C U R S O D E C

Versión hipertexto 1.54w
Nacho Cabanes, marzo de 1999.

Introducción:

Sobre el lector.

O bien escoja una de las siguientes posibilidades:

Temas básicos
Ampliaciones sobre los temas básicos
Fuentes de ejemplo
Introducción a C++

Otros apartados:

Notas sobre el curso.
Contactar con el autor
Compiladores empleados
Indice alfabético, de ejemplos, tablas o cambios

Sobre el lector.

Si ha llegado hasta aquí, eso quiere decir que ya tiene unas nociones básicas sobre cómo manejar el sistema de ayuda de Windows. Entonces no hay mucho que explicar, espero.

Pero sí quiero comentar las **diferencias** con el lector para MsDos, para quien haya manejado una versión anterior del curso:

En mi opinión, este sistema basado en la ayuda de Windows tiene apenas un par de **inconvenientes** de poca importancia: no existe la tecla Alt+E para volcar un tema en un fichero de texto, ni existe la tecla Alt+J para exportar un fuente o una tabla, ya con el nombre correcto.

Pero tiene varias **ventajas** claras: es muy fácil imitar los dos procesos anteriores empleando las facilidades para copiar u pegar entre programas de Windows, o imprimir directamente un tema o parte de él. Además, a mí me dará la posibilidad de incluir imágenes dentro del texto, o de permitir búsquedas según ciertas palabras claves.

Espero que consideren que mi elección al decantarme por este sistema haya sido la correcta. Para cualquier sugerencia, no dude en contactar conmigo.

N.

Volver al menú principal

CURSO DE C. TEMAS BASICOS.

Tema 0. Sobre el curso.
Tema 1. Generalidades.
Tema 2. Introd. a variables
Tema 3. Recapitulación.
Tema 4. Entrada/Salida básica.
Tema 5. Operaciones matemáticas.
Tema 6. Condiciones.
Tema 7. Bucles.
Tema 8. Constantes y tipos.
Tema 9. Funciones.
Tema 10. Punteros.
Tema 11. Ficheros.
Tema 12. Cadenas de texto.

(Indice más detallado de estos temas)

CURSO DE C. INDICE DE CONTENIDOS.

Tema 0. Sobre el curso. Cuenta cómo nació la idea del curso y poco más.

Tema 1. Generalidades. Una primera toma de contacto con el lenguaje C.

Tema 2. Introd. a variables Introducción a las variables: cómo se definen y se usan.

Tema 3. Recapitulación. Un poco más detalle sobre las "cosas raras" que se han pasado un poco por alto en el tema anterior, como el #include.

Tema 4. Entrada/Salida básica. Volvemos a "printf" con algo más de detalle, y vemos cómo aceptar la introducción de datos por parte del usuario.

Tema 5. Operaciones matemáticas. Cómo hacer las operaciones matemáticas más habituales (sumar, restar, multiplicar, dividir, etc.).

Tema 6. Condiciones. Formas de comprobar condiciones: "si ocurre esto, haz aquello; en caso contrario haz esto otro".

Tema 7. Bucles. Para permitir que una parte de un programa se repita un cierto número de veces, o hasta que se cumpla una condición, o mientras que se cumpla una condición.

Tema 8. Constantes y tipos. Mejoremos eso de las variables: qué ocurre cuando realmente no va a variar o cuando no nos basta con los tipos de datos que existen en el C.

Tema 9. Funciones. Cómo crear nuestras propias rutinas, para que los programas sean más fáciles de leer y de modificar/ampliar.

Tema 10. Punteros. Cómo reservar memoria dinámicamente (sólo en el momento en que el programa la necesite)

Tema 11. Ficheros. Para acceder a ficheros: como leer su contenido, grabar en ellos o crear nuestros propios ficheros.

Tema 12. Cadenas de texto. Las cadenas de texto son más difíciles de manejar en C que en otros lenguajes como Pascal o Basic. Por eso, les dedicamos un tema aparte.

CURSO DE C. INDICE GLOBAL.

Indice alfabético
Lista de ejemplos
Lista de tablas

Cambios entre versiones

Conseguir la última versión del curso
La versión ampliada del curso

Volver al menú principal

CURSO DE C. TEMA 0: SOBRE EL CURSO.

Este curso nace como **ampliación** de un curso de Pascal que impartí en la conferencia Pascal_E de Fidonet entre Enero y Agosto de 1995 (los que tengan dicho curso podrán comprobar que sigo prácticamente la misma estructura).

Por ello, supondré que se entienden términos como compilador, editor, linker, etc., y que se tienen unos conocimientos básicos de programación, a fin de poder avanzar más rápido.

En cuanto a estos conocimientos básicos, daré por supuesto también que son conocimientos de lenguaje **Pascal** (Turbo Pascal), que es en el que me apoyaré para la mayoría de las comparaciones. Aun así, no debería ser difícil seguirlo si se ha programado en otros lenguajes, como Basic (especialmente si es alguno "moderno", como QBasic).

Este programa se distribuye "tal cual", sin garantía de ningún tipo, implícita ni explícita. Aun así, mi intención es que resulte útil, así que le rogaría que me comunique cualquier error que encuentre.

(Si desea profundizar más, también existe una versión ampliada de este curso).

Para cualquier sugerencia, no dude en contactar conmigo.

N.

Pasar al tema 1
Volver al índice de temas básicos
Volver al menú principal

Contactar conmigo es fácil:

La mejor forma es por correo electrónico, en Internet, en la dirección:

ncabanes@arrakis.es

Si no tiene módem, puede escribirme por correo ordinario (si se trata de alguna consulta, por favor incluya **sobre y sellos** para la respuesta) a mi apartado de correos:

Nacho Cabanes
Apartado de Correos 5234
03080 Alicante
(España)

CURSO DE C. TEMA 1.

Generalidades del C

Nota: Para que todo esto no resulte demasiado pesado, voy a procurar ir lo más directo al grano que me sea posible, intentando que haya poca teoría y mucha práctica. Para ello, supondré que ya se tienen unos ciertos conocimientos de programación (me apoyaré principalmente en el lenguaje Pascal, como he mencionado en el apartado anterior.

Como a programar se aprende programando, voy a tratar de enfocar este curso al revés de como lo hace la mayoría de los libros: en vez de dar primero toda la carga teórica y después aplicarlo, voy a ir poniendo **ejemplos**, y a continuación la explicación.

Así que empezamos. Ahí va uno:

```
/*-------------------------*/
/*   Primer ejemplo en C:  */
/*   EJ01.C                */
/*                         */
/*   Programa elemental    */
/*                         */
/*   Comprobado con:       */
/*      - Turbo C++ 1.01   */
/*      - Djgpp 2.01       */
/*      - Symantec C++ 6.0 */
/*      - GCC 2.6.3        */
/*      - PCC 2.1c         */
/*-------------------------*/

#include <stdio.h>

main()
{
    printf("Hola");


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

Este es un programa que escribe "Hola" en la pantalla. Nos lo creemos, verdad? Bueno, pues ahora vamos a ir comentando cosas sobre él:

- Lo que hay escrito entre /* y */ es un **COMENTARIO**, que sirve para aclararnos algo a nosotros, pero que el compilador "se salta" sin ningún tipo de reparo. Es conveniente escribir comentarios que aclaren la misión de las partes de nuestros programas que puedan resultar menos claras a simple vista. Incluso es aconsejable que el programa comience con un comentario (como en este caso), que nos recuerde qué hace el programa sin que necesitemos mirarlo de arriba a abajo. Un comentario puede empezar en una línea y terminar en otra distinta, así:

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

- Eso de "#include" recuerda mucho al "uses" de Turbo Pascal, y de momento me limitaré a comentar que también nos permite **AMPLIAR** el lenguaje base. En este caso, hemos incluido un fichero "de cabecera" llamado "stdio.h" (standard i/o, entrada y salida estándar), que es el que define la orden "printf".

¿Y por qué se pone entre < y >? ¿Y por qué eso de # al principio? "Porque sí..." No, esto no es cierto del todo, hay razones... que ya iremos viendo más adelante.

- Ya que estamos con él: por si es poco evidente, "printf" es la orden que se encarga de **mostrar un texto** en pantalla. Como he comentado de pasada, esta función es la responsable de que hayamos escrito "stdio.h" al principio del programa. Resulta que en el lenguaje C base no hay predefinida ninguna orden para escribir en pantalla (!), sino que están definidas en "stdio.h". Pero esto tampoco es mayor problema, porque vamos a encontrar el dichoso "stdio.h" en cualquier compilador que usemos.

- Aun quedan cosas: ¿qué pintan esas llaves { y } ? Al igual que el Pascal y la mayoría de los lenguajes actuales, el 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 } (el equivalente de "begin" y "end" de Pascal).

- 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 lo que realmente es el **cuerpo** del programa.

  Por cierto, esta función llamada "main" **debe existir siempre**.

- Y por qué tiene un paréntesis vacío a continuación? Pues precisamente para indicar que se trata de una **función** (ya las estudiaremos más adelante, para quien aún no sepa lo que son). En Pascal, cuando una función no tiene parámetros, no se pone ni siquiera los paréntesis; en C hay que ponerlos, aunque queden vacíos (si no hay ningún parámetro, como en este caso).

Y la cosa no acaba aquí. Aún queda más miga de la que parece en este programa, pero cuando ya vayamos practicando un poco iremos concretando más alguna que otra cosa de las que aquí han quedado un tanto por encima.

Sólo un par de cosas más antes de seguir adelante:

- Cada orden de C debe terminar con un **PUNTO Y COMA** (al igual que en Pascal; aun así, hay algunas pequeñas diferencias que ya iremos viendo).

- 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 órden y donde empieza la siguiente son los puntos y coma, como acabo de comentar. Por ese motivo, el programa anterior se podría haber escrito también así (aunque no es aconsejable, porque puede resultar menos legible):

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

- La gran mayoría de las órdenes que encontraremos en el lenguaje C son palabras en inglés o abreviaturas de éstas, al igual que ocurre en la mayoría de los lenguajes de programación. En cambio, en C y al contrario que en otros lenguajes (como Pascal o Basic, por ejemplo) sí existe **DISTINCION** entre mayúsculas y minúsculas, por lo que "printf" es una palabra reconocida, pero "Printf", "PRINTF" o "PrintF" no lo son.

- Como veremos más adelante, en Pascal se distingue entre "funciones" (function), que realizan una serie de operaciones y devuelven un valor, y "procedimientos" (procedure), que no devuelven ningún valor. Ahora sólo tendremos **FUNCIONES**, pero algunas de ellas podrán ser de un cierto tipo "nulo", con lo cual equivalen a los procedimientos de Pascal.

Ya sabemos escribir, pero eso no nos bastará en la mayoría de los casos, así que sigamos viendo cosas nuevas...

N.

Pasar al tema 2
Volver al índice de temas básicos
Volver al menú principal

CURSO DE C. TEMA 2.

Introducción al manejo de variables

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.

En el primer ejemplo nos permitía escribir "Hola", pero normalmente no sabremos de antemano lo que vamos a escribir, sino que dependerá de una serie de cálculos previos (el total de una serie de números que hemos leído, por ejemplo).

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 primero cómo sumaríamos dos números enteros que fijemos en el programa:

```
/*-------------------------*/
/*   Ejemplo en C nº 2:       */
/*   EJ02.C                   */
/*                            */
/*   Introd. a variables      */
/*                            */
/*   Comprobado con:          */
/*      - Turbo C++ 1.01      */
/*      - Djgpp 2.01          */
/*      - Symantec C++ 6.0    */
/*      - GCC 2.6.3           */
/*      - PCC 2.1c            */
/*-------------------------*/

#include <stdio.h>

int primerNumero;      /* Nuestras variables */
int segundoNumero;
int suma;

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

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

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

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

En este ejemplo vemos como se declaran las variables **en C**: indicando primero el tipo del que son (en este caso "int", números enteros) y a continuación el nombre que les vamos a dar:

```
int primerNumero;
```

Esta línea quiere decir que usaremos una variable, cuyo nombre es "primerNumero", y que el valor que guardará esa variable será un número entero. Posteriormente, con

```
primerNumero = 234;
```

asignamos un valor a esa variable.

Estos nombres de variable (**identificadores**) siguen prácticamente las mismas reglas de sintaxis que en otros lenguajes como Pascal: 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.

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. No voy a incluir una lista de palabras reservadas de C, porque eso supondría tener que comentar al menos para que sirven, y además nos encontramos con que hay incluso ciertas palabras que están reservadas en unos compiladores sí y en otros no... La conclusión es que deberíamos usar nombres de variables que a nosotros nos resulten claros, y que tengan pinta de poder ser alguna orden de C.

Sigamos... Insisto en que en C las mayúsculas y minúsculas se consideran diferentes, de modo que si intentamos hacer

```
PrimerNumero = 0;
primernumero = 0;
```

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.

Por cierto, como las tres variables van a guardar números enteros, podemos declararlas **a la vez**, separadas por comas:

```
int primerNumero;
int segundoNumero;
int suma;
```

es lo mismo que

```
int primerNumero, segundoNumero, suma;
```

o que

```
int primerNumero,
    segundoNumero,
    suma;
```

Pero aquí hay más cosas "raras". Porque ahora en printf aparece ese "%d"? Habíamos hablado de printf como si escribiera en pantalla lo que nosotros le indicamos entre comillas. Esto no es del todo cierto:

Eso que le hemos indicado 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. Esto lo veremos con detalle un poco más adelante, pero de momento anticipo que cuando el mensaje aparezca en pantalla, ese "%d" se sustituirá por un número entero. Qué número? El que le indicamos a continuación, separado por una coma (en este caso, "suma").

Seguir avanzando en el tema 2

Volver al índice de temas básicos
Volver al menú principal

CURSO DE C. TEMA 2.2 (de 8).

En C es posible **dar un valor** a las variables a la vez que se las declara (algo parecido a lo que se puede hacer Turbo Pascal empleando "constantes con tipo"). Así, el programa anterior quedaría:

```
/*-------------------------*/
/*   Ejemplo en C nº 3:    */
/*   EJ03.C                */
/*                         */
/*   Inicialización de     */
/*   variables             */
/*                         */
/*   Comprobado con:       */
/*      - Turbo C++ 1.01   */
/*      - Djgpp 2.01       */
/*      - Symantec C++ 6.0 */
/*      - GCC 2.6.3        */
/*      - PCC 2.1c         */
/*-------------------------*/

#include <stdio.h>

int primerNumero = 234;      /* Nuestras variables */
int segundoNumero = 567;
int suma;

main()
{
    suma = primerNumero + segundoNumero;

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

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

Al igual que en el caso anterior, como las tres variables son del mismo tipo, podemos declararlas a la vez haciendo:

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

¿Y cómo escribimos más de un valor a la vez? Pues vamos a verlo con otro ejemplo...

Seguir avanzando en el tema 2
Volver al índice de temas básicos
Volver al menú principal

CURSO DE C. TEMA 2.3 (de 8).

Vamos a ver cómo imprimir más de un valor, también ampliando el ejemplo anterior:

```
/*-------------------------*/
/*   Ejemplo en C nº 4:    */
/*   EJ04.C                */
/*                         */
/*   Escribir más de una   */
/*   variable              */
/*                         */
/*   Comprobado con:       */
/*      - Turbo C++ 1.01   */
/*      - Djgpp 2.01       */
/*      - Symantec C++ 6.0 */
/*      - GCC 2.6.3        */
/*      - PCC 2.1c         */
/*-------------------------*/

#include <stdio.h>

int primerNumero = 234;      /* Nuestras variables */
int segundoNumero = 567;
int suma;

main()
{
    suma = primerNumero + segundoNumero;

    printf("El primer número es %d, el segundo %d y su suma %d.",
      primerNumero, segundoNumero, suma);
}

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

Un único comentario a este programa: hemos dicho que el C es un lenguaje de formato libre. Por eso, no hay ningún problema en dividir en dos la línea de la orden "printf", como hemos hecho, con la intención de que se vea la orden entera en una pantalla "normal". Lo de que la segunda línea del "printf" comience un poco más a la derecha es simplemente buscando mayor legibilidad: para que se vea que es continuación de la línea anterior, en vez de ser una orden nueva.

Sigamos. Hasta ahora hemos estado viendo cómo escribir variables, pero si queremos que realmente sean variables (que sus valores puedan variar), es lógico pensar que no tendríamos que ser nosotros quienes les damos su valor, sino los usuarios de nuestros programas. Entonces, debe haber alguna forma de que el usuario **introduzca** datos. Vamos a verlo...

Seguir avanzando en el tema 2
Volver al índice de temas básicos
Volver al menú principal

CURSO DE C. TEMA 2.4 (de 8).

Veamos cómo dejar que el usuario sea quien introduzca los datos:

```
/*-------------------------*/
/*   Ejemplo en C nº 5:      */
/*   EJ05.C                  */
/*                           */
/*   Leer valores para       */
/*   variables               */
/*                           */
/*   Comprobado con:         */
/*      - Turbo C++ 1.01     */
/*      - Djgpp 2.01         */
/*      - Symantec C++ 6.0   */
/*      - GCC 2.6.3          */
/*      - PCC 2.1c           */
/*-------------------------*/

#include <stdio.h>

int primerNumero, segundoNumero, suma;      /* Nuestras variables */

main()
{
    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);
}

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

Por si alguien no cae, la instrucción para que el usuario pueda dar un valor a una variable es "scanf".

Una vez que hemos visto que era ese %d de "printf", ya intuimos que en este caso servirá para indicar que lo que vamos a leer es un número entero.

Pero ¿qué es ese **&** que aparece antes de cada variable? Es porque lo que le estamos indicando a "scanf" es la **DIRECCION** en la que se debe guardar los datos. En nuestro caso, será la dirección de memoria que habíamos reservado para la variable "primerNumero" y posteriormente la reservada para "segundoNumero".

Bueno, ahora que ya vamos viendo cómo podemos ver en pantalla el contenido de una variable, cómo podemos fijarlo, y como podemos dejar que sea el usuario quien le dé un valor, creo que ya va siendo hora de ver qué **tipos de variables** más habituales que podemos usar...

- **int**. Ya comentado: es un número entero (en el DOS, desde -32768 hasta 32767; en otros sistemas operativos, como UNIX, pueden ser valores distintos -hasta unos dos mil millones-).
- **char**. Un carácter (una letra, una cifra, un signo de puntuación, etc.). Se indica entre comillas simples: letra = 'W'
- **float**. Un numero real (con decimales).

Seguir avanzando en el tema 2
Volver al índice de temas básicos
Volver al menú principal

CURSO DE C. TEMA 2.5 (de 8).

Hemos visto son los tipos más sencillos. También podemos crear arrays y registros:

· Un **array** es un conjunto de elementos, todos los cuales son del mismo tipo. Es la estructura que emplearemos normalmente para crear vectores y matrices. Por ejemplo, para definir un grupo de 5 números enteros y hallar su suma podemos hacer:

```
/*-------------------------*/
/*   Ejemplo en C nº 6:      */
/*   EJ06.C                  */
/*                           */
/*   Primer ejemplo de       */
/*   arrays                  */
/*                           */
/*   Comprobado con:         */
/*      - Turbo C++ 1.01     */
/*      - Djgpp 2.01         */
/*      - Symantec C++ 6.0   */
/*      - GCC 2.6.3          */
/*      - PCC 2.1c           */
/*-------------------------*/

#include <stdio.h>

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

main()
{
    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 y engorrosa */
    /* Ya lo iremos mejorando */
}

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

Una primera observación: hemos declarado un array de 5 elementos. Para numerarlos, **se empieza en 0**, luego tendremos desde "numero[0]" hasta "numero[4]", como se ve en el ejemplo.

Esto es muy mejorable. La primera mejora evidente (o casi) es no tener que dar los valores uno por uno. Podemos dar un valor a las variables, como hicimos en el ejemplo 3...

Seguir avanzando en el tema 2
Volver al índice de temas básicos
Volver al menú principal

CURSO DE C. TEMA 2.6 (de 8).

Vamos a dar un valor inicial a un "array":

```
/*-------------------------*/
/*   Ejemplo en C nº 7:       */
/*   EJ07.C                   */
/*                            */
/*   Inicialización de        */
/*   arrays                   */
/*                            */
/*   Comprobado con:          */
/*      - Turbo C++ 1.01      */
/*      - Djgpp 2.01          */
/*      - Symantec C++ 6.0    */
/*      - GCC 2.6.3           */
/*      - PCC 2.1c            */
/*-------------------------*/

#include <stdio.h>

int numero[5] =              /* Un array de 5 números enteros */
   {200, 150, 100, -50, 300;
int suma;                    /* Un entero que será la suma */

main()
{
    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 */
}

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

Quien haya llegado hasta aquí y ya sepa algo de programación, imaginará que la siguiente mejora es no repetir los valores en

```
suma = numero[0] + ...
```

La forma de hacerlo será empleando algún tipo de estructura que nos permita repetir varios pasos sin tener que indicarlos uno a uno. Es el caso del bucle "for" en Pascal (o en Basic), que ya veremos cómo usar en C.

Podemos declarar arrays de **dos o más dimensiones**, para guardar matrices, por ejemplo. A quien no haya estudiado nada de matrices, esto le puede sonar a chino, así que pongamos un ejemplo: Queremos guardar datos sobre 100 personas, y para cada persona nos interesa almacenar 15 números, todos ellos reales. Haríamos:

```
float datos[100][15]
```

El dato número 10 de la persona 20 sería "datos[19][9]" (recordemos que se empieza a numerar **en 0**).

Aún nos queda algo por ver...

Seguir avanzando en el tema 2
Volver al índice de temas básicos
Volver al menú principal

CURSO DE C. TEMA 2.7 (de 8).

Sigamos. El próximo paso es ver qué es eso de los registros.

· Un **registro** es una agrupación de datos, los cuales no necesariamente son del mismo tipo. Se definen con la palabra clave "struct", y su manejo (muy parecido a como se usan los "records" en Pascal) es: para acceder a cada uno de los datos que forman el registro, se debe indicar el nombre de la variable y el del dato (o campo) separados por un punto:

```
/*-------------------------*/
/*   Ejemplo en C nº 8:       */
/*   EJ08.C                   */
/*                            */
/*   Registros (struct)       */
/*                            */
/*   Comprobado con:          */
/*      - Turbo C++ 1.01      */
/*      - Djgpp 2.01          */
/*      - Symantec C++ 6.0    */
/*      - GCC 2.6.3           */
/*      - PCC 2.1c            */
/*-------------------------*/

#include <stdio.h>

struct {
   char   inicial;
   int    edad;
   float nota;
    persona;

main()
{
    persona.inicial = 'J';
    persona.edad = 20;
    persona.nota = 7.5;
    printf("La edad es %d", persona.edad);
}

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

Como es habitual en C, primero hemos indicado el tipo de la variable: (struct { ...) y después el nombre de la variable (persona).

Hemos guardado varios datos de una persona. Se pueden almacenar los de **varias** personas si combinamos esto con los array 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".

También hay otros tipos especiales de "struct", aunque de manejo más avanzado. Es el caso de uniones y campos de bits

Los tipos permiten **modificadores**: unsigned o signed, y long o short.

Por ejemplo, un char por defecto se considera que es un valor con signo, luego va desde -128 hasta +127. Si queremos que sea **positivo**, y entonces que vaya desde 0 hasta 255, deberíamos declararlo como "**unsigned** char".

De igual modo, un int va desde -32768 hasta 32767. Si queremos que sea positivo, y entonces vaya desde 0 hasta 65535, lo declararemos como "unsigned int".

Podemos ampliar el rango de valores de un entero usando "**long**": un "long int" irá desde -2147483648 hasta 2147483647. Por contraposición, un "**short** int" será un entero no largo ("normal").

También hay otros tipos de datos "menos habituales". Se trata de valores reales que permiten una mayor precisión y un mayor rango de valores que "float". Estos son "**double**" y "**long double**".

Los modificadores permitidos son:

Modificador	Aplicable a
short	int
long	int, double
signed	int, char
unsigned	int, char

Pero quizá sea mejor recopilar todo esto en una tabla...

Seguir avanzando en el tema 2
Volver al índice de temas básicos
Volver al menú principal

CURSO DE C. TEMA 2.7b (de 8). Uniones y campos de bits.

(--- Nota: este tema sólo está disponible en la versión ampliada del curso ---).

Volver al tema 2.7
Volver al índice de temas básicos
Volver al menú principal

CURSO DE C. TEMA 2.8 (de 8).

Vamos a recopilar todo esto de los modificadores en una tabla. Los paréntesis indican que una cierta parte del nombre es el valor "por defecto" (el que se considera, mientras no se indique lo contrario). A la derecha indico el valor mínimo y el valor máximo que puede tomar una variable de ese tipo:

Nombre	Bytes	Min	Max
(signed) char	1	-128	127
unsigned char	1	0	255
(signed) (short) int	2	-32768	32767
unsigned (short) (int)	2	0	65535
(signed) long (int)	4	-2147483648	2147483647
unsigned long (int)	4	0	4294967295
float	2	3.4E-38	3.4E+38
double	4	1.7E-308	1.7E+308
long double	5	3.4E-4932	1.1E+4932

Con los paréntesis me refiero a que "signed char" es lo mismo que "char", y que "long" es lo mismo que "long int" y que "signed long int", por ejemplo.

Nota: estos tamaños y rangos de valores son válidos en la mayoría de los compiladores **bajo DOS** (16 bits). Puede existir compiladores para otros sistemas operativos (incluso algún caso bajo DOS) en los que estos valores cambien. Por ejemplo, con GCC para Linux, el rango de los "int" coincide con el que en MsDos suele ser un "long int".

En cualquier caso, estos valores se pueden comprobar, porque tenemos definidas unas constantes como **MAXINT** o MAXLONG (en el fichero de cabecera "values.h"), que nos dicen hasta qué valor podemos llegar con cada tipo de número.

¿Y los **Booleanos**? Recordemos que en Pascal (y en otros lenguajes) contamos con un tipo especial de datos que puede valer "verdadero" o "falso". En C no es así, sino que se trabaja con enteros. Entonces, al comprobar una condición no obtendremos directamente "verdadero" o "falso", sino 0 (equivalente a FALSO) o un número distinto de cero (equivalente a VERDADERO; normalmente será 1).

¿Y las **cadenas de texto**? Aquí la cosa se complica un poco. No existe un tipo "string" como tal, sino que las cadenas de texto se consideran "arrays" de caracteres. Están formadas por una sucesión de caracteres **terminada con un carácter nulo (0)**, y no serán tan fáciles de manejar como lo son en Pascal y otros lenguajes. Por ejemplo, no podremos hacer cosas como

```
Nombre := 'Don ' + Nombre;
```

Es decir, algo tan habitual como concatenar cadenas se va a complicar (para quien venga de Pascal o Basic). Tendremos que usar unas funciones específicas, pero eso lo veremos más adelante. Ahora vamos a ver un primer ejemplo del uso de cadenas de texto, en su forma más sencilla, pero que también es la menos habitual en la práctica.

```
/*-------------------------*/
/*   Ejemplo en C nº 9:    */
/*   EJ09.C                */
/*                         */
/*   Introducción a las    */
/*   cadenas de texto      */
/*                         */
/*   Comprobado con:       */
/*     - Turbo C++ 1.01    */
/*     - Djgpp 2.01        */
/*     - Symantec C++ 6.0  */
/*     - GCC 2.6.3         */
/*     - PCC 2.1c          */
```

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

#include <stdio.h>

char texto1[80] = "Primer texto";
char texto2[80];

main()
{
    printf("Introduzca un texto: ");
    scanf("%s", &texto2);
    printf("El primer texto es %s", texto1);
    printf(" y Vd. ha tecleado %s", texto2);
}

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

Creo que el ejemplo se entiende por sí mismo, porque la única novedad con respecto a ejemplos anteriores es "**%s**" para indicar que lo que queremos leer o escribir es una cadena de texto.

Eso sí, varios comentarios:

- Hemos reservado 80 caracteres (realmente 79, porque recordemos que al guardarlo en memoria se le añade un carácter 0 para indicar el final de la cadena), así que si tecleamos **más de 79 caracteres**, nadie nos asegura lo que vaya a pasar: puede que nuestro programa intente guardar todo lo que hemos tecleado, sin pensar que, como no cabe, puede estar "machacando" otros datos o incluso instrucciones de nuestro programa.

- Segundo comentario: 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, en el próximo apartado veremos cómo evitarlo.

- Tercero: **no hace falta** el "&" de "&texto2" en "scanf", porque habíamos dicho que ese símbolo hacía referencia a la dirección en la que se nos había reservado la memoria para esa variable, cosa que no es necesaria en un array, porque es el propio nombre lo que está haciendo referencia a la dirección. A que parece un trabalenguas? No es tan complicado: para que se vea a qué me refiero, va un ejemplo, comparando una variable "normal" y un array:

  ```
  int Dato = 5;

    Dato vale 5;
    &Dato nos da la dirección en que está guardado

  char Texto[40] = "Una prueba";

    Texto[0] vale 'U' (la primera letra).
    Texto[1] vale 'n' (la segunda), y así sucesivamente.
    Texto (sin corchetes) da la dirección en está guardado el array.
  ```

  Por tanto, como "scanf" espera que le digamos la dirección en la que queremos guardar la cadena de texto (un array), podemos usar

  ```
  scanf("%s", texto2);
  ```

- Cuarto (y último): Algo que igual alguien ha pasado por alto, así que lo quiero recalcar. Las cadenas de texto se encierran entre comillas **dobles** ("Hola") y las letras aisladas entre comillas **simples** ('H').

N.

Pasar al tema 3
Volver al índice de temas básicos
Volver al menú principal

CURSO DE C. TEMA 3. Pequeña recapitulación.

Bueno, parece que ya vamos sabiendo algo. Pero antes de avanzar, va siendo el momento de puntualizar y/o concretar algunas de las cosas que hemos ido viendo, para que esto tenga también algo de rigor...

Si asusta, pues no hay más que saltar al siguiente tema, y volver más adelante, cuando ya se tenga más base.

#include y .h

Es muy frecuente que un programa en C esté dividido en varios módulos, que se deben ensamblar en el momento de crear el programa ejecutable.

Entonces puede ocurrir que en "prog2.c" hayamos definido una función que necesita "prog1.c". Cuando "prog1" comienza a compilar descubre que no conoce una de las funciones que aparecen por ahí, y el compilador protesta.

Una forma de evitarlo es poniendo "**la cabecera**" de la función que falta, es decir, su nombre y sus parámetros (pero sin detallar los pasos que debe dar; de eso ya se encarga "prog2.c").

Así "prog1" sabe que esa función existe, que no es que hayamos tecleado algo mal, sino que "está ahí", y que ya le llegarán los detalles concretos más adelante, cuando termine de ensamblar todos los módulos.

A quien venga de Pascal, eso de "la cabecera" ya le sonará, porque es igual que lo que ponemos en la sección "interface" de las unidades que creemos nosotros mismos.

Pero además tenemos más casos que éstos, porque incluso si nuestro programa está formado por un sólo módulo, normalmente necesitaremos funciones como "printf", que, como hemos comentado, no forman parte del **lenguaje base**. Nos encontramos con el mismo problema: si no ponemos la cabecera, el compilador no sabrá que esa función "existe pero aún no se la hemos explicado", sino que supondrá que hemos escrito algo mal y protestará.

Así que, entre unas y otras, tendríamos que llenar nuestro programa con montones y montones de cabeceras. La forma habitual de evitarlo es juntar esas cabeceras en "ficheros de cabecera", que luego incluimos en nuestro programa.

Así, todas las funciones que están relacionadas con la entrada y salida estándar, y que se enlazan en el momento de crear el ejecutable, las tenemos declaradas en el fichero de cabecera "stdio.h". Por eso, cuando escribimos

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

el compilador coge el fichero "stdio.h", lee todas las cabeceras que contiene y las "inserta" en ese mismo punto de nuestro programa.

¿Y porque eso de #? Pues porque no es una orden del lenguaje C, sino una orden directa al compilador (**una "directiva"**). Cuando llega el momento de comprobar la sintaxis del lenguaje C ya no existe eso de "include", sino que en su lugar el compilador ya ha dejado todas las cabeceras que queríamos.

¿Y eso de < >? Pues podemos encontrar líneas como

```
#include <stdio.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 de "includes". El segundo caso es un fichero de cabecera que hemos creado **nosotros**, por lo que lo indicamos entre " y ", y así el compilador sabe que no debe buscarlo entre SUS directorios, sino en el mismo directorio en el que está nuestro programa.

Warnings y main()

Según el compilador que se use, puede que con los ejemplos anteriores haya aparecido un "warning" que diga algo como

Warning: function should return a value in function main

Es decir "aviso: la función debería devolver un valor, en la función MAIN".

Vayamos por partes:

- Eso de los **WARNING** son avisos del compilador, una forma de decirnos "Cuidado, que puede que esto no esté del todo bien"

- ¿Por qué avisa en este caso? Recordemos que ya comentamos en el primer tema que en C no hay funciones y procedimientos, sino sólo funciones. Por tanto, siempre deberán **devolver un valor**, aunque puede que este sea de un tipo especial "nulo". Y en cambio, en nuestros programas se veía por ahí ningún valor devuelto en "main"...

- ¿Formas de hacer que esté del todo correcto, para que el compilador no proteste? Hay dos: una (menos correcta) es decir que "main" es de ese tipo "nulo", con lo cual no hay que devolver ningún valor, y otra (más correcta) es obligar a que devuelva un valor, que normalmente será 0 (para indicar que todo ha funcionado correctamente; otro valor indicaría algún tipo de error). Este valor es el "errorlevel" que se devuelve al DOS cuando termina la ejecución del programa. Pues vamos con la forma "buena":

```
/*-------------------------*/
/*   Ejemplo en C nº 10:     */
/*   EJ10.C                  */
/*                           */
/*   "main" como int         */
/*                           */
/*   Comprobado con:         */
/*      - Turbo C++ 1.01     */
/*      - Djgpp 2.01         */
/*      - Symantec C++ 6.0   */
/*      - GCC 2.6.3          */
/*      - PCC 2.1c           */
/*-------------------------*/

#include <stdio.h>

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

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

Con eso de "**int main()**" decimos que la función "main" va a devolver un valor entero. Es lo que se considera por defecto (si no ponemos "int", el compilador considera de todas formas que lo es).

Con eso de "**return 0**" devolvemos el valor 0 a la salida de "main", que indica que todo ha ido correctamente. Como la función ya devuelve un valor, el Warning desaparece.

Seguir avanzando en el tema 3
Volver al índice de temas básicos
Volver al menú principal

CURSO DE C. TEMA 3.2 (de 2).

La otra forma (la "menos buena") de que desaparezca el "Warning" es

```
/*-------------------------*/
/*   Ejemplo en C n 11:    */
/*   EJ11.C                */
/*                         */
/*   "main" como void      */
/*                         */
/*   Comprobado con:       */
/*      - Turbo C++ 1.01   */
/*      - Djgpp 2.01       */
/*      - Symantec C++ 6.0 */
/*      - GCC 2.6.3        */
/*      - PCC 2.1c         */
/*-------------------------*/

#include <stdio.h>

void main()
{
    printf("Hola");
}

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

Eso de "**void**" es el tipo nulo al que nos referíamos. Así indicamos que la función no va a devolver valores.

Esta es la opción "menos buena" porque puede que a algún compilador no le guste eso de que "main" no devuelva un valor entero, aunque no es lo habitual: les suele bastar con que exista la función "main", sea del tipo que sea.

N.

Pasar al tema 4
Volver al índice de temas básicos
Volver al menú principal

CURSO DE C. TEMA 4. Entrada/Salida básica

Hemos visto por encima cómo mostrar datos en pantalla y cómo aceptar la introducción de datos por parte del usuario, mediante "printf" y "scanf", respectivamente. Veamos ahora su manejo y algunas de sus posibilidades con más detalle, junto con otras órdenes alternativas:

El formato de **printf** es

```
printf( formato, lista de variables);
```

Dentro del apartado de "**formato**" habíamos comentado que "%d" indicaba que se iba a escribir un número entero, y que "%s" indicaba una cadena de texto. Vamos a resumir en una tabla las demás posibilidades, que no habíamos tratado todavía:

Código	Formato
%d	Número entero con signo, en notación decimal
%i	Número entero con signo, en notación decimal
%u	Número entero sin signo, en notación decimal
%o	Número entero sin signo, en notación octal (base 8)
%x	Número entero sin signo, en hexadecimal (base 16)
%X	Número entero sin signo, en hexadecimal, mayúsculas
%f	Número real (coma flotante, con decimales)
%e	Número real en notación científica
%g	Usa el más corto entre %e y %f
%c	Un único carácter
%s	Cadena de caracteres
%%	Signo de tanto por ciento: %
%p	Puntero (dirección de memoria)
%n	Se debe indicar la dirección de una variable entera (como en scanf), y en ella quedará guardado el número de caracteres impresos hasta ese momento

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.
- Si ese número empieza por 0, los espacios sobrantes (si los hay) de la anchura mínima se rellenan con 0.
- Si ese número tiene decimales, indica el número de dígitos enteros y decimales si los que se va a escribir es un número, o la anchura mínima y máxima si se trata de una cadena de caracteres.
- Si el número es negativo, la salida se justificará a la izquierda (en caso contrario, es a la derecha -por defecto-).
- Hay otros dos posibles modificadores: la letra l, que indica que se va a escribir un long, y la letra h, que indica que se trata de un short.

Todo esto es para printf, pero coincide prácticamente en el caso de scanf.

Antes de seguir, vamos a ver un ejemplo con los casos más habituales...

Seguir avanzando en el tema 4
Volver al índice de temas básicos
Volver al menú principal

CURSO DE C. TEMA 4.2 (de 4).

Este es un ejemplo de los formatos más habituales:

```
/*-------------------------*/
/*   Ejemplo en C nº 12:     */
/*   EJ12.C                  */
/*                           */
/*   Formatos con "printf"   */
/*                           */
/*   Comprobado con:         */
/*      - Turbo C++ 1.01     */
/*      - Djgpp 2.01         */
/*      - Symantec C++ 6.0   */
/*      - GCC 2.6.3          */
/*      - PCC 2.1c           */
/*-------------------------*/

#include <stdio.h>

int     entero = 1234;
int     enteroNeg = -1234;
float real = 234.567;
char    letra = 'E';
char    mensaje[20] = "Un texto";
int     contador;

main()
{
     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 %3.2f si limitamos a dos decimales,\n", real);
     printf("    y %e en notación científica (exponencial).\n", real);
     printf("La letra es %c y el texto %s.\n", letra, mensaje);
     printf("    Podemos poner \"tanto por ciento\": 50%%.\n");
     printf("Finalmente, podemos escribir direcciones. de memoria: %p.\n",
       &letra);
     printf("    y contar lo escrito hasta aquí%n", &contador);
     printf(", que ha sido: %d letras.\n", contador);
}

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

Aparecen cosas nuevas, como eso de "\n", hace que avancemos una línea en la pantalla, y que veremos con más detalle en un instante, pero antes vamos a analizar **el resultado** de este programa con Turbo C++:

```
- - - - - - - - - - - - - - - - - - - - - - - - - -

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: 64302 (incorrecto).
El número real vale 234.567001 en notación normal
   y 234.57 si limitamos a dos decimales,
   y 2.345670e+02 en notación científica (exponencial).
La letra es E y el texto Un texto.
   Podemos poner "tanto por ciento": 50%.
Finalmente, podemos escribir direcciones. de memoria: 00B0.
   y contar lo escrito hasta aquí, que ha sido: 32 letras.

- - - - - - - - - - - - - - - - - - - - - - - - -
```

Creo que queda todo bastante claro (puede volver al apartado anterior para consultar los distintos formatos si no es así), pero aun así hay una cosa desconcertante: Por qué el número real aparece como 234.567001, si nosotros lo hemos definido como 234.567? Porque los números reales se almacenan con una cierta **pérdida de precisión**. Si esta pérdida es demasiado grande para nosotros, deberemos usar otro tipo, como **double**.

Lo de que el número negativo quede mal al intentar escribirlo como positivo, lo comento de pasada para quien sepa ya algo de aritmética binaria: el primer bit a uno en un número con signo indica que es un número negativo, mientras que en uno positivo es el más significativo. Por eso, tanto el número -1234 como el 64302 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.

Y aun hay más: si lo compilamos y lo ejecutamos con **Symantec C++**, vemos que hay dos diferencias:

```
 -> Finalmente, podemos escribir direcciones. de memoria: 0068.
```

Esto no es problema, porque la dirección de memoria en la que el compilador nos reserve una variable no nos importa; nos basta con saber que realmente contamos con esa memoria para nuestros datos.

```
 -> y 182453458 si le hacemos que crea que es entero largo,
```

Esto ya asusta más. ¿Por qué el 1234 se ha convertido en esa cosa tan rara? Pues porque un entero largo ocupa el doble que un entero normal, así que es muy posible que si hacemos cosas como éstas, esté intentando leer 4 bytes donde nosotros sólo hemos definido 2, tomará datos que hayamos reservado para otras variables, y el resultado puede ser erróneo.

Para evitar esto, se puede hacer una "**conversión de tipos**" (en inglés "typecast"), para que el valor que vayamos a imprimir sea realmente un entero largo:

```
   printf("    y %ld si le hacemos que crea que es entero largo,\n",
      (long) entero);
```

Así, antes de imprimir coge nuestro valor, lo convierte realmente en un entero largo (el dato leído, porque no modifica el original) y lo muestra correctamente. Así nos aseguramos de que funcionará en cualquier compilador.

Finalmente, si lo compilamos con **GCC** (bajo Linux) los resultados también son "casi iguales", y esta vez las diferencias son:

```
   -> Finalmente, podemos escribir direcciones. de memoria: 0x2010.
```

Esto, como antes, no debe preocuparnos.

```
   -> El entero negativo vale -1234
   ->     y podemos hacer que crea que es positivo: 4294966062 (incorrecto).
```

Este valor es distinto del visto anteriormente simplemente porque para GCC bajo Linux, un entero ocupa 4 bytes en vez de 2. Por tanto, -1234 y 4294966062 dan lugar a la misma secuencia de ceros y unos, pero esta vez con 32 bits en vez de 16.

Este es el mismo resultado que se obtiene con Djgpp 2.01.

En el caso del compilador PCC 2.1c, obtenemos "casi" lo mismo, pero no idéntico, porque no permite algunos de los códigos de formato, como "%p" y "%n":

```
- - - - - - - - - - - - - - - - - - - - - - - - - -

El número entero vale 1234 en notación decimal,
   y 2322 en notación octal,
   y 4D2 en notación hexadecimal,
   y FFEC04D2 en notación hexadecimal en mayúsculas,
   y -1309486 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: 64302 (incorrecto).
El número real vale 234.567001 en notación normal
   y 234.57 si limitamos a dos decimales,
   y 2.345670E2 en notación científica (exponencial).
La letra es E y el texto Un texto.
   Podemos poner "tanto por ciento": 50%.
Finalmente, podemos escribir direcciones. de memoria: p.
   y contar lo escrito hasta aquín, que ha sido: 0 letras.

- - - - - - - - - - - - - - - - - - - - - - - - - -
```

Seguir avanzando en el tema 4
Volver al índice de temas básicos
Volver al menú principal

CURSO DE C. TEMA 4.3 (de 4).

Sigamos. Vamos a ver ahora qué era eso de "\n". Las letras que escribamos después de una barra invertida (en inglés, "backslash") tendrán un significado especial. Se trata simplemente de **códigos de control** (llamados también "secuencias de escape") que podemos meter entre medias del texto. En este caso, ese es el carácter de avance de línea (en inglés "new line"), que hace que se siga escribiendo al comienzo de la línea siguiente. Veamos en una tablita cuáles podemos usar:

Código	Significado
\n	nueva línea (new line / line feed)
\r	retorno de carro (carriage return)
\b	retroceso (backspace)
\f	salto de página (form feed)
\t	tabulación horizontal
\v	tabulación vertical
\"	comillas dobles (")
\'	apóstrofe o comillas simples (')
\\	barra invertida (\)
\a	alerta (un pitido)
\0	carácter nulo
\ddd	constante octal (máximo tres dígitos)
\xddd	constante hexadecimal (ídem)

Estos códigos equivalen a ciertos caracteres de control del código ASCII. Por ejemplo, el carácter de salto de página es el 12, luego sería **equivalente** hacer:

```
char FF = 12;              /* Asignación normal, sabiendo el código */
char FF = '\f';          /* Como secuencia de escape */
char FF = '\xC';         /* Como constante hexadecimal */
char FF = '\140;         /* Como constante octal */
```

Seguir avanzando en el tema 4
Volver al índice de temas básicos
Volver al menú principal

CURSO DE C. TEMA 4.4 (de 4).

Para leer datos del teclado, hemos visto cómo usar "**scanf**". Una vez vista la base y conocidos los códigos de formato, no tiene mayor dificultad. Por ejemplo, para leer un número real, haríamos:

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

Podemos leer **más de un dato** seguido, que el usuario deje separados por espacios, usando construcciones como

```
scanf("%f %f", &real1, &real2);
```

pero su uso es peligroso. Personalmente, yo prefiero leerlo como una cadena de texto y analizarlo yo mismo, para evitar errores. Y cómo leemos cadenas de texto, si habíamos visto que "scanf" paraba en cuanto encontraba un espacio? Vamos a verlo...

Cuando queremos trabajar con **cadenas de texto**, tenemos otras posibilidades: con "puts" podemos escribir un texto y con "gets" leer lo que se teclee (esta vez no se para en cuanto lea un espacio en blanco).

Va el ejemplo de turno:

```
/*-------------------------*/
/*   Ejemplo en C nº 13:   */
/*   EJ13.C                */
/*                         */
/*   "gets" y "puts"       */
/*                         */
/*   Comprobado con:       */
/*      - Turbo C++ 1.01   */
/*      - Djgpp 2.01       */
/*      - Symantec C++ 6.0 */
/*      - GCC 2.6.3        */
/*      - PCC 2.1c         */
/*-------------------------*/

#include <stdio.h>

char texto[80];

main()
{
    puts("Teclee una frase.");
    gets(texto);
    puts("Ha escrito: ");
    puts(texto);
}

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

Al ejecutar este programa, vemos que "**puts**" ya avanza automáticamente de línea tras escribir el texto, sin necesidad de que pongamos un '\n' al final.

En "stdio.h" tenemos muuuuchas más funciones, pero vamos sólo a comentar dos más: "getchar()" lee **un carácter** y "putchar()" escribe un carácter.

En la práctica, es más habitual usar otras funciones como "**getch()**", pero estas dependen (en teoría) del compilador que se use, así que las veremos cuando tratemos la pantalla en modo texto.

N.

Pasar al tema 5
Volver al índice de temas básicos
Volver al menú principal

CURSO DE C. TEMA 5. Operaciones matemáticas

Al igual que en Pascal, Basic y otros lenguajes, contamos con una serie de **operadores** para realizar sumas, restas, multiplicaciones y otras operaciones no tan habituales. Vamos a empezar por las cuatro elementales:

Operador	Operación
+	Suma
-	Resta, negación
*	Multiplicación
/	División
%	Resto de la div.

¿Qué ocurre en casos como el de 10/3? Si 10 y 3 son números enteros, ¿qué ocurre con su división? El resultado sería 3, la parte entera de la división. Será 3.3333 cuando ambos números sean reales.

```
/*-------------------------*/
/*  Ejemplo en C nº 14:    */
/*  EJ14.C                 */
/*                         */
/*  Operaciones con        */
/*  números enteros        */
/*                         */
/*  Comprobado con:        */
/*     - Turbo C++ 1.01    */
/*     - Djgpp 2.01        */
/*     - Symantec C++ 6.0  */
/*     - GCC 2.6.3         */
/*     - PCC 2.1c          */
/*-------------------------*/

#include <stdio.h>

int e1   = 10;
int e2   =  4;
float r1 = 10.0;
float r2 =  4.0;

main()
{
    printf("La suma de los enteros es: %d\n", e1+e2);
    printf("  Su producto: %d\n", e1*e2);
    printf("  Su resta: %d\n", e1-e2);
    printf("  Su división: %d\n", e1/e2);
    printf("  El resto de la división: %d\n", e1%e2);
    printf("La suma de los reales es: %f\n", r1+r2);
    printf("  Su producto: %f\n", r1*r2);
    printf("  Su resta: %f\n", r1-r2);
    printf("  Su división: %f\n", r1/r2);
    printf("Un real entre un entero, como real: %f\n", r1/e2);
    printf("  Lo mismo como entero: %d (erróneo)\n", r1/e2);
    printf("  Un entero entre un real, como real: %f\n", e1/r2);
}

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

Estos operadores son "**binarios**". Se llaman así porque trabajan con dos datos: sumamos dos números, restamos dos números,

multiplicamos dos números, etc. Por otro lado, tenemos operadores "**unarios**", que sólo afectan a un dato. El ejemplo más claro es el de la negación: -a pero veremos más adelante otros (++, --, &, ...)

Cuidado quien venga de Pascal, porque en C el operador + (suma) NO se puede utilizar también para **concatenar cadenas** de texto. Tendremos que utilizar funciones específicas (en este caso "strcat"). Tampoco podemos asignar valores directamente, haciendo cosas como "texto2 = texto1", sino que deberemos usar otra función ("strcpy"). Todo esto lo veremos más adelante, en un tema dedicado sólo a las cadenas de texto.

En Turbo Pascal (no en cualquier versión de Pascal), tenemos también formas **abreviadas** de incrementar una variable:

```
inc(a);          en vez de          a := a+1;
```

Algo parecido existe en C, aunque con otra notación:

```
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 tenemos **postdecremento** (a--) y **predecremento** (--a).

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

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

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;
```

Seguir avanzando en el tema 5
Volver al índice de temas básicos
Volver al menú principal

CURSO DE C. TEMA 5.2 (de 3). Operadores lógicos

En la próxima lección veremos cómo hacer comparaciones del estilo de "si A es mayor que B y B es mayor que C". Así que vamos a anticipar cuales son los operadores de comparación en C.

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

Cuidado con el operador de **igualdad**: el formato será if (a==b) ... Si no nos damos cuenta y escribimos if (a=b) estamos asignando a "a" el valor de "b" (lo veremos con más detalle en la próxima lección).

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").

Estas condiciones se puede **encadenar** con "y", "o", etc., que se indican de la siguiente forma (también lo aplicaremos en la próxima lección)

Operador	Significado
&&	Y
||	O
!	No

Así, una orden que sea mezcla de C y español (porque todavía no sabemos bien cómo comprobar condiciones) y que compruebe si una variable "x" vale 2 y si otra variable "z" no es mayor de 5, sería

```
Si (x==2) && !(z>5) entonces ...
```

Seguir avanzando en el tema 5
Volver al índice de temas básicos
Volver al menú principal

CURSO DE C. TEMA 5.3 (de 3). Operadores entre bits

Al igual que en Pascal, podemos hacer operaciones entre bits de dos números (producto, suma, suma exclusiva, etc.), que se indican:

Operador	Operación
~	Complemento (cambiar 0 por 1 y viceversa)
&	Producto lógico (and)
|	Suma lógica (or)
^	Suma exclusiva (xor)
<<	Desplazamiento hacia la izquierda
>>	Desplazamiento a la derecha

Explicar para qué sirven estos operadores implica conocer qué es eso de los bits, cómo se pasa un número decimal a binario, etc. Supondré que se tienen las nociones básicas, y pondré un ejemplo, cuyo resultado comentaré después:

```
/*-------------------------*/
/*   Ejemplo en C nº 14b:    */
/*   EJ14B.C                 */
/*                           */
/*   Operaciones de bits     */
/*   en números enteros      */
/*                           */
/*   Comprobado con:         */
/*      - Turbo C++ 1.01     */
/*      - Djgpp 2.01         */
/*      - Symantec C++ 6.0   */
/*      - GCC 2.6.3          */
/*      - PCC 2.1c           */
/*-------------------------*/

#include <stdio.h>

int a    = 67;
int b    =   33;

main()
{
    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);
}

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

Veamos qué ha ocurrido. La respuesta que nos da Turbo C++ 1.01 (y que coincide con Symantec C++ 6.0) 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 entender esto, deberemos convertir al sistema binario esos dos números:

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

¿Y 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; 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;
...
```

Pasemos ya al tema siguiente, para ver cómo se comprueban condiciones...

N.

Pasar al tema 6
Volver al índice de temas básicos
Volver al menú principal

CURSO DE C. TEMA 6. Condiciones.

Vamos a ver cómo podemos evaluar condiciones. Supongo que está memorizado del tema anterior cómo se expresa "mayor o igual", "distinto de", y todas esas cosas.

La primera construcción sería el "**si ... entonces ...**" (if..then en Pascal y otros lenguajes). El formato en C es

```
if (condición) sentencia;
```

Vamos a verlo con un ejemplo:

```
/*-------------------------*/
/*   Ejemplo en C nº 15:     */
/*   EJ15.C                  */
/*                           */
/*   "if" elemental          */
/*                           */
/*   Comprobado con:         */
/*      - Turbo C++ 1.01     */
/*      - Djgpp 2.01         */
/*      - Symantec C++ 6.0   */
/*      - GCC 2.6.3          */
/*      - PCC 2.1c           */
/*-------------------------*/

#include <stdio.h>

int numero;

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

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

Todo claro, verdad? Pero (como suele ocurrir en C) esto tiene más miga de la que parece: recordemos que en C no existen los números booleanos, luego la condición no puede valer "verdadero" o "falso". Dijimos que "falso" iba a corresponder a un 0, y "verdadero" a un número distinto de cero (normalmente uno).

Por tanto, si escribimos el número 12 y le pedimos que escriba el valor de la comparación (numero>0), un compilador de Pascal escribiría TRUE, mientras que uno de C escribiría 1.

Seguir avanzando en el tema 6
Volver al índice de temas básicos
Volver al menú principal

CURSO DE C. TEMA 6.2 (de 9).

Y aún hay más: como un valor de "verdadero" equivale a uno **distinto de cero**, podemos hacer cosas como ésta:

```
/*-------------------------*/
/*   Ejemplo en C nº 16:     */
/*   EJ16.C                  */
/*                           */
/*   "if" abreviado          */
/*                           */
/*   Comprobado con:         */
/*      - Turbo C++ 1.01     */
/*      - Djgpp 2.01         */
/*      - Symantec C++ 6.0   */
/*      - GCC 2.6.3          */
/*      - PCC 2.1c           */
/*-------------------------*/

#include <stdio.h>

int numero;

main()
{
    printf("Escriba un número: ");
    scanf("%d", &numero);
    if (numero != 0)                    /* Forma "normal" */
        printf("El número no es cero.\n");
    if (numero)                         /* Forma "con truco" */
        printf("Y sigue sin serlo.\n");
}

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

Seguir avanzando en el tema 6
Volver al índice de temas básicos
Volver al menú principal

CURSO DE C. TEMA 6.3 (de 9).

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

```
/*-------------------------*/
/*   Ejemplo en C nº 17:     */
/*   EJ17.C                  */
/*                           */
/*   "if" y sentencias       */
/*   compuestas              */
/*                           */
/*   Comprobado con:         */
/*      - Turbo C++ 1.01     */
/*      - Djgpp 2.01         */
/*      - Symantec C++ 6.0   */
/*      - GCC 2.6.3          */
/*      - PCC 2.1c           */
/*-------------------------*/

#include <stdio.h>

int numero;

main()
{
    printf("Escriba un número: ");
    scanf("%d", &numero);
    if (numero<0)
    {
      printf("El número es ");
      printf("negativo.\n");
    }
}

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

En este caso, si el número es negativo, se hacen dos cosas: escribir un un texto y luego... escribir otro! Claramente, esos dos "printf" podrían ser uno solo, pero es que entonces me quedaría sin ejemplo.

Seguir avanzando en el tema 6
Volver al índice de temas básicos
Volver al menú principal

CURSO DE C. TEMA 6.4 (de 9).

Sigamos... También podemos indicar lo que queremos que se haga si **no se cumple** la condición. Para ello tenemos la construcción "if (condición) sentencia1; else sentencia2;":

```
/*-------------------------*/
/*   Ejemplo en C nº 18:     */
/*   EJ18.C                  */
/*                           */
/*   "if" y "else"           */
/*                           */
/*   Comprobado con:         */
/*      - Turbo C++ 1.01     */
/*      - Djgpp 2.01         */
/*      - Symantec C++ 6.0   */
/*      - GCC 2.6.3          */
/*      - PCC 2.1c           */
/*-------------------------*/

#include <stdio.h>

int numero;

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

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

De nuevo, aquí debe llevar cuidado quien venga de Pascal, porque en C sí se debe poner un **punto y coma** antes de "else".

Seguir avanzando en el tema 6
Volver al índice de temas básicos
Volver al menú principal

CURSO DE C. TEMA 6.5 (de 9).

Las sentencias "if (condición) s1; else s2;" se pueden **encadenar**:

```
/*-------------------------*/
/*   Ejemplo en C nº 19:     */
/*   EJ19.C                  */
/*                           */
/*   "if..else" encadenados */
/*                           */
/*   Comprobado con:         */
/*      - Turbo C++ 1.01     */
/*      - Djgpp 2.01         */
/*      - Symantec C++ 6.0   */
/*      - GCC 2.6.3          */
/*      - PCC 2.1c           */
/*-------------------------*/

#include <stdio.h>

int numero;

main()
{
    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");
}

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

Atención en el ejemplo anterior al operador de comparación == para ver si dos valores son iguales. Insisto en que "if (numero = 0)" daría a número el valor 0. Ahora que sabemos más, nos damos cuenta de que además esto haría que tomase la condición como **falsa**. Para que se vea mejor, no hay más que teclear 0 como entrada a este programa:

Seguir avanzando en el tema 6
Volver al índice de temas básicos
Volver al menú principal

CURSO DE C. TEMA 6.6 (de 9).

```
/*-------------------------*/
/*   Ejemplo en C nº 20:   */
/*   EJ20.C                */
/*                         */
/*   "if" incorrecto       */
/*                         */
/*   Comprobado con:       */
/*      - Turbo C++ 1.01   */
/*      - Djgpp 2.01       */
/*      - Symantec C++ 6.0 */
/*      - GCC 2.6.3        */
/*      - PCC 2.1c         */
/*-------------------------*/

#include <stdio.h>

int numero;

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

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

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 (parecido a lo que hacíamos con "b = ++a", por ejemplo).

Como ya hemos comentado en el apartado anterior, puede darse el caso de que tengamos que comprobar varias condiciones **simultáneas**, y entonces deberemos usar "y" (&&), "o" (||) y "no" (!) para enlazarlas:

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

Seguir avanzando en el tema 6
Volver al índice de temas básicos
Volver al menú principal

CURSO DE C. TEMA 6.7 (de 9).

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

```
condicion ?  v1 : v2;
```

y es algo así como "si se da la condición, toma el valor v1; si no, toma el valor v2". Un ejemplo de cómo podríamos usarlo sería

```
resultado = (operacion == '-') ?  a-b : a+b;
```

que, aplicado a un programita quedaría:

```
/*-------------------------*/
/*   Ejemplo en C nº 21:     */
/*   EJ21.C                  */
/*                           */
/*   Operador condicional ? */
/*                           */
/*   Comprobado con:         */
/*     - Turbo C++ 1.01      */
/*     - Djgpp 2.01          */
/*     - Symantec C++ 6.0    */
/*     - GCC 2.6.3           */
/*     - PCC 2.1c            */
/*-------------------------*/

#include <stdio.h>

int a, b, resultado;
int operacion;

main()
{
    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);
}

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

(Recordemos que en C las expresiones lógicas valían cero -falso- o distinto de cero -verdadero-, de modo que eso que he llamado "condición" puede ser realmente otros tipos de expresiones, como por ejemplo una operación aritmética).

Una nota sobre este programa: alguien avispado puede haberse dado cuenta de que en el ejemplo comparo con el símbolo '-' y en cambio en el programa comparo con 1. Y este cambio de actitud? No se podría haber usado scanf("%c",...) o bien "getchar()" para leer si se pulsa la tecla '-'?

Pues no es tan sencillo, desafortunadamente. El motivo es que cuando pulsamos INTRO tras teclear un número, esta pulsación se queda en el buffer del teclado, y eso es lo que leería el getchar.

Seguir avanzando en el tema 6
Volver al índice de temas básicos
Volver al menú principal

Vamos a verlo con un ejemplo:

```
/*--------------------------*/
/*   Ejemplo en C nº 22:      */
/*   EJ22.C                   */
/*                            */
/*   Problemas de "getchar" */
/*                            */
/*   Comprobado con:          */
/*      - Turbo C++ 1.01      */
/*      - Djgpp 2.01          */
/*      - Symantec C++ 6.0    */
/*      - GCC 2.6.3           */
/*      - PCC 2.1c            */
/*--------------------------*/

#include <stdio.h>

int a, b, resultado;
char operacion;

main()
{
    printf("Escriba un número: ");
    scanf("%d", &a);
    printf("Escriba otro: ");
    scanf("%d", &b);
    printf("Escriba una operación (1 = resta; otro = suma): ");
    operacion = getchar();              /* Da resultado inesperado */
    printf("\nLa operación es %c (%d).\n", resultado, resultado);
    resultado = (operacion == '-') ?   a-b : a+b;
    printf("Y el resultado es %d.\n", resultado);
}

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

Leyéndolo, parece que todo debería salir bien, pero al ejecutarlo no nos deja ni teclear el símbolo, sino que toma el valor que hubiera en el buffer del teclado. Obtenemos el mismo resultado con

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

Por tanto, deberíamos **vaciar** primero el buffer del teclado. En el próximo apartado veremos cómo hacerlo.

Seguir avanzando en el tema 6
Volver al índice de temas básicos
Volver al menú principal

CURSO DE C. TEMA 6.9 (de 9).

Finalmente, cuando 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 caso1: sentencia1;
                 break;
     case caso2: sentencia2;
                 sentencia2b;
                 break;
      ...
     case casoN: sentenciaN;
                 break;
   default:
                 otraSentencia;
};
```

Para quien venga de Pascal, el equivalente sería

```
case expresión of
   caso1: sentencia1;
   caso2: begin sentencia2; sentencia2b; end;
   ...
   casoN: sentenciaN;
else
   otraSentencia;
end;
```

Como la mejor forma de verlo es con un ejemplo, vamos allá:

```
/*-------------------------*/
/*   Ejemplo en C nº 23:     */
/*   EJ23.C                  */
/*                           */
/*   Uso de "switch"         */
/*                           */
/*   Comprobado con:         */
/*      - Turbo C++ 1.01     */
/*      - Djgpp 2.01         */
/*      - Symantec C++ 6.0   */
/*      - GCC 2.6.3          */
/*      - PCC 2.1c           */
/*-------------------------*/

#include <stdio.h>

char tecla;

main()
{
     printf("Pulse una tecla y luego Intro: ");
     tecla = getchar();
     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");
    }
}

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

En primer lugar, insisto en una cosa: **getchar()** es lo mismo que scanf("%c",...). Por qué lo digo? Porque no basta con escribir la letra: la letra se leerá realmente cuando se procese todo el buffer del teclado, es decir, que debemos pulsar **Intro** después de la tecla que elijamos.

Otra cosa: no se pueden definir **subrangos** de la forma '0'..'9' ("desde 0 hasta 9", cosa que sí ocurre en Pascal), sino que debemos enumerar todos los casos. Pero, como se ve en el ejemplo, para los casos repetitivos no hace falta repetir las sentencias a ejecutar para cada uno, porque cada opción se analiza hasta que aparece la palabra "**break**". Por eso, las opciones '1' hasta '0' hacen lo mismo (todas terminan en el "break" que sigue a '0').

Finalmente, "**default**" indica la acción a realizar si no es ninguno de los casos que se han detallado anteriormente. Esta parte es opcional (si no ponemos la parte de "default", simplemente se sale del "switch" sin hacer nada cuando la opción no sea ninguna de las indicadas). Después de "default" no hace falta poner "break", porque se sabe que ahí acaba la sentencia "switch".

N.

Pasar al tema 7
Volver al índice de temas básicos
Volver al menú principal

CURSO DE C. TEMA 7. Bucles.

Vamos a ver cómo podemos crear partes del programa que **se repitan** un cierto número de veces (bucles).

Según cómo queramos que se controle ese bucle, tenemos tres posibilidades, que básicamene se pueden describir como:
· **for**: La orden se repite desde que una variable tiene un valor inicial hasta que alcanza otro valor final.
· **while**: Repite una sentencia mientras que sea cierta la condición que indicamos. La condición se comprueba antes de realizar la sentencia.
· **do..while**: Igual, pero la condición se comprueba después de realizar la sentencia.

Las diferencias son: "for" normalmente se usa para algo que se repite un número concreto de veces, mientras que "while" se basa en comprobar si una condición es cierta (se repetirá un número indeterminado de veces).

For

El formato de "for" es

```
for (valorInic; CondicRepetic; 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 24:     */
/*   EJ24.C                  */
/*                           */
/*   Escribe del 1 al 10,    */
/*   con "for".              */
/*                           */
/*   Comprobado con:         */
/*      - Turbo C++ 1.01     */
/*      - Djgpp 2.01         */
/*      - Symantec C++ 6.0   */
/*      - GCC 2.6.3          */
/*      - PCC 2.1c           */
/*-------------------------*/

#include <stdio.h>

int contador;

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

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

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

Realmente, la parte de la orden que he 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 ( ; ; )
```

Seguir avanzando en el tema 7
Volver al índice de temas básicos
Volver al menú principal

CURSO DE C. TEMA 7.2 (de 7).

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 25:     */
/*   EJ25.C                  */
/*                           */
/*   Escribe las tablas de   */
/*   multiplicar del 1 al    */
/*   5, con "for".           */
/*                           */
/*   Comprobado con:         */
/*      - Turbo C++ 1.01     */
/*      - Djgpp 2.01         */
/*      - Symantec C++ 6.0   */
/*      - GCC 2.6.3          */
/*      - PCC 2.1c           */
/*-------------------------*/

#include <stdio.h>

int tabla, numero;

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

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

Seguir avanzando en el tema 7
Volver al índice de temas básicos
Volver al menú principal

CURSO DE C. TEMA 7.3 (de 7).

En estos casos después de "for" había un 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 26:     */
/*   EJ26.C                  */
/*                           */
/*   Escribe las tablas de   */
/*   multiplicar del 1 al    */
/*   5, con "for". Deja      */
/*   espacios intermedios.   */
/*                           */
/*   Comprobado con:         */
/*      - Turbo C++ 1.01     */
/*      - Djgpp 2.01         */
/*      - Symantec C++ 6.0   */
/*      - GCC 2.6.3          */
/*      - PCC 2.1c           */
/*-------------------------*/

#include <stdio.h>

int tabla, numero;

main()
{
    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");
        }
}

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

Seguir avanzando en el tema 7
Volver al índice de temas básicos
Volver al menú principal

CURSO DE C. TEMA 7.4 (de 7).

Al igual que en Pascal, para "contar" no necesariamente hay que usar **números**:

```
/*-------------------------*/
/*   Ejemplo en C nº 27:      */
/*   EJ27.C                   */
/*                            */
/*   Escribe las letras de    */
/*   la 'a' a la 'z', con     */
/*   "for".                   */
/*                            */
/*   Comprobado con:          */
/*      - Turbo C++ 1.01      */
/*      - Djgpp 2.01          */
/*      - Symantec C++ 6.0    */
/*      - GCC 2.6.3           */
/*      - PCC 2.1c            */
/*-------------------------*/

#include <stdio.h>

char letra;

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

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

Creo que este programa se explica por sí solo: empezamos en la "a" y terminamos en la "z", aumentando de uno en uno.

Seguir avanzando en el tema 7
Volver al índice de temas básicos
Volver al menú principal

CURSO DE C. TEMA 7.5 (de 7).

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 28:     */
/*   EJ28.C                  */
/*                           */
/*   Escribe las letras de   */
/*   la 'z' a la 'a', (una   */
/*   sí y otra no), con      */
/*   "for".                  */
/*                           */
/*   Comprobado con:         */
/*      - Turbo C++ 1.01     */
/*      - Djgpp 2.01         */
/*      - Symantec C++ 6.0   */
/*      - GCC 2.6.3          */
/*      - PCC 2.1c           */
/*-------------------------*/

#include <stdio.h>

char letra;

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

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

Estos son los casos sencillos de la orden "for", pero también podemos rizar el rizo, y usar dos variables (o más) para controlarlo, o salir de un bucle "for" con otras órdenes, como "break", "continue" y "goto" .

Seguir avanzando en el tema 7 (orden "while")
Volver al índice de temas básicos
Volver al menú principal

CURSO DE C. TEMA 7.5b (de 7). for controlado por dos variables.

(--- Nota: este tema sólo está disponible en la versión ampliada del curso ---).

Seguir avanzando con "for" (orden "continue")
Volver al índice de temas básicos
Volver al menú principal

CURSO DE C. TEMA 7.5c (de 7). continue.

(--- Nota: este tema sólo está disponible en la versión ampliada del curso ---).

Seguir avanzando con "for" (orden "break")
Volver al índice de temas básicos
Volver al menú principal

CURSO DE C. TEMA 7.5d (de 7). break.

(--- Nota: este tema sólo está disponible en la versión ampliada del curso ---).

Seguir avanzando con "for" (orden "goto")
Volver al índice de temas básicos
Volver al menú principal

CURSO DE C. TEMA 7.5e (de 7). goto.

(--- Nota: este tema sólo está disponible en la versión ampliada del curso ---).

Seguir avanzando en el tema 7 (orden "while")
Volver al índice de temas básicos
Volver al menú principal

CURSO DE C. TEMA 7.6 (de 7). While.

Habíamos comentado que "while" podía aparecer en dos tipos de construcciones, 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 se 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 }.

Un ejemplo que nos diga el triple de cada número que tecleemos, y que pare cuando tecleemos el número 0, podría ser:

```
/*-------------------------*/
/*   Ejemplo en C nº 29:      */
/*   EJ29.C                   */
/*                            */
/*   Escribe el triple de     */
/*   los números tecleados, */
/*   con "while"              */
/*                            */
/*   Comprobado con:          */
/*      - Turbo C++ 1.01      */
/*      - Djgpp 2.01          */
/*      - Symantec C++ 6.0    */
/*      - GCC 2.6.3           */
/*      - PCC 2.1c            */
/*-------------------------*/

#include <stdio.h>

int numero;
char frase[60];

main()
{
    printf("Teclea un número (0 para salir): ");
    scanf("%d", &numero);
    while (numero)
    {
        printf("Su triple es %d.\n", numero*3);
        printf("Teclea otro número (0 para salir): ");
        scanf("%d", &numero);
    }
}

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

Y eso de "while (numero)"? Pues es lo mismo que "while (numero != 0)". En caso de duda, toca repasar el tema anterior.

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.

Seguir avanzando en el tema 7
Volver al índice de temas básicos
Volver al menú principal

CURSO DE C. TEMA 7.7 (de 7).

Como ejemplo de la otra aplicación (la condición se comprueba **al final**), vamos a ver cómo sería la típica clave de acceso, pero con una pequeña diferencia: como todavía no sabemos manejar cadenas de texto con una cierta soltura, la clave será un número:

```
/*-------------------------*/
/*   Ejemplo en C nº 30:     */
/*   EJ30.C                  */
/*                           */
/*   Clave de acceso numé-   */
/*   rica, con "do..while"   */
/*                           */
/*   Comprobado con:         */
/*      - Turbo C++ 1.01     */
/*      - Djgpp 2.01         */
/*      - Symantec C++ 6.0   */
/*      - GCC 2.6.3          */
/*      - PCC 2.1c           */
/*-------------------------*/

#include <stdio.h>

int valida = 711;
int clave;

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

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

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.

N.

Pasar al tema 8
Volver al índice de temas básicos
Volver al menú principal

CURSO DE C. TEMA 8. Definición de constantes.

Cuando desarrollamos un programa, nos podemos encontrar con que hay variables que realmente "no varían" a lo largo de la ejecución de un programa, sino que su valor es constante.

Hay una manera especial de definirlas, que es con el especificador "const", que tiene el formato

```
const Nombre = Valor;
```

Así, en el ejemplo anterior (la clave de acceso numérica) habría sido más correcto hacer

```
/*-------------------------*/
/*   Ejemplo en C nº 31:     */
/*   EJ31.C                  */
/*                           */
/*   Clave de acceso numé-   */
/*   rica, con "do..while"   */
/*   y "const"               */
/*                           */
/*   Comprobado con:           */
/*      - Turbo C++ 1.01       */
/*      - Djgpp 2.01           */
/*      - Symantec C++ 6.0     */
/*      - GCC 2.6.3            */
/*-------------------------*/

#include <stdio.h>

const valida = 711;
int clave;

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

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

El ejemplo lo he dejado así para que resulte más familiar a quien venga de Pascal, pero realmente en C la palabra "const" es un **modificador**, algo que da información extra, de modo que su uso habitual sería:

```
const int valida = 711;
```

que se leería algo parecido a "La variable 'válida' es un entero, de valor 711, y este valor deberá permanecer constante".

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 };
```

(Las he escrito en mayúsculas para que se sepa "de un vistazo" que son constantes, no variables. Más adelante comentaré algo

más sobre este convenio.)

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 justo 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
```

En C tenemos 3 tipos de variables, según si su valor se puede modificar o no, y de qué forma:

· Las que habíamos visto hasta ahora, a las que podíamos asignar un valor.
· Las que definimos y no permitiremos que se altere su valor, para lo que usamos el modificador "const".
· Las que pueden cambiar en cualquier momento, y para ellas usaremos el modificador "**volatile**". Es para el caso (poco habitual) de que su valor pueda ser cambiado por algo que no sea nuestro programa principal, y así obligamos al compilador en este caso a que lea el valor de la variable en memoria, en vez de mirarlo en algún registro temporal en el que lo pudiera haber guardado para mayor velocidad.

Seguir avanzando en el tema 8
Volver al índice de temas básicos
Volver al menú principal

CURSO DE C. TEMA 8.2 (de 4).

Hay otra forma muy frecuente en C de definir constantes, que es con la directiva **#define**. Al tratarse de una directiva del preprocesador, es una información que no llega al compilador cuando tiene que traducir a código máquina, sino que ya en una primera pasada se cambia su nombre por el valor correspondiente en cada punto en que aparezca.

Así, algo como

```
#define Valida 711
if (numero == Valida)   [...]
```

se convertiría inmediatamente a

```
if (numero == 711)   [...]
```

que es lo que realmente el compilador traduciría a código máquina. Por eso es frecuente llamarlas "constantes simbólicas".

Así, este último ejemplo quedaría

```
/*-------------------------*/
/*   Ejemplo en C nº 32:     */
/*   EJ32.C                  */
/*                           */
/*   Clave de acceso numé-   */
/*   rica, con "do..while"   */
/*   y "#define"             */
/*                           */
/*   Comprobado con:         */
/*      - Turbo C++ 1.01     */
/*      - Djgpp 2.01         */
/*      - Symantec C++ 6.0   */
/*      - GCC 2.6.3          */
/*      - PCC 2.1c           */
/*-------------------------*/

#include <stdio.h>

#define kVALIDA 711

int clave;

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

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

¿Y por qué lo he puesto en mayúsculas y precedido por una k?

Simplemente por **legibilidad**: en C es inmediato diferenciar una variable de una función, porque, como ya hemos mencionado y veremos de nuevo en el próximo tema, una función necesariamente acaba con paréntesis, incluso aunque no tenga parámetros; en cambio, no hay nada que distinga "a simple vista" una constante de una variable. Por eso, es frecuente "obligar" a que sea fácil

distinguir las constantes de las variables sólo con ojear el listado, y las formas más habituales son escribir las variables en minúsculas y las constantes en mayúsculas, o bien comenzar las constantes con una "k", o bien ambas cosas.

Seguir avanzando en el tema 8
Volver al índice de temas básicos
Volver al menú principal

CURSO DE C. TEMA 8.3 (de 4). 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;" y realmente lo que hacemos es darle un nuevo nombre, lo que nos puede resultar útil por ejemplo si venimos de Pascal (cómo no) y consideramos más legible tener un tipo "boolean". Vamos a verlo directamente con un ejemplo:

```
/*-------------------------*/
/*   Ejemplo en C nº 33:     */
/*   EJ33.C                  */
/*                           */
/*   Clave de acceso numé-   */
/*   rica retocada, con      */
/*   typedef y #define       */
/*                           */
/*   Comprobado con:         */
/*      - Turbo C++ 1.01     */
/*      - Djgpp 2.01         */
/*      - Symantec C++ 6.0   */
/*      - GCC 2.6.3          */
/*      - PCC 2.1c           */
/*-------------------------*/

#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;

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");
}

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

Todo esto se puede hacer más cortito, claro, pero es para que se vea un ejemplo de su uso.

Seguir avanzando en el tema 8
Volver al índice de temas básicos
Volver al menú principal

CURSO DE C. TEMA 8.4 (de 4).

También podemos usar "typedef" para dar un nombre corto a todo un struct:

```
/*-------------------------*/
/*   Ejemplo en C nº 34:     */
/*   EJ34.C                  */
/*                           */
/*   Uso de "typedef" con    */
/*   "struct".               */
/*                           */
/*   Comprobado con:         */
/*      - Turbo C++ 1.01     */
/*      - Djgpp 2.01         */
/*      - Symantec C++ 6.0   */
/*      - GCC 2.6.3          */
/*      - PCC 2.1c           */
/*-------------------------*/

#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 */

main()
{
    ejemplo.valor = 34;
    ejemplo.coste = 1200.5;
    ejemplo.ref = 'A';
    printf("El valor es %d", ejemplo.valor);
}

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

N.

Pasar al tema 9
Volver al índice de temas básicos
Volver al menú principal

CURSO DE C. TEMA 9.

La programación estructurada trata de dividir el programa el bloques más pequeños, buscando una mayor legibilidad, y más comodidad a la hora de corregir o ampliar.

La idea es que un programa que cumpla una cierta misión tendrá que seguir ciertos pasos. Por ejemplo: para resolver un sistema de ecuaciones, los pasos podrían ser:

Pedir Número de Ecuaciones y de Incógnitas.
Para cada ecuación:
- Pedir coeficientes de x(i).
- Pedir término independiente.

Escribir el sistema como matriz.
Hacer ceros bajo la diagonal principal.
Aplicar sustitución regresiva.
Mostrar resultados.

Esos son los pasos que uno piensa cuando se plantea cómo resolver el problema (el "algoritmo"). Nosotros pensamos en "cosas a hacer", no en líneas de programa, por eso la forma más "natural" de trabajar es descomponer el problema en partes, en vez de intentar resolverlo tecleando una línea tras otra.

Es la clásica idea del "divide y vencerás": un programa que hayamos dividido en bloques (con una cierta conexión, no a lo loco) será más fácil de crear y de mantener.

En muchos lenguajes de programación (como Pascal o Basic), estos bloques son de dos tipos: procedimientos ("procedure") y funciones ("function").

La diferencia entre ellos es que un **procedimiento** ejecuta una serie de acciones que están relacionadas entre sí, y no devuelve ningún valor, mientras que la **función** sí que va a devolver valores.

Por ejemplo, un procedimiento podría saludarnos: da unos ciertos pasos, muestra un mensaje en pantalla y ya está.

```
Saluda;
```

Por el contrario una función podría ser la raíz cuadrada: da una serie de pasos, destinados a calcular cual es el valor, pero finalmente tiene que respondernos diciendo qué valor ha obtenido.

```
x = Raiz(9);        (y el ordenador debe "devolverme" el número: 3)
```

En C sólo existen funciones, pero éstas pueden devolver unos valores de tipo "nulo", con lo cual salen a equivaler a un procedimiento (lo veremos algo más adelante).

Vamos a verlo mejor con un par de ejemplos.

Primero crearemos una función "potencia", que eleve un número entero a otro número entero, dando también como resultado un número entero. 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", usaremos la orden "for" para multiplicar tantas veces como haga falta:

```
/*-------------------------*/
/*   Ejemplo en C nº 35:     */
/*   EJ35.C                  */
/*                           */
/*   Función "potencia"      */
```

```
/*                                */
/*   Comprobado con:              */
/*      - Turbo C++ 1.01          */
/*      - Djgpp 2.01              */
/*      - Symantec C++ 6.0        */
/*      - GCC 2.6.3               */
/*--------------------------*/

#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;                 /* Al final de las multiplicaciones,
  }                                        obtengo el valor que buscaba */


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));
}

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

Vamos a comentar cosas sobre este programa:

· base y exponente son dos valores que se "le pasan" a la función: son sus "**parámetros**", unos datos que la función necesita. En nuestro caso, queremos elevar un número a otro, luego deberemos indicarle cual es el número que queremos elevar (la base) y a qué número lo queremos elevar (el exponente).

· El **cuerpo** de la función se indica entre llaves, como cualquier otro bloque de un programa.

· i y temporal son variables **locales** a la función "potencia": no se puede acceder a ellas desde ninguna otra parte del programa. Igual ocurre con num1 y num2: sólo se pueden leer o modificar desde "main". Las variables que habíamos visto hasta ahora no estaban dentro de ninguna función, por lo que eran **globales** a todo el programa. Una detalle importante: las variables locales se deberán declarar **al principio** de la función, antes de que aparezca ninguna orden.

· Con "return" **salimos** de la función e indicamos el valor que queremos que se devuelva. En este caso es lo que habíamos llamado "temporal", que era el valor que íbamos calculando para la potencia en cada paso.

· Recordemos que los "int" en MsDos están limitados a 32767, luego si el resultado es mayor, obtendremos valores erróneos. Esto se puede evitar usando enteros largos o números reales.

Seguir avanzando en el tema 9
Volver al índice de temas básicos
Volver al menú principal

CURSO DE C. TEMA 9.2 (de 6).

Ahora vamos a ver un ejemplo de función que no devuelva ningún valor:

```
/*-------------------------*/
/*   Ejemplo en C nº 36:     */
/*   EJ36.C                  */
/*                           */
/*   Función "hola"          */
/*                           */
/*   Comprobado con:           */
/*      - Turbo C++ 1.01     */
/*      - Djgpp 2.01         */
/*      - Symantec C++ 6.0   */
/*      - GCC 2.6.3          */
/*-------------------------*/

#include <stdio.h>

void saludo(char nombre[])
{
    printf("Hola, %s, cómo estás?", nombre);
}

main()
{
    saludo("Eva");
    printf("\n");
}

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

Y los comentarios de rigor:

· Esta función es de tipo "void" (nulo), con lo que indicamos que no queremos que devuelva ningún valor.

· Con eso de "char nombre[]" indicamos que le vamos a pasar una cadena de caracteres, pero no hace falta que digamos su tamaño, como hacíamos cuando las declarábamos. Así podremos pasar cadenas de caracteres tan grandes (o tan pequeñas) como queramos.

· Esta función **no** termina en "return", porque no queremos que devuelva ningún valor. Aun así, sí que podríamos haber escrito

```
void saludo(char nombre[])
{
    printf("Hola, %s, cómo estás?", nombre);
    return;
}
```

pero la diferencia con respecto a la función "potencia" está en que en este caso hemos dejado el "return" sólo para que se vea dónde termina la función, y no indicamos nada como "return valor", dado que no hay ningún valor que devolver.

Este tipo de funciones, que no devuelven ningún valor, son el equivalente de los "procedures" del lenguaje Pascal.

Seguir avanzando en el tema 9
Volver al índice de temas básicos
Volver al menú principal

CURSO DE C. TEMA 9.3 (de 6).

Recordemos en este punto que "main" es una función, de tipo entero (se considera así si no se indica el tipo, que es lo que estábamos haciendo hasta ahora), por lo que la forma más correcta de escribir el ejemplo de la potencia sería:

```
/*-------------------------*/
/*   Ejemplo en C nº 35 (b) */
/*   EJ35B.C                */
/*                          */
/*   Función "potencia";    */
/*   función "main"         */
/*                          */
/*   Comprobado con:        */
/*      - Turbo C++ 1.01    */
/*      - Djgpp 2.01        */
/*      - Symantec C++ 6.0  */
/*      - GCC 2.6.3         */
/*-------------------------*/

#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++)
       temporal *= base;

    return temporal;
}

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;
}

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

Hasta ahora no habíamos puesto eso de "int main()" porque en cuanto el compilador de C ve eso de (), sabe que se trata de una función, y si no le indicamos cómo es el valor que va a devolver, considera que se trata de un número entero. Por eso, cuando creamos la función "potencia", también podríamos haber escrito:

Seguir avanzando en el tema 9
Volver al índice de temas básicos
Volver al menú principal

CURSO DE C. TEMA 9.4 (de 6).

Vamos a ver ahora cómo **modificar** el valor de un parámetro de una función. Vamos a empezar por un ejemplo típico:

```
/*-------------------------*/
/*   Ejemplo en C nº 37:      */
/*   EJ37.C                   */
/*                            */
/*   Función "modif1"         */
/*   Parámetros por valor.    */
/*                            */
/*   Comprobado con:          */
/*      - Turbo C++ 1.01      */
/*      - Djgpp 2.01          */
/*      - Symantec C++ 6.0    */
/*      - GCC 2.6.3           */
/*-------------------------*/

#include <stdio.h>

void modif1(int v)
{
    v ++;
    printf("Ahora vale: %d\n", v);
}

main()
{
    int valor = 2;
    printf("Ahora vale: %d\n", valor);
    modif1(valor);
    printf("Ahora vale: %d\n", valor);
}

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

Sigamos cada paso que vamos dando

· Incluimos "stdio.h" para poder usar "printf".
· La función "modif1" no devuelve ningún valor. Se le pasa una variable de tipo entero, aumenta su valor en uno y lo escribe.
· Empieza el cuerpo del programa: valor = 2.
· Lo escribimos. Claramente, será 2.
· Llamamos al procedimiento "modif1", que asigna el valor=3 y lo escribe.
· Finalmente volvemos a escribir valor... 3?

¡Pues no! Escribe un 2. Las modificaciones que hagamos a "dato" dentro de la función "modif1" sólo son válidas mientras estemos dentro de esa función. Esto es debido a que realmente no modificamos "dato", sino una **copia** que se hace de su valor.

Seguir avanzando en el tema 9
Volver al índice de temas básicos
Volver al menú principal

CURSO DE C. TEMA 9.5 (de 6).

Eso es pasar un parámetro **por valor**. Pero, cómo lo hacemos si realmente queremos modificar el parámetro? Eso sería pasar un parámetro **por referencia**, y en C se hace pasando a la función la dirección de memoria en la que está el valor. Va un ejemplo, que comentaré después:

```
/*-------------------------*/
/*   Ejemplo en C nº 38:     */
/*   EJ38.C                  */
/*                           */
/*   Función "modif2"        */
/*   Parámetros por ref.     */
/*                           */
/*   Comprobado con:         */
/*      - Turbo C++ 1.01     */
/*      - Djgpp 2.01         */
/*      - Symantec C++ 6.0   */
/*      - GCC 2.6.3          */
/*-------------------------*/

#include <stdio.h>

void modif2(int *v)
{
    (*v) ++;
    printf("Ahora vale: %d\n", *v);
}


main()
{
    int valor = 2;
    printf("Ahora vale: %d\n", valor);
    modif2( &valor );
    printf("Ahora vale: %d\n", valor);
}

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

Este programa resulta bastante menos legible, pero al menos funciona, que es de lo que se trata. Vamos a ir comentando las cosas raras que aparecen:

· El parámetro lo pasamos como "int *v". Eso quiere decir que v es un "**puntero** a entero", o, en un castellano más normal, "una dirección de memoria en la que habrá un dato que es un número entero" (veremos los punteros con más detalle en el próximo tema). De ahí viene eso de "paso **por referencia**": no le decimos a la función el valor de la variable, sino que le damos referencias sobre ella, le decimos **dónde se encuentra**.

· Para modificarlo no hacemos "v++" sino "(*v) ++". Recordemos que ahora v es una dirección de memoria, que no es lo que queremos incrementar. Queremos aumentar el valor que hay **en esa dirección**. A ese valor se accede como "*v". Por tanto, lo aumentaremos con (*v)++

· A la hora de llamar a la función también tenemos que llevar cuidado, porque ésta espera que le digamos una **dirección** de memoria de la que leerá el dato. Eso es lo que hace el operador "&", de modo que "&valor" se leería como "la dirección de memoria en la que se encuentra la variable valor".

Pues con este jaleo de pasar una dirección de memoria y modificar el contenido de esa dirección de memoria sí que podemos variar el valor de un parámetro desde dentro de una función.

¿Y para qué vamos a querer hacer eso? Pues el ejemplo más claro es cuando queremos que una función nos devuelva más de un valor. Otro uso de los pasos por referencia es cuando tratamos de optimizar velocidad al máximo, dado que cuando pasamos un dato por valor, se crea una copia suya (ver los comentarios al ejemplo 37), cosa que no ocurre al pasar ese parámetro por referencia.

Seguir avanzando en el tema 9
Volver al índice de temas básicos
Volver al menú principal

CURSO DE C. TEMA 9.6 (de 6).

En el próximo tema volveremos a los punteros con más detalle. Ahora vamos a comentar qué es la **recursividad**, para dar este tema por finalizado:

La idea es simplemente que una función recursiva es aquella que se llama a sí misma. El ejemplo clásico es el "factorial de un número":

Partimos de la definición de factorial:
n! = n (n-1) (n-2) ... 3 2 1

Por otra parte,
(n-1)! = (n-1) (n-2) (n-3) ... 3 2 1

Luego podemos escribir cada factorial en función del factorial del siguiente número:
n! = n (n-1)!

Pues esta es la definición recursiva del factorial, ni más ni menos. Esto, programando, se haría:

```
/*--------------------------*/
/*   Ejemplo en C nº 39:      */
/*   EJ39.C                   */
/*                            */
/*   "factorial" recursivo    */
/*                            */
/*   Comprobado con:          */
/*      - Turbo C++ 1.01      */
/*      - Djgpp 2.01          */
/*      - Symantec C++ 6.0    */
/*      - GCC 2.6.3           */
/*--------------------------*/

#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 */
}

main()
{
    int num;
    printf("Introduzca un número entero: ");
    scanf("%d", &num);
    printf("Su factorial es: %ld\n", fact(num));
}

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

Las dos consideraciones de siempre:

- Atención a la primera parte de la función recursiva: es MUY IMPORTANTE comprobar que hay **salida** de la función, para que no se quede dando vueltas todo el tiempo y nos cuelgue el ordenador.
- Los factoriales **crecen rápidamente**, así que no conviene poner números grandes: el factorial de 16 es 2004189184, luego a partir de 17 empezaremos a obtener resultados erróneos, a pesar de haber usado enteros largos.

N.

Pasar al tema 10
Volver al índice de temas básicos

Volver al menú principal

CURSO DE C. TEMA 10. Variables dinámicas.

(**Nota** antes de empezar: este es un tema denso, y que a mucha gente le asusta al principio. Puede ser necesario leerlo varias veces y experimentar bastante para poder sacarle todo el jugo).

Hasta ahora teníamos una serie de variables que declaramos al principio del programa o de cada función. Estas variables, que reciben el nombre de **ESTATICAS**, 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 **DINAMICAS**, que puedan ir creciendo o disminuyendo según nos interesen. Ejemplos de este tipo de estructuras son:

- Las **pilas**. Justo como una pila de libros: vamos apilando cosas en la cima, o cogiendo de la cima.
- Las **colas**. Como las del cine, por ejemplo (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 (si hace falta, para que no "se rompa") y liberamos la memoria que estaba ocupando.

Así que para seguir, necesitamos saber cómo reservar memoria y cómo liberarla. Antes, vamos a ver algo que ya se comentó de pasada en el tema anterior: eso de 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 asociado: por ejemplo, un "puntero a entero" será una dirección de memoria en la que habrá almacenado (o podremos almacenar) un número entero.

Ahora vamos a ver qué **símbolos** 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)

num = 1;          ahora "num" vale 1
pos = 1000;       "pos" ahora es la dirección 1000 (peligroso)
*pos = 25;        en la posición "pos" guardamos un 25
pos = &num;       "pos" ahora es la dirección de "num"
```

Por tanto, con el símbolo * indicamos que se trata de un puntero, y & nos devuelve la dirección de memoria en la que se encuentra una variable.

Lo de "pos=1000" es peligroso, porque no sabemos qué hay en esa dirección, de modo que si escribimos allí podemos provocar una catástrofe. Por ejemplo, si ponemos un valor al azar que coincide con la instrucción en código máquina de formatear el disco duro, no nos hará nada de gracia cuando nuestro programa llegue hasta esa instrucción. Normalmente las consecuencias no son tan graves, pero hay que llevar cuidado. La forma de trabajar será **pedir** al compilador que nos reserve un poco de memoria donde él crea adecuado.

Para eso usamos la orden "malloc" (necesitaremos incluir el fichero de cabecera <stdlib.h> o <malloc.h> -es preferible usar el primero, que es estándar-). Una vez que hemos terminado de usar esa memoria, suele ser conveniente **liberarla**, y para eso empleamos "free". Vamos a ver un ejemplo:

```
/*-------------------------*/
/*   Ejemplo en C nº 40:     */
/*   EJ40.C                  */
/*                           */
/*   Introducción a los      */
/*   punteros.               */
/*                           */
/*   Comprobado con:         */
/*      - Turbo C++ 1.01     */
/*      - Djgpp 2.01         */
/*      - Symantec C++ 6.0   */
/*      - GCC 2.6.3          */
/*-------------------------*/

#include <stdio.h>
#include <stdlib.h>

int num;                /* Un número entero */
int *pos;               /* Un "puntero a entero" */

main()
{
    printf("Num vale: %d (arbitrario)\n", num);
    printf("La dirección Pos es: %p (arbitrario)\n", pos);

    /* La siguiente línea está deshabilitada porque es "peligrosa"
          (en GCC dará error) */
    /* printf("El contenido de Pos es: %d (arbitrario)\n", *pos); */

    num = 1;                /* ahora "num" vale 1 */
    printf("Num vale: %d (fijado)\n", num);

                            /* Reservamos espacio */
    pos = (int *) malloc (sizeof(int));

    printf("La dirección Pos es: %p (asignado)\n", pos);
    printf("El contenido de Pos es: %d (arbitrario)\n", *pos);

    *pos = 25;              /*   En la posición "pos" guardamos un 25 */
    printf("El contenido de Pos es: %d (fijado)\n", *pos);

    free(pos);              /* Liberamos lo reservado */

    pos = &num;             /*   "pos" ahora es la dirección de "num" */
    printf("Y ahora el contenido de Pos es: %d (valor de Num)\n", *pos);
}

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

El **formato** de "free" no tiene ninguna pega: "free(pos)" quiere decir "libera la memoria que ocupaba pos". El de "malloc" es más rebuscado:

malloc (tamaño)

Como queremos reservar espacio para un entero, ese "tamaño" será lo que ocupe un entero, y eso nos lo dice "sizeof(int)".

¿Y eso de (int*)? Es porque "malloc" nos devuelve un puntero sin tipo (un puntero a void: void*). Cómo queremos guardar un dato entero, primero debemos hacer una **conversión** de tipos (typecast), de "puntero sin tipo" a "puntero a entero" (int *).

De principio puede asustar un poco, pero se le pierde el miedo en cuanto se usa un par de veces. Vamos a analizar ahora la salida de este programa.

Con Turbo C++ 1.01 obtenemos:

```
Num vale: 0 (arbitrario)
La dirección Pos es: 0000:0000 (arbitrario)
El contenido de Pos es: 294 (arbitrario)
Num vale: 1 (fijado)
La dirección Pos es: 0000:0004 (asignado)
El contenido de Pos es: 1780 (arbitrario)
El contenido de Pos es: 25 (fijado)
Y ahora el contenido de Pos es: 1 (valor de Num)
```

Y con Symantec C++ 6.0

```
Num vale: 0 (arbitrario)
La dirección Pos es: 0000 (arbitrario)
El contenido de Pos es: 21061 (arbitrario)
Num vale: 1 (fijado)
La dirección Pos es: 2672 (asignado)
El contenido de Pos es: -9856 (arbitrario)
El contenido de Pos es: 25 (fijado)
Y ahora el contenido de Pos es: 1 (valor de Num)
```

En ambos casos vemos que

- Inicialmente "num" vale 0, pero como no es algo que hayamos obligado nosotros, no podemos fiarnos de que siempre vaya a ser así.

- El puntero POS es 0 al principio (no apunta todavía a ninguna dirección). Esto es un "**puntero nulo**", para el que todavía no se ha asignado un espacio en memoria. De hecho, para lograr una mayor legibilidad, está definida (en "stdio.h" y en otras cabeceras) una constante llamada **NULL** y de valor 0, de modo que podemos hacer comparaciones como

  ```
  if (pos == NULL)   ...
  ```

  Recordemos que `if (pos != NULL)` es igual que `if (pos) ...` (ya visto en el tema de condiciones).

- Como esta dirección ("pos") no está reservada aún, su valor puede ser cualquier cosa. Y de hecho, puede seguirlo siendo después de reservarla: el compilador nos dice dónde tenemos memoria disponible, pero no "la vacía" para nosotros.

- Una vez que hemos reservado memoria y hemos asignado el valor, ya sabemos con certeza que ese número 25 se guardará donde debe.

- Finalmente, si queremos asignar a "pos" el valor de la dirección en la que se encuentra "num", como hemos hecho en la penúltima línea, no hace falta reservar memoria con "malloc" (de hecho, lo he usado a propósito después de liberar la memoria con "free"). Esto es debido a que "num" ya tenía reservado su espacio de memoria, al que nosotros ahora podemos acceder de dos formas: sabiendo que corresponde a la variable "num", o bien teniendo en cuenta que "pos" es la dirección en la que está ese dato.

Finalmente, con GCC la siguiente línea da problemas:

```
printf("El contenido de Pos es: %d (arbitrario)\n", *pos);
```

Estamos intentando acceder a un espacio de memoria que **no hemos reservado**, lo cual no es posible bajo Linux. Esto hace que

en el momento de ejecutar el programa, aparezca un error de "Segmentation Fault". En cambio, los dos compiladores que trabajan bajo DOS aceptan esta orden. Aun así, insisto en que no se debe modificar el contenido de zonas de memoria que no hemos reservado.

Seguir avanzando en el tema 10
Volver al índice de temas básicos
Volver al menú principal

CURSO DE C. TEMA 10.2 (de 4). Aritmética de punteros.

Aquí hay que hacer una observación: como un puntero tiene un valor numérico (la dirección en la que se nos ha reservado memoria), podemos aumentar este número haciendo cosas como

```
pos ++;          o bien    pos += 3;
```

Pero también hay que recordar que normalmente un puntero va a estar asociado a un cierto tipo de datos. Por ejemplo, "int *" indica un puntero a entero.

¿A qué viene esto? Es sencillo: si un entero ocupa 2 bytes, al hacer "pos++" no deberíamos avanzar de 1 en 1, sino de 2 en 2, para saltar al siguiente dato.

Vamos a ver un ejemplo que cree un array de enteros dinámicamente, y que **lo recorra** usando los índices del array (como habríamos hecho hasta ahora) y también usando punteros:

```
/*-------------------------*/
/*   Ejemplo en C nº 41:   */
/*   EJ41.C                */
/*                         */
/*   Punteros y arrays     */
/*                         */
/*   Comprobado con:       */
/*     - Turbo C++ 1.01    */
/*     - Djgpp 2.01        */
/*     - Symantec C++ 6.0  */
/*     - GCC 2.6.3         */
/*-------------------------*/

#include <stdio.h>
#include <stdlib.h>

int *num;           /* Puntero a número(s) entero(s) */
int *temporal;      /* Temporal, para recorrer el array */
int i;              /* Para bucles */

main()
{
        /* Reservamos espacio para 10 números (array dinámico) */
    num = (int *) malloc (10 * sizeof(int));

    for (i=0; i<10; i++)      /* Recorremos el array */
      num[i] = i*2;           /* Dando valores */

    printf("La dirección de comienzo del array es: %p\n", num);

    printf("Valores del array: ");
    for (i=0; i<10; i++)      /* Recorremos el array */
      printf("%d ",num[i]);   /* Mostrando los valores */


    printf("\nValores del array (como puntero): ");
    temporal=num;
    for (i=0; i<10; i++)      /* Recorremos como puntero */
      printf("%d ",*temporal++);  /* Mostrando los valores y aumentando */

    free(num);                /* Liberamos lo reservado */

}
```

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

Como se ve, en C hay muy poca diferencia entre arrays y punteros: hemos declarado "num" como un puntero, pero hemos reservado espacio para más de un dato, y hemos podido recorrerlo como si hubiésemos definido

```
int num[10];
```

Esto es lo que da lugar a la gran diferencia que existe en el manejo de "strings" (cadenas de texto) en lenguajes como Pascal frente al C, por lo que he preferido dedicarles un tema entero (se verá más adelante).

Se puede profundizar mucho más en los punteros, especialmente tratando estructuras dinámicas como las pilas y colas, las listas simples y dobles, los árboles, etc., pero considero que cae fuera del propósito de este curso, que es básicamente introductorio (sí se tratan brevemente en la versión ampliada).

N.

Pasar al tema 11
Volver al índice de temas básicos
Volver al menú principal

CURSO DE C. TEMA 10.3 (de 4). Listas.

(--- Nota: este tema sólo está disponible en la versión ampliada del curso ---).

Pasar a los árboles binarios
Pasar al tema 11
Volver al índice de temas básicos
Volver al menú principal

CURSO DE C. TEMA 10.4 (de 4). Árboles.

(--- Nota: este tema sólo está disponible en la versión ampliada del curso ---).

CURSO DE C. TEMA 11. Manejo de ficheros.

Vamos a comenzar por ver cómo leer un fichero de texto. Primero voy a poner un ejemplo que lea y muestre el AUTOEXEC.BAT, y después lo iré comentando:

```
/*-------------------------*/
/*   Ejemplo en C nº 42:     */
/*   EJ42.C                  */
/*                           */
/*   Ficheros de texto (1)   */
/*                           */
/*   Comprobado con:         */
/*      - Turbo C++ 1.01     */
/*      - Djgpp 2.01         */
/*      - Symantec C++ 6.0   */
/*      - GCC 2.6.3          */
/*-------------------------*/

#include <stdio.h>

FILE* fichero;
char texto[80];

main()
{
   fichero = fopen("c:\\autoexec.bat", "rt");
   /* En Linux: fichero = fopen("/proc/cpuinfo", "rt");   */
   if (fichero == NULL)
   {
      printf("No existe el fichero!\n");
      exit(1);
   }
   while (! feof(fichero)) {
      fgets(texto, 80, fichero);
      printf("%s", texto);
   }
   fclose(fichero);
}

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

Aquí van los comentarios sobre este programa:

- **FILE** es el tipo asociado a un fichero. Realmente se trata de un "puntero a fichero", por eso aparece el asterisco * a su derecha.
- Para abrir el fichero usamos "fopen", que lleva dos parámetros: el nombre del fichero y el modo de lectura. En el nombre del fichero, la **barra** \ aparece repetida a propósito, porque (como vimos al hablar de "printf") es un código de control, de modo que \a sería la señal de alerta (un pitido), que no es lo que queremos leer. Por eso, ponemos \\, que se traduce como una sola barra. Lo de "rt" indica que el modo será de lectura (r) en un fichero de texto (t). Si usamos GCC bajo Linux, no existe "c:\autoexec.bat", así que podemos leer "/proc/cpuinfo", o "/proc/devices", o ".profile", por ejemplo.
- Como "fichero" es un puntero (a fichero), para mirar si ha habido algún problema, comprobamos si ese puntero sigue siendo **nulo** después de intentar acceder al fichero.
- Después repetimos una parte del programa hasta que **se acabe** el fichero, de lo que nos informa "feof" (EOF es la abreviatura de End Of File, fin de fichero).
- Con "fgets" leemos una cadena de texto, que podemos limitar en longitud (en este caso, a 80 caracteres), desde el fichero. Esta cadena de texto conservará los caracteres de avance de línea.
- Finalmente, **cerramos** el fichero con "fclose".

Seguir avanzando en el tema 11
Volver al índice de temas básicos

Volver al menú principal

CURSO DE C. TEMA 11.2 (de 4).

Si queremos **crear un fichero**, los pasos son muy parecidos, sólo que lo abriremos para escritura (w), y escribiremos con "fputs":

```
/*-------------------------*/
/*   Ejemplo en C nº 43:     */
/*   EJ43.C                  */
/*                           */
/*   Ficheros de texto (2)   */
/*                           */
/*   Comprobado con:         */
/*      - Turbo C++ 1.01     */
/*      - Djgpp 2.01         */
/*      - Symantec C++ 6.0   */
/*      - GCC 2.6.3          */
/*      - PCC 2.1c           */
/*-------------------------*/

#include <stdio.h>

FILE* fichero;
char texto[80];

main()
{
   fichero = fopen("basura.dat", "wt");
   if (fichero == NULL)
   {
      printf("No se ha podido crear el fichero!\n");
      exit(1);
   }
   fputs("Esto es una línea\n", fichero);
   fputs("Esto es otra", fichero);
   fputs(" y esto es continuación de la anterior\n", fichero);
   fclose(fichero);
}

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

Seguir avanzando en el tema 11
Volver al índice de temas básicos
Volver al menú principal

CURSO DE C. TEMA 11.3 (de 4).

Antes de seguir, vamos a ver las letras que pueden aparecer en el modo de apertura del fichero:

Tipo	Significado
r	Abrir sólo para lectura.
w	Crear para escribir. Sobreescribe si existiera ya.
a	Añade al final si existe, o crea si no existe.
+	Permite modificar. Por ejemplo: r+
t	Abrir en modo de texto.
b	Abrir en modo binario.

Si queremos leer o escribir sólo **una letra**, tenemos las órdenes "fgetc" y "fputc", que se usan:

```
letra = fgetc( fichero );
fputc (letra, fichero);
```

Podemos querer crear un "fichero **con tipo**", en el que todos los componentes vayan a ser del mismo tipo. Por ejemplo, podría ser para un agenda, en la que guardemos los datos de cada persona con un "struct".

En casos como éste, la solución más cómoda puede ser usar "fprintf" y "fscanf", análogos a "printf" y "scanf", que se emplearían así:

```
fprintf( fichero, "%40s%5d\n", persona.nombre, persona.numero);
fscanf( fichero, "%40s%5d\n", &persona.nombre, &persona.numero);
```

Seguir avanzando en el tema 11
Volver al índice de temas básicos
Volver al menú principal

CURSO DE C. TEMA 11.4 (de 4).

Finalmente, podemos crear ficheros "**sin tipo**", es decir, que la información que contengan no necesariamente sea sólo texto ni tampoco datos siempre iguales.

En este caso, utilizamos "fread" y "fwrite" (análogos a BlockRead y BlockWrite, para quien venga de Pascal). Los datos que se leen se van guardando 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.

Vamos a ver un ejemplo, que comentaré después:

```
/*-------------------------*/
/*   Ejemplo en C nº 44:     */
/*   EJ44.C                  */
/*                           */
/*   Ficheros sin tipo:      */
/*   Copiador elemental      */
/*                           */
/*   Comprobado con:         */
/*      - Turbo C++ 1.01     */
/*      - Djgpp 2.01         */
/*      - Symantec C++ 6.0   */
/*      - GCC 2.6.3          */
/*-------------------------*/

#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 */


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);
    }
   /* Cierro los ficheros */
   fclose(fichOrg);
   fclose(fichDest);
}

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

Las novedades en este programa son:

· Defino un buffer de 2048 bytes (2 K), en el que iré guardando lo que lea.

· En la misma línea intento abrir el fichero y compruebo si todo ha sido correcto. Es menos legible, pero más compacto, y, sobre todo, muy frecuente encontrarlo en "fuentes ajenos" de esos que circulan por ahí, de modo que he considerado adecuado incluirlo.

· A "fread" le digo que quiero leer 2048 datos de 1 byte cada uno, y él me devuelve la cantidad de bytes que ha leído realmente. Cuando sea menos de 2048 bytes, es que el fichero se ha acabado.

· A "fwrite" le indico el número de bytes que quiero que escriba.

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 posic, 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 abiertos el fichero).

· "fichero" indica el fichero dentro de el que queremos saltar. Este fichero debe estar abierto previamente (con fopen).

· "posic" 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 **constantes**:

SEEK_SET (0): Principio
SEEK_CUR (1): Actual
SEEK_END (2): Final

Finalmente, si queremos saber en **qué posición** de un fichero nos encontramos, podemos usar "ftell(fichero)".

Pues esto es el manejo de ficheros en C. Ahora sólo queda elegir un proyecto en el que aplicarlos, y ponerse con ello.

N.

Pasar al tema 12
Volver al índice de temas básicos
Volver al menú principal

CURSO DE C. TEMA 12. Cadenas de texto.

El que este tema se encuentre al final no quiere decir que sea más difícil que los anteriores. De hecho, es más fácil que los ficheros, y desde luego mucho más fácil que dominar los punteros. Se trata simplemente de separarlo un poco del resto, porque he visto que mucha gente que viene de programar en lenguajes como Pascal tiende a hacer cosas como estas:

```
texto = "Hola";
```

Es una forma muy legible de dar una valor a una cadena de texto... en otros lenguajes! En C **no se puede** hacer así. Por eso he puesto este tema separado: sólo para recalcar que ciertas cosas no se pueden hacer.

Vamos a empezar por lo fundamental, que no se debe olvidar:

Una **cadena de texto** en C no es más que un array de caracteres. Como a todo array, se le puede reservar espacio estáticamente o dinámicamente. Ya lo vimos en el tema de punteros, pero insisto: estáticamente es haciendo cosas como

```
char texto[80];
```

y dinámicamente es declarándola como puntero:

```
char *texto;
```

y reservando memoria con "malloc" cuando nos haga falta.

En cualquier caso, una cadena de caracteres siempre terminará con un **carácter nulo** (\0). Por eso, si necesitamos 7 letras para un teléfono, deberemos hacer

```
char telefono[8];
```

dado que hay que almacenar esas 7 y después un \0. Si sabemos lo que hacemos, podemos reservar sólo esas 7, pero tendremos que usar nuestras propias funciones, porque las que nos ofrece el lenguaje C se apoyan todas en que al final debe existir ese carácter nulo.

Ahora vamos ya con lo que es el manejo de las cadenas:

Para **copiar** el valor de una cadena de texto en otra, no podemos hacer texto1 = texto2; porque estaríamos igualando dos punteros. A partir de este momento, las dos cadenas se encontrarían en la misma posición de memoria, y los cambios que hiciéramos en una se reflejarían también en la otra.

En vez de eso, debemos usar una función de biblioteca, "strcpy" (string copy), que se encuentra, como todas las que veremos, en "string.h":

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

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.

Si queremos copiar sólo los primeros **n bytes** de origen, usamos

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

Para **añadir** una cadena al final de otra (concatenarla), usamos

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

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

strcmp (cad1, cad2);

Esta función devuelve un número entero, que será: - 0 si ambas cadenas son iguales. - negativo, si cad1 < cad2. - positivo, si cad1 > cad2.

Según el compilador que usemos, tenemos incluso funciones ya preparadas para convertir una cadena a **mayúsculas**: strupr (esa línea del siguiente ejemplo funciona en TC y SC, pero no en GCC).

Estas son las principales posibilidades, aunque hay muchas más funciones (quien tenga curiosidad puede mirar la ayuda sobre "string.h" en su compilador favorito). Vamos a aplicarlas a un ejemplo:

```
/*-------------------------*/
/*   Ejemplo en C nº 45:     */
/*   EJ45.C                  */
/*                           */
/*   Cadenas de texto        */
/*                           */
/*   Comprobado con:         */
/*      - Turbo C++ 1.01     */
/*      - Djgpp 2.01         */
/*      - Symantec C++ 6.0   */
/*      - GCC 2.6.3          */
/*-------------------------*/

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

char texto1[80] = "Hola";              /* Cadena estática */
char *texto2;                          /* Cadena dinámica */


main()
{ /* Reservo espacio para la cadena dinámica */
   texto2 = (char *) malloc (70* sizeof(char));

   strcpy(texto2, "Adios");         /* Le doy un valor */
   puts(texto1); puts(texto2);     /* Escribo las dos */

   strncpy(texto1, texto2, 3);     /* Copio las 3 primeras letras */
   puts(texto1);                     /* Escribo la primera -> Adia */

   strcat(texto1, texto2);          /* Añado texto2 al final */
   puts(texto1);                     /* Escribo la primera -> AdiaAdios */

            /* Comparo alfabéticamente */
   printf("Si las comparamos obtenemos: %d",
      strcmp(texto1, texto2));
   printf(" (Número negativo: texto1 es menor)\n");

            /* Longitud */
   printf("La longitud de la primera es %d\n", strlen(texto1));

            /* Mayúsculas -No en GCC- */
   printf("En mayúsculas es %s.\n",
      strupr(texto1));

   free(texto2);              /* Libero lo reservado */
}

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

N.

Ver las notas finales sobre el curso
Volver al índice de temas básicos
Volver al menú principal

Tema 0. Sobre el curso. Cuenta cómo nació la idea del curso y poco más.
Tema 1. Generalidades. Una primera toma de contacto con el lenguaje C.
Tema 2. Introd. a variables Introducción a las variables: cómo se definen y se usan.
Tema 3. Recapitulación. Un poco más detalle sobre las "cosas raras" que se han pasado un poco por alto en el tema anterior, como el #include.
Tema 4. Entrada/Salida básica. Volvemos a "printf" con algo más de detalle, y vemos cómo aceptar la introducción de datos por parte del usuario.
Tema 5. Operaciones matemáticas. Cómo hacer las operaciones matemáticas más habituales (sumar, restar, multiplicar, dividir, etc.).
Tema 6. Condiciones. Formas de comprobar condiciones: "si ocurre esto, haz aquello; en caso contrario haz esto otro".
Tema 7. Bucles. Para permitir que una parte de un programa se repita un cierto número de veces, o hasta que se cumpla una condición, o mientras que se cumpla una condición.
Tema 8. Constantes y tipos. Mejoremos eso de las variables: qué ocurre cuando realmente no va a variar o cuando no nos basta con los tipos de datos que existen en el C.
Tema 9. Funciones. Cómo crear nuestras propias rutinas, para que los programas sean más fáciles de leer y de modificar/ampliar.
Tema 10. Punteros. Cómo reservar memoria dinámicamente (sólo en el momento en que el programa la necesite)
Tema 11. Ficheros. Para acceder a ficheros: como leer su contenido, grabar en ellos o crear nuestros propios ficheros.
Tema 12. Cadenas de texto. Las cadenas de texto son más difíciles de manejar en C que en otros lenguajes como Pascal o Basic. Por eso, les dedicamos un tema aparte.

CURSO DE C. INDICE DE EJEMPLOS.

Total de ejemplos en esta versión: 82.

Los ejemplos básicos (51) que acompañan a este curso son:

01: Programa elemental (Escribe "Hola", tema 1)
02: Introd. a variables (suma de dos números, tema 2)
03: Inicialización de variables (casi igual que el anterior, tema 2.2)
04: Escribir más de una variable (algo más de detalle sobre printf, tema 2.3)
05: Leer valores para variables (ejemplo de scanf, tema 2.4)
06: Primer ejemplo de arrays (suma de 5 números enteros, tema 2.5)
07: Inicialización de arrays (basado en anterior, tema 2.6)
08: Registros (struct, tema 2.7)
09: Introducción a las cadenas de texto (inicializar, leer y mostrar, tema 2.8)
10: "main" como int, tema 3
11: "main" como void, tema 3.2
12: Formatos con "printf" (varios ejemplos, tema 4.2)
13: "gets" y "puts" (cadenas de texto, tema 4.4)
14: Operaciones con números enteros y reales, tema 5
14b: Operaciones de bits con números enteros, tema 5.3
15: "if" elemental, tema 6
16: "if" abreviado, tema 6.2
17: "if" y sentencias compuestas, tema 6.3
18: "if" y "else", tema 6.4
19: "if..else" encadenados, tema 6.5
20: "if" incorrecto, tema 6.6
21: Operador condicional ?, tema 6.7
22: Problemas de "getchar", tema 6.8
23: Uso de "switch", tema 6.9
24: Escribe del 1 al 10, con "for", tema 7
25: Escribe las tablas de multiplicar del 1 al 5, con "for", tema 7.2
26: Tablas de multiplicar del 1 al 5, con "for". Deja espacios intermedios. tema 7.3
27: Escribe las letras de la 'a' a la 'z', con "for",tema 7.4
28: Escribe las letras de la 'z' a la 'a', (una sí y otra no), con "for" tema 7.5
28b: Ejemplo de "for" controlado por 2 variables, tema 7.5b
28c: Saltar pasos de un "for" con "continue", tema 7.5c
28d: Salir de un "for" con "break", tema 7.5d
28e: Salir de varios "for" con "goto", tema 7.5e
29: Escribe el triple de los números tecleados, con "while", tema 7.6
30: Clave de acceso numérica, con "do..while", tema 7.7
31: Clave de acceso numérica, con "do..while" y "const", tema 8
32: Clave de acceso numérica, con "do..while" y "#define", tema 8.2
33: Clave de acceso numérica retocada, con typedef y #define, tema 8.3
34: Uso de "typedef" con "struct", tema 8.4
35: Función "potencia", tema 9
35b: Función "potencia"; función "main", tema 9.2
36: Función "hola", tema 9.3
37: Función "modif1" Parámetros por valor, tema 9.4
38: Función "modif2" Parámetros por ref, tema 9.5
39: "factorial" recursivo, tema 9.6
40: Introducción a los punteros, tema 10
41: Punteros y arrays, tema 10.2
42: Ficheros de texto (1), tema 11
43: Ficheros de texto (2), tema 11.2
44: Ficheros sin tipo: Copiador elemental, tema 11.4
45: Cadenas de texto, tema 12

Otros sin numerar entre los temas básicos (3):

- Primer ejemplo de listas simples.
- Segundo ejemplo de listas simples.
- Arboles binarios de búsqueda.

Entre las ampliaciones del curso (15):

- Pantalla de texto en Turbo C.
- Pantalla de texto en Symantec C++.
- "conio.h" para Symantec C++.
- Pantalla de texto en Symantec C++ al estilo de Turbo C.
- Pantalla de texto en Linux.
- Pantalla gráfica en Turbo C.
- Programa elemental de dibujo con Turbo C.
- Pantalla gráfica accediendo a memoria.
- Servicios del DOS en Turbo C.
- Interrupciones del DOS en Turbo C.
- Contenido de un directorio en Turbo C.
- Imprimir desde C.
- Prototipos de funciones.
- Ver la tabla ASCII.
- Acceder al lenguaje ensamblador en C.

Y como programas algo más completos (7):

- Adivinar un número.
- Adivinar un número (2).
- Mostrar un fichero de texto.
- Comparar dos ficheros.
- Figuras que se mueven por la pantalla.
- Agenda de direcciones.
- Sistema de menús para MsDos.

Ejemplos de C++ (6):

- Clase "pantalla de texto", en un solo fuente.

- Clase "pantalla de texto", en dos fuentes: cabecera.
- Clase "pantalla de texto", en dos fuentes: detalle.
- Clase "pantalla de texto", en dos fuentes: ejemplo que la usa.

- Clase "pantalla de texto mejorada", cabecera.
- Clase "pantalla de texto mejorada", detalle.
- Clase "pantalla de texto mejorada", ejemplo que la usa.

- Clase "ventana de texto", cabecera.
- Clase "ventana de texto", desarrollo.
- Clase "ventana de texto", ejemplo que la usa.

- Clase "pantalla de texto mejorada" para Symantec C++.

- Entrada y salida estándar en C++.

N.

CURSO DE C. INDICE DE TABLAS.

Las tablas incluidas en este curso son:

1.- Códigos de formato (tema 4).
2.- Códigos de control (tema 4.3).
3.- Modos de apertura de un fichero (tema 11.3).

N.

CURSO DE C. INDICE.

En este apartado se van a resumir alfabéticamente los temas tratados, así como **las palabras** clave o funciones de biblioteca que se han visto, y **el tema** en el que han aparecido.

Los nombres que empiecen en mayúsculas indicarán que se trata de un tema genérico, mientras que las palabras reservadas o las funciones comenzarán con minúsculas:

- A -
Adivinar un número, fuentes
Agenda de direcciones, fuentes
Arboles binarios, tema 10.4
Arrays en C, tema 2
ASCII, otros temas
Asignación de valores a variables, tema 2
Asignaciones abreviadas, tema 5
Asignaciones múltiples, tema 5

- B -
BGI, ampliaciones.
bloques, tema 1
Booleanos en C, tema 2
break, tema 6 y tema 7.5d
Bucles, tema 7

- C -
Cadenas de texto, tema 2 y tema 12
Campos de bits, tema 2.7b
case, tema 6
char, tema 2
Códigos de control, tema 4
Colas, tema 10.4
Comentarios, tema 1
Comillas dobles o simples, tema 2
Comparaciones, tema 5
Comparar dos ficheros, fuentes
Concatenar cadenas, tema 5 y tema 12
Condiciones, tema 6
const, tema 8
Constantes, tema 8
continue, tema 7.5c
Conversión de tipos, tema 4

- D -
Declaración de variables, tema 2
Decremento, tema 5
default, tema 6
Definición de tipos, tema 8
Desplazamiento, ampliaciones.
Directorios desde Turbo C, ampliaciones.
do, tema 7
Dos desde Turbo C, ampliaciones.

- E -
else, tema 6
Entrada/Salida básica, tema 4
enum, tema 8

- F -
fclose, tema 11
feof, tema 11
fgetc, tema 11
fgets, tema 11
Ficheros, tema 11
float, tema 2
fopen, tema 11
for, tema 7
Formato (printf y scanf), tema 4
fprintf, tema 11
fputc, tema 11
fputs, tema 11
fread, tema 11
free, tema 10
fscanf, tema 11
fseek, tema 11
ftell, tema 11
Funciones, tema 3 y tema 9
fwrite, tema 11

- G -
Generalidades, tema 1
getchar, tema 4 y tema 6
gets, tema 4
goto, tema 7.5e
Gráficos desde Turbo C, ampliaciones.
Gráficos accediendo a memoria, ampliaciones.

- H -
.h (ficheros de cabecera), tema 3

- I -
Identificadores, tema 2
if, tema 6
Imágenes que se mueven, fuentes
Impresora desde C, ampliaciones.
include, tema 1 y tema 3
Incremento, tema 5
int, tema 2
Interrupciones del Dos desde Turbo C, ampliaciones.

- L -
Línea de comandos, ampliaciones
Listas dinámicas simples, tema 10.3
Listas dobles, tema 10.4
long, tema 2

- M -
main(), tema 1 y tema 3
malloc, tema 10
Matrices, tema 2
MAXINT, tema 2
MAXLONG, tema 2
MK_FP, ampliaciones.
Modelos de memoria, ampliaciones.

- N -
NULL, tema 10

- O -
Operaciones matemáticas, tema 5

Operaciones lógicas, tema 5
Operaciones entre bits, tema 5
Operador condicional, tema 6

- P -
Parámetros de una función, tema 9
Parámetros de un programa, ampliaciones
Pilas, tema 10.4
printf, tema 1 y tema 4
Punteros, tema 9 y tema 10
putchar, tema 4
puts, tema 4

- R -
Recursividad, tema 9
Referencia (paso de parámetros), tema 9
Reservar memoria, tema 10
return, tema 9

- S -
scanf, tema 2 y tema 4
Secuencias de escape, tema 4
Segmentos, ampliaciones.
Sentencias compuestas, tema 6
short, tema 2
signed, tema 2
Sintaxis del C, tema 1
sizeof, tema 10
stdio.h, tema 1
strcat, tema 12
strcmp, tema 12
strcpy, tema 12
string.h, tema 12
strncpy, tema 12
struct, tema 2
strupr, tema 12
switch, tema 6

- T -
Tipos (definición), tema 8
Typecast, tema 4
typedef, tema 8

- U -
union, tema 2.7b
unsigned, tema 2

- V -
Valor (paso de parámetros), tema 9
Variables, tema 2
Variables locales y globales, tema 9
Variables dinámicas y estáticas, tema 10
Vectores, tema 2
void, tema 3
volatile, tema 8

- W -
while, tema 7

- Otros -
#include, tema 1 y tema 3
#define, tema 8

/* y */ (comentarios), tema 1
{ y (bloques, sentencias compuestas), tema 1 y tema 6
() (funciones), tema 1
; (fin de orden), tema 1
[] (arrays: matrices o vectores), tema 2
& (dirección de memoria), tema 2
" y "" (comillas), tema 2
<> y "" (include), tema 3
\ (secuencias de escape), tema 4
+ - / * % (operadores), tema 5
++ -- (incremento y decremento), tema 5
+= -= *= /= %=, tema 5
== != < > <= >= (comparaciones), tema 5
&& || ! (y,o,no), tema 5
& ! ^ << >> (operadores de bits), tema 5
? : (operador condicional), tema 6
* (puntero) y & (dirección), tema 9 y tema 10

N.

CURSO DE C. PASCAL -> C.

Este apartado es sólo una referencia rápida para quien haya programado en Pascal y ahora esté empezando con el C. No pretende cubrir todos los casos posibles, sino simplemente aportar una serie de ejemplos con las situaciones más habituales (creo), a modo de recordatorio:

Declaraciones de variables.
Escritura en pantalla.
Lectura de datos.
Comprobación de condiciones.
Estructuras de control.
Procedimientos y funciones.
Punteros.

Para cada una de estas categorías, iré comparando cómo se haría en Pascal y cómo en C.

Declaraciones de variables.

PASCAL	C
var	
a: byte;	char a;
b: integer;	int b;
c: longint;	long c;
d: real;	float d;
e: boolean;	(Se debe simular con char o int)
f: string[80];	char f[81];
g: string;	char g[256];
h: array[1..20,1..10] of integer;	int f[20][10];
i: record	struct {
nombre: string[20];	char nombre[20];
edad: byte;	char edad;
end;	i;

Escritura en pantalla.

```
writeln('Hola');                    printf("Hola\n");
write(a);                           printf("%d",a);
writeln('Nombre: ', f);             printf("Nombre: %s\n", f);
write('Elemento (5,6):', h[5,6]);   printf("Elemento (5,6): %d", h[4][5]);
```

Lectura de datos.

```
readln(f);                          gets(f);
read(b);                            scanf("%d", &b);
```

Comprobación de condiciones.

```
if a>2 then write('Sí');            if (a>2)   printf("Sí");
if not ((a<b) and (b<c)) then [...] if ! ((a<b) && (b<c))   [...]
if (a<>0) or (b=2) then [...]       if a || (b==2)   [...]

case a of                           switch (a) {
  1: write('Uno');                    case 1: printf("Uno"); break;
  2, 3: write('Dos o tres');        case 2:
  4: begin                            case 3: printf("Dos o tres"); break;
```

```
        writeln('Cuatro');                    case 4: printf("Cuatro");
        solucion := 8;                                solucion = 8;
        end;                                           break;
  else                                        default:
       write('No es 1, 2, 3 ni 4');                  printf("No es 1, 2, 3 no 4");
  end;
```

Estructuras de control.

```
  for a := 1 to 5 do write(a);          for (a=1; a<=5; a++)   printf("%d",a);

  while a<5 do                           while (a<5)
    begin                                {
    a := a+1;                             a++;
    write(a)                              printf("%d",a);
    end;

  repeat                                do {
    a := a+1;                             a++;
    write(a)                              printf("%d",a);
  until a>=5;                             while (a<5);
```

Procedimientos y funciones.

```
  procedure hola;                          void hola();
  function fatorial(x:real):real;       float factorial(float x);
  procedure modifca(var b:integer);     void modifica(int* b);
```

Punteros.

```
  var p: ^integer;                       int *p;
  p^ := 2;                                *p = 2;
```

Hay que recordar que en C las cadenas de texto son punteros, y en Pascal no, luego en C no se pueden concatenar cadenas haciendo a = b + c; ni darles valores con a = "Hola";

Por otra parte, los arrays en C se pueden recorrer como array o como puntero: s[5] o *(p+5).

N.

CURSO DE C. AMPLIACIONES SOBRE LOS TEMAS BASICOS.

(--- Nota: estos temas sólo están disponibles en la versión ampliada del curso ---).

Pantalla en modo texto en Turbo C
Pantalla en modo texto en Symantec C++
Pantalla en modo texto en Linux

Pantalla en modo gráfico en Turbo C
Pantalla en modo gráfico accediendo a la memoria
Funciones del DOS desde Turbo C
Acceso a la impresora desde C
Control del ratón en MsDos
Programas a partir de varios fuentes
Modelos de memoria
Incluir ensamblador en C
Leer la línea de comandos

Resumen rápido C -> Pascal

CURSO DE C. PANTALLA DE TEXTO EN TURBO C.

(--- Nota: este tema sólo está disponible en la versión ampliada del curso ---).

Ver cómo manejar la pantalla de texto en SC++
Volver al índice de ampliaciones sobre los temas básicos
Volver al menú principal

CURSO DE C. PANTALLA DE TEXTO EN SYMANTEC C++.

(--- Nota: este tema sólo está disponible en la versión ampliada del curso ---).

Ver cómo manejar la pantalla de texto en Linux
Volver al índice de ampliaciones sobre los temas básicos
Volver al menú principal

CURSO DE C. PANTALLA DE TEXTO EN GCC/LINUX.

(--- Nota: este tema sólo está disponible en la versión ampliada del curso ---).

Ver cómo manejar la pantalla gráfica en TC
Volver al índice de ampliaciones sobre los temas básicos
Volver al menú principal

CURSO DE C. PANTALLA GRAFICA EN TURBO C.

(--- Nota: este tema sólo está disponible en la versión ampliada del curso ---).

Ver otro ejemplo de programa que use órdenes gráficas
Pasar a las funciones del DOS con Turbo C
Volver al índice de ampliaciones sobre los temas básicos
Volver al menú principal

CURSO DE C. PANTALLA GRAFICA ACCEDIENDO A MEMORIA.

(--- Nota: este tema sólo está disponible en la versión ampliada del curso ---).

Pasar a las funciones del DOS con Turbo C
Volver al índice de ampliaciones sobre los temas básicos
Volver al menú principal

Gráficos en Turbo C (Continuación) -

(--- Nota: este tema sólo está disponible en la versión ampliada del curso ---).

Ver un índice de funciones gráficas de Turbo C
Pasar a las funciones del DOS con Turbo C
Volver al índice de ampliaciones sobre los temas básicos
Volver al menú principal

CURSO DE C. FUNCIONES GRAFICAS EN TURBO C.

(--- Nota: este tema sólo está disponible en la versión ampliada del curso ---).

Pasar a las funciones del DOS con Turbo C
Volver al índice de ampliaciones sobre los temas básicos
Volver al menú principal

CURSO DE C. FUNCIONES DEL DOS DESDE TURBO C.

(--- Nota: este tema sólo está disponible en la versión ampliada del curso ---).

Pasar al manejo de la impresora desde C
Volver al índice de ampliaciones sobre los temas básicos
Volver al menú principal

CURSO DE C. MANEJO DE LA IMPRESORA.

(--- Nota: este tema sólo está disponible en la versión ampliada del curso ---).

Ver cómo manejar el ratón en MsDos
Volver al índice de ampliaciones sobre los temas básicos
Volver al menú principal

CURSO DE C. RATON EN MSDOS.

(--- Nota: este tema sólo está disponible en la versión ampliada del curso ---).

Ver cómo crear programas a partir de varios fuentes
Volver al índice de ampliaciones sobre los temas básicos
Volver al menú principal

CURSO DE C. PROGRAMAS A PARTIR DE VARIOS FUENTES.

(--- Nota: este tema sólo está disponible en la versión ampliada del curso ---).

Ver qué son los modelos de memoria
Volver al índice de ampliaciones sobre los temas básicos
Volver al menú principal

CURSO DE C. MODELOS DE MEMORIA.

(--- Nota: este tema sólo está disponible en la versión ampliada del curso ---).

Ver cómo incluir lenguaje ensamblador
Volver al índice de ampliaciones sobre los temas básicos
Volver al menú principal

CURSO DE C. ENSAMBLADOR DESDE C.

(--- Nota: este tema sólo está disponible en la versión ampliada del curso ---).

Ver cómo leer la línea de comandos
Volver al índice de ampliaciones sobre los temas básicos
Volver al menú principal

CURSO DE C. LINEA DE COMANDOS.

(--- Nota: este tema sólo está disponible en la versión ampliada del curso ---).

Ver el resumen rápido C -> Pascal
Volver al índice de ampliaciones sobre los temas básicos
Volver al menú principal

Contactar con el autor...

Para cualquier tipo de duda, comentario o sugerencia, puedes localizarme en cualquiera de estas dos direcciones: si tienes módem

Nacho Cabanes
e-mail en Internet: ncabanes@arrakis.es

Puedes consultar mi página Web en cualquiera de las direcciones:

http://www.arrakis.es/~ncabanes
http://members.es.tripod.de/ncabanes

y si no tienes módem, mi apartado de correos es el siguiente:

Nacho Cabanes
Apdo. Correos 5234
03080 Alicante (España)

(Si es sólo una consulta, incluye sobre y sellos para la respuesta, por favor).

CURSO DE C. CODIGO ASCII.

(--- Nota: este tema sólo está disponible en la versión ampliada del curso ---).

CURSO DE C. FUENTES DE EJEMPLO.

(--- Nota: este tema sólo está disponible en la versión ampliada del curso ---).

En este apartado intentaré poner programas algo más completos que los fuentes de ejemplo que se han ido viendo en los temas anteriores. Serán "programitas" capaces de funcionar por sí mismos, y que hagan algo que tenga una cierta utilidad.

Por ahora, puede acceder a los siguientes:

- Adivinar un número.
- Mostrar un fichero de texto.
- Comparar dos ficheros.
- Figuras que se mueven por la pantalla.
- Agenda de direcciones.
- Sistema de menús para MsDos.

Volver al menú principal

Fuentes: Adivinar un número.

(--- Nota: este tema sólo está disponible en la versión ampliada del curso ---).

Volver al menú de fuentes
Volver al menú principal

Fuentes: Adivinar un número (2).

(--- Nota: este tema sólo está disponible en la versión ampliada del curso ---).

Volver al menú de fuentes
Volver al menú principal

Fuentes: Mostrar un fichero de texto.

(--- Nota: este tema sólo está disponible en la versión ampliada del curso ---).

Volver al menú de fuentes
Volver al menú principal

Fuentes: Comparar dos ficheros.

(--- Nota: este tema sólo está disponible en la versión ampliada del curso ---).

Volver al menú de fuentes
Volver al menú principal

Fuentes: Sprites (imágenes que se mueven).

(--- Nota: este tema sólo está disponible en la versión ampliada del curso ---).

Volver al menú de fuentes
Volver al menú principal

Fuentes: Agenda de direcciones.

(--- Nota: este tema sólo está disponible en la versión ampliada del curso ---).

Volver al menú de fuentes
Volver al menú principal

Fuentes: Sistema de menús para MsDos.

(--- Nota: este tema sólo está disponible en la versión ampliada del curso ---).

Volver al menú de fuentes
Volver al menú principal

CURSO DE C. NOTAS FINALES.

Sólo algunos comentarios para quien haya conseguido llegar hasta aquí:

Este curso tiene más de 80 fuentes de ejemplo, y el texto del curso ocupa más de 12.000 líneas y más de 400K. Pero...

--=== Esto no es "el C" ===--

Me refiero a que es no es "todo el C", sino sólo una ínfima parte: hay más ordenes de entrada y salida, más para manejo de ficheros, más posibilidades de punteros, etc. Por ejemplo, si alguien busca la ayuda sobre "malloc" en Turbo C++ 1.01, le aparecerán como temas relacionados, entre otros:

allocmem calloc farcalloc farmalloc realloc

Igualmente, al mirar la ayuda sobre "printf" aparecen referencias a

cprintf fprintf sprintf vprintf vsprintf

Es decir, es muy frecuente que haya muchas "variantes" de una orden dada. En el caso de printf puede ser para escribir en color, o en un fichero, o mandar la salida a una cadena de texto...

Lo que pretendo no es describir todas y cada una de las órdenes que uno se pueda encontrar delante de un compilador de C (muchas de las cuales ni siquiera existirán en otros compiladores), sino quitar un poco el miedo, dar una base, y abrir una puerta para quien quiera investigar más.

Nada más. Espero que este curso le haya resultado útil, al menos para coger una base y perder un poco el miedo.

N.

Volver al menú principal

Compiladores.

Los ejemplos de este curso están comprobados con al menos un compilador de lenguaje C. En este apartado voy a comentar brevemente dónde se pueden conseguir esos compiladores gratis o a bajo precio (sin ningún tipo de piratería, por supuesto), así como algunas notas básicas sobre su instalación y uso.

Turbo C++ 2nd Edition
Symantec C++ 6.0
PCC 2.1c
DJGPP 2.01

Turbo C++ 2nd Edition.

Este es un compilador de C y C++ para MsDos (en modo real), creado por la casa Borland. Incluye el compilador "de línea de comandos" (TCC) y un entorno integrado de desarrollo (IDE), TC.EXE.

Ocupa dos diskettes de 1.44 Mb, y fue **regalado** con los números 3 (Enero de 1995) y 4 (Febrero de 1995) de la revista **RPP** ("Revista profesional para programadores"), editada por Anaya Multimedia.

El precio de portada de la revista era de 795 pesetas, luego quien quiera pedir estos dos ejemplares para conseguir el compilador, deberá preparar unas 1.500 pesetas (más gastos de envío, supongo).

Para instalarlo, hay que introducir el primer diskette, teclear A:INSTALL y seguir las instrucciones en pantalla.

Para conseguir versiones **más modernas** (eso sí, ya no serán gratis), póngase en contacto con cualquier distribuidor oficial de Borland, o mire en Internet la página **http://www.borland.com**

Symantec C++ 6.0.

Este es un compilador de C y C++ para MsDos (en modo real y en modo protegido) y para Windows, creado por la casa Symantec (anteriormente era conocido como Zortech C++). Tiene algún fallo, pero aun así puede ser de mucha utilidad. Incluye el compilador "de línea de comandos" (SC, para MsDos) y un entorno integrado de desarrollo y depuración (IDDE), para Windows, con realce de la sintaxis en colores.

Los ficheros de instalación ocupa cerca de 80 Mb, y fue **regalado** en CdRom con el número 4 (Diciembre de 1994) de la revista **Solo Programadores** editada por Tower Communications. El precio de portada de la revista era de 1250 pesetas.

Para instalarlo, hay que introducir el CdRom y ejecutar desde Windows (con el Administrador de Archivos de Windows 3.1x o con el Explorador de Windows 95, por ejemplo) el fichero SETUP.EXE.

Hay al menos dos versiones más modernas: la 6.1, que básicamente corrige los fallos descubiertos, y la 7.0, pero no sé si se sigue distribuyendo esta versión o alguna más reciente. Puede consultar en Internet la página **http://www.symantec.com**

PCC 2.1c

Este es un compilador de C (no de C++) para MsDos , muy elemental (limitado a 64K de código y 64K de datos, sin entorno de desarrollo), distribuido como shareware, creado en junio de 1989 por:

C Ware Corporation
P.O. Box 428
Paso Robles, CA 93446

No sé si existen versiones más recientes, ni siquiera si sigue existiendo la empresa que lo crea. Esta versión se puede conseguir en mi página Web **http://www.arrakis.es/~ncabanes**

Para instalarlo no hay más que descomprimir el fichero PCC21C.ZIP. tecleando

```
PKUNZIP PCC21C
```

o bien

```
UNZIP PCC21C
```

según el descompresor que poseamos.

Para crear los fuentes necesitamos un editor de textos, ya que no incluye entorno de desarrollo. Para compilar un programa, hay que teclear dos líneas: una para crear el código objeto y otra para enlazar y crear el ejecutable:

```
PCC NombreDeFuente
PCCL NombreDeFuente
```

donde NombreDeFuente es el nombre del fichero en lenguaje C, sin extensión. Por ejemplo, si tenemos un fichero llamado HOLA.C, teclearíamos

```
PCC HOLA
PCCL HOLA
```

El primer paso crea HOLA.O a partir de HOLA.C, y el segundo crea HOLA.EXE a partir de HOLA.O.

Podemos automatizar estos dos pasos creando un fichero BAT. En mi caso, uso uno llamado NCC.BAT que contiene

```
PCC %1
PCCL %1
```

Este fichero se puede crear con cualquier editor de textos, como el propio EDIT del DOS. Entonces, para dar los dos pasos de compilación bastará hacer NCC HOLA.

DJGPP

DJGPP es la versión para MsDos del compilador de C y C++ desarrollado por GNU. Esta versión ha sido desarrollada por DJ Delorie.

Se trata de un compilador gratis, que crea ejecutables para MsDos en modo protegido. Voy a tratar de describir los ficheros necesarios para instalar la versión 2.01, así como los pasos que hay que dar para completar dicha instalación.

Esta versión se regalaba en CdRom con el número 63 de la revista PcManía (Diciembre 97), que tenía un precio de portada de 1.200 pts. La versión más reciente se podrá conseguir en Internet, en la página Web de su creador, DJ Delorie, **http://www.delorie.com**

Vamos a hacer una "instalación típica". Para ello necesitaremos los siguientes ficheros:

Para compilar fuentes en C:
- djdev201.zip (del directorio DJGPP\V2)
- gcc2721b.zip (del directorio DJGPP\V2GNU)
- bnu27b.zip (del directorio DJGPP\V2GNU)

Para C++, además
- gpp2721b.zip (del directorio DJGPP\V2GNU)
- lgp271b.zip (del directorio DJGPP\V2GNU)

Para leer los manuales en línea
- txi390b.zip (del directorio DJGPP\V2GNU)

Si no tenemos servidor DPMI (para acceder al modo protegido del DOS)
- csdpmi3b.zip (del directorio DJGPP\V2MISC)
  (No es necesario si trabajamos de Windows, OS/2, Linux DOSEmu u OpenDOS, entre otros, pero tampoco está de más por si acaso).

Para las preguntas más frecuentes y sus respuestas
- faq210b.zip (del directorio DJGPP\V2)

Si queremos tener un entorno de desarrollo con realce de sintaxis en colores y otras ventajas
- rhide14b.zip (del directorio DJGPP\V2APPS)

En total, estos ficheros ZIP ocupan unos 7.900 Kb (como aviso para aquellos que tengan el valor de traerlos desde Internet).

Los pasos a seguir son los siguientes:

1) Crear el directorio en el que queremos instalar DJGPP, por ejemplo
   C:\DGPPP

   C:\> MD DJGPP

2) Copiar en ese directorio los ficheros ZIP que nos interese y descomprimirlos con su estructura de subdirectorios, por ejemplo así:

   C:\DJGPP> PKUNZIP -D DJDEV201.ZIP
   ...

3) Borrar estos ficheros ZIP que ya hemos descomprimido.

   C:\DJGPP> DEL DJDEV201.ZIP
   ...

4) Preparar las variables de entorno que le dicen al sistema dónde encontrar DJGPP y sus ficheros auxiliares:

SET DJGPP=C:\DJGPP\DJGPP.ENV
SET PATH=C:\DJGPP\BIN;%PATH%

Estas dos líneas se pueden teclear antes de empezar a usar DJGPP (cada día que lo usemos), o mejor incluirlas al final del fichero AUTOEXEC.BAT.

En mi caso, yo prefiero no cargar mucho el AUTOEXEC.BAT, y me creo un fichero BAT por cada compilador, por ejemplo DJGPPINI.BAT. Cuando voy a usar DJGPP accedo primero a ese fichero BAT, y así no desperdicio memoria (espacio de entorno, para ser más exactos) cuando uso mi ordenador pero no voy a acceder a DJGPP.

5) Si estas dos variables las hemos incluido en AUTOEXEC.BAT, deberemos reiniciar el equipo para que los cambios tengan efecto. Si hemos creado otro fichero BAT, bastará con ejecutarlo.

Después de hacer una de ambas cosas, tecleamos GO32-V2 para saber cuanta memoria tenemos disponible para el servidor. Nos dirá dos cantidades: la memoria disponible y el espacio de intercambio, algo parecido a esto:

DPMI memory available: 4339 Kb
DPMI swap space available: 3469 Kb

Si la memoria es inferior a 4 Mb, deberemos acudir a las FAQs a buscar alguna solución alternativa. Si nos indica más de esta cantidad, estamos listos para compilar un programa de prueba.

Creamos el clásico programa HOLA.C o similar

```
#include <stdio.h>

main()
{
   printf("Hola\n");
}
```

y lo compilamos tecleando

gcc hola.c -o hola.exe

(La opción -o sirve para indicar el nombre del fichero ejecutable; si no incluimos esa opción, el fichero se llamará A.OUT y/o A.EXE).

En mi caso, este programa tan sencillo se convierte en un fichero EXE de 80K de tamaño.

6) Si queremos trabajar con el entorno de desarrollo RHIDE, basta con teclear RHIDE. En el menú File tenemos las opciones de Abrir ficheros ya existentes (Open) o crear un fichero nuevo (New).

Un comentario: si creamos un fichero con la opción New y empezamos a teclear, veremos que no realza la sintaxis en colores. Para que

lo haga, necesita saber que se trata de un fuente en C, y lo sabrá en cuanto grabemos el fichero. Aunque no esté terminado, pulsamos F2 (o elegimos la opción Save del menú File), le damos un nombre como HOLA.C y a partir de entonces ya tendremos sintaxis en colores, que hace más legible el fuente y más fácil descubrir errores.

Como comentario final, esta versión de DJGPP (2.01), con esta instalación típica, supone instalar en nuestro disco duro ficheros hasta un total de más de 17.000 Kb. En sí mismo no es un tamaño exagerado, pero hay muchos ficheros pequeños, lo que hace que en un disco duro de 1.6 Gb (con clusters de 32Kb) realmente nos estará ocupando más de 47.000 Kb !!!!

En un disco duro más pequeño, o si se emplea un sistema de ficheros que no haga clusters tan grandes, ocupará un valor intermedio entre esos 17.000 y 47.000.

(**Nota:** la mayor parte de esta información se puede encontrar en un fichero llamado README.1ST, en el directorio DJGPP\V2; para versiones posteriores a la 2.01, es casi seguro que habrán cambiado los nombres de los ficheros, así que convendrá mirar README.1ST para ver las diferencias).

Próximas mejoras:

Añadido un índice de contenidos, similar al índice detallado de los temas básicos, pero aplicado a todo el curso. (*)
Añadidos mayor número de enlaces, para poder consultar ciertas tablas (códigos de formato, p.ej.) y el formato de ls órdenes más habituales desde los sitios en que se mencionen. (*)
Añadido un ejemplo de cómo hacer un menú para MsDos
Ejemplos de cómo mostrar PCX, DBF y similares.

La lista de cambios (ésta) permite saltar desde aquí a los temas que se mencionan, para que sea más fácil revisar los cambios a aquellos usuarios que ya tengan una versión anterior del curso.

CURSO DE C. CAMBIOS.

Versión	Fecha	Cambios
1.54w	21-03-99	Creada una versión del curso con lector para Windows (en formato Ayuda de Windows). La lista de ejemplos permite saltar a la lección en la que se encuentra. Corregida alguna pequeña errata de poca importancia.
1.53	12-03-98	Corregido un error del lector que hacía que los ejemplos se exportasen con un símbolo raro al comienzo de cada línea.
1.52	19-01-98	Añadido un tema sobre los compiladores que he empleado para probar los ejemplos: dónde conseguirlos y como instalarlos. Corregido un error en el ejemplo 17 y alguna errata más. Otras ligeras mejoras internas al lector.
1.51	27-10-97	Versión "de mantenimiento": corregidas erratas (p.ej.: modidicador->modificador, concer-> conocer, etc). Añadido un apartado sobre los modelos de memoria en el PC bajo MsDos al final del tema 12.
1.50	7-04-97	Primera versión hipertexto. Texto casi igual que la 1.00. Se puede manejar con ratón. Se pueden exportar los ejemplos y las tablas.
1.00	20-12-96	Versión inicial.

CURSO DE C. OOP y C++.

Indice de temas.

Introducción a la Programación Orientada a Objetos

1. Primera toma de contacto con C++
2. Constructores y destructores; fuente en dos bloques
3. Primer ejemplo de herencia
4. Dos objetos que interaccionan
5. Modificaciones para otro compilador

6. Otros cambios en C++ frente a C

CURSO DE C. OOP y C++.

Indice alfabético.

==========

--- Este apartado aún está sin realizar. Mil disculpas. --- ##

CURSO DE C. Introducción a OOP y C++.

(--- Nota: este tema sólo está disponible en la versión ampliada del curso ---).

Volver al índice global

Curso de C - Versión Ampliada.

Lo siento, has escogido un tema que sólo está disponible en la versión ampliada. Espero que aun así esta versión te resulte útil.

Las **ventajas** que incluye la versión ampliada son:

- Más ampliaciones y ejemplos.
- Lecciones revisadas con más frecuencia.
- Mayor cantidad de enlaces, para aclarar más términos desde casi cualquier parte del curso.
- Las 1.000 pesetas que cuesta incluyen gastos de envío (por correo certificado).
- Tener la conciencia tranquila.
- Yo veo que esto sirve para algo, y me anima a preparar otros cursos y ampliar éste.

Si quieres **conseguir** la versión ampliada, imprime el formulario de solicitud y envíamelo, tras rellenarlo (con letras mayúsculas o letra muy legible)

Recuerda que la **forma** más barata, pero a la vez menos fiable (aunque hasta ahora parece que nadie ha tenido problemas) es incluir el billete en el sobre. Un poco más caro, pero totalmente seguro, es el giro postal. Finalmente, las opciones más caras, pero igualmente seguras son el talón nominativo y el envío contra reembolso.

CURSO DE C, VERSION 1.54w - Impreso de solicitud

Fecha: ___ /___ /____

Sí, quiero colaborar con

( ) 1.000 pesetas o 6 euros (billete, que adjunto -menos fiable-)

( ) 1.000 pesetas o 6 euros (giro postal al apartado, previo a la carta)

( ) 1.500 pesetas o 9 euros (talón nominativo, que adjunto)

( ) 1.000 pesetas (o 6 euros) + gastos de envío contra reembolso
(aprox. 500 ptas más -3 euros más-)

( ) 10 US$ (10 Dólares USA, billete, que adjunto, para usuarios
de fuera de España).

Y con ello recibir la última **versión ampliada** del curso. Mis datos son:

Nombre: _____

Dirección: _____

Ciudad y código postal: _____

_____

País (si es de fuera de España): _____

Conseguí el curso a través de:

[ ] BBS (_____) [ ] CD-Rom (_____)
[ ] Amigos [ ] Internet (_____)
[ ] Otros (_____)

Datos estadísticos (no es obligatorio contestar):

Dirección e-mail : _____

Edad: _____ Teléfono: _____

Estudios o Trabajo : _____

Ordenador : _____

Enviar a:

José Ignacio Cabanes Andreu
Apdo. Correos 5234
03080 Alicante
(España)

Conseguir la última versión.

Conseguir la última versión de este curso es fácil. Intentaré mantenerla siempre en mi página Web. Ahora mismo, mi página Web está en las siguientes direcciones:

http://www.arrakis.es/~ncabanes
http://members.es.tripod.de/ncabanes

Es fácil que la segunda dure más que la primera, porque no depende de un servidor español concreto, sino que es una dirección cedida por Lycos.

En cualquier caso, si no apareciera ninguna de éstas, haz una búsqueda en Olé, Ozú o algún otro de los buscadores de habla hispana más conocidos y por ahí apareceré yo...