf("Comenzamos... ");

    int suma;
    suma = primerNumero + segundoNumero;
    printf("Su suma es %d", suma);

    return 0;
}
```

puede no parecerle correcto a compiladores de C previos al estándar C99, porque la variabe "suma" se declara muy tarde (a partir de este estándar sí se permite declarar variables en cualquier punto, cosa que también permitirán algunos compiladores más antiguos pero que sean compiladores de lenguaje C y también de C++).

Podemos **resumir** un poco este fuente (ejemplo 003c), si damos los valores a las variables al inicializarlas:

```
#include <stdio.h>

int main()
{
    int primerNumero = 234;
    int segundoNumero = 567;
    int suma;

    suma = primerNumero + segundoNumero;
    printf("Su suma es %d", suma);

    return 0;
}
```

**Ejercicio propuesto:** (1.4.3.1) Hacer un programa que calcule el producto de los números 121 y 132, usando variables.

## *1.5. Identificadores*

Estos nombres de variable (lo que se conoce como "**identificadores**") pueden estar formados por letras, números o el símbolo de subrayado (_) y deben comenzar por letra o subrayado. No deben tener espacios entre medias, y hay que recordar que las vocales acentuadas y la eñe son problemáticas, porque no son letras "estándar" en todos los idiomas. Algunos compiladores permiten otros símbolos, como el $, pero es aconsejable no usarlos, de modo que el programa sea más portable (funcione con facilidad en distintos sistemas).

Por eso, no son nombres de variable válidos:

```
1numero        (empieza por número)
un numero      (contiene un espacio)
Año1           (tiene una eñe)
MásDatos       (tiene una vocal acentuada)
```

Tampoco podremos usar como identificadores las **palabras reservadas** de C. Por ejemplo, la palabra "int" se refiere a que cierta variable guardará un número entero, así que esa palabra "int" no la podremos usar tampoco como nombre de variable (pero no vamos a incluir ahora una lista de palabras reservadas de C, ya nos iremos encontrando con ellas).

De momento, intentaremos usar nombres de variables que a nosotros nos resulten claros, y que no parezca que puedan ser alguna orden de C.

Hay que recordar que en C las **mayúsculas y minúsculas** se consideran diferentes, de modo que si intentamos hacer

```
    PrimerNumero = 234;
    primernumero = 234;
```

o cualquier variación similar, el compilador protestará y nos dirá que no conoce esa variable, porque la habíamos declarado como

```
    int primerNumero;
```

El **número de letras** que puede tener un "identificador" (el nombre de una variable, por ejemplo) depende del compilador que usemos. Es frecuente que permitan cualquier longitud, pero que realmente sólo se fijen en unas cuantas letras (por ejemplo, en las primeras 8 o en las primeras 32). Eso quiere decir que puede que algún compilador considerase como iguales las variables NumeroParaAnalizar1 y NumeroParaAnalizar2, porque tienen las primeras 18 letras iguales. El C estándar (ANSI C) permite cualquier longitud, pero sólo considera las primeras 31.

## *1.6. Comentarios*

Podemos escribir comentarios, que el compilador ignora, pero que pueden servir para aclararnos cosas a nosotros. Se escriben entre /* y */:

```
int suma;  /* Porque guardaré el valor para usarlo más tarde */
```

Es conveniente escribir comentarios que aclaren la misión de las partes de nuestros programas que puedan resultar menos claras a simple vista. Incluso suele ser aconsejable que el programa comience con un comentario, que nos recuerde qué hace el programa sin que necesitemos mirarlo de arriba a abajo. Un ejemplo casi exagerado:

```
/* ---- Ejemplo en C (004): sumar dos números prefijados ---- */

#include <stdio.h>

int main()
{
    int primerNumero = 234;
    int segundoNumero = 567;
    int suma;  /* Porque guardaré el valor para usarlo más tarde */

    /* Primero calculo la suma */
    suma = primerNumero + segundoNumero;
    /* Y después muestro su valor */
    printf("Su suma es %d", suma);

    return 0;
}
```

Un comentario puede empezar en una línea y terminar en otra distinta, así:

```
/*  Este
    es un comentario que
    ocupa más de una línea
*/
```

El estándar C99, que siguen algunos compiladores posteriores a 1999, permite también **comentarios de una sola línea**, que comienzan con doble barra y terminan al final de la línea actual. Al igual que pasaba con las declaraciones en cualquier punto del programa, es posible que este tipo de comentarios también lo permitan algunos compiladores más antiguos pero que sean compiladores de lenguaje C y también de C++.

```
// Este es un comentario hasta fin de línea
```

El fuente anterior se podría rescribir de la siguiente forma, usando comentarios de una línea:

```
// ---- Ejemplo en C (004b): sumar dos números prefijados ----

#include <stdio.h>

int main()
{
    int primerNumero = 234;
    int segundoNumero = 567;
    int suma;  // Porque guardaré el valor para usarlo más tarde

    // Primero calculo la suma
    suma = primerNumero + segundoNumero;
    // Y después muestro su valor
    printf("Su suma es %d", suma);

    return 0;
}
```

## *1.7. Datos por el usuario: scanf*

Si queremos que sea el usuario de nuestro programa quien teclee los valores, necesitamos una nueva orden, llamada "scanf". Su manejo recuerda al de "printf", con una pequeña diferencia:

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

Con ese "%d" indicamos que esperamos leer un número entero (igual que para "printf") y con &primerNumero decimos que queremos que ese valor leído se guarde en la variable llamada "primerNumero". La diferencia está en ese símbolo & que nos obliga scanf a poner antes del nombre de la variable. Más adelante veremos qué quiere decir ese símbolo y en qué otros casos se usa.

Un ejemplo de programa que sume dos números tecleados por el usuario sería:

```
/*-------------------------*/
/*  Ejemplo en C nº 5:     */
/*  c005.C                 */
/*                         */
/*  Leer valores para      */
/*  variables              */
/*                         */
/*  Curso de C,            */
/*    Nacho Cabanes        */
/*-------------------------*/

#include <stdio.h>

int main()    /* Cuerpo del programa */
{
    int primerNumero, segundoNumero, suma;    /* Nuestras variables */

    printf("Introduce el primer número ");
    scanf("%d", &primerNumero);
    printf("Introduce el segundo número ");
    scanf("%d", &segundoNumero);
    suma = primerNumero + segundoNumero;
    printf("Su suma es %d", suma);

    return 0;
}
```

**Ejercicios propuestos**:

- (1.7.1) Multiplicar dos números tecleados por usuario
- (1.7.2) El usuario tecleará dos números (x e y), y el programa deberá calcular cual es el resultado de su división y el resto de esa división.
- (1.7.3) El usuario tecleará dos números (a y b), y el programa mostrar el resultado de la operación (a+b)*(a-b) y el resultado de la operación $a^2$-$b^2$.

# 2. Tipos de datos básicos

## *2.1. Tipo de datos entero*

### 2.1.1. Tipos de enteros: signed/unsigned, short/long

Hemos hablado de números enteros, de cómo realizar operaciones sencillas y de cómo usar variables para reservar espacio y poder trabajar con datos cuyo valor no sabemos de antemano.

Empieza a ser el momento de refinar, de dar más detalles. El primer "matiz" importante es el signo de los números: hemos hablado de números enteros (sin decimales), pero no hemos detallado si esos números son positivos, negativos o si podemos elegirlo nosotros.

Pues es sencillo: si no decimos nada, se da por sentado que el número puede ser negativo o positivo. Si queremos dejarlo más claro, podemos añadir la palabra "**signed**" (con signo) antes de "int". Este es uno de los "**modificadores**" que podemos emplear. Otro modificador es "**unsigned**" (sin signo), que nos sirve para indicar al compilador que no vamos a querer guardar números negativos, sólo positivos. Vamos a verlo con un ejemplo:

```
/*-------------------------*/
/*  Ejemplo en C nº 6:     */
/*  c006.C                 */
/*                         */
/*  Numeros enteros con y  */
/*  sin signo              */
/*                         */
/*  Curso de C,            */
/*    Nacho Cabanes        */
/*-------------------------*/

#include <stdio.h>

int main()    /* Cuerpo del programa */
{
    int primerNumero;
    signed int segundoNumero;
    unsigned int tercerNumero;

    primerNumero = -1;
    segundoNumero = -2;
    tercerNumero = 3;
    printf("El primer numero es %d, ", primerNumero);
    printf("el segundo es %d, ", segundoNumero);
    printf("el tercer numero es %d.", tercerNumero);

    return 0;
}
```

El resultado de este programa es el que podíamos esperar:

```
El primer numero es -1, el segundo es -2, el tercer numero es 3
```

¿Y sí hubiéramos escrito "tercerNumero=-3" después de decir que va a ser un entero sin signo, pasaría algo? No, el programa mostraría un –3 en la pantalla. El lenguaje C nos deja ser tan descuidados como queramos ser, así que generalmente deberemos trabajar con un cierto cuidado.

La pregunta que puede surgir ahora es: ¿resulta útil eso de no usar números negativos? Sí, porque entonces podremos usar números positivos de mayor tamaño (dentro de poco veremos por qué ocurre esto).

De igual modo que detallamos si queremos que un número pueda ser negativo o no, tenemos disponible otro modificador que nos permite decir que queremos más espacio, para poder almacenar números más grandes. Un "int" normalmente nos permite guardar números inferiores al 2.147.483.647, pero si usamos el modificador "**long**", ciertos sistemas nos permitirán usar números mucho mayores, o bien con el modificador "**short**" podremos usar números menores (sólo hasta 32.767, en caso de que necesitemos optimizar la cantidad de memoria que utilizamos).

**Ejercicio propuesto**: (2.1.1.1) Multiplicar dos números de 4 cifras que teclee el usuario, usando el modificador "long".

### 2.1.2. Problemática: asignaciones y tamaño de los números; distintos espacios ocupados según el sistema

El primer problema a tener en cuenta es que si asignamos a una variable "demasiado pequeña" un valor más grande del que podría almacenar, podemos obtener valores incorrectos. Un caso típico es intentar asignar un valor "long" a una variable "short":

```
/*-------------------------*/
/*  Ejemplo en C nº 7:     */
/*  c007.C                 */
/*                         */
/*  Numeros enteros        */
/*  demasiado grandes      */
/*                         */
/*  Curso de C,            */
/*    Nacho Cabanes        */
/*-------------------------*/

#include <stdio.h>

int main()    /* Cuerpo del programa */
{
    int primerNumero;
    signed int segundoNumero;
    unsigned int tercerNumero;

    primerNumero = -1;
    segundoNumero = 33000;
    tercerNumero = 123456;
    printf("El primer numero es %d, ", primerNumero);
    printf("el segundo es %d, ", segundoNumero);
```

```
    printf("el tercer numero es %d.", tercerNumero);

    return 0;
}
```

El resultado en pantalla de este programa, si usamos el compilador Turbo C 2.01 no sería lo que esperamos:

```
El primer numero es -1, el segundo es -32536, el tercer numero es -7616
```

Y un problema similar lo podríamos tener si asignamos valores de un número sin signo a uno con signo (o viceversa).

Pero el problema llega más allá: el espacio ocupado por un "int" **depende del sistema operativo** que usemos, a veces incluso del compilador. Por ejemplo, hemos comentado que con un "int" podemos almacenar números cuyo valor sea inferior a 2.147.483.647, pero el ejemplo anterior usaba números pequeños y aun así daba problemas.

¿Por qué? Por que este último ejemplo lo hemos probado con un compilador **para MsDos**. Se trata de un sistema operativo más antiguo, de 16 bits, capaz de manejar números de menor tamaño. En estos sistemas, los "int" llegaban hasta 32.767 (lo que equivale a un short en los sistemas modernos de 32 bits) y los "short" llegaban sólo hasta 127. En los sistemas de 64 bits existen "int" de mayor tamaño.

Para entender por qué ocurre esto, vamos a hablar un poco sobre unidades de medida utilizadas en informática y sobre sistemas de numeración.

### 2.1.3. Unidades de medida empleadas en informática (1): bytes, kilobytes, megabytes...

En informática, la unidad básica de información es el **byte**. En la práctica, podemos pensar que un byte es el equivalente a una **letra**. Si un cierto texto está formado por 2000 letras, podemos esperar que ocupe unos 2000 bytes de espacio en nuestro disco.

Eso sí, suele ocurrir que realmente un texto de 2000 letras que se guarde en el ordenador ocupe más de 2000 bytes, porque se suele incluir información adicional sobre los tipos de letra que se han utilizado, cursivas, negritas, márgenes y formato de página, etc.

Un byte se queda corto a la hora de manejar textos o datos algo más largos, con lo que se recurre a un múltiplo suyo, el **kilobyte**, que se suele abreviar **Kb** o K.

En teoría, el prefijo kilo querría decir "mil", luego un kilobyte debería ser 1000 bytes, pero en los ordenadores conviene buscar por comodidad una potencia de 2 (pronto veremos por qué), por lo que se usa $2^{10}$ =1024. Así, la equivalencia exacta es 1 K = 1024 bytes.

Los K eran unidades típicas para medir la memoria de ordenadores: 640 K ha sido mucho tiempo la memoria habitual en los IBM PC y similares. Por otra parte, una página mecanografiada suele ocupar entre 2 K (cerca de 2000 letras) y 4 K.

Cuando se manejan datos realmente extensos, se pasa a otro múltiplo, el **megabyte** o Mb, que es 1000 K (en realidad 1024 K) o algo más de un millón de bytes. Por ejemplo, en un diskette "normal" caben 1.44 Mb, y en un Compact Disc para ordenador (Cd-Rom) se pueden almacenar hasta 700 Mb. La memoria principal (RAM) de un ordenador actual suele andar por encima de los 512 Mb, y un disco duro actual puede tener una capacidad superior a los 80.000 Mb.

Para estas unidades de gran capacidad, su tamaño no se suele medir en megabytes, sino en el múltiplo siguiente: en **gigabytes**, con la correspondencia 1 Gb = 1024 Mb. Así, son cada vez más frecuentes los discos duros con una capacidad de 120, 200 o más gigabytes.
Y todavía hay unidades mayores, pero que aún se utilizan muy poco. Por ejemplo, un **terabyte** son 1024 gigabytes.

Todo esto se puede resumir así:

| Unidad | Equivalencia | Valores posibles |
|---|---|---|
| Byte | - | 0 a 255 (para guardar 1 letra) |
| Kilobyte (K o Kb) | 1024 bytes | Aprox. media página mecanografiada |
| Megabyte (Mb) | 1024 Kb | - |
| Gigabyte (Gb) | 1024 Mb | - |
| Terabyte (Tb) | 1024 Gb | - |

Pero por debajo de los bytes también hay unidades más pequeñas...

**Ejercicios propuestos**:

- (2.1.3.1) Crea un programa que te diga cuántos bytes son 3 megabytes.
- (2.1.3.2) ¿Cuántas letras se podrían almacenar en una agenda electrónica que tenga 32 Kb de capacidad? (primero calcúlalo en papel y luego crea un programa en C que te dé la respuesta).
- (2.1.3.3) Si suponemos que una canción típica en formato MP3 ocupa cerca de 3.500 Kb, ¿cuántas se podrían guardar en un reproductor MP3 que tenga 256 Mb de capacidad?
- (2.1.3.4) ¿Cuántos diskettes de 1,44 Mb harían falta para hacer una copia de seguridad de un ordenador que tiene un disco duro de 6,4 Gb? ¿Y si usamos compact disc de 700 Mb, cuántos necesitaríamos?
- (2.1.3.5) ¿A cuantos CD de 700 Mb equivale la capacidad de almacenamiento de un DVD de 4,7 Gb? ¿Y la de uno de 8,5 Gb?

### 2.1.4. Unidades de medida empleadas en informática (2): los bits

Dentro del ordenador, la información se debe almacenar realmente de alguna forma que a él le resulte "cómoda" de manejar. Como la memoria del ordenador se basa en componentes

electrónicos, la unidad básica de información será que una posición de memoria esté usada o no (totalmente llena o totalmente vacía), lo que se representa como un 1 o un 0. Esta unidad recibe el nombre de **bit**.

Un bit es demasiado pequeño para un uso normal (recordemos: sólo puede tener dos valores: 0 ó 1), por lo que se usa un conjunto de ellos, 8 bits, que forman un **byte**. Las matemáticas elementales (combinatoria) nos dicen que si agrupamos los bits de 8 en 8, tenemos 256 posibilidades distintas (variaciones con repetición de 2 elementos tomados de 8 en 8: $VR_{2,8}$):

```
00000000
00000001
00000010
00000011
00000100
...
11111110
11111111
```

Por tanto, si en vez de tomar los bits de 1 en 1 (que resulta cómodo para el ordenador, pero no para nosotros) los utilizamos en grupos de 8 (lo que se conoce como un byte), nos encontramos con 256 posibilidades distintas, que ya son más que suficientes para almacenar una letra, o un signo de puntuación, o una cifra numérica o algún otro símbolo.

Por ejemplo, se podría decir que cada vez que encontremos la secuencia 00000010 la interpretaremos como una letra A, y la combinación 00000011 como una letra B, y así sucesivamente.

También existe una correspondencia entre cada grupo de bits y un número del 0 al 255: si usamos el sistema binario de numeración (que aprenderemos dentro de muy poco), en vez del sistema decimal, tenemos que:

```
0000 0000 (binario) = 0 (decimal)
0000 0001 (binario) = 1 (decimal)
0000 0010 (binario) = 2 (decimal)
0000 0011 (binario) = 3 (decimal)
...
1111 1110 (binario) = 254 (decimal)
1111 1111 (binario) = 255 (decimal)
```

En la práctica, existe un código estándar, el **código ASCII** (American Standard Code for Information Interchange, código estándar americano para intercambio de información), que relaciona cada letra, número o símbolo con una cifra del 0 al 255 (realmente, con una secuencia de 8 bits): la "a" es el número 97, la "b" el 98, la "A" el 65, la "B", el 32, el "0" el 48, el "1" el 49, etc. Así se tiene una forma muy cómoda de almacenar la información en ordenadores, ya que cada letra ocupará exactamente un byte (8 bits: 8 posiciones elementales de memoria).

Aun así, hay un inconveniente con el código ASCII: sólo los primeros 127 números son estándar. Eso quiere decir que si escribimos un texto en un ordenador y lo llevamos a otro, las letras básicas (A a la Z, 0 al 9 y algunos símbolos) no cambiarán, pero las letras internacionales (como la Ñ o las vocales con acentos) puede que no aparezcan correctamente, porque se les asignan números que no son estándar para todos los ordenadores. Esta limitación la evitan sistemas más modernos que el ASCII clásico, como es el caso de la codificación UTF-8, pero aún no la emplean todos los sistemas operativos.

**Nota:** Eso de que realmente el ordenador trabaja con ceros y unos, por lo que le resulta más fácil manejar los números que son potencia de 2 que los números que no lo son, es lo que explica que el prefijo *kilo* no quiera decir "exactamente mil", sino que se usa la potencia de 2 más cercana: $2^{10}$ =1024. Por eso, la equivalencia exacta es **1 K = 1024 bytes**.

### 2.1.5. Sistemas de numeración: 1- Sistema binario

Nosotros normalmente utilizamos el **sistema decimal** de numeración: todos los números se expresan a partir de potencias de 10, pero normalmente lo hacemos sin pensar.

Por ejemplo, el número 3.254 se podría desglosar como:

$3.254 = 3 \cdot 1000 + 2 \cdot 100 + 5 \cdot 10 + 4 \cdot 1$

o más detallado todavía:

$254 = 3 \cdot 10^3 + 2 \cdot 10^2 + 5 \cdot 10^1 + 4 \cdot 10^0$

(aunque realmente nosotros lo hacemos automáticamente: no nos paramos a pensar este tipo de cosas cuando sumamos o multiplicamos dos números).

Para los ordenadores no es cómodo contar hasta 10. Como partimos de "casillas de memoria" que están completamente vacías (0) o completamente llenas (1), sólo les es realmente cómodo contar con 2 cifras: 0 y 1.

Por eso, dentro del ordenador cualquier número se deberá almacenar como ceros y unos, y entonces los números se deberán desglosar en potencias de 2 (el llamado "**sistema binario**"):

$13 = 1 \cdot 8 + 1 \cdot 4 + 0 \cdot 2 + 1 \cdot 1$

o más detallado todavía:

$13 = 1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0$

de modo que el número decimal 13 se escribirá en binario como 1101.

En general, **convertir** un número binario al sistema decimal es fácil: lo expresamos como suma de potencias de 2 y sumamos:

0110 1101 (binario) = $0 \cdot 2^7 + 1 \cdot 2^6 + 1 \cdot 2^5 + 0 \cdot 2^4 + 1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0$ =
= $0 \cdot 128 + 1 \cdot 64 + 1 \cdot 32 + 0 \cdot 16 + 1 \cdot 8 + 1 \cdot 4 + 0 \cdot 2 + 1 \cdot 1$ = 109 (decimal)

Convertir un número de decimal a binario resulta algo menos intuitivo. Una forma sencilla es ir dividiendo entre las potencias de 2, y coger todos los cocientes de las divisiones:

109 / 128 = 0 (resto: 109)
109 / 64 = 1 (resto: 45)
45 / 32 = 1 (resto: 13)
13 / 16 = 0 (resto: 13)
13 / 8 = 1 (resto: 5)
5 / 4 = 1 (resto: 1)
1 / 2 = 0 (resto: 1)
1 / 1 = 1 (se terminó).

Si "juntamos" los cocientes que hemos obtenido, aparece el número binario que buscábamos:
109 decimal = 0110 1101 binario

(Nota: es frecuente separar los números binarios en grupos de 4 cifras -medio byte- para mayor legibilidad, como yo he hecho en el ejemplo anterior; a un grupo de 4 bits se le llama **nibble**).

Otra forma sencilla de convertir de decimal a binario es dividir consecutivamente entre 2 y coger los restos que hemos obtenido, pero en orden inverso:

109 / 2 = 54, resto 1
54 / 2 = 27, resto 0
27 / 2 = 13, resto 1
13 /2 = 6, resto 1
6 / 2 = 3, resto 0
3 / 2 = 1, resto 1
1 / 2 = 0, resto 1
(y ya hemos terminado)

Si leemos esos restos de abajo a arriba, obtenemos el número binario: 1101101 (7 cifras, si queremos completarlo a 8 cifras rellenamos con ceros por la izquierda: 01101101).

¿Y se puede hacer operaciones con números binarios? Sí, casi igual que en decimal:

0·0 = 0  0·1 = 0  1·0 = 0  1·1 = 1
0+0 = 0  0+1 = 1  1+0 = 1  1+1 = 10 (en decimal: 2)

**Ejercicios propuestos**:

- (2.1.5.1) Expresar en sistema binario los números decimales 17, 101, 83, 45.
- (2.1.5.2) Expresar en sistema decimal los números binarios de 8 bits: 01100110, 10110010, 11111111, 00101101
- (2.1.5.3) Sumar los números 01100110+10110010, 11111111+00101101. Comprobar el resultado sumando los números decimales obtenidos en el ejercicio anterior.
- (2.1.5.4) Multiplicar los números binarios de 4 bits 0100·1011, 1001·0011. Comprobar el resultado convirtiéndolos a decimal.

### 2.1.6. Sistemas de numeración: 2- Sistema octal

Hemos visto que el sistema de numeración más cercano a como se guarda la información dentro del ordenador es el sistema binario. Pero los números expresados en este sistema de numeración "ocupan mucho". Por ejemplo, el número 254 se expresa en binario como 11111110 (8 cifras en vez de 3).

Por eso, se han buscado otros sistemas de numeración que resulten más "compactos" que el sistema binario cuando haya que expresar cifras medianamente grandes, pero que a la vez mantengan con éste una correspondencia algo más sencilla que el sistema decimal. Los más usados son el sistema octal y, sobre todo, el hexadecimal.

El sistema octal de numeración trabaja en base 8. La forma de convertir de decimal a binario será, como siempre dividir entre las potencias de la base. Por ejemplo:

254 (decimal) ->
254 / 64 = 3 (resto: 62)
62 / 8 = 7 (resto: 6)
6 / 1 = 6 (se terminó)

de modo que

$254 = 3 \cdot 8^2 + 7 \cdot 8^1 + 6 \cdot 8^0$

o bien

254 (decimal) = 376 (octal)

Hemos conseguido otra correspondencia que, si bien nos resulta a nosotros más incómoda que usar el sistema decimal, al menos es más compacta: el número 254 ocupa 3 cifras en decimal, y también 3 cifras en octal, frente a las 8 cifras que necesitaba en sistema binario.

Pero además existe una correspondencia muy sencilla entre el sistema octal y el sistema binario: si agrupamos los bits de 3 en 3, el paso de **binario a octal** es rapidísimo

254 (decimal) = 011 111 110 (binario)

011 (binario ) = 3 (decimal y octal)
111 (binario ) = 7 (decimal y octal)
110 (binario ) = 6 (decimal y octal)

de modo que

254 (decimal) = 011 111 110 (binario) = 376 (octal)

El paso desde el **octal al binario** y al decimal también es sencillo. Por ejemplo, el número 423 (octal) sería 423 (octal) = 100 010 011 (binario)

o bien

423 (octal) = 4 · 64 + 2 · 8 + 3 · 1 = 275 (decimal)

De cualquier modo, el sistema octal no es el que más se utiliza en la práctica, sino el hexadecimal…

**Ejercicios propuestos**:

- (2.1.6.1) Expresar en sistema octal los números decimales 17, 101, 83, 45.
- (2.1.6.2) Expresar en sistema octal los números binarios de 8 bits: 01100110, 10110010, 11111111, 00101101
- (2.1.6.3) Expresar en el sistema binario los números octales 171, 243, 105, 45.
- (2.1.6.4) Expresar en el sistema decimal los números octales 162, 76, 241, 102.

### 2.1.7. Sistemas de numeración: 3- Sistema hexadecimal

El sistema octal tiene un inconveniente: se agrupan los bits de 3 en 3, por lo que convertir de binario a octal y viceversa es muy sencillo, pero un byte está formado por 8 bits, que no es múltiplo de 3.

Sería más cómodo poder agrupar de 4 en 4 bits, de modo que cada byte se representaría por 2 cifras. Este sistema de numeración trabajará en base 16 ($2^4$ =16), y es lo que se conoce como sistema hexadecimal.

Pero hay una dificultad: estamos acostumbrados al sistema decimal, con números del 0 al 9, de modo que no tenemos cifras de un solo dígito para los números 10, 11, 12, 13, 14 y 15, que utilizaremos en el sistema hexadecimal. Para representar estas cifras usaremos las letras de la A a la F, así:

```
0 (decimal) = 0 (hexadecimal)
```

```
 1 (decimal) = 1 (hexadecimal)
 2 (decimal) = 2 (hexadecimal)
 3 (decimal) = 3 (hexadecimal)
 4 (decimal) = 4 (hexadecimal)
 5 (decimal) = 5 (hexadecimal)
 6 (decimal) = 6 (hexadecimal)
 7 (decimal) = 7 (hexadecimal)
 8 (decimal) = 8 (hexadecimal)
 9 (decimal) = 9 (hexadecimal)
10 (decimal) = A (hexadecimal)
11 (decimal) = B (hexadecimal)
12 (decimal) = C (hexadecimal)
13 (decimal) = D (hexadecimal)
14 (decimal) = E (hexadecimal)
15 (decimal) = F (hexadecimal)
```

Con estas consideraciones, expresar números en el sistema hexadecimal ya no es difícil:

254 (decimal) ->
254 / 16 = 15 (resto: 14)
14 / 1 = 14 (se terminó)

de modo que

$254 = 15 \cdot 16^{1} + 14 \cdot 16^{0}$

o bien

254 (decimal) = FE (hexadecimal)

Vamos a repetirlo para convertir de **decimal a hexadecimal** número más grande:

54331 (decimal) ->
54331 / 4096 = 13 (resto: 1083)
1083 / 256 = 4 (resto: 59)
59 / 16 = 3 (resto: 11)
11 / 1 = 11 (se terminó)

de modo que

$54331 = 13 \cdot 4096 + 4 \cdot 256 + 3 \cdot 16 + 11 \cdot 1$

o bien

$254 = 13 \cdot 16^{3} + 4 \cdot 16^{2} + 3 \cdot 16^{1} + 11 \cdot 16^{0}$

es decir

54331 (decimal) = D43B (hexadecimal)

Ahora vamos a dar el paso inverso: convertir de **hexadecimal a decimal**, por ejemplo el número A2B5

A2B5 (hexadecimal) = $10 \cdot 16^3 + 2 \cdot 16^2 + 11 \cdot 16^1 + 5 \cdot 16^0 = 41653$

El paso de **hexadecimal a binario** también es (relativamente) rápido, porque cada dígito hexadecimal equivale a una secuencia de 4 bits:

```
0 (hexadecimal) =  0 (decimal) = 0000 (binario)
1 (hexadecimal) =  1 (decimal) = 0001 (binario)
2 (hexadecimal) =  2 (decimal) = 0010 (binario)
3 (hexadecimal) =  3 (decimal) = 0011 (binario)
4 (hexadecimal) =  4 (decimal) = 0100 (binario)
5 (hexadecimal) =  5 (decimal) = 0101 (binario)
6 (hexadecimal) =  6 (decimal) = 0110 (binario)
7 (hexadecimal) =  7 (decimal) = 0111 (binario)
8 (hexadecimal) =  8 (decimal) = 1000 (binario)
9 (hexadecimal) =  9 (decimal) = 1001 (binario)
A (hexadecimal) = 10 (decimal) = 1010 (binario)
B (hexadecimal) = 11 (decimal) = 1011 (binario)
C (hexadecimal) = 12 (decimal) = 1100 (binario)
D (hexadecimal) = 13 (decimal) = 1101 (binario)
E (hexadecimal) = 14 (decimal) = 1110 (binario)
F (hexadecimal) = 15 (decimal) = 1111 (binario)
```

de modo que A2B5 (hexadecimal) = 1010 0010 1011 0101 (binario)

y de igual modo, de **binario a hexadecimal** es dividir en grupos de 4 bits y hallar el valor de cada uno de ellos:

11001010010010010101010100111 =>
0110 0101 0010 0100 1010 1010 0111 = 6524AA7

**Ejercicios propuestos**:

- (2.1.7.1) Expresar en sistema hexadecimal los números decimales 18, 131, 83, 245.
- (2.1.7.2) Expresar en sistema hexadecimal los números binarios de 8 bits: 01100110, 10110010, 11111111, 00101101
- (2.1.7.3) Expresar en el sistema binario los números hexadecimales 2F, 37, A0, 1A2.
- (2.1.7.4) Expresar en el sistema decimal los números hexadecimales 1B2, 76, E1, 2A.

### 2.1.8. Formato de constantes enteras: oct, hex

En C tenemos la posibilidad de dar un valor a una variable usando el sistema decimal, como hemos hecho hasta ahora, pero también podemos usar el sistema octal si ponemos un 0 a la izquierda del número, o el sistema hexadecimal, si usamos 0x (pero no existe una forma directa de trabajar con números en binario):

```
/*-------------------------*/
/*  Ejemplo en C nº 8:     */
/*  c008.C                 */
/*                         */
/*  Numeros enteros en     */
/*  decimal, octal y       */
/*  hexadecimal            */
/*                         */
/*  Curso de C,            */
/*    Nacho Cabanes        */
/*-------------------------*/

#include <stdio.h>

int main()    /* Cuerpo del programa */
{
    int primerNumero;
    int segundoNumero;
    int tercerNumero;

    primerNumero = 15;    /* Decimal */
    segundoNumero = 015;  /* Octal: 8+5=13 */
    tercerNumero = 0x15;  /* Hexadecimal: 16+5=21 */
    printf("El primer numero es %d, ", primerNumero);
    printf("el segundo es %d, ", segundoNumero);
    printf("el tercer numero es %d.", tercerNumero);

    return 0;
}
```

El resultado de este programa sería

```
El primer numero es 15, el segundo es 13, el tercer numero es 21.
```

**Ejercicios propuestos**:

- (2.1.8.1) Crea un programa en C que exprese en el sistema decimal los números octales 162, 76, 241, 102.
- (2.1.8.2) Crea un programa en C que exprese en el sistema decimal los números hexadecimales 1B2, 76, E1, 2A.

### 2.1.9. Representación interna de los enteros

Ahora que ya sabemos cómo se representa un número en sistema binario, podemos detallar un poco más cómo se almacenan los números enteros en la memoria del ordenador, lo que nos

ayudará a entender por qué podemos tener problemas al asignar valores entre variables que no sean exactamente del mismo tipo.

En principio, los números positivos se almacenan como hemos visto cuando hemos hablado del sistema binario. El único matiz que falta es indicar cuantos bits hay disponibles para cada número. Lo habitual es usar 16 bits para un "int" si el sistema operativo es de 16 bits (como MsDos) y 32 bits para los sistemas operativos de 32 bits (como la mayoría de las versiones de Windows y de Linux –o sistemas Unix en general-).

En cuanto a los "short" y los "long", depende del sistema. Vamos a verlo con un ejemplo:

| | Turbo C 2.01 (MsDos) | GCC 3.4.2 (Windows 32b) | GCC 3.4.2 (Linux 64b) |
|---|---|---|---|
| int: bits | 16 | 32 | 32 |
| int: valor máximo | 32.767 | 2.147.483.647 | 2.147.483.647 |
| short: bits | 16 | 16 | 16 |
| short: valor máximo | 32.767 | 32.767 | 32.767 |
| long: bits | 32 | 32 | 64 |
| long: valor máximo | 2.147.483.647 | 2.147.483.647 | $9 \cdot 10^{18}$ |

Para los **números enteros negativos**, existen varias formas posibles de representarlos. Las más habituales son:

- **Signo y magnitud**: el primer bit (el de más a la izquierda) se pone a 1 si el número es negativo y se deja a 0 si es positivo. Los demás bits se calculan como ya hemos visto.

  Por ejemplo, si usamos 4 bits, tendríamos
  3 (decimal) = 0011 -3 = 1011
  6 (decimal) = 0110 -6 = 1110

  Es un método muy sencillo, pero que tiene el inconveniente de que las operaciones en las que aparecen números negativos no se comportan correctamente. Vamos a ver un ejemplo, con números de 8 bits:

  13 (decimal) = 0000 1101 - 13 (decimal) = 1000 1101
  34 (decimal) = 0010 0010 - 34 (decimal) = 1010 0010
  13 + 34 = 0000 1101 + 0010 0010 = 0010 1111 = 47 (correcto)
  (-13) + (-34) = 1000 1101 + 1010 0010 = 0010 1111 = 47 (INCORRECTO)
  13 + (-34) = 0000 1101 + 1010 0010 = 1010 1111 = -47 (INCORRECTO)

- **Complemento a 1**: se cambian los ceros por unos para expresar los números negativos.

  Por ejemplo, con 4 bits
  3 (decimal) = 0011 -3 = 1100
  6 (decimal) = 0110 -6 = 1001

También es un método sencillo, en el que las operaciones con números negativos salen bien, y que sólo tiene como inconveniente que hay dos formas de expresar el número 0 (0000 0000 o 1111 1111), lo que complica algunos trabajos internos del ordenador.

**Ejercicio propuesto**: (2.1.9.1) convertir los números decimales 13, 34, -13, -34 a sistema binario, usando complemento a uno para expresar los números negativos. Calcular (en binario) el resultado de las operaciones 13+34, (-13)+(-34), 13+(-34) y comprobar que los resultados que se obtienen son los correctos.

- **Complemento a 2**: para los negativos, se cambian los ceros por unos y se suma uno al resultado.

  Por ejemplo, con 4 bits
  3 (decimal) = 0011 -3 = 1101
  6 (decimal) = 0110 -6 = 1010

  Es un método que parece algo más complicado, pero que no es difícil de seguir, con el que las operaciones con números negativos salen bien, y no tiene problemas para expresar el número 0 (00000000).

  Éste es el método que se suele usar para codificar los números negativos en un sistema informático "real".

  **Ejercicio propuesto**: (2.1.9.2) convertir los números decimales 13, 34, -13, -34 a sistema binario, usando complemento a dos para expresar los números negativos. Calcular (en binario) el resultado de las operaciones 13+34, (-13)+(-34), 13+(-34) y comprobar que los resultados que se obtienen son los correctos.

En general, todos los formatos que permiten guardar números negativos usan el primer bit para el signo. Por eso, si declaramos una variable como "unsigned", ese primer bit se puede utilizar como parte de los datos, y podemos almacenar números más grandes. Por ejemplo, un "unsigned int" en MsDos podría tomar valores entre 0 y 65.535, mientras que un "int" (con signo) podría tomar valores entre +32.767 y –32.768.

### 2.1.10. Incremento y decremento

Hay una operación que es muy frecuente cuando se crean programas, pero que no tiene un símbolo específico para representarla en matemáticas. Es incrementar el valor de una variable en una unidad:

```
a = a+1;
```

Pues bien, en C, existe una notación más compacta para esta operación, y para la opuesta (el decremento):

```
a++;          es lo mismo que    a = a+1;
a--;          es lo mismo que    a = a-1;
```

Pero esto tiene más misterio todavía del que puede parecer en un primer vistazo: podemos distinguir entre "preincremento" y "postincremento". En C es posible hacer asignaciones como

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

Así, si "a" valía 2, lo que esta instrucción hace es dar a "b" el valor de "a" y aumentar el valor de "a". Por tanto, al final tenemos que b=2 y a=3 (**postincremento**: se incrementa "a" tras asignar su valor).

En cambio, si escribimos

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

y "a" valía 2, primero aumentamos "a" y luego los asignamos a "b" (**preincremento**), de modo que a=3 y b=3.

Por supuesto, también podemos distinguir **postdecremento** (a--) y **predecremento** (--a).

**Ejercicios propuestos**:

- (2.1.10.1) Crear un programa que use tres variables x,y,z. Sus valores iniciales serán 15, -10, 2.147.483.647. Se deberá incrementar el valor de estas variables. ¿Qué valores esperas que se obtengan? Contrástalo con el resultado obtenido por el programa.
- (2.1.10.2) ¿Cuál sería el resultado de las siguientes operaciones? a=5; b=++a; c=a++; b=b*5; a=a*2;

Y ya que estamos hablando de las asignaciones, hay que comentar que en C es posible hacer **asignaciones múltiples**:

```
a = b = c = 1;
```

### 2.1.11. Operaciones abreviadas: +=

Pero aún hay más. Tenemos incluso formas reducidas de escribir cosas como "a = a+5". Allá van

```
a += b ;          es lo mismo que    a = a+b;
a -= b ;          es lo mismo que    a = a-b;
a *= b ;          es lo mismo que    a = a*b;
a /= b ;          es lo mismo que    a = a/b;
a %= b ;          es lo mismo que    a = a%b;
```

**Ejercicios propuestos**:

- (2.1.11.1) Crear un programa que use tres variables x,y,z. Sus valores iniciales serán 15, -10, 214. Se deberá incrementar el valor de estas variables en 12, usando el formato abreviado. ¿Qué valores esperas que se obtengan? Contrástalo con el resultado obtenido por el programa.
- (2.1.11.2) ¿Cuál sería el resultado de las siguientes operaciones? a=5; b=a+2; b-=3; c=-3; c*=2; ++c; a*=b;

### 2.1.12. Modificadores de acceso: const, volatile

Podemos encontrarnos con variables cuyo valor realmente no varíe durante el programa. Entonces podemos usar el modificador "**const**" para indicárselo a nuestro compilador, y entonces ya no nos dejará modificarlas por error.

```
const int MAXIMO = 10;
```

si luego intentamos

```
MAXIMO = 100;
```

obtendríamos un mensaje de error que nos diría que no podemos modificar una constante.

Por convenio, para ayudar a identificarlas dentro de un fuente, se suele escribir el nombre de una constante **en mayúsculas**, como en el ejemplo anterior.

También podemos encontrarnos (aunque es poco frecuente) con el caso contrario: una variable que pueda cambiar de valor sin que nosotros modifiquemos (porque accedamos a un valor que cambie "solo", como el reloj interno del ordenador, o porque los datos sean compartidos con otro programa que también pueda modicarlos, por ejemplo). En ese caso, usaremos el modificador "volatile", que hace que el compilador siempre compruebe el valor más reciente de la variable antes de usarlo, por si hubiera cambiado:

```
volatile int numeroDeUsuarios = 1;
```

## *2.2. Tipo de datos real*

Cuando queremos almacenar datos con decimales, no nos sirve el tipo de datos "int". Necesitamos otro tipo de datos que sí esté preparado para guardar números "reales" (con decimales). En el mundo de la informática hay dos formas de trabajar con números reales:

- **Coma fija**: el número máximo de cifras decimales está fijado de antemano, y el número de cifras enteras también. Por ejemplo, con un formato de 3 cifras enteras y 4 cifras decimales, el número 3,75 se almacenaría correctamente, el número 970,4361 también, pero el 5,678642 se guardaría como 5,6786 (se perdería a partir de la cuarta cifra decimal) y el 1010 no se podría guardar (tiene más de 3 cifras enteras).

- **Coma flotante**: el número de decimales y de cifras enteras permitido es variable, lo que importa es el número de cifras significativas (a partir del último 0). Por ejemplo, con 5 cifras significativas se podrían almacenar números como el 13405000000 o como el 0,0000007349 pero no se guardaría correctamente el 12,0000034, que se redondearía a un número cercano.

### 2.2.1. Simple y doble precisión

Tenemos dos tamaños para elegir, según si queremos guardar números con mayor cantidad de cifras o con menos. Para números con pocas cifras significativas (un máximo de 6) existe el tipo "float" y para números que necesiten más precisión (unas 10) tenemos el tipo "double":

| | float | double |
|---|---|---|
| Tamaño en bits | 32 | 64 |
| Valor máximo | $-3{,}4 \cdot 10^{-38}$ | $-1{,}7 \cdot 10^{-308}$ |
| Valor mínimo | $3{,}4 \cdot 10^{38}$ | $1{,}7 \cdot 10^{308}$ |
| Cifras significativas | 6 o más | 10 o más |

En algunos sistemas existe un tipo "long double", con mayor precisión todavía (40 bits o incluso 128 bits).

Para definirlos, se hace igual que en el caso de los números enteros:

```
float x;
```

o bien, si queremos dar un valor inicial en el momento de definirlos (recordando que para las cifras decimales no debemos usar una coma, sino un punto):

```
float x = 12.56;
```

### 2.2.2. Mostrar en pantalla números reales

En principio es sencillo: usaremos "printf", al que le indicaremos "%f" como código de formato:

```
printf("El valor de x es %f", x);  /* Escribiría 12.5600 */
```

Pero también podemos detallar la anchura, indicando el número de cifras totales y el número de cifras decimales:

```
printf("El valor de x es %5.2f", x);  /* Escribiría 12.56 */
```

Si indicamos una anchura mayor que la necesaria, se rellena con espacios al principio (queda alineado a la derecha)

```
printf("El valor de x es %7.2f", x);  /* Escribiría "  12.56" */
```

Si quisiéramos que quede alineado a la izquierda (con los espacios de sobra al final), debemos escribir la anchura como un número negativo

```
printf("El valor de x es %-7.2f", x);  /* Escribiría "12.56  " */
```

Si indicamos menos decimales que los necesarios, se redondeará el número

```
printf("El valor de x es %4.1f", x);  /* Escribiría 12.6 */
```

Y si indicamos menos cifras enteras que las necesarias, no se nos hará caso y el número se escribirá con la cantidad de cifras que sea necesario usar

```
printf("El valor de x es %1.0f", x);  /* Escribiría 13 */
```

Vamos a juntar todo esto en un ejemplo:

```
/*---------------------------*/
/*  Ejemplo en C nº 9:       */
/*  c009.c                   */
/*                           */
/*  Numeros en coma flotante */
/*                           */
/*  Curso de C,              */
/*    Nacho Cabanes          */
/*---------------------------*/

#include <stdio.h>

int main()
{
    float x = 12.56;

    printf("El valor de x es %f", x);
    printf(" pero lo podemos escribir con 2 decimales %5.2f", x);
    printf(" o solo con uno %5.1f", x);
    printf(" o con 7 cifras %7.1f", x);
    printf(" o alineado a la izquierda %-7.1f", x);
    printf(" o sin decimales %2.0f", x);
    printf(" o solo con una cifra %1.0f", x);

    return 0;
}
```

El resultado sería

```
El valor de f es 12.560000 pero lo podemos escribir con 2 decimales 12.56 o solo
con uno  12.6 o con 7 cifras    12.6 o alineado a la izquierda 12.6    o sin
decimales 13 o solo con una cifra 13
```

Si queremos que sea el usuario el que introduzca los valores, usaremos "%f" como código de formato en "scanf":

```
scanf("%f", &x);
```

**Ejercicios propuestos**:

- (2.2.1) El usuario de nuestro programa podrá teclear dos números de hasta 8 cifras significativas. El programa deberá mostrar el resultado de dividir el primer número entre el segundo, utilizando tres cifras decimales.
- (2.2.2) Crear un programa que use tres variables x,y,z. Las tres serán números reales, y nos bastará con dos cifras decimales. Deberá pedir al usuario los valores para las tres variables y mostrar en pantalla cual es el mayor de los tres números tecleados.

## *2.3. Operador de tamaño: sizeof*

Hemos comentado lo que habitualmente ocupa una variable de tipo int, de tipo long int, de tipo float... Pero tenemos una forma de saber exactamente lo que ocupa: un operador llamado "sizeof" (tamaño de). Veamos un ejemplo de su uso

```
/*--------------------------*/
/*  Ejemplo en C nº 10:     */
/*  c010.c                  */
/*                          */
/*  Tamaño de una variable o */
/*  de un tipo              */
/*                          */
/*  Curso de C,             */
/*    Nacho Cabanes         */
/*--------------------------*/

#include <stdio.h>

int main()
{
    float f;
    short int i;

    printf("El tamaño de mi float es %d", sizeof f);
    printf(" y lo normal para un float es %d", sizeof(float) );
    printf(" pero un entero corto ocupa %d", sizeof i);

    return 0;
}
```

que nos diría lo siguiente:

```
El tamaño de mi float es 4 y lo normal para un float es 4 pero un entero corto ocupa 2
```

Como se puede ver, hay una **peculiaridad**: si quiero saber lo que ocupa un tipo de datos, tengo que indicarlo entre paréntesis: sizeof(float), pero si se trata de una variable, puedo no usar paréntesis: sizeof i. Eso sí, el compilador no dará ningún mensaje de error si uso un paréntesis cuando sea una variable sizeof(i), así que puede resultar cómodo poner siempre el paréntesis, sin pararse a pensar si nos lo podríamos haber ahorrado.

**Ejercicios propuestos**:

- (2.3.1) Descubrir cual es el espacio ocupado por un "int" en el sistema operativo y compilador que utilizas.

## *2.4. Operador de molde: (tipo) operando*

Si tenemos dos números enteros y hacemos su división, el resultado que obtenemos es otro número entero, sin decimales:

```
float f = 5/2;  /* f valdrá 2.000000 */
```

Esto se debe a que la operación se realiza entre números enteros, se obtiene un resultado que es un número entero, y ese valor obtenido se asigna a la variable "float"... pero ya es demasiado tarde.

Para evitar ese tipo de problemas, podemos indicar que queremos convertir esos valores a numeros reales. Cuando son números, basta con que indiquemos algún decimal:

```
float f = 5.0/2.0;  /* ahora f valdrá 2.500000 */
```

y si son variables, añadiremos antes de ellas "(float)" para que las considere como números reales antes de trabajar con ellas:

```
float f = (float) x / (float) y;
```

Vamos a verlo mejor en un programa completo:

```
/*--------------------------*/
/*  Ejemplo en C nº 11:     */
/*  c011.c                  */
/*                          */
/*  Conversión de int a     */
/*  float                   */
/*                          */
/*  Curso de C,             */
/*    Nacho Cabanes         */
/*--------------------------*/

#include <stdio.h>

int main()
{
```

```
    int n1 = 5, n2 = 2;
    float division1, division2;

    printf("Mis números son %d y %d", n1, n2);
    division1 = n1/n2;
    printf(" y su division es %f", division1 );
    division2 = (float)n1 / (float)n2;
    printf(" pero si convierto antes a float: %f", division2 );

    return 0;
}
```

que tendría como resultado

```
Mis números son 5 y 2 y su division es 2.000000 pero si convierto antes a float: 2.500000
```

De igual modo, podemos convertir un "float" a "int" para despreciar sus decimales y quedarnos con la parte entera:

```
/*----------------------------*/
/*  Ejemplo en C nº 12:       */
/*  c012.c                    */
/*                            */
/*  Conversión de float a     */
/*  int                       */
/*                            */
/*  Curso de C,               */
/*    Nacho Cabanes           */
/*----------------------------*/

#include <stdio.h>

int main()
{
    float x = 5, y = 3.5;
    float producto;

    printf("Mis números son %3.1f y %3.1f", x, y);
    producto = x*y;
    printf(" y su producto es %3.1f", producto);
    printf(", sin decimales sería %d", (int) producto);

    return 0;
}
```

que daría

```
Mis números son 5.0 y 3.5 y su producto es 17.5, sin decimales sería 17
```

## *2.5. Tipo de datos carácter*

También tenemos un tipo de datos que nos permite almacenar una única letra (ya veremos que manipular una cadena de texto completa es relativamente complicado). Es el tipo "char":

```
char letra;
```

Asignar valores es sencillo:

```
letra = 'a';
```

(hay que destacar que se usa una **comilla simple** en vez de comillas dobles). Mostrarlos en pantalla también es fácil:

```
printf("%c", letra);
```

Así, un programa que leyera una letra tecleada por el usuario, fijara otra y mostrara ambas podría ser:

```
/*---------------------------*/
/*  Ejemplo en C nº 13:      */
/*  c013.c                   */
/*                           */
/*  Tipo de datos char       */
/*                           */
/*  Curso de C,              */
/*    Nacho Cabanes          */
/*---------------------------*/

#include <stdio.h>

int main()
{
    char letra1, letra2;

    printf("Teclea una letra ");
    scanf("%c", &letra1);
    letra2 = 'a';
    printf("La letra que has tecleado es %c y la prefijada es %c",
        letra1, letra2);

    return 0;
}
```

### 2.5.1. Secuencias de escape: \n y otras.

Al igual que ocurría con expresiones como %d, que tenían un significado especial, ocurre lo mismo con ciertos caracteres, que nos permiten hacer cosas como bajar a la línea siguiente o mostrar las comillas en pantalla.

Son las siguientes:

| Secuencia | Significado |
|---|---|
| \a | Emite un pitido |
| \b | Retroceso (permite borrar el último carácter) |
| \f | Avance de página (expulsa una hoja en la impresora) |
| \n | Avanza de línea (salta a la línea siguiente) |
| \r | Retorno de carro (va al principio de la línea) |
| \t | Salto de tabulación horizontal |
| \v | Salto de tabulación vertical |
| \' | Muestra una comilla simple |
| \" | Muestra una comilla doble |
| \\ | Muestra una barra invertida |
| \0 | Carácter nulo (NULL) |
| \7 | Emite un pitido (igual que \a) |
| \ddd | Un valor en octal |
| \xddd | Un valor en hexadecimal |

**Ejercicio propuesto**: (2.5.1.1) Crear un programa que pida al usuario que teclee cuatro letras y las muestre en pantalla juntas, pero en orden inverso, y entre comillas dobles. Por ejemplo si las letras que se teclean son a, l, o, h, escribiría "hola".

### 2.5.2. Introducción a las dificultades de las cadenas de texto

En el lenguaje C, no existe un tipo de datos para representar una cadena de texto. Eso supone que su manejo no sea tan sencillo como el de los números enteros, numeros reales y las letras. Deberemos tratarla como un bloque de varias letras. Por eso lo veremos más adelante.

# 3. Estructuras de control

## *3.1. Estructuras alternativas*

### 3.1.1. if

Vamos a ver cómo podemos comprobar si se cumplen condiciones. La primera construcción que usaremos será **"si ... entonces ..."**. El formato en C es

```
if (condición) sentencia;
```

Vamos a verlo con un ejemplo:

```
/*---------------------------*/
/*  Ejemplo en C nº 14:      */
/*  c014.c                   */
/*                           */
/*  Condiciones con if       */
/*                           */
/*  Curso de C,              */
/*    Nacho Cabanes          */
/*---------------------------*/

#include <stdio.h>

int main()
{
    int numero;

    printf("Escribe un número: ");
    scanf("%d", &numero);
    if (numero>0)  printf("El número es positivo.\n");

    return 0;
}
```

Nota: para comprobar si un valor numérico es mayor que otro, usamos el símbolo ">", como se ve en este ejemplo. Para ver si dos valores son iguales, usaremos dos símbolos de "igual": `if (numero==0)`. Las demás posibilidades las veremos algo más adelante. En todos los casos, la condición que queremos comprobar deberá indicarse entre paréntesis.

**Ejercicios propuestos**:

- (3.1.1.1) Crear un programa que pida al usuario un número entero y diga si es par (pista: habrá que comprobar si el resto que se obtiene al dividir entre dos es cero: if (x % 2 == 0) …).
- (3.1.1.2) Crear un programa que pida al usuario dos números enteros y diga cual es el mayor de ellos.
- (3.1.1.3) Crear un programa que pida al usuario dos números enteros y diga si el primero es múltiplo del segundo (pista: igual que antes, habrá que ver si el resto de la división es cero: a % b == 0).

### 3.1.2. if y sentencias compuestas

La "sentencia" que se ejecuta si se cumple la condición puede ser una sentencia simple o una compuesta. Las sentencias **compuestas** se forman agrupando varias sentencias simples entre llaves ( { y } ):

```
/*--------------------------*/
/*  Ejemplo en C nº 15:     */
/*  c015.c                  */
/*                          */
/*  Condiciones con if (2)  */
/*  Sentencias compuestas   */
/*                          */
/*  Curso de C,             */
/*    Nacho Cabanes         */
/*--------------------------*/

#include <stdio.h>

int main()
{
    int numero;

    printf("Escribe un número: ");
    scanf("%d", &numero);
    if (numero>0)
    {
        printf("El número es positivo.\n");
        printf("Recuerde que también puede usar negativos.\n");
    }  /* Aqui acaba el "if" */

    return 0;
}    /* Aqui acaba el cuerpo del programa */
```

En este caso, si el número es positivo, se hacen dos cosas: escribir un texto y luego... ¡escribir otro! (Claramente, en este ejemplo, esos dos "printf" podrían ser uno solo; más adelante iremos encontrando casos en lo que necesitemos hacer cosas "más serias" dentro de una sentencia compuesta).

Se pueden incluir siempre las llaves después de un "if", como medida de seguridad: un **fallo frecuente** es escribir una única sentencia tras "if", sin llaves, luego añadir una segunda sentencia y olvidar las llaves… de modo que la segunda orden no se ejecutará si se cumple la condición, sino siempre, como en este ejemplo:

```
/*--------------------------*/
/*  Ejemplo en C nº 15b:    */
/*  c015b.c                 */
/*                          */
/*  Condiciones con if (2b) */
/*  Sentencias compuestas   */
/*  incorrectas!!!          */
/*                          */
/*  Curso de C,             */
/*    Nacho Cabanes         */
```

```
/*--------------------------*/

#include <stdio.h>

int main()
{
    int numero;

    printf("Escribe un número: ");
    scanf("%d", &numero);
    if (numero>0)
        printf("El número es positivo.\n");
        printf("Recuerde que también puede usar negativos.\n");

    return 0;
}
```

En este caso, siempre se nos dirá "Recuerde que también puede usar negativos", incluso cuando hemos tecleado un número negativo. En un vistazo rápido, vemos las dos sentencias tabuladas a la derecha y tendemos a pensar que las dos dependen del "if", pero no es así. Se trata de un error difícil de detectar. Por eso, muchos autores recomiendan incluir siempre las llaves tras un "if", e incluso algún lenguaje de programación posterior a C obliga a que siempre sea así.

### 3.1.3. Operadores relacionales: <, <=, >, >=, ==, !=

Hemos visto que el símbolo ">" es el que se usa para comprobar si un número es mayor que otro. El símbolo de "menor que" también es sencillo, pero los demás son un poco menos evidentes, así que vamos a verlos:

| Operador | Operación |
|---|---|
| < | Menor que |
| > | Mayor que |
| <= | Menor o igual que |
| >= | Mayor o igual que |
| == | Igual a |
| != | No igual a (distinto de) |

Y un ejemplo:

```
/*--------------------------*/
/*  Ejemplo en C nº 16:     */
/*  c016.c                  */
/*                          */
/*  Condiciones con if (3)  */
/*                          */
/*  Curso de C,             */
/*    Nacho Cabanes         */
/*--------------------------*/

#include <stdio.h>

int main()
```

```
{
    int numero;

    printf("Escribe un número: ");
    scanf("%d", &numero);
    if (numero!=0)  printf("El número no es cero.\n");

    return 0;
}
```

Los operadores sólo se pueden usar tal y como aparecen en esa tabla. Por ejemplo, no es un operador válido "!<" para expresar que un número no es menor que otro.

**Ejercicio propuesto**:

- (3.1.3.1) Crear un programa que multiplique dos números enteros de la siguiente forma: pedirá al usuario un primer número entero. Si el número que se que teclee es 0, escribirá en pantalla "El producto de 0 por cualquier número es 0". Si se ha tecleado un número distinto de cero, se pedirá al usuario un segundo número y se mostrará el producto de ambos.
- (3.1.3.2) Crear un programa que pida al usuario dos números reales. Si el segundo no es cero, mostrará el resultado de dividir entre el primero y el segundo. Por el contrario, si el segundo número es cero, escribirá "Error: No se puede dividir entre cero".

### 3.1.4. if-else

Podemos indicar lo que queremos que ocurra en caso de que no se cumpla la condición, usando la orden "else" (en caso contrario), así:

```
/*---------------------------*/
/*  Ejemplo en C nº 17:      */
/*  c017.c                   */
/*                           */
/*  Condiciones con if (4)   */
/*  Uso de else              */
/*                           */
/*  Curso de C,              */
/*    Nacho Cabanes          */
/*---------------------------*/

#include <stdio.h>

int main()
{
    int numero;

    printf("Escribe un número: ");
    scanf("%d", &numero);
    if (numero>0)  printf("El número es positivo.\n");
        else  printf("El número es cero o negativo.\n");

    return 0;
}
```

Podríamos intentar evitar el uso de "else" si utilizamos un "if" a continuación de otro, así:

```
/*--------------------------*/
/*  Ejemplo en C nº 18:     */
/*  c018.c                  */
/*                          */
/*  Condiciones con if (5)  */
/*  Esquivando else         */
/*                          */
/*  Curso de C,             */
/*    Nacho Cabanes         */
/*--------------------------*/

#include <stdio.h>

int main()
{
    int numero;

    printf("Escribe un número: ");
    scanf("%d", &numero);
    if (numero>0)  printf("El número es positivo.\n");
    if (numero<=0) printf("El número es cero o negativo.\n");

    return 0;
}
```

Pero el comportamiento no es el mismo: en el primer caso (ejemplo 17) se mira si el valor es positivo; si no lo es, se pasa a la segunda orden, pero si lo es, el programa ya ha terminado. En el segundo caso (ejemplo 18), aunque el número sea positivo, se vuelve a realizar la segunda comprobación para ver si es negativo o cero, por lo que el programa es algo más lento.

Podemos enlazar los "if" usando "else", para decir "si no se cumple esta condición, mira a ver si se cumple esta otra":

```
/*--------------------------*/
/*  Ejemplo en C nº 19:     */
/*  c019.c                  */
/*                          */
/*  Condiciones con if (6)  */
/*  if encadenados          */
/*                          */
/*  Curso de C,             */
/*    Nacho Cabanes         */
/*--------------------------*/

#include <stdio.h>

int main()
{
    int numero;

    printf("Escriba un número: ");
    scanf("%d", &numero);
    if (numero < 0)
```

```
        printf("El número es negativo.\n");
    else
        if (numero == 0)
            printf("El número es cero.\n");
        else
            printf("El número es positivo.\n");

    return 0;
}
```

**Ejercicio propuesto**:

- (3.1.4.1) Mejorar la solución a los dos ejercicios del apartado anterior, usando "else".

### 3.1.5. Operadores lógicos: &&, ||, !

Estas condiciones se puede **encadenar** con "y", "o", etc., que se indican de la siguiente forma

| Operador | Significado |
|---|---|
| && | Y |
| \|\| | O |
| ! | No |

De modo que podremos escribir cosas como

```
if ((opcion==1) && (usuario==2))  ...
if ((opcion==1) || (opcion==3))   ...
if ((!(opcion==opcCorrecta)) || (tecla==ESC))   ...
```

(Suponiendo que ESC fuera una constante ya definida con anterioridad).

La siguiente forma de escribir una condición es **incorrecta** (y es un error muy frecuente en los que empiezan a programar en C):

```
if ((opcion1 || opcion2 == 3))   ...
```

porque la forma correcta de comprobar si la variable "opcion1" o bien "opcion2" valen "3" sería ésta:

```
if ((opcion1==3) || (opcion2==3))   ...
```

**Ejercicios propuestos**:

- (3.1.5.1) Crear un programa que pida una letra al usuario y diga si se trata de una vocal.
- (3.1.5.2) Crear un programa que pida al usuario dos números enteros y diga "Uno de los números es positivo", "Los dos números son positivos" o bien "Ninguno de los números es positivo", según corresponda.
- (3.1.5.3) Crear un programa que pida al usuario tres números reales y muestre cuál es el mayor de los tres.

- (3.1.5.4) Crear un programa que pida al usuario dos números enteros cortos y diga si son iguales o, en caso contrario, cuál es el mayor de ellos.

### 3.1.6. Cómo funciona realmente la condición en un "if"

Como suele ocurrir en C, lo que hemos visto tiene más miga de la que parece: una condición cuyo resultado sea "falso" nos devolverá un 0, y otra cuyo resultado sea "verdadero" devolverá el valor 1:

```
/*--------------------------*/
/*  Ejemplo en C nº 20:     */
/*  c020.c                  */
/*                          */
/*  Condiciones con if (7)  */
/*  Valor de verdad         */
/*                          */
/*  Curso de C,             */
/*    Nacho Cabanes         */
/*--------------------------*/

#include <stdio.h>

int main()
{
    int numero;

    printf("2==3 vale %d\n", 2==3);  /* Escribe 0 */
    printf("2!=3 vale %d\n", 2!=3);  /* Escribe 1 */

    return 0;
}
```

En general, si la "condición" de un if es algo que valga 0, se considerará que la condición es falsa (no se cumple), y si es algo distinto de cero, se considerará que la condición es verdadera (sí se cumple). Eso nos permite hacer cosas como ésta:

```
/*--------------------------*/
/*  Ejemplo en C nº 21:     */
/*  c021.c                  */
/*                          */
/*  Condiciones con if (8)  */
/*                          */
/*  Curso de C,             */
/*    Nacho Cabanes         */
/*--------------------------*/

#include <stdio.h>

int main() {
    int numero;

    printf("Escribe un número: ");
    scanf("%d", &numero);
    if (numero!=0)  /* Comparación normal */
        printf("El número no es cero.\n");
    if (numero)     /* Comparación "con truco" */
```

```
        printf("Y sigue sin ser cero.\n");

    return 0;
}
```

En este ejemplo, la expresión "if (numero)" se fijará en el valor de la variable "numero". Si es distinto de cero, se considerará que la condición es correcta, y se ejecutará la sentencia correspondiente (el "printf"). En cambio, si la variable "numero" vale 0, es considera que la condición es falsa, y no se sigue analizando.

En general, es preferible evitar este tipo de construcciones. Resulta mucho más legible algo como "if (numero!=0)" que "if(numero)".

### 3.1.7. El peligro de la asignación en un "if"

Cuidado con el operador de **igualdad**: hay que recordar que el formato es  if (a==b) ... Si no nos damos cuenta y escribimos  if (a=b)  estamos asignando a "a" el valor de "b".

Afortunadamente, la mayoría de los compiladores nos **avisan** con un mensaje parecido a "Possibly incorrect assignment" (que podríamos traducir por "posiblemente esta asignación es incorrecta") o "Possibly unintended assignment" (algo así como "es posible que no se pretendiese hacer esta asignación"). Aun así, sólo es un aviso, la compilación prosigue, y se genera un ejecutable, que puede que se comporte incorrectamente. Vamos a verlo con un ejemplo:

```
/*--------------------------*/
/*  Ejemplo en C nº 22:     */
/*  c022.c                  */
/*                          */
/*  Condiciones con if (9)  */
/*  Comportamiento          */
/*    incorrecto            */
/*                          */
/*  Curso de C,             */
/*    Nacho Cabanes         */
/*--------------------------*/

#include <stdio.h>

int main()
{
    int numero;

    printf("Escriba un número: ");
    scanf("%d", &numero);
    if (numero < 0)
        printf("El número es negativo.\n");
    else
    if (numero = 0)
        printf("El número es cero.\n");
    else
        printf("El número es positivo.\n");

    return 0;
}
```

En este caso, si tecleamos un número negativo, se comprueba la primera condición, se ve que es correcta y se termina sin problemas. Pero si se teclea cualquier otra cosa (0 o positivo), la expresión "if (numero=0)" no comprueba su valor, sino que le asigna un valor 0 (falso), por lo que siempre se realiza la acción correspondiente a el "caso contrario" (else): siempre se escribe "El número es positivo"... ¡aunque hayamos tecleado un 0!

Y si esto es un error, ¿por qué el compilador "avisa" en vez de parar y dar un error "serio"? Pues porque no tiene por qué ser necesariamente un error: podemos hacer

```
a = b;
if (a > 2) ...
```

o bien

```
if ((a=b) > 2) ...
```

Es decir, en la misma orden asignamos el valor y comparamos (algo parecido a lo que hacíamos con "b = ++a", por ejemplo). En este caso, la asignación dentro del "if" sería correcta.

Una forma simple de **evitar este riesgo** es escribir las comparaciones al revés: en vez de escribir

```
if (opcion == 2)  ...
```

podemos poner en primer lugar el número y en segundo lugar la variable:

```
if (2 == opcion)  ...
```

En este segundo caso, si olvidamos poner dos símbolos de "igual", se trata de una construcción no válida (no se puede cambiar el valor de un número) y el programa no compilaría. Lógicamente, este "truco" no sirve cuando comparamos dos variables, sólo cuando se trata de una variable y de un valor inmediato.

**Ejercicios resueltos**:

- ¿Qué escribiría en pantalla este fragmento de código?

  ```
  int x = 5;  if (x==5) printf("%d", x);
  ```

  Respuesta: x vale 5, luego se cumple la condición y se escribe un 5.

- ¿Qué escribiría en pantalla este fragmento de código?

  ```
  int x = 5;  if (x) printf("Si"); else printf("No");
  ```

Respuesta: x vale 5, luego la condición se evalúa como verdadera y se escribe Si.

- ¿Qué escribiría en pantalla este fragmento de código?

```
int x = 0;  if (x=5) printf("Si"); else printf("No");
```

Respuesta: no hemos escrito una comparación dentro de "if", sino una asignación. Por tanto, lo que hay escrito dentro del paréntesis se evalúa como verdadero (distinto de cero) y se escribe Si.

- ¿Qué escribiría en pantalla este fragmento de código?

```
int x = 5;  if (x=0) printf("Si"); else printf("No");
```

Respuesta: de nuevo, no hemos escrito una comparación dentro de "if", sino una asignación. Por tanto, lo que hay escrito dentro del paréntesis se evalúa como falso (cero) y se escribe No.

- ¿Qué escribiría en pantalla este fragmento de código?

```
int x = 0;  if (x==5) printf("Si") else printf("No");
```

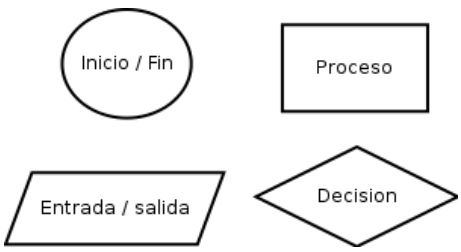
Respuesta: no compila, falta un punto y coma antes de "else".

### 3.1.8. Introducción a los diagramas de flujo

A veces puede resultar difícil ver claro donde usar un "else" o qué instrucciones de las que siguen a un "if" deben ir entre llaves y cuales no. Generalmente la dificultad está en el hecho de intentar teclear directamente un programa en C, en vez de pensar en el problema que se pretende resolver.

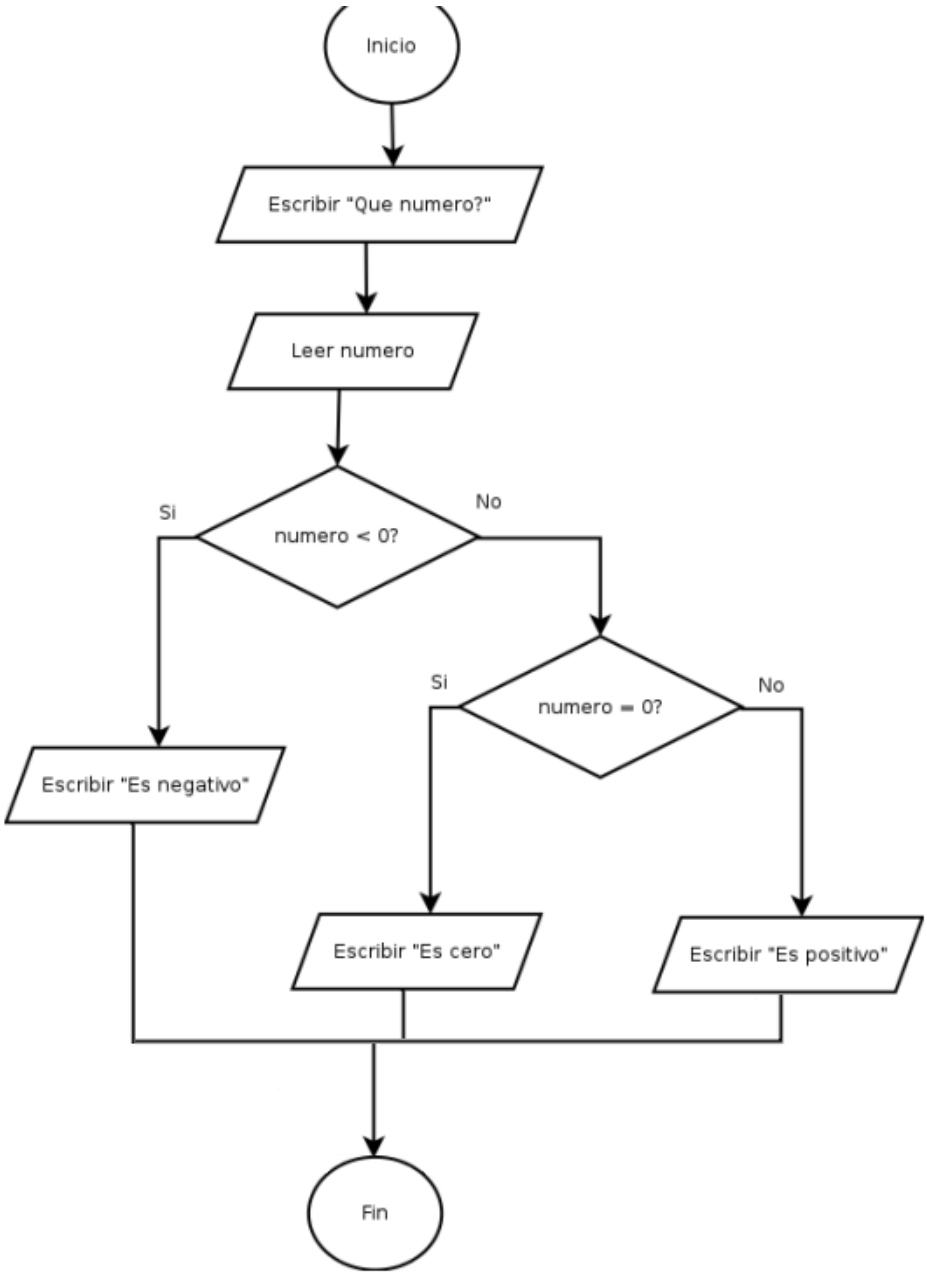
Para ayudarnos a centrarnos en el problema, existen notaciones gráficas, como los diagramas de flujo, que nos permiten ver mejor qué se debe hacer y cuando.

En primer lugar, vamos a ver los 4 elementos básicos de un diagrama de flujo, y luego los aplicaremos a un caso concreto.



El inicio o el final del programa se indica dentro de un círculo. Los procesos internos, como realizar operaciones, se encuadran en un rectángulo. Las entradas y salidas (escrituras en pantalla y lecturas de teclado) se indican con un paralelogramo que tenga su lados superior e inferior horizontales, pero no tenga verticales los otros dos. Las decisiones se indican dentro de un rombo.

Vamos a aplicarlo al ejemplo de un programa que pida un número al usuario y diga si es positivo, negativo o cero:




El paso de aquí al correspondiente programa en lenguaje C (el que vimos en el ejemplo 19) debe ser casi inmediato: sabemos como leer de teclado, como escribir en pantalla, y las decisiones serán un "if", que si se cumple ejecutará la sentencia que aparece en su salida "si"  y si no se cumple ("else") ejecutará lo que aparezca en su salida "no".

**Ejercicios propuestos:**

- (3.1.8.1) Crear el diagrama de flujo y la versión en C de un programa que dé al usuario tres oportunidades para adivinar un número del 1 al 10.
- (3.1.8.2) Crear el diagrama de flujo para el programa que pide al usuario dos números y dice si uno de ellos es positivo, si lo son los dos o si no lo es ninguno.
- (3.1.8.3) Crear el diagrama de flujo para el programa que pide tres números al usuario y dice cuál es el mayor de los tres.

### 3.1.9. Operador condicional: ?

En C hay otra forma de asignar un valor según se dé una condición o no. Es el "**operador condicional**" **?** : que se usa

```
condicion ?  valor1 : valor2;
```

y equivale a decir "si se cumple la condición, toma el valor v1; si no, toma el valor v2". Un ejemplo de cómo podríamos usarlo sería

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

que, aplicado a un programa sencillo, podría ser

```
/*--------------------------*/
/*  Ejemplo en C nº 23:     */
/*  c023.c                  */
/*                          */
/*  El operador condicional */
/*                          */
/*  Curso de C,             */
/*    Nacho Cabanes         */
/*--------------------------*/

#include <stdio.h>

int main()
{
    int a, b, mayor;

    printf("Escriba un número: ");
    scanf("%d", &a);
    printf("Escriba otro: ");
    scanf("%d", &b);
    mayor = (a>b) ?  a : b;
    printf("El mayor de los números es %d.\n", mayor);

    return 0;
}
```

Un segundo ejemplo, que sume o reste dos números según la opción que se escoja, sería:

```
/*--------------------------*/
/*  Ejemplo en C nº 24:     */
/*  c024.c                  */
```

```
/*                           */
/*  Operador condicional (2) */
/*                           */
/*  Curso de C,              */
/*    Nacho Cabanes          */
/*--------------------------*/

#include <stdio.h>

int main()
{
     int a, b, resultado;
     int operacion;

     printf("Escriba un número: ");
     scanf("%d", &a);
     printf("Escriba otro: ");
     scanf("%d", &b);
     printf("Escriba una operación (1 = resta; otro = suma): ");
     scanf("%d", &operacion);
     resultado = (operacion == 1) ?  a-b : a+b;
     printf("El resultado es %d.\n", resultado);

     return 0;
}
```

**Ejercicios propuestos:**

- (3.1.9.1) Crear un programa que use el operador condicional para mostrar un el valor absoluto de un número de la siguiente forma: si el número es positivo, se mostrará tal cual; si es negativo, se mostrará cambiado de signo.
- (3.1.9.2) Crear un programa que use el operador condicional para dar a una variable llamada "iguales" (entera) el valor 1 si los dos números que ha tecleado el usuario son iguales, o el valor 0 si son distintos.
- (3.1.9.3) Usar el operador condicional para calcular el mayor de dos números.

### 3.1.10. switch

Si queremos ver **varios posibles valores**, sería muy pesado tener que hacerlo con muchos "if" seguidos o encadenados.  La alternativa es la orden "switch", cuya sintaxis es

```
switch (expresión)
{
   case valor1: sentencia1;
          break;
   case valor2: sentencia2;
          sentencia2b;
          break;
    ...
   case valorN: sentenciaN;
          break;
  default:
          otraSentencia;
}
```

Es decir, se escribe tras "switch" la expresión a analizar, entre paréntesis. Después, tras varias órdenes "case" se indica cada uno de los valores posibles. Los pasos (porque pueden ser varios) que se deben dar si se trata de ese valor se indican a continuación, terminando con "break". Si hay que hacer algo en caso de que no se cumpla ninguna de las condiciones, se detalla tras "default".

Vamos con un ejemplo:

```
/*--------------------------*/
/*  Ejemplo en C nº 25:     */
/*  c025.c                  */
/*                          */
/*  La orden "switch"       */
/*                          */
/*  Curso de C,             */
/*    Nacho Cabanes         */
/*--------------------------*/

#include <stdio.h>

int main()
{
    char tecla;

    printf("Pulse una tecla y luego Intro: ");
    scanf("%c", &tecla);
    switch (tecla)
    {
        case ' ': printf("Espacio.\n");
                  break;
        case '1':
        case '2':
        case '3':
        case '4':
        case '5':
        case '6':
        case '7':
        case '8':
        case '9':
        case '0': printf("Dígito.\n");
                  break;
        default:  printf("Ni espacio ni dígito.\n");
    }

    return 0;
}
```

**Ejercicios propuestos:**

- (3.1.10.1) Crear un programa que lea una letra tecleada por el usuario y diga si se trata de una vocal, una cifra numérica o una consonante.
- (3.1.10.2) Crear un programa que lea una letra tecleada por el usuario y diga si se trata de un signo de puntuación, una cifra numérica o algún otro carácter.
- (3.1.10.3) Repetir los dos programas anteriores, empleando "if" en lugar de "switch".

## 3.2. Estructuras repetitivas

Hemos visto cómo comprobar condiciones, pero no cómo hacer que una cierta parte de un programa se repita un cierto número de veces o mientras se cumpla una condición (lo que llamaremos un "**bucle**"). En C tenemos varias formas de conseguirlo.

### 3.2.1. while

Si queremos hacer que una sección de nuestro programa se repita mientras se cumpla una cierta condición, usaremos la orden "while". Esta orden tiene dos formatos distintos, según comprobemos la condición al principio o al final.

En el primer caso, su sintaxis es

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

Es decir, la sentencia se repetirá **mientras** la condición sea cierta. Si la condición es falsa ya desde un principio, la sentencia no se ejecuta nunca. Si queremos que se repita más de una sentencia, basta agruparlas entre { y }. Como ocurria con "if", puede ser recomendable incluir siempre las llaves, aunque sea una única sentencia, para evitar errores posteriores difíciles de localizar.

Un ejemplo que nos diga si cada número que tecleemos es positivo o negativo, y que pare cuando tecleemos el número 0, podría ser:

```
/*--------------------------*/
/*  Ejemplo en C nº 26:     */
/*  c026.c                  */
/*                          */
/*  La orden "while"        */
/*                          */
/*  Curso de C,             */
/*    Nacho Cabanes         */
/*--------------------------*/

#include <stdio.h>

int main()
{
    int numero;

    printf("Teclea un número (0 para salir): ");
    scanf("%d", &numero);
    while (numero!=0)
    {
        if (numero > 0) printf("Es positivo\n");
            else printf("Es negativo\n");
        printf("Teclea otro número (0 para salir): ");
        scanf("%d", &numero);
    }

    return 0;
}
```

En este ejemplo, si se introduce 0 la primera vez, la condición es falsa y ni siquiera se entra al bloque del "while", terminando el programa inmediatamente.

Nota: si recordamos que una condición falsa se evalúa como el valor 0 y una condición verdadera como una valor distinto de cero, veremos que ese "while (numero != 0)" se podría abreviar como "while (numero)".

**Ejercicios propuestos:**

- (3.2.1.1) Crear un programa que pida al usuario su contraseña (numérica). Deberá terminar cuando introduzca como contraseña el número 4567, pero volvérsela a pedir tantas veces como sea necesario.
- (3.2.1.2) Crea un programa que escriba en pantalla los números del 1 al 10, usando "while".
- (3.2.1.3) Crea un programa que escriba en pantalla los números pares del 26 al 10 (descendiendo), usando "while".
- (3.2.1.4) Crear un programa calcule cuantas cifras tiene un número entero positivo (pista: se puede hacer dividiendo varias veces entre 10).

### 3.2.2. do ... while

Este es el otro formato que puede tener la orden "while": la condición se comprueba **al final**. El punto en que comienza a repetirse se indica con la orden "do", así:

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

Al igual que en el caso anterior, si queremos que se repitan varias órdenes (es lo habitual), deberemos encerrarlas entre llaves. Nuevamente, puede ser recomendable incluir siempre las llaves, como costumbre.

Como ejemplo, vamos a ver cómo sería el típico programa que nos pide una clave de acceso y nos nos deja entrar hasta que tecleemos la clave correcta. Eso sí, como todavía no sabemos manejar cadenas de texto, la clave será un número:

```
/*--------------------------*/
/*  Ejemplo en C nº 27:     */
/*  c027.c                  */
/*                          */
/*  La orden "do..while"    */
/*                          */
/*  Curso de C,             */
/*    Nacho Cabanes         */
/*--------------------------*/

#include <stdio.h>

int main()
{
    int valida = 711;
```

```
    int clave;

    do
    {
        printf("Introduzca su clave numérica: ");
        scanf("%d", &clave);
        if (clave != valida) printf("No válida!\n");
    }
    while (clave != valida);
    printf("Aceptada.\n");

    return 0;
}
```

En este caso, se comprueba la condición al final, de modo que se nos preguntará la clave al menos una vez. Mientras que la respuesta que demos no sea la correcta, se nos vuelve a preguntar. Finalmente, cuando tecleamos la clave correcta, el ordenador escribe "Aceptada" y termina el programa.

**Ejercicios propuestos:**

- (3.2.2.1) Crear un programa que pida números positivos al usuario, y vaya calculando la suma de todos ellos (terminará cuando se teclea un número negativo o cero).
- (3.2.2.2) Crea un programa que escriba en pantalla los números del 1 al 10, usando "do..while".
- (3.2.2.3) Crea un programa que escriba en pantalla los números pares del 26 al 10 (descendiendo), usando "do..while".
- (3.2.2.4) Crea un programa que pida al usuario su código de usuario (un número entero) y su contraseña numérica (otro número entero), y no le permita seguir hasta que introduzca como código 1024 y como contraseña 4567.

### 3.2.3. for

Ésta es la orden que usaremos habitualmente para crear partes del programa que **se repitan** un cierto número de veces. El formato de "for" es

```
for (valorInicial; CondiciónRepetición; Incremento)
  Sentencia;
```

Así, para **contar del 1 al 10**, tendríamos 1 como valor inicial, <=10 como condición de repetición, y el incremento sería de 1 en 1. Por tanto, el programa quedaría:

```
/*---------------------------*/
/*  Ejemplo en C nº 28:      */
/*  c028.c                   */
/*                           */
/*  Uso básico de "for"      */
/*                           */
/*  Curso de C,              */
/*    Nacho Cabanes          */
/*---------------------------*/
```

```
#include <stdio.h>

int main()
{
    int contador;

    for (contador=1; contador<=10; contador++)
        printf("%d ", contador);

    return 0;
}
```

Recordemos que "contador++" es una forma abreviada de escribir "contador=contador+1", de modo que en este ejemplo aumentamos la variable de uno en uno.

**Ejercicios propuestos:**

- (3.2.3.1) Crear un programa que muestre los números del 15 al 5, descendiendo (pista: en cada pasada habrá que descontar 1, por ejemplo haciendo `i--`).
- (3.2.3.2) Crear un programa que muestre los primeros ocho números pares (pista: en cada pasada habrá que aumentar de 2 en 2, o bien mostrar el doble del valor que hace de contador).

En un "for", realmente, la parte que hemos llamado "Incremento" no tiene por qué incrementar la variable, aunque ése es su uso más habitual. Es simplemente una orden que se ejecuta cuando se termine la "Sentencia" y antes de volver a comprobar si todavía se cumple la condición de repetición.

Por eso, si escribimos la siguiente línea:

```
for (contador=1; contador<=10; )
```

la variable "contador" no se incrementa nunca, por lo que nunca se cumplirá la condición de salida: nos quedamos encerrados dando vueltas dentro de la orden que siga al "for".

Un caso todavía más exagerado de algo a lo que se entra y de lo que no se sale sería la siguiente orden:

```
for ( ; ; )
```

Los bucles "for" se pueden **anidar** (incluir uno dentro de otro), de modo que podríamos escribir las tablas de multiplicar del 1 al 5 con:

```
/*--------------------------*/
/*  Ejemplo en C nº 29:     */
/*  c029.c                  */
/*                          */
/*  "for" anidados          */
/*                          */
```

```
/*  Curso de C,              */
/*    Nacho Cabanes          */
/*---------------------------*/

#include <stdio.h>

int main()
{
    int tabla, numero;

    for (tabla=1; tabla<=5; tabla++)
        for (numero=1; numero<=10; numero++)
            printf("%d por %d es %d\n", tabla, numero, tabla*numero);

    return 0;
}
```

En estos ejemplos que hemos visto, después de "for" había una única sentencia. Si queremos que se hagan varias cosas, basta definirlas como un **bloque** (una sentencia compuesta) encerrándolas entre llaves. Por ejemplo, si queremos mejorar el ejemplo anterior haciendo que deje una línea en blanco entre tabla y tabla, sería:

```
/*---------------------------*/
/*  Ejemplo en C nº 30:      */
/*  C30.C                    */
/*                           */
/*  "for" anidados (2)       */
/*                           */
/*  Curso de C,              */
/*    Nacho Cabanes          */
/*---------------------------*/

#include <stdio.h>

int main()
{
    int tabla, numero;

    for (tabla=1; tabla<=5; tabla++)
    {
        for (numero=1; numero<=10; numero++)
           printf("%d por %d es %d\n", tabla, numero, tabla*numero);
        printf("\n");
    }

    return 0;
}
```

Al igual que ocurría con "if" y con "while", suele ser buena costumbre **incluir siempre las llaves**, aunque haya una única orden que se repita en el "for", para evitar funcionamientos incorrectos si después hay que añadir más sentencias que deban repetirse y se olvida incluir las llaves en ese momento.

Para "contar" no necesariamente hay que usar **números**. Por ejemplo, podemos contar con letras así:

```
/*--------------------------*/
/*  Ejemplo en C nº 31:     */
/*  c031.c                  */
/*                          */
/*  "for" que usa "char"    */
/*                          */
/*  Curso de C,             */
/*    Nacho Cabanes         */
/*--------------------------*/

#include <stdio.h>

int main()
{
    char letra;

    for (letra='a'; letra<='z'; letra++)
        printf("%c", letra);

    return 0;
}
```

En este caso, empezamos en la "a" y terminamos en la "z", aumentando de uno en uno.

Si queremos contar de forma **decreciente**, o de dos en dos, o como nos interese, basta indicarlo en la condición de finalización del "for" y en la parte que lo incrementa:

```
/*--------------------------*/
/*  Ejemplo en C nº 32:     */
/*  c032.c                  */
/*                          */
/*  "for" que descuenta     */
/*                          */
/*  Curso de C,             */
/*    Nacho Cabanes         */
/*--------------------------*/

#include <stdio.h>

int main()
{
    char letra;

    for (letra='z'; letra>='a'; letra-=2)
        printf("%c", letra);

    return 0;
}
```

**Ejercicios propuestos:**

- (3.2.3.3) Crear un programa que muestre las letras de la Z (mayúscula) a la A (mayúscula, descendiendo).

- (3.2.3.4) Crear un programa que escriba en pantalla la tabla de multiplicar del 6.
- (3.2.3.5) Crear un programa que escriba en pantalla los números del 1 al 50 que sean múltiplos de 3 (pista: habrá que recorrer todos esos números y ver si el resto de la división entre 3 resulta 0).

## *3.3. Sentencia break: termina el bucle*

Podemos salir de un bucle "for" antes de tiempo con la orden "**break**":

```
/*--------------------------*/
/*  Ejemplo en C nº 33:     */
/*  c033.c                  */
/*                          */
/*  "for" interrumpido con  */
/*  "break"                 */
/*                          */
/*  Curso de C,             */
/*    Nacho Cabanes         */
/*--------------------------*/

#include <stdio.h>

int main()
{
    int i;

    for (i=0; i<=10; i++)
    {
        if (i==5) break;
        printf("%d ", i);
    }

    return 0;
}
```

El resultado de este programa es:

```
 0 1 2 3 4
```

(en cuanto se llega al valor 5, se interrumpe el "for", por lo que no se alcanza el valor 10).

**Ejercicio propuesto:**

- (3.3.1) Crear un programa que pida un número al usuario (entre 1 y 100) y muestre tantas letras A como indique ese número, usando "break" para terminar.

## *3.4. Sentencia continue: fuerza la siguiente iteración*

Podemos saltar alguna repetición de un bucle con la orden "**continue**":

```
/*--------------------------*/
/*  Ejemplo en C nº 34:     */
/*  c034.c                  */
```

```
/*                          */
/*  "for" interrumpido con  */
/*  "continue"              */
/*                          */
/*  Curso de C,             */
/*    Nacho Cabanes         */
/*--------------------------*/

#include <stdio.h>

int main()
{
    int i;

    for (i=0; i<=10; i++)
    {
       if (i==5) continue;
       printf("%d ", i);
    }

    return 0;
}
```

El resultado de este programa es:

0 1 2 3 4 6 7 8 9 10

En él podemos observar que no aparece el valor 5.

**Ejercicio propuesto:**

- (3.4.1) Crear un programa que pida un número al usuario (entre 1 y 20) y muestre los números el 1 al 20, excepto el indicado por el usuario, usando "continue" para evitar ese valor.

**Ejercicios resueltos**:

- ¿Qué escribiría en pantalla este fragmento de código?

  ```
  for (i=1; i<4; i++) printf("%d", i);
  ```

  Respuesta: los números del 1 al 3 (se empieza en 1 y se repite mientras sea menor que 4).

- ¿Qué escribiría en pantalla este fragmento de código?

  ```
  for (i=1; i>4; i++) printf("%d", i);
  ```

  Respuesta: no escribiría nada, porque la condición es falsa desde el principio.

- ¿Qué escribiría en pantalla este fragmento de código?

```
for (i=1; i<=4; i++); printf("%d", i);
```

Respuesta: escribe un 5, porque hay un punto y coma después del "for", de modo que repite cuatro veces una orden vacía, y cuando termina, "i" ya tiene el valor 5.

- ¿Qué escribiría en pantalla este fragmento de código?

```
for (i=1; i<4; ) printf("%d", i);
```

Respuesta: escribe "1" continuamente, porque no aumentamos el valor de "i", luego nunca se llegará a cumplir la condición de salida.

- ¿Qué escribiría en pantalla este fragmento de código?

```
for (i=1; ; i++) printf("%d", i);
```

Respuesta: escribe números continuamente, comenzando en uno y aumentando una unidad en cada pasada, pero si terminar nunca.

- ¿Qué escribiría en pantalla este fragmento de código?

```
for ( i= 0 ; i<= 4 ; i++) {
    if ( i == 2 ) continue ; printf( "%d " , i); }
```

Respuesta: escribe los números del 0 al 4, excepto el 2.

- ¿Qué escribiría en pantalla este fragmento de código?

```
for ( i= 0 ; i<= 4 ; i++) {
    if ( i == 2 ) break ; printf( "%d " , i); }
```

Respuesta: escribe los números 0 y 1 (interruumpe en el 2).

- ¿Qué escribiría en pantalla este fragmento de código?

```
for ( i= 0 ; i<= 4 ; i++) {
    if ( i == 10 ) continue ; printf( "%d " , i); }
```

Respuesta: escribe los números del 0 al 4, porque la condición del "continue" nunca se llega a dar.

- ¿Qué escribiría en pantalla este fragmento de código?

```
for ( i= 0 ; i<= 4 ; i++)
```

```
        if ( i == 2 ) continue ; printf( "%d " , i);
```

Respuesta: escribe 5, porque no hay llaves tras el "for", luego sólo se repite la orden "if".

## *3.5. Sentencia goto*

El lenguaje C también permite la orden "**goto**", para hacer saltos incondicionales. **Su uso indisciplinado está muy mal visto**, porque puede ayudar a hacer programas llenos de saltos, difíciles de seguir. Pero en casos concretos puede ser muy útil, por ejemplo, para salir de un bucle muy anidado (un "for" dentro de otro "for" que a su vez está dentro de otro "for": en este caso, "break" sólo saldría del "for" más interno).

El formato de "goto" es

```
goto donde;
```

y la posición de salto se indica con su nombre seguido de dos puntos (:)

```
donde:
```

como en el siguiente ejemplo:

```
/*--------------------------*/
/*  Ejemplo en C nº 35:     */
/*  c035.c                  */
/*                          */
/*  "for" y "goto"          */
/*                          */
/*  Curso de C,             */
/*    Nacho Cabanes         */
/*--------------------------*/

#include <stdio.h>

int main()
{
    int i, j;

    for (i=0; i<=5; i++)
        for (j=0; j<=20; j+=2)
        {
            if ((i==1) && (j>=7)) goto salida;
            printf("i vale %d y j vale %d.\n", i, j);
        }
    salida:
        printf("Fin del programa\n");

    return 0;
}
```

El resultado de este programa es:

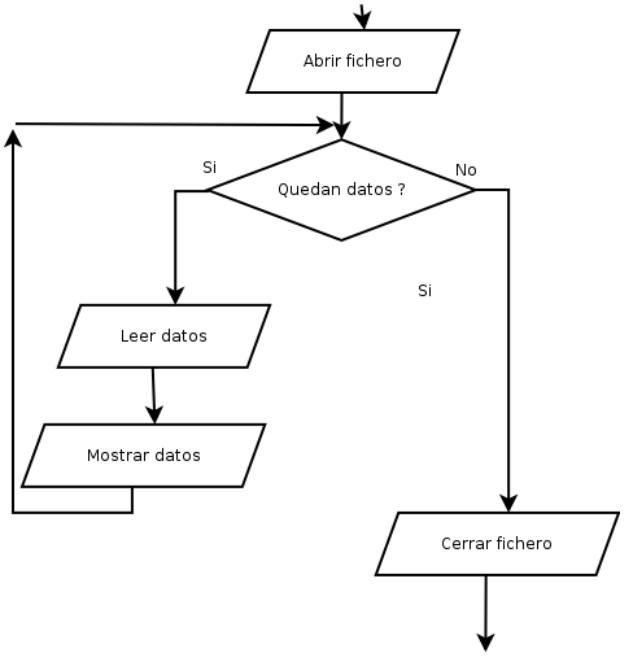
```
i vale 0 y j vale 0.
i vale 0 y j vale 2.
i vale 0 y j vale 4.
i vale 0 y j vale 6.
i vale 0 y j vale 8.
i vale 0 y j vale 10.
i vale 0 y j vale 12.
i vale 0 y j vale 14.
i vale 0 y j vale 16.
i vale 0 y j vale 18.
i vale 0 y j vale 20.
i vale 1 y j vale 0.
i vale 1 y j vale 2.
i vale 1 y j vale 4.
i vale 1 y j vale 6.
Fin del programa
```

Vemos que cuando i=1 y j>=7, se sale de los dos "for".

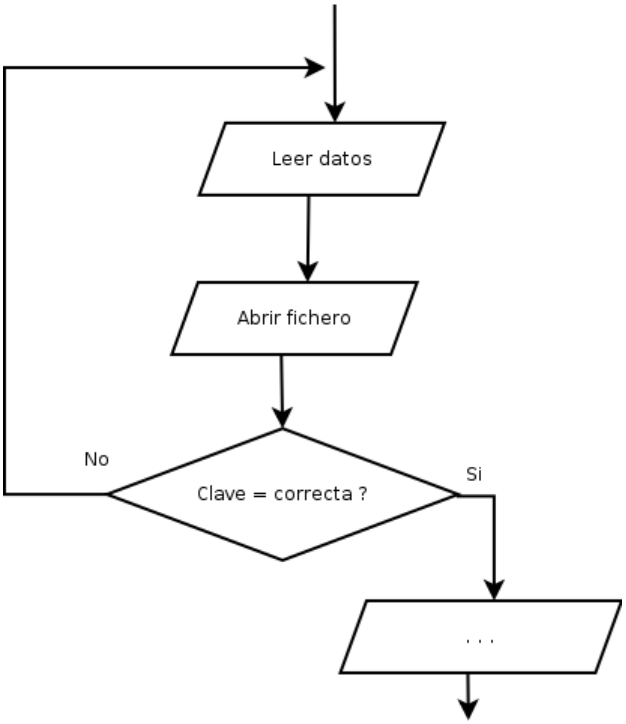
## *3.6. Más sobre diagramas de flujo. Diagramas de Chapin.*

Cuando comenzamos el tema, vimos cómo ayudarnos de los diagramas de flujo para plantear lo que un programa debe hacer. Si entendemos esta herramienta, el paso a C (o a casi cualquier otro lenguaje de programación es sencillo). Pero este tipo de diagramas es antiguo, no tiene en cuenta todas las posibilidades del lenguaje C (y de muchos otros lenguajes actuales). Por ejemplo, no existe una forma clara de representar una orden "switch", que equivaldría a varias condiciones encadenadas.

Por su parte, un bucle "while" se vería como una condición que hace que algo se repita (una flecha que vuelve hacia atrás, al punto en el que se comprobaba la condición):



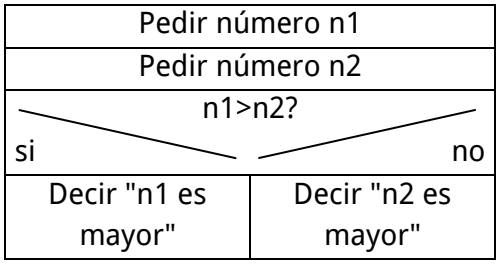

Y un "do..while" como una condición al final de un bloque que se repite:



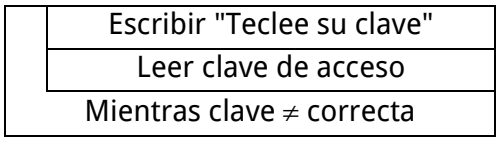

Aun así, existen otras notaciones más modernas y que pueden resultar más cómodas. Sólo comentaremos una: los diagramas de Chapin. En ellos se representa cada orden dentro de una caja:

| Pedir primer número |
|---|
| Pedir segundo número |
| Mostrar primer num+segundo num |

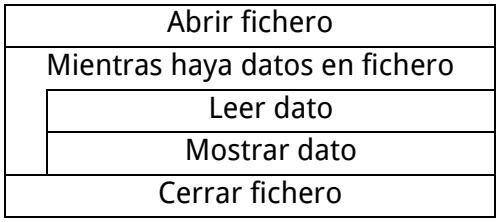
Las condiciones se indican diviendo las cajas en dos:




Y las condiciones repetitivas se indican dejando una barra a la izquierda, que marca qué es lo que se repite, tanto si la condición se comprueba al final (do..while):




como si se comprueba al principio (while):




En ambos casos, no existe una gráfica "clara" para los "for".

## *3.7. Recomendación de uso de los distintos tipos de bucle*

- En general, nos interesará usar "**while**" cuando puede que la parte repetitiva no se llegue a repetir nunca (por ejemplo: cuando leemos un fichero, si el fichero está vacío, no habrá datos que leer).
- De igual modo, "**do...while**" será lo adecuado cuando debamos repetir al menos una vez (por ejemplo, para pedir una clave de acceso, se le debe preguntar al menos una vez al usuario, o quizá más veces, si la teclea correctamente).
- En cuanto a "**for**", es equivalente a un "while", pero la sintaxis habitual de la oren "for" hace que sea especialmente útil cuando sabemos exactamente cuantas veces queremos que se repita (por ejemplo: 10 veces sería "`for (i=1; i<=10; i++)`").

**Ejercicios propuestos:**

- (3.7.1) Crear un programa que dé al usuario la oportunidad de adivinar un número del 1 al 100 (prefijado en el programa) en un máximo de 6 intentos. En cada pasada deberá avisar de si se ha pasado o se ha quedado corto.
- (3.7.2) Crear un programa que descomponga un número (que teclee el usuario) como producto de sus factores primos. Por ejemplo, 60 = 2 · 2 · 3 · 5

# 4. Entrada/salida básica

Vamos a ver con algo más de detalle las órdenes habituales de entrada y salida estándar: printf y scanf, que ya conocemos, putchar y getchar, que aun no habíamos visto. Todas ellas se encuentran definidas en <stdio.h>

## *4.1. printf*

Ya conocemos el manejo básico de la orden "printf":

printf( formato [, dato1, dato2, ...])

(el corchete indica que la cadena de texto de formato debe existir siempre, pero los datos adicionales son opcionales, pueden no aparecer).

Esta orden muestra un texto formateado en pantalla. Para usarla es necesario incluir <stdio.h> al principio de nuestro programa. Hemos visto cómo emplearla para mostrar número enteros, números reales y caracteres. Ahora vamos a ver más detalles sobre qué podemos mostrar y cómo hacerlo:

Los "**especificadores de formato**" que podemos usar son:

| | |
|---|---|
| c | Un único carácter |
| d | Un número entero decimal (en base 10) con signo |
| f | Un número real (coma flotante) |
| e | Un número real en notación exponencial, usando la "e" minúscula |
| E | Un número real en notación exponencial, usando la "E" mayúscula |
| g | Usa "e" o "f" (el más corto), con "e" minúscula |
| G | Usa "e" o "f" (el más corto), con "E" mayúscula |
| i | Un número entero decimal con signo |
| u | Un número entero decimal sin signo (unsigned) |
| h | Corto (modificador para un entero) |
| l | Largo (modificador para un entero) |
| x | Un número entero decimal sin signo en hexadecimal (base 16) |
| X | Un número entero decimal sin signo en hexadecimal en mayúsculas |
| o | Un número entero decimal sin signo en octal (base 8) |
| s | Una cadena de texto (que veremos en el próximo tema) |

Si usamos %% se mostrará el símbolo de porcentaje en pantalla.

Queda alguna otra posibilidad que todavía es demasiado avanzada para nosotros, y que comentaremos mucho más adelante, cuando hablemos de "punteros".

Además, las órdenes de formato pueden tener **modificadores**, que se sitúan entre el % y la letra identificativa del código.

- Si el modificador es un número, especifica la anchura mínima en la que se escribe ese argumento (por ejemplo: %5d).
- Si ese número empieza por 0, los espacios sobrantes (si los hay) de la anchura mínima se rellenan con 0 (por ejemplo: %07d).
- Si ese número tiene decimales, indica el número de dígitos totales y decimales si los que se va a escribir es un número (por ejemplo %5.2f), o la anchura mínima y máxima si se trata de una cadena de caracteres (como %10.10s).
- Si el número es negativo, la salida se justificará a la izquierda, dejando espacios en blanco al final (en caso contrario, si no se dice nada, se justifica a la derecha, dejando los espacios al principio).

Vamos a ver un ejemplo de todo esto:

```
/*--------------------------*/
/*  Ejemplo en C nº 36:     */
/*  c036.c                  */
/*                          */
/*  Detalles de "printf"    */
/*                          */
/*  Curso de C,             */
/*    Nacho Cabanes         */
/*--------------------------*/

#include <stdio.h>

int main()
{
    int   entero = 1234;
    int   enteroNeg = -1234;
    float real = 234.567;
    char  letra = 'E';
    int   contador;

    printf("El número entero vale %d en notación decimal,\n", entero);
    printf("  y %o en notación octal,\n", entero);
    printf("  y %x en notación hexadecimal,\n", entero);
    printf("  y %X en notación hexadecimal en mayúsculas,\n", entero);
    printf("  y %ld si le hacemos que crea que es entero largo,\n", entero);
    printf("  y %10d si obligamos a una cierta anchura,\n", entero);
    printf("  y %-10d si ajustamos a la izquierda.\n", entero);
    printf("El entero negativo vale %d\n", enteroNeg);
    printf("  y podemos hacer que crea que es positivo: %u (incorrecto).\n",
        enteroNeg);
    printf("El número real vale %f en notación normal\n", real);
    printf("  y %5.2f si limitamos a dos decimales,\n", real);
    printf("  y %e en notación científica (exponencial).\n", real);
    printf("La letra es %c y un texto es %s.\n", letra, "Hola");
    printf("  Podemos poner \"tanto por ciento\": 50%%.\n");

    return 0;
}
```

El resultado de este programa sería:

```
El número entero vale 1234 en notación decimal,
  y 2322 en notación octal,
  y 4d2 en notación hexadecimal,
  y 4D2 en notación hexadecimal en mayúsculas,
  y 1234 si le hacemos que crea que es entero largo,
  y       1234 si obligamos a una cierta anchura,
  y 1234       si ajustamos a la izquierda.
El entero negativo vale -1234
  y podemos hacer que crea que es positivo: 4294966062 (incorrecto).
El número real vale 234.567001 en notación normal
  y 234.57 si limitamos a dos decimales,
  y 2.345670e+002 en notación científica (exponencial).
La letra es E y el texto Hola.
  Podemos poner "tanto por ciento": 50%.
```

Casi todo el fácil de seguir, pero aun así hay vemos alguna cosa desconcertante...

Por ejemplo, ¿por qué el número real aparece como 234.567001, si nosotros lo hemos definido como 234.567? Hay que recordar que los números reales se almacenan con una cierta **pérdida de precisión**. En un "float" sólo se nos garantiza que unas 6 cifras sean correctas. Si mostramos el número con más de 6 cifras (estamos usando 9), puede que el resultado no sea totalmente correcto. Si esta pérdida de precisión es demasiado grande para el uso que queramos darle, deberemos usar otro tipo de datos, como **double**.

Lo de que el número negativo quede mal cuando lo intentamos escribir como positivo, también nos suena conocido: si el primer bit de un número con signo es uno, indica que es un número negativo, mientras que en un número positivo el primer bit es la cifra binaria más grande (lo que se conoce como "el bit más significativo"). Por eso, tanto el número -1234 como el 4294966062 se traducen en la **misma secuencia de ceros y unos**, que la sentencia "printf" interpreta de una forma u otra según le digamos que el número el positivo o negativo.

Como curiosidad, ese número 4294966062 sería un valor distinto (64302) si usáramos un compilador de 16 bits en vez de uno de 32, porque sería una secuencia de 16 ceros y unos, en vez de una secuencia de 32 ceros y unos.

Otra opción más avanzada es que si usamos un asterisco (*) para indicar la anchura o la precisión, los primeros datos de la lista serán los valores que se tomen para indicar la anchura y la precisión real que se usarán:

```
int minimo = 5;
int máximo = 10;
printf("%*.*s", minimo, maximo, "mensaje");
```

**Ejercicios propuestos:**

- (4.1.1) Un programa que pida al usuario un número entero y muestre sus equivalentes en formato hexadecimal y en octal. Deberá seguir pidiendo (y convirtiendo) números hasta que se introduzca 0.
- (4.1.2) Un programa que pida al usuario 2 números reales y muestre su división con 2 decimales (excepto si el segundo es 0; en ese caso, deberá decir "no se puede dividir").

## *4.2. scanf*

Como ya sabemos, el uso de "scanf" recuerda mucho al de "printf", salvo porque hay que añadir el símbolo & antes de los nombres de las variables en las que queremos almacenar los datos. Aun así, los códigos de formato no son exactamente los mismos. Vamos a ver los más importantes:

| | |
|---|---|
| c | Un único carácter |
| d | Un número entero decimal (base 10) con signo |
| D | Un entero largo en base 10 sin signo |
| f | Un número real (coma flotante) |
| e,E | Un número real en notación exponencial |
| g,G | Permite "e" o "f" |
| i | Un número entero con signo |
| I | Un número entero largo con signo |
| u | Un número entero decimal sin signo (unsigned) |
| U | Un número entero decimal largo sin signo (unsigned) |
| h | Corto (modificador, para un entero) |
| l | Largo (modificador, para un entero) |
| x | Un número entero sin signo en hexadecimal (base 16) |
| X | Un número entero largo sin signo en hexadecimal |
| o | Un número entero sin signo en octal (base 8) |
| O | Un número entero largo sin signo en octal (base 8) |
| s | Una cadena de texto (que veremos en el próximo tema) |

Como vemos, la diferencia más importante es que si en vez de un entero queremos un entero largo, se suele usar el mismo carácter escrito en **mayúsculas**.

Al igual que en "printf", se puede indicar un número antes del identificador, que nos serviría para indicar la **cantidad máxima** de caracteres a leer. Por ejemplo, "scanf("%10s", &texto)" nos permitiría leer un texto de un tamaño máximo de 10 letras.

Aun así, "scanf" a veces da resultados desconcertantes, por lo que no es la orden más adecuada cuando se pretende crear un entorno amigable. Más adelante veremos formas alternativas de leer del teclado.

**Ejercicios propuestos:**

- (4.2.1) Un programa que pida al usuario un número hexadecimal y muestre su equivalente en base 10.
- (4.2.2) Un programa que pida al usuario un número octal y muestre su equivalente en base 10.
- (4.2.3) Un programa que pida al usuario una letra, después le pida una segunda letra, y las muestre en el orden contrario al que se introdujeron.

## *4.3. putchar*

Es una forma sencilla de escribir un único carácter en la pantalla:

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

o si usamos una variable:

```
putchar(x);
```

**Ejercicios propuestos:**

- (4.3.1) Un programa que escriba las letras de la A (a mayúscula) a la Z (z mayúscula), usando "for" y "putchar".

## *4.4. getchar*

Lo habíamos usado desde un principio en algunos entornos de programación para Windows, como forma de detener momentáneamente la ejecución. Realmente es más que eso: lee el siguiente carácter que esté disponible en el buffer del teclado (una memoria intermedia que almacena todas las pulsaciones de teclas que hagamos):

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

Si no quedaran más letras por leer, el valor que nos daría es EOF, una constante predefinida que nos indicará el final de un fichero (End Of File) o, en este caso, el final de la entrada disponible por teclado. Se usaría así:

```
letra = getchar();
if (letra==EOF) printf("No hay más letras");
```

Vamos a ver un ejemplo del uso de getchar y de putchar:

```
/*--------------------------*/
/*  Ejemplo en C nº 37:     */
/*  c037.c                  */
/*                          */
/*  getchar y putchar       */
/*                          */
/*  Curso de C,             */
/*    Nacho Cabanes         */
/*--------------------------*/
```

```
#include <stdio.h>

int main()
{
    char letra1, letra2;

    printf("Introduce dos letras y pulsa Intro: ");
    letra1 = getchar();
    letra2 = getchar();
    printf("Has tecleado: ");
    putchar(letra1);
    printf(" y también %c", letra2);

    return 0;
}
```

Vemos que aunque "getchar" lea tecla a tecla, no se analiza lo que hemos tecleado hasta que **se pulsa Intro**. Si tecleamos varias letras, la primera vez que usemos getchar nos dirá cual era la primera, la siguiente vez que usemos getchar nos dirá la segunda letra y así sucesivamente.

En este ejemplo sólo leemos dos letras. Si se teclean tres o más, las últimas se pierden. Si se teclea una letra y se pulsa Intro, "letra1" será la letra tecleada... ¡y "letra2" será el Intro (el carácter '\n' de avance de línea)!

Estos **problemas** también los tenemos si intentamos leer letra a letra con "scanf("%c", ..." así que para hacer esto de forma fiable necesitaremos otras órdenes, que no son estándar en C sino que dependerán de nuestro compilador y nuestro sistema operativo, y que veremos más adelante.

Una alternativa sería una segunda orden "getchar" para absorber la pulsación de la tecla Intro:

```
tecla = getchar(); getchar();
```

O bien leer dos caracteres con "scanf" (la letra esperada y el Intro que queremos absorber):

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

**Ejercicios propuestos:**

- (4.4.1) Un programa que pida al usuario 4 letras (usando "getchar") y las muestre en orden inverso (por ejemplo, si las letras son "h o l a", escribiría "aloh").

# 5. Arrays y estructuras

## *5.1. Conceptos básicos sobre tablas*

### 5.1.1. Definición de una tabla y acceso a los datos

Una tabla, vector, matriz o **array** (que algunos autores traducen por "**arreglo**") es un conjunto de elementos, todos los cuales son del mismo tipo. Estos elementos tendrán todos el mismo nombre, y ocuparán un espacio contiguo en la memoria.

Por ejemplo, si queremos definir un grupo de 4 números enteros, usaríamos

```
int ejemplo[4];
```

Podemos acceder a cada uno de los valores individuales indicando su nombre (ejemplo) y el número de elemento que nos interesa, pero con una precaución: se empieza a numerar desde 0, así que en el caso anterior tendríamos 4 elementos, que serían ejemplo[0], ejemplo[1], ejemplo[2], ejemplo[3].

Como ejemplo, vamos a  definir un grupo de 5 números enteros y hallar su suma:

```
/*---------------------------*/
/*  Ejemplo en C nº 38:      */
/*  c038.c                   */
/*                           */
/*  Primer ejemplo de tablas */
/*                           */
/*  Curso de C,              */
/*    Nacho Cabanes          */
/*---------------------------*/

#include <stdio.h>

int main()
{
    int numero[5];           /* Un array de 5 números enteros */
    int suma;                /* Un entero que será la suma */

    numero[0] = 200;       /* Les damos valores */
    numero[1] = 150;
    numero[2] = 100;
    numero[3] = -50;
    numero[4] = 300;
    suma = numero[0] +     /* Y hallamos la suma */
        numero[1] + numero[2] + numero[3] + numero[4];
    printf("Su suma es %d", suma);
    /* Nota: esta es la forma más ineficiente e incómoda */
    /* Ya lo iremos mejorando */

    return 0;
}
```

**Ejercicios propuestos:**

- (5.1.1.1) Un programa que pida al usuario 4 números, los memorice (utilizando una tabla), calcule su media aritmética y después muestre en pantalla la media y los datos tecleados.
- (5.1.1.2) Un programa que pida al usuario 5 números reales y luego los muestre en el orden contrario al que se introdujeron.

### 5.1.2. Valor inicial de una tabla

Al igual que ocurría con las variables "normales", podemos dar valor a los elementos de una tabla al principio del programa. Será más cómodo que dar los valores uno por uno, como hemos hecho antes. Esta vez los indicaremos todos entre llaves, separados por comas:

```
/*--------------------------*/
/*  Ejemplo en C nº 39:     */
/*  c039.c                  */
/*                          */
/*  Segundo ejemplo de      */
/*  tablas                  */
/*                          */
/*  Curso de C,             */
/*    Nacho Cabanes         */
/*--------------------------*/

#include <stdio.h>
int main()
{
    int numero[5] =            /* Un array de 5 números enteros */
        {200, 150, 100, -50, 300};
    int suma;                  /* Un entero que será la suma */

    suma = numero[0] +    /* Y hallamos la suma */
        numero[1] + numero[2] + numero[3] + numero[4];
    printf("Su suma es %d", suma);
    /* Nota: esta forma es algo menos engorrosa, pero todavía no */
    /* está bien hecho.  Lo seguiremos mejorando */

    return 0;
}
```

**Ejercicios propuestos:**

- (5.1.2.1) Un programa que almacene en una tabla el número de días que tiene cada mes (supondremos que es un año no bisiesto), pida al usuario que le indique un mes (1=enero, 12=diciembre) y muestre en pantalla el número de días que tiene ese mes.
- (5.1.2.2) Un programa que almacene en una tabla el número de días que tiene cada mes (año no bisiesto), pida al usuario que le indique un mes (ej. 2 para febrero) y un día (ej. el día 15) y diga qué número de día es dentro del año (por ejemplo, el 15 de febrero sería el día número 46, el 31 de diciembre sería el día 365).

### 5.1.3. Recorriendo los elementos de una tabla

Es de esperar que exista una forma más cómoda de acceder a varios elementos de un array, sin tener siempre que repetirlos todos, como hemos hecho en

```
   suma = numero[0] + numero[1] + numero[2] + numero[3] + numero[4];
```

El "truco" consistirá en emplear cualquiera de las estructuras repetitivas que ya hemos visto (while, do-while, for), por ejemplo así:

```
    suma = 0;   /* Valor inicial */
    for (i=0; i<=4; i++)
        suma += numero[i];
```

En este caso, que sólo sumábamos 5 números, no hemos escrito mucho menos, pero si trabajásemos con 100, 500 o 1000 números, la ganancia en comodidad sí que está clara.

El fuente completo sería:

```
/*---------------------------*/
/*  Ejemplo en C nº 39:      */
/*  c039.c                   */
/*                           */
/*  Segundo ejemplo de       */
/*  tablas                   */
/*                           */
/*  Curso de C,              */
/*    Nacho Cabanes          */
/*---------------------------*/

#include <stdio.h>
int main()
{
    int numero[5] =            /* Un array de 5 números enteros */
        {200, 150, 100, -50, 300};

    suma = 0;   /* Valor inicial */
    for (i=0; i<=4; i++)
        suma += numero[i];

    printf("Su suma es %d", suma);
    return 0;
}
```

**Ejercicios propuestos:**

- (5.1.3.1) A partir del programa propuesto en 5.1.2, que almacenaba en una tabla el número de días que tiene cada mes, crear otro que pida al usuario que le indique la fecha, detallando el día (1 al 31) y el mes (1=enero, 12=diciembre), como respuesta muestre en pantalla el número de días que quedan hasta final de año.
- (5.1.3.2) Crear un programa que pida al usuario 10 números y luego los muestre en orden inverso (del último al primero).
- (5.1.3.3) Crear un programa que pida al usuario 10 números, calcule su media y luego muestre los que están por encima de la media.
- (5.1.3.4) Un programa que pida al usuario 10 números enteros y calcule (y muestre) cuál es el mayor de ellos.

## 5.2. Cadenas de caracteres

### 5.2.1. Definición. Lectura desde teclado

Para las **cadenas de texto**, la situación se complica un poco: se crean como "arrays" de caracteres. Están formadas por una sucesión de caracteres **terminada con un carácter nulo** (**\0**), de modo que tendremos que reservar una letra más de las que necesitamos. Por ejemplo, para guardar el texto "Hola" usaríamos "char saludo[5]".

Este carácter nulo lo utilizarán todas las órdenes estándar que tienen que ver con manejo de cadenas: las que las muestran en pantalla, las que comparan cadenas, las que dan a una cadena un cierto valor, etc. Por tanto, si no queremos usar esas funciones y sólo vamos a acceder letra a letra (como hemos hecho con los números en los últimos ejemplos) nos bastaría con "char saludo[4]", pero si queremos usar cualquiera de esta órdenes estándar (será lo habitual), deberemos tener la prudencia de reservar una letra más de las "necesarias", para ese carácter nulo, que indica el final de la cadena, y que todas esas órdenes utilizan para saber cuando deben terminar de manipular la cadena.

Un primer ejemplo que nos pidiese nuestro nombre y nos saludase sería:

```
/*---------------------------*/
/*  Ejemplo en C nº 40:      */
/*  c040.c                   */
/*                           */
/*  Primer ejemplo de        */
/*  cadenas de texto         */
/*                           */
/*  Curso de C,              */
/*    Nacho Cabanes          */
/*---------------------------*/

#include <stdio.h>

int main()
{
    char texto[40];         /* Para guardar hasta 39 letras */

    printf("Introduce tu nombre: ");
    scanf("%s", &texto);
    printf("Hola, %s\n", texto);

    return 0;
}
```

Dos comentarios:

- Si la cadena contiene espacios, se lee sólo **hasta el primer espacio**. Esto se puede considerar una ventaja o un inconveniente, según el uso que se le quiera dar. En cualquier caso, dentro de muy poco veremos cómo evitarlo si queremos.

- Siendo estrictos, **no hace falta el "&"** en "scanf" cuando estamos leyendo cadenas de texto (sí para los demás tipos de datos). Los motivos exactos los veremos más adelante, cuando hablemos de direcciones de memoria y de punteros. Pero este programa se podría haber escrito así:

```
/*--------------------------*/
/*  Ejemplo en C nº 41:     */
/*  c041.c                  */
/*                          */
/*  Segundo ejemplo de      */
/*  cadenas de texto: scanf */
/*  sin &                   */
/*                          */
/*  Curso de C,             */
/*    Nacho Cabanes         */
/*--------------------------*/

#include <stdio.h>

int main()
{
    char texto[40];         /* Para guardar hasta 39 letras */

    printf("Introduce tu nombre: ");
    scanf("%s", texto);
    printf("Hola, %s\n", texto);

    return 0;
}
```

**Ejercicios propuestos:**

- (5.2.1.1) Un programa que te pida tu nombre y una cifra numérica, y escriba tu nombre tantas veces como indique esa cifra numérica.
- (5.2.1.2) Un programa similar al anterior, pero que pida en primer lugar la cifra numérica y después tu nombre, y luego escriba el nombre tantas veces como indique esa cifra numérica.

### 5.2.2. Cómo acceder a las letras que forman una cadena

Podemos leer (o modificar) una de las letras de una cadena de igual forma que leemos o modificamos los elementos de cualquier tabla: el primer elemento será texto[0], el segundo será texto[1] y así sucesivamente:

```
/*--------------------------*/
/*  Ejemplo en C nº 42:     */
/*  c042.c                  */
/*                          */
/*  Tercer ejemplo de       */
/*  cadenas de texto:       */
/*  acceder letra a letra   */
/*                          */
/*  Curso de C,             */
/*    Nacho Cabanes         */
/*--------------------------*/
```

```c
#include <stdio.h>

int main()
{
    char texto[40];         /* Para guardar hasta 39 letras */

    printf("Introduce tu nombre: ");
    scanf("%s", texto);
    printf("Hola, %s. Tu inicial es %c\n", texto, texto[0]);

    return 0;
}
```

**Ejercicio propuesto:**

- (5.2.2.1) Un programa que pida al usuario que introduzca una palabra, cambie su primera letra por una "A" y muestre la palabra resultante.

### 5.2.3. Longitud de la cadena.

En una cadena que definamos como "char texto[40]" lo habitual es que realmente no ocupemos las 39 letras que podríamos llegar a usar. Si guardamos 9 letras (y el carácter nulo que marca el final), tendremos 30 posiciones que no hemos usado. Pero estas 30 posiciones generalmente contendrán "basura", lo que hubiera previamente en esas posiciones de memoria, porque el compilador las reserva para nosotros pero no las "limpia". Si queremos saber cual es la longitud real de nuestra cadena tenemos dos opciones:

- Podemos leer la cadena carácter por carácter desde el principio hasta que encontremos el carácter nulo (\0) que marca el final.

- Hay una orden predefinida que lo hace por nosotros, y que nos dice cuantas letras hemos usado realmente en nuestra cadena. Es "**strlen**", que se usa así:

```c
/*---------------------------*/
/*  Ejemplo en C nº 43:      */
/*  c043.c                   */
/*                           */
/*  Longitud de una cadena   */
/*                           */
/*  Curso de C,              */
/*    Nacho Cabanes          */
/*---------------------------*/

#include <stdio.h>
#include <string.h>

int main()
{
    char texto[40];

    printf("Introduce una palabra: ");
    scanf("%s", texto);
    printf("Has tecleado %d letras", strlen(texto));
```

```
    return 0;
}
```

Como es de esperar, si escribimos "Hola", esta orden nos dirá que hemos tecleado 4 letras (no cuenta el \0 que se añade automáticamente al final).

Si empleamos "strlen", o alguna de las otras órdenes relacionadas con cadenas de texto que veremos en este tema, debemos **incluir <string.h>**, que es donde se definen todas ellas.

**Ejercicios propuestos:**

- (5.2.3.1) Un programa que te pida tu nombre y lo muestre en pantalla separando cada letra de la siguiente con un espacio. Por ejemplo, si tu nombre es "Juan", debería aparecer en pantalla "J u a n".
- (5.2.3.2) Un programa que te pida tu nombre y lo muestre en pantalla separando al revés. Por ejemplo, si tu nombre es "Juan", debería aparecer en pantalla "nauJ".

### 5.2.4. Entrada/salida para cadenas: gets, puts

Hemos visto que si leemos una cadena de texto con "scanf", se paraba en el primer espacio en blanco y no seguía leyendo a partir de ese punto. Existen otras órdenes que están diseñadas específicamente para manejar cadenas de texto, y que nos podrán servir en casos como éste.

Para leer una cadena de texto (completa, sin parar en el primer espacio), usaríamos la orden "gets", así:

```
gets(texto);
```

De igual modo, para escribir un texto en pantalla podemos usar "puts", que muestra la cadena de texto y avanza a la línea siguiente:

```
puts(texto);
```

Sería equivalente a esta otra orden:

```
printf("%s\n", texto);
```

**Ejercicio propuesto:**

- (5.2.4.1) Un programa que te pida una frase y la muestre en pantalla sin espacios. Por ejemplo, si la frase es "Hola, como estás", debería aparecer en pantalla "Hola,comoestás".

Existe un **posible problema** cuando se mezcla el uso de "gets" y el de "scanf": si primero leemos un número, al usar "scanf("%d", ...", la variable numérica guardará el número... pero el "Intro" que pulsamos en el teclado después de introducir ese número queda esperando en el buffer (la memoria intermedia del teclado). Si a continuación leemos un segundo número, no hay problema, porque se omite ese Intro, pero si leemos una cadena de texto, ese Intro es aceptable,

porque representaría una cadena vacía. Por eso, cuando primero leemos un número y luego una cadena usando "gets", tendremos que "absorber" el Intro, o de lo contrario el texto no se leería correctamente. Una forma de hacerlo sería usando "getchar":

```
scanf("%d", &numero);
getchar();
gets(texto);
```

Además, existe un **problema adicional**: muchos compiladores que sigan el estándar C99 pueden dar el aviso de que "gets es una orden **no segura** y no debería utilizarse". En compiladores que sigan el estándar C11 (propuesto en diciembre de 2011), es posible que esta orden ni siquiera esté disponible. Se debe a que "gets" no comprueba que haya espacio suficiente para los datos que introduzca el usuario, lo que puede dar lugar a un error de desbordamiento y a que el programa se comporte de forma imprevisible. En un fuente de práctica, creado por un principiante, no es peligroso, pero sí en caso de un programa en C "profesional" que esté dando servicio a una página Web (por ejemplo), donde el uso de "gets" sí podría suponer una vulnerabilidad. Por eso, en el estándar C11 se propone una orden "gets_s", en la que se indica el nombre de la variable pero también el tamaño máximo permitido. En el próximo tema (ficheros, tema 6) veremos una alternativa segura a "gets" que sí se puede utilizar en cualquier compilador.

**Ejercicios propuestos:**

- (5.2.4.2) Un programa que te pida tu nombre (usando "gets") y una cifra numérica, y escriba tu nombre tantas veces como indique esa cifra numérica.
- (5.2.4.3) Un programa similar al anterior, pero que pida en primer lugar la cifra numérica y después tu nombre (con "gets"), y luego escriba el nombre tantas veces como indique esa cifra numérica.

### 5.2.5. Asignando a una cadena el valor de otra: strcpy, strncpy; strcat

Cuando queremos dar a una variable el valor de otra, normalmente usamos construcciones como a =2, o como a = b. Pero en el caso de las cadenas de texto, esta NO es la forma correcta, no podemos hacer algo como saludo="hola" ni algo como texto1=texto2. Si hacemos algo así, haremos que las dos cadenas estén en la misma posición de memoria, y que los cambios que hagamos a una de ellas se reflejen también en la otra. La forma correcta de guardar en una cadena de texto un cierto valor es:

```
strcpy (destino, origen);
```

Es decir, debemos usar una función llamada "**strcpy**" (string copy, copiar cadena), que se encuentra también en "string.h". Vamos a ver dos ejemplos de su uso:

```
strcpy (saludo, "hola");

strcpy (textoDefinitivo, textoProvisional);
```

Es nuestra responsabilidad que en la cadena de destino haya **suficiente espacio** reservado para copiar lo que queremos.  Si no es así, estaremos sobreescribiendo direcciones de memoria en las que no sabemos qué hay.

Para evitar este problema, tenemos una forma de indicar que queremos copiar sólo los primeros **n bytes** de origen, usando la función "**strncpy**", así:

```
    strncpy (destino, origen, n);
```

Vamos a ver un ejemplo, que nos pida que tecleemos una frase y guarde en otra variable sólo las 4 primeras letras:

```
/*--------------------------*/
/*  Ejemplo en C nº 44:     */
/*  c044.c                  */
/*                          */
/*  Tomar 4 letras de una   */
/*  cadena                  */
/*                          */
/*  Curso de C,             */
/*    Nacho Cabanes         */
/*--------------------------*/

#include <stdio.h>
#include <string.h>

int main()
{
    char texto1[40], texto2[40], texto3[10];

    printf("Introduce un frase: ");
    gets(texto1);

    strcpy(texto2, texto1);
    printf("Una copia de tu texto es %s\n", texto2);
    strncpy(texto3, texto1, 4);
    printf("Y sus 4 primeras letras son %s\n", texto3);

    return 0;
}
```

Finalmente, existe otra orden relacionada con estas dos: podemos **añadir** una cadena al final de otra (concatenarla), con

```
   strcat (destino, origen);
```

Vamos a ver un ejemplo de su uso, que nos pida nuestro nombre, nuestro apellido y cree una nueva cadena de texto que contenga los dos, separados por un espacio:

```
/*--------------------------*/
/*  Ejemplo en C nº 45:     */
/*  c045.c                  */
```

```
/*                          */
/*  Concatenar dos cadenas  */
/*                          */
/*  Curso de C,             */
/*    Nacho Cabanes         */
/*--------------------------*/

#include <stdio.h>
#include <string.h>

int main()
{
    char texto1[40], texto2[40], texto3[40];

    printf("Introduce tu nombre: ");
    gets(texto1);

    printf("Introduce tu apellido: ");
    gets(texto2);

    strcat(texto1, " ");     /* Añado un espacio al nombre */
    strcat(texto1, texto2);  /* Y luego el apellido */
    printf("Te llamas %s\n", texto1);

  return 0;
}
```

**Ejercicio propuesto:**

- (5.2.5.1) Un programa que te pida una palabra, y la almacene en la variable llamada "texto". Luego deberá pedir una segunda palabra, y añadirla al final de "texto". Finalmente, deberá pedir una tercera palabra, y guardarla en la variable "texto" y en otra variable llamada "texto2".

### 5.2.6. Comparando cadenas: strcmp

Para **comparar** dos cadenas alfabéticamente (para ver si son iguales o para poder ordenarlas, por ejemplo), usamos

```
strcmp (cadena1, cadena2);
```

Esta función devuelve un número entero, que será:

- 0 si ambas cadenas son iguales.
- Un número negativo, si cadena1 < cadena2.
- Un número positivo, si cadena1 > cadena2.

Hay que tener cuidado, porque las cadenas se comparan como en un diccionario, pero hay que tener en cuenta ciertas cosas:

- Al igual que en un diccionario, todas las palabras que empiecen por B se consideran "mayores" que las que empiezan por A (la B está a continuación de la A).

- Si dos cadenas empiezan por la misma letra (o las mismas letras), se ordenan basándose en la primera letra diferente, también al igual que en el diccionario (AMA es "mayor" que ALA).
- La primera diferencia está que en que se distingue entre mayúsculas y minúsculas. Para más detalles, en el código ASCII las mayúsculas aparecen antes que las minúsculas, así que las palabras escritas en mayúsculas se consideran "menores" que las palabras escritas en minúsculas. Por ejemplo, "ala" es menor que "hola", porque una empieza por "a" y la otra empieza por "h", pero "Hola" es menor que "ala" porque la primera empieza con una letra en mayúsculas y la segunda con una letra en minúsculas.
- La segunda diferencia es que el código ASCII estándar no incluye eñe, vocales acentuadas ni caracteres internacionales, así que estos caracteres "extraños" aparecen después de los caracteres "normales", de modo que "adiós" se considera "mayor" que "adiposo", porque la "o" acentuada está después de todas las letras del alfabeto inglés.

Vamos a ver un primer ejemplo que nos pida dos palabras y diga si hemos tecleado la misma las dos veces:

```
/*--------------------------*/
/*  Ejemplo en C nº 46:     */
/*  c046.c                  */
/*                          */
/*  Comparar dos cadenas    */
/*                          */
/*  Curso de C,             */
/*    Nacho Cabanes         */
/*--------------------------*/

#include <stdio.h>
#include <string.h>


int main()
{
    char texto1[40], texto2[40];

    printf("Introduce una palabra: ");
    gets(texto1);

    printf("Introduce otra palabra: ");
    gets(texto2);

    if (strcmp(texto1, texto2)==0)
        printf("Son iguales\n");
    else
        printf("Son distintas\n");

    return 0;
}
```

**En ningún caso** deberemos comparar dos cadenas usando "==", porque estaríamos comprobando si están en la misma posición de memoria, algo que practicamente nunca ocurrirá, de

modo que nuestro programa se comportará de forma incorrecta, dando como distintas dos cadenas que quizá sí contengan el mismo texto:

```
/*--------------------------*/
/*  Ejemplo en C nº 46b:    */
/*  c046b.c                 */
/*                          */
/*  Comparar dos cadenas    */
/*  de forma INCORRECTA     */
/*                          */
/*  Curso de C,             */
/*    Nacho Cabanes         */
/*--------------------------*/

#include <stdio.h>
#include <string.h>


int main()
{
    char texto1[40], texto2[40];

    printf("Introduce una palabra: ");
    gets(texto1);

    printf("Introduce otra palabra: ");
    gets(texto2);

    if (texto1 == texto2)  /* INCORRECTO ! */
        printf("Son iguales\n");
    else
        printf("Son distintas\n");

    return 0;
}
```

Podemos mejorarlo ligeramente para que nos diga qué palabra es "menor" de las dos:

```
/*--------------------------*/
/*  Ejemplo en C nº 47:     */
/*  c047.c                  */
/*                          */
/*  Comparar dos cadenas (2) */
/*                          */
/*  Curso de C,             */
/*    Nacho Cabanes         */
/*--------------------------*/

#include <stdio.h>
#include <string.h>

int main()
{
    char texto1[40], texto2[40];
    int comparacion;

    printf("Introduce una palabra: ");
```

```
    gets(texto1);

    printf("Introduce otra palabra: ");
    gets(texto2);

    comparacion = strcmp(texto1, texto2);

    if (comparacion==0)
        printf("Son iguales\n");
    else if (comparacion>0)
        printf("La primera palabra es mayor\n");
    else
        printf("La segunda palabra es mayor\n");

    return 0;
}
```

**Ejercicio propuesto:**

- (5.2.6.1) Crear un programa que pida al usuario su contraseña. Deberá terminar cuando introduzca como contraseña la palabra "clave", pero volvérsela a pedir tantas veces como sea necesario.
- (5.2.6.2) Crear un programa que pida al usuario su nombre y su contraseña, y no le permita seguir hasta que introduzca como nombre "Pedro" y como contraseña "Peter".

### 5.2.7. Otras funciones de cadenas: sprintf, sscanf, strstr, atoi...

Hay dos posibilidades más de las cadenas de texto que merece la pena comentar. Son las que nos ofrecen las funciones "sprintf" y "sscanf":

La funcion **"sprintf"** crea una cadena de texto a partir de una especificación de formato y unos ciertos parámetros, al igual que hace "printf", pero la diferencia está en que "printf" manda su salida a la pantalla, mientras que "sprintf" la deja guardada en una cadena de texto.

Por ejemplo, si escribimos

```
printf("El número %d multiplicado por 2 vale %d\n", 50, 50*2);
```

En pantalla aparecerá escrito

```
El número 50 multiplicado por 2 vale 100
```

Pues bien, si tenemos una cadena de texto que hayamos definido (por ejemplo) como char cadena[100] y escribimos

```
sprintf(cadena,"El número %d multiplicado por 2 vale %d\n", 50, 50*2);
```

Esta vez en pantalla no aparece nada escrito, sino que "cadena" pasa a contener el texto que antes habíamos mostrado.  Ahora ya podríamos escribir este texto con:

```
puts(cadena);
```

o bien con

```
printf("%s", cadena);
```

¿Qué utilidad tiene esta orden? Nos puede resultar cómoda cuando queramos formatear texto que no vaya a aparecer directamente en pantalla de texto, sino que lo vayamos a enviar a un fichero, o que queramos mostrar en pantalla gráfica, o enviar a través de una red mediante "sockets", por ejemplo.

Por otra parte **"sscanf"** es similar a "scanf", con la diferencia de que los valores para las variables no se leen desde el teclado, sino desde una cadena de texto

```
strcpy(cadena, "20 30");
sscanf(cadena, "%d %d", &primerNum, &segundoNum);
```

Será muy útil para analizar el contenido de ficheros de texto.

Nota: sscanf devuelve el número de valores que realmente se han detectado, de modo que podemos comprobar si ha tomado todos los que esperábamos o alguno menos (porque el usuario haya tecleado menos de los que esperábamos o porque alguno esté tecleado incorrectamente).

```
if (sscanf(cadena, "%d %d", &primerNum, &segundoNum)<2)
    printf("Debia teclear dos numeros");
```

**Ejercicio propuesto:** (5.2.7.1) Un programa que pida tu nombre, tu día de nacimiento y tu mes de nacimiento y lo junte todo en una cadena, separando el nombre de la fecha por una coma y el día del mes por una barra inclinada, así: "Juan, nacido el 31/12".

Una tercera orden que puede resultar útil más de una vez es "**strstr**". Permite comprobar si una cadena contiene un cierto texto. Devuelve NULL (un valor especial, que nos encontraremos cada vez más a partir de ahora) si no la contiene, y otro valor (no daremos más detalles por ahora sobre qué tipo de valor ni por qué) en casi de que sí la contenga:

```
if (strstr (frase, "Hola ") == NULL)
    printf("No has dicho la palabra Hola ");
```

Si queremos extraer el valor numérico de una cadena, no sólo tenemos "sscanf". Otras alternativas más sencillas (aunque menos fiables, porque devuelven 0 en caso de que no exista valor numérico), son "**atoi**" si queremos convertir a "int" y "**atof**" si queremos extraer un valor con decimales (un "float"):

```
valorEntero = atoi(texto);
```

```
valorReal = atof(texto);
```

**Nota:** éstas no son todas las posibilidades que tenemos para manipular cadenas, pero posiblemente sí son las más habituales. Hay otras que nos permiten buscar una letra dentro de una cadena (strchr), "dar la vuelta" a una cadena (strrev), etc. Según el compilador que usemos, podemos tener incluso funciones ya preparadas para convertir una cadena a **mayúsculas** (strupr) o a minúsculas (strlwr).

### 5.2.8. Valor inicial de una cadena de texto

Podemos dar un valor inicial a una cadena de texto, usando dos formatos distintos:

El formato "clásico" para dar valores a tablas:

```
  char nombre[50]= {'J','u','a','n'};
```

O bien un formato más compacto:

```
  char nombre[50]="Juan";
```

Pero cuidado con este último formato: hay que recordar que **sólo se puede usar cuando se declara** la variable, al principio del programa. Si ya estamos dentro del programa, deberemos usar necesariamente la orden "strcpy" para dar un valor a una cadena de texto.

## *5.3. Tablas bidimensionales*

Podemos declarar tablas de **dos o más dimensiones**. Por ejemplo, si queremos guardar datos de dos grupos de alumnos, cada uno de los cuales tiene 20 alumnos, tenemos dos opciones:

- Podemos usar `int datosAlumnos[40]` y entonces debemos recordar que los 20 primeros datos corresponden realmente a un grupo de alumnos y los 20 siguientes a otro grupo.
- O bien podemos emplear `int datosAlumnos[2][20]` y entonces sabemos que los datos de la forma `datosAlumnos[0][i]` son los del primer grupo, y los `datosAlumnos[1][i]` son los del segundo.

En cualquier caso, si queremos indicar valores iniciales, lo haremos entre llaves, igual que si fuera una tabla de una única dimensión. Vamos a verlo con un ejemplo de su uso:

```
/*--------------------------*/
/*  Ejemplo en C nº 48:     */
/*  c048.c                  */
/*                          */
/*  Array de dos dimensiones */
/*                          */
/*  Curso de C,             */
/*    Nacho Cabanes         */
/*--------------------------*/

#include <stdio.h>
```

```
int main()
{
    int notas[2][10] =
        { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10,
        11, 12, 13, 14, 15, 16, 17, 18, 19, 20 };

    printf("La nota del tercer alumno del grupos 1 es %d",
        notas[0][2]);

    return 0;
}
```

Este tipo de tablas son las que se usan también para guardar matrices, cuando hay que resolver problemas matemáticos más complejos.

También podemos usar arrays de dos dimensiones si queremos guardar una lista de cadenas de texto, como en este ejemplo:

```
/*--------------------------*/
/*  Ejemplo en C nº 49:     */
/*  c049.c                  */
/*                          */
/*  Array de cadenas        */
/*                          */
/*  Curso de C,             */
/*    Nacho Cabanes         */
/*--------------------------*/

#include <stdio.h>

int main()
{
    char mensajeError[5][80] = {
        "Fichero no encontrado",
        "El fichero no se puede abrir para escritura",
        "El fichero está vacío",
        "El fichero contiene datos de tipo incorrecto"
        "El fichero está siendo usado"
    };

    printf("El segundo mensaje de error es: %s",
        mensajeError[1]);
    printf("La primera letra del tercer mensaje de error es: %c",
        mensajeError[2][0]);

    return 0;
}
```

**Ejercicios propuestos**:

- (5.3.1) Un programa guarde los nombres de los meses. El usuario deberá indicar un número de mes (por ejemplo, 3) y se le mostrará el nombre de dicho mes (por ejemplo, Marzo).

- (5.3.2) Usar un array de 3 dimensiones para guardar los nombres de los meses en español e inglés. El usuario deberá indicar un número de mes (por ejemplo, 3) y se le mostrará el nombre de dicho mes en español (Marzo) y en inglés (March).

## *5.4. Arrays indeterminados.*

Si damos un valor inicial a un array, no será necesario que indiquemos su tamaño, porque el compilador lo puede saber contando la cantidad de valores hemos detallado, así:

```
int punto[] = {10, 0, -10};
char saludo[] = "hola";
char mensajes[][80] = {"Bienvenido", "Hasta otra"};
```

**Ejercicios propuestos:**

- (5.4.1) Un programa que pida 10 nombres y los memorice. Después deberá pedir que se teclee un nombre y dirá si se encuentra o no entre los 10 que se han tecleado antes. Volverá a pedir otro nombre y a decir si se encuentra entre ellos, y así sucesivamente hasta que se teclee "fin".
- (5.4.2) Un programa que prepare espacio para un máximo de 100 nombres (de un máximo de 80 letras cada uno). El usuario deberá ir introduciendo un nombre cada vez, hasta que se pulse Intro sin teclear nada, momento en el que dejarán de pedirse más nombres y se mostrará en pantalla la lista de los nombres que se han introducido hasta entonces.

## *5.5. Estructuras o registros*

### 5.5.1. Definición y acceso a los datos

Un **registro** es una agrupación de datos, los cuales no necesariamente son del mismo tipo. Se definen con la palabra "**struct**".

Para acceder a cada uno de los datos que forman el registro, tanto si queremos leer su valor como si queremos cambiarlo, se debe indicar el nombre de la variable y el del dato (o campo) separados por un punto:

```
/*---------------------------*/
/*  Ejemplo en C nº 50:      */
/*  c050.c                   */
/*                           */
/*  Registros (struct)       */
/*                           */
/*  Curso de C,              */
/*    Nacho Cabanes          */
/*---------------------------*/

#include <stdio.h>

int main()
{
    struct
    {
```

```
        char  inicial;
        int   edad;
        float nota;
    } persona;

    persona.inicial = 'J';
    persona.edad = 20;
    persona.nota = 7.5;
    printf("La edad es %d", persona.edad);

    return 0;
}
```

Como es habitual en C, para declarar la variable hemos indicado primero el tipo de datos (struct { ...} ) y después el nombre que tendrá esa variable (persona).

También podemos declarar primero cómo van a ser nuestros registros, y más adelante definir variables de ese tipo:

```
/*---------------------------*/
/*  Ejemplo en C nº 51:      */
/*  c051.c                   */
/*                           */
/*  Registros (2)            */
/*                           */
/*  Curso de C,              */
/*    Nacho Cabanes          */
/*---------------------------*/

#include <stdio.h>

struct datosPersona
{
    char  inicial;
    int   edad;
    float nota;
};

int main()
{
    struct datosPersona ficha;

    ficha.inicial = 'J';
    ficha.edad = 20;
    ficha.nota = 7.5;
    printf("La edad es %d", ficha.edad);

    return 0;
}
```

(Cuidado: como se ve en este ejemplo, la orden "struct" debe terminar con **punto y coma**, incluso si no declaramos todavía ninguna variable; de lo contrario, tendremos un error en tiempo de compilación).

**Ejercicios propuestos:**

- (5.5.1.1) Un "struct" que almacene datos de una canción en formato MP3: Artista, Título, Duración (en segundos), Tamaño del fichero (en KB). Un programa debe pedir los datos de una canción al usuario, almacenarlos en dicho "struct" y después mostrarlos en pantalla.

### 5.5.2. Arrays de estructuras

Hemos guardado varios datos de una persona. Se pueden almacenar los de **varias personas** si combinamos el uso de los "struct" con las tablas (arrays) que vimos anteriormente. Por ejemplo, si queremos guardar los datos de 100 alumnos podríamos hacer:

```
struct
{
    char  inicial;
    int   edad;
    float nota;
} alumnos[100];
```

La inicial del primer alumno sería "alumnos[0].inicial", y la edad del último sería "alumnos[99].edad".

**Ejercicios propuestos:**

- (5.5.2.1) Ampliar el programa del apartado 5.5.1, para que almacene datos de hasta 100 canciones. Deberá tener un menú que permita las opciones: añadir una nueva canción, mostrar el título de todas las canciones, buscar la canción que contenga un cierto texto (en el artista o en el título).
- (5.5.2.2) Un programa que permita guardar datos de "imágenes" (ficheros de ordenador que contengan fotografías o cualquier otro tipo de información gráfica). De cada imagen se debe guardar: nombre (texto), ancho en píxeles (por ejemplo 2000), alto en píxeles (por ejemplo, 3000), tamaño en Kb (por ejemplo 145,6). El programa debe ser capaz de almacenar hasta 700 imágenes (deberá avisar cuando su capacidad esté llena). Debe permitir las opciones: añadir una ficha nueva, ver todas las fichas (número y nombre de cada imagen), buscar la ficha que tenga un cierto nombre. (Nota: para algún ejercicio de mayor complejidad, como éste, puedes encontrar un ejemplo de solución

### 5.5.3. Estructuras anidadas

Podemos encontrarnos con un registo que tenga varios datos, y que a su vez ocurra que uno de esos datos esté formado por varios datos más sencillos. Para hacerlo desde C, incluiríamos un "struct" dentro de otro, declarando primero el "interior" y luego el "exterior", así:

```
/*--------------------------*/
/*  Ejemplo en C nº 52:     */
/*  c052.c                  */
/*                          */
/*  Registros anidados      */
/*                          */
/*  Curso de C,             */
```

```
/*    Nacho Cabanes          */
/*---------------------------*/

#include <stdio.h>

struct fechaNacimiento
{
    int  dia;
    int  mes;
    int anyo;
};

struct
{
    char  inicial;
    struct fechaNacimiento diaDeNacimiento;
    float nota;
} persona;

int main()
{
    persona.inicial = 'I';
    persona.diaDeNacimiento.mes = 8;
    persona.nota = 7.5;
    printf("La nota es %f", persona.nota);

    return 0;
}
```

**Ejercicios propuestos:**

- (5.5.3.1) Ampliar el programa del primer apartado de 5.5.2, para que el campo "duración" se almacene como minutos y segundos, usando un "struct" anidado que contenga a su vez estos dos campos.

## *5.6 Ejemplo completo*

Vamos a hacer un ejemplo completo que use tablas ("arrays"), registros ("struct") y que además manipule cadenas.

La idea va a ser la siguiente: Crearemos un programa que pueda almacenar datos de hasta 1000 ficheros (archivos de ordenador). Para cada fichero, debe guardar los siguientes datos: Nombre del fichero (max 40 letras), Tamaño (en KB, número de 0 a 2.000.000.000). El programa mostrará un menú que permita al usuario las siguientes operaciones:

1- Añadir datos de un nuevo fichero
2- Mostrar los nombres de todos los ficheros almacenados
3- Mostrar ficheros que sean de más de un cierto tamaño (por ejemplo, 2000 KB).
4- Ver todos los datos de un cierto fichero (a partir de su nombre)
5- Salir de la aplicación (como todavía no sabemos almacenar los datos, éstos se perderán).

No debería resultar difícil. Vamos a ver directamente una de las formas en que se podría plantear y luego comentaremos alguna de las mejoras que se podría (incluso se debería) hacer.

Una opción que podemos a tomar para resolver este problema es la de contar el número de fichas que tenemos almacenadas, y así podremos añadir de una en una. Si tenemos 0 fichas, deberemos almacenar la siguiente (la primera) en la posición 0; si tenemos dos fichas, serán la 0 y la 1, luego añadiremos en la posición 2; en general, si tenemos "n" fichas, añadiremos cada nueva ficha en la posición "n". Por otra parte, para revisar todas las fichas, recorreremos desde la posición 0 hasta la n-1, haciendo algo como

```
  for (i=0; i<=n-1; i++) { ... más órdenes ...}
```

o bien algo como

```
  for (i=0; i<n; i++) { ... más órdenes ...}
```

El resto del programa no es difícil: sabemos leer y comparar textos y números. Sólo haremos tres consideraciones:

- Los textos (nombre del fichero, por ejemplo) pueden contener espacios, por lo que usaremos "gets" en vez de "scanf".
- Es **"peligroso" mezclar órdenes "gets" y "scanf"**: si leemos un número con "scanf", la pulsación de la tecla "Intro" posterior se queda en el buffer del teclado, lo que puede provocar que después intentemos leer con "gets" un texto, pero sólo leamos esa pulsación de la tecla "Intro". Para evitarlo, los números los leeremos "en dos etapas": primero leeremos una cadena con "gets" y luego la convertiremos a número con "sscanf".
- Hemos limitado el número de fichas a 1000, así que, si nos piden añadir, deberíamos asegurarnos antes de que todavía tenemos hueco disponible.

Con todo esto, nuestro fuente quedaría así:

```
/*---------------------------*/
/*  Ejemplo en C nº 53:      */
/*  c053.c                   */
/*                           */
/*  Tabla con muchos struct  */
/*  y menu para manejarla    */
/*                           */
/*  Curso de C,              */
/*    Nacho Cabanes          */
/*---------------------------*/

#include <stdio.h>
#include <string.h>

struct{
   char nombreFich[41];     /* Nombre del fichero */
   unsigned long tamanyo;   /* El tamaño en bytes */
} fichas[1000];
```

```
int numeroFichas=0;   /* Número de fichas que ya tenemos */
int i;              /* Para bucles */

int opcion;          /* La opcion del menu que elija el usuario */

char textoTemporal[40]; /* Para cuando preguntemos al usuario */
unsigned long numeroTemporal;

int main()
{
    do
    {
        /* Menu principal */
        printf("Escoja una opción:\n");
        printf("1.- Añadir datos de un nuevo fichero\n");
        printf("2.- Mostrar los nombres de todos los ficheros\n");
        printf("3.- Mostrar ficheros que sean de mas de un cierto tamaño\n");
        printf("4.- Ver datos de un fichero\n");
        printf("5.- Salir\n");

        /* Para evitar problemas con datos mal introducidos,
          leemos con "gets" y luego lo filtramos con "sscanf" */
        gets (textoTemporal);
        sscanf(textoTemporal, "%d", &opcion);

        /* Hacemos una cosa u otra según la opción escogida */
        switch(opcion)
        {

            case 1: /* Añadir un dato nuevo */
                if (numeroFichas < 1000)  /* Si queda hueco */
                {
                    printf("Introduce el nombre del fichero: ");
                    gets(fichas[numeroFichas].nombreFich);
                    printf("Introduce el tamaño en KB: ");
                    gets(textoTemporal);
                    sscanf(textoTemporal,"%ld",&fichas[numeroFichas].tamanyo);
                    /* Y ya tenemos una ficha más */
                    numeroFichas++;
                } else   /* Si no hay hueco para más fichas, avisamos */
                    printf("Máximo de fichas alcanzado (1000)!\n");
                break;

            case 2: /* Mostrar todos */
                for (i=0; i<numeroFichas; i++)
                    printf("Nombre: %s; Tamaño: %ld Kb\n",
                        fichas[i].nombreFich, fichas[i].tamanyo);
                break;

            case 3: /* Mostrar según el tamaño */
                printf("¿A partir de que tamaño quieres que te muestre?");
                gets(textoTemporal);
                sscanf(textoTemporal, "%ld", &numeroTemporal);
                for (i=0; i<numeroFichas; i++)
                    if (fichas[i].tamanyo >= numeroTemporal)
                        printf("Nombre: %s; Tamaño: %ld Kb\n",
                            fichas[i].nombreFich, fichas[i].tamanyo);
                break;
```

```
            case 4: /* Ver todos los datos (pocos) de un fichero */
                printf("¿De qué fichero quieres ver todos los datos?");
                gets(textoTemporal);
                for (i=0; i<numeroFichas; i++)
                    if (strcmp(fichas[i].nombreFich, textoTemporal) == 0)
                        printf("Nombre: %s; Tamaño: %ld Kb\n",
                            fichas[i].nombreFich, fichas[i].tamanyo);
                break;

            case 5: /* Salir: avisamos de que salimos */
                printf("Fin del programa\n");
                break;

            default: /* Otra opcion: no válida */
                printf("Opción desconocida!\n");
                break;
        }
    }
    while (opcion != 5);  /* Si la opcion es 5, terminamos */

    return 0;
}
```

Funciona, y hace todo lo que tiene que hacer, pero es mejorable. Por supuesto, en un caso real es habitual que cada ficha tenga que guardar más información que sólo esos dos apartados de ejemplo que hemos previsto esta vez. Si nos muestra todos los datos en pantalla y se trata de muchos datos, puede ocurrir que aparezcan en pantalla tan rápido que no nos dé tiempo a leerlos, así que sería deseable que parase cuando se llenase la pantalla de información (por ejemplo, una pausa tras mostrar cada 25 datos). Por supuesto, se nos pueden ocurrir muchas más preguntas que hacerle sobre nuestros datos. Y además, cuando salgamos del programa se borrarán todos los datos que habíamos tecleado, pero eso es lo único "casi inevitable", porque aún no sabemos manejar ficheros.

**Ejercicios propuestos:**

- (5.6.1) Un programa que pida el nombre, el apellido y la edad de una persona, los almacene en un "struct" y luego muestre los tres datos en una misma línea, separados por comas.
- (5.6.2) Un programa que pida datos de 8 personas: nombre, dia de nacimiento, mes de nacimiento, y año de nacimiento (que se deben almacenar en una tabla de structs). Después deberá repetir lo siguiente: preguntar un número de mes y mostrar en pantalla los datos de las personas que cumplan los años durante ese mes. Terminará de repetirse cuando se teclee 0 como número de mes.
- (5.6.3) Un programa que sea capaz de almacenar los datos de hasta 50 personas: nombre, dirección, teléfono, edad (usando una tabla de structs). Deberá ir pidiendo los datos uno por uno, hasta que un nombre se introduzca vacío (se pulse Intro sin teclear nada). Entonces deberá aparecer un menú que permita:
    - Mostrar la lista de todos los nombres.
    - Mostrar las personas de una cierta edad.

    - Mostrar las personas cuya inicial sea la que el usuario indique.
    - Salir del programa

  (lógicamente, este menú debe repetirse hasta que se escoja la opción de "salir").
- (5.6.4) Mejorar la base de datos de ficheros (ejemplo 53) para que no permita introducir tamaños incorrectos (números negativos) ni nombres de fichero vacíos.
- (5.6.5) Ampliar la base de datos de ficheros (ejemplo 53) para que incluya una opción de búsqueda parcial, en la que el usuario indique parte del nombre y se muestre todos los ficheros que contienen ese fragmento (usando "strstr").
- (5.6.6) Ampliar la base de datos de ficheros (ejemplo 53) para que se pueda borrar un cierto dato (habrá que "mover hacia atrás" todos los datos que había después de ese, y disminuir el contador de la cantidad de datos que tenemos).
- (5.6.7) Mejorar la base de datos de ficheros (ejemplo 53) para que se pueda modificar un cierto dato a partir de su número (por ejemplo, el dato número 3). En esa modificación, se deberá permitir al usuario pulsar Intro sin teclear nada, para indicar que no desea modificar un cierto dato, en vez de reemplazarlo por una cadena vacía.
- (5.6.8) Ampliar la base de datos de ficheros (ejemplo 53) para que se permita ordenar los datos por nombre. Para ello, deberás buscar información sobre algún método de ordenación sencillo, como el "método de burbuja" (en el siguiente apartado tienes algunos), y aplicarlo a este caso concreto.

## *5.7 Ordenaciones simples*

Es muy frecuente querer ordenar datos que tenemos en un array. Para conseguirlo, existen varios algoritmos sencillos, que no son especialmente eficientes, pero son fáciles de programar. La falta de eficiencia se refiere a que la mayoría de ellos se basan en dos bucles "for" anidados, de modo que en cada pasada quede ordenado un dato, y se dan tantas pasadas como datos existen, de modo que para un array con 1.000 datos, podrían llegar a tener que hacerse un millón de comparaciones.

Existen ligeras mejoras (por ejemplo, cambiar uno de los "for" por un "while", para no repasar todos los datos si ya estaban parcialmente ordenados), así como métodos claramente más efectivos, pero más difíciles de programar, alguno de los cuales veremos más adelante.

Veremos tres de estos métodos simples de ordenación, primero mirando la apariencia que tiene el algoritmo, y luego juntando los tres en un ejemplo que los pruebe:

**Método de burbuja**

(Intercambiar cada pareja consecutiva que no esté ordenada)

```
Para i=1 hasta n-1
    Para j=i+1 hasta n
        Si A[i] > A[j]
          Intercambiar ( A[i], A[j])
```

(Nota: algunos autores hacen el bucle exterior creciente y otros decreciente, así:)

```
Para i=n descendiendo hasta 1
    Para j=2 hasta i
        Si A[j-1] > A[j]
          Intercambiar ( A[j-1], A[j])
```

**Selección directa**

(En cada pasada busca el menor, y lo intercambia al final de la pasada)

```
Para i=1 hasta n-1
    menor = i
    Para j=i+1 hasta n
        Si A[j] < A[menor]
            menor = j
    Si menor <> i
        Intercambiar ( A[i], A[menor])
```

**Inserción directa**

(Comparar cada elemento con los anteriores -que ya están ordenados- y desplazarlo hasta su posición correcta).

```
Para i=2 hasta n
      j=i-1
      mientras (j>=1) y (A[j] > A[j+1])
          Intercambiar ( A[j], A[j+1])
          j = j - 1
```

(Es mejorable, no intercambiando el dato que se mueve con cada elemento, sino sólo al final de cada pasada, pero no entraremos en más detalles).

**Ejercicios propuestos:**

- (5.7.1) Un programa que cree un array de 7 números enteros y lo ordene con cada uno de estos tres métodos, mostrando el resultado de los pasos intermedios.

# 6. Manejo de ficheros

## *6.1. Escritura en un fichero de texto*

Para manejar ficheros, siempre deberemos realizar tres operaciones básicas:

- Abrir el fichero.
- Leer datos de él o escribir datos en él.
- Cerrar el fichero.

Eso sí, no siempre podremos realizar esas operaciones, así que además tendremos que comprobar los posibles errores. Por ejemplo, puede ocurrir que intentemos abrir un fichero que realmente no exista, o que queramos escribir en un dispositivo que sea sólo de lectura.

Vamos a ver un ejemplo, que cree un fichero de texto y escriba algo en él:

```
/*--------------------------*/
/*  Ejemplo en C nº 55:     */
/*  c055.c                  */
/*                          */
/*  Escritura en un fichero  */
/*  de texto                */
/*                          */
/*  Curso de C,             */
/*    Nacho Cabanes         */
/*--------------------------*/

#include <stdio.h>

int main()
{
    FILE* fichero;

    fichero = fopen("prueba.txt", "wt");
    fputs("Esto es una línea\n", fichero);
    fputs("Esto es otra", fichero);
    fputs(" y esto es continuación de la anterior\n", fichero);
    fclose(fichero);

    return 0;
}
```

Hay varias cosas que comentar sobre este programa:

- **FILE** es el tipo de datos asociado a un fichero. Siempre aparecerá el asterisco a su derecha, por motivos que veremos más adelante (cuando hablemos de "punteros").
- Para abrir el fichero usamos "**fopen**", que necesita dos datos: el nombre del fichero y el modo de lectura. El modo de lectura estará formado por varias letras, de las cuales por ahora nos interesan dos: "w" indicaría que queremos escribir (*write*) del fichero, y "t" avisa de que se trata de un fichero de texto (*text*). Como abrimos el fichero para escribir

en él, se creará el fichero si no existía, y **se borrará** su contenido si ya existía (más adelante veremos cómo añadir a un fichero sin borrar su contenido).

- Para **escribir** en el fichero y para leer de él, tendremos órdenes muy parecidas a las que usábamos en pantalla. Por ejemplo, para escribir una cadena de texto usaremos "fputs", que recuerda mucho a "puts" pero con la diferencia de que no avanza de línea después de cada texto (por eso hemos añadido \n al final de cada frase).
- Finalmente, **cerramos** el fichero con "fclose".

**Ejercicios propuestos:**

- (6.1.1) Crea un programa que vaya leyendo las frases que el usuario teclea y las guarde en un fichero de texto llamado "registroDeUsuario.txt". Terminará cuando la frase introducida sea "fin" (esa frase no deberá guardarse en el fichero).

## *6.2. Lectura de un fichero de texto*

Si queremos **leer de un fichero**, los pasos son muy parecidos, sólo que lo abriremos para lectura (el modo de escritura tendrá una "r", de "read", en lugar de "w"), y leeremos con "fgets":

```
/*--------------------------*/
/*  Ejemplo en C nº 56:     */
/*  c056.c                  */
/*                          */
/*  Lectura de un fichero de */
/*  texto                   */
/*                          */
/*  Curso de C,             */
/*    Nacho Cabanes         */
/*--------------------------*/

#include <stdio.h>

int main()
{
    FILE* fichero;
    char nombre[80] = "c:\\autoexec.bat";
    char linea[81];

    fichero = fopen(nombre, "rt");

    if (fichero == NULL)
    {
      printf("No existe el fichero!\n");
      exit(1);
    }
    fgets(linea, 80, fichero);
    puts(linea);
    fclose(fichero);

    return 0;
}
```

En este fuente hay un par de cambios:

- En el nombre del fichero, hemos indicado un nombre algo más complejo. En estos casos, hay que recordar que si aparece alguna barra invertida (\), deberemos duplicarla, porque la barra invertida se usa para indicar ciertos códigos de control. Por ejemplo, \n es el código de avance de línea y \a es un pitido. El modo de lectura en este caso es "r" para indicar que queremos leer (*read*) del fichero, y "t" avisa de que es un fichero de texto.
- Para **leer** del fichero y usaremos "fgets", que se parece mucho a "gets", pero podemos limitar la longitud del texto que leemos (en este ejemplo, a 80 caracteres) desde el fichero. Esta cadena de texto **conservará** los caracteres de avance de línea.
- Si **no se consigue** abrir el fichero, se nos devolverá un valor especial llamado NULL (que también veremos con mayor detalle más adelante, cuando hablemos de punteros).
- La orden "**exit**" es la que nos permite abandonar el programa en un punto. La veremos con más detalle un poco más adelante.

**Ejercicios propuestos:**

- (6.2.1) Crea un programa que lea las tres primeras líneas del fichero creado en el apartado anterior y las muestre en pantalla. Si el fichero no existe, se deberá mostrar un aviso.

## *6.3. Lectura hasta el final del fichero*

Normalmente no querremos leer sólo una frase del fichero, sino procesar todo su contenido. Para ayudarnos, tenemos una orden que nos permite saber si ya hemos llegado al final del fichero. Es "feof" (EOF es la abreviatura de End Of File, fin de fichero).

Por tanto, nuestro programa deberá repetirse **mientras que no se acabe** el fichero, así:

```
/*--------------------------*/
/*  Ejemplo en C nº 57:     */
/*  c057.c                  */
/*                          */
/*  Lectura hasta el final  */
/*  de un fichero de texto  */
/*                          */
/*  Curso de C,             */
/*    Nacho Cabanes         */
/*--------------------------*/

#include <stdio.h>

int main()
{
    FILE* fichero;
    char nombre[80] = "c:\\autoexec.bat";
    char linea[81];

    fichero = fopen(nombre, "rt");

    if (fichero == NULL)
    {
```

```
        printf("No existe el fichero!\n");
        exit(1);
    }

    while (! feof(fichero))
    {
        fgets(linea, 80, fichero);
        if (! feof(fichero))
            puts(linea);
    }
    fclose(fichero);

  return 0;
}
```

Esa será la estructura básica de casi cualquier programa que deba leer un fichero completo, de principio a fin: abrir, comprobar que se ha podido acceder correctamente, leer con "while !(feof(…))" y cerrar.

**Ejercicios propuestos:**

- (6.3.1) Un programa que pida al usuario que teclee frases, y las almacene en el fichero "frases.txt". Acabará cuando el usuario pulse Intro sin teclear nada. Después deberá mostrar el contenido del fichero.
- (6.3.2) Un programa que pregunte un nombre de fichero y muestre en pantalla el contenido de ese fichero, haciendo una pausa después de cada 25 líneas, para que dé tiempo a leerlo. Cuando el usuario pulse "Intro", se mostrarán las siguientes 25 líneas, y así hasta que termine el fichero. (Pista: puedes usar un contador, volverlo a poner a cero tras cada 25 líneas o bien comprobar si has avanzado otras 25 líneas usando la operación "resto de la división", y hacer la pausa con "getchar()").

## *6.4. Ficheros con tipo*

Es frecuente que los ficheros que queramos manejar no sean de texto, pero que aun así tengan un formato bastante definido. Por ejemplo, podemos querer crear una agenda, en la que los datos de cada persona estén guardados en un "struct". En este caso, podríamos guardar los datos usando "fprintf" y "fscanf", análogos a "printf" y "scanf" que ya conocemos.

```
fprintf( fichero, "%40s%5d\n", persona.nombre, persona.numero);
fscanf( fichero, "%40s%5d\n", &persona.nombre, &persona.numero);
```

Como se puede ver en este ejemplo, suele ser recomendable indicar la anchura que debe tener cada dato cuando guardamos con "fprintf", para que se pueda recuperar después de la misma forma con "fscanf".

Aun así, "fscanf" tiene el mismo problema que "scanf": si leemos una cadena de texto, la considera terminada después del primer espacio en blanco, y lo que haya a continuación lo asignará a la siguiente cadena. Por eso, cuando manejemos textos con espacios, será preferible

usar "fgets" o bien otras dos órdenes para manejo de ficheros que veremos un poco más adelante.

**Ejercicios propuestos:**

- (6.4.1) Crear un "struct" que almacene los siguientes datos de una persona: nombre, edad, ciudad de residencia. Pedir al usuario esos datos de una persona y guardarlos en un fichero llamado "gente.dat". Cerrar el fichero, volverlo a abrir para lectura y mostrar los datos que se habían guardado.
- (6.4.2) Ampliar el programa anterior para que use un "array de structs", de forma que se puedan tener datos de 10 personas. Se deberá pedir al usuario los datos de las 10 personas y guardarlos en el fichero. Después se pedirá al usuario un número del 1 al 10 y se mostrarán los datos de la persona indicada por ese número, que se deberán leer de fichero (1 será la primera ficha, y 10 será la última). Por ejemplo, si el usuario indica que quiere ver los datos de la persona 3 (tercera), se deberá leer las dos primeras, ignorando su contenido, y después leer la tercera, que sí se deberá mostrar.
- (6.4.3) Una agenda que maneje los siguientes datos: nombre, dirección, tlf móvil, email, y día, mes y año de nacimiento (estos tres últimos datos deberán ser números enteros cortos). Deberá tener capacidad para 100 fichas. Se deberá poder añadir un dato nuevo, visualizar los nombres de las fichas existentes, o mostrar todos los datos de una persona (se preguntará al usuario cual es el nombre de esa persona que quiere visualizar). Al empezar el programa, leerá los datos de un fichero llamado "agenda.dat" (si existe). Al terminar, guardará todos los datos en ese fichero.

## *6.5 Leer y escribir letra a letra*

Si queremos leer o escribir **sólo una letra**, tenemos las órdenes "fgetc" y "fputc", que se usan:

```
letra = fgetc(fichero);
fputc (letra, fichero);
```

## *6.6 Modos de apertura*

Antes de seguir, vamos a ver las letras que pueden aparecer en el **modo de apertura** del fichero, para poder añadir datos a un fichero ya existente:

| Tipo | Significado |
|---|---|
| r | Abrir sólo para lectura. |
| w | Crear para escribir. Sobreescribe el fichero si existiera ya (borrando el original). |
| a | Añade al final del fichero si existe, o lo crea si no existe. |
| + | Se escribe a continuación de los modos anteriores para indicar que también queremos modificar. Por ejemplo: r+ permite leer y modificar el fichero. |
| t | Abrir en modo de texto. |
| b | Abrir en modo binario. |

**Ejercicios propuestos:**

- (6.6.1) Un programa que pida al usuario que teclee frases, y las almacene en el fichero "registro.txt", que puede existir anteriormente (y que no deberá borrarse, sino añadir al final de su contenido). Cada sesión acabará cuando el usuario pulse Intro sin teclear nada.
- (6.6.2) Crear un programa que pida al usuario pares de números enteros y escriba su suma (con el formato "20 + 3 = 23") en pantalla y en un fichero llamado "sumas.txt", que se encontrará en un subdirectorio llamado "resultados". Cada vez que se ejecute el programa, deberá añadir los nuevos resultados a continuación de los resultados de las ejecuciones anteriores.

## *6.7 Ficheros binarios*

Hasta ahora nos hemos centrado en los ficheros de texto, que son sencillos de crear y de leer. Pero también podemos manejar ficheros que contengan información de cualquier tipo.

En este caso, utilizamos "fread" para leer un bloque de datos y "fwrite" para guardar un bloque de datos. Estos datos que leamos se guardan en un **buffer** (una zona intermedia de memoria). En el momento en que se lean menos bytes de los que hemos pedido, quiere decir que hemos llegado al final del fichero.

En general, el manejo de "fread" es el siguiente:

```
cantidadLeida = fread(donde, tamañoDeCadaDato, cuantosDatos, fichero);
```

Por ejemplo, para leer 10 números enteros de un fichero (cada uno de los cuales ocuparía 4 bytes, si estamos en un sistema operativo de 32 bits), haríamos

```
int datos[10];
resultado = fread(&datos, 4, 10, fichero);
if (resultado < 10)
  printf("Había menos de 10 datos!");
```

Al igual que ocurría con "scanf", la variable en la que guardemos los datos se deberá indicar precedida del símbolo &, por motivos con detalle que veremos cuando hablemos sobre punteros. También al igual que pasaba con "scanf", si se trata de una cadena de caracteres (bien porque vayamos a leer una cadena de texto, o bien porque queramos leer datos de cualquier tipo pero con la intención de manejarlos byte a byte), como `char dato[500]` no será necesario indicar ese símbolo &, como en este ejemplo:

```
char cabecera [40];
resultado = fread(cabecera, 1, 40, fichero);
if (resultado < 40)
  printf("Formato de fichero incorrecto, no está toda la cabecera!");
else
```

```
   printf("El byte en la posición 5 es un %d", cabecera[4]);
```

## 6.8 Ejemplo: copiador de ficheros

Vamos a ver un ejemplo, que duplique un fichero de cualquier tipo (no necesariamente de texto), y después veremos las novedades:

```
/*---------------------------*/
/*  Ejemplo en C nº 58:      */
/*  c058.c                   */
/*                           */
/*  Copiador de ficheros     */
/*                           */
/*  Curso de C,              */
/*    Nacho Cabanes          */
/*---------------------------*/

#include <stdio.h>

FILE *fichOrg, *fichDest;        /* Los dos ficheros */
char buffer[2048];              /* El buffer para guardar lo que leo */
char nombreOrg[80],             /* Los nombres de los ficheros */
    nombreDest[80];
int cantidad;                  /* El número de bytes leídos */

int main()
{
    /* Accedo al fichero de origen */
    printf("Introduzca el nombre del fichero Origen: ");
    scanf("%s",nombreOrg);
    if ((fichOrg = fopen(nombreOrg, "rb")) == NULL)
    {
        printf("No existe el fichero origen!\n");
        exit(1);
    }

    /* Y ahora al de destino */
    printf("Introduzca el nombre del fichero Destino: ");
    scanf("%s",nombreDest);
    if ((fichDest = fopen(nombreDest, "wb")) == NULL)
    {
        printf("No se ha podido crear el fichero destino!\n");
        exit(1);
    }

    /* Mientras quede algo que leer */
    while (! feof(fichOrg) )
    {
        /* Leo datos: cada uno de 1 byte, todos los que me caben */
        cantidad = fread( buffer, 1, sizeof(buffer), fichOrg);
        /* Escribo tantos como haya leído */
        fwrite(buffer, 1, cantidad, fichDest);
    }

    /* Finalmente, cierro los ficheros */
    fclose(fichOrg);
    fclose(fichDest);
```

```
    return 0;
}
```

Los cambios con relación a lo que conocíamos de ficheros de texto son los siguientes:

- Los ficheros pueden no ser de texto, de modo que leemos uno como fichero binario (con "rb") y escribimos el otro también como fichero binario (con "wb").

- Definimos un buffer de 2048 bytes (2 K), para ir leyendo la información por bloques (y guardando después cada bloque en el otro fichero).

- En la misma línea intentamos abrir el fichero y comprobamos si todo ha sido correcto, con

  ```
  if ((fichOrg = fopen(nombreOrg, "rb")) == NULL)
  ```

  Esto puede resultar menos legible que hacerlo en dos líneas separadas, como hemos hecho hasta ahora, pero es más compacto, y, sobre todo, muy frecuente encontrarlo en "fuentes ajenos" más avanzados, como los que se puedan encontrar en Internet o cuando se programe en grupo con otras personas, de modo que he considerado adecuado incluirlo.

- A "fread" le decimos que queremos leer 2048 datos de 1 byte cada uno, y él nos devuelve la cantidad de bytes que ha leído realmente.  Para que el fuente sea más fácil de aplicar a otros casos en los que no sean bloques de 2048 bytes exactamente, suele ser preferible indicar que queremos leer el tamaño del bloque, usando "sizeof":

  ```
  cantidad = fread( buffer, 1, sizeof(buffer), fichOrg);
  ```

  Cuando la cantidad leida sea menos de 2048 bytes, es que el fichero se ha acabado (lo podemos comprobar mirando esta cantidad o con "feof").

- "fwrite" se maneja igual que fread: se le indica dónde están los datos, el tamaño de cada dato, cuantos datos hay que escribir y en qué fichero almacenarlos. En nuestro ejemplo, el número de bytes que debe escribir será el que haya leido:

  ```
  fwrite(buffer, 1, cantidad, fichDest);
  ```

**Ejercicios propuestos:**

- (6.8.1) Crear un "struct" que almacene los siguientes datos de una persona: nombre, edad, ciudad de residencia. Pedir al usuario esos datos de una persona y guardarlos en un fichero llamado "gente.dat", usando "fwrite". Cerrar el fichero, volverlo a abrir para lectura y mostrar los datos que se habían guardado, que se deben leer con "fread".

- (6.8.2) Ampliar el programa anterior para que use un "array de structs", de forma que se puedan tener datos de 10 personas. Se deberá pedir al usuario los datos de las 10 personas y guardarlos en el fichero, usando "fwrite". Después se pedirá al usuario un número del 1 al 10 y se mostrarán los datos de la persona indicada por ese número, que se deberán leer de fichero (1 será la primera ficha, y 10 será la última). Por ejemplo, si el usuario indica que quiere ver los datos de la persona 3 (tercera), se deberá leer las dos primeras (con "fread"), ignorando su contenido, y después leer la tercera, que sí se deberá mostrar.
- (6.8.3) Mejorar la agenda anterior (del apartado 6.4), para guardar y leer cada "ficha" (struct) de una vez, usando fwrite/fread y sizeof, como en el último ejemplo.

## *6.9 Acceder a cualquier posición de un fichero*

Cuando trabajamos con un fichero, es posible que necesitemos **acceder** directamente a una cierta posición del mismo. Para ello usamos "**fseek**", que tiene el formato:

```
int fseek(FILE *fichero, long posicion, int desde);
```

Como siempre, comentemos qué es cada cosa:

- Es de tipo "int", lo que quiere decir que nos va a devolver un valor, para que comprobemos si realmente se ha podido saltar a la dirección que nosotros le hemos pedido: si el valor es 0, todo ha ido bien; si es otro, indicará un error (normalmente, que no hemos abierto el fichero).
- "fichero" indica el fichero dentro de el que queremos saltar. Este fichero debe estar abierto previamente (con fopen).
- "posición" nos permite decir a qué posición queremos saltar (por ejemplo, a la 5010).
- "desde" es para poder afinar más: la dirección que hemos indicado con posic puede estar referida al comienzo del fichero, a la posición en la que nos encontramos actualmente, o al final del fichero (entonces posic deberá ser negativo). Para no tener que recordar que un 0 quiere decir que nos referimos al principio, un 1 a la posición actual y un 2 a la final, tenemos definidas las siguientes **constantes**:

```
SEEK_SET (0):  Principio
SEEK_CUR (1):  Actual
SEEK_END (2):  Final
```

Vamos a ver tres ejemplos de su uso:

- Ir a la posición 10 del fichero: fseek(miFichero, 10, SEEK_SET);
- Avanzar 5 posiciones a partir de la actual: fseek(miFichero, 5, SEEK_CUR);
- Ir a la posición 8 antes del final del fichero: fseek(miFichero, -8, SEEK_END);

Finalmente, si queremos saber en **qué posición** de un fichero nos encontramos, podemos usar "**ftell**(fichero)".

Esta orden nos permite saber también la **longitud** de un fichero: nos posicionamos primero al final con "fseek" y luego comprobamos con "ftell" en qué posición estamos:

```
fseek(fichero, 0, SEEK_END);
longitud = ftell(fichero);
```

**Ejercicios propuestos**:

- (6.9.1) Ampliar el programa anterior (el "array de structs" con 10 personas) para que el dato que indique el usuario se lea sin leer y descartar antes los que le preceden, sino que se salte directamente a la ficha deseada usando "fseek".

## *6.10 Ejemplo: leer información de un fichero BMP*

Ahora vamos a ver un ejemplo un poco más sofisticado: vamos a abrir un fichero que sea una imagen en formato BMP y a mostrar en pantalla si está comprimido o no.

Para eso necesitamos antes saber cómo se guarda la información en un fichero BMP, pero esto es algo fácil de localizar:

**FICHEROS .BMP**

Un fichero BMP está compuesto por las siguientes partes: una cabecera de fichero, una cabecera del bitmap, una tabla de colores y los bytes que definirán la imagen.

En concreto, los datos que forman la cabecera de fiochero y la cabecera de bitmap son los siguientes:

| TIPO DE INFORMACIÓN | POSICIÓN EN EL ARCHIVO |
|---|---|
| Tipo de fichero (letras BM) | 0-1 |
| Tamaño del archivo | 2-5 |
| Reservado | 6-7 |
| Reservado | 8-9 |
| Inicio de los datos de la imagen | 10-13 |
| Tamaño de la cabecera de bitmap | 14-17 |
| Anchura (píxeles) | 18-21 |
| Altura (píxeles) | 22-25 |
| Número de planos | 26-27 |
| Tamaño de cada punto | 28-29 |
| Compresión (0=no comprimido) | 30-33 |
| Tamaño de la imagen | 34-37 |
| Resolución horizontal | 38-41 |
| Resolución vertical | 42-45 |
| Tamaño de la tabla de color | 46-49 |

| | |
|---|---|
| Contador de colores importantes | 50-53 |

Con esta información nos basta para nuestro propósito: la compresión se indica en la posición 30 del fichero, ocupa 4 bytes (lo mismo que un "int" en los sistemas operativos de 32 bits), y si es un 0 querrá decir que la imagen no está comprimida.

Entonces lo podríamos comprobar así:

```
/*--------------------------*/
/*  Ejemplo en C nº 59:     */
/*  c059.c                  */
/*                          */
/*  Información sobre un    */
/*  fichero BMP (1)         */
/*                          */
/*  Curso de C,             */
/*    Nacho Cabanes         */
/*--------------------------*/

#include <stdio.h>

int main()
{
    char nombre[60];
    FILE* fichero;
    int compresion;

    puts("Comprobador de imágenes BMP\n");
    printf("Dime el nombre del fichero: ");
    gets(nombre);
    fichero = fopen(nombre, "rb");
    if (fichero==NULL)
        puts("No encontrado\n");
    else
    {
        fseek(fichero, 30, SEEK_SET);
        fread(&compresion, 1, 4, fichero);
        fclose(fichero);
        if (compression == 0)
            puts("Sin compresión");
        else
            puts ("BMP Comprimido ");
    }
    return 0;
}
```

Ya que estamos, podemos mejorarlo un poco para que además nos muestre el ancho y el alto de la imagen, y que compruebe antes si realmente se trata de un fichero BMP:

```
/*--------------------------*/
/*  Ejemplo en C nº 60:     */
/*  c060.c                  */
/*                          */
/*  Información sobre un    */
/*  fichero BMP (2)         */
```

```
/*                          */
/*  Curso de C,             */
/*    Nacho Cabanes         */
/*--------------------------*/

#include <stdio.h>

int main()
{
    char nombre[60];
    FILE* fichero;
    int compresion, ancho, alto;
    char marca1, marca2;

    puts("Comprobador de imágenes BMP\n");
    printf("Dime el nombre del fichero: ");
    gets(nombre);
    fichero = fopen(nombre, "rb");
    if (fichero==NULL)
    puts("No encontrado\n");
    else
    {
        marca1 = fgetc(fichero);  /* Leo los dos primeros bytes */
        marca2 = fgetc(fichero);
        if ((marca1 =='B') && (marca2 =='M'))  /* Si son BM */
        {
            printf("Marca del fichero: %c%c\n", marca1, marca2);
            fseek(fichero, 18, SEEK_SET);    /* Posición 18: ancho */
            fread(&ancho, 1, 4, fichero);
            printf("Ancho: %d\n", ancho);
            fread(&alto, 1, 4, fichero);     /* Siguiente dato: alto */
            printf("Alto: %d\n", alto);
            fseek(fichero, 4, SEEK_CUR);     /* 4 bytes después: compresión */
            fread(&compresion, 1, 4, fichero);
            fclose(fichero);
            switch (compresion)
            {
                case 0: puts("Sin compresión"); break;
                case 1: puts("Compresión RLE 8 bits"); break;
                case 2: puts("Compresión RLE 4 bits"); break;
            }
        }
        else
            printf("No parece un fichero BMP\n");  /* Si la marca no es BM */
    }
    return 0;
}
```

**Ejercicios propuestos:**

- (6.10.1) Mejorar la última versión de la agenda anterior (la que usa fwrite, fread y sizeof, del apartado 6.8) para que no lea todas las fichas a la vez, sino que lea una única ficha del disco cada vez que lo necesite, saltando a la posición en que se encuentra dicha ficha con "fseek".
- (6.10.2) Hacer un programa que muestre información sobre una imagen en formato GIF (se deberá localizar en Internet los detalles sobre dicho formato): versión, ancho de la imagen (en píxeles), alto de la imagen y cantidad de colores.

- (6.10.3) Hacer un programa que muestre información sobre una imagen en formato PCX: ancho de la imagen (en píxeles), alto de la imagen y cantidad de colores.
- (6.10.4) Mejorar la base de datos de ficheros (ejemplo 53) para que los datos se guarden en disco al terminar la sesión de uso, y se lean de disco cuando comienza una nueva sesión.
- (6.10.5) Mejorar la base de datos de ficheros (ejemplo 53) para que cada dato introducido se guarde inmediatamente en disco, sin esperar a que termine la sesión de uso. En vez de emplear un "array de structs", debe existir un solo "struct" en memoria cada vez, y para las búsquedas se recorra todo el contenido del fichero.
- (6.10.6) Mejorar el ejercicio anterior (ejemplo 53 ampliado con ficheros, que se manejan ficha a ficha) para que se pueda modificar un cierto dato a partir de su número (por ejemplo, el dato número 3). En esa modificación, se deberá permitir al usuario pulsar Intro sin teclear nada, para indicar que no desea modificar un cierto dato, en vez de reemplazarlo por una cadena vacía.

## *6.11. Ficheros especiales 1: la impresora*

Mandar algo a impresora desde C no es difícil (al menos en principio): en muchos sistemas operativos, la impresora es un dispositivo al que se puede acceder a través como si se tratara de un fichero.

Por ejemplo, en MsDos, se puede mostrar un fichero de texto en pantalla usando

```
TYPE DATOS.TXT
```

y lo mandaríamos a impresora si redirigimos la salida hacia el dispositivo llamado PRN:

```
TYPE DATOS.TXT > PRN:
```

De igual manera, desde C podríamos crear un programa que mandara información al fichero ficticio PRN: para escribir en impresora, así:

```
/*---------------------------*/
/*  Ejemplo en C nº 61:      */
/*  c061.c                   */
/*                           */
/*  Escritura en impresora   */
/*  (con MsDos)              */
/*                           */
/*  Curso de C,              */
/*    Nacho Cabanes          */
/*---------------------------*/

#include <stdio.h>

int main()
{
    FILE* impresora;
```

```
    impresora = fopen("prn:", "wt");
    fputs("Esto va a la impresora\n", impresora;);
    fclose(impresora);

    return 0;
}
```

(este mismo ejemplo debería funcionar desde muchas versiones de Windows, con bastante independencia de la impresora que tengamos instalada).

En Linux la idea sería la misma, pero el nombre de dispositivo sería "/dev/lp". Como inconveniente, normalmente sólo puede escribir en este dispositivo el administrador y los usuarios que pertenezcan a su grupo. Si pertenecemos a ese grupo, haríamos:

```
impresora = fopen("/dev/lp", "wt");
```

## 6.12. Ficheros especiales 2: salida de errores

Hemos comentado que en muchos sistemas operativos se puede usar el símbolo ">" para redirigir hacia "otro sitio" (la impresora o un fichero de texto, por ejemplo) la información que iba destinada originalmente a la pantalla. Esto funciona, entre otros, en Windows, MsDos y toda la familia de sistemas operativos Unix (incluido Linux).

Pero en el caso de Linux (y los Unix en general) podemos redirigir además los mensajes de error hacia otro sitio distinto del resto de mensajes (que iban destinados a pantalla). Esto se consigue con el símbolo "2>" :

```
calculaResultados  > valores.txt  2> errores.txt
```

Esta orden pone en marcha un programa llamado "calculaResultados", guarda en el fichero "valores.txt" los mensajes que normalmente aparecerían en pantalla, y guarda en el fichero "errores.txt" los mensajes de error.

Esta política de separar los mensajes de información y los mensajes de error es fácil de llevar a nuestros programas. Basta con que los mensajes de error no los mandemos a pantalla con órdenes como "printf", sino que los mandemos a un fichero especial llamado "stderr" (salida estándar de errores).

Por ejemplo, a la hora de intentar abrir un fichero podríamos hacer:

```
fichero = fopen("ejemplo.txt", "rt");
if (fichero == NULL)
  fprintf(stderr, "Fichero no encontrado!\n");
else
  printf("Accediendo al fichero...\n");
```

Si el usuario de nuestro programa no usa "2>", los mensajes de error le aparecerían en pantalla junto con cualquier otro mensaje, pero si se trata de un usuario avanzado, le estamos dando la posibilidad de analizar los errores cómodamente.

## 6.13. Un ejemplo de lectura y escritura: TAG de un MP3

Los ficheros de sonido en formato MP3 pueden contener información sobre el autor, el título, etc. Si la contienen (y se trata del formato "ID3, versión 1"), se encontraría a 128 bytes del final del fichero. Los primeros 3 bytes de esos 128 deberían ser las letras TAG. A continuación, tendríamos otros 30 bytes que serían el título de la canción, y otros 30 bytes que serían el nombre del autor. Con esto ya podríamos crear un programa que lea esa información de un fichero MP3 (si la contiene) e incluso que la modifique.

Estos textos (título, autor y otros) deberían estar rellenos con caracteres nulos al final, pero es algo de lo que no tenemos la certeza, porque algunas aplicaciones lo rellenan con espacios (es el caso de alguna versión de WinAmp). Por eso, leeremos los datos con "fread" y añadiremos un carácter nulo al final de cada uno.

Además, haremos que el programa nos muestre la información de varios ficheros: nos pedirá un nombre, y luego otro, y así sucesivamente hasta que pulsemos Intro sin teclear nada más.

```
/*---------------------------*/
/*  Ejemplo en C nº 62:      */
/*  c062.c                   */
/*                           */
/*  MP3 ID3 Tag v1           */
/*                           */
/*  Curso de C,              */
/*    Nacho Cabanes          */
/*---------------------------*/

#include <stdio.h>
#include <string.h>

int main()
{
    FILE* fich;
    char temp[31];
    int i;

    do
    {
        /* Pido el nombre del fichero */
        printf("\nEscribe el nombre del fichero MP3 a comprobar: ");
        gets(temp);

        /* Si no teclea nada, terminaré */
        if (strcmp(temp,"")==0)
            puts("\nAplicacion finalizada.");

        /* Si existe nombre, intento abrir */
        else if ( (fich=fopen(temp,"r+b"))!=NULL )
        {
            /* Si he podido abrir, muestro el nombre */
```

```
            printf("Nombre del fichero: %s\n",temp);
            /* Miro el tamaño del fichero */
            fseek(fich,0,SEEK_END);
            printf("Tamaño: %d\n",ftell(fich));
            /* A 128 bytes está la marca "TAG" */
            fseek(fich,-128,SEEK_END);
            fread(temp,3,1,fich);
            /* Son 3 letras, añado caracter nulo al final */
            temp[3]='\0';
            if (strcmp(temp,"TAG")!=0)
                puts("No se encontró información válida.");
            else
            {
                /* Si existe la marca, leo los datos */
                /* Primero, 30 letras de titulo */
                fread(temp,30,1,fich);
                temp[strlen(temp)]='\0';
                printf("Titulo: %s\n",temp);
                /* Luego 30 letras de autor */
                fread(temp,30,1,fich);
                temp[strlen(temp)]='\0';
                printf("Artista: %s\n",temp);
                /* Ahora vamos a modificar el titulo */
                printf("\nIntroduce el nuevo titulo: ");
                gets(temp);
                /* Lo rellenamos con ceros, para seguir el estándar */
                for (i=strlen(temp); i<=29; i++)
                    temp[i]='\0';
                /* Y lo guardamos en su posición */
                fseek(fich,-125,SEEK_END);
                fwrite(&temp, 30, 1, fich);
                printf("Titulo actualizado.\n");
                fclose(fich);
            } /* Fin del "else" de MP3 con informacion */
        } /* Fin del "else" de fichero existente */
        else puts("No se pudo abrir el fichero\n");
    } /* Fin del "do..while" que repite hasta que no se escriba nombre */
    while (strcmp(temp,"")!=0);

    return 0;
}
```

## *6.14. Evitar los problemas de "gets"*

Como comentamos en el tema 5, el uso de "gets" está desaconsejado a partir del estándar C99 (de 1999), por tratarse de una orden poco segura, pero esta orden ni siquiera existirá en los compiladores que sigan el estándar C11 (de diciembre de 2011). Una alternativa, en compiladores modernos, es usar "get_s" que recibe como segundo parámetro el tamaño máximo del texto. Otra alternativa, más segura que gets, que permite que la cadena contenga espacios, y que funcionará en cualquier compilador de C, es usar "**fgets**".

"fgets" espera 3 parámetros: la variable en la que se guardará el texto, la anchura máxima y el fichero desde el que leer. El "truco" para usar "fgets" para leer de teclado es indicar "stdin"

(standard in, entrada estándar) como nombre que fichero, que es un identificador predefinido en el sistema:

```
fgets(nombre, 20, stdin);
```

Eso sí, "fgets" tiene un problema para este uso: conserva el avance de línea (\n). Por eso, habrá que eliminarlo "a mano" (si la cadena no está vacía y si realmente termina en avance de línea, porque puede no ser así si la cadena era más larga que lo esperado y se ha truncado):

```
if ((strlen(nombre)>0) && (nombre[strlen (nombre) - 1] == '\n'))
    nombre[strlen (nombre) - 1] = '\0';
```

Con esas dos órdenes podríamos evitar los problemas de "gets" de una forma que se comporte bien en cualquier compilador, usando "fgets" para leer de teclado como si fuera un fichero.

No es la única forma: hay autores que prefieren no usar ficheros sino un formato avanzado de "scanf", en el que se le indica que acepte todo hasta llegar a un avance de línea, pero esto puede no ser admitido por todos los compiladores:

```
scanf("%[^\n]s",nombre);
```

Para evitar problemas de desbordamiento, deberíamos indicar la anchura máxima:

```
scanf("%20[^\n]s",nombre);
```

También podemos delimitar los caracteres admisibles (nuevamente, hay que tener presente que quizá no todos los compiladores lo permitan):

```
scanf("%10[0-9a-zA-Z ]s", nombre);
```

Muchos expertos recomiendan no usar directamente "gets" ni "scanf", ni siquiera para leer datos numéricos, sino hacer la lectura en una cadena de texto con "fgets" y luego extraer la información de ella con "sscanf".

**Ejercicio propuesto:**

- Pedir al usuario dos números y la operación (suma o resta) a realizar con ellos, y mostrar el resultado de dicha operación. Utilizar "fgets" y "sscanf" para la lectura de teclado.

# 7. Introducción a las funciones

## *7.1. Diseño modular de programas: Descomposición modular*

Hasta ahora hemos estado pensando los pasos que deberíamos dar para resolver un cierto problema, y hemos creado programas a partir de cada uno de esos pasos. Esto es razonable cuando los problemas son sencillos, pero puede no ser la mejor forma de actuar cuando se trata de algo más complicado.

A partir de ahora vamos a empezar a intentar descomponer los problemas en trozos más pequeños, que sean más fáciles de resolver. Esto nos puede suponer varias ventajas:

- Cada "trozo de programa" independiente será más fácil de programar, al realizar una función breve y concreta.
- El "programa principal" será más fácil de leer, porque no necesitará contener todos los detalles de cómo se hace cada cosa.
- Podremos repartir el trabajo, para que cada persona se encargue de realizar un "trozo de programa", y finalmente se integrará el trabajo individual de cada persona.

Esos "trozos" de programa son lo que suele llamar "subrutinas", "procedimientos" o "funciones". En el lenguaje C, el nombre que más se usa es el de **funciones**.

## *7.2. Conceptos básicos sobre funciones*

En C, un programa está formado por varios "trozos", que son funciones, incluyendo el propio cuerpo de programa, **main**. De hecho, la forma básica de **definir** una función será indicando su nombre seguido de unos paréntesis vacíos, como hacíamos con "main". Después, entre llaves indicaremos todos los pasos que queremos que dé ese "trozo de programa".

Por ejemplo, podríamos crear una función llamada "saludar", que escribiera varios mensajes en la pantalla:

```
saludar()
{
  printf("Bienvenido al programa\n");
  printf(" de ejemplo\n");
  printf("Bienvenido al programa\n");
}
```

Ahora desde dentro del cuerpo de nuestro programa, podríamos "**llamar**" a esa función:

```
int main()
{
  saludar();
  …
}
```

Así conseguimos que nuestro programa sea más fácil de leer. Como ejemplo, la parte principal de nuestra agenda podría ser simplemente:

```
leerDatosDeFichero();
do {
  mostrarMenu();
  pedirOpcion();
  switch( opcion ) {
    case 1: buscarDatos(); break;
    case 2: modificarDatos(); break;
    case 3: anadirDatos(); break;
    …
```

## *7.3. Parámetros de una función*

Es muy frecuente que nos interese además indicarle a nuestra función ciertos datos especiales con los que queremos que trabaje. Por ejemplo, si escribimos en pantalla números reales con frecuencia, nos puede resultar útil que nos los muestre con el formato que nos interese. Lo podríamos hacer así:

```
escribeNumeroReal( float n )
{
    printf("%4.2f", n);
}
```

Y esta función se podría usar desde el cuerpo de nuestro programa así:

```
int main()
{
  x= 5.1;
  printf("El primer numero real es: ");
  escribeNumeroReal(x);
  printf(" y otro distinto es: ");
  escribeNumeroReal(2.3);
}
```

Estos datos adicionales que indicamos a la función es lo que llamaremos sus "parámetros". Como se ve en el ejemplo, tenemos que indicar un nombre para cada parámetro (puede haber varios) y el tipo de datos que corresponde a ese parámetro. Si hay más de un parámetro, deberemos indicar el tipo y el nombre para cada uno de ellos:

```
sumar ( int x, int y )
{
  ...
}
```

Antes de probar todo esto, necesitamos saber un detalle más...

## 7.4. Valor devuelto por una función

También es habitual que queramos que nuestra función realice una serie de cálculos y nos "devuelva" el resultado de esos cálculos, para poderlo usar desde cualquier otra parte de nuestro programa. Por ejemplo, podríamos crear una función para elevar un número entero al cuadrado así:

```
/*--------------------------*/
/*  Ejemplo en C nº 63:     */
/*  c063.c                  */
/*                          */
/*  Función que devuelve un */
/*  valor y recibe un       */
/*  parámetro               */
/*                          */
/*  Curso de C,             */
/*    Nacho Cabanes         */
/*--------------------------*/

#include <stdio.h>

int cuadrado ( int n )
{
    return n*n;
}

int main()
{
    int numero;
    int resultado;

    numero= 5;
    resultado = cuadrado(numero);
    printf("El cuadrado del numero es %d", resultado);
    printf(" y el de 3 es %d", cuadrado(3));

    return 0;
}
```

Podemos hacer una función que nos diga cual es el mayor de dos números reales así:

```
float mayor ( float n1, float n2 )
{
    if (n1>n2)
        return n1;
    else
        return n2;
}
```

Esto tiene mucho que ver con el "return 0" que siempre estamos indicando al final de "main". Lo veremos en el siguiente apartado.

**Ejercicios propuestos:**

- (7.4.1) Crear una función que calcule el cubo de un número real (float). El resultado deberá ser otro número real. Probar esta función para calcular el cubo de 3.2 y el de 5.

- (7.4.2) Crear una función que calcule cual es el menor de dos números enteros. El resultado será otro número entero.
- (7.4.3) Crear una función llamada "signo", que reciba un número real, y devuelva un número entero con el valor: -1 si el número es negativo, 1 si es positivo o 0 si es cero.
- (7.4.4) Crear una función que devuelva la primera letra de una cadena de texto. Probar esta función para calcular la primera letra de la frase "Hola"
- (7.4.5) Crear una función que devuelva la última letra de una cadena de texto. Probar esta función para calcular la última letra de la frase "Hola".

## *7.5. El valor de retorno "void". El valor de retorno de "main"*

Cuando queremos dejar claro que una función no tiene que devolver ningún valor, podemos hacerlo indicando al principio que el tipo de datos va a ser "void" (nulo). Por ejemplo, nuestra función "saludar", que se limitaba a escribir varios textos en pantalla, quedaría más correcta si fuera así:

```
void saludar()
{
  printf("Bienvenido al programa\n");
  printf(" de ejemplo\n");
  printf("Bienvenido al programa\n");
}
```

Muchos compiladores antiguos (previos al estándar C99) permiten que una función se indique sin tipo devuelto, como hicimos en el apartado 1.1, y en ese caso, el lenguaje C no supondrá que no vaya a devolver ningún valor, sino que **devolverá un valor entero** (int):

```
main()
{
  ...
}
```

A partir del estándar C99, es obligatorio indicar el tipo devuelto, así que deberemos usar explícitamente "void", si la función no va a devolver ningún dato. Por eso, la forma habitual de declarar el cuerpo de un programa es:

```
int main()
{
  ...
}
```

Eso quiere decir que "main" también devuelve un valor, que se leerá desde fuera de nuestro programa. Lo normal es devolver 0 si todo ha funcionado correctamente

```
int main()
{
  ...
```

```
  return 0;
}
```

Y devolveríamos otro valor si hubiera habido algún problema durante el funcionamiento de nuestro programa (por ejemplo, si no hemos podido abrir algún fichero):

```
int main()
{
  FILE* fichero;
  fichero = fopen("nombre.txt", "rt");
  if (fichero == NULL) return 1;
  ...
  return 0;
}
```

Este valor se podría comprobar desde el sistema operativo. Por ejemplo, en MsDos y Windows se lee con "IF ERRORLEVEL", así:

```
IF ERRORLEVEL 1 ECHO Ha habido un error en el programa
```

Nota 1: En la mayoría de compiladores, el cuerpo del programa ("main") debe ser devolver un entero (int). Algunos compiladores permiten declarar "main" como "void", pero la mayoría no lo consideran aceptable.

Nota 2: En algunos lenguajes de programación se llama "procedimientos" (en inglés "procedure") o "subrutinas" a las funciones que no devuelven ningún valor (void), y se reserva el nombre "función" para las que sí dan un resultado.

**Ejercicios propuestos:**

- (7.5.1) Crear una función "escribirGuiones" que escriba en pantalla tantos guiones ("-") como se indique como parámetro y no devuelva ningún valor.
- (7.5.2) Crear una función "dibujarRectangulo", que reciba como parámetros la anchura y la altura del rectángulo a mostrar, y muestre en pantalla un rectángulo de ese tamaño, relleno de caracteres "#". Por ejemplo, para anchura 4 y altura 2 sería:

  ```
  ####
  ####
  ```

- (7.5.3) Crear una función "borrarPantalla" que borre la pantalla dibujando 25 líneas en blanco. No debe devolver ningún valor.
- (7.5.4) Crear una función "mostrarPerimSuperf" que reciba un número y muestre en pantalla el perímetro y la superficie de un cuadrado que tenga como lado el número que se ha indicado como parámetro.
- (7.5.5) Crear una función "escribirCentrado" que reciba un texto y lo escriba centrado en la siguiente línea de pantalla (suponiendo 80 columnas en pantalla). Por ejemplo, si el texto es "Hola", que tiene 4 letras, se escribirán 38 espacios antes de él.

## 7.6. Variables locales y variables globales

Hasta ahora, hemos declarado las variables antes de "main". Ahora nuestros programas tienen varios "bloques", así que se comportarán de forma distinta según donde declaremos las variables.

Las variables se pueden declarar dentro de un bloque (una función), y entonces sólo ese bloque las conocerá, no se podrán usar desde ningún otro bloque del programa. Es lo que llamaremos "variables **locales**".

Por el contrario, si declaramos una variable al comienzo del programa, fuera de todos los "bloques" de programa, será una "**variable global**", a la que se podrá acceder desde cualquier parte.

Vamos a verlo con un ejemplo. Crearemos una función que calcule la potencia de un número entero (un número elevado a otro), y el cuerpo del programa que la use.

La forma de conseguir elevar un número a otro será a base de multiplicaciones, es decir:

```
3 elevado a 5   =   3 · 3 · 3 · 3 · 3
```

(multiplicamos 5 veces el 3 por sí mismo). En general, como nos pueden pedir cosas como "6 elevado a 100" (o en general números que pueden ser grandes), usaremos la orden "for" para multiplicar tantas veces como haga falta:

```
/*---------------------------*/
/*  Ejemplo en C nº 63b:     */
/*  c063b.c                  */
/*                           */
/*  Ejemplo de función con   */
/*  variables locales        */
/*                           */
/*  Curso de C,              */
/*    Nacho Cabanes          */
/*---------------------------*/

#include <stdio.h>

int potencia(int base, int exponente)
{
    int temporal = 1;        /* Valor que voy hallando */
    int i;                  /* Para bucles */

    for(i=1; i<=exponente; i++)   /* Multiplico "n" veces */
        temporal *= base;        /* Y calculo el valor temporal */
    return temporal;             /* Tras las multiplicaciones, */
}                               /*    obtengo el valor que buscaba */

int main()
{
```

```
    int num1, num2;
    printf("Introduzca la base: ");
    scanf("%d", &num1);
    printf("Introduzca el exponente: ");
    scanf("%d", &num2);
    printf("%d elevado a %d vale %d", num1, num2, potencia(num1,num2));
    return 0;
}
```

En este caso, las variables "temporal" e "i" son locales a la función "potencia": para "main" no existen. Si en "main" intentáramos hacer  i=5;  obtendríamos un mensaje de error.

De igual modo, "num1" y "num2" son locales para "main": desde la función "potencia" no podemos acceder a su valor (ni para leerlo ni para modificarlo), sólo desde "main".

En general, **deberemos intentar que la mayor cantidad de variables posible sean locales** (lo ideal sería que todas lo fueran). Así hacemos que cada parte del programa trabaje con sus propios datos, y ayudamos a evitar que un error en un trozo de programa pueda afectar al resto. La forma correcta de pasar datos entre distintos trozos de programa es usando los parámetros de cada función, como en el anterior ejemplo.

**Ejercicios propuestos**:

- (7.6.1) Crear una función "pedirEntero", que reciba como parámetros el texto que se debe mostrar en pantalla, el valor mínimo aceptable y el valor máximo aceptable. Deberá pedir al usuario que introduzca el valor tantas veces como sea necesario, volvérselo a pedir en caso de error, y devolver un valor correcto. Probarlo con un programa que pida al usuario un año entre 1800 y 2100.
- (7.6.2) Crear una función "escribirTablaMultiplicar", que reciba como parámetro un número entero, y escriba la tabla de multiplicar de ese número (por ejemplo, para el 3 deberá llegar desde 3x0=0 hasta 3x10=30).
- (7.6.3) Crear una función "esPrimo", que reciba un número y devuelva el valor 1 si es un número primo o 0 en caso contrario.
- (7.6.4) Crear una función que reciba una cadena y una letra, y devuelva la cantidad de veces que dicha letra aparece en la cadena. Por ejemplo, si la cadena es "Barcelona" y la letra es 'a', debería devolver 2 (aparece 2 veces).
- (7.6.5) Crear una función que reciba un número cualquiera y que devuelva como resultado la suma de sus dígitos. Por ejemplo, si el número fuera 123 la suma sería 6.
- (7.6.6) Crear una función que reciba una letra y un número, y escriba un "triángulo" formado por esa letra, que tenga como anchura inicial la que se ha indicado. Por ejemplo, si la letra es * y la anchura es 4, debería escribir
  ****
  ***
  **
  *

## 7.7. Los conflictos de nombres en las variables

¿Qué ocurre si damos el mismo nombre a dos variables locales? Vamos a comprobarlo con un ejemplo:

```
/*--------------------------*/
/*  Ejemplo en C nº 64:     */
/*  c064.c                  */
/*                          */
/*  Dos variables locales   */
/*  con el mismo nombre     */
/*                          */
/*  Curso de C,             */
/*    Nacho Cabanes         */
/*--------------------------*/

#include <stdio.h>

void duplica(int n)
{
    n = n * 2;
}

int main()
{
    int n = 5;
    printf("n vale %d\n", n);
    duplica(n);
    printf("Ahora n vale %d\n", n);
    return 0;
}
```

El resultado de este programa es:

```
n vale 5
Ahora n vale 5
```

¿Por qué? Sencillo: tenemos una variable local dentro de "duplica" y otra dentro de "main". El hecho de que las dos tengan el mismo nombre no afecta al funcionamiento del programa, siguen siendo distintas. El programa se comporta como si "duplica" fuera así:

```
void duplica(int x)
{
   x = x * 2;
}
```

es decir, como si ambas tuvieran nombres distintos.

¿Y qué ocurre si una de ellas es una variable global? Retoquemos el ejemplo:

```
/*--------------------------*/
/*  Ejemplo en C nº 65:     */
```

```
/*  c065.c                  */
/*                          */
/*  Variables locales y     */
/*  globales con el mismo   */
/*  nombre                  */
/*                          */
/*  Curso de C,             */
/*    Nacho Cabanes         */
/*--------------------------*/

#include <stdio.h>

int n = 5;

void duplica(int n)
{
   n = n * 2;
}

int main()
{
   printf("n vale %d\n", n);
   duplica(n);
   printf("Ahora n vale %d\n", n);
   return 0;
}
```

El resultado será exactamente el mismo: la línea "`void duplica(int n)`" hace que dentro de "duplica", n se comporte como una variable local, por lo que los cambios que le hagamos no afectan a la variable global.

¿Y si queremos que se pueda modificar un dato indicado como parámetro? Todavía no sabemos cómo hacerlo (lo veremos en el próximo tema). Por ahora sólo sabemos hacerlo devolviendo el nuevo valor con "return", con lo que nuestro último ejemplo quedaría así:

```
/*--------------------------*/
/*  Ejemplo en C nº 66:     */
/*  c066.c                  */
/*                          */
/*  Modificar la variable   */
/*  indicada como parámetro: */
/*  devolviendo su valor    */
/*                          */
/*  Curso de C,             */
/*    Nacho Cabanes         */
/*--------------------------*/

#include <stdio.h>

int duplica(int n)
{
   return n * 2;
}

int main()
{
   int n = 5;
```

```
   printf("n vale %d\n", n);
   n = duplica(n);
   printf("Ahora n vale %d\n", n);
   return 0;
}
```

## *7.8. El orden importa*

En general, una función debe estar declarada antes de usarse. Por ejemplo, este fuente daría un error en muchos compiladores, porque dentro de "main" intentamos usar algo llamado "duplica", que no se ha mencionado antes:

```
/*--------------------------*/
/*  Ejemplo en C nº 67:     */
/*  c067.c                  */
/*                          */
/*  Función desordenada:    */
/*  puede no compilar       */
/*                          */
/*  Curso de C,             */
/*    Nacho Cabanes         */
/*--------------------------*/

#include <stdio.h>

int main()
{
   float n = 5;
   printf("n vale %f\n", n);
   n = duplica(n);
   printf("Ahora n vale %f\n", n);
   return 0;
}

float duplica(float n)
{
   return n * 2;
}
```

La forma de evitarlo es colocar la definición de las funciones antes de usarlas (si se puede) o bien incluir al menos su "**prototipo**", la cabecera de la función sin incluir los detalles de cómo trabaja internamente, así:

```
/*--------------------------*/
/*  Ejemplo en C nº 68:     */
/*  c068.c                  */
/*                          */
/*  Prototipo de la función */
/*  antes de main para que  */
/*  compile sin problemas   */
/*                          */
/*  Curso de C,             */
/*    Nacho Cabanes         */
/*--------------------------*/
```

```
#include <stdio.h>

float duplica(float n) ;

int main()
{
   float n = 5;
   printf("n vale %f\n", n);
   n = duplica(n);
   printf("Ahora n vale %f\n", n);
   return 0;
}

float duplica(float n)
{
   return n * 2;
}
```

Como curiosidad, si no declaramos la función ni su prototipo antes de "main", más de un compilador dará por sentado que es "int", de modo que este otro fuente sí compilaría correctamente en algunos sistemas:

```
/*---------------------------*/
/*  Ejemplo en C nº 68b:     */
/*  c068b.c                  */
/*                           */
/*  Función sin prototipo,   */
/*  puede compilar en        */
/*  algunos entornos         */
/*                           */
/*  Curso de C,              */
/*    Nacho Cabanes          */
/*---------------------------*/

#include <stdio.h>

int main()
{
   int n = 5;
   printf("n vale %d\n", n);
   n = duplica(n);
   printf("Ahora n vale %d\n", n);
   return 0;
}

int duplica(int n)
{
   return n * 2;
}
```

## *7.9. Algunas funciones útiles*

### 7.9.1. Números aleatorios

En un programa de gestión o una utilidad que nos ayuda a administrar un sistema, no es habitual que podamos permitir que las cosas ocurran al azar. Pero los juegos se encuentran muchas

veces entre los ejercicios de programación más completos, y para un juego sí suele ser conveniente que haya algo de azar, para que una partida no sea exactamente igual a la anterior.

Generar **números al azar** ("números aleatorios") usando C no es difícil. Si nos ceñimos al estándar ANSI C, tenemos una función llamada "rand()", que nos devuelve un número entero entre 0 y el valor más alto que pueda tener un número entero en nuestro sistema. Generalmente, nos interesarán números mucho más pequeños (por ejemplo, del 1 al 100), por lo que "recortaremos" usando la operación módulo ("%", el resto de la división).

Vamos a verlo con algún ejemplo:

Para obtener un número del 0 al 9 haríamos `x = rand() % 10;`
Para obtener un número del 0 al 29 haríamos `x = rand() % 30;`
Para obtener un número del 10 al 29 haríamos `x = rand() % 20 + 10;`
Para obtener un número del 1 al 100 haríamos `x = rand() % 100 + 1;`
Para obtener un número del 50 al 60 haríamos `x = rand() % 11 + 50;`
Para obtener un número del 101 al 199 haríamos `x = rand() % 100 + 101;`

Pero todavía nos queda un detalle para que los números aleatorios que obtengamos sean "razonables": los números que genera un ordenador no son realmente al azar, sino "pseudo-aleatorios", cada uno calculado a partir del siguiente. Podemos elegir cual queremos que sea el primer número de esa serie (la "semilla"), pero si usamos uno prefijado, los números que se generarán serán siempre los mismos. Por eso, será conveniente que el primer número se base en el reloj interno del ordenador: como es casi imposible que el programa se ponga en marcha dos días exactamente a la misma hora (incluyendo milésimas de segundo), la serie de números al azar que obtengamos será distinta cada vez.

La "semilla" la indicamos con "srand", y si queremos basarnos en el reloj interno del ordenador, lo que haremos será `srand(time(0));` antes de hacer ninguna llamada a "rand()".

Para usar "rand()" y "srand()", deberíamos añadir otro fichero a nuestra lista de "includes", el llamado "stdlib":

```
#include <stdlib.h>
```

Si además queremos que la semilla se tome a partir del reloj interno del ordenador (que es lo más razonable), deberemos incluir también "time":

```
#include <time.h>
```

Vamos a ver un ejemplo, que muestre en pantalla un número al azar entre 1 y 10:

```
/*--------------------------*/
/*  Ejemplo en C nº 69:     */
/*  c069.c                  */
/*                          */
/*  Obtener un número al    */
/*  azar                    */
```

```
/*                          */
/*  Curso de C,             */
/*    Nacho Cabanes         */
/*--------------------------*/

#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int main()
{
   int n;
   srand(time(0));
   n = rand() % 10 + 1;
   printf("Un número entre 1 y 10: %d\n", n);
   return 0;
}
```

**Ejercicios propuestos:**

- (7.9.1.1) Crea un programa que escriba varias veces "Hola" (entre 5 y 10 veces, al azar).
- (7.9.1.2) Crea un programa que genere un número al azar entre 1 y 100. El usuario tendrá 6 oportunidades para acertarlo.
- (7.9.1.3) Crea un programa que muestre un "fondo estrellado" en pantalla: mostrará 24 líneas, cada una de las cuales contendrá entre 1 y 78 espacios (al azar) seguidos por un asterisco ("*").

### 7.9.2. Funciones matemáticas

Dentro del fichero de cabecera "**math.h**" tenemos acceso a muchas funciones matemáticas predefinidas en C, como:

- acos(x): Arco coseno
- asin(x): Arco seno
- atan(x): Arco tangente
- atan2(y,x): Arco tangente de y/x (por si x o y son 0)
- ceil(x): El valor entero superior a x y más cercano a él
- cos(x): Coseno
- cosh(x): Coseno hiperbólico
- exp(x): Exponencial de x (e elevado a x)
- fabs(x): Valor absoluto
- floor(x): El mayor valor entero que es menor que x
- fmod(x,y): Resto de la división x/y
- log(x): Logaritmo natural (o neperiano, en base "e")
- log10(x): Logaritmo en base 10
- pow(x,y): x elevado a y
- sin(x): Seno
- sinh(x): Seno hiperbólico
- sqrt(x): Raíz cuadrada

- tan(x): Tangente
- tanh(x): Tangente hiperbólica

(todos ellos usan parámetros X e Y de tipo "double")

y una serie de constantes como

M_E, el número "e", con un valor de 2.71828...
M_PI, el número "Pi", 3.14159...

La mayoría de ellas son específicas para ciertos problemas matemáticos, especialmente si interviene la trigonometría o si hay que usar logaritmos o exponenciales. Pero vamos a destacar las que sí pueden resultar útiles en situaciones más variadas:

La raiz cuadrada de 4 se calcularía haciendo `x = sqrt(4);`
La potencia: para elevar 2 al cubo haríamos `y = pow(2, 3);`
El valor absoluto: si queremos trabajar sólo con números positivos usaríamos `n = fabs(x);`

Un detalle importante: en ciertos compiladores de C, especialmente en plataforma Linux, es frecuente que, al compilar un fuente que usa funciones matemáticas, tengamos indicar expresamente que queremos **enlazar la biblioteca "math"**, o de lo contrario obtendremos mensajes de error similares a "undefined reference to 'sqrt'". La forma correcta de compilar será añadiendo "-lm" a la línea de compilación:

```
gcc fuente.c -o ejecutable -lm
```

**Ejercicios propuestos:**

- (7.9.2.1) Crear un programa que halle cualquier raíz de un número. El usuario deberá indicar el número (por ejemplo, 2) y el índice de la raiz (por ejemplo, 3 para la raíz cúbica). Pista: hallar la raíz cúbica de 2 es lo mismo que elevar 2 a 1/3.
- (7.9.2.2) Crear un programa que resuelva ecuaciones de segundo grado, del tipo $ax^2 + bx + c = 0$ El usuario deberá introducir los valores de a, b y c. Pista: la solución se calcula con $x = \pm$ raíz $(b^2 – 4 \cdot a \cdot c) / 2 \cdot a$
- (7.9.2.3) Crear un programa que muestre el seno de los ángulos de 30 grados, 45 grados, 60 grados y 90 grados. Cuidado: la función "sin" espera que se le indique el ángulo en radianes, no en grados. Tendrás que recordar que 180 grados es lo mismo que Pi radianes (con Pi = 3,1415926535). Puedes crearte una función auxiliar que convierta de grados a radianes.

### 7.9.3. Pero casi todo son funciones...

Pero en C hay muchas más funciones de lo que parece. De hecho, casi todo lo que hasta ahora hemos llamado "órdenes", son realmente "funciones", y la mayoría de ellas incluso devuelven algún valor, que hasta ahora habíamos despreciado en muchos casos.

Vamos a hacer un repaso rápido a las funciones que ya conocemos y el valor que devuelven:

| Función | Valor devuelto | Significado |
|---|---|---|
| main | int | Programa terminado correctamente (0) o no (otro) |
| printf | int | Número de caracteres escritos |
| scanf | int | Número de datos leídos, o EOF si hay error |
| putchar | int | El carácter escrito, o EOF si hay algún error |
| getchar | int | Siguiente carácter de la entrada, o EOF en caso de error |
| gets | char* | Cadena si todo va bien o NULL si hay error |
| puts | int | EOF si hay algún error, otro número (un entero positivo) si no lo hay |
| strcpy, strncpy | char* | Cadena resultado de la asignación |
| strcat | char* | Cadena resultado |
| strcmp | int | 0 si las cadenas son iguales, <0 si la primera "es menor" o >0 si la primera "es mayor" |
| sprintf | int | Número de caracteres almacenados en la cadena (en alguna versión, como BSD, devuelve la cadena creada) |
| sscanf | int | Número de datos leídos, o EOF si hay error |
| fopen | FILE* | NULL si hay error |
| fclose | int | 0 si todo va bien o EOF si hay error |
| fputs | int | EOF si hay algún error, otro número (no especificado) si no lo hay |
| fgets | char* | NULL si hay error o fin de fichero |
| feof | int | 0 si no es final de fichero, otro valor si lo es |
| fprintf | int | Número de bytes escritos (puede no ser fiable si se está escribiendo a un buffer antes de mandar a disco). |
| fscanf | int | Número de datos leídos, o EOF si hay error |
| fgetc | int | El carácter leido, o EOF si hay algún error |
| fputc | int | El carácter escrito, o EOF si hay algún error |
| fread | int | Número de bytes leidos (0 o menor de lo previsto si hay error) |
| fwrite | int | Número de bytes escritos (0 o menor de lo previsto si hay error) |
| fseek | int | 0 si se ha saltado correctamente; otro valor si el fichero no está abierto o no se ha podido saltar |
| ftell | long (size_t) | Posición actual en el fichero (en bytes) o -1L en caso de error |

(Nota: expresiones como "FILE*" ya las conocemos, aunque todavía no entendemos bien a qué se refieren: otras como "char*" son nuevas para nosotros, pero las veremos con detalle en el próximo tema).

Por el contrario, las siguientes "órdenes" no son funciones, sino "palabras reservadas" del lenguaje C: if, else, do, while, for, switch, case, default, break, int, char, float, double, struct.

**Ejercicios propuestos:**

- (7.9.3.1) Crea un programa que escriba la raíz cuadrada de 10 y muestre la cantidad de cifras que se han escrito en pantalla. (Pista: mira el valor devuelto por "printf").

## *7.10. Recursividad*

Una función recursiva es aquella que se define a partir de ella misma. Dentro de las matemáticas tenemos varios ejemplos. Uno clásico es el "factorial de un número":

$$n! = n \cdot (n-1) \cdot (n-2) \cdot ... \cdot 3 \cdot 2 \cdot 1$$

(por ejemplo, el factorial de 4 es $4 \cdot 3 \cdot 2 \cdot 1 = 24$)

Si pensamos que el factorial de n-1 es

$$(n-1)! = (n-1) \cdot (n-2) \cdot (n-3) \cdot ... \cdot 3 \cdot 2 \cdot 1$$

Entonces podemos escribir el factorial de un número a partir del factorial del siguiente número:

$$n! = n \cdot (n-1)!$$

Esta es la definición recursiva del factorial, ni más ni menos. Esto, programando, se haría:

```
/*---------------------------*/
/*  Ejemplo en C nº 70:      */
/*  c070.c                   */
/*                           */
/*  Funciones recursivas:    */
/*  factorial                */
/*                           */
/*  Curso de C,              */
/*    Nacho Cabanes          */
/*---------------------------*/

#include <stdio.h>

long fact(int n)
{
  if (n==1)               /* Aseguramos que termine */
    return 1;
  return n * fact (n-1);   /* Si no es 1, sigue la recursión */
}

int main()
```

```
{
  int num;
  printf("Introduzca un número entero: ");
  scanf("%d", &num);
  printf("Su factorial es: %ld\n", fact(num));
  return 0;
}
```

Dos consideraciones importantes:

- Atención a la primera parte de la función recursiva: es MUY IMPORTANTE **comprobar que hay salida** de la función, para que nuestro programa no se quede dando vueltas todo el tiempo y deje el ordenador (o la tarea actual) "colgado".
- Los factoriales **crecen rápidamente**, así que no conviene poner números grandes: el factorial de 16 es 2.004.189.184, luego a partir de 17 podemos obtener resultados erróneos, según sea el tamaño de los números enteros en nuestro sistema.

¿Qué utilidad tiene esto? Pues más de la que parece: muchos problemas complicados se pueden expresar a partir de otro más sencillo. En muchos de esos casos, ese problema se podrá expresar de forma recursiva. Más adelante veremos algún otro ejemplo.

**Ejercicios propuestos:**

- (7.10.1) Crear una función que calcule el valor de elevar un número entero a otro número entero (por ejemplo, 5 elevado a 3 = $5^3$ = 5 ·5 · 5 = 125). Esta función se debe crear de forma recursiva.
- (7.10.2) Como alternativa, crear una función que calcule el valor de elevar un número entero a otro número entero de forma NO recursiva (lo que llamaremos "de forma iterativa"), usando la orden "for".
- (7.10.3) Crear un programa que emplee recursividad para calcular un número de la serie Fibonacci (en la que los dos primeros elementos valen 1, y para los restantes, cada elemento es la suma de los dos anteriores).
- (7.10.4) Crear un programa que emplee recursividad para calcular la suma de los elementos de un vector.
- (7.10.5) Crear un programa que emplee recursividad para calcular el mayor de los elementos de un vector.
- (7.10.6) Crear un programa que emplee recursividad para dar la vuelta a una cadena de caracteres (por ejemplo, a partir de "Hola" devolvería "aloH").
- (7.10.7) Crear, tanto de forma recursiva como de forma iterativa, una función diga si una cadena de caracteres es simétrica (un palíndromo). Por ejemplo, "DABALEARROZALAZORRAELABAD" es un palíndromo.
- (7.10.8) Crear un programa que encuentre el máximo común divisor de dos números usando el algoritmo de Euclides: Dados dos números enteros positivos m y n, tal que m > n, para encontrar su máximo común divisor, es decir, el mayor entero positivo que divide a ambos: - Dividir m por n para obtener el resto r ($0 \leq r < n$) ; - Si r = 0, el MCD es n.; - Si no, el máximo común divisor es MCD(n,r).

## *7.11. Cómo interrumpir el programa.*

Si queremos que nuestro programa se interrumpa en un cierto punto, podemos usar la orden "exit", que ya habíamos utilizado en el apartado 6.2, cuando vimos si podíamos abrir un fichero sin errores.

Su manejo habitual es algo como

```
exit(1);
```

Es decir, entre paréntesis indicamos un cierto código, que suele ser (por convenio) un 0 si no ha habido ningún error, u otro código distinto en caso de que sí exista algún error, al igual que vimos con el valor de retorno de "main" (y el uso de este código es el que vimos entonces).

Si queremos utilizar "exit", deberíamos añadir otro fichero a nuestra lista de "includes", el llamado "stdlib" (igual que hicimos al manejar números al azar):

```
#include <stdlib.h>
```

(Aun así, al igual que ocurría con "stdio.h", algunos compiladores permitirán que nuestro programa funcione correctamente aunque no tenga esta línea).

La diferencia entre "exit" y "return" es que la primera de estas órdenes abandona el programa, se encuentre en el punto en que se encuentre, mientras que la segunda sale de la función actual (pero no se interrumpirá al programa a no ser que ésta sea "main").

# 8. Cómo depurar los programas

## *8.1. Conceptos básicos sobre depuración*

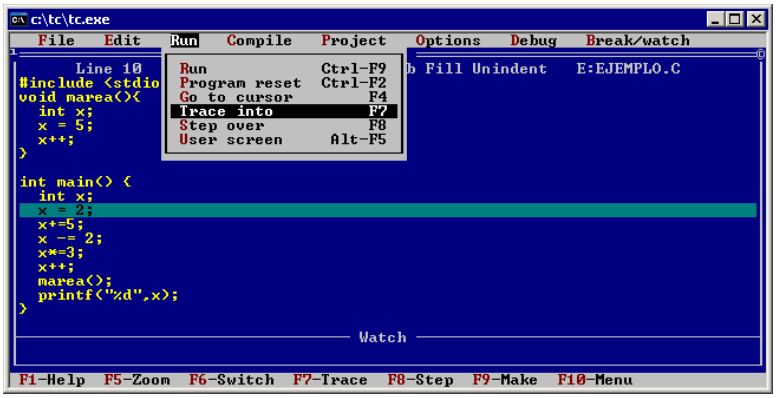
La depuración es el análisis de un programa para descubrir fallos. El nombre en inglés es "debug", porque esos fallos de programación reciben el nombre de "bugs" (bichos).

Para eliminar esos fallos que hacen que un programa no se comporte como debería, se usan unas herramientas llamadas "depuradores". Estos nos permiten avanzar paso a paso para ver cómo avanza realmente nuestro programa, y también nos dejan ver los valores de las variables.

Como nuestros conocimientos ya nos permiten hacer programas de una cierta complejidad, es el momento de ver formas de descubrir dónde están los posibles errores. Lo haremos desde varios entornos distintos.

## *8.2. Ejemplos de algunos entornos*

**Turbo C** es un compilador antiguo, pero sencillo de manejar, y la depuración también es sencilla con él:

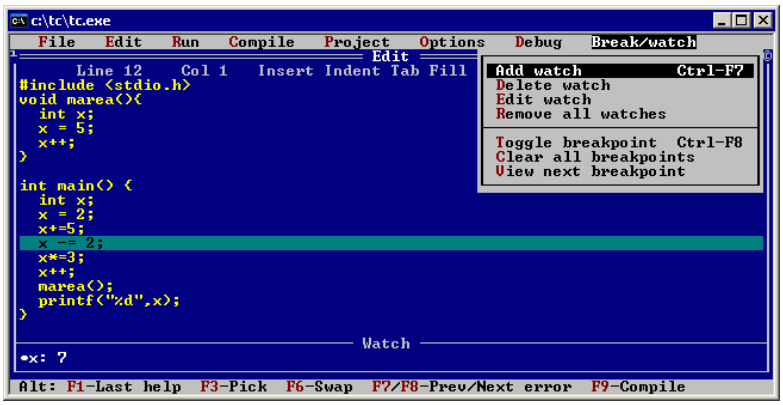



El menú "Run" es el que nos permite poner nuestro programa en marcha normalmente ("Run"), pero también el que nos permite avanzar paso a paso por las órdenes que lo forman.

Hay dos maneras de hacerlo:

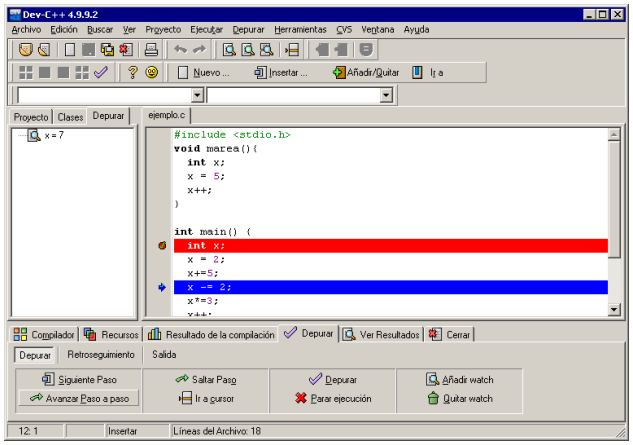
- "Trace into" va paso a paso por todas las órdenes del programa. Si hay una llamada a una función, también sigue paso a paso por las órdenes que forma esa función.
- "Step over" es similar, salvo que cuando haya una llamada a una función, ésta se tomará como una única orden, sin ver los detalles de dicha función.

En cualquiera de ambos casos, se nos muestra con una línea azul por dónde va avanzando la depuración.




Si queremos observar cómo evoluciona el valor de alguna variable, podemos añadir un "vigía" (en inglés "watch") desde el menú "Break/Watch. Este vigía quedará en la parte inferior de la pantalla.

Si trabajamos con **DevC++ para Windows**, la situación no es muy diferente. Primero debemos indicar dónde queremos que se interrumpa el programa (añadir un "breakpoint"), haciendo clic con el ratón en el margen izquierdo de nuestro programa. Esa línea quedará resaltada en color rojo.

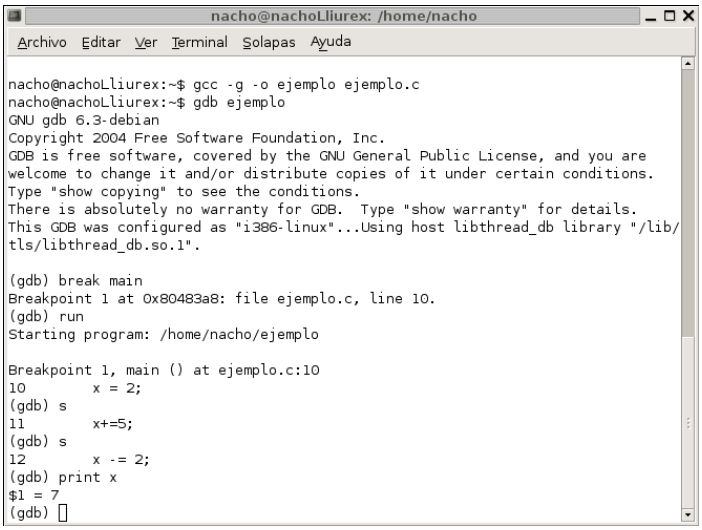



Ahora ponemos en marcha el programa desde el menú "Depurar".

En la parte inferior de la pantalla aparecerán botones que nos permiten avanzar paso a paso, saltar las funciones o añadir un "watch" para comprobar el valor de una variables. La línea que estamos depurando se marcará en azul (y destacada con una flecha). Los "watches" estarán visibles en la parte izquierda de la pantalla.

Bajo **Linux**, el entorno estándar (pero incómodo) es el de la herramienta "**gdb**". Para usarla primero debemos compilar nuestro fuente como siempre, pero añadiendo la opción "-g", que se encarga de incluir en nuestro ejecutable toda la información adicional que el depurador necesita para poder mostrarnos cual es la línea que estamos procesando:

```
gcc –g ejemplo.c –o ejemplo
```




Después ponemos en marcha el depurador, indicándole el nombre del programa a depurar:

```
gdb ejemplo
```

Deberemos indicar un punto de interrupción, con la orden "break". La forma más sencilla de usarla es indicando una función en la que queramos que se interrumpa nuestro programa. Si nuestro programa no tiene funciones auxiliares o bien si queremos depurar desde el principio, usaríamos la función "main", así:

```
break main
```

Podemos en marcha el programa con la orden "run":

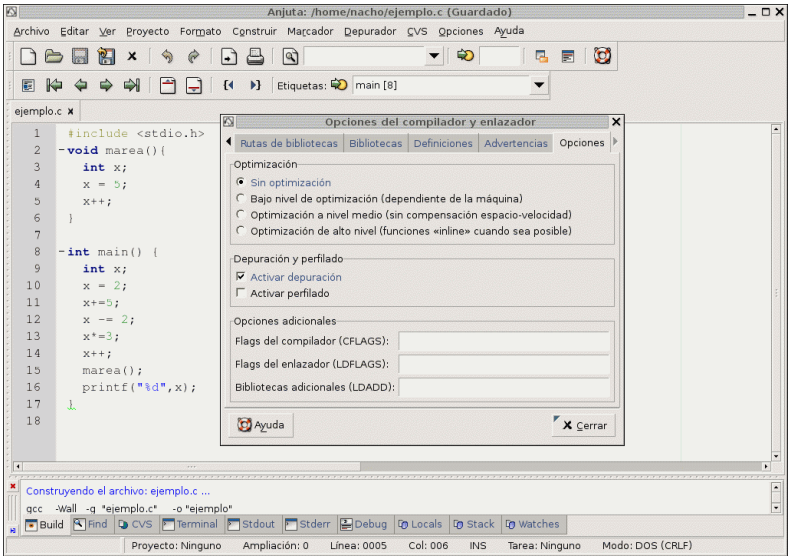
```
run
```

Se parará al llegar al punto de interrupción, y a partir de ahí podemos usar la orden "step" (que se puede abreviar con "s") para avanzar paso a paso, o la orden "print" si queremos ver el valor de alguna variable:

```
print x
```

Si usamos otra herramienta de desarrollo, como **Anjuta**, la depuración puede ser mucho más sencilla, similar al caso de Turbo C y al de DevC++.

En este caso, podemos depurar fácilmente desde el propio entorno de desarrollo. Aun así, el primer paso es el mismo que si fueramos a usar gdb: debemos compilar con la opción –g. La forma más sencilla de hacerlo es indicar que queremos "activar la depuración", desde el menú "Opciones".

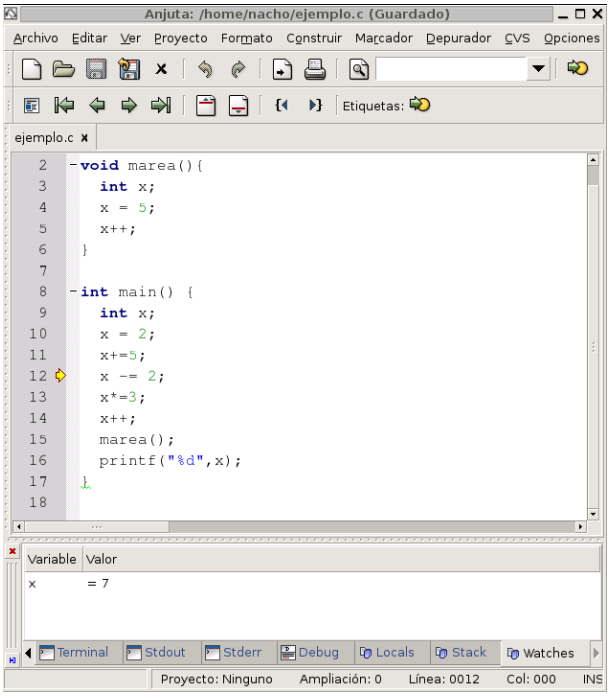



Entonces ya podemos construir nuestro ejecutable normalmente, e iniciar la depuración desde el menú "Depurador".

Podremos avanzar paso a paso (desde el propio menú o pulsando la tecla F5), y añadir "watches" para observar cómo evoluciona el valor de las variables.

Estos "watches" no estarán visibles hasta que no escojamos la pestaña correspondiente en la parte inferior de la pantalla.

La línea que estamos depurando se indicará mediante una flecha en su margen izquierdo.



# 9. Punteros y gestión dinámica de memoria

## *9.1. ¿Por qué usar estructuras dinámicas?*

Hasta ahora teníamos una serie de variables que declaramos al principio del programa o de cada función. Estas variables, que reciben el nombre de **estáticas**, tienen un tamaño asignado desde el momento en que se crea el programa.

Este tipo de variables son sencillas de usar y rápidas... si sólo vamos a manejar estructuras de datos que no cambien, pero resultan poco eficientes si tenemos estructuras cuyo tamaño no sea siempre el mismo.

Es el caso de una agenda: tenemos una serie de fichas, e iremos añadiendo más. Si reservamos espacio para 10, no podremos llegar a añadir la número 11, estamos limitando el máximo. Una solución sería la de trabajar siempre en el disco: no tenemos límite en cuanto a número de fichas, pero es muchísimo más lento.

Lo ideal sería aprovechar mejor la memoria que tenemos en el ordenador, para guardar en ella todas las fichas o al menos todas aquellas que quepan en memoria.

Una solución "típica" (pero mala) es sobredimensionar: preparar una agenda contando con 1000 fichas, aunque supongamos que no vamos a pasar de 200. Esto tiene varios inconvenientes: se desperdicia memoria, obliga a conocer bien los datos con los que vamos a trabajar, sigue pudiendo verse sobrepasado, etc.

La solución suele ser crear estructuras **dinámicas**, que puedan ir creciendo o disminuyendo según nos interesen. Ejemplos de este tipo de estructuras son:

- Las **pilas**. Como una pila de libros: vamos apilando cosas en la cima, o cogiendo de la cima.
- Las **colas**. Como las del cine (en teoría): la gente llega por un sitio (la cola) y sale por el opuesto (la cabeza).
- Las **listas**, en las que se puede añadir elementos, consultarlos o borrarlos en cualquier posición.

Y la cosa se va complicando: en los **árboles** cada elemento puede tener varios sucesores, etc. Todas estas estructuras tienen en común que, si se programan bien, pueden ir creciendo o decreciendo según haga falta, al contrario que un array, que tiene su tamaño prefijado.

En todas ellas, lo que vamos haciendo es reservar un poco de memoria para cada **nuevo elemento** que nos haga falta, y enlazarlo a los que ya teníamos. Cuando queramos borrar un elemento, enlazamos el anterior a él con el posterior a él (para que no "se rompa" nuestra estructura) y liberamos la memoria que estaba ocupando.

## 9.2. ¿Qué son los punteros?

Un puntero no es más que una **dirección de memoria**. Lo que tiene de especial es que normalmente un puntero tendrá un tipo de datos asociado: por ejemplo, un "puntero a entero" será una dirección de memoria en la que habrá almacenado (o podremos almacenar) un número entero.

Vamos a ver qué **símbolo** usamos en C para designar los punteros:

```
    int num;            /* "num" es un número entero */
    int *pos;           /* "pos" es un "puntero a entero" (dirección de
                           memoria en la que podremos guardar un entero) */
```

Es decir, pondremos un asterisco entre el tipo de datos y el nombre de la variable. Ese asterisco puede ir junto a cualquiera de ambos, también es correcto escribir

```
int* pos;
```

Esta nomenclatura ya la habíamos utilizado aun sin saber que era eso de los punteros. Por ejemplo, cuando queremos acceder a un fichero, hacemos

```
FILE* fichero;
```

Antes de entrar en más detalles, y para ver la diferencia entre trabajar con "arrays" o con punteros, vamos a hacer dos programas que pidan varios números enteros al usuario y muestren su suma. El primero empleará un "array" (una tabla, de tamaño predefinido) y el segundo empleará memoria que reservaremos durante el funcionamiento del programa.

El primero podría ser así:

```
/*---------------------------*/
/*  Ejemplo en C nº 71:      */
/*  c071.c                   */
/*                           */
/*  Sumar varios datos       */
/*  Version 1: con arrays    */
/*                           */
/*  Curso de C,              */
/*    Nacho Cabanes          */
/*---------------------------*/

#include <stdio.h>

int main()
{
    int datos[100];  /* Preparamos espacio para 100 numeros */
    int cuantos;     /* Preguntaremos cuantos desea introducir */
    int i;           /* Para bucles */
    long suma=0;     /* La suma, claro */

    do
```

```
    {
        printf("Cuantos numeros desea sumar? ");
        scanf("%d", &cuantos);
        if (cuantos>100)  /* Solo puede ser 100 o menos */
        printf("Demasiados. Solo se puede hasta 100.");
    }
    while (cuantos>100);  /* Si pide demasiado, no le dejamos */

    /* Pedimos y almacenamos los datos */
    for (i=0; i<cuantos; i++)
    {
        printf("Introduzca el dato número %d: ", i+1);
        scanf("%d", &datos[i]);
    }

    /* Calculamos la suma */
    for (i=0; i<cuantos; i++)
        suma += datos[i];

    printf("Su suma es: %ld\n", suma);
    return 0;
}
```

Los más avispados se pueden dar cuenta de que si sólo quiero calcular la suma, lo podría hacer a medida que leo cada dato, no necesitaría almacenar todos. Vamos a suponer que sí necesitamos guardarlos (en muchos casos será verdad, si los cálculos son más complicados). Entonces nos damos cuenta de que lo que hemos estado haciendo hasta ahora **no es eficiente**:

- Si quiero sumar 1000 datos, o 500, o 101, no puedo. Nuestro límite previsto era de 100, así que no podemos trabajar con más datos.
- Si sólo quiero sumar 3 números, desperdicio el espacio de 97 datos que no uso.
- Y el problema sigue: si en vez de 100 números, reservamos espacio para 5000, es más difícil que nos quedemos cortos pero desperdiciamos muchísima más memoria.

La solución es reservar espacio estrictamente para lo que necesitemos, y eso es algo que podríamos hacer así:

```
/*---------------------------*/
/*  Ejemplo en C nº 72:      */
/*  c072.c                   */
/*                           */
/*  Sumar varios datos       */
/*  Version 2: con punteros  */
/*                           */
/*  Curso de C,              */
/*    Nacho Cabanes          */
/*---------------------------*/

#include <stdio.h>
#include <stdlib.h>

int main()
{
    int* datos;      /* Necesitaremos espacio para varios numeros */
```

```
    int cuantos;     /* Preguntaremos cuantos desea introducir */
    int i;           /* Para bucles */
    long suma=0;     /* La suma, claro */

    do
    {
        printf("Cuantos numeros desea sumar? ");
        scanf("%d", &cuantos);
        datos = (int *) malloc (cuantos * sizeof(int));
        if (datos == NULL)  /* Si no hay espacio, avisamos */
        printf("No caben tantos datos en memoria.");
    }
    while (datos == NULL);  /* Si pide demasiado, no le dejamos */

    /* Pedimos y almacenamos los datos */
    for (i=0; i<cuantos; i++)
    {
        printf("Introduzca el dato número %d: ", i+1);
        scanf("%d", datos+i);
    }

    /* Calculamos la suma */
    for (i=0; i<cuantos; i++)
        suma += *(datos+i);

    printf("Su suma es: %ld\n", suma);
    free(datos);
    return 0;
}
```

Este fuente es más difícil de leer, pero a cambio es mucho más eficiente: funciona perfectamente si sólo queremos sumar 5 números, pero también si necesitamos sumar 120.000 (y si caben tantos números en la memoria disponible de nuestro equipo, claro).

Vamos a ver las diferencias:

En primer lugar, lo que antes era `int datos[100]` que quiere decir "a partir de la posición de memoria que llamaré *datos*, querré espacio para a guardar 100 números enteros", se ha convertido en `int* datos` que quiere decir "a partir de la posición de memoria que llamaré *datos*, voy a guardar varios números enteros (pero aún no sé cuantos)".

Luego reservamos el espacio exacto que necesitamos, haciendo `datos = (int *) malloc (cuantos * sizeof(int));` Esta orden suena complicada, así que vamos a verla por partes:

- "malloc" es la orden que usaremos para reservar memoria cuando la necesitemos (es la abreviatura de las palabra "memory" y "allocate").
- Como parámetro, le indicamos cuanto espacio queremos reservar. Para 100 números enteros, sería "100*sizeof(int)", es decir, 100 veces el tamaño de un entero. En nuestro caso, no son 100 números, sino el valor de la variable "cuantos". Por eso hacemos "malloc (cuantos*sizeof(int))".
- Para terminar, ese es el espacio que queremos reservar para nuestra variable "datos". Y esa variable es de tipo "int *" (un puntero a datos que serán números enteros). Para que

todo vaya bien, debemos "convertir" el resultado de "malloc" al tipo de datos correcto, y lo hacemos forzando una conversión como vimos en el apartado 2.4 (operador "molde"), con lo que nuestra orden está completa:
*datos = (int *) malloc (cuantos * sizeof(int));*

- Si "malloc" nos devuelve NULL como resultado (un "puntero nulo"), quiere decir que no ha encontrado ninguna posición de memoria en la que nos pudiera reservar todo el espacio que le habíamos solicitado.
- Para usar "malloc" deberemos incluir "stdlib.h" al principio de nuestro fuente.

La forma de guardar los datos que teclea el usuario también es distinta. Cuando trabajábamos con un "array", hacíamos `scanf("%d", &datos[i])` ("el dato número i"), pero con punteros usaremos `scanf("%d", datos+i)` (en la posición datos + i). Ahora ya no necesitamos el símbolo **"ampersand" (&)**. Este símbolo se usa para indicarle a C en qué posición de memoria debe almacenar un dato. Por ejemplo, `float x;` es una variable que podremos usar para guardar un número real. Si lo hacemos con la orden "scanf", esta orden no espera que le digamos en qué variable deber guardar el dato, sino en qué posición de memoria. Por eso hacemos `scanf("%f", &x);` En el caso que nos encontramos ahora, `int* datos` ya se refiere a una posición de memoria (un puntero), por lo que no necesitamos & para usar "scanf".

Finalmente, la forma de acceder a los datos también cambia. Antes leíamos el primer dato como datos[0], el segundo como datos[1], el tercero como datos[2] y así sucesivamente. Ahora usaremos el asterisco (*) para indicar que queremos saber el valor que hay almacenado en una cierta posición: el primer dato será *datos, el segundo *(datos+1), el tercero será *(datos+2) y así en adelante. Por eso, donde antes hacíamos `suma += datos[i];` ahora usamos `suma += *(datos+i);`

También aparece otra orden nueva: **free**. Hasta ahora, teníamos la memoria reservada estáticamente, lo que supone que la usábamos (o la desperdiciábamos) durante todo el tiempo que nuestro programa estuviera funcionando. Pero ahora, igual que reservamos memoria justo en el momento en que la necesitamos, y justo en la cantidad que necesitamos, también podemos volver a dejar disponible esa memoria cuando hayamos terminado de usarla. De eso se encarga la orden "free", a la que le debemos indicar qué puntero es el que queremos liberar.

## *9.3. Repasemos con un ejemplo sencillo*

El ejemplo anterior era "un caso real". Generalmente, los casos reales son más aplicables que los ejemplos puramente académicos, pero también más difíciles de seguir. Por eso, antes de seguir vamos a ver un ejemplo más sencillo que nos ayude a asentar los conceptos: Reservaremos espacio para un número real de forma estática, y para dos números reales de forma dinámica, daremos valor a dos de ellos, guardaremos su suma en el tercer número y mostraremos en pantalla los resultados.

```
/*--------------------------*/
/*  Ejemplo en C nº 73:     */
/*  c073.c                  */
/*                          */
```

```
/*  Manejo básico de         */
/*  punteros                 */
/*                           */
/*  Curso de C,              */
/*    Nacho Cabanes          */
/*--------------------------*/

#include <stdio.h>
#include <stdlib.h>

int main()
{
    float n1;           /* Primer número, estático */
    float *n2, *suma;    /* Los otros dos números */

    n1 = 5.0;           /* Damos un valor prefijado a n1 (real) */
    n2 = (float *) malloc (sizeof(float)); /* Reservamos espacio para n2 */
    *n2 = 6.7;          /* Valor prefijado para n2 (puntero a real) */

    suma = (float *) malloc (sizeof(float)); /* Reservamos espacio para suma */
    *suma = n1 + *n2;    /* Calculamos la suma */

    printf("El valor prefijado para la suma era %4.2f\n", *suma);

    printf("Ahora es tu turno: Introduce el primer número ");
    scanf("%f",&n1);   /* Leemos valor para n1 (real) */

    printf("Introduce el segundo número ");
    scanf("%f",n2);    /* Valor para n2 (puntero a real) */

    *suma = n1 + *n2;  /* Calculamos nuevamente la suma */

    printf("Ahora la suma es %4.2f\n", *suma);

    free(n2);           /* Liberamos la memoria reservada */
    free(suma);
    return 0;
}
```

Las diferencias son:

- n1 es un "float", así que le damos valor normalmente: n1 = 0;  Y pedimos su valor con scanf usando & para indicar en qué dirección de memoria se encuentra: scanf("%f", &n1);
- n2 (y también "suma") es un "puntero a float", así que debemos reservarle espacio con "malloc" antes de empezar a usarlo, y liberar con "free" el epacio que ocupaba cuando terminemos de utilizarlo. Para guardar un valor en la dirección de memoria "a la que apunta", usamos un asterisco: *n2 = 0;  Y pedimos su valor con scanf, pero sin necesidad de usar &, porque el puntero ES una dirección de memoria: scanf("%f", n2);

(En este ejemplo, no hemos comprobado si el resultado de "malloc" era NULL, porque sólo pedíamos espacio para dos variables, y hemos dado por sentado que sí habría memoria disponible suficiente para almacenarlas; en un caso general, deberemos asegurarnos siempre de que se nos ha concedido ese espacio que hemos pedido).

## 9.4. Aritmética de punteros

Si declaramos una variable como `int n=5` y posteriormente hacemos `n++`, debería resultar claro que lo que ocurre es que aumenta en una unidad el valor de la variable n, pasando a ser 6. Pero ¿qué sucede si hacemos esa misma operación sobre un puntero?

```
int *n;
n = (int *) malloc (sizeof(int));
*n = 3;
n++;
```

Después de estas líneas de programa, lo que ha ocurrido no es que el contenido de la posición n sea 4. Eso lo conseguiríamos modificando "*n", de la misma forma que le hemos dado su valor inicial. Es decir, deberíamos usar

```
(*n) ++;
```

En cambio, nosotros hemos aumentado el valor de "n". Como "n" es un puntero, estamos modificando una dirección de memoria. Por ejemplo, si "n" se refería a la posición de memoria número 10.000 de nuestro ordenador, ahora ya no es así, ahora es otra posición de memoria distinta, por ejemplo la 10.001.

¿Y por qué "por ejemplo"? Porque, como ya sabemos, el espacio que ocupa una variable en C depende del sistema operativo. Así, en un sistema operativo de 32 bits, un "int" ocuparía 4 bytes, de modo que la operación

```
n++;
```

haría que pasáramos de mirar la posición 10.000 a la 10.004. Generalmente no es esto lo que querremos, sino modificar el valor que había almacenado en esa posición de memoria. Olvidar ese * que indica que queremos cambiar el dato y no la posición de memoria puede dar lugar a fallos muy difíciles de descubrir (o incluso a que el programa se interrumpa con un aviso de "Violación de segmento" porque estemos accediendo a zonas de memoria que no hemos reservado).

## 9.5. Punteros y funciones: parámetros por referencia

Hasta ahora no sabíamos cómo modificar los parámetros que pasábamos a una función. Recordemos el ejemplo 64:

```
/*---------------------------*/
/*  Ejemplo en C nº 64:      */
/*  c064.c                   */
/*                           */
/*  Dos variables locales    */
/*  con el mismo nombre      */
/*                           */
/*  Curso de C,              */
/*    Nacho Cabanes          */
```

```
/*--------------------------*/

#include <stdio.h>

void duplica(int n)
{
    n = n * 2;
}

int main()
{
    int n = 5;
    printf("n vale %d\n", n);
    duplica(n);
    printf("Ahora n vale %d\n", n);
    return 0;
}
```

Cuando poníamos este programa en marcha, el valor de n que se mostraba era un 5, porque los cambios que hiciéramos dentro de la función se perdían al salir de ella. Esta forma de trabajar (la única que conocíamos hasta ahora) es lo que se llama "pasar **parámetros por valor**".

Pero existe una alternativa. Es lo que llamaremos "pasar **parámetros por referencia**". Consiste en que el parámetro que nosotros pasamos a la función no es realmente la variable, sino la dirección de memoria en que se encuentra dicha variable (usando &). Dentro de la función, modificaremos la información que se encuentra dentro de esa dirección de memoria (usando *), así:

```
/*--------------------------*/
/*  Ejemplo en C nº 74:     */
/*  c074.c                  */
/*                          */
/*  Modificar el valor de   */
/*  un parámetro            */
/*                          */
/*  Curso de C,             */
/*    Nacho Cabanes         */
/*--------------------------*/

#include <stdio.h>

void duplica(int *x)
{
   *x = *x * 2;
}

int main()
{
   int n = 5;
   printf("n vale %d\n", n);
   duplica(&n);
   printf("Ahora n vale %d\n", n);
   return 0;
}
```

Esto permite que podamos obtener más de un valor a partir de una función. Por ejemplo, podemos crear una función que intercambie los valores de dos variables enteras así:

```
/*---------------------------*/
/*  Ejemplo en C nº 75:      */
/*  c075.c                   */
/*                           */
/*  Intercambiar el valor de */
/*  dos parámetros           */
/*                           */
/*  Curso de C,              */
/*    Nacho Cabanes          */
/*---------------------------*/

#include <stdio.h>

void intercambia(int *x, int *y)
{
   int auxiliar;
   auxiliar = *x;
   *x = *y;
   *y = auxiliar ;
}

int main()
{
   int a = 5;
   int b = 12;
   intercambia(&a, &b);
   printf("Ahora a es %d y b es %d\n", a, b);
   return 0;
}
```

Este programa escribirá en pantalla que a vale 12 y que b vale 5. Dentro de la función "intercambia", nos ayudamos de una variable auxiliar para memorizar el valor de x antes de cambiarlo por el valor de y.

**Ejercicio propuesto**: (9.5.1) Crear una función que calcule las dos soluciones de una ecuación de segundo grado ($Ax^2 + Bx + C = 0$) y devuelva las dos soluciones como parámetros.

## *9.6. Punteros y arrays*

En C hay muy poca diferencia "interna" entre un puntero y un array. En muchas ocasiones, podremos declarar un dato como array (una tabla con varios elementos iguales, de tamaño predefinido) y recorrerlo usando punteros. Vamos a ver un ejemplo:

```
/*---------------------------*/
/*  Ejemplo en C nº 76:      */
/*  c076.c                   */
/*                           */
/*  Arrays y punteros (1)    */
/*                           */
/*  Curso de C,              */
```

```
/*    Nacho Cabanes          */
/*---------------------------*/

#include <stdio.h>

int main()
{
    int datos[10];
    int i;

    /* Damos valores normalmente */
    for (i=0; i<10; i++)
        datos[i] = i*2;

    /* Pero los recorremos usando punteros */
    for (i=0; i<10; i++)
        printf ("%d ", *(datos+i));
    return 0;
}
```

Pero también podremos hacer lo contrario: declarar de forma dinámica una variable usando "malloc" y recorrerla como si fuera un array:

```
/*---------------------------*/
/*  Ejemplo en C nº 77:      */
/*  c077.c                   */
/*                           */
/*  Arrays y punteros (2)    */
/*                           */
/*  Curso de C,              */
/*    Nacho Cabanes          */
/*---------------------------*/

#include <stdio.h>
#include <stdlib.h>

int main()
{
    int *datos;
    int i;

    /* Reservamos espacio */
    datos = (int *) malloc (20*sizeof(int));

    /* Damos valores como puntero */
    printf("Uso como puntero... ");
    for (i=0; i<20; i++)
        *(datos+i) = i*2;
    /* Y los mostramos */
    for (i=0; i<20; i++)
        printf ("%d ", *(datos+i));

    /* Ahora damos valores como array */
    printf("\nUso como array... ");
    for (i=0; i<20; i++)
        datos[i] = i*3;
    /* Y los mostramos */
```

```
    for (i=0; i<20; i++)
        printf ("%d ", datos[i]);

    /* Liberamos el espacio */
    free(datos);
    return 0;
}
```

## 9.7. Arrays de punteros

Igual que creamos "arrays" para guardar varios datos que sean números enteros o reales, podemos hacerlo con punteros: podemos reservar espacio para "20 punteros a enteros" haciendo

```
int *datos[20];
```

Tampoco es algo especialmente frecuente en un caso general, porque si fijamos la cantidad de datos, estamos perdiendo parte de la versatilidad que podríamos tener al usar memoria dinámica. Pero sí es habitual cuando se declaran varias cadenas:

```
char *mensajesError[3]={"Fichero no encontrado",
    "No se puede escribir",
    "Fichero sin datos"};
```

Un ejemplo de su uso sería este:

```
/*---------------------------*/
/*  Ejemplo en C nº 78:      */
/*  c078.c                   */
/*                           */
/*  Arrays de punteros       */
/*                           */
/*  Curso de C,              */
/*    Nacho Cabanes          */
/*---------------------------*/

#include <stdio.h>

int main()
{
    char *mensajesError[3]={"Fichero no encontrado",
        "No se puede escribir",
        "Fichero sin datos"};

    printf("El primer mensaje de error es: %s\n",
        mensajesError[0]);
    printf("El segundo mensaje de error es: %s\n",
        mensajesError[1]);
    printf("El tercer mensaje de error es: %s\n",
        mensajesError[2]);
    return 0;
}
```

## 9.8. Punteros y estructuras

Igual que creamos punteros a cualquier tipo de datos básico, le reservamos memoria con "malloc" cuando necesitamos usarlo y lo liberamos con "free" cuando terminamos de utilizarlo, lo mismo podemos hacer si se trata de un tipo de datos no tan sencillo, como un "struct".

Eso sí, la forma de acceder a los datos en un struct cambiará ligeramente. Para un dato que sea un número entero, ya sabemos que lo declararíamos con `int *n` y cambiaríamos su valor haciendo algo como `*n=2`, de modo que para un struct podríamos esperar que se hiciera algo como `*persona.edad = 20`. Pero esa no es la sintaxis correcta: deberemos utilizar el nombre de la variable y el del campo, con una flecha (->) entre medias, así: `persona->edad = 20`. Vamos a verlo con un ejemplo:

```
/*---------------------------*/
/*  Ejemplo en C nº 79:      */
/*  c079.c                   */
/*                           */
/*  Punteros y structs       */
/*                           */
/*  Curso de C,              */
/*    Nacho Cabanes          */
/*---------------------------*/

#include <stdio.h>

int main()
{
        /* Primero definimos nuestro tipo de datos */
    struct datosPersona
    {
        char nombre[30];
        char email[25];
        int edad;
    };

    /* La primera persona será estática */
    struct datosPersona persona1;
    /* La segunda será dinámica */
    struct datosPersona *persona2;

    /* Damos valores a la persona estática */
    strcpy(persona1.nombre, "Juan");
    strcpy(persona1.email, "j@j.j");
    persona1.edad = 20;

    /* Ahora a la dinámica */
    persona2 = (struct datosPersona*)
        malloc (sizeof(struct datosPersona));
    strcpy(persona2->nombre, "Pedro");
    strcpy(persona2->email, "p@p.p");
    persona2->edad = 21;

    /* Mostramos los datos y liberamos la memoria */
    printf("Primera persona: %s, %s, con edad %d\n",
    persona1.nombre, persona1.email, persona1.edad);
```

```
    printf("Segunda persona: %s, %s, con edad %d\n",
    persona2->nombre, persona2->email, persona2->edad);
    free(persona2);
    return 0;
}
```

**Ejercicio propuesto**: (9.8.1) Mejorar la versión de la agenda que leía todos los datos al principio de la ejecución y guardaba todos los datos cuando terminábamos su uso (apartado 6.4). Esta nueva versión deberá estar preparada para manejar hasta 1000 fichas, pero sólo reservará espacio para las que realmente sean necesarias.

## *9.9. Opciones de la línea de comandos: parámetros de "main"*

Es muy frecuente que un programa que usamos desde la "línea de comandos" tenga ciertas opciones que le indicamos como argumentos. Por ejemplo, bajo Linux o cualquier otro sistema operativo de la familia Unix, podemos ver la lista detallada de ficheros que terminan en .c haciendo

```
ls –l *.c
```

En este caso, la orden sería "ls", y las dos opciones (argumentos o parámetros) que le indicamos son "-l" y "*.c".

Pues bien, estas opciones que se le pasan al programa se pueden leer desde C. La forma de hacerlo es con dos parámetros. El primero (que por convenio se suele llamar "argc") será un número entero que indica cuantos argumentos se han tecleado. El segundo (que se suele llamar "argv") es una tabla de cadenas de texto, que contiene cada uno de esos argumentos.

Por ejemplo, si bajo Windows o MsDos tecleamos la orden "DIR *.EXE P", tendríamos que:

- argc es la cantidad de parámetros, incluyendo el nombre del propio programa (3, en este ejemplo).
- argv[0] es el nombre del programa (DIR, en este caso).
- argv[1] es el primer argumento (*.EXE).
- argv[2] es el segundo argumento (/P).

Un fuente en C de ejemplo, que mostrara todos los parámetros que se han tecleado sería:

```
/*--------------------------*/
/*  Ejemplo en C nº 80:     */
/*  c080.c                  */
/*                          */
/*  Argumentos de "main"    */
/*                          */
/*  Curso de C,             */
/*    Nacho Cabanes         */
/*--------------------------*/

#include <stdio.h>
```

```
int main (int argc, char *argv[])
{
  int i;

  printf ("Nombre del programa:\"%s\".\n",argv[0]);
  if (argc > 0)
      for (i = 1; i<argc; i++)
           printf("Parámetro %d = %s\n", i, argv[i]);
  else
      printf("No se han indicado parámetros.\n");
  return 0;
}
```

**Ejercicios propuestos**:

- (9.9.1) Crear un programa llamado "suma", que calcule (y muestre) la suma de dos números que se le indiquen como parámetro. Por ejemplo, si se teclea "suma 2 3" deberá responder "5", y si se teclea "suma 2" deberá responder "no hay suficientes datos (pista: deberás usar "atoi" o "sscanf" para obtener el valor numérico de cada parámetro).
- (9.9.2) Crear una calculadora básica, llamada "calcula", que deberá sumar, restar, multiplicar o dividir los dos números que se le indiquen como parámetros. Ejemplos de su uso sería "calcula 2 + 3" o "calcula 5 * 60".

## *9.10. Estructuras dinámicas habituales 1: las listas enlazadas*

Ahora vamos a ver dos tipos de estructuras totalmente dinámicas (que puedan aumentar o disminuir realmente de tamaño durante la ejecución del programa). Primero veremos las **listas**, y más adelante los árboles binarios. Hay otras muchas estructuras, pero no son difíciles de desarrollar si se entienden bien estas dos.

Ahora "el truco" consistirá en que dentro de cada dato almacenaremos todo lo que nos interesa, pero también una referencia que nos dirá dónde tenemos que ir a buscar el siguiente.

Sería algo como:

```
(Posición: 1023).
Nombre         : 'Nacho Cabanes'
Web            : 'www.nachocabanes.com'
SiguienteDato  : 1430
```

Este dato está almacenado en la posición de memoria número 1023. En esa posición guardamos el nombre y la dirección (o lo que nos interese) de esta persona, pero también una información extra: la siguiente ficha se encuentra en la posición 1430.

Así, es muy cómodo recorrer la lista de forma **secuencial**, porque en todo momento sabemos dónde está almacenado el siguiente dato. Cuando lleguemos a uno para el que no esté definido cual es el siguiente dato, quiere decir que se ha acabado la lista.

Por tanto, en cada dato tenemos un enlace con el dato siguiente. Por eso este tipo de estructuras recibe el nombre de "listas simplemente enlazadas" o **listas simples**. Si tuvieramos enlaces hacia el dato siguiente y el posterior, se trataría de una "lista doblemente enlazada" o **lista doble**, que pretende hacer más sencillo el recorrido hacia delante o hacia atrás.

Con este tipo de estructuras de información, hemos perdido la ventaja del acceso directo: ya no podemos saltar directamente a la ficha número 500. Pero, por contra, podemos tener tantas fichas como la memoria nos permita, y eliminar una (o varias) de ellas cuando queramos, recuperando inmediatamente el espacio que ocupaba.

Para añadir una ficha, no tendríamos más que reservar la memoria para ella, y el compilador de C nos diría "le he encontrado sitio en la posición 4079". Entonces nosotros iríamos a la última ficha y le diríamos "tu siguiente dato va a estar en la posición 4079".

Esa es la "idea intuitiva". Ahora vamos a concretar cosas en forma de programa en C.

Primero veamos cómo sería ahora cada una de nuestras fichas:

```
struct f {           /* Estos son los datos que guardamos: */
  char nombre[30];          /*   Nombre, hasta 30 letras */
  char direccion[50];       /*   Direccion, hasta 50 */
  int edad;                 /*   Edad, un numero < 255 */
  struct f* siguiente;      /*   Y dirección de la siguiente */
}
```

La diferencia con un "struct" normal está en el campo "siguiente" de nuestro registro, que es el que indica donde se encuentra la ficha que va después de la actual, y por tanto será otro puntero a un registro del mismo tipo, un "struct f *".

Un puntero que "no apunta a ningún sitio" tiene el valor **NULL** (realmente este identificador es una constante de valor 0), que nos servirá después para comprobar si se trata del final de la lista: todas las fichas "apuntarán" a la siguiente, menos la última, que "no tiene siguiente", y apuntará a NULL.

Entonces la primera ficha definiríamos con

```
  struct f *dato1;            /* Va a ser un puntero a ficha */
```

y la comenzaríamos a usar con

```
  dato1 = (struct f*) malloc (sizeof(struct f)); /* Reservamos memoria */
  strcpy(dato1->nombre, "Pepe");                 /* Guardamos el nombre, */
  strcpy(dato1->direccion, "Su casa");           /*   la dirección */
  dato1->edad = 40;                              /*   la edad */
  dato1->siguiente = NULL;                       /* y no hay ninguna más */
```

(No debería haber anada nuevo: ya sabemos cómo reservar memoria usando "malloc" y como acceder a los campos de una estructura dinámica usando ->).

Ahora que ya tenemos una ficha, podríamos **añadir** otra ficha detrás de ella. Primero guardamos espacio para la nueva ficha, como antes:

```
struct f *dato2;

dato2 = (struct f*) malloc (sizeof(struct f)); /* Reservamos memoria */
strcpy(dato2->nombre, "Juan");           /* Guardamos el nombre, */
strcpy(dato2->direccion, "No lo sé");    /*   la dirección */
dato2->edad = 35;                        /*   la edad */
dato2->siguiente = NULL;                 /* y no hay ninguna más */
```

y ahora enlazamos la anterior con ella:

```
dato1->siguiente = dato2;
```

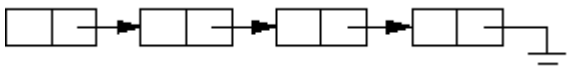
Si quisieramos introducir los datos **ordenados alfabéticamente**, basta con ir comparando cada nuevo dato con los de la lista, e insertarlo donde corresponda. Por ejemplo, para insertar un nuevo dato entre los dos anteriores, haríamos:

```
dato3 = (struct f*) malloc (sizeof(struct f)); /* La tercera */
strcpy(dato3->nombre, "Carlos");
strcpy(dato3->direccion, "Por ahí");
dato3->edad = 14;
dato3->siguiente = dato2;                /* enlazamos con la siguiente */
dato1->siguiente = dato3;                /* y la anterior con ella */
printf("La lista inicialmente es:\n");
```

La estructura que hemos obtenido es la siguiente

```
Dato1 - Dato3 - Dato2 - NULL
```

Gráficamente:



Es decir: cada ficha está enlazada con la siguiente, salvo la última, que no está enlazada con ninguna (apunta a NULL).

Si ahora quisiéramos **borrar** Dato3, tendríamos que seguir dos pasos:

1.- Enlazar Dato1 con Dato2, para no perder información.
2.- Liberar la memoria ocupada por Dato3.

Esto, escrito en "C" sería:

```
  dato1->siguiente = dato2;        /* Borrar dato3: Enlaza Dato1 y Dato2 */
  free(dato3);                    /* Libera lo que ocupó Dato3 */
```

Hemos empleado tres variables para guardar tres datos. Si tenemos 20 datos, ¿necesitaremos 20 variables? ¿Y 3000 variables para 3000 datos?

Sería tremendamente ineficiente, y no tendría mucho sentido. Es de suponer que no sea así. En la práctica, basta con dos variables, que nos indicarán el principio de la lista y la posición actual, o incluso sólo una para el principio de la lista.

Por ejemplo, una rutina que **muestre en pantalla** toda la lista se podría hacer de forma recursiva así:

```
void MuestraLista ( struct f *inicial )
{
    if (inicial!=NULL)                      /* Si realmente hay lista */
    {
        printf("Nombre: %s\n", inicial->nombre);
        printf("Dirección: %s\n", inicial->direccion);
        printf("Edad: %d\n\n", inicial->edad);
        MuestraLista ( inicial->siguiente );    /* Y mira el siguiente */
    }
}
```

Lo llamaríamos con "MuestraLista(Dato1)", y a partir de ahí el propio procedimiento se encarga de ir mirando y mostrando los siguientes elementos hasta llegar a NULL, que indica el final.

Antes de seguir, vamos a juntar todo esto en un programa, para comprobar que realmente funciona: añadimos los 3 datos y decimos que los muestre desde el primero; luego borramos el del medio y los volvemos a mostrar:

```
/*--------------------------*/
/*  Ejemplo en C nº 81:     */
/*  c081.c                  */
/*                          */
/*  Primer ejemplo de lista */
/*  enlazada simple         */
/*                          */
/*  Curso de C,             */
/*    Nacho Cabanes         */
/*--------------------------*/

#include <stdio.h>
#include <stdlib.h>

struct f            /* Estos son los datos que guardamos: */
{
    char nombre[30];          /*   Nombre, hasta 30 letras */
    char direccion[50];       /*   Direccion, hasta 50 */
    int edad;                 /*   Edad, un numero < 255 */
    struct f* siguiente;      /*   Y dirección de la siguiente */
};
```

```
struct f *dato1;            /* Va a ser un puntero a ficha */
struct f *dato2;            /* Otro puntero a ficha */
struct f *dato3;            /* Y otro más */

void MuestraLista ( struct f *inicial )
{
    if (inicial!=NULL)                    /* Si realmente hay lista */
    {
        printf("Nombre: %s\n", inicial->nombre);
        printf("Dirección: %s\n", inicial->direccion);
        printf("Edad: %d\n\n", inicial->edad);
        MuestraLista ( inicial->siguiente );    /* Y mira el siguiente */
    }
}

int main()
{
    dato1 = (struct f*) malloc (sizeof(struct f)); /* Reservamos memoria */
    strcpy(dato1->nombre, "Pepe");           /* Guardamos el nombre, */
    strcpy(dato1->direccion, "Su casa");     /*   la dirección */
    dato1->edad = 40;                        /*   la edad */
    dato1->siguiente = NULL;                 /* y no hay ninguna más */

    dato2 = (struct f*) malloc (sizeof(struct f)); /* Reservamos memoria */
    strcpy(dato2->nombre, "Juan");           /* Guardamos el nombre, */
    strcpy(dato2->direccion, "No lo sé");    /*   la dirección */
    dato2->edad = 35;                        /*   la edad */
    dato2->siguiente = NULL;                 /* y no hay ninguna más */
    dato1->siguiente = dato2;                /* Enlazamos anterior con ella */

    dato3 = (struct f*) malloc (sizeof(struct f)); /* La tercera */
    strcpy(dato3->nombre, "Carlos");
    strcpy(dato3->direccion, "Por ahí");
    dato3->edad = 14;
    dato3->siguiente = dato2;                /* enlazamos con la siguiente */
    dato1->siguiente = dato3;                /* y la anterior con ella */
    printf("La lista inicialmente es:\n");

    MuestraLista (dato1);
    dato1->siguiente = dato2;        /* Borrar dato3: Enlaza Dato1 y Dato2 */
    free(dato3);                     /* Libera lo que ocupó Dato3 */
    printf("Y tras borrar dato3:\n\n");
    MuestraLista (dato1);
    return 0;
}
```

Vamos a ver otro ejemplo, que cree una lista de números, y vaya insertando en ella varios valores **ordenados**. Ahora tendremos una función para mostrar datos, otra para crear la lista insertando el primer dato, y otra que inserte un dato ordenado

```
/*--------------------------*/
/*  Ejemplo en C nº 82:     */
/*  c082.c                  */
/*                          */
/*  Segundo ejemplo de lista */
/*  enlazada (ordenada)     */
/*                          */
```

```
/*  Curso de C,              */
/*    Nacho Cabanes          */
/*---------------------------*/

#include <stdio.h>
#include <stdlib.h>

struct lista              /* Cada nodo de nuestra lista */
{
    int numero;           /* Solo guarda un numero */
    struct lista* sig;    /* Y el puntero al siguiente dato */
};

struct lista* CrearLista(int valor)   /* Crea la lista, claro */
{
    struct lista* r;        /* Variable auxiliar */
    r = (struct lista*)
        malloc (sizeof(struct lista)); /* Reserva memoria */
    r->numero = valor;      /* Guarda el valor */
    r->sig = NULL;          /* No hay siguiente */
    return r;               /* Crea el struct lista* */
}

void MuestraLista ( struct lista *lista )  /* Muestra toda la lista, recursivo */
{
    if (lista)                          /* Si realmente hay lista */
    {
        printf("%d\n", lista->numero);    /* Escribe el valor */
        MuestraLista (lista->sig );       /* Y mira el siguiente */
    }
}

void InsertaLista( struct lista **lista, int valor)
{
    struct lista* r;        /* Variable auxiliar, para reservar */
    struct lista* actual;   /* Otra auxiliar, para recorrer */

    actual = *lista;
    if (actual)                                /*  Si hay lista */
        if (actual->numero < valor)            /*    y todavía no es su sitio */
            InsertaLista(&actual->sig,valor); /*   mira la siguiente posición */
        else                          /* Si hay lista pero ya es su sitio */
        {
            r = CrearLista(valor);    /* guarda el dato */
            r->sig = actual;          /* pone la lista a continuac. */
            *lista = r;               /* Y hace que comience en el nuevo dato */
        }
    else               /* Si no hay lista, hay que crearla */
    {
        r = CrearLista(valor);
        *lista = r;       /* y hay que indicar donde debe comenzar */
    }
}

int main()
{
    struct lista* l;              /* La lista que crearemos */
    l = CrearLista(5);            /* Crea una lista e introduce un 5 */
    InsertaLista(&l, 3);          /* Inserta un 3 */
```

```
    InsertaLista(&l, 2);           /* Inserta un 2 */
    InsertaLista(&l, 6);           /* Inserta un 6 */
    MuestraLista(l);               /* Muestra la lista resultante */
    return 0;                      /* Se acabó */
}
```

No es un fuente fácil de leer (en general, no lo serán los que manejen punteros), pero aun así los cambios no son grandes:

- Hemos automatizado la inserción de un nuevo dato con la función CrearLista.

- Cuando después de ese dato debemos enlazar datos existentes, lo hacemos en dos pasos: `r = CrearLista(valor);  r->sig = actual;`

- Si además el dato que modificamos es el primero de toda la lista, deberemos modificar la dirección de comienzo para que lo refleje: `r = CrearLista(valor); *lista = r;`

- Como ya vimos, cuando queremos modificar un dato de tipo "int", pasamos un parámetro que es de tipo "int *". En este caso, la función InsertaLista puede tener que modificar un dato que es de tipo "struct lista *", por lo que el parámetro a modificar deberá ser de tipo "struct lista **". Al final del tema comentaremos algo más sobre este tipo de expresiones.

- Por otra parte, también hemos abreviado un poco alguna expresión: `if (lista)` es lo mismo que `if (lista!=NULL)` (recordemos que NULL es una constante que vale 0).

Finalmente, hay varios casos particulares que resultan más sencillos que una lista "normal". Vamos a comentar los más habituales:

- Una **pila** es un caso particular de lista, en la que los elementos siempre se introducen y se sacan por el mismo extremo (se apilan o se desapilan). Es como una pila de libros, en la que para coger el tercero deberemos apartar los dos primeros (excluyendo malabaristas, que los hay). Este tipo de estructura se llama **LIFO** (Last In, First Out: el último en entrar es el primero en salir).

- Una **cola** es otro caso particular, en el que los elementos se introducen por un extremo y se sacan por el otro. Es como se supone que debería ser la cola del cine: los que llegan, se ponen al final, y se atiende primero a los que están al principio. Esta es una estructura **FIFO** (First In, First Out).

**Ejercicio propuesto**:

- (9.10.1) Crea una "pila de enteros", que permitirá añadir un nuevo número (lo que llamaremos "apilar") o bien extraer el número que se encuentra en la parte más alta de la pila ("desapilar"). En una primera aproximación, usa un array para guardar los datos.
- (9.10.2) Crea una nueva versión de la "pila de enteros", que utilice memoria dinámica.

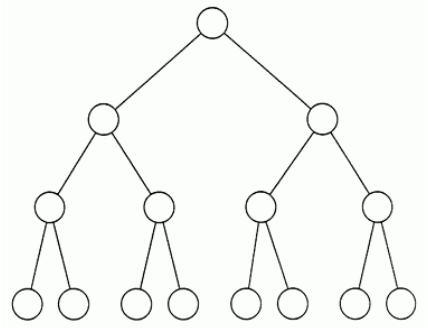
- (9.10.3) Crea una "cola de enteros", que permitirá añadir un nuevo número al final de la cola (lo que llamaremos "encolar") o bien extraer el número que se encuentra al principio de la cola ("desencolar"). En esta primera aproximación, usa un array para guardar los datos.
- (9.10.4) Crea una nueva versión de la "cola de enteros", que utilice memoria dinámica.
- (9.10.5) Las listas simples, tal y como las hemos tratado, tienen la ventaja de que no hay limitaciones tan rígidas en cuanto a tamaño como en las variables estáticas, ni hay por qué saber el número de elementos desde el principio. Pero siempre hay que recorrerlas desde DELANTE hacia ATRAS, lo que puede resultar lento. Una mejora relativamente evidente es lo que se llama una **lista doble** o lista doblemente enlazada: si guardamos punteros al dato anterior y al siguiente, en vez de sólo al siguiente, podremos avanzar y retroceder con comodidad. Implementa una lista doble enlazada que almacene números enteros.

## *9.11. Estructuras dinámicas habituales 2: los árboles binarios*

En las listas, después de cada elemento había otro, el "siguiente" (o ninguno, si habíamos llegado al final).

Pero también nos puede interesar tener varias posibilidades después de cada elemento, varios "hijos", por ejemplo 3. De cada uno de estos 3 "hijos" saldrían otros 3, y así sucesivamente. Obtendríamos algo que recuerda a un árbol: un tronco del que nacen 3 ramas, que a su veces se subdividen en otras 3 de menor tamaño, y así sucesivamente hasta llegar a las hojas.

Pues eso mismo será un árbol: una estructura dinámica en la que cada nodo (elemento) puede tener más de un "siguiente". Nos centraremos en los **árboles binarios**, en los que cada nodo puede tener un hijo izquierdo, un hijo derecho, ambos o ninguno (dos hijos como máximo).



Para puntualizar aun más, trataremos los árboles binarios de **búsqueda**, en los que tenemos prefijado un cierto orden, que nos ayudará a encontrar un cierto dato dentro de un árbol con mucha rapidez.

Este **"orden** prefijado" será el siguiente: para cada nodo tendremos que

- la rama de la izquierda contendrá elementos menores que él.
- la rama de la derecha contendrá elementos mayores que él.

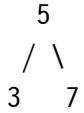
Para que se entienda mejor, vamos a introducir en un árbol binario de búsqueda los datos 5,3,7,2,4,8,9

Primer número: 5 (directo)

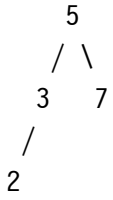
5

Segundo número: 3 (menor que 5)

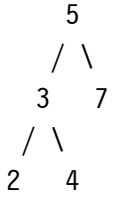

Tercer número: 7 (mayor que 5)



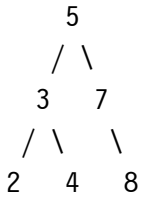

Cuarto: 2 (menor que 5, menor que 3)



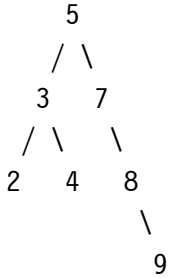

Quinto: 4 (menor que 5, mayor que 3)




Sexto: 8 (mayor que 5, mayor que 7)




Séptimo: 9 (mayor que 5, mayor que 7, mayor que 8)



¿Qué **ventajas** tiene esto? La rapidez: tenemos 7 elementos, lo que en una lista supone que si buscamos un dato que casualmente está al final, haremos 7 comparaciones; en este árbol, tenemos 4 alturas => 4 comparaciones como máximo.

Y si además hubiéramos "**equilibrado**" el árbol (irlo recolocando, de modo que siempre tenga la menor altura posible), serían 3 alturas. Esto es lo que se hace en la práctica cuando en el árbol se va a hacer muchas más lecturas que escrituras: se reordena internamente después de añadir cada nuevo dato, de modo que la altura sea mínima en cada caso. De este modo, el número máximo de comparaciones que tendríamos que hacer sería $\log_2(n)$, lo que supone que si tenemos 1000 datos, en una lista podríamos llegar a tener que hacer 1000 comparaciones, y en un árbol binario, $\log_2(1000)$ => 10 comparaciones como máximo. La ganancia en velocidad de búsqueda es clara.

No vamos a ver cómo se hace eso de los "equilibrados", que sería propio de un curso de programación más avanzado, pero sí vamos a empezar a ver rutinas para manejar estos árboles binarios de búsqueda.

Recordemos que la idea importante es que todo dato menor estará a la izquierda del nodo que miramos, y los datos mayores estarán a su derecha.

Ahora la estructura de cada **nodo** (dato) será:

```
struct arbol {             /*  El tipo base en sí: */
      int dato;               /*    - un dato (entero) */
      struct arbol* hijoIzq;  /*    - puntero a su hijo izquierdo */
      struct arbol* hijoDer;  /*    - puntero a su hijo derecho */
 }
```

Y las rutinas de inserción, búsqueda, escritura, borrado, etc., podrán ser recursivas. Como primer ejemplo, la de **escritura** de todo el árbol en orden sería:

```
void Escribir(struct arbol *punt)
{
    if (punt)                  /*  Si no hemos llegado a una hoja */
    {
    Escribir(punt->hijoIzq);   /*  Mira la izqda recursivamente */
    printf("%d ",punt->dato);  /*  Escribe el dato del nodo */
    Escribir(punt->hijoDer);   /*  Y luego mira por la derecha */
    }
}
```

Quien no se crea que funciona, debería coger lápiz y papel comprobarlo con el árbol que hemos visto antes como ejemplo. Es muy importante que esta función quede clara antes de seguir leyendo, porque los demás serán muy parecidos.

La rutina de **inserción** sería parecida, aunque algo más "pesada" porque tenemos que pasar el puntero por referencia, para que se pueda modificar el puntero:

```
void Insertar(struct arbol **punt, int valor)
{
    struct arbol * actual= *punt;
    if (actual == NULL)              /*  Si hemos llegado a una hoja */
    {
      *punt = (struct arbol *)
         malloc (sizeof(struct arbol));  /* Reservamos memoria */
      actual= *punt;
      actual->dato = valor;         /* Guardamos el dato */
      actual->hijoIzq = NULL;       /* No tiene hijo izquierdo */
      actual->hijoDer = NULL;       /* Ni derecho */
    }
    else                                  /*      Si no es hoja      */
   if (actual->dato > valor)              /* Y encuentra un dato mayor */
      Insertar(&actual->hijoIzq, valor);  /* Mira por la izquierda */
   else                                   /* En caso contrario (menor) */

      Insertar(&actual->hijoDer, valor);  /* Mira por la derecha */
}
```

Y finalmente, la de **borrado** de todo el árbol, casi igual que la de escritura, sólo que en vez de borrar la izquierda, luego el nodo y luego la derecha, borraremos primero las dos ramas y en último lugar el nodo, para evitar incongruencias (intentar borrar el hijo de algo que ya no existe):

```
void Borrar(struct arbol *punt)
{
    if (punt)                  /*  Si no hemos llegado a una hoja */
    {
    Borrar(punt->hijoIzq);     /*  Va a la izqda recursivamente */
    Borrar(punt->hijoDer);     /*  Y luego a la derecha */
    free (punt);               /*  Finalmente, libera lo que ocupa el nodo */
    }
}
```

Vamos a juntar todo esto en un ejemplo "que funcione":

```
/*--------------------------*/
/*  Ejemplo en C nº 83:     */
/*  c083.c                  */
/*                          */
/*  Arbol binario de        */
/*  búsqueda                */
/*                          */
/*  Curso de C,             */
/*    Nacho Cabanes         */
/*--------------------------*/
```

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

struct arbol               /*  El tipo base en sí: */
{
    int dato;               /*    - un dato (entero) */
    struct arbol* hijoIzq;  /*    - puntero a su hijo izquierdo */
    struct arbol* hijoDer;  /*    - puntero a su hijo derecho */
 };

void Escribir(struct arbol *punt)
{
    if (punt)                  /*  Si no hemos llegado a una hoja */
    {
        Escribir(punt->hijoIzq);   /*  Mira la izqda recursivamente */
        printf("%d ",punt->dato);  /*  Escribe el dato del nodo */
        Escribir(punt->hijoDer);   /*  Y luego mira por la derecha */
    }
}

void Insertar(struct arbol **punt, int valor)
{
    struct arbol * actual= *punt;
    if (actual == NULL)              /*  Si hemos llegado a una hoja */
    {
        *punt = (struct arbol *)
            malloc (sizeof(struct arbol));  /* Reservamos memoria */
        actual= *punt;
        actual->dato = valor;         /* Guardamos el dato */
        actual->hijoIzq = NULL;       /* No tiene hijo izquierdo */
        actual->hijoDer = NULL;       /* Ni derecho */
    }
    else                                        /*      Si no es hoja      */
        if (actual->dato > valor)               /* Y encuentra un dato mayor */
            Insertar(&actual->hijoIzq, valor);  /* Mira por la izquierda */
        else                                    /* En caso contrario (menor) */
            Insertar(&actual->hijoDer, valor);  /* Mira por la derecha */
}

/*  Cuerpo del programa  */
int main()
{
    struct arbol *arbol = NULL;
    Insertar(&arbol, 5);
    Insertar(&arbol, 3);
    Insertar(&arbol, 7);
    Insertar(&arbol, 2);
    Insertar(&arbol, 4);
    Insertar(&arbol, 8);
    Insertar(&arbol, 9);
    Escribir(arbol);
    return 0;
}
```

## 9.12. Indirección múltiple

Lo que estamos haciendo mediante los punteros es algo que técnicamente se conoce como "direccionamiento indirecto": cuando hacemos `int *n`, generalmente no nos va a interesar el valor de n, sino que n es una dirección de memoria a la que debemos ir a mirar el valor que buscamos.

Pues bien, podemos hacer que ese direccionamiento sea todavía menos directo que en el caso normal: algo como `int **n` se referiría a que n es una dirección de memoria, en la que a su vez se encuentra como dato otra dirección de memoria, y dentro de esta segunda dirección de memoria es donde se encuentra el dato. Es decir, n sería un "puntero a puntero a entero".

Esta no es una situación habitual, así que no profundizaremos más en ella.

Aun así, lo que sí se debe recordar es que `char **datos` es algo muy parecido a `char datos[][]`, por lo que alguna vez lo veremos indicado como parámetros de una función de una forma o de la otra.

## 9.13. Un ejemplo: copiador de ficheros en una pasada

Como ejemplo de un fuente en el que se apliquen algunas de las ideas más importantes que hemos visto, vamos a crear un copidor de ficheros, que intente copiar todo el fichero de origen en una única pasada: calculará su tamaño, intentará reservar la memoria suficiente para almacenar todo el fichero a la vez, y si esa memoria está disponible, leerá el fichero completo y lo guardará con un nuevo nombre.

```
/*---------------------------*/
/*  Ejemplo en C nº 84:      */
/*  c084.c                   */
/*                           */
/*  Copiador de ficheros en  */
/*  una pasada               */
/*                           */
/*  Curso de C,              */
/*    Nacho Cabanes          */
/*---------------------------*/

#include <stdio.h>

FILE *fichOrg, *fichDest;      /* Los dos ficheros */
char *buffer;                  /* El buffer para guardar lo que leo */
char nombreOrg[80],            /* Los nombres de los ficheros */
    nombreDest[80];
long longitud;                 /* Tamaño del fichero */
long cantidad;                 /* El número de bytes leídos */

int main()
{
    /* Accedo al fichero de origen */
    printf("Introduzca el nombre del fichero Origen: ");
    scanf("%s",nombreOrg);
    if ((fichOrg = fopen(nombreOrg, "rb")) == NULL)
    {
```

```
        printf("No existe el fichero origen!\n");
        exit(1);
    }
    /* Y ahora al de destino */
    printf("Introduzca el nombre del fichero Destino: ");
    scanf("%s",nombreDest);
    if ((fichDest = fopen(nombreDest, "wb")) == NULL)
    {
        printf("No se ha podido crear el fichero destino!\n");
        exit(2);
    }

    /* Miro la longitud del fichero de origen */
    fseek(fichOrg, 0, SEEK_END);
    longitud = ftell(fichOrg);
    fseek(fichOrg, 0, SEEK_SET);

    /* Reservo espacio para leer todo */
    buffer = (char *) malloc (longitud);
    if (buffer == NULL)
    {
        printf("No se ha podido reservar tanto espacio!\n");
        exit(3);
    }

    /* Leo todos los datos a la vez */
    cantidad = fread( buffer, 1, longitud, fichOrg);

    /* Escribo tantos como haya leído */
    fwrite(buffer, 1, cantidad, fichDest);

    if (cantidad != longitud)
        printf("Cuidado: no se han leido (ni copiado) todos los datos\n");

    /* Cierro los ficheros */
    fclose(fichOrg);
    fclose(fichDest);
    return 0;
}
```

# 10. Bibliotecas de uso frecuente

Este tema va a tratar algunas de las bibliotecas adicionales que se pueden acompañar junto con un compilador de lenguaje C. Su lectura no es imprescindible para entender los temas anteriores ni el tema posterior. Puedes saltar los apartados que traten posibilidades que consideres que no vas a utilizar.

## *10.1. Llamadas al sistema: system*

Si hay algo que no sepamos o podamos hacer, pero que alguna utilidad del sistema operativo sí es capaz de hacer por nosotros, podemos hacer que ella trabaje por nosotros. La forma de llamar a otras órdenes del sistema operativo (incluso programas externos de casi cualquier tipo) es utilizar la orden "system". Por ejemplo, podríamos mostrar la lista de ficheros de la carpeta actual con la orden "ls", así:

```
/*--------------------------*/
/*  Ejemplo en C nº 85:     */
/*  c085.c                  */
/*                          */
/*  Llamadas a servicios del */
/*  sistema                 */
/*                          */
/*  Curso de C,             */
/*    Nacho Cabanes         */
/*--------------------------*/

#include <stdio.h>
#include <stdlib.h>

int main()
{
    printf("El contenido de la carpeta actual es:\n");
    system ("ls");
    return 0;
}
```

## *10.2. Temporización*

Si lo que queremos es hacer una pausa en un programa, en ocasiones tendremos funciones que nos permitan hacer una pausa de ciertas milésimas de segundo, como la función "delay" de Turbo C.

```
/*--------------------------*/
/*  Ejemplo en C nº 86:     */
/*  c086.c                  */
/*                          */
/*  Pausa con Turbo C       */
/*                          */
/*  Curso de C,             */
/*    Nacho Cabanes         */
/*--------------------------*/
```

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

int main()
{
    printf("Vamos a esperar 2 segundos... ");
    delay(2000);
    printf("El tiempo ha pasado.\n");
    return 0;
}
```

Si queremos comprobar la fecha y hora del sistema, lo podemos hacer con las funciones disponibles en "time.h", que sí son parte del estandar ANSI C, por lo que deberían estar disponibles para casi cualquier compilador:

```
/*--------------------------*/
/*  Ejemplo en C nº 87:     */
/*  c087.c                  */
/*                          */
/*  Leer fecha y hora       */
/*                          */
/*  Curso de C,             */
/*    Nacho Cabanes         */
/*--------------------------*/

#include <stdio.h>
#include <time.h>

int main()
{
    time_t segundos;
    struct tm *fecha;

    segundos = time(NULL);
    printf("Instante actual: %u s\n", segundos);

    fecha = gmtime(&segundos);
    printf("Como texto es: %s\n", asctime(fecha));

    printf("En concreto, la hora Greenwich es: %2d:%02d:%02d\n",
        fecha->tm_hour, fecha->tm_min, fecha->tm_sec);

    return 0;
}
```

Dentro de "time.h", tenemos definido un tipo llamado "time_t" que representará a una cierta fecha (incluyendo hora). La mayoría de los sistemas lo representan internamente como un entero largo (el número de segundos desde cierta fecha), aunque es algo que a nosotros no nos debería hacer falta saber si usamos directamente ese tipo "time_t".

Tenemos también un tipo de registro (struct) predefinido, llamdo "struct tm", que guarda la información desglosada del día, el mes, el año, la hora, etc. Los principales campos que contiene son éstos:

```
    int tm_hour;   /* hora (0 - 23) */
    int tm_mday;   /* Dia del mes (1 - 31) */
    int tm_min;    /* Minutos (0 - 59) */
    int tm_mon;    /* Mes (0 - 11 : 0 = Enero) */
    int tm_sec;    /* Segundos (0 - 59) */
    int tm_wday;   /* Dia de la semana (0 - 6 : 0 = Domingo) */
    int tm_yday;   /* Dia del año (0 - 365) */
    int tm_year;   /* Año menos 1900 */
```

y el modo de usarlas se ve en el fuente anterior:

```
    printf("En concreto, la hora Greenwich es: %2d:%02d:%02d\n",
        fecha->tm_hour, fecha->tm_min, fecha->tm_sec);
```

Como hemos visto en este ejemplo, tenemos varias funciones para manipular la fecha y la hora:

- "time" devuelve el número de segundos que han pasado desde el 1 de enero de 1970. Su uso habitual es `hora = time(NULL);`
- "gmtime" convierte ese número de segundos que nos indica "time" a una variable de tipo "struct tm *" para que podamos conocer detalles como la hora, el minuto o el mes. En la conversión, devuelve la hora universal (UTC o GMT, hora en Greenwich), que puede no coincidir con la hora local.
- "localtime" es similar, pero devuelve la hora local, en vez de la hora universal (el sistema debe saber correctamente en qué zona horaria nos encontramos).
- "asctime" convierte un dato horario de tipo "struct tm *" a una cadena de texto que representa fecha, hora y día de la semana, siguiendo el formato `Sat May 20 15:21:51 2000` (día de la semana en inglés abreviado a 3 letras, mes en inglés abreviado a 3 letras, número de día, horas, minutos, segundos, año).

Pero aún hay más:

- "difftime" calcula la diferencia entre dos fechas.
- "mktime" crea un dato de tipo "struct tm *" a partir de otro incompleto. Es útil por ejemplo para saber el día de la semana si conocemos el día, mes y año.

Si queremos imitar el funcionamiento de la orden "delay" de Turbo C, lo podemos hacer leyendo continuamente la fecha y la hora, o bien usar la función "clock()", que da una estimación (lo más aproximada que el sistema permita) del tiempo que hace que nuestro programa comenzó a ponerse en marcha:

```
/*---------------------------*/
/*  Ejemplo en C nº 88:      */
/*  c088.c                   */
/*                           */
/*  Pausa usando "clock"     */
/*                           */
/*  Curso de C,              */
/*    Nacho Cabanes          */
/*---------------------------*/
```

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

void espera (int segundos)
{
    clock_t instanteFinal;
    instanteFinal = clock () + segundos * CLOCKS_PER_SEC ;
    while (clock() < instanteFinal)
    {
        /* No hacer nada, solo esperar... */
    }
}

int main ()
{
    int n;
    printf ("Comienza la cuenta atras...\n");
    for (n=10; n>0; n--)
    {
        printf ("%d\n",n);
        espera (1);
    }
    printf ("Terminado!\n");
    return 0;
}
```

Nota: en Turbo C no existe la constante CLOCKS_PER_SEC, sino una llamada CLK_TCK con el mismo significado ("ticks" del reloj en cada segundo, para poder convertir a segundos el valor que nos indica "clock()").

## *10.3. Pantalla y teclado con Turbo C*

La familia Turbo C / Turbo C++ / Borland C++ incluía una serie de compiladores creados por Borland para Dos y para Windows. Con ellos se podía utilizar ciertas órdenes para escribir en cualquier posición de la pantalla, para usar colores, para comprobar qué tecla se había pulsado, etc. Eso sí, estas órdenes no son C estándar, así que lo más habitual es que no se encuentren disponibles para otros compiladores o para otros sistemas operativos.

Aun así, como primer acercamiento al control de estos dispositivos desde Linux, puede ser interesante conocer lo que ofrecía la familia de Turbo C y posteriores, porque sientan muchas de las bases que después utilizaremos, pero a la vez se trata de funciones muy sencillas.

Comencemos por las más habituales en cuanto a manejo de pantalla:

- clrscr - Borra la pantalla.
- gotoxy - Desplaza el cursor a ciertas coordenadas (X, la primera, indicará la columna; Y, la segunda, será la fila).
- textcolor - Cambia el color del texto (el de primer plano).
- textbackground - Cambia el color del texto (el de fondo).
- textattr - Cambia el color (fondo y primer plano) del texto.
- cprintf - Escribe un mensaje en color.

- cputs - Escribe una cadena de texto en color.

Por lo que respecta al teclado, tenemos

- getch - Espera hasta que se pulse una tecla, pero no la muestra en pantalla.
- getche - Espera hasta que se pulse una tecla, y la muestra en pantalla.
- kbhit - Comprueba si se ha pulsado alguna tecla (pero no espera).

Todas ellas se encuentran definidas en el fichero de cabecera "conio.h", que deberemos incluir en nuestro programa.

Los colores de la pantalla se indican por números. Por ejemplo: 0 es el negro, 1 es el azul oscuro, 2 el verde, 3 el azul claro, 4 el rojo, etc. Aun así, para no tener que recordarlos, tenemos definidas constantes con su nombre en inglés:

```
BLACK, BLUE, GREEN, CYAN, RED, MAGENTA, BROWN, LIGHTGRAY, DARKGRAY,
   LIGHTBLUE, LIGHTGREEN, LIGHTCYAN, LIGHTRED, LIGHTMAGENTA, YELLOW, WHITE.
```

Pero hay que tener una precaución: en MsDos sólo se pueden usar como colores de fondo los 7 primeros: desde BLACK hasta LIGHTGRAY. Se podía evitar en los ordenadores más modernos, a cambio de perder la posibilidad de que el texto parpadee, pero es un detalle en el que no entraremos. El caso es que "normalmente" si hacemos algo como

```
textbackground(LIGHTBLUE);
```

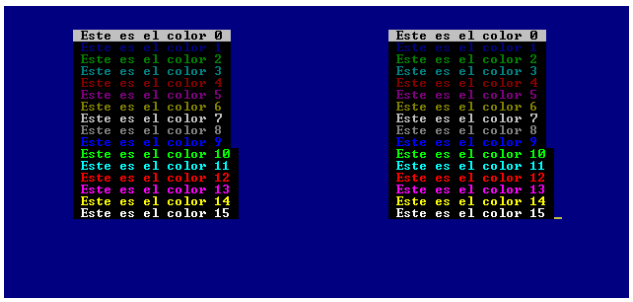
no obtendremos los resultados esperados, sino que será como si hubiésemos utilizado el color equivalente en el rango de 0 a 7:

```
textbackground(BLUE);
```

Para usarlas, tenemos que incluir "**conio.h**". Vamos a ver un ejemplo que emplee la mayoría de ellas:

```
/*---------------------------*/
/*  Ejemplo en C nº 89:      */
/*  c089.c                   */
/*                           */
/*  Pantalla y teclado       */
/*  con Turbo C              */
/*                           */
/*  Curso de C,              */
/*    Nacho Cabanes          */
/*---------------------------*/

#include <conio.h>      /* Para funciones de pantalla */

int main()
{
    int i,j;                /* Para los bucles "for" */
```

```
    textbackground(BLUE);           /* Fondo de la pantalla en azul */
    clrscr();                       /* Borro la pantalla */
    for(i=0; i<=1; i++)             /* Dos columnas */
        for(j=0; j<=15; j++)        /* Los 16 colores */
        {
            gotoxy(10+ 40*i , 3+j);             /* Coloco el cursor */
            textcolor(j);                       /* Elijo el color */
            if (j == 0)                         /* Si el color es 0 (negro) */
                textbackground(LIGHTGRAY);      /*   dejo fondo gris */
            else                                /* Si es otro color */
                textbackground(BLACK);          /*    dejo fondo negro */
            cprintf(" Este es el color %d ",j); /* Escribo en color */
        }
    getch();    /* Final: espero que se pulse una tecla, sin mostrarla */
    return 0;
}
```




El resultado sería el que se muestra a la derecha.

Tenemos más funciones definidas en "conio.h", que nos permiten saber en qué posición de la pantalla estamos, definir "ventanas" para trabajar sólo con una zona de la pantalla, etc. Pero como el trabajo en modo texto se considera cada vez más anticuado, y especialmente dentro del entorno Windows, no profundizaremos más.

**Ejercicios propuestos**:

- (10.3.1) Crea un menú para MsDos que muestre varias opciones en el centro de la pantalla, y el reloj en la parte superior derecha de la pantalla. Mientras el usuario no pulse una tecla, el reloj debe actualizarse continuamente.

## *10.4. Acceso a pantalla en Linux: curses.h*

En Linux, hay una biblioteca llamada "curses", que es la que deberemos incluir si queremos usar algo más que el simple "printf": nos permitirá borrar la pantalla, escribir en unas ciertas coordenadas, cambiar colores o apariencias del texto, dibujar recuadros, etc.

Eso sí, el manejo es más complicado que en el caso de Turbo C: hay que tener en cuenta que en Linux (y en los Linux en general) podemos encontrarnos con que nuestro programa se use desde un terminal antiguo, que no permita colores, sino sólo negrita y subrayado, o que no actualice la pantalla continuamente, sino sólo en ciertos momentos. Es algo cada vez menos frecuente, pero si queremos que nuestro programa funciones siempre, deberemos llevar ciertas precauciones.

Por eso, el primer paso será activar el acceso a pantalla con "**initscr**" y terminar con "**endwin**". Además, cuando queramos asegurarnos de que la información aparezca en pantalla, deberíamos usar "**refresh**".

Como primeras órdenes, vamos a usar:

- "clear" para borrar la pantalla.
- "move" para desplazarnos a ciertas coordenadas.
- "printw" para escribir en pantalla.

Con todo esto, un primer ejemplo sería:

```
/*--------------------------*/
/*  Ejemplo en C nº 90:     */
/*  c090.c                  */
/*                          */
/*  Pantalla con Curses (1) */
/*                          */
/*  Curso de C,             */
/*    Nacho Cabanes         */
/*--------------------------*/

#include <curses.h>

int main()
{
    initscr();
    clear();
    move(10,10);
    printw("hola");
    refresh();
    getch();
    endwin();
    return 0;
}
```

Un detalle importante: a la hora de **compilar** hay que añadir la opción de que enlace la librería "ncurses", bien desde las opciones de nuestro entorno de desarrollo (si usamos Ajuta, KDevelop o algún otro), o bien desde la línea de comandos:

```
cc ejemplo.c –lncurses –o ejemplo
```

(hay que recordar que se distinguen las mayúsculas de las minúsculas).

Podemos mejorarlo nuestro ejemplo un poco:

- En la inicialización, es frecuente que nos interese comprobar el teclado cada vez que se pulse una tecla, sin esperar a que se complete con Intro, y de asegurarse de eso se encarga la orden "cbreak".
- También es frecuente que queramos que cuando se pulse una tecla, ésta no aparezca en pantalla. Eso lo hacemos con "noecho".

Ya en el cuerpo del programa, también podemos seguir haciendo mejoras:

- Podemos desplazarnos a una cierta posición y escribir un texto, todo ello con la misma orden, si usamos "mvaddstr".
- Podemos cambiar la apariencia del texto si le aplicamos ciertos atributos, usando la orden "attron". Los atributos más habituales en un terminal Unix serían la negrita (A_BOLD) y el subrayado (A_UNDERLINE). Estos atributos se desactivarían con "attroff".

Con todo esto, ahora tendríamos algo como

```
/*---------------------------*/
/*  Ejemplo en C nº 91:      */
/*  c091.c                   */
/*                           */
/*  Pantalla con Curses (2)  */
/*                           */
/*  Curso de C,              */
/*    Nacho Cabanes          */
/*---------------------------*/

#include <curses.h>

int main()
{
    initscr();              /* Inicialización */
    cbreak();               /* Sin esperar a Intro en el teclado */
    noecho();               /* Para que getch no escriba en pantalla */
    clear();                /* Borramos la pantalla */
    attron(A_BOLD);         /* Activamos la negrita (bold) */
    mvaddstr(2,10,"hola");  /* Escribimos en cierto sitio */
    attroff(A_BOLD);        /* Desactivamos la negrita */
    refresh();              /* Actualizamos la info en pantalla */
    getch();                /* Esperamos a que se pulse una tecla */
    endwin();               /* Se acabó */
    return 0;
}
```

Finalmente, si queremos escribir en pantalla usando colores, tendremos que decir que queremos comenzar el modo de color con `start_color();` durante la inicialización. Eso sí, antes deberíamos comprobar con `has_colors()` si nuestro terminal puede manejar colores (tenemos definida una constante FALSE para poder comprobarlo).

Entonces podríamos definir nuestros pares de color de fondo y de primer plano, con "init_pair", así: `init_pair(1, COLOR_CYAN, COLOR_BLACK);` (nuestro par de colores 1 será texto azul claro sobre fondo negro).

Para usar estos colores, sería muy parecido a lo que ya conocemos: `attron(COLOR_PAIR(1));`

Y si queremos comprobar las teclas extendidas, haríamos `keypad(stdscr, TRUE);` durante la inicialización y en cuerpo del programa ya podríamos usar órdenes como

```
tecla = getch();
```

```
if (tecla == KEY_LEFT) ...
```

Vamos a ver un ejemplo que junte todo esto:

```
/*--------------------------*/
/*  Ejemplo en C nº 92:     */
/*  c092.c                  */
/*                          */
/*  Pantalla con Curses (3) */
/*                          */
/*  Curso de C,             */
/*    Nacho Cabanes         */
/*--------------------------*/

#include <curses.h>

int main()
{
    int tecla;

    initscr();
    if(has_colors() == FALSE)
    {
        endwin();
        printf("Su terminal no permite usar colores!\n");
        exit(1);
    }
    start_color();
    cbreak();
    noecho();
    keypad(stdscr, TRUE);

    clear();
    init_pair(1, COLOR_CYAN, COLOR_BLACK);
    attron(COLOR_PAIR(1));
    mvaddstr(2,10,"hola");
    attroff(COLOR_PAIR(1));
    refresh();
    tecla = getch();
    if (tecla == KEY_LEFT)
        printw("Has pulsado izquierda");
    getch();
    endwin();
    return 0;
}
```

Al igual que ocurría con Turbo C, esto no es todo lo que hay. También podemos crear ventanas, definir colores "a medida" a partir de su cantidad de rojo, verde y azul, podemos leer lo que hay en la pantalla, podemos manejar el ratón, e incluso tenemos rutinas para manejo de paneles y de formularios, pero todo ello queda fuera del cometido de este apartado, que es puramente introductorio.

**Ejercicios propuestos**:

- (10.4.1) Crea, tanto para MsDos como para Linux, un "protector de pantalla" que muestre tu nombre rebotando en los laterales de la pantalla. Deberá avanzar de posición cada segundo.

## *10.5. Juegos multiplataforma: SDL*

### 10.5.1. Dibujando una imagen de fondo y un personaje

Existen distintas bibliotecas que permiten crear gráficos desde el lenguaje C. Unas son específicas para un sistema, y otras están diseñadas para ser portables de un sistema a otro. Por otra parte, unas se centran en las funciones básicas de dibujo (líneas, círculos, rectángulos, etc), y otras se orientan más a la representación de imágenes que ya existan como fichero.

Nosotros veremos las nociones básicas del uso de SDL, que es una de las bibliotecas más adecuadas para crear juegos, porque es multiplataforma (existe para Windows, Linux y otros sistemas) y porque esta orientada a la manipulación de imágenes, que es algo más frecuente en juegos que el dibujar líneas o polígonos.

No veremos detalles de su instalación, porque en los sistemas Linux debería bastar un instalar el paquete SDL-Dev (que normalmente tendrá un nombre como libsdl1.2-dev), y en Windows hay entornos que permiten crear un "proyecto de SDL" tan sólo con dos clics, como CodeBlocks.

Vamos a ver algunas de las operaciones básicas, y un primer ejemplo:

Para poder utilizar SDL, debemos incluir SDL.h, así

```
#include <SDL/SDL.h>
```

Ya dentro del cuerpo del programa, el primer paso sería tratar de inicializar la biblioteca SDL, y abandonar el programa en caso de no conseguirlo:

```
if (SDL_Init(SDL_INIT_VIDEO) < 0)
{
    printf("No se pudo inicializar SDL: %s\n", SDL_GetError());
    exit(1);
}
```

Al terminar nuestro programa, deberíamos llamar a SQL_Quit:

```
SDL_Quit();
```

Para escoger modo de pantalla de 800x600 puntos, con 16 bits de color haríamos

```
screen = SDL_SetVideoMode( 800, 600, 16, SDL_HWSURFACE );
if( screen == NULL )
{
    printf( "Error al entrar a modo grafico: %s\n", SDL_GetError() );
    SDL_Quit();
```

```
    return -1;
}
```

Podemos cambiar simplemente el texto (caption) de la ventana:

```
SDL_WM_SetCaption( "Prueba 1 de SDL", "Prueba 1 de SDL" );
```

Para mostrar una imagen en pantalla: deberemos declararla del tipo SDL_Surface, cargarla con SDL_LoadBMP, y volcarla con SDL_BlitSurface usando como dato auxiliar la posición de destino, que será de tipo SDL_Rect:

```
SDL_Surface *protagonista;
protagonista = SDL_LoadBMP("protag.bmp");
SDL_Rect destino;
destino.x=400;
destino.y=300;
SDL_BlitSurface(protagonista, NULL, screen, &destino);
```

Con SDL_Flip hacemos toda la imagen visible:

```
SDL_Flip(screen);
```

Y para esperar 5 segundos y que nos dé tiempo a comprobar que todo ha funcionado, utilizaríamos:

```
SDL_Delay( 5000 );
```

Todo esto, junto en un programa, quedaría:

```
/*---------------------------*/
/*  Ejemplo en C son SDL     */
/*  sdl01.c                  */
/*                           */
/*  Ejemplo de SDL (1)       */
/*                           */
/*  Curso de C,              */
/*    Nacho Cabanes          */
/*---------------------------*/

#include <stdio.h>
#include <stdlib.h>
#include <SDL/SDL.h>

int main()
{
    SDL_Surface *screen;
    SDL_Surface *fondo;
    SDL_Surface *protagonista;
    SDL_Rect destino;
    int i, j;

    /* Tratamos de inicializar la biblioteca SDL */
    if (SDL_Init(SDL_INIT_VIDEO) < 0)
    {
```

```
        printf("No se pudo inicializar SDL: %s\n", SDL_GetError());
        exit(1);
    }

    /* Preparamos las imagenes a mostrar */
    fondo = SDL_LoadBMP("fondo.bmp");
    protagonista = SDL_LoadBMP("protag.bmp");

    /* Si todo ha ido bien, hacemos algo:
        entrar a modo grafico y cambiar el título de la ventana */
    screen = SDL_SetVideoMode( 800, 600, 16, SDL_HWSURFACE );
    if( screen == NULL )
    {
        printf( "Error al entrar a modo grafico: %s\n", SDL_GetError() );
        SDL_Quit();
        return -1;
    }

    /* Titulo de la ventana */
    SDL_WM_SetCaption( "Prueba 1 de SDL", "Prueba 1 de SDL" );

    /* Dibujamos la imagen de fondo */
    destino.x=0;
    destino.y=0;
    SDL_BlitSurface(fondo, NULL, screen, &destino);

    /* Dibujamos el protagonista */
    destino.x=400;
    destino.y=300;
    SDL_BlitSurface(protagonista, NULL, screen, &destino);

    /* Actualizamos la pantalla */
    SDL_Flip(screen);

    /* Y esperamos antes de salir */
    SDL_Delay( 5000 );

    /* Finalmente, preparamos para salir */
    SDL_Quit();
    return 0;
}
```

En principio, si sólo usamos SDL, las imágenes tendrán que ser en formato BMP, pero hay otras bibliotecas adicionales, como SDL_Image, que permiten mostrar también imágenes en formatos PNG, JPG, etc.

El tipo SDL_Rect representa un rectángulo, y se trata de un registro (struct), que tiene como campos:

* x: posición horizontal
* y: posición vertical
* w: anchura (width)
* h: altura (height)

(nosotros no hemos usado la anchura ni la altura, pero podrían ser útiles si no queremos volcar toda la imagen, sino sólo parte de ella).

Para **compilar** este fuente en Linux deberíamos teclear lo siguiente:

```
cc -o sdl01 sdl01.c  `sdl-config --cflags --libs`
```

Hay cosas que ya conocemos: el compilador (cc), el nombre para el ejecutable (-o sdl01) y el nombre del fuente (sdl01.c). Las opciones adicionales debemos indicarlas tal y como aparecen, entre acentos graves (acentos "hacia atrás"): `sdl-config --cflags --libs`

En Windows (con entornos como CodeBlocks), bastaría con pulsar el botón "compilar el proyecto" de nuestro entorno de desarrollo.

**Ejercicios propuestos**:

- (10.5.1.1) Crea (o busca en Internet) una imagen de fondo de tamaño 800x600 que represente un cielo negro con estrellas, y tres imágenes de planetas con un tamaño menor (cercano a 100x100, por ejemplo). Entra a modo gráfico 800x600 con 24 bits de color usando SDL, dibuja la imagen de fondo, y sobre ella, las de los tres planetas. El título de la ventana deberá ser "Planetas". Las imágenes deberán mostrarse durante 7 segundos.

### 10.5.2. Un personaje móvil

Para poder mover a ese protagonista por encima del fondo de forma "no planificada" necesitamos poder comprobar qué teclas se están pulsando. Lo podemos conseguir haciendo:

```
teclas = SDL_GetKeyState(NULL);

if (teclas[SDLK_UP])
{
    posicionY -= 2;
}
```

Donde la variable "teclas", será un array de "unsigned int". La forma normal de declararla será:

```
Uint8* teclas;
```

Eso sí, antes de poner acceder al estado de cada tecla, deberemos poner en marcha todo el sistema de comprobación de sucesos ("events", en inglés). Al menos deberemos comprobar si hay alguna petición de abandonar el programa (por ejemplo, pulsando la X de la ventana), a lo que corresponde el suceso de tipo SDL_QUIT. De paso, podríamos comprobar en este mismo paso si se ha pulsado la tecla ESC, que es otra forma razonable de indicar que queremos terminar el programa:

```
while (SDL_PollEvent(&suceso))
{
    if (suceso.type == SDL_QUIT)    terminado = 1;
    if (suceso.type == SDL_KEYDOWN)
        if (suceso.key.keysym.sym == SDLK_ESCAPE)  terminado = 1;
}
```

donde la variable suceso se declararía con

```
SDL_Event suceso;
```

**Ejercicios propuestos**:

- (10.5.2.1) Amplía el ejercicio anterior, añadiendo la imagen de una nave espacial, que deberá moverse cada vez que el usuario pulse una de las flechas del cursor. Ya no se saldrá al cabo de varios segundos, sino cuando el usuario pulse la tecla ESC.

### 10.5.3. Imágenes transparentes, escribir texto y otras mejoras

Hemos visto cómo dibujar imágenes en pantalla, y cómo comprobar qué teclas se han pulsado, para hacer que una imagen se mueva sobre la otra.

Pero tenía un defecto, que hacía que no quedara vistoso: si la imagen del "protagonista" tiene un recuadro negro alrededor, al moverse tapará parte del fondo. Esto se debe evitar: una imagen que se mueve en pantalla (lo que se suele llamar un "sprite") debería tener zonas **transparentes**, a través de las que se vea el fondo.

Es algo fácil de conseguir con SDL: podemos hacer que un color se considere transparente, usando

```
SDL_SetColorKey(surface, SDL_SRCCOLORKEY,
   SDL_MapRGB(surface->format, r, g, b));
```

donde "surface" es la superficie que queremos que tenga un color transparente (por ejemplo, nuestro protagonista), y "r, g, b" son las componentes roja, verde y azul del color que queremos que se considere transparente (si es el color negro el que queremos que no sea vea, serían 0,0,0).

Así, con nuestro protagonista haríamos

```
/* Preparamos las imagenes a mostrar */
fondo = SDL_LoadBMP("fondo.bmp");
protagonista = SDL_LoadBMP("protag.bmp");
/* El protagonista debe tener contorno transparente */
SDL_SetColorKey(protagonista, SDL_SRCCOLORKEY,
    SDL_MapRGB(protagonista->format, 0, 0, 0));
```

Si usamos la biblioteca adicional SDL_Image, podremos usar imágenes PNG con transparencia, y entonces no necesitaríamos usar esta orden.

Escribir **texto** con SDL no es algo "trivial": A no ser que empleemos la biblioteca adicional SDL_TTF, no tendremos funciones predefinidas que muestren una frase en ciertas coordenadas de la pantalla.

Pero podríamos hacerlo "a mano": preparar una imagen que tenga las letras que queremos mostrar (o una imagen para cada letra), y tratarlas como si fueran imágenes. Podemos crearnos nuestras propias funciones para escribir cualquier texto. Podríamos comenzar por una función "escribirLetra(int x, int y, char letra)", y apoyándonos en ella, crear otra "escribirFrase(int x, int y, char *frase)"

Si queremos que nuestro juego funcione a **pantalla completa**, los cambios son mínimos: basta añadir SDL_FULLSCREEN a la lista de parámetros que indicamos al escoger el modo de pantalla. Estos parámetros se deben indicar separados por una barra vertical (|), porque entre ellos se va a realizar una operación OR (suma lógica) a nivel de bit:

```
screen = SDL_SetVideoMode( 800, 600, 16, SDL_FULLSCREEN | SDL_HWSURFACE );
```

**Ejercicios propuestos**:

- (10.5.3.1) Amplía el ejercicio anterior, para que la imagen de la nave espacial tenga el contorno transparente (y quizá también alguna zona interior).
- (10.5.3.2) Crea una imagen que contenga varias letras (al menos H, O, L, A), y úsala para escribir las frases HOLA y OH en modo gráfico con SDL.

### 10.5.4. El doble buffer

Si intentamos mover varias imágenes a la vez en pantalla, es probable que el resultado parpadee.

El motivo es que mandamos información a la pantalla en distintos instantes, por lo que es fácil que alguno de todos esos bloques de información llegue en un momento que no coincida con el barrido de la pantalla (el movimiento del haz de electrones que redibuja la información que vemos).

Una solución habitual es preparar toda la información, trozo a trozo, en una "imagen oculta", y sólo volcar a la pantalla visible cuando la imagen está totalmente preparada. Esta técnica es la que se conoce como "usar un doble buffer".

El segundo paso es sincronizar con el barrido, algo que en la mayoría de bibliotecas hace una función llamada "retrace" o "sync", y que en SDL se hace automáticamente cuando volcamos la información con "SDL_Flip".

Ya en la práctica, en SDL, comenzaremos por añadir el parámetro correspondiente (SDL_DOUBLEBUF) cuando entramos a modo gráfico:

```
screen=SDL_SetVideoMode(800,600,16,SDL_HWSURFACE|SDL_DOUBLEBUF);
```

A la hora de dibujar, no lo hacemos directamente sobre "screen", sino sobre una superficie ("surface") auxiliar. Cuando toda esta superficie está lista, es cuando la volcamos a la pantalla, así:

```
SDL_BlitSurface(fondo, NULL, pantallaOculta, &destino);
SDL_BlitSurface(protagonista, NULL, pantallaOculta, &destino);
...
SDL_BlitSurface(pantallaOculta, NULL, screen, &destino);
SDL_Flip(screen);
```

Sólo queda un detalle: ¿cómo reservamos espacio para esa pantalla oculta?

Si la pantalla oculta es del mismo tamaño que nuestro fondo o que alguna otra imagen, nos puede bastar con cargar la imagen :

```
fondo = SDL_LoadBMP("fondo.bmp");
```

Si no es el caso (por ejemplo, porque el fondo se forme repitiendo varias imágenes de pequeño tamaño), podemos usar "SDL_CreateRGBSurface", que reserva el espacio para una superficie de un cierto tamaño y con una cierta cantidad de colores, así:

```
pantallaOculta = SDL_CreateRGBSurface(SDL_SWSURFACE, 640, 480, 16,
   0,0,0,0);
```

(el parámetro SDL_SWSURFACE indica que no se trabaje en memoria física de la tarjeta, sino en memoria del sistema; 640x480 es el tamaño de la superficie; 16 es la cantidad de color -16bpp = 655356 colores-; los otros 0,0,0,0 se refieren a la cantidad de rojo, verde, azul y transparencia -alpha- de la imagen).

**Ejercicios propuestos**:

- (10.5.4.1) Amplía el ejercicio de la nave, para que emplee un doble buffer que permita evitar parpadeos.

### 10.5.5. El bucle de juego (game loop)

Para abordar un juego completo, es frecuente no tener claro cómo debería ser la estructura del juego: qué repetir y cuando.

Pues bien, en un juego típico encontraríamos:

- Cosas que no se repiten, como la inicialización para entrar a modo gráfico, o la liberación de recursos al final del programa.
- Cosas que no deberían repetirse, como la lectura de las imágenes que vamos a usar (si no son muy grandes, bastará con leerlas al principio y memorizarlas).

- Cosas que sí deberán repetirse, como dibujar el fondo, comprobar si el usuario ha pulsado alguna tecla, mover los elementos de la pantalla si procede, etc.

Esta parte que se repite es lo que se suele llamar el "bucle de juego" (en inglés, "game loop"). Su apariencia exacta depende de cada juego, pero en la mayoría podríamos encontrar algo parecido a:

```
Inicializar
Mientras (partida en marcha)
    Comprobar sucesos (teclas / ratón / joystick)
    Dibujar fondo (en pantalla oculta)
    Actualizar posición de personajes (en pantalla oculta)
    Comprobar colisiones y otras situaciones del juego
    Corregir personajes según la situación
    Mostrar pantalla oculta
    Atender a situaciones especiales (una vida menos, etc)
Fin Mientras
Liberar recursos
```

El orden no tiene por qué ser exactamente éste, pero habitualmente será parecido. Vamos a detallarlo un poco más:

- Al principio del programa, toda la inicialización, que no se repite.
- Mientras la partida esté en marcha, una de las cosas que haremos es comprobar si hay que atender a alguna orden del usuario, que haya pulsado alguna tecla o utilizado su ratón o su joystick/gamepad para indicar alguna acción del juego.
- Otra de las cosas que hay que hacer siempre es "componer" la imagen del juego (fondo + personajes), generalmente en pantalla oculta, para evitar parpadeos.
- Normalmente, antes de dibujar (quizá incluso antes del paso anterior) deberemos comprobar si ha habido alguna colisión o alguna otra situación que atender. Por ejemplo, si un disparo enemigo ha chocado con nuestra nave, no deberíamos dibujar la nave, sino una explosión. O quizá nuestro protagonista recoja un objeto y cambie su apariencia.
- Cuando ya queda claro cómo debe ser la pantalla, la dibujaremos.
- También hay situaciones especiales que se pueden dar durante la partida, y que pueden suponer interrupciones del bucle normal de juego, como cuando el usuario pulsa la tecla de pausa, o pide ayuda, o pierde una vida.
- Finalmente, hay que recordar que cuando acaba la partida, en casi cualquier biblioteca de funciones que usemos deberemos liberar los recursos que hemos usado (en SDL, con SDL_Quit).

**Ejercicios propuestos**:

- (10.5.5.1) Amplía el ejercicio de la nave, para que emplee tenga una función "buclePrincipal", que siga la apariencia de un bucle de juego clásico, llamando a funciones

con los nombres "comprobarTeclas", "dibujarElementos", "moverEstrellas", "mostrarPantallaOculta". Existirán algunas estrellas, cuya posición cambiará cuando se llame a "moverEstrellas" y que se dibujarán, junto con los demás componentes, al llamar a en "dibujarElementos". Al final de cada "pasada" por el bucle principal habrá una pausa de 20 milisegundos, para que la velocidad del "juego" no dependa del ordenador en que se ejecute.

- (10.5.5.2) Ampliar el ejercicio anterior, para que también exista un "enemigo" que se mueva de lado a lado de la pantalla.
- (10.5.5.3) Ampliar el ejercicio anterior, para que haya 20 "enemigos" (un array) que se muevan de lado a lado de la pantalla.
- (10.5.5.4) Ampliar el ejercicio anterior, para añadir la posibilidad de que nuestro personaje "dispare".
- (10.5.5.5) Ampliar el ejercicio anterior, para que si un "disparo" toca a un "enemigo", éste deje de dibujarse.
- (10.5.5.6) Ampliar el ejercicio anterior, para que la partida termine si un "enemigo" toca a nustro personaje.

# 11. Otras características avanzadas de C

## *11.1 Operaciones con bits*

Podemos hacer desde C operaciones entre bits de dos números (AND, OR, XOR, etc). Vamos primero a ver qué significa cada una de esas operaciones.

| Operación | Qué hace | En C | Ejemplo |
|---|---|---|---|
| Complemento (not) | Cambiar 0 por 1 y viceversa | ~ | ~1100 = 0011 |
| Producto lógico (and) | 1 sólo si los 2 bits son 1 | & | 1101 & 1011 = 1001 |
| Suma lógica (or) | 1 si uno de los bits es 1 | \| | 1101 \| 1011 = 1111 |
| Suma exclusiva (xor) | 1 sólo si los 2 bits son distintos | ^ | 1101 ^ 1011 = 0110 |
| Desplazamiento a la izquierda | Desplaza y rellena con ceros | << | 1101 << 2 = 110100 |
| Desplazamiento a la derecha | Desplaza y rellena con ceros | >> | 1101 >> 2 = 0011 |

Ahora vamos a aplicarlo a un ejemplo completo en C:

```
/*----------------------------*/
/*  Ejemplo en C nº 93:       */
/*  c093.c                    */
/*                            */
/*  Operaciones de bits en    */
/*  números enteros           */
/*                            */
/*  Curso de C,               */
/*    Nacho Cabanes           */
/*----------------------------*/

#include <stdio.h>

int main()
{
    int a = 67;
    int b = 33;

    printf("La variable a vale %d\n", a);
    printf("y b vale %d\n", b);
    printf("  El complemento de a es: %d\n", ~a);
    printf("  El producto lógico de a y b es: %d\n", a&b);
    printf("  Su suma lógica es: %d\n", a|b);
    printf("  Su suma lógica exclusiva es: %d\n", a^b);
    printf("  Desplacemos a a la izquierda: %d\n", a << 1);
    printf("  Desplacemos a a la derecha: %d\n", a >> 1);
    return 0;
}
```

La respuesta que nos da Dev-C++ 4.9.9.2 es la siguiente:

```
La variable a vale 67
y b vale 33
  El complemento de a es: -68
```

```
El producto lógico de a y b es: 1
Su suma lógica es: 99
Su suma lógica exclusiva es: 98
Desplacemos a a la izquierda: 134
Desplacemos a a la derecha: 33
```

Para comprobar que es correcto, podemos convertir al sistema binario esos dos números y seguir las operaciones paso a paso:

```
67 = 0100 0011
33 = 0010 0001
```

En primer lugar complementamos "a", cambiando los ceros por unos:

```
1011 1100 = -68
```

Después hacemos el producto lógico de A y B, multiplicando cada bit, de modo que 1*1 = 1, 1*0 = 0, 0*0 = 0

```
0000 0001 = 1
```

Después hacemos su suma lógica, sumando cada bit, de modo que 1+1 = 1, 1+0 = 1, 0+0 = 0

```
0110 0011 = 99
```

La suma lógica exclusiva devuelve un 1 cuando los dos bits son distintos: 1^1 = 0, 1^0 = 1, 0^0 = 0

```
0110 0010 = 98
```

Desplazar los bits una posición a la izquierda es como multiplicar por dos:

```
1000 0110 = 134
```

Desplazar los bits una posición a la derecha es como dividir entre dos:

```
0010 0001 = 33
```

¿Qué utilidades puede tener todo esto? Posiblemente, más de las que parece a primera vista. Por ejemplo: desplazar a la izquierda es una forma muy rápida de multiplicar por potencias de dos; desplazar a la derecha es dividir por potencias de dos; la suma lógica exclusiva (xor) es un método rápido y sencillo de cifrar mensajes; el producto lógico nos permite obligar a que ciertos bits sean 0 (algo que se puede usar para comprobar máscaras de red); la suma lógica, por el contrario, puede servir para obligar a que ciertos bits sean 1...

Un último comentario: igual que hacíamos operaciones abreviadas como

```
x += 2;
```

también podremos hacer cosas como

```
x <<= 2;
x &= 2;
x |= 2;
...
```

## *11.2 Directivas del preprocesador*

Desde el principio hemos estado manejando cosas como

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

Y aquí hay que comentar bastante más de lo que parece. Ese "include" no es una orden del lenguaje C, sino una orden directa al compilador (una "**directiva**"). Realmente es una orden a una cierta parte del compilador que se llama "preprocesador". Estas directivas indican una serie de pasos que se deben dar antes de empezar realmente a traducir nuestro programa fuente.

Aunque "include" es la directiva que ya conocemos, vamos a comenzar por otra más sencilla, y que nos resultará útil cuando lleguemos a ésta.

### 11.2.1. Constantes simbólicas: #define

La directiva "define" permite crear "constantes simbólicas". Podemos crear una constante haciendo

```
#define MAXINTENTOS 10
```

y en nuestro programa lo usaríamos como si se tratara de cualquier variable o de cualquier valor numérico:

```
if (intentoActual >= MAXINTENTOS) ...
```

El primer paso que hace nuestro compilador es reemplazar esa "falsa constante" por su valor, de modo que la orden que realmente va a analizar es

```
if (intentoActual >= 10) ...
```

pero a cambio nosotros tenemos el valor numérico sólo al principio del programa, por lo que es muy fácil de modificar, mucho más que si tuviéramos que revisar el programa entero buscando dónde aparece ese 10.

Comparado con las constantes "de verdad", que ya habíamos manejado (`const int MAXINTENTOS=10;`), las constantes simbólicas tienen la ventaja de que no son variables, por lo que no se les reserva memoria adicional y las comparaciones y demás operaciones suelen ser más rápidas que en el caso de un variable.

Vamos a ver un ejemplo completo, que pida varios números y muestre su suma y su media:

```
/*---------------------------*/
/*  Ejemplo en C nº 94:      */
/*  c094.c                   */
/*                           */
/*  Ejemplo de #define       */
/*                           */
/*  Curso de C,              */
/*    Nacho Cabanes          */
/*---------------------------*/

#include <stdio.h>

#define CANTIDADNUMEROS 5

int main()
{
    int numero[CANTIDADNUMEROS];
    int suma=0;
    int i;

    for (i=0; i<CANTIDADNUMEROS; i++)
    {
        printf("Introduzca el dato número %d: ", i+1);
        scanf("%d", &numero[i]);
    }

    for (i=0; i<CANTIDADNUMEROS; i++)
        suma += numero[i];
    printf("Su suma es %d\n", suma);
    printf("Su media es %4.2f\n", (float) suma/CANTIDADNUMEROS);

    return 0;
}
```

Las constantes simbólicas se suelen escribir en mayúsculas por convenio, para que sean más fáciles de localizar. De hecho, hemos manejado hasta ahora muchas constantes simbólicas sin saberlo. La que más hemos empleado ha sido NULL, que es lo mismo que un 0, está declarada así:

```
#define NULL 0
```

Pero también en casos como la pantalla en modo texto con Turbo C aparecían otras constantes simbólicas, como éstas

```
#define BLUE 1
#define YELLOW 14
```

Y al acceder a ficheros teníamos otras constantes simbólicas como SEEK_SET (0), SEEK_CUR (1), SEEK_END (2).

A "define" también se le puede dar también un uso más avanzado: se puede crear "macros", que en vez de limitarse a dar un valor a una variable pueden comportarse como pequeñas órdenes, más rápidas que una función. Un ejemplo podría ser:

```
#define SUMA(x,y) x+y
```

Vamos a crear un fuente completo que defina y utilice una macro sencilla:

```
/*--------------------------*/
/*  Ejemplo en C nº 95:     */
/*  c095.c                  */
/*                          */
/*  Ejemplo de #define (2)  */
/*  macro básica            */
/*                          */
/*  Curso de C,             */
/*    Nacho Cabanes         */
/*--------------------------*/

#include <stdio.h>

#define SUMA(x,y) x+y

int main()
{
   int n1, n2;

   printf("Introduzca el primer dato: ");
   scanf("%d", &n1);

   printf("Introduzca el segundo dato: ");
   scanf("%d", &n2);

   printf("Su suma es %d\n", SUMA(n1,n2));
   return 0;
}
```

### 11.2.2 Inclusión de ficheros: #include

Ya nos habíamos encontrado con esta directiva. Lo que hace es que cuando llega el momento de que nuestro compilador compruebe la sintaxis de nuestro fuente en C, ya no existe ese "include", sino que en su lugar el compilador ya ha insertado los ficheros que le hemos indicado.

¿Y eso de por qué se escribe <stdio.h>, entre < y >?  No es la única forma de usar #include. Podemos encontrar líneas como

```
#include <stdlib.h>
```

y como

```
#include "misdatos.h"
```

El primer caso es un fichero de cabecera **estándar** del compilador. Lo indicamos entre < y > y así el compilador sabe que tiene que buscarlo en su directorio (carpeta) de "includes". El segundo caso es un fichero de cabecera que hemos creado **nosotros**, por lo que lo indicamos entre comillas, y así el compilador sabe que no debe buscarlo entre sus directorios, sino en el mismo directorio en el que está nuestro programa.

Vamos a ver un ejemplo: declararemos una función "suma" dentro de un fichero ".h" y lo incluiremos en nuestro fuente para poder utilizar esa función "suma" sin volver a definirla. El fichero de cabecera sería así:

```
/*--------------------------*/
/*  Ejemplo en C nº 96 (a):  */
/*  c096.h                  */
/*                          */
/*  Ejemplo de #include     */
/*                          */
/*  Curso de C,             */
/*    Nacho Cabanes         */
/*--------------------------*/

int suma(int x,int y)
{
  return x+y;
}
```

(Nota: si somos puristas, esto no es correcto del todo. Un fichero de cabecera no debería contener los detalles de las funciones, sólo su "cabecera", lo que habíamos llamado el "prototipo", y la implementación de la función debería estar en otro fichero, pero eso lo haremos dentro de poco).

Un fuente que utilizara este fichero de cabecera podría ser

```
/*--------------------------*/
/*  Ejemplo en C nº 96 (b):  */
/*  c096.c                  */
/*                          */
/*  Ejemplo de #include (2) */
/*                          */
/*  Curso de C,             */
/*    Nacho Cabanes         */
/*--------------------------*/

#include <stdio.h>
#include "c096.h"

int main()
{
   int n1, n2;

   printf("Introduzca el primer dato: ");
   scanf("%d", &n1);

   printf("Introduzca el segundo dato: ");
```

```
   scanf("%d", &n2);

   printf("Su suma es %d\n", suma(n1,n2));
   return 0;
}
```

### 11.2.3. Compilación condicional: #ifdef, #endif

Hemos utilizado #define para crear constantes simbólicas. Desde dentro de nuestro fuente, podemos comprobar si está definida una constante, tanto si ha sido creada por nosotros como si la ha creado el sistema.

Cuando queramos que nuestro fuente funcione en varios sistemas distintos, podemos hacer que se ejecuten unas órdenes u otras según de qué compilador se trate, empleando #ifdef (o #if) al principio del bloque y #endif al final. Por ejemplo,

```
#ifdef _GCC_
  instanteFinal = clock () + segundos * CLOCKS_PER_SEC ;
#endif
#ifdef _TURBOC_
  instanteFinal = clock () + segundos * CLK_TCK ;
#endif
```

Los programas que utilicen esta idea podrían compilar sin ningún cambio en distintos sistemas operativos y/ distintos compiladores, y así nosotros esquivaríamos las incompatibilidades que pudieran existir entre ellos (a cambio, necesitamos saber qué constantes simbólicas define cada sistema).

Esta misma idea se puede aplicar a nuestros programas. Uno de los usos más frecuentes es hacer que ciertas partes del programa se pongan en funcionamiento durante la fase de depuración, pero no cuando el programa esté terminado.

Vamos a mejorar el ejemplo 94, para que nos muestre el valor temporal de la suma y nos ayude a descubrir errores:

```
/*---------------------------*/
/*  Ejemplo en C nº 97:      */
/*  c097.c                   */
/*                           */
/*  Compilacion condicional  */
/*  con #define              */
/*                           */
/*  Curso de C,              */
/*    Nacho Cabanes          */
/*---------------------------*/

#include <stdio.h>

#define CANTIDADNUMEROS 5
#define DEPURANDO 1

int main()
```

```
{
    int numero[CANTIDADNUMEROS];
    int suma;   /* Aqui hay un error */
    int i;

    for (i=0; i<CANTIDADNUMEROS; i++)
    {
        printf("Introduzca el dato número %d: ", i+1);
        scanf("%d", &numero[i]);
    }

    for (i=0; i<CANTIDADNUMEROS; i++)
    {
        #ifdef DEPURANDO
        printf("Valor actual de la suma: %d\n", suma);
        #endif
        suma += numero[i];
    }
    printf("Su suma es %d\n", suma);
    printf("Su media es %4.2f\n", (float) suma/CANTIDADNUMEROS);

    return 0;
}
```

Este fuente tiene intencionadamente un error: no hemos dado un valor inicial a la suma, con lo que contendrá basura, y obtendremos un resultado incorrecto.

El resultado de nuestro programa sería

```
Introduzca el dato numero 1: 2
Introduzca el dato numero 2: 3
Introduzca el dato numero 3: 4
Introduzca el dato numero 4: 5
Introduzca el dato numero 5: 7
Valor actual de la suma: 2009055971
Valor actual de la suma: 2009055973
Valor actual de la suma: 2009055976
Valor actual de la suma: 2009055980
Valor actual de la suma: 2009055985
Su suma es 2009055992
Su media es 401811198.40
```

Vemos que ya en la primera pasada, el valor de la suma no es 2, sino algo que parece absurdo, así que falta el valor inicial de la suma, que debería ser "int suma=0;". Cuando acaba la fase de depuración, basta con eliminar la frase `#define DEPURANDO 1` (no es necesario borrarla, podemos dejarla comentada para que no haga efecto), de modo que el fuente corregido sería:

```
/*--------------------------*/
/*  Ejemplo en C nº 97b:    */
/*  c097b.c                 */
/*                          */
/*  Compilacion condicional */
/*  con #define (2)         */
/*                          */
```

```
/*  Curso de C,              */
/*    Nacho Cabanes          */
/*---------------------------*/

#include <stdio.h>

#define CANTIDADNUMEROS 5
/*#define DEPURANDO 1*/

int main()
{
    int numero[CANTIDADNUMEROS];
    int suma=0;   /* Error corregido */
    int i;

    for (i=0; i<CANTIDADNUMEROS; i++)
    {
        printf("Introduzca el dato número %d: ", i+1);
        scanf("%d", &numero[i]);
    }

    for (i=0; i<CANTIDADNUMEROS; i++)
    {
        #ifdef DEPURANDO
        printf("Valor actual de la suma: %d\n", suma);
        #endif
        suma += numero[i];
    }
    printf("Su suma es %d\n", suma);
    printf("Su media es %4.2f\n", (float) suma/CANTIDADNUMEROS);

    return 0;
}
```

También tenemos otra alternativa: en vez de comentar la línea de #define, podemos anular la definición con la directiva #undef:

```
#undef DEPURANDO
```

Por otra parte, también tenemos una directiva #ifndef para indicar qué hacer si no está definida una contante simbólico, y un #else, al igual que en los "if" normales, para indicar qué hacer si no se cumple la condición, y una directiva #elif (abreviatura de "else if"), por si queremos encadenar varias condiciones.

### 11.2.4. Otras directivas

Las que hemos comentado son las directivas más habituales, pero también existen otras. Una de las que son frecuentes (pero menos estándar que las anteriores) es #pragma, que permite indicar opciones avanzadas específicas de cada compilador.

No veremos detalles de su uso en ningún compilador concreto, pero sí un ejemplo de las cosas que podemos encontrar su manejamos fuentes creados por otros programadores:

```
#if __SC__ || __RCC__
```

```
#pragma once
#endif

#ifndef RC_INVOKED
#pragma pack(__DEFALIGN)
#endif
```

## *11.3. Programas a partir de varios fuentes*

### 11.3.1. Creación desde la línea de comandos

Es bastante frecuente que un programa no esté formado por un único fuente, sino por varios. Puede ser por hacer el programa más modular debido a su complejidad, por reparto de trabajo entre varias personas, etc.

En cualquier caso, la gran mayoría de los compiladores de C serán capaces de juntar varios fuentes independientes y crear un único ejecutable a partir de todos ellos. Vamos a ver cómo conseguirlo.

Empezaremos por el caso más sencillo: supondremos que tenemos un programa principal, y otros dos módulos auxiliares en los que hay algunas funciones que se usarán desde el programa principal.

Por ejemplo, el primer módulo (UNO.C) podría ser simplemente:

```
/*--------------------------*/
/*  Ejemplo en C nº 98a:    */
/*  uno.c                   */
/*                          */
/*  Programa a partir de    */
/*  varios fuentes (1)      */
/*                          */
/*  Curso de C,             */
/*    Nacho Cabanes         */
/*--------------------------*/

void uno()
{
     printf("Función uno\n");
}
```

y el segundo módulo (DOS.C):

```
/*--------------------------*/
/*  Ejemplo en C nº 98b:    */
/*  dos.c                   */
/*                          */
/*  Programa a partir de    */
/*  varios fuentes (2)      */
/*                          */
/*  Curso de C,             */
/*    Nacho Cabanes         */
/*--------------------------*/
```

```
void dos(int numero)
{
   printf("Función dos, con el parámetro %d\n", numero);
}
```

Un programa principal simple, que los utilizase, sería (TRES.C):

```
/*--------------------------*/
/*  Ejemplo en C nº 98c:    */
/*  tres.c                  */
/*                          */
/*  Programa a partir de    */
/*  varios fuentes (3)      */
/*                          */
/*  Curso de C,             */
/*    Nacho Cabanes         */
/*--------------------------*/

#include <stdio.h>

int main()
{
   printf("Estamos en el cuerpo del programa.\n");
   uno();
   dos(3);
   return 0;
}
```

Para compilar los tres fuentes y crear un único ejecutable, desde la mayoría de los compiladores de Dos o Windows bastaría con acceder a la línea de comandos, y teclear el nombre del compilador seguido por los de los tres fuentes:

```
TCC UNO DOS TRES
```

(TCC es el nombre del compilador, en el caso de Turbo C y de Turbo C++; sería BCC para el caso de Borland C++, SCC para Symantec C++, etc.).

Entonces el compilador convierte los tres ficheros fuente (.C) a ficheros objeto (.OBJ), los enlaza y crea un único ejecutable, que se llamaría UNO.EXE (porque UNO es el nombre del primer fuente que hemos indicado al compilador), y cuya salida en pantalla sería:

```
Estamos en el cuerpo del programa.
Función uno
Función dos, con el parámetro 3
```

En el caso de GCC para Linux (o de alguna de sus versiones para Windows, como MinGW, o DevC++, que se basa en él), tendremos que indicarle el nombre de los ficheros de entrada (con extensión) y el nombre del fichero de salida, con la opción "-o":

```
gcc uno.c dos.c tres.c  -o resultado
```

Pero puede haber compiladores en los que la situación no sea tan sencilla. Puede ocurrir que al compilar el programa principal, que era:

```
int main()
{
   printf("Estamos en el cuerpo del programa.\n");
   uno();
   dos(3);
   return 0;
}
```

el compilador nos dé un mensaje de error, diciendo que no conoce las funciones "uno()" y "dos()". No debería ser nuestro caso, si al compilar le hemos indicado los fuentes en el orden correcto (TCC UNO DOS TRES), pero puede ocurrir si se los indicamos en otro orden, o bien si tenemos muchos fuentes, que dependan los unos de los otros.

La forma de evitarlo sería indicándole que esas funciones existen, y que ya le llegarán más tarde los detalles en concreto sobre cómo funcionan.

Para decirle que existen, lo que haríamos sería incluir en el programa principal los **prototipos** de las funciones (las cabeceras, sin el desarrollo) que se encuentran en otros módulos, así:

```
/*--------------------------*/
/*  Ejemplo en C nº 98d:    */
/*  tresB.c                 */
/*                          */
/*  Programa a partir de    */
/*  varios fuentes (3b)     */
/*                          */
/*  Curso de C,             */
/*    Nacho Cabanes         */
/*--------------------------*/

#include <stdio.h>

void uno();              /* Prototipos de las funciones externas */
void dos(int numero);

int main()               /* Cuerpo del programa */
{
   printf("Estamos en el cuerpo del programa.\n");
   uno();
   dos(3);
   return 0;
}
```

Esta misma solución de poner los prototipos al principio del programa nos puede servir para casos en los que, teniendo un único fuente, queramos declarar el cuerpo del programa antes que las funciones auxiliares:

```
/*--------------------------*/
/*  Ejemplo en C nº 99:     */
/*  c099.c                  */
/*                          */
/*  Funciones después de    */
/*  "main",usando prototipos */
/*                          */
/*  Curso de C,             */
/*    Nacho Cabanes         */
/*--------------------------*/

#include <stdio.h>

void uno();              /* Prototipos de las funciones */
void dos(int numero);

int main()               /* Cuerpo del programa */
{
   printf("Estamos en el cuerpo del programa.\n");
   uno();
   dos(3);
   return 0;
}

void uno()
{
   printf("Función uno\n");
}

void dos(int numero)
{
   printf("Función dos, con el parámetro %d\n", numero);
}
```

En ciertos compiladores puede que tengamos problemas con este programa si no incluimos los prototipos al principio, porque en "main()" se encuentra la llamada a "uno()", que no hemos declarado. Al poner los prototipos antes, el compilador ya sabe qué tipo de función es "uno()" (sin parámetros, no devuelve ningún valor, etc.), y que los datos concretos los encontrará más adelante.

De hecho, si quitamos esas dos líneas, este programa no compila en Turbo C++ 1.01 ni en Symantec C++ 6.0, porque cuando se encuentra en "main()" la llamada a "uno()", da por supuesto que va a ser una función de tipo "int". Como después le decimos que es "void", protesta. (En cambio, GCC, que suele ser más exigente, en este caso se limita a avisarnos, pero compila el programa sin problemas).

La solución habitual en estos casos en que hay que declarar prototipos de funciones (especialmente cuando se trata de funciones compartidas por varios fuentes) suele ser agrupar estos fuentes en "ficheros de cabecera". Por ejemplo, podríamos crear un fichero llamado EJEMPLO.H que contuviese:

```
/*  EJEMPLO.H */
```

```
void uno();              /* Prototipos de las funciones */
void dos(int numero);
```

y en el fuente principal escribiríamos:

```
/*--------------------------*/
/*  Ejemplo en C nº 98e:    */
/*  tresC.c                 */
/*                          */
/*  Programa a partir de    */
/*  varios fuentes (3c)     */
/*                          */
/*  Curso de C,             */
/*    Nacho Cabanes         */
/*--------------------------*/

#include <stdio.h>
#include "ejemplo.h"

int main()
{
    printf("Estamos en el cuerpo del programa.\n");
    uno();
    dos(3);
    return 0;
}
```

Aquí es importante recordar la diferencia en la forma de indicar los dos ficheros de cabecera:

<stdio.h> Se indica entre corchetes angulares porque el fichero de cabecera es propio del compilador (el ordenador lo buscará en los directorios del compilador).

"ejemplo.h" Se indica entre comillas porque el fichero H es nuestro (el ordenador lo buscará en el mismo directorio que están nuestros fuentes).

Finalmente, conviene hacer una consideración: si varios fuentes distintos necesitaran acceder a EJEMPLO.H, deberíamos **evitar** que este fichero se incluyese varias veces. Esto se suele conseguir definiendo una variable simbólica la primera vez que se enlaza, de modo que podamos comprobar a partir de entonces si dicha variable está definida, con #ifdef, así:

```
/* EJEMPLO.H mejorado */

#ifndef _EJEMPLO_H
#define _EJEMPLO_H

void uno();              /* Prototipos de las funciones */
void dos(int numero);

#endif
```

### 11.3.2. Introducción al uso de la herramienta Make

Make es una herramienta que muchos compiladores incorporan y que nos puede ser de utilidad cuando se trata de proyectos de un cierto tamaño, o al menos creados a partir de bastantes fuentes.

Su uso normal consiste simplemente en teclear MAKE. Entonces esta herramienta buscará su fichero de configuración, un fichero de texto que deberá llamarse "makefile" (podemos darle otro nombre; ya veremos cómo). Este fichero de configuración le indica las dependencias entre ficheros, es decir, qué ficheros es necesario utilizar para crear un determinado "objetivo". Esto permite que no se recompile todos y cada uno de los fuentes si no es estrictamente necesario, sino sólo aquellos que se han modificado desde la última compilación.

En general, el contenido del fichero "makefile" será algo parecido a esto:

```
objetivo: dependencias
    órdenes
```

(En la primera línea se escribe el objetivo, seguido por dos puntos y por la lista de dependencias; en la segunda línea se escribe la orden que hay que dar en caso de que sea necesario recompilar, precedida por un tabulador). Si queremos añadir algún comentario, basta con precederlos con el símbolo #.

Vamos a crear un ejecutable llamado PRUEBA.EXE a partir de cuatro ficheros fuente llamados UNO.C, DOS.C, TRES.C, CUATRO.C, usando Turbo C++.

Así, nuestro primer "makefile" podría ser un fichero que contuviese el siguiente texto:

```
PRUEBA.EXE: UNO.C DOS.C TRES.C CUATRO.C
    TCC -ePRUEBA.EXE UNO.C DOS.C TRES.C CUATRO.C
```

Es decir: nuestro objetivo es conseguir un fichero llamado PRUEBA.EXE, que queremos crear a partir de varios ficheros llamados UNO.C, DOS.C, TRES.C y CUATRO.C. La orden que queremos dar es la que aparece en la segunda línea, y que permite, mediante el compilador TCC, crear un ejecutable llamado PRUEBA.EXE a partir de cuatro fuentes con los nombres anteriores. (La opción "-e" de Turbo C++ permite indicar el nombre que queremos que tenga el ejecutable; si no, se llamaría UNO.EXE, porque tomaría su nombre del primero de los fuentes).

¿Para qué nos sirve esto? De momento, nos permite **ahorrar tiempo**: cada vez que tecleamos MAKE, se lee el fichero MAKEFILE y se compara la fecha (y hora) del objetivo con la de las dependencias; si el fichero objetivo no existe o si es más antiguo que alguna de las dependencias, se realiza la orden que aparece en la segunda línea (de modo que evitamos escribirla completa cada vez).

En nuestro caso, cada vez que tecleemos MAKE, ocurrirá una de estas tres posibilidades

- Si no existe el fichero PRUEBA.EXE, se crea uno nuevo utilizando la orden de la segunda línea.
- Si ya existe y es más reciente que los cuatro fuentes, no se recompila ni se hace nada, todo queda como está.
- Si ya existe, pero se ha modificado alguno de los fuentes, se recompilará de nuevo para obtener un fichero PRUEBA.EXE actualizado.

Eso sí, estamos dando por supuesto varias cosas "casi evidentes":

- Que tenemos la herramienta MAKE y está accesible (en el directorio actual o en el PATH).
- Que hemos creado el fichero MAKEFILE.
- Que existen los cuatro ficheros fuente UNO.C, DOS.C, TRES.C y CUATRO.C.
- Que existe el compilador TCC y está accesible (en el directorio actual o en el PATH).

Vayamos mejorando este MAKEFILE rudimentario. La primera mejora es que si la lista de dependencias no cabe en una única linea, podemos partirla en dos, empleando la barra invertida ("\"):

```
PRUEBA.EXE: UNO.C DOS.C \       #Objetivo y dependencias
    TRES.C CUATRO.C                     # Mas dependencias
    TCC -ePRUEBA.EXE UNO.C DOS.C TRES.C CUATRO.C    #Orden a dar
```

Al crear el MAKEFILE habíamos ganado en "velocidad de tecleo" y en que no se recompilase todo nuevamente si no se ha modificado nada. Pero en cuanto un fuente se modifica, nuestro MAKEFILE recompila todos otra vez, aunque los demás no hayan cambiado. Esto podemos mejorarlo añadiendo un paso intermedio (la creación cada fichero objeto OBJ) y más objetivos (cada fichero OBJ, a partir de cada fichero fuente), así:

```
# Creacion del fichero ejecutable

prueba.exe: uno.obj dos.obj tres.obj
    tcc -eprueba.exe uno.obj dos.obj tres.obj

# Creacion de los ficheros objeto

uno.obj: uno.c
    tcc -c uno.c

dos.obj: dos.c
    tcc -c dos.c

tres.obj: tres.c
    tcc -c tres.c
```

Estamos detallando los pasos que normalmente se dan al compilar, y que muchos compiladores realizan en una única etapa, sin que nosotros nos demos cuenta: primero se convierten los ficheros fuente (ficheros con extensión C) a código máquina (código objeto, ficheros con extensión OBJ) y finalmente los ficheros objeto se enlazan entre sí (y con las bibliotecas propias del lenguaje) para dar lugar al programa ejecutable (en MsDos y Windows normalmente serán ficheros con extensión EXE).

Así conseguimos que cuando modifiquemos un único fuente, se recompile sólo este (y no todos los demás, que pueden ser muchos) y después se pase directamente al proceso de enlazado, con lo que se puede ganar mucho en velocidad si los cambios que hemos hecho al fuente son pequeños.

(Nota: la opción "-c" de Turbo C++ es la que le indica que sólo compile los ficheros de C a OBJ, pero sin enlazarlos después).

Si tenemos varios MAKEFILE distintos (por ejemplo, cada uno para un compilador diferente, o para versiones ligeramente distintas de nuestro programa), nos interesará poder utilizar nombres distintos.

Esto se consigue con la opción "-f" de la orden MAKE, por ejemplo si tecleamos

```
MAKE -fPRUEBA
```

la herramienta MAKE buscaría un fichero de configuración llamado PRUEBA o bien PRUEBA.MAK.

Podemos mejorar más aún estos ficheros de configuración.  Por ejemplo, si precedemos la orden por @, dicha orden no aparecerá escrita en pantalla

```
PRUEBA.EXE: UNO.C DOS.C TRES.C CUATRO.C
    @TCC -ePRUEBA.EXE UNO.C DOS.C TRES.C CUATRO.C
```

Y si precedemos la orden por & , se repetirá para los ficheros indicados como "dependencias". Hay que usarlo en conjunción con la macro $**, que hace referencia a todos los ficheros dependientes, o $?, que se refiere a los ficheros que se hayan modificado después que el objetivo.

```
copiaSeguridad: uno.c dos.c tres.c
    &copy $** a:\fuentes
```

Una última consideración: podemos crear nuestras propias macros, con la intención de que nuestro MAKEFILE resulte más fácil de leer y de mantener, de modo que una versión más legible de nuestro primer fichero sería:

```
FUENTES = uno.c dos.c tres.c
COMPIL = tcc

prueba.exe: $(FUENTES)
   $(COMPIL) -eprueba.exe $(FUENTES)
```

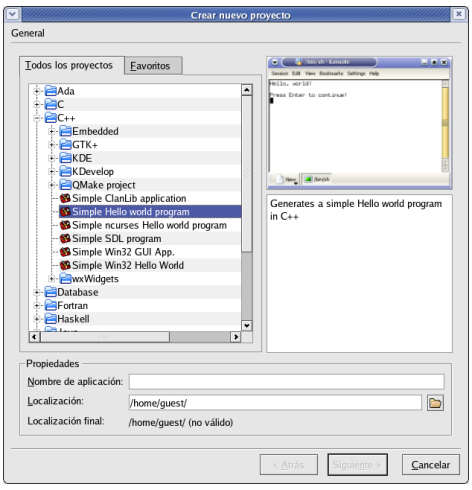
Es decir, las macros se definen poniendo su nombre, el signo igual y su definición, y se emplean precediéndolas de $ y encerrándolas entre paréntesis.

Pero todavía hay más que no hemos visto. Las herramientas MAKE suelen permitir otras posibilidades, como la comprobación de condiciones (con las directivas "!if", "!else" y similares) o la realización de operaciones (con los operadores estándar de C: +, *, %, >>, etc). Quien quiera profundizar en estos y otros detalles, puede recurrir al manual de la herramienta MAKE que incorpore su compilador.

¿Alguna diferencia en Linux? Pocas. Sólo hay que recordar que en los sistemas Unix se distingue entra mayúsculas y minúsculas, por lo que la herramienta se llama "make", y el fichero de datos "Makefile" o "makefile" (preferible la primera nomenclatura, con la primera letra en mayúsculas). De igual modo, el nombre del compilador y los de los fuentes se deben escribir dentro del "Makefile" exactamente como se hayan creado (habitualmente en minúsculas).

### 11.3.3. Introducción a los "proyectos"

En la gran mayoría de los compiladores que incorporan un "entorno de desarrollo", existe la posibilidad de conseguir algo parecido a lo que hace la herramienta MAKE, pero desde el propio entorno. Es lo que se conoce como "crear un proyecto".

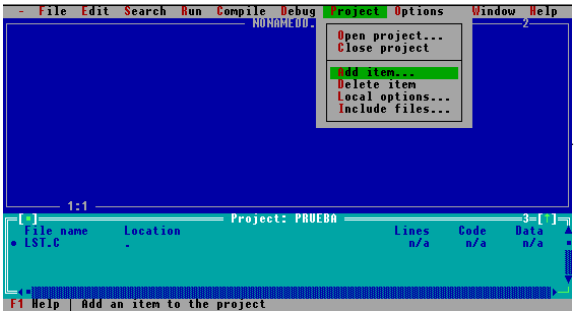



Se suele poder hacer desde desde un menú llamado "Proyecto", donde existirá una opción "Nuevo proyecto" (en inglés Project / New Project), o a veces desde el menú Archivo.

En muchas ocasiones, tendremos varios tipos de proyectos disponibles, gracias a que algún asistente deje el esqueleto del programa preparado para nosotros.

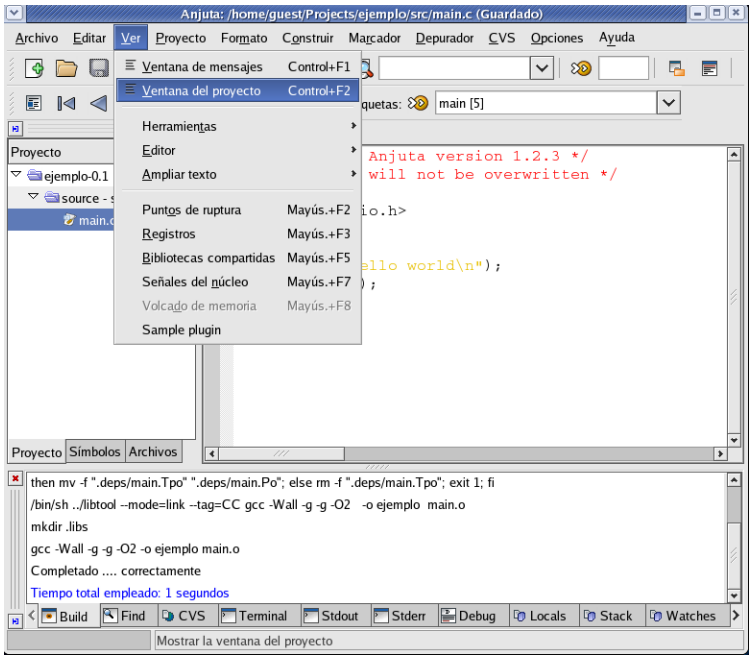
Desde esta primera ventana también le daremos ya un nombre al proyecto (será el nombre que tendrá el fichero ejecutable), y también desde aquí podemos añadir los fuentes que formarán parte del proyecto, si los tuviéramos creados desde antes (suele ser algo como "Añadir fichero", o en inglés "Add Item").

En una cierta ventana de la pantalla tendremos información sobre los fuentes que componen



nuestro proyecto (en el ejemplo, tomado de "Turbo C++ Second Edition", esta ventana aparece en la parte inferior de la pantalla).

En otros entornos, como Anjuta o KDevelop, esta información aparece en la parte izquierda de la pantalla:




Las ventajas que obtenemos al usar "proyectos" son:

- La posibilidad de manipular varios fuentes a la vez y de recompilar un programa que esté formado por todos ellos, sin necesidad de salir del entorno de desarrollo.
- La ventaja añadida de que el gestor de proyectos sólo recompilará aquellos fuentes que realmente se han modificado, como comentábamos que hacía la herramienta MAKE.

## *11.4 Uniones y campos de bits*

Conocemos lo que es un struct: un dato formado por varios "trozos" de información de distinto tipo. Pero C también tiene dos tipos especiales de "struct", de manejo más avanzado. Son las uniones y los campos de bits.

Una **unión** recuerda a un "struct" normal, con la diferencia de que sus "campos" comparten el mismo espacio de memoria:

```
union
{
    char letra;          /* 1 byte */
    int numero;          /* 4 bytes */
} ejemplo;
```

En este caso, la variable "ejemplo" ocupa 4 bytes en memoria (suponiendo que estemos trabajando en un compilador de 32 bits, como lo son la mayoría de los de Windows y Linux). El

primer byte está compartido por "letra" y por "numero", y los tres últimos bytes sólo pertenecen a "numero".

Si hacemos

```
ejemplo.numero = 25;
ejemplo.letra  = 50;
printf("%d", ejemplo.numero);
```

Veremos que "ejemplo.numero" ya no vale 25, puesto que al modificar "ejemplo.letra" estamos cambiando su primer byte. Ahora "ejemplo.numero" valdría 50 o un número mucho más grande, según si el ordenador que estamos utilizando almacena en primer lugar el byte más significativo o el menos significativo.

Un **campo de bits** es un elemento de un registro (struct), que se define basándose en su tamaño en bits. Se define de forma muy parecida (pero no igual) a un "struct" normal, indicando el número de bits que se debe reservar a cada elemento:

```
struct  campo_de_bits
{
   int bit_1        : 1;
   int bits_2_a_5   : 4;
   int bit_6        : 1;
   int bits_7_a_16  : 10;
} variableDeBits;
```

Esta variable ocuparía 1+4+1+10 = 16 bits (2 bytes). Los campos de bits pueden ser interesantes cuando queramos optimizar al máximo el espacio ocupado por nuestro datos.

## *11.5. El operador coma*

Cuando vimos la orden "for", siempre usábamos una única variable como contador, pero esto no tiene por qué ser siempre así. Vamos a verlo con un ejemplo:

```
/*---------------------------*/
/*  Ejemplo en C nº 100:     */
/*  c100.c                   */
/*                           */
/*  Operador coma (1)        */
/*                           */
/*  Curso de C,              */
/*    Nacho Cabanes          */
/*---------------------------*/

#include <stdio.h>

int main()
{
    int i, j;

    for (i=0,j=1; i<=5, j<=30; i++, j+=2)
```

```
        printf("i vale %d y j vale %d.\n", i, j);

    return 0;
}
```

Vamos a ver qué hace este "for":

- Los valores iniciales son i=0, j=1.
- Se repetirá mientras que i <= 5, j <= 30.
- Al final de cada paso, i aumenta en una unidad, y j en dos unidades.

El único problema está en saber cúando terminará el bucle: si se parará en cuanto se cumpla una de las dos condiciones o si tendrán que cumplirse las dos.

El resultado de este programa servirá como respuesta:

```
i vale 0 y j vale 1.
i vale 1 y j vale 3.
i vale 2 y j vale 5.
i vale 3 y j vale 7.
i vale 4 y j vale 9.
i vale 5 y j vale 11.
i vale 6 y j vale 13.
i vale 7 y j vale 15.
i vale 8 y j vale 17.
i vale 9 y j vale 19.
i vale 10 y j vale 21.
i vale 11 y j vale 23.
i vale 12 y j vale 25.
i vale 13 y j vale 27.
i vale 14 y j vale 29.
```

Como podemos observar, llega un momento en que deja de cumplise que i<=5, pero el programa sigue avanzando: no se sale del bucle "for" hasta que se cumplen **las dos condiciones** (realmente, hasta que se cumple la segunda).

La idea es que, en general, si se usa el operador coma para separar dos expresiones, nuestro compilador evalúa la primera expresión, luego la segunda, y devuelve como valor el resultado de la segunda. Veámoslo con un segundo ejemplo algo más rebuscado:

```
/*----------------------------*/
/*  Ejemplo en C nº 101:      */
/*  c101.c                    */
/*                            */
/*  Operador coma (2)         */
/*                            */
/*  Curso de C,               */
/*    Nacho Cabanes           */
/*----------------------------*/

#include <stdio.h>
```

```
int main()
{
    int i, j, k;
    k = ((i=5), (j=6));

    printf("i=%d, j=%d, k=%d", i, j, k);
    return 0;
}
```

Aun así, en la práctica, el único uso habitual del operador coma es el primero que hemos visto: utilizar dos condiciones simultáneas para controlar un bucle "for".

## *11.6. Enumeraciones*

Cuando tenemos varias constantes, cuyos valores son números enteros, y especialmente si son números enteros consecutivos, tenemos una forma abreviada de definirlos. Se trata de **enumerarlos**:

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

(Al igual que las constantes de cualquier otro tipo, se suele escribir en mayúsculas para recordar en cualquier parte que se sepa "de un vistazo" que son constantes, no variables)

La primera constante valdrá 0, y las demás irán aumentando de una en una, de modo que en nuestro caso valen:

```
LUNES = 0, MARTES = 1, MIERCOLES = 2, JUEVES = 3, VIERNES = 4,
SABADO = 5, DOMINGO = 6
```

Si queremos que los valores no sean exactamente estos, podemos dar valor a cualquiera de las contantes, y las siguientes irán aumentando de uno en uno. Por ejemplo, si escribimos

```
enum diasSemana { LUNES=1, MARTES, MIERCOLES, JUEVES=6, VIERNES,
  SABADO=10, DOMINGO };
```

Ahora sus valores son:

```
LUNES = 1, MARTES = 2, MIERCOLES = 3, JUEVES = 6, VIERNES = 7,
SABADO = 10, DOMINGO = 11
```

## *11.7. Definición de tipos*

El tipo de una variable nos indica el rango de valores que puede tomar. Tenemos creados para nosotros los tipos básicos, pero puede que nos interese crear nuestros propios tipos de variables, para lo que usamos "**typedef**". El formato es

```
typedef tipo nombre;
```

Lo que hacemos es darle un nuevo nombre a un tipo de datos. Puede ser cómodo para hacer más legible nuestros programas. Por ejemplo, alguien que conozca Pascal o Java, puede echar en falta un tipo de datos "boolean", que permita hacer comparaciones un poco más legibles, siguiendo esta estructura:

```
  if (encontrado == FALSE) ...
```

o bien como

```
  if (datosCorrectos == TRUE) ...
```

```
/*--------------------------*/
/*  Ejemplo en C nº 102:    */
/*  c102.c                  */
/*                          */
/*  Definición de tipos (1) */
/*                          */
/*  Curso de C,             */
/*    Nacho Cabanes         */
/*--------------------------*/

#include <stdio.h>

#define VALIDA 711          /* Clave correcta */
#define TRUE 1              /* Nostalgia de los boolean */
#define FALSE 0

typedef int boolean;        /* Definimos un par de tipos */
typedef int integer;

integer clave;              /* Y dos variables */
boolean acertado;

int main()
{
    do
    {
        printf("Introduzca su clave numérica: ");
        scanf("%d", &clave);
        acertado = (clave == VALIDA);

        if (acertado == FALSE)
            printf("No válida!\n");
    }
    while (acertado != TRUE);

    printf("Aceptada.\n");
    return 0;
}
```

También podemos usar "typedef" para dar un nombre corto a todo un struct:

```c
/*--------------------------*/
/*  Ejemplo en C nº 103:    */
/*  c103.c                  */
/*                          */
/*  Definición de tipos (2) */
/*                          */
/*  Curso de C,             */
/*    Nacho Cabanes         */
/*--------------------------*/

#include <stdio.h>

typedef struct          /* Defino el tipo "datos" */
{
    int valor;
    float coste;
    char ref;
} datos;

datos ejemplo;           /* Y creo una variable de ese tipo */

int main()
{
    ejemplo.valor = 34;
    ejemplo.coste = 1200.5;
    ejemplo.ref = 'A';
    printf("El valor es %d", ejemplo.valor);
    return 0;
}
```

# *Apéndice 1. Revisiones de este texto*

- 0.95, de 11-Ago-13. Ampliados los apartados 1.1, 1.4 y 1.6 para hablar de ciertas peculiaridades de C99 y para dar más detalles sobre cómo probar los fuentes con Linux y Windows. Ampliado el apartado 3.2.7, para incluir la sintaxis "if (3==x)". Ampliado el apartado 5.2.1, para hablar de los posibles problemas al leer tanto texto como números en un mismo programa. Ampliado el 5.2.7 para mencionar atoi y atof. Ligeramente ampliado el apartado 7.5. Ligeramente ampliado el apartado 7.9 para hablar de la opción de compilación "-lm" para fuentes que usen funciones matemáticas. Corregida alguna errata menor: la solución al ejercicio 1.7.1b, el enunciado de 7.10a, el ejemplo 77 (9.6b). Revisada la sintaxis en colores, para usar el mismo criterio de colores en todo el texto. Ligeramente ampliado el tamaño del texto (10 a 11 puntos). Por homogeneidad, cambiadas las comillas tipográficas por comillas rectas en todo el texto. Incluidos muchos ejercicios propuestos adicionales (50). Numerados los ejercicios propuestos, para que sea más fácil resolver dudas en foros. Ampliada la cantidad de soluciones de ejemplo a los ejercicios propuestos, pero dichas soluciones ahora se encuentran en un documento independiente (212 páginas, 117 ejemplos, 195 ejercicios propuestos).
- 0.90, de 06-Ene-10. Añadido un apartado sobre SDL (10.5). Ligeramente ampliados los apartados 0.2, 1.1. Algún párrafo reescrito en 1.2. Ligeramente reescrito algún párrafo del tema 2, añadido un párrafo en 2.1.12, reescrito un párrafo en 4.1. Ampliado el contenido del apartado 4.4. Añadidos ejercicios propuestos adicionales en 3.1.3 (1), 3.1.4 (2), 3.1.5 (2), 3.1.8 (1), 3.1.9 (2), 3.1.10 (2), 3.2.1 (2), 3.2.3 (3), 3.7 (2), 4.1 (2), 4.2 (3), 4.3 (1), 4.4 (1), 5.1.1 (2), 5.1.2 (1), 5.1.4 (2), 5.2.3 (2), 5.2.4 (1), 5.2.5 (1), 5.2.6 (2), 5.3 (2), 5.5.1 (1), 5.5.2 (1), 5.5.3 (1), 5.6 (5), 5.7 (1), 6.1 (1), 6.2 (1), 6.4 (2), 6.6 (2), 6.8 (2), 6.10 (4), 7.4 (3), 7.5 (1), 7.6 (6), 7.10 (6). Incluidos ejemplos de posibles soluciones a los ejercicios propuestos en 1.2 (199 páginas, 115 ejemplos, 145 ejercicios propuestos -15 de ellos resueltos-).
- 0.23, de 11-Ene-07. Más detallado el apartado 5.2.7 y el 7.8. Añadido un ejemplo más a 7.3. Revisados muchos fuentes para usar variables locales, en vez de globales, siempre que fuera posible. En el tema 5.6 había dos ejemplos de agenda muy parecidos, eliminado uno de ellos. (174 páginas).
- 0.22, de 31-Oct-06. Añadidos al tema 3 varios ejercicios resueltos (14), relacionados con fallos frecuentes de los alumnos. Ligera revisión ortográfica. (175 páginas).
- 0.21, de 07-Jul-06. Incluye hasta el apartado 11.7 (172 páginas).
- 0.20, de 09-Jun-06. Incluye hasta el apartado 11.5 (169 páginas).
- 0.19, de 08-May-06. Incluye hasta el apartado 11.3.2 (160 páginas).
- 0.18, de 27-Abr-06. Incluye hasta el apartado 11.2.3 (152 páginas).
- 0.17, de 04-Abr-06. Completado el tema de "punteros y memoria dinámica" (tema 9) y el de "bibliotecas útiles" (tema 10) (145 páginas).
- 0.16, de 21-Mar-06. Ampliado el tema de "punteros y memoria dinámica", hasta el apartado 9.10 (132 páginas).
- 0.15, de 19-Mar-06. Añadido un tema sobre depuración (nuevo tema 8, el tema de "punteros" pasa a ser el tema 9). Ampliado el tema de "punteros y memoria dinámica" con un ejemplo básico, información sobre parámetros por valor y por referencia, aritmética de punteros y la equivalencia entre punteros y arrays. Ampliado el apartado de

"funciones" para hablar de el orden de las funciones (tema 7.8, renumerados los posteriores) (123 páginas).

- 0.14, de 16-Feb-06. Ampliado el apartado de "funciones" para hablar de números aleatorios y de funciones matemáticas. Ampliado el apartado de "punteros y memoria dinámica" con un primer ejemplo de su uso (114 páginas).
- 0.13, de 05-Feb-06. Ampliado el apartado de "funciones" y comenzado el apartado de "punteros y memoria dinámica". Añadido el apartado 6.13, con un ejemplo de lectura y escritura en un fichero. Añadidos ejercicios en los apartados 2.1.3, 2.1.7, 2.1.10, 2.2, 2.3, 3.1.5, 3.2.3... Corregida una errata en el formato de "fread" y "fwrite". (108 páginas, cambia la numeración a partir de la página 27 por la inclusión de esos ejercicios básicos).
- 0.12, de 25-Ene-06. Completado el capítulo de "ficheros" y comenzado el de "funciones" (hasta el apartado 7.6, 100 páginas).
- 0.11, de 11-Ene-06. Añadido un ejemplo de "array" para almacenar muchos registros. Más ejemplos sobre ficheros: cómo mostrar datos de una imagen BMP (89 páginas, cambia la numeración a partir de la página 79 por incluir el ejemplo de registros).
- 0.10, de 12-Dic-05. Más información sobre ficheros: binarios, lectura y escritura, acceso directo... (86 páginas).
- 0.09, de 30-Nov-05. Añadida información sobre "structs" y sobre lectura y escritura de ficheros de texto (tema 6.2). Corregidos alguno de los formatos de "scanf", que el procesador de textos había convertido a mayúsculas incorrectamente (82 páginas).
- 0.08, de 14-Nov-05. Incluye hasta el tema 5.4 (76 páginas).
- 0.07, de 30-Oct-05. Incluye hasta el tema 5.2.4 (70 páginas).
- 0.06, de 23-Oct-05. Incluye todo el tema 3 (60 páginas).
- 0.05, de 10-Oct-05. Incluye hasta el tema 3.3 (56 páginas).
- 0.04, de 04-Oct-05. Incluye hasta el tema 3.1 (49 páginas).
- 0.03, de 30-Sept-05. Incluye hasta el tema 2 (42 páginas).
- 0.02, de 21-Sept-05. Incluye el tema 0 y el tema 1.
- 0.01, de 18-Sept-05, como texto de apoyo para los alumnos de Fundamentos de Programación en el I.E.S. San Vicente. Incluye el tema 0.