

Programación en C

Apuntes creados por:
José María Bea González

Algunos de los ejercicios han sido extraídos de www.elrincondec.com

Contenido

Programación en C	1
Tema 1: Introducción.....	4
Introducción.....	4
El lenguaje C	4
Compiladores y editores de C.....	4
Bibliotecas y enlazado	5
Compilación de un programa en C	5
Tema 2: Primeros programas. Instrucciones básicas.....	6
Instalación Turbo C	6
Primeros pasos con Turbo C.....	6
El primer programa: Hola mundo.....	6
Como ejecutar el programa.....	8
Imprimir en pantalla	8
Tema 3: Tipos de variables.....	11
Introducción.....	11
El tipo Int	11
El tipo Char.....	14
Otros tipos de datos:	15
Códigos de printf()	16
Ejercicios:	17
Tema 4: Operadores.....	19
Introducción.....	19
Operador de asignación:	19
Operadores aritméticos	19
Ejercicios:	20
Operadores relacionales (de comparación).....	21
Operadores lógicos	21
Operadores de bits	22
Tema 5: Sentencias condicionales y bucles.....	23
La sentencia IF.....	23
if – else-if anidados	23
Sentencia switch	24
Ejercicios:	25
Bucles	27
El bucle while	27
Bucle do while	27
Bucle for	28
break	28
exit()	29
continue	29
Ejercicios:	29
Tema 6: Arrays (Matrices)	31
Introducción:.....	31
Arrays de caracteres:	31
Arrays de varias dimensiones:.....	32

Inicialización de arrays:.....	32
Funciones de cadenas:	32
Ejemplo:	33
Ejercicios:	34
Tema 7: Funciones	35
Introducción:.....	35
Definición de la función	35
Llamada a la función	35
Cuerpo de la función.....	35
La sentencia return.....	37
Llamada por valor y llamada por referencia.....	37
Diferentes métodos de pasar un array por referencia a una función.	38
Variables globales:	39
Creación de bibliotecas:.....	39
Ejercicios:	39
Tema 8: Tipos de datos.....	43
Estructuras de datos	43
Enumeraciones.....	45
Tema: 9 Clasificación en memoria primaria.	47
Tema 10: Punteros	49
Variables	49
Malloc y free.....	50
Tema 11: Listas enlazadas	52
Introducción.....	52
¿En qué consisten?.....	52
Ejercicios:	54
Paso de punteros a funciones:.....	55
Liberar la memoria de una lista enlazada:	56
Tema 12: Ficheros	57
Introducción.....	57
Redireccionamiento	57
freopen	58
Lectura de ficheros	58
Ejercicios:	60
Escritura en ficheros:	60
Ejercicios:	61
fprintf y fscanf	61
fseek y ftell	61
Ejercicios:	62
Ficheros binarios	63
Apéndice A: Puliendo nuestros programas	66

Tema 1: Introducción

Introducción

Las razones de haber escogido el lenguaje C para comenzar a programar son varias: es un lenguaje muy potente, se puede aprender rápidamente. Este es sin duda el objetivo de éste curso. No pretende ser un completo manual de la programación, sino una base útil para que cualquiera pueda introducirse en este apasionante mundo.

El lenguaje C es uno de los más rápidos y potentes que hay hoy en día. el sistema operativo Linux está desarrollado en C en su práctica totalidad. Así que creo que no sólo no perdemos nada aprendiéndolo sino que ganamos mucho. Para empezar nos servirá como base para aprender C++ e introducirnos en el mundo de la programación de Windows.

No debemos confundir C con C++, que no son lo mismo. Se podría decir que C++ es una extensión de C. Para empezar en C++ conviene tener una sólida base de C. Existen otros lenguajes como Visual Basic que son muy sencillos de aprender y de utilizar; nos dan casi todo hecho. Pero cuando queremos hacer algo complicado o que sea rápido debemos recurrir a otros lenguajes (c++, delphi,...).

El lenguaje C

El lenguaje C fue inventado por Dennis Ritchie en 1972, tiene como antecedente el lenguaje B diseñado por Ken Thompson en 1970.

C es un lenguaje de **nivel medio**, ya que combina elementos de lenguaje de alto nivel con la funcionalidad del lenguaje ensamblador (bajo nivel).

Generalmente se diferencian dos tipos de lenguajes, los de *alto nivel* y los de *bajo nivel*.

Lenguajes de alto nivel: Son los que están más cerca del usuario, esto es, que es más fácil de comprender para el hombre. En este grupo tenemos lenguajes como el Pascal, Basic, Cobol, Modula-2, Ada, etc...

Lenguajes de bajo nivel: Estos lenguajes están más cerca de la máquina y son más complicados para el hombre. En este grupo tenemos el ensamblador y el código máquina (para programar con este último se haría directamente con 1's y 0's).

C es un lenguaje estructurado: permite compartir porciones de código. Más adelante veremos que habrá veces en las que debemos hacer una misma función varias veces en un mismo programa, para no repetir el mismo código una y otra vez tendremos la posibilidad de usar funciones que nos facilitarán mucho la tarea. Otros lenguajes estructurados son Ada, Modula-2, Pascal... Y lenguajes no estructurados son el Basic, Cobol, etc...

Compiladores y editores de C

Un **compilador** es un programa que convierte nuestro código fuente en un programa ejecutable. El ordenador trabaja con 0 y 1. Si escribiéramos un programa en el lenguaje del ordenador nos volveríamos locos. Para eso están lenguajes como el C. Nos permiten escribir un programa de manera que sea fácil entenderlo por una persona (el código

fuente). Luego es el compilador el que se encarga de convertirlo al complicado código de un ordenador.

El compilador en sí mismo sólo es un programa que traduce nuestro código fuente y lo convierte en un ejecutable. Para escribir nuestros programas necesitamos un editor. La mayoría de los compiladores al instalarse incorporan ya un editor; es el caso de los conocidos Turbo C, Borland C, Visual C++,... Pero otros no lo traen por defecto. No debemos confundir por tanto el editor con el compilador. Estos editores suelen tener unas características que nos facilitan mucho el trabajo: permiten compilar y ejecutar el programa directamente, depurarlo (corregir errores), gestionar complejos proyectos,...

Bibliotecas y enlazado

Es posible crear un programa en C desde cero y sin ningún tipo de ayuda, utilizando únicamente funciones creadas por nosotros, pero la mayoría de los programas incluyen una serie de funciones muy usuales contenidas en bibliotecas. Por ejemplo, cuando queramos sacar un mensaje de texto por pantalla, dispondremos una función que realiza esta tarea, sin embargo si quisiéramos darnos el gusto de hacer esto sin ninguna función prediseñada necesitaríamos unos conocimientos mucho más amplios y tamaño del programa se multiplicaría varias veces.

Cuando el compilador detecta el nombre de una función que no ha definido el programador, más tarde el enlazador (o linkador) busca la función en las librerías que hemos declarado al inicio del programa.

Compilación de un programa en C

Tiene cuatro pasos:

- 1.- Creación del programa
- 2.- Compilación del programa
- 3.- Enlazado del programa
- 4.- Ejecución del programa

Tema 2: Primeros programas. Instrucciones básicas

Instalación Turbo C

El editor-compilador que vamos a utilizar es Turbo C 3.0. Cada compilador tiene sus características y alguna vez nos encontraremos que un programa que compila con un compilador, con otro da algún tipo de error. Existe un C estándar pero alguna vez nos encontraremos con alguna excepción.

Su instalación es bien muy sencilla. Ejecutaremos el fichero install.exe y seguiremos las instrucciones que aparecen en pantalla. Lo instalaremos en la carpeta C:\TC

Primeros pasos con Turbo C

Para ejecutar TC (Turbo C) deberemos acceder a su carpeta bien desde el explorador de Windows o desde MSDOS. Los pasos a seguir desde MSDOS son los siguientes:

```
C:\>CD TC
C:\TC>CD BIN
C:\TC\BIN>TC
```

En caso de hacerlo desde el explorador de Windows accederemos a la carpeta C:\TC\BIN\ y ejecutaremos el fichero TC.EXE

Si estamos acostumbrados al entorno de Windows notaremos que la pantalla tiene menos definición y menos colores, pero no debemos asustarnos, el entorno es muy parecido. Cuando queramos acceder a la primera línea (la de menús) haremos un clic en la opción deseada o bien accederemos con la tecla ALT + Letra subrayada. También podremos desplazarnos con los cursores.

La zona de color azul es la ventana de edición, en ella escribiremos el código fuente de los programas. En la parte inferior aparecerá 1:1 (a no ser que nos hayamos movido), esto nos indica la fila y la columna respectivamente.

El primer programa: Hola mundo

En un alarde de originalidad vamos a hacer nuestro primer programa: hola mundo. Nadie puede llegar muy lejos en el mundo de la programación sin haber empezado su carrera con este original y funcional programa. Allá va:

```
#include <stdio.h>

void main()
{
/* Aquí va el cuerpo del programa */
printf( "Hola mundo" ); /* Esto imprime hola mundo en pantalla */
printf("\nEste es mi primer programa");
}
```

¿Qué fácil eh? Este programa lo único que hace es sacar por pantalla el mensaje:
Hola mundo

Vamos ahora a comentar el programa línea por línea (Esto no va a ser más que una primera aproximación).

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

`#include` es lo que se llama una directiva. Sirve para indicar al compilador que incluya otro archivo. Cuando en compilador se encuentra con esta directiva la sustituye por el archivo indicado. En este caso es el archivo `stdio.h` que es donde está definida la función `printf`, que veremos luego. `stdio.h` es una librería o biblioteca, dentro de este fichero hay definidas una serie de funciones. Más adelante veremos diferentes librerías en las que encontraremos muchas más funciones.

```
void main()
```

Es la función principal del programa. Todos los programas de C deben tener una función llamada `main`. Es la que primero se ejecuta. El `void` (vacío) que tiene al principio significa que cuando la función `main` acabe no devolverá nada.

Se puede usar la definición `'void main()'`, que no necesita devolver ningún valor, pero se recomienda la forma con `'int'` que es más correcta, de esta forma se devuelve un `1` si el programa no ha acabado correctamente y un `0` si lo ha hecho de forma correcta. A lo largo de este curso verás muchos ejemplos que uso `'void main'`.

```
{
```

Son las llaves que indican el comienzo de una función, en este caso la función `main`.

```
/* Aquí va el cuerpo del programa */
```

Esto es un comentario, no se ejecuta. Sirve para describir el programa. Conviene acostumbrarse a comentar los programas. Un comentario puede ocupar más de una línea. Por ejemplo el comentario:

```
/* Este es un comentario  
   que ocupa dos filas */
```

es perfectamente válido.

```
printf( "Hola mundo" );  
printf ( "\nEste es mi primer programa" );
```

Aquí es donde por fin el programa hace algo que podemos ver al ejecutarlo. La función `printf` muestra un mensaje por la pantalla. Después del mensaje “Hola mundo” volvemos a escribir `printf`, esta vez hemos escrito “\n” al principio de la línea, este sirve para hacer un salto de línea.

Fíjate en el “;” del final. Es la forma que se usa en C para separar una instrucción de otra. Se pueden poner varias en la misma línea siempre que se separen por el punto y coma.

```
}
```

...y cerramos llaves con lo que termina el programa. Todos los programas finalizan cuando se llega al final de la función `main`.

Como ejecutar el programa

Para **compilar** el programa iremos al menú COMPILE → COMPILE y aparecerá una ventana en la que se nos indicará si tenemos algún error. En caso de tenerlo se nos indicará la línea (si no encontramos ningún error en la línea que nos indica deberemos mirar en la anterior y posterior).

Para **linkar** el programa seleccionaremos COMPILE → MAKE.

Una vez hecho esto ya podremos **ejecutar** el programa, esto lo haremos desde RUN → RUN. Si no hemos hecho los pasos anteriores y seleccionamos esta opción el programa se encargará de hacer los 2 pasos anteriores. Para realizar esta opción también podemos usar la tecla rápida Ctrl + F9

Para ver el resultado de nuestro programa podremos presionar Alt + F5.

Una vez linkado el programa aparecerá un fichero .EXE, este es nuestro programa. Si queremos hacer alguna modificación buscaremos el código fuente, que tendrá la extensión .CPP o .C.

Imprimir en pantalla

Ya hemos visto la función `printf` que nos sirve para imprimir un mensaje en pantalla, vamos a ver otras funciones que se utilizan al imprimir por pantalla. Las 3 primeras funciones que siguen se encuentra en la librería `<conio.h>`, la otra en la `<stdio.h>`

```
clrscr();
```

Con esta función borraremos la pantalla.

```
gotoxy (columna, fila);
```

Sitúa el cursor en la posición indicada. En una pantalla de texto hay 25 filas y 80 columnas (la primera posición es la 1 y la última la 25 u 80).

```
getch();
```

Con esta instrucción el programa esperará a que pulsemos una tecla antes de finalizar.

```
puts ("mensaje");
```

Escribe un texto por pantalla.

Ejercicios

Ejercicio 1: Busca los errores en el programa.

```
#include <stdio.h>

int main()
{
    ClrScr();
    gotoxy( 10, 10 )
    printf( Estoy en la fila 10 columna 10 );
}
```

(*Nota: int main () no es ningún error, podemos poner int en vez de void y el programa también funcionará, más adelante, cuando veamos las funciones se verá la diferencia)

Ejercicio 2. Escribe un programa que borre la pantalla y escriba en la primera línea su nombre y en la segunda su apellido:

Ejercicio 3. Escriba un programa que borre la pantalla y muestre el texto "Estoy aquí" en la fila 10, columna 20 de la pantalla y que no finalice hasta que pulsemos una tecla.

Ejercicio 4. Busca los fallos de este programa.

```
#include <stdio.h>

void main(void);
{
    clrscr(); puts ( "Programa con fallos en STARNET" );
    printf("\nAlgunos tienen truco" );
};
```

Soluciones a los ejercicios:

Solución ejercicio 1:

- ClrScr está mal escrito, debe ponerse todo en minúsculas, recordemos una vez más que el **C** diferencia las mayúsculas de las minúsculas. Además no hemos incluido la directiva `#include <conio.h>`, que necesitamos para usar `clrscr()` y `gotoxy()`.
- Tampoco hemos puesto el punto y coma (;) después del `gotoxy(10, 10)`. Después de cada instrucción debe ir un punto y coma.
- El último fallo es que el texto del `printf` no lo hemos puesto entre comillas. Lo correcto sería: `printf("Estoy en la fila 10 columna 10");`

Solución ejercicio 2:

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

int main()
{
    clrscr();
    printf( "Academia\n" );
    printf( "Starnet" );
}
```

También se podía haber hecho todo de golpe:

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

int main()
{
    clrscr();
    printf( "Academia\nStarnet" );
}
```

Solución ejercicio 3:

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

void main()
{
    clrscr();
    gotoxy( 20, 10 );
    printf( "Estoy aquí" );
    getch();
}
```

Solución ejercicio 4:

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

void main(void)
{
    clrscr(); puts ( "Programa con fallos en STARNET" );
    printf("\nAlgunos tienen truco" );
}
```

Los fallos son:

1. Se necesita la librería <conio.h>
2. Después de void main(void) no va “;”
3. Después de las llaves finales no va “;”

No son fallo:

1. void main (void)
2. Poner dos instrucciones en la misma línea (si están separados por ;).

Tema 3: Tipos de variables

Introducción

Cuando usamos un programa es muy importante manejar datos. En **C** podemos almacenar los datos en variables. El contenido de las variables se puede ver o cambiar en cualquier momento. Estas variables pueden ser de distintos tipos dependiendo del tipo de dato que queramos meter. No es lo mismo guardar un nombre que un número.

Hay que recordar también que la memoria del ordenador es limitada, así que cuando guardamos un dato, debemos usar sólo la memoria necesaria. Por ejemplo si queremos almacenar el número 400 usaremos una variable tipo *int* (la estudiamos más abajo) que ocupa sólo 16 bits, y no una de tipo *long* que ocupa 32 bits. Si tenemos un ordenador con 32Mb de RAM parece una tontería ponernos a ahorrar bits (1Mb=1024Kb, 1Kb=1024 bytes, 1byte=8 bits), pero si tenemos un programa que maneja muchos datos puede no ser una cantidad despreciable. Además ahorrar memoria es una buena costumbre.

(No hay que confundir la memoria con el espacio en el disco duro. Son dos cosas distintas. La capacidad de ambos se mide en bytes, y la del disco duro suele ser mayor que la de la memoria RAM. La información en la RAM se pierde al apagar el ordenador, la del disco duro permanece. Cuando queremos guardar un fichero lo que necesitamos es espacio en el disco duro. Cuando queremos ejecutar un programa lo que necesitamos es memoria RAM. La mayoría me imagino que ya lo sabéis, pero me he encontrado muchas veces con gente que los confunde.)

Variable: Es un dato que puede cambiar su valor a lo largo del programa

Notas sobre los nombres de las variables

El nombre de las variables tiene algunas restricciones:

- Se pueden poner letras las de la ‘a’ a la ‘z’ minúsculas o mayúsculas, números y el signo ‘_’
- No se puede poner la letra ‘ñ’
- No pueden comenzar por un número.
- No pueden contener signos de admiración, interrogación, caracteres acentuados, ni símbolos “raros”.

El tipo Int

En una variable de este tipo se almacenan números enteros (sin decimales). El rango de valores que admite es -32768 a 32767. Cuando definimos una variable lo que estamos haciendo es decirle al compilador que nos reserve una zona de la memoria para almacenar datos de tipo *int*. Para guardarla necesitaremos 16 bits de la memoria del ordenador ($2^{16}=65536$, y ese número nos indica los valores diferentes que puede coger la variable *int*). Para poder usar una variable primero hay que declararla (definirla). Hay

que decirle al compilador que queremos crear una variable y hay que indicarle de qué tipo. Por ejemplo:

```
int numero;
```

Esto hace que declaremos una variable llamada *numero* que va a contener un número entero.

¿Pero dónde se declaran las variables?

Tenemos dos posibilidades, una es declararla como global y otra como local. Por ahora vamos a decir que global es aquella variable que se declara fuera de la función main y local la que se declara dentro:

Variable Global

```
#include <stdio.h>

int x;

int main()
{
}
```

Variable Local

```
#include <stdio.h>

int main()
{
    int x;
}
```

La diferencia práctica es que las variables globales se pueden usar en cualquier procedimiento. Las variables locales sólo pueden usarse en el procedimiento en el que se declaran. Como por ahora sólo tenemos el procedimiento (o función, o rutina, o subrutina, como prefieras) `main` esto no debe preocuparnos mucho por ahora. Cuando estudiemos cómo hacer un programa con más funciones aparte de `main` volveremos sobre el tema. Sin embargo debes saber que es buena costumbre usar variables locales que globales. Ya veremos por qué.

Podemos declarar más de una variable en una sola línea:

```
int x, y;
```

Mostrar variables por pantalla

Vamos a ir un poco más allá con la función `printf`. Supongamos que queremos mostrar el contenido de la variable

`x`
por pantalla:

```
printf( "%i", x );
```

Suponiendo que `x` valga 10 (`x=10`) en la pantalla tendríamos:

```
10
```

Empieza a complicarse un poco ¿no? Vamos poco a poco. ¿Recuerdas el símbolo `"\"` que usábamos para sacar ciertos caracteres? Bueno, pues el uso del `"%"` es parecido.

"%i" no se muestra por pantalla, se sustituye por el valor de la variable que va detrás de las comillas. (%i, de integer=entero en inglés).

Para ver el contenido de dos variables, por ejemplo x e y, podemos hacer:

```
printf( "%i ", x );  
printf( "%i", y );
```

resultado (suponiendo x=10, y=20):

```
10 20
```

Pero hay otra forma mejor:

```
printf( "%i %i", x, y );
```

... y así podemos poner el número de variables que queramos. Obtenemos el mismo resultado con menos trabajo. No olvidemos que por cada variable hay que poner un %i dentro de las comillas.

También podemos mezclar texto con enteros:

```
printf( "El valor de x es %i, ¡que bien!\n", x );
```

que quedará como:

```
El valor de x es 10, ¡que bien!
```

Como vemos %i al imprimir se sustituye por el valor de la variable.

Asignar valores a variables de tipo int

La asignación de valores es tan sencilla como:

```
x = 10;
```

También se puede dar un valor inicial a la variable cuando se define:

```
int x = 15;
```

También se pueden inicializar varias variables en una sola línea:

```
int x = 15, y = 20;
```

Hay que tener cuidado con lo siguiente:

```
int x, y = 20;
```

Podríamos pensar que x e y son igual a 20, pero no es así. La variable x está sin valor inicial y la variable 'y' tiene el valor 20.

Veamos un ejemplo para resumir todo:

```
#include <stdio.h>

void main()
{
    int x = 10;

    printf( "El valor inicial de x es %i\n", x );
    x = 50;
    printf( "Ahora el valor es %i\n", x );
}
```

Cuya salida será:

```
El valor inicial de x es 10
Ahora el valor es 50
```

Importante! Si imprimimos una variable a la que no hemos dado ningún valor no obtendremos ningún error al compilar pero la variable tendrá un valor cualquiera. Prueba el ejemplo anterior quitando

```
int x = 10;
```

Puede que te imprima el valor 10 o puede que no.

El tipo Char

Las variables de tipo char sirven para almacenar caracteres. Los caracteres se almacenan en realidad como números del 0 al 255. Los 128 primeros (0 a 127) son el ASCII estándar. El resto es el ASCII extendido y depende del idioma y del ordenador.

Para declarar una variable de tipo char hacemos:

```
char letra;
```

En una variable char **sólo podemos almacenar solo una letra**, no podemos almacenar ni frases ni palabras. Eso lo veremos más adelante (strings, cadenas). Para almacenar un dato en una variable char tenemos dos posibilidades:

```
letra = 'A';
o
letra = 65;
```

En ambos casos se almacena la letra 'A' en la variable. Esto es así porque el código ASCII de la letra 'A' es el 65.

Para imprimir un char usamos el símbolo %c (c de character=carácter en inglés):

```
letra = 'A';
printf( "La letra es: %c.", letra );
```

Resultado:

```
La letra es A.
```

También podemos imprimir el valor ASCII de la variable usando %i en vez de %c:

```
letra = 'A';  
printf( "El número ASCII de la letra %c es: %i.",  
letra, letra );
```

Resultado:

```
El código ASCII de la letra A es 65.
```

Como vemos la única diferencia para obtener uno u otro es el modificador (%c ó %i) que usemos.

Las variables tipo char se pueden usar (y de hecho se usan mucho) para almacenar enteros. Si necesitamos un número pequeño (entre -127 y 127) podemos usar una variable char (8bits) en vez de una int (16bits), con el consiguiente ahorro de memoria.

Todo lo demás dicho para los datos de tipo

int
se aplica también a los de tipo
char

Una curiosidad:

```
letra = 'A';  
printf( "La letra es: %c y su valor ASCII es: %i\n",  
letra, letra );  
letra = letra + 1;  
printf( "Ahora es: %c y su valor ASCII es: %i\n",  
letra, letra );
```

Otros tipos de datos:

Disponemos de otros tipos de datos y de unos modificadores que ahora veremos. El funcionamiento es el mismo que en los casos anteriores (char e int). Aquí tenemos una tabla resumen que nos ayudará a recordar todos estos tipos.

Tipo	Bytes	Rango	printf()
char	1	-128 a 127	%c
int	2	-32.768 a 32.767	%i
long	4	-2.147.483.648 a 2.147.483.647	%li
float	4	Aprox. 6 dígitos → Reales	%f, %e
double	8	Aprox 12 dígitos → Reales	%lf
unsigned char	1	0 a 255	%c

unsigned int	2	0 a 65.535	%u
short int	1	-128 a 127	%i
unsigned short int	1	0 a 255	%i
long int	4	-2.147.483.648 a 2.147.483.647	%li
unsigned long int	4	0 a 4.294.967.296	%i

Es importante conocer las limitaciones de cada uno de estos tipos. Los tipos float y double utilizan decimales y si tenemos un número muy grande puede ser que perdamos decimales por redondeo (sólo hay que fijarse en el tipo long y el tipo float, que tienen el mismo número de bytes).

sizeof(variable) ;

La función sizeof() nos devolverá el número de bytes que ocupa una variable.

Códigos de printf()

Ahora que vamos a representar variables por pantalla nos vendrá muy bien conocer códigos que podremos usar con esta función.

Nombre completo	en C	ASCII
Sonido de alerta	\a	7
Nueva línea	\n	10
Tabulador horizontal	\t	9
Retroceso	\b	8
Retorno de carro	\r	13
Salto de página	\f	12
Barra invertida	\\	92
Apóstrofo	'\''	39
Comillas	"\""	34
Carácter nulo	\0	0

Ejercicio 1:

Para poner en práctica lo visto vamos a preparar un programa que nos ayudará a comprender todo lo anterior.

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

void main()
{
char c=0;
int i=0;
long l;
float f=0;
double d=0;
clrscr();
```

```
printf("\n\tEste programa muestra propiedades de los
Tipos de variables\n\n");
printf( "Tamaño (en bytes) de char = %i\n", sizeof( c ));
printf( "Tamaño (en bytes) de int = %i\n", sizeof( i ));
printf( "Tamaño (en bytes) de long = %i\n", sizeof( l ));
printf( "Tamaño (en bytes) de float = %i\n", sizeof( f ));
printf( "Tamaño (en bytes) de double = %i\n", sizeof( d ));
printf("\n");
c = 127;
printf("\nchar = %i", c);

i = 32767;
printf("\nint = %i", i);

l = 2147483647;
printf("\nlong = %li", l);

f = 2147483647;
printf("\nfloat = %f", f);

d = 716548.2648;
printf("\ndouble = %lf", d);
}
```

scanf () ;

Esta función nos abrirá fronteras, gracias a ella podremos leer valores introducidos por teclado y convertirlos en el tipo de variable apropiado.

Su sintaxis es:

```
scanf("Cadena de control", lista argumentos);
```

En la cadena de control indicaremos el tipo de dato a leer y en la lista de argumentos indicaremos la variable en la que se guardará la información. Todas las variables irán precedidas del símbolo &.

```
scanf("%i", &entero);
```

Ejercicios:

- 1.-Haremos un programa que nos pregunte 3 números (long int) y después nos los imprima.
- 2.-Programa que nos pida un número, un float, y un char y nos lo imprima (primero lo haremos utilizando 3 printf a la hora de imprimir y después usando 1 sólo).
- 3.-Aunque será en el siguiente tema cuando trataremos a fondo los operadores, vamos a hacer un programa que nos pregunte el año de nacimiento, el año actual, y nos calcule

nuestra edad. Usaremos 2 operadores, el de asignación y el de suma (*en la práctica será algo así como $edad = nacimiento - actual$*).

*(*Nota: Recordad las restricciones a la hora de declarar variables).*

Tema 4: Operadores

Introducción

Un operador es un símbolo que le indica al compilador que debe realizar alguna operación matemática o lógica. Tenemos varios tipos de operadores:

- Asignación
- Relación
- Lógicos
- Aritméticos
- Manipulación de bits

Operador de asignación:

Este operador ya lo hemos visto en programas anteriores, sirve para dar un valor a una variable. Este valor puede ser un número o hacer referencia a otra variable.

```
a = 5;           /*La variable a valdrá 5*/  
b = a;          /*La variable b cogerá el mismo valor que  
                tiene a, en este caso 5*/
```

Podemos asignar un valor a varias variables a la vez:

```
a = b = c = 10;  
a = b = c = d;
```

Operadores aritméticos

Los operadores aritméticos permitidos son:

- Resta, o cambiar de signo
- + Suma
- * Multiplicación
- / División
- % División en módulo (resto)
- Decremento
- ++ Incremento

Además podremos utilizar paréntesis para facilitar la comprensión de las fórmulas.

Podemos utilizar este operador para incrementar el valor de una variable:

```
x = x + 5; /*La x incrementará su valor en 5*/
```

Pero existe una forma abreviada:

```
x += 5;
```

El operador incremento:

```
#include <stdio.h>

void main()
{
    int x = 5;

    printf ( "Valor de x = %i\n", x );
    x++; /*Es lo mismo que: x = x + 1;*/
    printf ( "Valor de x = %i\n", x );
}
```

Aquí tenemos un ejemplo de cómo calcular el resto de 2 números. El resultado en este caso será 4, que es el resto de dividir 19 entre 5.

```
#include <stdio.h>

void main()
{
    int a, b;
    a = 19;
    b = 5;
    printf( "Resto de dividir %i entre %i es: %d \n", a,
b, a % b ); /*%d es lo mismo que %i*/
}
```

Ejercicios:

- 1.- Crea un programa que calcule la velocidad media en km/h y m/s
- 2.- Crea un programa que calcule el resto de 2 números, otro que calcule la suma, otro la resta, multiplicación y división (5 programas).
- 3.- Crea un programa que calcule el área y perímetro de un cuadrado, otro la del triángulo y otro la de la circunferencia. Para definir el número PI puedes escribir antes de la función main `#define PI 3.1415926`
- 4.- Ahora uno que calcule el área y volumen del cilindro.
- 5.- Uno que resuelva ecuaciones de 1er grado y otro de 2º grado.

Consejos para los ejercicios:

Usar paréntesis dará más claridad a los cálculos, por ejemplo:

```
x=x/2-38*temp/17
x = (x/2) - ((38 * temp) / 17)
```

Las 2 expresiones tendrán el mismo resultado pero la segunda es mucho más clara.

Fórmulas: Circunferencia: Área = $\Pi * R^2$; Perímetro = $2 \Pi R$

Ecuaciones de 1er grado: $ax+b=0$; $x = -b/a$

Ecuaciones de 2º grado: $ax^2+bx+c=0$; $x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$ (Para hacer la raíz cuadrada debemos incluir la librería `<math.h>` y utilizar la función `sqrt()` .

Operadores relacionales (de comparación)

Este tipo de operadores se utilizan para comprobar condiciones de las sentencias de control de flujo que veremos en el capítulo siguiente.

Cuando evaluemos una condición podremos obtener 2 posibles valores, verdadero (1) o falso (0).

Los operadores de comparación son:

==	igual que	se cumple si son iguales
!=	distinto que	se cumple 1 si son diferentes
>	mayor que	se cumple si el primero es mayor que el segundo
<	menor que	se cumple si el primero es menor que el segundo
>=	mayor o igual que	se cumple si el primero es mayor o igual que el segundo
<=	menor o igual que	se cumple si el primero es menor o igual que el segundo

Ejemplo:

```
#include <stdio.h>
int main()
{
    printf( "10 > 5 da como resultado %i\n", 10>5 );
    printf( "10 < 5 da como resultado %i\n", 10<5 );
    printf( "5 <= 5 da como resultado %i\n", 5<=5 );
    printf( "10==5 da como resultado %i\n", 10==5 );
}
```

Operadores lógicos

Al igual que los operadores anteriores, la utilidad de este tipo de operadores se verá cuando evaluemos condiciones.

Estos operadores también nos devolverán como resultado, verdadero (1) o falso (0).

Estamos muy familiarizados al uso de estos operadores en el lenguaje común, aunque no siempre seamos muy estrictos en su uso. Estos operadores son:

Operador	Nombre	En Castellano	Acción
&&	AND	Y	Devuelve 1 si todas las condiciones son ciertas
	OR	O	Devuelve 1 si al menos una condición es cierta
^	XOR	O exclusiva	Devuelve 1 si sólo una de las condiciones es cierta
!	NOT	Negar	Invertimos en resultado, si teníamos un 1 cambiamos a 0 y viceversa

Operadores de bits

Los bits son la unidad de información más pequeña, digamos que son la base para almacenar la información. Son como los átomos a las moléculas. Los valores que puede tomar un bit son 0 ó 1. Si juntamos ocho bits tenemos un byte.

Un byte puede tomar 256 valores diferentes (de 0 a 255). ¿Cómo se consigue esto? Imaginemos nuestro flamante byte con sus ocho bits. Supongamos que los ocho bits valen cero. Ya tenemos el valor 0 en el byte. Ahora vamos a darle al último byte el valor 1.

```
00000001 -> 1
```

Este es el uno para el byte. Ahora vamos a por el dos y el tres:

```
00000010 -> 2  
00000011 -> 3
```

y así hasta 255. Como vemos con ocho bits podemos tener 256 valores diferentes, que en byte corresponden a los valores entre 0 y 255.

Ya hemos visto que un byte son ocho bits. Pues bien, con los operadores de bits podemos manipular las variables *por dentro*. Los diferentes operadores de bits son:

Operador	Descripción
	OR (O)
&	AND (Y)
^	XOR (O exclusivo)
~	Complemento a uno o negación
>>	Desplazamiento a la derecha
<<	Desplazamiento a la izquierda

Tema 5: Sentencias condicionales y bucles

Hasta ahora todos los programas que hemos visto eran lineales, se ejecutaban todas las instrucciones una vez. Suele interesarnos que algunas instrucciones se ejecuten y otras no, que algunas se ejecuten una sola vez y otras se ejecuten un determinado número de veces.

La sentencia IF

La palabra if significa si (condicional). La sintaxis de esta función es:

```
if (condición) sentencia1;
[else sentencia2;]
```

Cuando pongamos la condición podremos utilizar los operadores vistos en el capítulo anterior. Si la condición se cumple se ejecutará la sentencia1, si no se cumple se ejecutará la sentencia2. No es necesario poner else después del if, se puede poner si queremos que en caso de que no se cumpla una condición haga algo el programa.

Esto se entenderá con el siguiente ejemplo. Este sencillo programa nos pedirá una clave, si acertamos la clave nos dará la enhorabuena, en caso contrario se nos avisará de que la contraseña es incorrecta.

```
#include <stdio.h>
void main()
{
int contrasena = 2002; /*Nótese que no ponemos '\n'*/
int i;

printf( "\n\n\tIntroduce la contraseña: " );
scanf("%i", &i);

if ( i == contrasena)
{
printf ("\nCONTRASEÑA CORRECTA");
}
else
{
printf ("\a\nACCESO DENEGADO, CONTRASEÑA INCORRECTA");
}
}
```

if – else-if anidados

```
#include <stdio.h>
void main()
{
int a;
printf( "Introduce un número " );
scanf( "%i", &a );
if ( a < 10 )
```

```
        {
            printf ( "El número introducido era menor de 10.\n" );
        }
    else if ( a>10 && a<100 )
    {
        printf ( "El número está entre 10 y 100\n" );
    }
    else if ( a>100 )
    {
        printf( "El número es mayor que 100\n" );
    }
    else printf("El valor 10 ó 100");
    printf( "Fin del programa\n" );
}
```

En el anterior programa se nos indicará en qué rango está el número introducido. Aparece algo nuevo y es else if, esto indica al compilador que si no se cumple la condición se comprobará otra condición, en el ejemplo ($a > 10 \ \&\& \ a < 100$).

El último else se ejecutaría si no se cumplen los anteriores. En el ejemplo se ejecutaría en caso de valer 10 ó 100.

Si un if o un else tienen una sola sentencia dentro, podremos prescindir de utilizar llaves.

Sentencia switch

Utilizar if-else-if es poco elegante y puede confundirnos, así que utilizaremos esta función que nos será de gran utilidad en caso de que queramos crear un menú para nuestro programa.

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

void main()
{
    char opcion = 'A';

    clrscr();
    puts ("MENÚ DE OPCIONES");
    puts ("1.-Bienvenida al programa");
    puts ("2.-Saludo");
    puts ("3.-Despedir");
    puts ("4.-Salir");

    opcion = getch();
    switch (opcion)
    {
        case '1': puts ("Bienvenido");
                break;
        case '2': puts ("Hola");
                break;
        case '3': puts ("Adiós");
                break;
        case '4': break;
        default: printf ("\aOPCIÓN INCORRECTA");
    }
}
```

```
    }  
    getch();  
}
```

Ejercicios:

1. Haz un programa que nos indique si un número es par o impar.
2. Programa que nos identifique si una letra es vocal o consonante.
3. Crea un programa que intercambie el valor de 2 variables.
4. Programa que ordene 3 números enteros.
5. Programa que calcule áreas de distintas figuras: -Cuadrado, rectángulo, triángulo, circunferencia y cilindro (pista: se calcula sumando el área de las 2 bases y después la de el contorno que si lo extendemos es un rectángulo).
6. Haz un programa parecido al número 1 que nos indique si un número es par o impar, y en caso de ser par si además es divisible entre 4.
7. Un año es bisiesto si es divisible entre 4 excepto si es divisible entre 100, aunque los años divisibles entre 400 también son bisiestos. Crea un programa que te indique si un año es bisiesto o no.
8. Programa una Eurocalculadora. La Eurocalculadora nos mostrará un menú en el que indicaremos si queremos pasar de Euros a Ptas o viceversa. (Nota: Cuando quieras imprimir un número concreto de decimales puedes especificarlo de esta forma: `%.2f`)

Solución al ejercicio 2

```
#include <stdio.h>  
  
void main ()  
{  
    char c;  
    short int i = 0;  
  
    while (c != '.')  
    {  
        scanf ("%c", &c);  
        if (c == 'a' || c == 'e' || c == 'i' || c == 'o' || c == 'u') i ++;  
    }  
  
    printf ("\nVocales: %i", i);  
}
```

Solución ejercicio 4

```
#include <stdio.h>
#include <conio.h>
void main ()
{
int n1, n2, n3, aux;
clrscr();
printf("\n\tEste programa ordena 3 números");
printf("\nIntroduce 3 números presionando ENTER cada
vez\n");
scanf ("%i", &n1);
scanf ("%i", &n2);
scanf ("%i", &n3);

if (n1 > n2)
{
aux = n1;
n1 = n2;
n2 = aux;
}

if (n1 > n3)
{
aux = n1;
n1 = n3;
n3 = aux;
}
if (n2 > n3)
{
aux = n2;
n2 = n3;
n3 = aux;
}
printf ("\n\tLos números ordenados de menor a mayor:\n");
printf ("\n%i, %i, %i", n1, n2, n3);
getch();
}
```

En este programa podéis comprobar que algo que nos es muy sencillo a nosotros, a la hora de explicárselo al ordenador, puede convertirse en una tarea bastante complicada.

Es **muy recomendable** hacerse un **pequeño esquema** antes de empezar a programar para facilitar la tarea. Estos esquemas se hacen en lo que se llama pseudocódigo. El pseudocódigo es un programa escrito de manera formal (igual que cuando programamos), pero no es necesario que esté desarrollado por completo y no es necesario hacerlo en ningún lenguaje de programación en concreto.

Bucles

En todos los programas que hemos visto cada instrucción podía ejecutarse una sola vez o como vimos en los casos condicionales podía no ejecutarse. Muchas veces nos interesa que una instrucción se ejecute varias veces, para conseguir esto utilizaremos los bucles.

El bucle while

While quiere decir mientras en Inglés, así que este bucle se ejecutará mientras se cumpla una condición. Su sintaxis es la siguiente:

```
while (condicion)
{
instrucciones;
}
```

Veremos un ejemplo de un contador:

```
#include <stdio.h>

void main()
{
    int contador = 0;
    while ( contador<100 )
    {
        contador++;
        printf( "El contador vale %i.\n", contador );
    }
}
```

Este bucle se ejecutará 100 veces. Los valores que imprimirá por pantalla serán desde el 1 hasta el 100. Es importante darnos cuenta que los valores impresos por pantalla no serán desde 0 hasta 99 porque incrementamos el contador antes de imprimirlo. Así inicialmente vale 0 pero al entrar en el bucle lo incrementamos. Este suele ser un error típico que además es difícil de detectar.

Algo que también es importante tener en cuenta es que la condición que analizamos en el `while` debe cambiar dentro del bucle, en caso contrario esta condición se cumpliría siempre y el programa nunca acabaría, es lo que llamamos un **bucle infinito**.

Bucle do while

Este bucle es muy parecido al anterior, solamente tiene una diferencia, y es que como mínimo se ejecuta una vez. El bucle anterior no llegaba a ejecutarse si la condición no se cumplía al principio, en cambio este bucle se ejecutará una vez y se comprobará la condición una vez que ya se haya ejecutado una vez. Su sintaxis es la siguiente:

```
do
{
instrucciones;
} while (condicion);
```

Bucle for

Aunque con el bucle while() podríamos obtener el bucle for(), hay veces en las que es más aconsejable utilizar este para una mayor claridad en el programa. Su sintaxis es esta:

```
for (inicializacion; condicion; incremento)
{
    sentencias;
}
```

Para entenderlo mejor podemos decir: desde xxx; mientras xxx; hacer xxx.

El siguiente ejemplo imprimirá 100 veces Hola con el número de veces que lo ha escrito delante:

```
#include <stdio.h>
void main()
{
    int i;
    for ( i=1 ; i<=100 ; i++ )
    {
        printf( "%iHola\n", i);
    }
}
```

Debemos tener muy en cuenta la condición que debemos poner para ejecutar el bucle el número de veces que queremos. En este caso hemos puesto desde $i = 1$ mientras $i \leq 100$, lo que implica 100 veces, ya que incrementamos de 1 en 1. Si hubiéramos puesto `for (i = 0; i <= 100; i++)` el bucle se habría ejecutado 101 veces.

Si queremos saltarnos alguna de las partes del for (inicializacion; condicion; incremento), podremos dejarlo vacío, pero siempre habrá que poner “;”.

Algo a tener en cuenta también es que en la misma línea del `for` no ponemos “;”, si lo pusiéramos el bucle se ejecutaría una sola vez. Esto mismo sucede en los bucles anteriores y en los condicionales.

Por último recordar que si dentro del bucle sólo ponemos una instrucción, no será necesario el uso de llaves.

break

Esta sentencia tiene 2 utilidades, finalizar un case de la sentencia switch, o forzar el final de un bucle.

```
#include <stdio.h>

void main ()
{
    int x;

    for (x = 0;; x++)
    {
        printf ("%i", x);
    }
}
```

```
        if (x == 10) break;
    }
    puts ("Se acabó");
}
```

En cada iteración se comprueba si el valor de x es 10, si lo es el bucle finalizará.

exit()

La función `exit()` finaliza el programa entero. Se le puede pasar el argumento 0 (`exit(0)`) para indicar que el programa ha finalizado correctamente.

continue

Esta sentencia es parecida al `break`, pero en este caso no se fuerza la terminación del bucle, si no que se provoca una nueva iteración del bucle saltando las instrucciones que quedaban hasta el final de la iteración actual.

Ejercicios:

- 1.- Crear un programa que escriba los números del 1 al 1000.
- 2.- Haz un programa que sume los 1000 primeros números naturales.
- 3.- Crear un juego que consista en averiguar un número. El programa nos indicará cada vez si el número introducido es mayor o menor que la constante almacenada que tendremos que averiguar.
- 4.- Mejorar el programa anterior de forma que el número que debemos averiguar sea aleatorio. Para conseguir números aleatorios utilizaremos 2 funciones que están en la librería `<stdlib.h>`. Una de estas funciones habrá que introducirla al principio el programa: `randomize()`; Servirá para que los números aleatorios sean diferentes cada vez, cambia la raíz. La otra función nos devolverá el número:
`i = random(100)`.
Como argumento le pasamos el número de aleatorios diferentes que nos devolverá, en el ejemplo desde 0 hasta 99.

- 5.- Crear un programa utilizando bucles que nos dibuje esto:

```
*
**
***
****
*****
*****
*****
*****
*****
*****
*****
*****
```

- 6.- De forma parecida ahora el programa dibujará:

```
*****
*****
```


Tema 6: Arrays (Matrices)

Introducción:

Un array es un conjunto ordenado de variables del mismo tipo que se referencian con un mismo nombre y la posición que ocupan en dicho conjunto. También se llaman vectores (si tienen una dimensión), tablas.

Supongamos que queremos llevar un control de la temperatura que tenemos en cada hora del día. Podríamos declarar 24 variables, una para cada hora del día, pero esto no es nada cómodo, por ello tenemos los arrays; gracias a ellos podremos declarar de forma rápida y cómoda un número elevado de variables.

La forma general de declarar un array unidimensional es:

```
tipo nombre [tamaño];
```

Para acceder a cada uno de los elementos del vector lo haremos indicando su posición. Lo veremos con el ejemplo anterior:

```
int temp [24];
```

Hemos declarado 24 variables de tipo entero.

Las posiciones que ocupan van de la 0 a la 23, y accederemos a cada una así: temp [0], temp [1] ... temp[23].

Si ahora quisiéramos inicializarlas todas con el valor 5 necesitaríamos un bucle. Vemos el ejemplo:

```
for (i = 0; i < 24; i++)  
{  
temp [i] = 5;  
}
```

Es muy importante que el programador tenga en cuenta los límites de la tabla para no apuntar fuera de esta. Si en el ejemplo anterior intentásemos acceder a la posición 24 ó 25 no aparecería ningún error al compilar el programa, lo que sucederá es que apuntaremos a direcciones de memoria de las que desconocemos su valor y esto podría ocasionar errores.

Arrays de caracteres:

También se les llama cadena o strings. Tienen la peculiaridad de terminar con el carácter nulo ‘\0’, por esto es necesario crear la cadena con un carácter más de los que necesitamos. Este valor se introducirá solo, por lo que sólo deberemos preocuparnos de añadir un elemento extra a la cadena.

Cabe destacar que como se indicó anteriormente, podremos salirnos del array si no tenemos en cuenta su tamaño, por esto mismo si quisiéramos introducir la cadena “Hola” en un array de 4 posiciones (recordemos que necesitaríamos una más), seguramente el programa funcionaría igual, aunque no es recomendable hacerlo.

Arrays de varias dimensiones:

De momento hemos visto arrays de una sola dimensión (vectores). Podemos declarar arrays de 2 dimensiones, de 3 o incluso de más.

Un array de 2 dimensiones podría servirnos para crear un juego de tablero como un ajedrez o el 3 en raya. Además los arrays de 2 dimensiones tienen una especial importancia ya que hemos de recordar que el monitor es una matriz de 2 dimensiones.

La forma de declararlos es bien simple:

```
tipo    nombre    [tamaño_dimension_1][tamaño_dimension_1]...[tamaño_dimension_n]
```

Si queremos una matriz de 2 dimensiones haremos esto:

```
int matriz [3][4];
```

Si necesitamos un array de 2 dimensiones de caracteres de 3X3 haremos:

```
char matriz [3][4]; //El 4 es debido al carácter nulo.
```

Obtendremos una tabla de este tipo:

	0	1	2	3
0				
1				
2				

Inicialización de arrays:

Es posible inicializar los arrays en el momento de definirlos, lo veremos con un par de ejemplos:

```
int matriz [3][2] = {1,2,  
                    4,0  
                    2,19}
```

(Se podría haber inicializado en la misma línea).

```
char cadena[25] = "Estoy en STARNET";
```

También tenemos otro método que además de servir para inicializar nos ahorraremos indicar el tamaño del array, lo vemos en el ejemplo:

```
char cadena [] = "Buenas tardes";
```

En este ejemplo el tamaño del array será el mismo que la cadena introducida más uno (por que al final tenemos el carácter nulo '\0').

Funciones de cadenas:

Uno de los usos más habituales de los arrays son las cadenas de caracteres, los strings.

gets () ;

Esta función lee por teclado una cadena de caracteres. Entre paréntesis debemos indicar la variable que recogerá ese valor.

puts () ;

Aunque ya hemos utilizado esta función, añadiremos que se puede utilizar con strings.

Las funciones siguientes se encuentran en la librería **<string.h>**

strcpy () ;

Su sintaxis es:

```
strcpy (variable, "cadena_a_copiar");
```

Así introduciremos un valor a esta cadena. Sería **incorrecto** el siguiente ejemplo:

```
char cadena [20];  
cadena = "Hola mundo";
```

strlen () ;

Nos devuelve un entero que indica la longitud real de una cadena (al declararla indicamos su longitud máxima).

strcmp () ;

Esta función nos devolverá un número. Su sintaxis es esta:

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

Según el número que devuelva tendremos:

< 0 → La primera cadena es menor.

= 0 → Las dos cadenas son iguales-

> 0 → La primera cadena es mayor.

strcat () ;

Si necesitamos unir dos strings utilizaremos esta función.

```
strcat (cadena1, cadena2);
```

Se añadirá la cadena2 al final de la cadena1.

strlwr () ;

Esta función devuelve la cadena que le pasemos como parámetro en minúsculas.

strupr () ;

Esta otra función la devolverá en mayúsculas.

Ejemplo:

Vamos a ver un ejemplo que nos pedirá 10 números y después los imprimirá:

```
#include <stdio.h>  
  
void main (void)  
{  
int vector[10];
```

```
int i;

for (i = 0; i <= 9; i++) /*Pido número 10 veces*/
{
    printf ("\nNúmero para la posición %i: ", i);
    scanf ("%i", &vector [i]);
}
printf ("\n\nOK, Números introducidos\n\n");
//Los números ya están introducidos

for (i = 0; i < 10; i++)
{
    printf ("Posición %i: %i\n", i, vector[i]);
}
//Números impresos
}
```

Ejercicios:

1. Programa que nos pida 10 números y a continuación nos pregunte por uno, deberemos decir si está en la matriz y en que posición.
2. Programa que nos rellene de forma aleatoria un vector de 10 posiciones con números del 0 al 20, después tendremos 3 intentos para adivinar un número, cuando adivinemos un número nos indicará las posiciones en las que se encuentra.
3. Crear un programa que nos pase una frase a mayúsculas sin utilizar la función **strupr()**.
4. Programa que elimine los espacios en blanco de una frase.
5. Crear un programa que nos indique los números que hemos acertado en una primitiva. Primero nos preguntará nuestros 6 números de nuestra combinación y después la combinación ganadora.
6. Realiza un programa que rellene aleatoriamente una matriz de 3 filas y 4 columnas y la muestre por pantalla (números entre 0 y 20).
7. Ampliar el programa anterior para que se nos pregunte por un número y nos diga la fila y la columna en la que se encuentra.
8. Programa que calcule los n primeros números primos.
9. * Hacer el juego del 3 en raya para 2 jugadores.
10. *Crear el juego de la ruleta. La ruleta consta de 37 números, del 0 al 36. En nuestro programa simplificaremos algo las reglas. Nosotros podremos apostar por número par/impar (en caso de acierto ganaremos el doble de nuestra apuesta), a una de las 3 docenas (en caso de acertar ganaremos el triple de nuestra apuesta) y a un número en concreto, en caso de acertar multiplicaremos nuestra apuesta por 36. Si la bola cae en el número 0, la banca se lleva todas las apuestas, además nadie puede apostar en este número. Tendremos que saber en cada momento el dinero que nos queda.

(*Nota: Estos programas resultarán más sencillos después de ver el Tema 7)

Tema 7: Funciones

Introducción:

Una función es una porción del programa con identidad propia que separamos del programa principal para poder utilizarla todas las veces que lo necesitemos. Cada vez que necesitemos realizar este proceso simplemente llamaremos a la función. Esto, lejos de complicar el programa lo hará mucho más claro, y nos permitirá reutilizar acciones que realicemos muy a menudo.

Definición de la función

Una función se define antes de la función principal utilizando esta sintaxis:

```
tipo_a_devolver nombre_funcion ([argumentos]);
```

El nombre de la función no puede contener espacios, acentos, ni símbolos “raros”.

El tipo a devolver: cuando la función se ejecuta, termina y devuelve un valor, este valor puede ser de cualquiera de los tipos que hemos visto: int, float, char, etc... Si no devuelve nada (por ejemplo una pantalla de bienvenida) pondremos void. En caso de dejar el tipo a devolver vacío se sobreentenderá que devolvemos un entero tipo int.

Argumentos: es una lista de variables separadas por comas que podemos utilizar dentro de la función o incluso modificarlos como veremos más adelante. En caso de no necesitar ningún parámetro deberemos poner, igualmente, los paréntesis y dentro podremos poner void.

Llamada a la función

Podremos llamar a una función dentro del programa principal o dentro de otra función. La sintaxis es la siguiente:

```
[variable =] nombre_funcion ([argumentos]);
```

Cuerpo de la función

El cuerpo de la función es la parte es la que definimos lo que debe hacer, su sintaxis es:

```
tipo_a_devolver nombre_función ([argumentos])  
{  
    variables_locales;  
    cuerpo_funcion;  
    [return valor;]  
}
```

Dentro de la función declararemos las variables que necesitemos, utilizaremos las que tenemos en la lista de argumentos pero no podremos utilizar las variables que hayamos declarado dentro de la función main ni las que estén declaradas en otras funciones.

Ejemplo 1:

Vamos a ver un ejemplo que ayudará a comprender todo esto mejor.

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

void bienvenida (void);
char menu (void);
float area_triangulo (float b, float h);

void main ()
{
    char opcion;
    float area, base, altura;
    bienvenida();
    opcion = menu();
    switch (opcion)
    {
        case '1': puts ("Has elegido la opción 1"); break;
        case '2': puts ("Has elegido la opción 2"); break;
        case '3': clrscr();
                    puts ("Base del triángulo: ");
                    scanf ("%f", &base);
                    puts ("Altura del triángulo: ");
                    scanf ("%f", &altura);
                    area = area_triangulo (base, altura);
                    printf ("El área del triángulo es %.3f\n", area);
                    break;
        default: puts ("Opción Incorrecta");
    }
    puts ("Pulsa una tecla para finalizar");
    getch();
}

void bienvenida()
{
    clrscr(); gotoxy(20,11);
    puts ("PRÁCTICA DE FUNCIONES EN STARNET");
    gotoxy (20,13);
    puts ("pulsa una tecla para continuar"); getch();
}

char menu ()
{
    char op;
    clrscr(); puts ("MENÚ");
    puts ("1.-Opción 1");
    puts ("2.-Opción 2");
    puts ("3.-Área de un triángulo");
    puts ("OPCION: ");
    op = getch ();
    return op;
}

float area_triangulo (float b, float h)
{
    float area;
    area = (b * h) / 2;
    return area;
}
```

La sentencia return

Se utiliza para:

- Devolver el control al programa que llamó la función.
- Devolver un valor.
- Forzar la finalización de la función (igual que `break` con los bucles).

Llamada por valor y llamada por referencia

Hay dos formas de pasar argumentos a las funciones, por valor y por referencia.

Por valor: Este método crea variables locales con los valores de los argumentos que le hemos pasado y no se modificarán las variables de la función principal.

Por referencia: Con este método podremos modificar las variables del programa principal o la función que llama a esta otra función; esto lo lograremos pasando la dirección de memoria de la variable, y para hacerlo pasaremos la variable así: `&variable`.

Lo veremos con unos ejemplos:

Por valor:

```
#include <stdio.h>

void incrementar(int i);

void main ()
{
    int i;
    i = 0;
    printf ("\n\nFunción main. Valor inicial: %i\n", i);
    incrementar (i);
    printf ("\nFunción main. Valor final: %i", i);
}

void incrementar (int i)
{
    printf ("\nDentro de la función incrementar:\nAntes de incrementar:
%i", i);
    i++;
    printf ("\nDespués de incrementar: %i\n", i);
}
```

Por referencia:

```
#include <stdio.h>

void incrementar (int *a);

void main ()
{
int i;
i = 0;
printf ("\n\nFunción main. Valor inicial: %i\n", i);
incrementar (&i);
printf ("\nFunción main. Valor final: %i", i);
}

void incrementar (int *a)
{
printf ("\nFunción incrementar:\nAntes de incrementar: %i", *a);
(*a)++;
printf ("\nDespués de incrementar: %i\n", *a);
}
```

En el paso por referencia utilizamos & para referirnos a una dirección de memoria, y el * para referirnos a su contenido, para comprender esto mejor es necesario conocer punteros.

Paso de cadenas a funciones:

En el caso de pasar una cadena como parámetro, tendremos una pequeña variación, a la hora de llamar la función no habrá que poner &, ya que el nombre de una cadena sin su índice nos devuelve la dirección de memoria de la variable de la cadena.

Diferentes métodos de pasar un array por referencia a una función.

Existen tres formas de declarar un parámetro que va a recibir un puntero a un array. Veamos con un ejemplo las tres formas.

```
#include <stdio.h>

void funcion_ejemplo_1 (int a[10]);
void funcion_ejemplo_2 (int a[]);
void funcion_ejemplo_3 (int *a);

void main (void)
{
int array [10];
register int i;
for (i = 0; i < 10; i++)
array[i] = i;
funcion_ejemplo_1 (array);
funcion_ejemplo_2 (array);
funcion_ejemplo_3 (array);
}

void funcion_ejemplo_1 (int a[10])
{
register int i;
for (i = 0; i < 10; i++)
printf ("%d ", a[i]);
}
```

```
}  
  
void funcion_ejemplo_2 (int a[])  
{  
    register int i;  
    for (i = 0; i < 10; i++)  
        printf ("%d ", a[i]);  
}  
  
void funcion_ejemplo_3 (int *a)  
{  
    register int i;  
    for (i = 0; i < 10; i++)  
        printf ("%d ", a[i]);  
}
```

En la función `funcion_ejemplo_1()`, el parámetro `a` se declara como un array de enteros de diez elementos, el compilador de C automáticamente lo convierte a un puntero a entero. Esto es necesario porque ningún parámetro puede recibir un array de enteros; de esta manera sólo se pasa un puntero a un array. Así, debe haber en las funciones un parámetro de tipo puntero para recibirlo.

En la función `funcion_ejemplo_2()`, el parámetro `a` se declara como un array de enteros de tamaño desconocido. Ya que el C no comprueba los límites de los arrays, el tamaño real del array es irrelevante al parámetro (pero no al programa, por supuesto). Además, este método de declaración define `a` como un puntero a entero.

En la función `funcion_ejemplo_3()`, el parámetro `a` se declara como un puntero a entero. Esta es la forma más común en los programas escritos profesionalmente en C. Esto se permite porque cualquier puntero se puede indexar usando `[]` como si fuese un array. (En realidad, los arrays y los punteros están muy relacionados).

Los tres métodos de declarar un parámetro de tipo array llevan al mismo resultado: un puntero.

Variables globales:

Una forma que simplificar el paso de argumentos a una función es la utilización de variables globales. Estas variables las declararemos antes de la función principal y se podrán modificar en cualquier parte del programa, sin embargo hay que evitar su utilización porque estas variables ocupan memoria durante todo el programa; lo más correcto es pasar las variables que debamos modificar por referencia a las funciones.

Creación de bibliotecas:

Una vez creadas una serie de funciones que vayamos a utilizar a menudo podemos optar por guardarlas en nuestra propia biblioteca. Lo único que deberemos hacer es crear el archivo con los includes necesarios para que funcione correctamente y guardar el fichero como “`nombre.h`” en vez de “`nombre.cpp`”. Este fichero lo guardaremos en la carpeta `INCLUDE` dentro de `TC` y ya lo podremos utilizar desde cualquier programa, lo único que deberemos hacer es incluir la librería en la parte de los includes y llamar a la función pasándole los argumentos necesarios (estos argumentos deberán ser del mismo tipo que declaramos en las funciones).

Ejercicios:

1-Se trata de coger los programas que hicimos en el tema 4 y 5 sobre áreas, volúmenes y perímetros de algunas figuras geométricas y hacerlos utilizando funciones. Notaremos que el programa principal queda mucho más claro y más simplificado.

Para hacer este programa necesitaremos funciones que nos hagan los cálculos, algo así como:

```
float area_cilindro (float radio, float altura); /*En esta función pasamos unos parámetros y hacemos cálculos con ellos*/
```

También tendremos funciones que no devolverán nada, otras a las que no habrá que pasarles nada... Nos ayudará mucho guiarnos con el ejemplo de esta lección.

2-Podemos hacer una función que nos intercambie 2 variables, para ello habrá que usar la llamada por referencia, ya que no devolvemos 1 valor, en todo caso, habría que devolver 2 valores.

3-Crear un array de 1 dimensión, rellenarlo de forma aleatoria, pedir un número al usuario y buscarlo en el array indicando la posición. Habrá que utilizar funciones.

4-Repetir el ejercicio anterior con un array de 2 dimensiones y con otro de 3.

1-Juego del 3 en raya para 2 jugadores.

Ideas:

Podremos dibujar el tablero con el código ASCII.

También podemos pedir el nombre de los 2 jugadores.

Debemos tener en cuenta que la casilla en la que el jugador quiere tirar esté vacía.

2-Juego de la ruleta.

Simplificaremos la ruleta que podemos encontrar en los casinos. La ruleta consta de 36 números.

Al iniciar el programa podremos elegir si queremos apostar:

1. Par/impar: en caso de acertar multiplicaremos lo apostado X2
2. Por docenas: multiplicaremos X3 si acertamos
3. A un número concreto: X36 si acertamos.

El programa nos preguntará el dinero que queremos apostar, si sale 0 la banca se lo llevará todo. Habrá que tener un control del dinero y si no tenemos dinero no podremos seguir apostando.

3-Codificador de código Morse.

Buscaremos el código Morse por Internet. Podéis escribir los puntos y rayas, pero también podéis hacer que se escuchen, debéis tener en cuenta que no hay una duración determinada para los pitidos, aunque hay que seguir unas normas:

- ♣ La raya tiene la duración de 3 pitidos.
- ♣ Los signos de una misma letra se separan con la duración del punto.
- ♣ El tiempo que hay que esperar entre letras es el de una raya.
- ♣ El tiempo que esperaremos entre palabras es de 5 puntos.

Una vez que todo funcione podéis hacer que el jugador pueda cambiar la velocidad.

Notas generales:

- Es muy recomendable hacerse un esquema con papel y boli antes de comenzar a programar.
- Nuestro programa ganará en claridad si usamos funciones.
- Nos facilitará el trabajo hacer partes del programa que compilen y después unirlos. No hace falta programarlo todo de golpe, podemos ir mejorando y refinando nuestro programa poco a poco.

4-Ordenación de vectores.

Aunque pueda resultar curioso, el tema de ordenación es un tema muy extenso y del que se ha estudiado mucho. En la segunda parte del curso dedicaremos un tema a la ordenación de números.

Lo que se pide en este programa es crear un vector de números de forma aleatoria para después ordenarlo.

- Es imprescindible hacerse un esquema antes de programar y buscar un método para hacerlo. (Existen muchísimos métodos diferentes).
- No hace falta que el método sea muy laborioso, pero sí es necesario tener bien claro qué hará el algoritmo antes de empezar a programarlo.

Un posible esquema para el 3 en raya sería:

```
Mientras ((jugadas < 9) Y (ganado(tablero)=' ')) hacer
    Dibujar tablero(tablero);
    Pedir_posición(x,y);
    Mientras posición_correcta (tablero, x, y) = FALSO volver a pedir posición;
FMientras;
FMientras;

ganado(tablero)
    si alguna columna es igual
        si columna = 'X' devuelve 'X'
        sino devuelve 'O'
    fin si;
    sino si alguna fila es igual
...
    sino devuelve = ' ';
fin ganado;

posición_correcta (tablero, x, y)
    si posición = ' ' devuelve CIERTO
    sino devuelve FALSO
fin posición_correcta;
```

Este ejemplo no está desarrollado completamente pero hacer algo parecido **antes** de empezar a programar nos ayudará mucho.

El esquema anterior es lo que se conoce como **pseudocódigo**. No es ningún lenguaje de programación ni su traducción a C es inmediata pero nos ayudará a dividir nuestro programas en partes y tendremos algo con lo que comenzar a programar.

Si no os da tiempo de hacer todos los programas anteriores podéis practicar haciendo su pseudocódigo.

Tema 8: Tipos de datos

En este capítulo aprenderemos a crear nuestros propios tipos de datos.

Estructuras de datos

Una estructura es un conjunto de variables de diferentes tipos que se referencian por el mismo nombre. Supongamos que queremos crear una agenda con números de teléfono, necesitaríamos un array de cadenas para los nombres, otro para los apellidos, otro para los números de teléfono. El programa sería complicado de seguir, aquí es donde intervienen las estructuras.

A cada variable de la estructura le llamaremos campo, y cada uno de ellos está relacionado con los otros. La sintaxis de una estructura es la siguiente:

```
struct nombre_estructura
{
    tipo1 nombre1;
    tipo2 nombre2;
    ...
};
```

Lo que hemos hecho con las líneas anteriores es definir la estructura, definir un nuevo tipo, pero todavía tenemos que definir la estructura, pero todavía nos queda definir una variable de este tipo:

```
struct nombre_estructura nombre_variable;
```

Ahora ya declaramos una variable del tipo estructura. Estas dos sentencias las podemos reducir a una sola de esta forma:

```
struct [nombre_estructura]
{
    tipo1 nombre1;
    tipo2 nombre2;
    ...
} nombre_variable;
```

De esta forma no es necesario poner el nombre de la estructura.

Referencia a los elementos de la estructura

Los elementos individuales de la estructura se referencian utilizando el operador punto (.), siguiendo la sintaxis siguiente:

```
variable.campo
```

Ejemplo

Supongamos que una estructura queremos acceder al campo edad de pepe y le queremos dar el valor 20, haremos:

```
pepe.edad = 20;
```

Para imprimir este valor haremos:

```
printf ("%i", pepe.edad);
```

Y si quisiéramos coger por teclado el valor deberíamos teclear:

```
scanf ("%i", &juan.edad);
```

Para nuestro ejemplo podemos crear una estructura en la que almacenaremos los datos de cada persona. Vamos a crear una declaración de estructura llamada *amigo*:

```
struct estructura_amigo {
    char nombre[30];
    char apellido[40];
    char telefono[10];
    short int edad;
};
```

Ahora ya tenemos definida la estructura, pero aun no podemos usarla. Necesitamos declarar una variable con esa estructura.

```
struct estructura_amigo amigo;
```

Ahora la variable *amigo* es de tipo *estructura_amigo*. Para acceder al nombre de *amigo* usamos: **amigo.nombre**. Vamos a ver un ejemplo de aplicación de esta estructura. (NOTA: En el siguiente ejemplo los datos no se guardan en disco así que cuando acaba la ejecución del programa se pierden).

```
#include <stdio.h>

struct estructura_amigo { /* Definimos la estructura
estructura_amigo */
    char nombre[30];
    char apellido[40];
    char telefono[10];
    unsigned short int edad;
};

struct estructura_amigo amigo;

int main()
{
    printf( "Escribe el nombre del amigo: " );
    fflush( stdout );
    scanf( "%s", &amigo.nombre );
    printf( "Escribe el apellido del amigo: " );
    fflush( stdout );
    scanf( "%s", &amigo.apellido );
    printf( "Escribe el número de teléfono del amigo: " );
    fflush( stdout );
    scanf( "%s", &amigo.telefono );
    printf("Edad: ");
    fflush(stdout);
    scanf ("%ui", &amigo.edad);
    printf( "El amigo %s %s tiene el número: %s.\nEdad: %ui",
    amigo.nombre, amigo.apellido, amigo.telefono, edad );
}
```

Enumeraciones

Ya hemos visto que podíamos definir constantes con `#define`, sin embargo habrá veces que será más cómodo utilizar enumeraciones que `#define`. Las enumeraciones son un conjunto de variables **enteras** con nombre, especificando los valores que puede tomar. Su sintaxis es parecida a la de un `struct`.

```
enum nombre {lista} variables;
```

Ejemplos:

```
enum boolean {false, true} encontrado;
```

El valor de `false` será 0 y el de `true` 1. Este tipo, los booleanos, están definidos en otros muchos lenguajes de programación, en C deberemos definirlo nosotros mismos.

```
enum tipo_monedas {pesetas = 1, duro = 5, diez_pesetas = 10, cinco_duros = 25, otro} moneda;
```

En este caso hemos indicado el valor que queremos que tomen cada uno de los elementos de la lista, excepto en el último caso. Este último caso cogerá el valor siguiente, en este caso 26.

Este Ejemplo nos pedirá el nombre de 5 amigos y después nos buscará los datos de uno de ellos.

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

struct persona
{
    char nombre [20];
    char apellidos [40];
    char telefono [10];
    short int edad;
};

void main ()
{
    short int i;
    persona amigos[5];
    char buscar [20];
    clrscr();
    for (i = 0; i < 5; i++)
    {
        printf ("\n\nAMIGO %i.\n\nNombre: ", i);
        gets (amigos[i].nombre);
        fflush (stdin);
        printf ("\nApellidos: ");
```

```
        fflush (stdin);
        gets (amigos[i].apellidos);
        fflush (stdin);
        printf ("\nTeléfono: ");
        fflush (stdin);
        gets (amigos[i].telefono);
        printf ("\nEdad: ");
        scanf ("%i", &amigos[i].edad);
        fflush (stdin);
    }

    printf ("\n\n\t Datos Introducidos\nAmigo a Buscar:
");
    gets (buscar);

    for (i = 0; i < 5; i++)
    {
        if (strcmp (amigos[i].nombre, buscar) == 0)
        {
            printf ("\n\nNombre: ");
            puts (amigos[i].nombre);
            printf ("\nApellidos: ");
            puts (amigos [i].apellidos);
            printf ("\nTel, fono: ");
            puts (amigos[i].telefono);
            printf ("\nEdad: %i", amigos[i].edad);

        }

    }

}
```

Tema: 9 Clasificación en memoria primaria.

La clasificación por inserción se basa en insertar un elemento dado en el lugar que corresponde de una parte ya ordenada del array.

Inserción Directa: Este método es bastante intuitivo, en resumen, lo que hace es:

Toma un elemento en la posición i .

Busca su lugar en las posiciones anteriores.

Mueve hacia la derecha los restantes.

Lo inserta.

Inserción binaria: Es una mejora del anterior, básicamente cambia en un punto, a la hora de buscar la posición en la que debe ir el elemento lo hace de forma dicotómica.

Selección directa:

También es un método bastante intuitivo, y de hecho es el que a más alumnos se les ocurre antes de haber visto ningún método de ordenación.

Se trata de:

Buscar el menor elemento de la parte del vector que no está ordenada.

Colocarlo en la primera posición de la parte no ordenada de la tabla.

Clasificación por partición (quicksort)

Este método fue desarrollado por C.A.R. Hoare y es el método más rápido que se conoce hasta la fecha.

Se basa en que cuanto mayor sea la distancia entre los elementos que se intercambian más eficaz será la clasificación. Para ello se elige un pivote (está demostrado que la eficacia del algoritmo es mayor si esta llave es el elemento central). Se trata de buscar un elemento mayor que el pivote en la parte izquierda, y otro menor que el pivote en la parte derecha e intercambiarlos. Este proceso se repite hasta que se cruzan el índice de incremento y el de decremento, en este momento nos queda dividido el vector en 2 partes, la parte izquierda con los elementos menores que el pivote y la parte derecha con los mayores que el pivote. A cada una de estas partes se les llama partición. Esta operación se repite con cada partición hasta que no haya particiones a ordenar.

Este último es un método no muy intuitivo pero podréis comprobar su eficacia con el ejemplo propuesto en clase en el que podréis comprobar empíricamente las diferencias de tiempo a la hora de ordenar un vector del mismo tamaño con estos 3 métodos.

Existen otros muchos métodos de ordenación en memoria primaria y también existen otros métodos para la clasificación en memoria secundaria.

La clasificación en memoria secundaria no está incluida en el temario del curso pero me permito hacer una breve explicación para que se comprendan las limitaciones de los métodos expuestos anteriormente.

El acceso a un elemento en memoria primaria tiene el mismo coste siempre, esté donde esté el elemento. O sea, si estamos en el primer elemento tardamos lo mismo en acceder al segundo que al último. Este tipo de ordenación se hace sobre la memoria RAM: Random Acces Memory (Memoria de Acceso Aleatorio).

En cambio, en la memoria secundaria es secuencial (igual que una cinta de video). Para ir a una posición, antes tenemos que recorrer todas las anteriores; por lo tanto, los métodos de ordenación no pueden ser los mismos, ya que no interesa que los números

hagan saltos grandes continuamente. Para poder ordenar cintas necesitaríamos de cintas auxiliares.

Para *entender la diferencia entre memoria secuencial y de acceso aleatorio podemos comparar en escuchar música con una cinta y con un CD. En la cinta, si queremos escuchar la última canción tendremos que correr la cinta hacia adelante pasando por todas las anteriores, en cambio en un CD de música eso no es necesario.

(*Nota: no digo que un CD sea de acceso aleatorio, simplemente pongo este ejemplo para que comprendamos la diferencia. Los CD's son de acceso directo (una combinación entre acceso aleatorio y acceso secuencial)).

Tema 10: Punteros

Ha llegado el momento de enfrentarnos a los punteros. Los punteros nos permiten acceder a cualquier parte de la memoria del ordenador (RAM), esto puede provocar un bloqueo del sistema en caso de no utilizarlos correctamente, por lo que será necesario saber en todo momento a donde apuntan. Con los punteros podemos acceder a memoria que están utilizando otros programas, periféricos, etc...

Variables

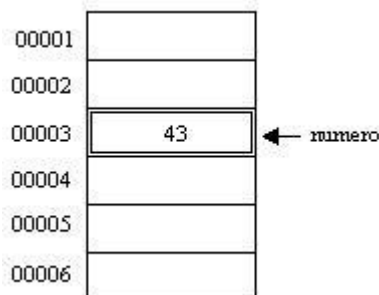
Hemos utilizado desde el principio variables, y sabemos el espacio que ocupa cada variable, lo que no sabemos es donde se guardan las variables; vale, se guardan en la RAM, ¿pero en qué parte de la RAM? Pues eso depende entre otros factores de los programas que se estén ejecutando y de la memoria libre.

Probaremos el programa siguiente y sabremos en qué lugar de la RAM está nuestra variable.

```
#include <stdio.h>
void main()
{
int i = 10;
printf ("\nDirección de i: %p\nValor de i: %i", &i,i);
}
```

Como ya sabemos, el valor de una variable puede ir cambiando a lo largo del programa, pero lo que no puede cambiar es su dirección de memoria, ésta se le asigna al ejecutarse el programa y no cambia.

Los punteros son unas variables que almacenan direcciones de memoria. Tenemos a continuación una representación de la memoria del ordenador:



Los números de la izquierda representan la dirección de memoria, y el número 43 representaría el contenido de la posición de memoria 00003. Tiene que quedar muy clara la diferencia entre el contenido de la memoria y su dirección.

Vamos a ir más allá y vamos a declarar una variable de tipo puntero.

```
#include <stdio.h>
void main ()
{
int i=43;
int *puntero;
puntero = &i;
printf ("\nLa variable i vale: %i\nSu dirección es %p\nEl valor del puntero es %i", i, puntero, *puntero);
*puntero = 10;
printf ("\nAhora i vale: %i",i);
}
```

```
}
```

Fijémonos bien como hemos declarado puntero. `int *puntero;` esto significa que declaramos un puntero que apunta a un entero.

Ahora fijémonos como hemos impreso el valor de el puntero, con `*puntero`. Esto ya nos suena del paso por referencia del tema de funciones. A la hora de cambiarle el valor a `i` lo hemos hecho también utilizando el `*`.

Ejercicios:

Ejercicio 1: Encuentra un fallo muy grave:

```
#include <stdio.h>

void main()
{
    int *i;
    *i = 43;
}
```

Ejercicio 2: Crear un programa que tenga una constante de tipo char (`const char letra = 'A';`), un puntero que apunte a esta, le intentáis cambiar el valor y me explicáis que ha pasado.

Ejercicio 3: Escribe un programa en el que declares una variable de tipo long, asignes la dirección de memoria de esa variable a un puntero, lo dividas entre 2 desde el puntero e imprimas el valor utilizando también el puntero.

Ejercicio 4: Prueba lo que sucede si declaramos un array y hacemos un puntero que apunte a la primera posición. Ves incrementando el puntero e imprimiendo el contenido del puntero y la dirección de memoria contenida en el puntero. Hazlo con una tabla de caracteres y con una tabla de enteros.

Hasta ahora cada vez que declarábamos una variable debíamos reservar un lugar de memoria al comenzar el programa. Como ya hemos visto en programas como el de Morse esto supone una limitación muy grande porque podíamos agotar la memoria reservada en un array por muy grande que este sea, y si creamos un array muy grande y no lo utilizamos todo, estamos desperdiciando memoria. Por esto es necesario poder reservar memoria cuando la necesitemos.

Malloc y free

Estas dos funciones nos servirán para asignar dinámicamente memoria. Estas 2 funciones se encuentran en la librería `<alloc.h>`

<code>malloc ();</code>

Reserva memoria y devuelve la posición de memoria del comienzo de ésta, por lo que deberemos guardar el resultado en

un puntero.

`puntero = (Tipo_variable *) malloc (bytes_reservar);`

Si no se ha podido reservar memoria, malloc devolverá el valor NULL, así que el puntero apuntará a NULL. Es importante asegurarnos de que se ha podido reservar la memoria, haremos algo así:

```
if(puntero = (int *) malloc (sizeof(char))) {puts  
("Correcto");}
```

free ();

Cuando ya no necesitemos el espacio que habíamos reservado, liberaremos la memoria, haremos: free(puntero);

Donde el puntero que pasamos como parámetro apunta al principio de la memoria que habíamos reservado.

Veremos un ejemplo:

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

void main()
{
unsigned long int bytes;
char *texto;
clrscr();

printf("Cuantos bytes quieres reservar: ");
scanf("%li",&bytes);
texto = (char *) malloc(bytes);
/* Comprobamos si ha tenido ,xito la operación */
if (texto)
{
printf("Memoria reservada: %li bytes = %li kbytes = %li Mbytes\n",
bytes, bytes/1024,bytes/(1048576));
printf("El bloque comienza en la dirección: %p\n", texto);
/* Ahora liberamos la memoria */
free( texto );
}
else printf("No se ha podido reservar memoria\n");
}
```

Si queremos reservar un número muy grande de bytes el programa nos dará error.

Tema 11: Listas enlazadas

Introducción

En el tema anterior vimos que para optimizar la memoria podíamos reservar memoria en el momento que la necesitásemos y no como hasta la fecha, que lo hacíamos al principio del programa.

Como ya hemos comentado, esto es una limitación muy grande, por ejemplo lo vimos en el ejercicio de la agenda (en la que debíamos saber a cuanta gente tendríamos como máximo), o en el ejercicio de calcular números primos. Si reservábamos mucha memoria para curarnos en salud y nunca agotar una tabla, estamos desperdiciando mucha memoria; si por el contrario reservamos poca memoria podemos agotar la memoria que tenemos reservada.

¿En qué consisten?

Las listas enlazadas pueden ser simples, dobles o circulares. De momento veremos las simples. Estas listas tendrán una estructura como la de la figura:



Para crear listas necesitaremos estructuras asignación dinámica de memoria. Vamos a ver como utilizar una lista en el ejemplo de la agenda:

```
struct _agenda {
    char nombre[20];
    char telefono[12];
    struct _agenda *siguiente;
};
```

Siguiente
Nombre
Telefono

De esta forma hemos creado una estructura formada por un nombre y un teléfono como datos principales y visibles para el usuario, lo que sucede esta vez es que no sé a cuanta gente voy a meter en mi agenda, por lo que utilizo un puntero que apunta a otro elemento con esta estructura. Cada vez que queramos meter a un nuevo elemento en la agenda deberemos reservar memoria para este elemento con el malloc.

Nos quedará algo así:

```
nuevo = (_agenda *) malloc (sizeof(_agenda));
```

Llenaremos los campos de ese nuevo elemento y lo meteremos en la lista. Para meter un elemento en la lista tendremos que hacer que el puntero del elemento anterior apunte a este nuevo elemento. Si nos paramos a pensar ya vemos que necesitamos un puntero que nos vaya indicando donde está el elemento actual, pero es que también necesitaremos un puntero que nos indique donde comienza la lista para poder recorrerla desde el principio ya que en cada elemento de agenda tenemos un puntero hacia el

siguiente elemento, pero no tenemos ninguno hacia el elemento anterior (si lo tuviéramos estaríamos hablando de una lista doblemente enlazada).

Ejemplo: Veremos un ejemplo que nos servirá de gran ayuda:

```
#include <stdio.h>
#include <conio.h>
#include <stdlib.h> //exit()
#include <alloc.h>

typedef struct _agenda {
    char nombre[20];
    char telefono[12];
    _agenda *siguiente;
};

void mostrar_menu();
void anadir_elemento();
void mostrar_lista();

_agenda *primero, *ultimo; /*De momento lo haremos con variables
globales, más adelante veremos como pasar punteros a funciones*/

void main() {
    char opcion;

    primero = (_agenda *) NULL;
    ultimo = (_agenda *) NULL;
    do {
        mostrar_menu();
        opcion = getch();
        switch ( opcion ) {
            case '1': anadir_elemento();
                break;
            case '2': printf("Ir pensando como hacer esto :D\n");
                break;
            case '3': mostrar_lista();
                break;
            case '4': exit( 1 );
            default: printf( "Opción no válida\n" );
                break;
        }
    } while (opcion!='4');
}

void mostrar_menu() {
    printf("\n\nMenú:\n=====\n\n");
    printf("1.- Añadir elementos\n");
    printf("2.- Borrar elementos\n");
    printf("3.- Mostrar lista\n");
    printf("4.- Salir\n\n");
    printf("Escoge una opción: ");fflush(stdin);
}

/* Con esta función añadimos un elemento al final de la lista */
void anadir_elemento() {
    _agenda *nuevo;
    /* reservamos memoria para el nuevo elemento */
```

```
nuevo = (_agenda *) malloc (sizeof(_agenda));
if (nuevo==NULL) printf( "No hay memoria disponible!\n");
else {
    printf("\nNuevo elemento:\n");
    printf("Nombre: "); fflush(stdin);
    gets(nuevo->nombre);
    printf("Teléfono: "); fflush(stdin);
    gets(nuevo->telefono);
/* el campo siguiente va a ser NULL por ser el último elemento
la lista */
    nuevo->siguiente = NULL;
/* ahora metemos el nuevo elemento en la lista. lo situamos
al final de la lista */
/* comprobamos si la lista está vacía. si primero==NULL es que no
hay ningún elemento en la lista. también vale ultimo==NULL */
    if (primero==NULL) {
        printf( "Primer elemento\n");
        primero = nuevo;
        ultimo = nuevo;
    }
    else {
        /* el que hasta ahora era el último tiene que apuntar al nuevo
*/
        ultimo->siguiente = nuevo;
        /* hacemos que el nuevo sea ahora el último */
        ultimo = nuevo;
    }
}

void mostrar_lista() {
_agenda *auxiliar; /* lo usamos para recorrer la lista */
int i;
i=0;
auxiliar = primero;
printf("\nMostrando la lista completa:\n");
while (auxiliar!=NULL) {
    printf( "Nombre: %s, Teléfono: %s\n", auxiliar->nombre,auxiliar-
>telefono);
    auxiliar = auxiliar->siguiente;
    i++;
}
if (i==0) printf( "\nLa lista está vacía!!\n" );
}
```

En el ejemplo vemos una función por terminar, la de eliminar un elemento. ¿Os atrevéis?

En este ejemplo también vemos algo nuevo, “->”, esta flecha la utilizamos en punteros cuando nos referimos a algún campo apuntado por el puntero. Por ejemplo,

```
ultimo->siguiente = nuevo;
```

En esta línea decimos que el puntero que siguiente (apuntado por ultimo) coge el valor de nuevo. Recordemos que último está apuntando a toda una estructura, con la flecha nos podemos referir a cada uno de los campos.

Ejercicios:

1-Crear la función eliminar un elemento de la lista enlazada.

2-Transformar nuestro programa para calcular números primos para almacenarlos con una lista enlazada y de esta forma poder seguir calculando números hasta que queramos.

Podéis utilizar la función `kbhit()` que nos devuelve cierto cuando pulsamos una tecla. Así podemos calcular números primos hasta que pulsemos una tecla.

Paso de punteros a funciones:

En este tema hemos utilizado variables globales, algo que como ya hemos comentado durante el curso debemos evitar. En este apartado veremos como pasar punteros a funciones por valor y por referencia para que no necesitemos recurrir a variables globales.

Paso por valor:

Observaremos el ejemplo

```
#include <stdio.h>
#include <conio.h>
void cambia (int *p);

void main()
{
int i = 10;
int *p;
p = &i;
clrscr();

printf ("\n1-MAIN\nDirección: %p\nValor: %i", p,*p);
cambia (p);
printf ("\n3-MAIN\nDespués de cambiar\nDirección: %p\nValor: %i",p,*p);
}

void cambia (int *p)
{
int j = 20;
p = &j;
printf ("\n2-FUNCION\nDirección: %p\nValor: %i", p,*p);
}
```

En este ejemplo vemos que el paso por valor lo hacemos igual que hasta ahora, hay que tener en cuenta que en la definición de la función ponemos `int *p` porque este es el tipo de variable.

Paso por referencia:

Veremos el ejemplo anterior modificado para que ahora sea paso por referencia:

```
#include <stdio.h>
#include <conio.h>
void cambia (int **p);

void main()
{
int i = 10;
int *p;
p = &i;
clrscr();

printf ("\n1-MAIN\nDirección: %p\nValor: %i", p,*p);
cambia (&p);
}
```

```
printf ("\n\n3-MAIN\nDespués de cambiar\nDirección: %p\nValor: %i",p,
*p);
}

void cambia (int **p)
{
int j = 20;
*p = &j;
printf ("\n\n2-FUNCION\nDirección: %p\nValor: %i", *p,**p);
}
```

Al igual que hacemos con el resto de variables, en la llamada añadimos &: cambia (&p). En los parámetros añadimos un * en la variable y en el cuerpo de la función añadimos un * cuando queremos cambiar su valor y al imprimirlo.

Liberar la memoria de una lista enlazada:

Ya hemos utilizado la función free para liberar la memoria que ocupa un puntero cuando ya no lo necesitamos, vamos a ver un ejemplo de cómo liberar la memoria de toda una lista enlazada:

```
void liberar()
{
primos *aux, *ant;
ant = primero;
for (aux = primero->siguiente; aux != NULL; aux = aux->siguiente)
{
free (ant);
ant = aux;
}
}
```

Vemos en el ejemplo que no basta sólo con una variable auxiliar, hemos utilizado aux y ant ya que si liberamos aux ya no podremos pasar al siguiente elemento. Es muy recomendable liberar la memoria cuando ya no la necesitamos.

Tema 12: Ficheros

Introducción

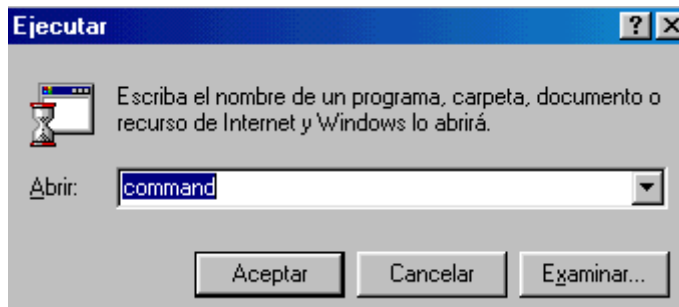
Durante el curso hemos ido superando poco a poco las barreras que nos íbamos encontrando, en el último tema, el de punteros conseguimos reservar memoria cuando la necesitásemos. Pero todavía tenemos una limitación muy grande en nuestros programas, y es que cada vez que salimos perdemos nuestra información, no podíamos guardar nuestros números primos en ningún fichero, ni podíamos guardar la configuración de nuestro programa, etc...

En esta lección superaremos al fin esta limitación.

Redireccionamiento

Antes de ponernos de lleno con los ficheros veremos unas curiosidades. Antes de nada, crearemos un sencillo programa que nos servirá para nuestras pruebas. El programa debe leer línea por línea lo que escribamos y volver a escribirlo hasta que escribamos fin. No hace falta que os dé el código fuente, ¿verdad?.

Dejaremos aparcado por unos momentos nuestro programa y abriremos una ventana MSDOS.



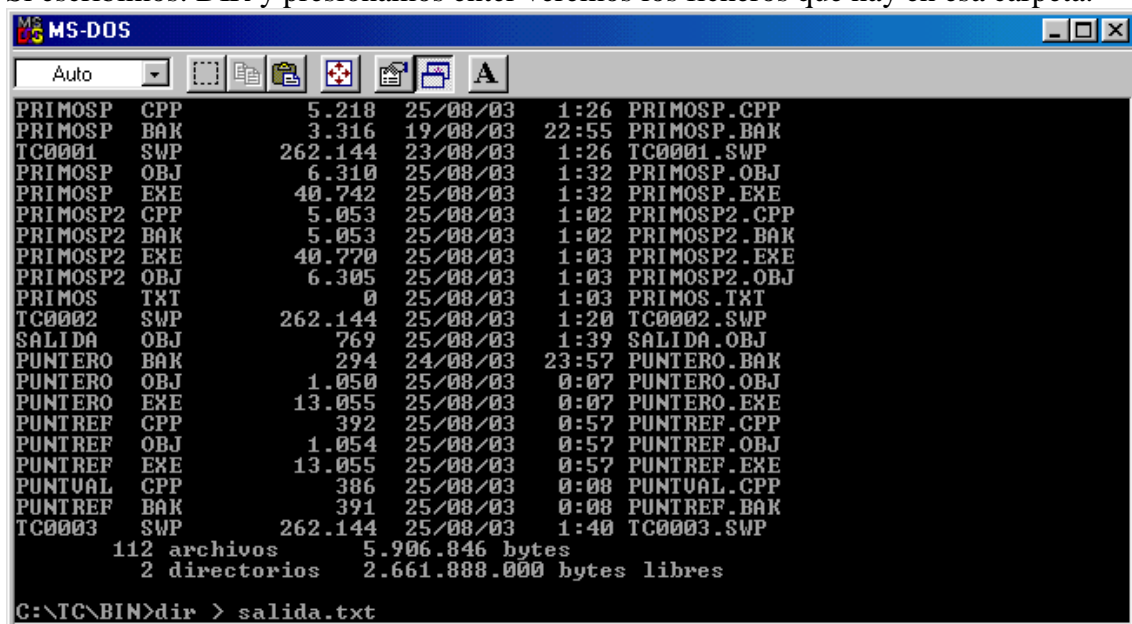
Inicio → Ejecutar → command

Ahora accederemos a la carpeta “C:\TC\BIN”

Para ir al directorio anterior utilizamos “CD..” y para entrar en un directorio usamos:

“CD TC”

Si escribimos: DIR y presionamos enter veremos los ficheros que hay en esa carpeta.



Si ahora escribimos “DIR > SALIDA.TXT” no se imprimirá el resultado por pantalla, se imprimirá en un fichero llamado “SALIDA.TXT”. Curioso, ¿Verdad? Para ver el resultado podremos abrir el nuevo fichero con cualquier editor de texto, por ejemplo el bloc de notas, el EDIT de MSDOS, o incluso hacer esto: “TYPE SALIDA.TXT”.

Pues ahora se trata de que hagáis lo mismo con vuestro programa. Para que lo podáis hacer es necesario que hayáis creado el fichero .EXE que como ya comentamos al principio del curso se crea cuando utilizamos la opción MAKE (en Turbo C es suficiente ejecutando el programa).

Una vez comprobados los resultados haremos lo mismo pero redireccionando la entrada. Ahora se trata de crear un fichero de texto con varias líneas y que termine con “fin”. Para redireccionar la entrada haremos: “NOMBRE < ENTRADA.TXT” Podemos redireccionar la salida y la entrada a la vez. La entrada con “<” y la salida con “>”. Si queremos continuar escribiendo en un fichero utilizamos “>>”.

freopen

```
freopen("fichero.txt", "modo", redirección);
```

Con esta instrucción podremos cambiar la salida estándar (la pantalla) desde el propio programa. Introduciremos esta línea en nuestro programa y más adelante veremos exactamente qué hace cada uno de los parámetros:

```
freopen( "resultado.txt", "wb", stdout );
```

Lectura de ficheros

Ya hemos visto el redireccionamiento de ficheros, pero esta forma no es muy cómoda para trabajar con ficheros. A continuación vamos a trabajar con ficheros utilizando funciones específicas para ello.

Hay que destacar que el C los ficheros no son solamente los archivos que guardamos en el ordenador, todos los dispositivos se tratan como ficheros: impresora, teclado, pantalla...

Para comenzar veremos y teclearemos el código siguiente. Es un programa que lee carácter por carácter todo un fichero y lo escribe por pantalla:

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

void main()
{
FILE *fichero;
char letra;
char nombre[40];

printf ("Dime el nombre del archivo que quieres imprimir:\n ");
gets (nombre);

fichero = fopen(nombre, "r");
```

```

if (fichero==NULL)
{
    printf( "No se puede abrir el fichero.\n" );
    exit( 1 );
}
printf( "Contenido del fichero:\n" );

letra=getc(fichero);
while (!feof(fichero))
{
    printf( "%c", letra );
    letra=getc(fichero);
}

if (fclose(fichero)!=0)
printf( "Problemas al cerrar el fichero\n" );
}

```

Cuando el programa nos pida el nombre del fichero le tendremos que indicar la ruta exacta en la que se encuentra (por ejemplo: C:\PRUEBAS\ENTRADA.TXT), en caso contrario nos buscará el fichero en la carpeta en la que se encuentra el fichero ejecutable.

Ahora analizaremos cada línea del programa:

FILE *fichero: Es un puntero que apunta hacia una estructura que contiene información de este fichero, nombre del fichero, tamaño que ocupa, donde está.

fichero = fopen(nombre,"r"); Con esta línea abrimos el fichero. La función fopen tiene este formato:

```
FILE *fopen(const char *nombre_fichero, const char *modo);
```

En el ejemplo habíamos leído la variable nombre y el modo de apertura era r. Aquí tenemos los diferentes modos de apertura de un fichero:

r	Abre un fichero existente para lectura.
w	Crea un fichero nuevo (o borra su contenido si existe) y lo abre para escritura.
a	Abre un fichero (si no existe lo crea) para escritura. El puntero se sitúa al final del archivo, de forma que se puedan añadir datos si borrar los existentes.

Además podemos añadir unos modificadores:

b	Abre el fichero en modo binario.
t	Abre el fichero en modo texto.
+	Abre el fichero para lectura y escritura.

if (fichero==NULL): Si por alguna razón no se ha podido abrir el fichero (por ejemplo que no exista) el valor del puntero será NULL. Es importante comprobar que se ha podido abrir el fichero.

letra=getc(fichero); getch nos lee carácter por carácter, guardamos ese valor en letra. Podríamos haber utilizado la función fgetc de la misma forma.

while (!feof(fichero)): Así comprobamos si es final de fichero. Cuando llegamos al final de fichero y leemos un carácter éste es EOF; así que podríamos haber comprobado si estamos en el final del fichero comprobando si letra cogía el valor de EOF.

if (fclose(fichero)!=0): En esta línea cerramos el fichero y comprobamos si lo hemos podido hacer correctamente. fclose devolverá 0 si se ha cerrado correctamente. Si tuviéramos el disco duro lleno podríamos tener un fallo al cerrar el fichero.

Ejercicios:

- 1- Crear un programa que nos lea un fichero de texto y nos lo imprima por pantalla y además nos imprima el número de caracteres que hay hasta cada “.”.
- 2- Hacer un programa que nos cuente el número de caracteres que hay en un fichero.
- 3- Programar un contador de vocales parecido al que ya hicimos al principio del curso pero adaptado a ficheros.
- 4- Programa que nos imprima el contenido de un fichero por pantalla pero todo en mayúsculas.

Escritura en ficheros:

Al igual que hemos hecho antes, vamos a ver como se escribe en ficheros con un ejemplo que nos hará una copia de un fichero en otro:

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

void main()
{
FILE *origen, *destino;
char letra;
char no [50], nd[50];

printf ("Fichero origen: ");
gets (no);
printf ("Fichero destino: ");
gets (nd);
origen=fopen(no,"r");
destino=fopen(nd,"w");
if (origen==NULL || destino==NULL)
{
printf( "Problemas con los ficheros.\n" );
exit( 1 );
}
letra=getc(origen);
while (feof(origen)==0)
{
putc(letra,destino);
printf( "%c",letra );
letra=getc(origen);
}
if (fclose(origen)!=0)
printf( "problemas al cerrar el fichero origen.txt\n" );
if (fclose(destino)!=0)
printf( "Problemas al cerrar el fichero destino.txt\n" );
}
```

Nos detendremos únicamente en las funciones nuevas.

`putc(letra,destino);` Su funcionamiento es igual que `getc`, pero esta función escribirá letra en el fichero destino. Con la función `fputc` habríamos obtenido el mismo resultado.

`fgets` y `fputs`: Estas funciones no están en el ejemplo pero las vamos a necesitar para los ejercicios siguientes. Hacen lo mismo que las funciones `gets` y `puts` pero hacia un fichero. Su estructura es:

```
int fputs(const char *cadena, FILE *fichero);
char fgets(char *cadena, int longitud, FILE *fichero);
```

En la función `fgets` podemos indicar el tamaño que queremos leer, esta función leerá hasta que llegue al final de línea o se lea el número de caracteres que hemos indicado (lo que se cumpla primero).

Antes de empezar con los ejercicios comentaremos que todos estos ficheros son ficheros de texto, son ficheros que podemos abrir con el bloc de notas de windows o con cualquier editor de texto. En un fichero de texto únicamente hay caracteres del código ASCII. Si guardamos el número 1000 en un fichero de texto, ocupará 4 bytes porque está formado por 4 cifras, no ocupará 2 bytes por ser de tipo entero.

Ejercicios:

- 1- Crea un programa que nos lea cada línea de un fichero de texto y nos la escriba en otro en el que se especificará el número de caracteres que hay en cada línea.
- 2- Haz un programa que nos cuente las líneas de un documento.
- 3- Modifica una vez más nuestro programa de números primos añadiendo la opción de guardarlos en un fichero y poder leer ese fichero y meterlo en una lista enlazada para no tener que volver a calcular los números que ya tengamos calculados.

fprintf y fscanf

Estas 2 funciones son muy parecidas a `printf` y `scanf`, la diferencia es que estas trabajan sobre ficheros. En estas funciones habrá que indicar el fichero sobre el que queremos operar. Ejemplo:

```
fprintf(fichero, "Tengo %i años", edad);
```

fseek y ftell

Hemos visto como leer y escribir información en un texto, pero no teníamos forma de colocarnos en la posición que quisiéramos, estas 2 funciones nos ayudarán a hacer esto.

fseek nos permite situarnos en la posición que queramos del fichero. Podremos indicar los bytes que nos queremos desplazar y desde donde lo haremos:

- SEEK_SET El puntero se desplaza desde el principio del fichero.
- SEEK_CUR El puntero se desplaza desde la posición actual del fichero.
- SEEK_END El puntero se desplaza desde el final del fichero.

La función `fseek` sigue esta forma:

```
int fseek(FILE *pfichero, long desplazamiento, int modo);
```

Aquí tenemos un ejemplo de esta función:

```
#include <stdio.h>

void main()
{
FILE *fichero;
long posicion;
int resultado;
char c, nombre[50];

printf ("Nombre del fichero: ");
gets (nombre);
fichero = fopen(nombre, "rt" );

if (fichero == NULL) printf ("FALLO");
else
{
printf( "Qué posición quieres leer? " );
scanf( "%i", &posicion );
resultado = fseek( fichero, posicion, SEEK_SET);
c = fgetc (fichero);
if (!resultado)
printf( "En la posición %i est la letra %c.\n",
posicion, c);
else
printf( "Problemas posicionando el cursor.\n" );
fclose( fichero );
}
}
```

ftell nos devuelve la posición actual dentro del fichero y nos puede servir para volver a una posición inicial después de haberla modificado. Su estructura es:

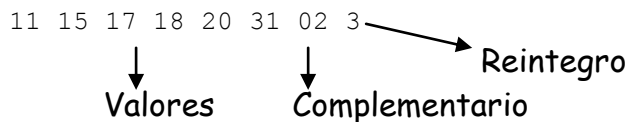
```
long ftell(FILE *pfichero);
```

Ejercicios:

1) Nos encontramos nuevamente ante un ejercicio que nos llevará unos días de clase. Es un ejercicio cogido de la URV, de la facultad de electrónica y cabe decir que es la penúltima práctica de esa asignatura (podéis ver que no hemos perdido el tiempo ☺).

El ejercicio consiste en analizar un fichero de texto que contiene una serie de números pertenecientes a unos resultados en la lotería. El fichero tiene el nombre de sorteos.txt y en su interior podéis comprobar que contiene líneas de 8 números. Los 6 primeros números pertenecen a los resultados, el siguiente es el complementario y el último el reintegro.

05	14	29	31	37	43	11	0
----	----	----	----	----	----	----	---



Deseamos obtener la siguiente información:

1. Estadística de la frecuencia de aparición de cada número del sorteo, del complementario y del reintegro. Los resultados quedarán almacenados un fichero llamado resul.txt
2. Dada una combinación de 6 números indicar por pantalla si ha aparecido o no. Se ignoran el complementario y el reintegro.
3. (Opcional) Dada una combinación de 6 o menos valores indicar por pantalla el número de veces que ha aparecido ese trozo de combinación.

* Indicaciones:

- Es muy importante hacerse un esquema antes de empezar a programar.
- Los 3 apartados del programa deben quedar cada uno en una función diferente que a su vez podrá llamar a otras.
- No os recomiendo que intentéis hacerlo todo de golpe, siempre será más fácil conseguir que compile un programa simple y poco a poco ir ampliándolo hasta llegar al ejercicio propuesto o incluso añadirle más opciones.
- En el apartado 2 se ignoran el complementario y el reintegro pero, ¿cómo los ignoramos? Yo los leería y los ignoraría sin guardarlos en ninguna variable.

2) Crea un programa que nos diga las veces que se repite cada palabra de un fichero de texto. Para conseguirlo podemos utilizar un fichero auxiliar en el que metamos todas las palabras sin repetir o podríamos hacerlo también con una lista enlazada.

Ficheros binarios

Ya hemos comentado las diferencias entre un fichero binario y un fichero de texto. Hoy lo llevaremos a la práctica. Antes de comenzar quiero que quede bien claro la diferencia que hay entre los ficheros de texto y los binarios, así que veremos un caso práctico.

```
#include <stdio.h>
void main ()
{
char cadena[]="Aprendo en STARNET\n";
int edad = 30;
FILE *f;
f = fopen ("fichero.bin", "wt");
fputs (cadena, f);
fprintf (f, "Mi edad es: %i",edad);
fclose (f);
}
```

Bien, se trata de copiar este programa y ejecutarlo. Abrimos el fichero de texto que ha creado y vemos que el resultado es el que esperábamos. Ahora cambiaremos la línea en la que abrimos el fichero en modo "wt" y pondremos "wb", volveremos a ejecutar el programa y observaremos el cambio. Bien, parece una tontería, sólo cambia en que no ha escrito el salto de línea y en su lugar ha escrito un signo raro.

Puede que penséis que el cambio no es muy grande y que yo os dije que los ficheros binarios los editamos con un editor hexadecimal y que también os dije que si guardábamos un entero deberíamos ver los 2 bytes que ocupaba... ¿Qué ha pasado? Espero que ya os imaginéis la respuesta, si no es así yo os lo digo: Lo que pasa es que hemos escrito cadenas de texto, por eso lo podemos ver con el bloc de notas, el salto de línea es normal que no lo veamos porque allí está escrito es código ASCII de la tecla enter. La explicación de porqué vemos el número es más sencilla todavía, y es que la función `fprintf` nos escribe cadenas de texto, y sin que nos demos cuenta transforma los enteros a cadenas de texto.

A continuación veremos dos funciones con las que sacaremos provecho de los ficheros binarios, son el `fwrite` y el `fread`. Estas funciones nos permiten leer o escribir cualquier tipo de datos de golpe, incluso estructuras. La estructura de estas funciones es:

```
longitud_total fwrite(void *buffer, longitud, numero, FILE *pfichero);
```

Buffer es la variable que vamos a escribir.

Tamaño es el tamaño de lo que vamos a leer o escribir, puede ser un `int`, `char`, una estructura... Aquí utilizaremos el operador `sizeof()`.

Numero es cuántos elementos voy a leer o escribir.

Pfichero es un puntero a un fichero.

Retomaremos nuestro ejemplo de la agenda pero guardando ahora la información en un fichero:

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

typedef struct {
    char nombre [8];
    char apellidos [10];
    int edad;
    float altura;
} registro;

void main ()
{
    FILE *fichero;
    registro persona;

    printf ("\nAGENDA\nPara dejar de introducir registros introducir un
nombre en blanco\n\n");
    fichero = fopen ("agenda.bin", "ab");
    do {
        printf ("Nombre: "); fflush (stdin);
        gets (persona.nombre);
        if (strcmp (persona.nombre, ""))
        {
            printf ("Apellidos: ");
            gets (persona.apellidos);
            printf ("Edad: ");
            scanf ("%i", &persona.edad);
            printf ("Altura: ");
            scanf ("%f", &persona.altura);
            fwrite (&persona, sizeof (persona), 1, fichero);
        }
    } while (1);
}
```

```
    }  
    } while (strcmp (persona.nombre, "") !=0);  
fclose (fichero);  
}
```

Aunque el código fuente de este ejemplo no es muy complicado, vamos a detenernos en algunos detalles. Observamos la línea en la que abrimos el fichero y vemos que el modo de apertura es “ab”, append binary (añadir binario), de esta forma podemos ir añadiendo a personas en nuestra agenda sin necesidad de borrar las anteriores. Hacerlo en modo binario es imprescindible en este ejercicio ya que usamos la función fwrite.

Fijémonos también en la estructura del fwrite, comentar que poner el signo & en persona no sería necesario porque persona es una estructura y devuelve una dirección de memoria, pero en Turbo C es necesario poner este signo.

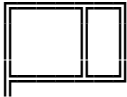
Algo que os puede llamar la atención es que he utilizado strings de tamaño reducido, la explicación es que ahora vamos a abrir el fichero creado con el bloc de notas (no entenderemos nada) y a continuación lo haremos con un editor hexadecimal y aunque al principio tampoco entenderemos nada os explicaré como funciona y podremos ver nuestra agenda.

Por costumbre pondremos la extensión txt a los ficheros de texto y bin o dat a los ficheros binarios.

Apéndice A: Puliendo nuestros programas

A medida que vamos avanzando nuestros programas se van complicando cada vez más, y algunos de ellos son dignos de ser pulidos para que tengan una mejor presentación. Para pulir nuestros programas vamos a ver una serie de funciones que nos ayudarán a darles un aspecto más atractivo.

Antes de ver estas funciones os recuerdo que todos disponemos del código ASCII en el que podemos encontrar caracteres que nos servirán para hacer **bordes y tablas**. Por ejemplo, esta tabla está hecha con los códigos 205, 201, 203, 187, 186, 202:



Estas funciones se encuentran en la librería `<conio.h>`

textcolor() ;

Cambia el color del texto de la consola. Como argumento le podemos pasar un número o un color (en mayúsculas y en inglés).

cprintf() ;

Esta función es parecida al `printf` pero con ella podremos utilizar colores si previamente hemos usado la función anterior.

Estas funciones se encuentran en la librería `<dos.h>`

sound() ;

Hace que suene el PCSpeaker con la frecuencia en Hz que le pasemos como argumento.

nosound() ;

Esta función para el sonido del PCSpeaker.

delay() ;

Con esta función detenemos el programa los milisegundos que le pasemos como argumento. Lo podemos combinar con `sound() ;` y `nosound() ;` para que suene el tiempo que queramos.