Bajado de Internet, Cátedra de Computación

- CURSO DE C -

Por Nacho Cabanes

Esto va mejorando...

Sí, aunque no todo lo que debería. El texto del curso ya está aquí, pero an está sin formatear: poner negritas y cursivas para que resulte más legible, hacer un índice para saltar directamente a cada tema, etc.

Esto está basado en la versión 0.94 de mi curso de C, es decir, ni siquiera es la 1.0... me explico: no está todo lo revisado que debería, así que puede haber algún error. Si lo encuentra, le ruego que me avise escribiéndome a...

ncabanes@arrakis.es

He conservado todo junto en un único fichero para que sea más sencillo de coger, de imprimir o incluso de grabar a otras personas.

Tema 1. Generalidades.

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, 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    */
/*                         */
/*  Programa elemental     */
/*                         */
/*  Comprobado con:        */
/*    - Turbo C++ 1.01     */
/*    - Symantec C++ 6.0   */
/*    - GCC 2.6.3          */
/*-------------------------*/

#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 a la hora de crear nuestro programa ejecutable.

- 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 función 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. Es 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”? Pues bien, En Pascal, el cuerpo del programa se delimitaba entre “begin” y “end”. En C no basta con indicarlo entre { y }, sino que además dicho cuerpo del programa ya es en sí una función, que necesariamente debe llamarse “main” (en minúsculas, tal cual). 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. 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 una par de cosas más antes de seguir adelante: La gran mayoría de las palabras clave de C, al igual que ocurre en Pascal, son palabras en inglés o abreviaturas de éstas. En cambio, en C sí existe DISTINCION entre mayúsculas y minúsculas, por lo que “for” es una palabra reconocida, pero “For”, “FOR” o “fOr” no lo son. También al igual que en Pascal, cada sentencia de C debe terminar con un PUNTO Y COMA. Aun así, hay algunas pequeñas diferencias que ya iremos viendo. En Pascal distinguíamos 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” (que ya veremos), con lo cual equivalen a los procedimientos de Pascal. Ya sabemos escribir, pero eso nos soluciona pocas cosas, así que sigamos viendo cosas nuevas...

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 leido, 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      */
/*                         */
/*  Introd. a variables    */
/*                         */
/*  Comprobado con:        */
/*    - Turbo C++ 1.01     */
/*    - Symantec C++ 6.0   */
/*    - GCC 2.6.3          */
/*-------------------------*/

#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);
}

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

Aquí seguimos encontrando cosas nuevas: las variables se declaran 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;

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. Eso sí, 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.

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      */
/*                         */
/*  Inicialización de      */
/*  variables              */
/*                         */
/*  Comprobado con:        */
```

```
/*    - Turbo C++ 1.01     */
/*    - Symantec C++ 6.0   */
/*    - GCC 2.6.3          */
/*-------------------------*/

#include stdio.h
int primerNumero = 234;    /* Nuestras variables */
int segundoNumero = 567;
int suma;

main()
{
suma = primerNumero + segundoNumero;
printf(“Su suma es %d”, 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: Lo que le indicamos entre comillas es realmente un codigo de FORMATO. Dentro de él 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 ese %d se sustituirá por un número entero. ¿Qué número? El que le indicamos a continuación, separado por una coma (en este caso, “suma”). ¿Y cómo indicamos más de un valor? Pues vamos a verlo con otro ejemplo, también ampliando el anterior:

```
/*-------------------------*/
/*  Ejemplo en C nº 4      */
/*                         */
/*  Escribir más de una    */
/*  variable               */
/*                         */
/*  Comprobado con:        */
/*    - Turbo C++ 1.01     */
/*    - Symantec C++ 6.0   */
/*    - GCC 2.6.3          */
/*-------------------------*/

#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);
}

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

Bueno, pero hasta ahora hemos estado viendo cómo escribir variables, pero si queremos que realmente sean variables, 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:

```
/*-------------------------*/
/*  Ejemplo en C nº 5      */
/*                         */
/*  Leer valores para      */
```

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

#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 ú 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). Estos son los tipos más habituales. 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     */
/*                        */
/*  Primer ejemplo de     */
/*  arrays                */
/*                        */
/*  Comprobado con:       */
/*    - Turbo C++ 1.01    */
/*    - Symantec C++ 6.0  */
/*    - GCC 2.6.3         */
```

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

#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. En este caso, tendríamos:

```
/*-------------------------*/
/*  Ejemplo en C nº 7      */
/*                         */
/*  Inicialización de      */
/*  arrays                 */
/*                         */
/*  Comprobado con:        */
/*    - Turbo C++ 1.01     */
/*    - Symantec C++ 6.0   */
/*    - GCC 2.6.3          */
/*-------------------------*/

#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).

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      */
/*                         */
/*  Registros (struct)     */
/*                         */
/*  Comprobado con:        */
/*    - Turbo C++ 1.01     */
/*    - Symantec C++ 6.0   */
/*    - GCC 2.6.3          */
/*-------------------------*/

#include stdio.h
struct {
int valor;
float coste;
char ref;
} ejemplo;

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

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

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

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 -22147483648 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". Vamos a recopilar todo esto en una tabla. Los paréntesis indican que una cierta parte del nombre es el valor por defecto, y que no hace falta indicarlo:

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	4	3.4E-38	3.4E+38
double	8	1.7E-308	1.7E+308
long double	10	3.4E-4932	1.1E+4932

(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

Nota: estos tamaños y rangos de valores son válidos en la mayoría de los compiladores bajo DOS. 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      */
/*                         */
/*  Introducción a las     */
/*  cadenas de texto       */
/*                         */
/*  Comprobado con:        */
/*    - Turbo C++ 1.01     */
/*    - Symantec C++ 6.0   */
/*    - GCC 2.6.3          */
/*-------------------------*/

#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 “ú” 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’).

Tema 3: Pequeña recapitulación.

Este tema vuelve atrás y explica un poco más eso del include, y del .h, y del main... Pero no lo he incluido aquí (en la página Web) porque posiblemente no es imprescindible, pero ocupa sitio..

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 má 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 par&uacutenbsp;metros (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
```

Eso quiere 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 no 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     */
/*                         */
/* “main” como int         */
/*                         */
/*  Comprobado con:        */
/*    - Turbo C++ 1.01     */
/*    - Symantec C++ 6.0   */
/*    - GCC 2.6.3          */
/*-------------------------*/

#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.

La otra forma (la “menos buena”) es

```
/*-------------------------*/
/*  Ejemplo en C nº 11     */
/*                         */
/* “main” como void        */
/*                         */
/*  Comprobado con:        */
/*    - Turbo C++ 1.01     */
/*    - Symantec C++ 6.0   */
/*    - GCC 2.6.3          */
/*-------------------------*/

#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.

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 modidicadores: 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:

```
/*-------------------------*/
/*  Ejemplo en C nº 12     */
/*                         */
/*  Formatos con “printf”  */
/*                         */
/*  Comprobado con:        */
/*    - Turbo C++ 1.01     */
/*    - Symantec C++ 6.0   */
/*    - GCC 2.6.3          */
/*-------------------------*/

#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”, que veremos 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, 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.

Sigamos. Vamos a ver ahora qué era eso de “\n”. Pues se trata simplemente de códigos de

control (llamados “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”), para pasar a la línea siguiente. Veamos en una tablita cuáles podemos usar:

Código	Significado
\n	nueva línea (new linea)
\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 */
```

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     */
/*                         */
/*  “gets” y “puts”        */
/*                         */
/*  Comprobado con:        */
/*    - Turbo C++ 1.01     */
/*    - Symantec C++ 6.0   */
/*    - GCC 2.6.3          */
/*-------------------------*/

#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.

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
*	Multiplication
/	Division

¿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. Si queremos saber el resto de la división, deberemos usar el operador %. el resultado sería 3.333333, un número real. Si queremos la parte entera de la división, deberemos utilizar “div”. Finalmente, “mod” nos indica cual es el resto de la división. El signo - se puede usar también para indicar negación. Allá van unos ejemplillos:

```
/*-------------------------*/
/*  Ejemplo en C nº 14     */
/*                         */
/*  Operaciones con        */
/*  números enteros        */
/*                         */
/*  Comprobado con:        */
/*    - Turbo C++ 1.01     */
/*    - Symantec C++ 6.0   */
/*    - GCC 2.6.3          */
/*-------------------------*/

#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);
}

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

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 estoy 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, primeru 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;
```

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
 ---------  ------------

   úú        Y
   ||        O
   !         No
```

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)
   ú       Producto lógico (and)
   |       Suma lógica (or)
   ^       Suma exclusiva (xor)
   <<      Desplazamiento hacia la izquierda
   >>      Desplazamiento a la derecha
```

Estas las usaremos en casos muy concretos (pocos), así que pasemos ya al tema siguiente...

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     */
/*                         */
/*  “if” elemental         */
/*                         */
/*  Comprobado con:        */
/*    - Turbo C++ 1.01     */
/*    - Symantec C++ 6.0   */
/*    - GCC 2.6.3          */
/*-------------------------*/

#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. 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     */
/*                         */
/*  “if” abreviado         */
/*                         */
/*  Comprobado con:        */
/*    - Turbo C++ 1.01     */
/*    - Symantec C++ 6.0   */
/*    - GCC 2.6.3          */
/*-------------------------*/

#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”);
}

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

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     */
/*                         */
/*  “if” y sentencias      */
/*  compuestas             */
/*                         */
/*  Comprobado con:        */
/*    - Turbo C++ 1.01     */
/*    - Symantec C++ 6.0   */
/*    - GCC 2.6.3          */
/*-------------------------*/

#include stdio.h
int numero;
main()
{
printf(“Escriba un número: “);
scanf(“%d”, numero);
if (numero>0)
   {
printf(“El número es “);
printf(“positivo..\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 quedaria sin ejemplo. 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     */
/*                         */
```

```
/*  “if” y “else”          */
/*                         */
/*  Comprobado con:        */
/*    - Turbo C++ 1.01     */
/*    - Symantec C++ 6.0   */
/*    - GCC 2.6.3          */
/*-------------------------*/

#include stdio.h
int numero;
main()
{
printf(“Escriba un número: “);
scanf(“%d”, numero);
if (numero0)
printf(“El número es negativo.\n”);
else
printf(“El número positivo es 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”. Las sentencias “if (condición) s1; else s2;” se pueden encadenar:

```
/*-------------------------*/
/*  Ejemplo en C nº 19     */
/*                         */
/*  “if..else” encadenados */
/*                         */
/*  Comprobado con:        */
/*    - Turbo C++ 1.01     */
/*    - Symantec C++ 6.0   */
/*    - GCC 2.6.3          */
/*-------------------------*/

#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:

```
/*-------------------------*/
/*  Ejemplo en C nº 20     */
/*                         */
/*  “if” incorrecto        */
/*                         */
/*  Comprobado con:        */
```

```
/*    - Turbo C++ 1.01     */
/*    - Symantec C++ 6.0   */
/*    - GCC 2.6.3          */
/*-------------------------*/

#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” (úú), “o” (||) y “no” (!) para enlazarlas:

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

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     */
/*                         */
/*  Operador condicional ? */
/*                         */
/*  Comprobado con:        */
/*    - Turbo C++ 1.01     */
/*    - Symantec C++ 6.0   */
/*    - GCC 2.6.3          */
/*-------------------------*/

#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, este pulsación se queda en el buffer del teclado, y eso es lo que leería el getchar. Vamos a verlo con un ejemplo:

```
/*-------------------------*/
/*  Ejemplo en C nº 22     */
/*                         */
/*  Problemas de “getchar” */
/*                         */
/*  Comprobado con:        */
/*    - Turbo C++ 1.01     */
/*    - Symantec C++ 6.0   */
/*    - GCC 2.6.3          */
/*-------------------------*/

#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.

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     */
/*                         */
/*  Uso de “switch”        */
/*                         */
/*  Comprobado con:        */
/*    - Turbo C++ 1.01     */
/*    - Symantec C++ 6.0   */
/*    - GCC 2.6.3          */
/*-------------------------*/

#include stdio.h
char tecla;
main()
{
printf(“Pulse una tecla: “);
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”.

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     */
/*                         */
/*  Escribe del 1 al 10,   */
/*  con “for”.             */
/*                         */
/*  Comprobado con:        */
/*    - Turbo C++ 1.01     */
/*    - Symantec C++ 6.0   */
/*    - GCC 2.6.3          */
/*-------------------------*/

#include stdio.h
int contador;
main()
```

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

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

Los bucles “for” se pueden enlazar uno dentro de otro, de modo podríamos escribir las tablas de multiplicar del 1 al 5 con:

```
/*-------------------------*/
/*  Ejemplo en C nº 25     */
/*                         */
/*  Escribe las tablas de  */
/*  multiplicar del 1 al   */
/*  5, con “for”.          */
/*                         */
/*  Comprobado con:        */
/*    - Turbo C++ 1.01     */
/*    - Symantec C++ 6.0   */
/*    - GCC 2.6.3          */
/*-------------------------*/

#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);
}

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

En estos casos después de “for” había un única sentencia. Si queremos que se hagan varias cosas, basta definirlas como un bloque, 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     */
/*                         */
/*  Escribe las tablas de  */
/*  multiplicar del 1 al   */
/*  5, con “for”. Deja     */
/*  espacios intermedios.  */
/*                         */
/*  Comprobado con:        */
/*    - Turbo C++ 1.01     */
/*    - Symantec C++ 6.0   */
/*    - GCC 2.6.3          */
/*-------------------------*/

#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”);
      }
}

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

Al igual que en Pascal, para “contar” no necesariamente hay que usar números:

```
/*-------------------------*/
/*  Ejemplo en C nº 27     */
/*                         */
/*  Escribe las letras de  */
/*  la ‘a’ a la ‘z’, con   */
/*  “for”.                 */
/*                         */
/*  Comprobado con:        */
/*    - Turbo C++ 1.01     */
/*    - Symantec C++ 6.0   */
/*    - GCC 2.6.3          */
/*-------------------------*/

#include stdio.h
char letra;
main()
{
for (letra=’a’; letra<=’z’; letra++)
printf(“%c “, letra);
}

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

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     */
/*                         */
/*  Escribe las letras de  */
/*  la ‘z’ a la ‘a’, (una  */
/*  sí y otra no), con     */
/*  “for”.                 */
/*                         */
/*  Comprobado con:        */
/*    - Turbo C++ 1.01     */
/*    - Symantec C++ 6.0   */
/*    - GCC 2.6.3          */
/*-------------------------*/

#include stdio.h
char letra;
main()
{
for (letra=’z’; letra>=’a’; letra-=2)
printf(“%c “, letra);
}

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

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 queremos que sea más de una sentencia, basta agruparlas entre { y }. Un ejemplo que nos diga el triple de cada número que tecleemos podría ser:

```
/*-------------------------*/
/*  Ejemplo en C nº 29     */
/*                         */
/*  Escribe el triple de   */
/*  los números tecleados, */
/*  con “while”            */
/*                         */
/*  Comprobado con:        */
/*    - Turbo C++ 1.01     */
/*    - Symantec C++ 6.0   */
/*    - GCC 2.6.3          */
/*-------------------------*/

#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”. 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     */
/*                         */
/*  Clave de acceso numé-  */
/*  rica, con “do..while”  */
/*                         */
/*  Comprobado con:        */
/*    - Turbo C++ 1.01     */
/*    - Symantec C++ 6.0   */
/*    - GCC 2.6.3          */
/*-------------------------*/

#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”);
}

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

Tema 8. Constantes y tipos.

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     */
/*                         */
/*  Clave de acceso numé-  */
/*  rica, con “do..while”  */
/*  y “const”              */
/*                         */
/*  Comprobado con:        */
/*    - Turbo C++ 1.01     */
/*    - 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”.

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 modidicador “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.

Pero no son los únicos modificadores, tenemos otros como “extern”, que no voy a comentar porque cae fuera del propósito del curso.

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     */
/*                         */
/*  Clave de acceso numé-  */
/*  rica, con “do..while”  */
/*  y “#define”            */
/*                         */
/*  Comprobado con:        */
/*    - Turbo C++ 1.01     */
/*    - Symantec C++ 6.0   */
/*    - GCC 2.6.3          */
/*-------------------------*/

#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.

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     */
/*                         */
/*  Clave de acceso numé-  */
/*  rica retocada, con     */
/*  typedef y #define      */
/*                         */
/*  Comprobado con:        */
/*    - Turbo C++ 1.01     */
/*    - Symantec C++ 6.0   */
/*    - GCC 2.6.3          */
/*-------------------------*/

#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. También podemos usar “typedef” para dar un nombre corto a todo un struct:

```
/*-------------------------*/
/*  Ejemplo en C nº 34     */
/*                         */
/*  Uso de “typedef” con   */
/*  “struct”.              */
/*                         */
/*  Comprobado con:        */
/*    - Turbo C++ 1.01     */
```

```
/*    - Symantec C++ 6.0   */
/*    - GCC 2.6.3          */
/*-------------------------*/

#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);
}

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

Tema 9. Funciones.

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.

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. En C sólo existen funciones, pero éstas pueden devolver unos valores de tipo “nulo”, con lo cual salen a equivaler a un procedimiento. 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.

```
/*-------------------------*/
/*  Ejemplo en C nº 35     */
/*                         */
/*  Función “potencia”     */
/*                         */
/*  Comprobado con:        */
/*    - Turbo C++ 1.01     */
/*    - 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;
}

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”.
- 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.
- 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.

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

```
/*-------------------------*/
/*  Ejemplo en C nº 36     */
/*                         */
/*  Función “hola”         */
/*                         */
/*  Comprobado con:        */
/*    - Turbo C++ 1.01     */
/*    - 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 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.

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) */
/*                         */
/*  Función “potencia”;    */
/*  función “main”         */
/*                         */
/*  Comprobado con:        */
/*    - Turbo C++ 1.01     */
/*    - 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;
}

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

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     */
/*                         */
/*  Función “modif1”       */
```

```
/*  Parámetros por valor.  */
/*                         */
/*  Comprobado con:        */
/*    - Turbo C++ 1.01     */
/*    - 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. 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     */
/*                         */
/*  Función “modif2”       */
/*  Parámetros por ref.    */
/*                         */
/*  Comprobado con:        */
/*    - Turbo C++ 1.01     */
/*    - 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 comentanto 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”. De ahí viene eso de “paso por referencia”: no le decimos el valor de la variable, sino 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. 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 ú (n-1) ú (n-2) ú ... ú 3 ú 2 ú 1 Por otra parte, (n-1)! = (n-1) ú (n-2) ú (n-3) ú ... ú 3 ú 2 ú 1 Luego podemos escribir cada factorial en función del factorial del siguiente número: n! = n ú (n-1)! Pues esta es la definición recursiva del factorial, ni más ni menos. Esto, programando, se haría:

```
/*-------------------------*/
/*  Ejemplo en C nº 39     */
/*                         */
/*  “factorial” recursivo  */
/*                         */
/*  Comprobado con:        */
/*    - Turbo C++ 1.01     */
/*    - 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.

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 (cómo no): 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 concer 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 = &uacutenum;         “pos” ahora es la dirección de “num”
```

Por tanto, con el símbolo * indicamos que se trata de un puntero, y ú 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”. 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     */
/*                         */
/*  Introducción a los     */
/*  punteros.              */
/*                         */
/*  Comprobado con:        */
/*    - Turbo C++ 1.01     */
/*    - Symantec C++ 6.0   */
/*    - GCC 2.6.3          */
/*-------------------------*/

#include stdio.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);
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 hallamos 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.

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     */
/*                         */
/*  Punteros y arrays      */
/*                         */
/*  Comprobado con:        */
/*    - Turbo C++ 1.01     */
/*    - Symantec C++ 6.0   */
/*    - GCC 2.6.3          */
/*-------------------------*/

#include stdio.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; i10; 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; i10; i++)        /* Recorremos el array */
     printf(“%d “,num[i]);     /* Mostrando los valores */


printf(“\nValores del array (como puntero): “);
temporal=num;
   for (i=0; i10; 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.

Tema 11. 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     */
/*                         */
/*  Ficheros de texto (1)  */
/*                         */
/*  Comprobado con:        */
/*    - Turbo C++ 1.01     */
/*    - Symantec C++ 6.0   */
/*    - GCC 2.6.3          */
/*-------------------------*/

#include stdio.h
FILE* fichero;
char texto[80];

main()
{
fichero = fopen(“c:\\autoexec.bat”, “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 ® en un fichero de texto (t).

- 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.

- Despues repetimos una parte del programa hasta que se acabe el fichero, de lo que nos

informa “feof”.

- 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”.

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     */
/*                         */
/*  Ficheros de texto (2)  */
/*                         */
/*  Comprobado con:        */
/*    - Turbo C++ 1.01     */
/*    - Symantec C++ 6.0   */
/*    - GCC 2.6.3          */
/*-------------------------*/

#include stdio.h
FILE* fichero;
char texto[80];

main()
{
fichero = fopen(“basura.bat”, “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);
}

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

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);

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     */
/*                         */
/*  Ficheros sin tipo:     */
/*  Copiador elemental     */
/*                         */
/*  Comprobado con:        */
/*    - Turbo C++ 1.01     */
/*    - 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 leidos */

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 leido */
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 leido 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:

- La función "fseek" 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.

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. 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     */
/*                         */
/*  Cadenas de texto       */
/*                         */
/*  Comprobado con:        */
/*    - Turbo C++ 1.01     */
/*    - 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 */
printf(“En mayúsculas es %s.\n”,
strupr(texto1));
free(texto2);                /* Libero lo reservado */
}

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

N.

No definitivo.

Esta versión del curso no está terminada y puede contener errores. La versión definitiva será distribuida como programa (búscala como CC.ZIP ó CC???.ZIP en BBSs y CdRoms de revistas) y contendrá:

- Todos los ejemplos comprobados con los 3 compiladores (Turbo C++, Symantec C++, GCC), porque alguno aún está sin comprobar, especialmente con GCC, que trabaja bajo Linux.
- Las lecciones releidas con tranquilidad, porque es posible que se me haya colado alguna errata por ahí.
- Lo que amplíe los temas en la versión 1.0, claro.
- Un índice, que diga en qué tema se ha tratado cada concepto o cada orden.
- Una lista de palabras reservadas.

- Una referencia rápida con las principales órdenes de Pascal y su equivalente en C.
- Las cosas que he resumido al crear esta página (pocas) y que en el original sí estaban.
- Seguro que algo más...

Notas finales.

Sólo algunos comentarios para quien haya conseguido llegar hasta aquí: Este curso tiene 46 ejemplos, y el texto de este curso ocupa más de 3500 líneas y más de 110K (casi 130K en formato HTML). 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.

Yo mismo estoy creando una **versión ampliada** de este curso (mire el tema 0: “Introducción” si quiere ver cómo conseguirla), que tendrá colores en el texto para una mayor legibilidad, que permitirá exportar los ejemplos e imprimir las lecciones, y que contendrá otras cosas como:

- Mayor detalle: por ejemplo, esta versión del curso no habla de uniones, ni de campos de bits, ni de “for” controlados con más de una variable, ni de otras tantas cosas “menos básicas”.
- Cómo acceder a la pantalla en modo texto (situar el cursor, cambiar colores, etc) con Turbo C++/Borland C++ y Symantec C++.
- Cómo crear gráficos con Turbo C++/Borland C++.
- Cómo crear programas a partir de más de un fuente.
- ... (Seguramente iré añadiendo más temas)
- Más ejemplos.

Pero insisto en que esta versión seguirá sin ser “el C”, siempre quedará mucho que se podría añadir, aunque pretendo que al menos lo necesario para “un uso normal” quede cubierto, y que cualquiera que quiera avanzar a partir de ahí lo haga ya con una cierta base...

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

N.

Si tiene dudas, ha visto algún error, le interesa la versión ampliada, o quiere encargar algún proyecto de programación a medida, puede contactar conmigo en la dirección:

ncabanes@arrakis.es

(Ultima modificacion: 31-Diciembre-96)