PROGRAMACIÓN EN C

elprogramador83@hotmail.com

Este material aun no esta terminado, faltan algunas correciones en los temas, falta un indice para saber exactamente donde estan los temas, pero a mi parecer para empezar con la programación "C"esta casi a punto.

Este material tiene muchas contribuciones de materiales Online, tutoriales, manuales, libros, no podria negar que en muchas partes de este manual tan solo copie lo que me parecia interesante de otros buenos manuales, bueno aunque les hice algunas modificaciones, pero al fin de al cabo lo copie, asi es que algunas de las lineas que se encuentran aquí bien podrian estar en algun otro manual que ustedes hayan leido.

Pero creo que hice eso para poner lo mejor de los manuales que encontre y sacar lo que no me parecia interesante.

Bueno ademas de este material estoy preparando otros manuales de: ALGORITMOS, C++, ESTRUCTURAS DE DATOS, JAVA y para los que les interese el curso de Ing. De Sistemas: MATEMATICAS DISCRETAS, a esperar esos manuales que van a estar my buenos

Bueno sin mas preámbulo vayamo directo al grano … a PROGRAMAR al fin carajo!!!

AQP, OCTUBRE 25 DE 2005

CAPITULO I:

INTRODUCCION

1.1 Introducción

El lenguaje **C** es uno de los más rápidos y potentes que hay hoy en dia. Algunos dicen que está en desuso y que el futuro es Java. No se si tendrá futuro pero está claro que presente si tiene. No hay más que decir que el sistema operativo Linux está desarrollado en **C** en su práctica totalidad. Así que creo que no sólo no perdemos nada aprendiendolo sino que ganamos mucho. Para empezar nos servirá como base para aprender **C++** e introducirnos en el mundo de la programación Windows. Si optamos por Linux existe una biblioteca llamada gtk (o librería, como prefieras) que permite desarrollas aplicaciones estilo windows con **C**.

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,...).

1.2 Marco Histórico

Dennis Ritchie de AT&T Bell Laboratories inventó e implementó en 1972 el primer C en un DEC PDP-11 que usaba el sistema Operativo UNIX. El Lenguaje C es el resultado de un proceso de desarrollo de un lenguaje llamado BCLP. Dicho lenguaje fue desarrollado en 1967 por Martin Richards con la idea de escribir software y compiladores en lenguaje ensamblador. A su vez el BCLP influyo en un lenguaje llamado B que invento Ken Thompson en 1970, este lenguaje surge para crear las primeras versiones del sistema operativo UNIX y se instalo en una máquina PDP-7, por los laboratorios BELL.

C surge de la necesidad de re-escribir el UNIX en un lenguaje de alto nivel, pero que algunos procedimientos se manejaran a nivel ensamblador, es entonces cuando se comienza a desarrollar un lenguaje que solucionara dicho problema. C es un lenguaje de aplicación general que sirve para escribir cualquier tipo de programa, pero su éxito y popularidad están íntimamente ligados al sistema operativo UNIX. Si se quiere dar mantenimiento a un sistema UNIX, es preciso emplear C.

C y UNIX se acoplaron tan bien que pronto no sólo los programas de sistemas, sino casi todos los programas comerciales que se ejecutaban bajo UNIX se escribieron en C. Este lenguaje se volvió tan popular que se escribieron versiones de él para otros sistemas operativos populares, así que su uso no está limitado a las computadoras que utilizan UNIX.

El sistema operativo UNIX se inició en una DEC PDP-7, en los Laboratorios Bell durante 1969. En 1973, Ritchie y Thompson re-escribieron el kernel (núcleo) de UNIX en C, rompiendo así con la tradición de que el software de sistemas está escrito en lenguaje ensamblador. Hacia 1974 fue introducido en las universidades "con fines educacionales" y al cabo de pocos años estaba ya disponible para uso comercial.

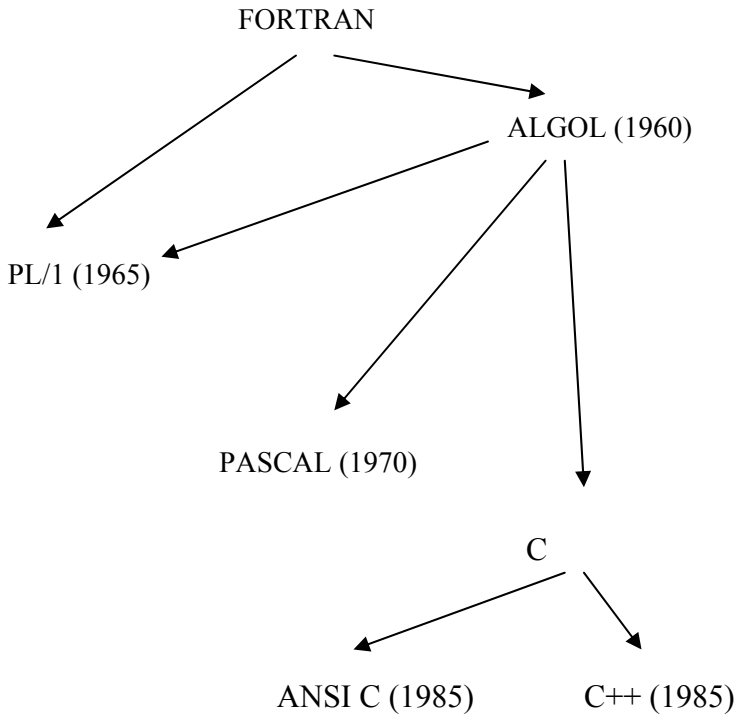
El UNIX es un sistema portable (se ejecutan en una extensa variedad de computadoras), flexible, potente, con entorno programable, multiusuario y multitarea

C trabaja con tipos de datos que son directamente tratables por el hardware de la mayoría de computadoras actuales, como son los caracteres, números y direcciones. Estos tipos de datos pueden ser manipulados por las operaciones aritméticas que proporcionan las computadoras. No proporciona mecanismos para tratar tipos de datos que no sean los básicos, debiendo ser el programador el que los desarrolle. Esto permite que el código generado sea muy eficiente y de ahí el éxito que ha tenido como lenguaje de desarrollo de sistemas.

No proporciona otros mecanismos de almacenamiento de datos que no sea el estático y no proporciona mecanismos de entrada ni salida. Ello permite que el lenguaje sea reducido y los compiladores de fácil implementación en distintos sistemas. Por contra, estas carencias se compensan mediante la inclusión de funciones de librería para realizar todas estas tareas, que normalmente dependen del sistema operativo.

Originariamente, el manual de referencia del lenguaje para el gran público fue el libro de Kernighan y Ritchie, escrito en 1977. Es un libro que explica y justifica totalmente el desarrollo de aplicaciones en **C**, aunque en él se utilizaban construcciones, en la definición de funciones, que podían provocar confusión y errores de programación que no eran detectados por el compilador. Como los tiempos cambian y las necesidades también, en 1983 ANSI establece el comité X3J11 para que desarrolle una definición moderna y comprensible del **C**. El estándar está basado en el manual de referencia original de 1972 y se desarrolla con el mismo espíritu de sus creadores originales. La primera versión de estándar se publicó en 1988 y actualmente todos los compiladores utilizan la nueva definición. Una aportación muy importante de ANSI consiste en la definición de un conjunto de librerías que acompañan al compilador y de las funciones contenidas en ellas. Muchas de las operaciones comunes con el sistema operativo se realizan a través de estas funciones. Una colección de ficheros de encabezamiento, *headers*, en los que se definen los tipos de datos y funciones incluidas en cada librería. Los programas que utilizan estas bibliotecas para interactuar con el sistema operativo obtendrán un comportamiento equivalente en otro sistema.

Relaciones del "C" con otros lenguajes de programación:



1.3 Caracteristicas Del Lenguaje

C es un lenguaje de programación de propósito general que ofrece economía sintáctica, control de flujo y estructuras sencillas y un buen conjunto de operadores. No es un lenguaje de muy alto nivel y más bien un lenguaje pequeño, sencillo y no está especializado en ningún tipo de aplicación. Esto lo hace un lenguaje potente, con un campo de aplicación ilimitado y sobre todo, se aprende rápidamente. En poco tiempo, un programador puede utilizar la totalidad del lenguaje.

Se trata de un *lenguaje estructurado*, que es la capacidad de un lenguaje de seccionar y esconder del resto del programa toda la información y las instrucciones necesarias para llevar a cabo una determinada tarea. Un lenguaje estructurado permite colocar las sentencias en cualquier parte de una línea. Todos los lenguajes modernos tienden a ser estructurados.

Este lenguaje ha sido estrechamente ligado al sistema operativo UNIX, puesto que fueron desarrollados conjuntamente. Sin embargo, este lenguaje no está ligado a ningún sistema operativo ni a ninguna máquina concreta. Se le suele llamar *lenguaje de programación de sistemas* debido a su utilidad para escribir compiladores y sistemas operativos, aunque de igual forma se pueden desarrollar cualquier tipo de aplicación.

Algunas características del lenguaje **C** son las siguientes:

- Es un lenguaje de propósito general. Este lenguaje se ha utilizado para el desarrollo de aplicaciones tan dispares como: hojas de cálculos, gestores de bases de datos, compiladores, sistemas operativos, …
- Es un lenguaje de medio nivel. Este lenguaje permite programar a alto nivel (pensando a nivel lógico y no en la máquina física) y a bajo nivel (con lo que se puede obtener la máxima eficiencia y un control absoluto de cuanto sucede en el interior del ordenador). Trabaja directo con bits, bytes y direcciones de memoria.
- Esta basado en funciones.
- Maneja una biblioteca estándar. Dicha biblioteca esta formada por un conjunto de archivos donde cada archivo esta formado por un grupo de funciones y/o procedimientos que permiten resolver problemas en forma rápida y eficiente.
- Es un lenguaje portátil. La portabiblidad significa que es posible adaptar el software escrito para un tipo de computadora o sistema operativo en otro.
- **C** no lleva a cabo comprobaciones de errores en tiempo de ejecución, es el programador el único responsable de llevar a cabo esas comprobaciones.
- Es un lenguaje potente y eficiente. Usando **C**, un programador puede casi alcanzar la eficiencia del código ensamblador junto con la estructura del Algol o Pascal.

Como desventajas habría que reseñar que es más complicado de aprender que otros lenguajes como Pascal o Basic y que requiere una cierta experiencia para poder aprovecharlo a fondo.

1.4 Compiladores de C

Un compilador es un programa que convierte nuestro código fuente en un programa ejecutable (Me imagino que la mayoría ya lo sabe, pero más vale asegurar). 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 idioma de un ordenador.

En la práctica a la hora de crear un programa nosotros escribimos el código fuente, en nuestro caso en **C**, que normalmente será un fichero de texto normal y corriente que contiene las instrucciones de nuestro programa. Luego se lo pasamos al compilador y este se encarga de convertirlo en un programa.

Si tenemos el código fuente podemos modificar el programa tantas veces como queramos (sólo tenemos que volver a compilarlo), pero si tenemos el ejecutable final no podremos cambiar nada (realmente sí se puede pero es mucho más complicado y requiere más conocimientos).

1.5 El Editor

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,...

Si nuestro compilador no trae editor la solución más simple en MS-Dos puede ser usar el edit, en windows el notepad. Pero no son más que editores sin ninguna otra funcionalidad. Otra posibilidad es un entorno de desarrollo llamado RHIDE, un programa muy útil que automatiza muchas de las tareas del programador (del estilo del Turbo C y Turbo Pascal). Si queremos una herramienta muy avanzada podemos usar Emacs, que es

un editor muy potente, aunque para algunos puede parecer muy complicado (valientes y a por ello). Estos dos programas están disponibles tanto en Linux como en MS-Dos.

1.6 Fases De Desarrollo

Los pasos a seguir desde el momento que se comienza a escribir el programa **C** hasta que se ejecuta son los siguientes:

1.6.1 Escribirlo En Un Editor

El programa se puede escribir en cualquier editor que genere ficheros de texto estándar, esto es, que los ficheros generados no incluyan códigos de control y caracteres no imprimibles.

Estos ficheros que contienen código C se llaman ficheros fuentes. Los ficheros fuentes son aquellos que contienen código fuente, es decir, ficheros con texto que el usuario puede leer y que son utilizados como entrada al compilador de C.

Los programas pequeños suelen ocupar un solo fichero fuente; pero a medida que el programa crece, se va haciendo necesario distribuirlo en más ficheos fuentes.

1.6.2 Compilarlo

El compilador produce ficheros objetos a partir de los ficheros fuentes. Los ficheros objetos son los ficheros que contienen código objeto, es decir, ficheros con código máquina (número binarios que tiene significado para el microprocesador) y que son utilizados como entrada al enlazador.

La extensión de estos ficheros es OBJ, aunque también los hay con extensión LIB. A estos últimos se les llama también ficheros de librería o biblioteca; contienen código máquina perteneciente a código compilado suministrado por el compilador.

1.6.3 Enlazarlo

El enlazador produce un fichero ejecutable a partir de los ficheros objetos.

Los ficheros ejecutables son aquellos que contienen código máquina y se pueden ejecutar directamente por el sistema operativo.

La extensión estos ficheros es EXE o COM.

Al proceso de enlazado también se le suele llamar el proceso de linkado.

1.6.4 Ejecutarlo

El programa se puede ejecutar simplemente tecleando su nombre desde la línea de comandos del sistema operativo.

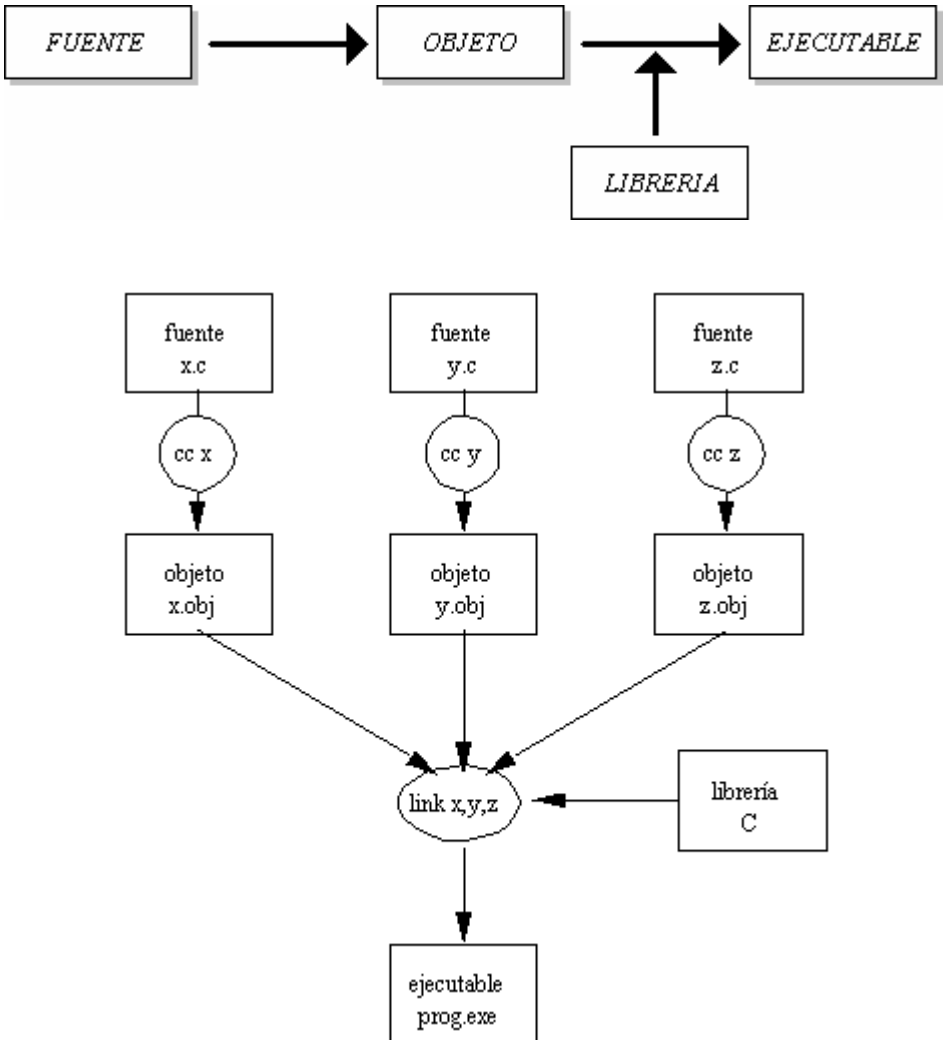
Hoy día los compiladores de **C** son muy sofisticados e incluyen entornos integrados desde los cuales editamos, compilamos, enlazamos, y podemos realizar una multitud de servicios más.

En algunos de ellos se pueden realizar los pasos de compilado, enlazado y ejecutado con la pulsación de una sola tecla.

Un programa **C** puede estar formado por diferentes módulos o fuentes. Es conveniente mantener los fuentes de un tamaño no muy grande, para que la compilación sea rápida. También, al dividirse un programa en partes, puede facilitar la legibilidad del programa y su estructuración. Los diferentes fuentes son compilados

de forma separada, únicamente los fuentes que han sido modificados desde la última compilación, y después combinados con las librerías necesarias para formar el programa en su versión ejecutable.

Los comandos necesarios para compilar, linkar y ejecutar un programa dependen del sistema operativo y debemos dirigirnos a los manuales correspondientes para conocer la sintaxis exacta.



Creación de un programa en C

1.7 Tokens

Existen seis clases de componentes sintácticos o tokens en el vocabulario del lenguaje **C**: palabras clave, identificadores, constantes, cadenas de caracteres, operadores y separadores. Los separadores –uno o varios espacios en blanco, tabuladores, caracteres de nueva línea (denominados "espacios en blanco" en conjunto), y también los comentarios escritos por el programador– se emplean para separar los demás tokens; por lo demás son ignorados por el compilador. El compilador descompone el texto fuente o programa en cada uno de sus tokens, y a partir de esta descomposición genera el código objeto correspondiente.

El compilador ignora también los sangrados al comienzo de las líneas.

1.7.1 Palabras Clave Del *C*

En C, como en cualquier otro lenguaje, existen una serie de palabras clave (keywords) que el usuario no puede utilizar como identificadores (nombres de variables y/o de funciones). Estas palabras sirven para indicar al computador que realice una tarea muy determinada (desde evaluar una comparación, hasta definir el tipo de una variable) y tienen un especial significado para el compilador. El C es un lenguaje muy conciso, con muchas menos palabras clave que otros lenguajes. A continuación se presenta la lista de las 32 palabras

clave del ANSI C, para las que más adelante se dará detalle de su significado (algunos compiladores añaden otras palabras clave, propias de cada uno de ellos. Es importante evitarlas como identificadores):

auto	break	case	char	const
continue	double	default	do	else
enum	extern	float	for	goto
int	if	long	register	return
short	signed	sizeof	static	struct
switch	typedef	union	unsigned	void
volatile	while			

1.7.2 Identificadores

Un identificador es un nombre con el que se hace referencia a una función o al contenido de una zona de la memoria (variable). Cada lenguaje tiene sus propias reglas respecto a las posibilidades de elección de nombres para las funciones y variables. En ANSI C estas reglas son las siguientes:

- Un identificador se forma con una secuencia de letras (minúsculas de la a a la z; mayúsculas de la A a la Z; y dígitos del 0 al 9).
- El carácter subrayado o underscore (_) se considera como una letra más.
- Un identificador no puede contener espacios en blanco, ni otros caracteres distintos de los citados, como por ejemplo (*,;.:-+, etc.).
- El primer carácter de un identificador debe ser siempre una letra o un (_), es decir, no puede ser un dígito.
- Se hace distinción entre letras mayúsculas y minúsculas. Así, Masa es considerado como un identificador distinto de masa y de MASA.
- No se pueden utilizar palabras reservadas como **int**, **char** o **while**.
- Muchos compiladores no permiten contener caracteres españoles (acentos y eñes).
- ANSI C permite definir identificadores de hasta 31 caracteres de longitud.

En general es muy aconsejable elegir los nombres de las funciones y las variables de forma que permitan conocer a simple vista qué tipo de variable o función representan, utilizando para ello tantos caracteres como sean necesarios. Esto simplifica enormemente la tarea de programación y –sobre todo– de corrección y mantenimiento de los programas. Es cierto que los nombres largos son más laboriosos de teclear, pero en general resulta rentable tomarse esa pequeña molestia.

Ejemplo válidos de identificadores:

```
letra;
Letra;
CHAR;
 __variable__;
cantidad_envases;
precio123;
__;
```

Ejemplo no válidos de identificadores

```
123var;        /* Empieza por dígitos */
int;           /* Palabra reservada */
una sola;      /* Contiene espacios */
US$;           /* Contiene $ */
var.nueva;     /* Contiene el punto */
eñe;           /* Puede no funcionar */
nombre?        /*No puede ir signos de admiración o interrogación*/
```

1.7.3 Comentarios

La inclusión de comentarios en un programa es una saludable práctica, como lo reconocerá cualquiera que haya tratado de leer un listado hecho por otro programador ó por sí mismo, varios meses atrás. Para el compilador, los comentarios son inexistentes, por lo que no generan líneas de código, permitiendo abundar en ellos tanto como se desee.

En el lenguaje **C** se toma como comentario todo caracter interno a los simbolos: /* */. Los comentarios pueden ocupar uno o más renglones, por ejemplo:

```
/* este es un comentario corto */

/* este otro

        es mucho
        más largo
        que el anterior */
```

Todo caracter dentro de los símbolos delimitadores es tomado como comentario incluyendo a " * " ó " (" , etc.

Los comentarios se pueden poner casi en cualquier parte. Excepto en medio de una instrucción. Por ejemplo lo siguiente no es válido:

```
pri/* Esto es un comentario */ntf( "Hola mundo" );
```

GARRAFAL

No podemos cortar a printf por en medio, tendríamos un error al compilar. Lo siguiente no daría error (al menos usando DJGPP, en Visual C++ y en Borland C++), pero es una fea costumbre:

```
printf( /* Esto es un comentario */ "Hola mundo" );
```

Y por último tenemos:

```
printf( "Hola/* Esto es un comentario */ mundo" );
```

Que no daría error, pero al ejecutar tendríamos:

```
Hola /* Esto es un comentario */ mundo
```

porque /* Esto es un comentario */ queda dentro de las comillas y C lo interpreta como texto, no como un comentario.

El lenguaje ANSI C permite también otro tipo de comentarios, tomado del C++. Todo lo que va en cualquier línea del código detrás de la doble barra (//) y hasta el final de la línea, se considera como un comentario y es ignorado por el compilador. Para comentarios cortos, esta forma es más cómoda que la anterior, pues no hay que preocuparse de cerrar el comentario (el fin de línea actúa como cierre). Como contrapartida, si un comentario ocupa varias líneas hay que repetir la doble barra (//) en cada una de las líneas. Con este segundo procedimiento de introducir comentarios, el último ejemplo podría ponerse en la forma:

```
variable_1 = variable_2; // En esta línea se asigna a
// variable_1 el valor
// contenido en variable_2
```

1.7.4 Operadores

Un operador es un carácter o grupo de caracteres que actúa sobre una, dos o más variables para realizar una determinada operación con un determinado resultado. Ejemplos típicos de operadores son la suma (+), la diferencia (-), el producto (*), etc. Los operadores pueden ser unarios, binarios y ternarios, según actúen sobre uno, dos o tres operandos, respectivamente.

En C existen muchos operadores de diversos tipos (éste es uno de los puntos fuertes del lenguaje), que se verán a continuación. Ahora solo dire las clases de operadoes que hay, ya que en el capitulo 2 explicare esto un poco mas de ellos.

1.7.4.1 Operadores Aritméticos

Los operadores aritméticos son los más sencillos de entender y de utilizar. Todos ellos son operadores binarios. En C se utilizan los cinco operadores siguientes:

- Suma (+)
- Resta (–)
- Multiplicación (*)
- División (/)

El quinto operador es el que proporciona el resto de la división (residuo)

- Módulo (%)

Todos estos operadores se pueden aplicar a constantes, variables y expresiones. El resultado es el que se obtiene de aplicar la operación correspondiente entre los dos operandos.

El único operador que requiere una explicación adicional es el operador resto %. En realidad su nombre completo es resto de la división entera. Este operador se aplica solamente a constantes, variables o expresiones de tipo int. Aclarado esto, su significado es evidente: 23%4 es 3, puesto que el resto de dividir 23 por 4 es 3. Si a%b es cero, a es múltiplo de b.

Como se verá más adelante, una expresión es un conjunto de variables y constantes –y también de otras expresiones más sencillas– relacionadas mediante distintos operadores. Un ejemplo de expresión en la que intervienen operadores aritméticos es el siguiente polinomio de grado 2 en la variable x:

```
5.0 + 3.0*x - x*x/2.0
```

Las expresiones pueden contener paréntesis (...) que agrupan a algunos de sus términos. Puede haber paréntesis contenidos dentro de otros paréntesis. El significado de los paréntesis coincide con el habitual en las expresiones matemáticas, con algunas características importantes que se verán más adelante. En ocasiones, la introducción de espacios en blanco mejora la legibilidad de las expresiones.

1.7.4.2 Operadores De Asignación Aritmética

Estos resultan de la unión de los operadores aritméticos con el operador de asignación el signo (=), o sea:

- Igual (=)
- Suma igual (+=)
- Resta igual (– =)
- Multiplicación igual (*=)
- División igual (/=)

Estos operadores se aplican de la siguiente manera: (x += 5), en este ejemplo se toma el operando de la izquierda lo suma con el operando de la derecha y lo asigna al operando de la izquierda, en este caso la variable x.

1.7.4.3 Operadores de Incremento y Decremento

El operador de incremento es el (++) y el de decremento es el (--), son operadores unarios de muy elevada prioridad y sirven para incrementar o decrementar una unidad el valor de la variable a la que afectan.

Pre-incremento y Post-incremento: Estos operadores pueden ir inmediatamente delante o detrás de la variable. Si preceden a la variable, ésta es incrementada antes de que el valor de dicha variable sea utilizado en la expresión en la que aparece. Si es la
variable la que precede al operador, la variable es incrementada después de ser utilizada en la expresión. A continuación se presenta un ejemplo de estos operadores:

```
i = 2;
j = 2;
m = i++; /* despues de ejecutarse esta sentencia m=2 e i=3*/
n = ++j; /* despues de ejecutarse esta sentencia n=3 y j=3*/
```

Estos operadores son muy utilizados. Es importante entender muy bien por qué los resultados m y n del ejemplo anterior son diferentes.

1.7.4.4 Operadores Relaciónales

Estos establecen la magnitud relativa de dos elementos y son los siguientes:

Expresión	Significado
a < b	Es a menor que b
a > b	Es a mayor que b
a == b	Es a igual a b
a != b	Es a diferente o no igual a b
a <= b	Es a menor o igual a b
a >=b	Es a mayor o igual a b

Recordemos que estas operaciones nos dan resultados lógicos de 1 ó 0 es decir valores de verdadero o falso; lenguaje **C** considera todo valor no cero como un valor verdadero.

1.7.4.5 Operadores Lógicos

C proporciona operadores lógicos para combinar los resultados de varias condiciones. Una expresión compuesta es aquella que utiliza operadores como estos y que se pueden evaluar para obtener un único resultado de verdadero o falso.

Dos de los operadores lógicos son binarios porque usan dos operandos, devuelven un resultado basado en los operandos recibidos y en el operador.

- AND (&&): Este operador conocido como producto lógico retorna un valor de verdadero si los operandos son verdaderos.
- OR (||): El operador de suma lógica retorna un valor de verdadero si los operandos o uno de los operandos es verdadero.
- NOT (!): Operador de negación, tiene por efecto invertir el resultado de la expresión que le sigue es decir si la expresión es verdadera después de aplicar este operador la expresión será falsa y viceversa.

Los operadores lógicos tienen una prioridad bastante baja, menos que los operadores de igualdad pero mayor que el operador de asignación.

1.7.4.6 Jerarquía De Operadores

Operadores	Unario / Binario	Comentario
!, &, +, -, sizeof()	Unario	Operadores Unarios
***, /, %**	Binario	Multiplicador Aritmético
+, -	Binario	Aditivos Aritméticos
<, <=, >, >=	Binario	Operadores Relaciónales
==, !=	Binario	Operadores de Igualdad
&&	Binario	Multiplicador Lógico
||	Binario	Aditivo Lógico
=	Binario	Operador de Asignación

1.8 Programas En C

1.8.1 Estructura General De Un Programa

Un fichero fuente en lenguaje **C** tendrá esta estructura típica:

```
#include <biblioteca1.h>
#include <biblioteca2.h>

... declaraciones de funciones ...
... definiciones (cuerpos de funciones) ...
... declaraciones de variables globales ...

main (void)
{
... cuerpo del main ...
}
... otras definiciones de funciones ...
```

Las declaraciones y definiciones se pueden hacer en cualquier orden, aunque es preferible declarar las funciones al principio del programa (por legibilidad).

main es simplemente una función más del programa, con la particularidad de que es el punto de entrada al programa.

1.8.2 Primer Programa

La mejor forma de aprender a programar en cualquier lenguaje es editar, compilar, corregir y ejecutar pequeños programas descriptivos. Todos los programas en C consisten en una o más funciones, la única función que debe estar presente es la denominada **main()**, siendo la primera función que se invoca cuando comienza la ejecución del programa.

Este programa se ha convertido en un clásico dentro de los libros de programación. Simplemente muestra en pantalla el mensaje HOLA MUNDO, esto que puede parecer muy tonto es algo fundamental puesto que si no sabemos imprimir mensajes ó datos en la pantalla difícilmente nuestro programa se podrá comunicar con los usuaris que lo utilicen.

Mostraremos el programa y a continuación describiremos cada una de las instrucciones que lo forman.

```
/* Programa: HOLA MUNDO */
#include <stdio.h>
int main (void)
{
        printf ("\nHola mundo");
        return 0;
}
```

Como podemos observar se trata de un programa muy sencillo.

La primera línea es lo que se conoce como un comentario, un mensaje que el programa añade al código del programa para explicar o aclarar su funcionamiento o el de una parte de él. Los comentarios se pueden situar en cualquier parte de nuestro código y se considerará como comentarios cualquier mensaje que se encuentre entre los caracteres /* y */. Los "/*" y "*/" no son caracteres, sino símbolos o banderas.

La siguiente línea es lo que se conoce como directiva del preprocesador, todos los compiladores de **C** disponen de un preprocesador, un programa que examina el código antes de compilarlo y que permite modificarlo de cara al compilador en distintos sentidos. En temas posteriores trataremos en profundidad estas directivas del preprocesador, pero para el desarrollo de los temas anteriores a este debemos conocer al menos la directiva #include. Las directivas se caracterizan por comenzar con el carácter # y se deben incluir al comienzo de la línea aunque es probable que esto dependa de la implementación del compilador con el que estemos trabajando. La directiva include permite añadir a nuestro código algún fichero de texto, de tal forma que la directiva es sustituida por el texto que contiene el fichero indicado. En general los ficheros que acompañan a esta directiva son ficheros .H (Header - Cabecera), en los que se incluyen definiciones de funciones que deseamos utilizar en nuestros programas, constantes o tipos complejos de datos.

La librería stdio.h (STandarD Input/Output) contiene las funciones estándar de entrada salida, y en ella se encuentra la función printf que utilizamos en nuestro programa. Como se observa en el código el nombre de la función a incluir debe ir entre los caracteres <...>. A medida que vayan surgiendo iremos indicando las funciones estándar que deberían incorporar todos los compiladores **C** y cual es su fichero de definición .H.

En la siguiente línea nos encontramos con int main (void). Todos los programas de **C** deben tener una función llamada main (principal en inglés). Es la que primero se ejecuta. El int (entero) que tiene al principio significa que cuando la función main acabe devolverá un número entero. Este valor se suele usar para saber cómo ha terminado el prorama. Normalmente este valor será 0 si todo ha ido bien, o un valor distinto si se ha producido algún error (pero esto lo decidimos nosotros, ya lo veremos). De esta forma si nuestro programa se ejecuta desde otro el programa 'padre' sabe como ha finalizado, si ha habido errores o no. Se puede usar la definición 'void main()', que no necesita devolver ningún valor, pero se recomienda la forma con 'int' que es más correcta.

Finalmente nos encontramos el programa principal, una sentencia printf y otra return entre llaves ({, }). Las llaves en **C** representan bloques, y encierran un conjunto de sentencias o instrucciones (lo que el computador ejecutará), considerando todas ellas como una sola, permitiendo una definición homogénea de los distintos bloques que constituyen el programa. En nuestro caso tenemos un sólo bloque que no es ni más ni menos que el programa principal, que en nuestro caso está compuesto por dos sentencias (la línea que contiene el printf y el return). Como se observa en el código las sentencias en **C** deben terminar con un punto y coma (;), #include <stdio.h> y main (void) no son sentencias, dan información sobre la estructura del programa, y por tanto no finalizan con un punto y coma.

NOTA: La razón de que la línea "int main (void)" no tenga un punto y coma ";" al final es debido a que la sentencia en sí termina al cerrar el corchete, no en que dicha línea proporcione información sobre la estructura del programa. De hecho, si el "int main (void)" constituyese una línea de prototipo tendría un ";" al final.

La función printf permite visualizar datos formateados en pantalla, es decir, permite indicar un formato como si de un impreso o formulario se tratase indicando donde se deben visualizar cada uno. En el siguiente tema

cuando se introduzcan los tipos básicos de datos se comprenderá mejor ésto. Por ahora sólo nos interesa conocer que printf visualiza mensajes en pantalla.

El mensaje debe ir entre comillas dobles (") y dentro de las comillas se puede mostrar cualquier secuencia de caracteres. El formato de esta función para este segundo tema será:

```
printf ("mensaje");
```

En el siguiente tema, cuando expliquemos en profundidad la instrucción, ampliaremos esta definición.

En nuestro programa observamos que el mensaje de texto que visualiza la instrucción printf comienza con los caracteres \n. Estos caracteres nos permiten algunas funciones especiales para controlar la forma de visualizar los mensajes, la más utilizada en \n que significa nueva línea, así nuestra sentencia printf ("\nHOLA MUNDO"); moverá el cursor (la posición de la pantalla donde actualmente se visualizan los datos) a una nueva línea situándolo a la izquierda de la pantalla y visualizará el mensaje HOLA MUNDO.

Luego viene la sentencia return. Como he indicado antes el programa al finalizar develve un valor entero. Como en este programa no se pueden producir errores (nunca digas nunca jamás) la salida siempre será 0. La forma de hacer que el programa devuelva un 0 es usando return. Esta línea significa 'finaliza la función main haz que devuelva un 0.

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

En este tema hemos aprendido a escribir programas que visualicen en pantalla mensajes utilizando la función estándar de la librería “stdio”. Antes de terminar unos breves comentarios. Una de las cosas importantes de **C** que debes recordar es que es **Case Sensitive** (sensible a las mayúsculas o algo así). Es decir que para **C** no es lo mismo escribir Printf que printf, serían identificadores distintos para el compilador, en general en **C** se suele escribir todo en minúsculas a excepción de los mensajes a visualizar (cuyo uso depende del programador) y de las constantes, esto no tiene por que hacerse así pero digamos que se trata de una tradición de la programación en **C**.

Las separaciones entre líneas también son arbitrarias y su única función es facilitar la legibilidad del código, sirviendo de separadores entre fragmentos de programa relacionados entre sí. Y esto es todo por ahora.

1.9 Entrada/Salida por consola

Algo muy usual en un programa es esperar que el usuario introduzca datos por el teclado. Para ello contamos con varias posibilidades: Usar las funciones de la biblioteca estándar, crear nuestras propias interrupciones de teclado (MS-Dos) o usar funciones de alguna biblioteca diferente (como por ejemplo Allegro).

La entrada y la salida se realizan a través de funciones de la biblioteca, que ofrece un mecanismo flexible a la vez que consistente para transferir datos entre dispositivos. Las funciones printf() y scanf() realizan la salida y entrada con formato que pueden ser controlados. Aparte hay una gran variedad de funciones E/S.

Las funciones estándar están bien para un programa sencillito. Pero cuando queremos hacer juegos por ejemplo, no suelen ser suficiente. Demasiado lentas o no nos dan todas las posibilidades que buscamos, como comprobar si hay varias teclas pulsadas. Para solucionar esto tenemos dos posibilidades:

- La más coplicada es crear nuestras propias interrupciones de teclado. ¿Qué es una interrupción de teclado? Es un pequeño programa en memoria que se ejecuta contínuamente y comprueba el estado del teclado. Podemos crear uno nuestro y hacer que el ordenador use el que hemos creado en vez del suyo. Este tema no lo vamos a tratar ahora, quizás en algún capítulo posterior.
- Otra posibilidad más sencilla es usar una biblioteca que tenga funciones para controlar el teclado. Por ejemplo si usamos la biblioteca stdio, ella misma hace todo el trabajo y nosotros no tenemos más que recoger sus frutos con un par de sencillas instrucciones. Esto soluciona mucho el trabajo y nos libra de tener que aprender cómo funcionan los aspectos más oscuros del control del teclado.

Vamos ahora con las funciones de la biblioteca estándar.

1.9.1 printf

Escribe los datos en pantalla. Se puede indicar la posición donde mostrar mediante una función gotoxy(columna,fila) o mediante las constantes de carácter. La sintaxis general que se utiliza la función pritnf() depende de la operación a realizar.

```
printf("mensaje [const_carácter]");
printf("[const_carácter]");
printf("ident(es)_formato[const_carácter]",variable(s));
```

Para utilizar la función printf en nuestros programas debemos incluir la directiva:

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

al principio de programa. Como hemos visto en el programa hola mundo.

Si sólo queremos imprimir una cadena basta con hacer (no olvides el ";" al final):

```
printf( "Cadena" );
```

Esto resultará por pantalla:

```
Cadena
```

Lo que pongamos entre las comillas es lo que vamos a sacar por pantalla. Si volvemos a usar otro printf, por ejemplo:

```
#include <stdio.h>
int main()
{
        printf( "Cadena" );
        printf( "Segunda" );
}
```

Obtendremos:

```
CadenaSegunda
```

Este ejemplo nos muestra cómo funciona printf. Para escribir en la pantalla se usa un cursor que no vemos. Cuando escribimos algo el cursor va al final del texto. Cuando el texto llega al final de la fila, lo siguiente que pongamos irá a la fila siguiente. Si lo que queremos es sacar cada una en una línea deberemos usar "\n". Es el indicador de retorno de carro. Lo que hace es saltar el cursor de escritura a la línea siguiente:

```
#include <stdio.h>

int main()
{
        printf( "Cadena\n" );
        printf( "Segunda" );
}
```

y tendremos:

```
Cadena
Segunda
```

También podemos poner más de una cadena dentro del printf:

```
printf( "Primera cadena" "Segunda cadena" );
```

Lo que no podemos hacer es meter cosas entre las cadenas:

```
printf( "Primera cadena" texto en medio  "Segunda cadena" );
```

esto no es válido. Cuando el compilador intenta interpretar esta sentencia se encuentra "Primera cadena" y luego texto en medio, no sabe qué hacer con ello y da un error.

Pero ¿qué pasa si queremos imprimir el símbolo " en pantalla? Por ejemplo imaginemos que queremos escribir:

```
Esto es "extraño"
```

Si para ello hacemos:

```
printf( "Esto es "extraño"" );
```

obtendremos unos cuantos errores. El problema es que el símbolo " se usa para indicar al compilador el comienzo o el final de una cadena. Así que en realidad le estaríamos dando la cadena "Esto es", luego extraño y luego otra cadena vacía "". Pues resulta que printf no admite esto y de nuevo tenemos errores.

La solución es usar \". Veamos:

```
printf( "Esto es \"extraño\"" );
```

Esta vez todo irá como la seda. Como vemos la contrabarra '\' sirve para indicarle al compilador que escriba caracteres que de otra forma no podríamos.

Esta contrabarra se usa en C para indicar al compilador que queremos meter símbolos especiales. Pero ¿Y si lo que queremos es usar '\' como un carácter normal y poner por ejemplo Hola\Adiós? Pues muy fácil, volvemos a usar '\':

```
printf( "Hola\\Adiós" );
```

y esta doble '\' indica a C que lo que queremos es mostrar una '\'.

En las siguientes tablas se muestran todos los identificadores de formato y las constantes de carácter las que se utilizan para realizar operaciones automáticamente sin que el usuario tenga que intervenir en esas operaciones.

IDENTIFICADORES DE FORMATO	
IDENTIFICADOR	***DESCRIPCION***
%c	Carácter
%d	Entero
%e	N. Científica

CONSTANTES DE CARÁCTER	
CONSTANTE	***DESCRIPCION***
\a	Alert (Pitido, alerta)
\b	Retroceso (backspace)
\f	Formfeed (Salto de página)

%E	N. Científica
%f	Coma flotante
%o	Octal
%s	Cadena
%u	Sin signo
%x	Hexadecimal
%X	Hexadecimal
%p	Puntero
%ld	Entero Largo
%h	Short
%%	signo %

\n	Newline (Salto de línea)
\r	carriage return (Retorno de carro)
\t	horizontal tab (Tabulador horizontal)
\v	vertical tab (Tabulador Vertical)
\’	single quote (Comilla simple)
\”	double quote (Comillas)
\\	backslash (Barra invertida)
\?	Interrogación

1.9.1.1 Especificadores de formato

Existen especificadores de formato asociados a los identificadores que alteran su significado ligeramente. Se puede especificar la longitud mínima, el número de decimales y la alineación. Estos modificadores se sitúan entre el signo de porcentaje y el identificador.

% modificador identificador

El especificador de longitud mínima hace que un dato se rellene con espacios en blanco para asegurar que este alcanza una cierta longitud mínima. Si se quiere rellenar con ceros o espacios hay que añadir un cero delante antes del especificador de longitud.

```
printf(“%f ”,numero); /*salida normal.*/
printf(“%10f ”,numero);  /*salida con 10 espacios.*/
printf(“%010f “,numero);  /*salida con los espacios poniendo 0.*/
```

El especificador de precisión sigue al de longitud mínima(si existe). Consiste en un nulo y un valor entero. Según el dato al que se aplica su función varia. Si se aplica a datos en coma flotante determina el número de posiciones decimales. Si es a una cadena determina la longitud máxima del campo. Si se trata de un valor entero determina el número mínimo de dígitos.

```
printf(“%10.4f “,numero);  /*salida con 10 espacios con 4 decimales.*/
printf(“%10.15s”,cadena); /*salida con 10 caracteres dejando 15 espacios.*/
printf(“%4.4d”,numero); /*salida de 4 dígitos mínimo. */
```

El especificador de ajuste fuerza la salida para que se ajuste a la izquierda, por defecto siempre lo muestra a la derecha. Se consigue añadiendo después del porcentaje un signo menos.

```
printf(“%8d”,numero);  /* salida ajustada a la derecha.*/
printf(“%-8d”,numero);  /*salida ajustada a la izquierda.*/
```

Por último Número octal y Número hexadecimal nos permite introducir directamente el código numérico en octal o hexadecimal del carácter que deseemos visualizar, dependiendo del que esté activo en nuestro computador. En general el código utilizado para representar internamente los caracteres por los computadores es el código ASCII (American Standard Code for Information Interchange).

1.9.1.2 Constantes de caracter

Algunos comentarios sobre estos códigos. En primer lugar el primer grupo (hasta carácter \), eran utilizados para mover el cursor en terminales. Los terminales podían ser una pantalla o una impresora, esta es la razón por la que nos encontramos cosas como avance de página o retorno de carro. Los caracteres \ ? ' " son especiales puesto que se utilizan dentro del mensaje a visualizar para indicar como se visualiza, o sea, si escribimos \ el compilador buscará el siguiente carácter y si es alguno de los anteriores los visualiza sino corresponde con ninguno simplemente lo ignora, con lo cual no podríamos visualizar el carácter \. Otro tanto sucede con las comillas puesto que para C, las comillas representan una cadena de caracteres, si escribimos " en nuestro mensaje se considerará que éste termina ahí con lo que lo que se encuentre después no tendrá sentido para el compilador.

1.9.2 scanf

Es la rutina de entrada por consola. Puede leer todos los tipos de datos incorporados y convierte los números automáticamente al formato incorporado. En caso de leer una cadena lee hasta que encuentra un carácter de espacio en blanco. El formato general:

```
scanf("identificador",&variable_numerica o char);
scanf("identificador",variable_cadena);
```

Ejemplo:

```
#include <stdio.h>
int main()
{
        int num;
        printf( "Introduce un número " );
        scanf( "%d", &num );
        printf( "Has tecleado el número %i\n", num );
}
```

Veamos cómo funciona el scanf. Lo primero nos fijamos que hay una cadena entre comillas. Esta es similar a la de printf, nos sirve para indicarle al compilador qué tipo de datos estamos pidiendo. Como en este caso es un integer usamos %d. Después de la coma tenemos la variable donde almacenamos el dato, en este caso 'num'.

Fíjate que en el scanf la variable 'num' lleva delante el símbolo &, este es muy importante, sirve para indicar al compilador cual es la dirección (o posición en la memoria) de la variable. Por ahora no te preocupes por eso, ya volveremos más adelante sobre el tema.

Podemos preguntar por más de una variable a la vez en un sólo scanf, hay que poner un %d por cada variable:

```
#include <stdio.h>
int main()
{
        int a, b, c;

        printf( "Introduce tres números: " );
        scanf( "%d %d %d", &a, &b, &c );
        printf( "Has tecleado los números %d %d %d\n", a, b, c );
}
```

De esta forma cuando el usuario ejecuta el programa debe introducir los tres datos separados por un espacio. También podemos pedir en un mismo scanf variables de distinto tipo:

```
#include <stdio.h>
int main()
{
        int a;
        float b;

        printf( "Introduce dos números: " );
        scanf( "%d %f", &a, &b );

        printf( "Has tecleado los números %d %f\n", a, b );
}
```

A cada modificador (%d, %f) le debe corresponder una variable de su mismo tipo. Es decir, al poner un %d el compilador espera que su variable correspondiente sea de tipo int. Si ponemos %f espera una variable tipo float.

Para trabajar con cadenas de texto se trabajare con arreglos, aquí solo mostraremos un pequeño ejemplo de cómo realizar esto, ya que en el capitulo de arreglos lo analizaremos con mas detalle:

```
#include<stdio.h>
void main()
{
        char a[20];
        printf("Ingrese su nombre: ");
        scanf("%s",a);
        printf("Hola Sr. %s",a);
}
```

Si nos fijamos bien en el scanf no es necesario porner &a, pero como dije esto lo veremos en el capitulo de arregles. Otra forma de escribir una cadena es:

```
#include<stdio.h>
void main()
{
        char a[20];
        printf("Ingrese su nombre: ");
        gets(a);
        printf("Hola Sr. %s",a);
}
```

La función scanf() también utiliza modificadores de formato, uno especifica el número máximo de caracteres de entrada y eliminadores de entrada.. Para especificar el número máximo solo hay que poner un entero después del signo de porcentaje. Si se desea eliminar entradas hay que añadir %*c en la posición donde se desee eliminar la entrada.

```
scanf("%10s",cadena);
scanf("%d%*c%d",&x,&y);
```

Ejemplo: permite ver la utilización de scanf con diferentes formatos

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

```
int main()
{
	static char uno[50], dos[50],tres[50], cuatro[50];
	int i=0, k;

	puts("Pulsa numeros y letras + Intro:");
	scanf("%[0-9]", uno);
	puts(uno);
	puts("\nAhora solo vamos a tomar 5 caracteres:");
	for(; i<5; i++)
		dos[i]=getche();
	printf("\n");
	puts(dos);
	fflush(stdin);
	puts("\nAhora hasta que pulse z + Intro:");
	scanf("%[^z]", tres);
	printf("\n");
	puts(tres);
	fflush(stdin);
	printf("\nHasta 8 caracteres:\n");
	scanf("%8s", cuatro);
	printf("%s", cuatro);
	getch();
	getch();
}
```

1.9.3 Gotoxy: Posicionando el cursor (DOS)

Esta función sólo está disponible en compiladores de C que dispongan de la biblioteca <conio.h>. Hemos visto que cuando usamos printf se escribe en la posición actual del cursor y se mueve el cursor al final de la cadena que hemos escrito.

Pero ¿qué pasa cuando queremos escribir en una posición determinada de la pantalla? La solución está en la función gotoxy. Supongamos que queremos escribir 'Hola' en la fila 10, columna 20 de la pantalla:

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

int main()
{
	gotoxy( 20, 10 );
	printf( "Hola" );
}
```

Fíjate que primero se pone la columna (x) y luego la fila (y). La esquina superior izquierda es la posición (1, 1).

1.9.4 Clrscr (Clear Screen): Borrar la pantalla (DOS)

Ahora ya sólo nos falta saber cómo se borra la pantalla. Pues es tan fácil como usar:

```
clrscr()
```

Esta función nó solo borra la pantalla, sino que además sitúa el cursor en la posición (1, 1), en la esquina superior izquierda.

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

int main()
{
        clrscr();
        printf( "Hola" );
}
```

Este método sólo vale para compiladores que incluyan el fichero stdio.h. Si tu sistema no lo tiene puedes consultar la sección siguiente.

1.9.5 Borrar la pantalla (otros métodos)

Existen otras formas de borrar la pantalla aparte de usar stdio.h.

Si usas DOS:

```
system ("cls"); /*Para DOS*/
```

Si usas Linux:

```
system ("clear"); /* Para Linux*/
```

Otra forma válida para ambos sistemas:

```
char a[5]={27,'[','2','J',0}; /* Para ambos (en DOS cargando antes ansi.sys) */
printf("%s",a);
```

1.9.6 Getch y getche

Si lo que queremos es que el usuario introduzca un caracter por el teclado usamos las funciones getch y getche. Estas esperan a que el usuario introduzca un carácter por el teclado. La diferencia entre getche y getch es que la primera saca por pantalla la tecla que hemos pulsado y la segunda no (la e del final se refiere a echo=eco). Ejemplos:

```
#include <stdio.h>
int main()
{
        char letra;

        printf( "Introduce una letra: " );
        fflush( stdout );
        letra = getche();
        printf( "\nHas introducido la letra: %c", letra );
}
```

Resultado:

```
Introduce una letra: a
Has introducido la letra: a
```

Quizás te estés preguntando qué es eso de fflush(stdout). Pues bien, cuando usamos la función printf, no escribimos directamente en la pantalla, sino en una memoria intermedia (lo que llaman un buffer). Cuando este buffer se llena o cuando metemos un carácter '\n' es cuando se envia el texto a la pantalla. En este ejemplo yo quería que apareciera el mensaje Introduce una letra: y el cursor se quedara justo después, es decir, sin usar '\n'. Si se hace esto, en algunos compiladores el mensaje no se muestra en pantalla hasta que se pulsa una tecla (pruébalo en el tuyo). Y la función fflush(stdout) lo que hace es enviar a la pantalla lo que hay en ese bufer. Tambien hay las funciones:

```
fflush(stdin);
fflushall();
```

Y ahora un ejemplo con getch:

```
#include <stdio.h>
int main()
{
        char letra;
        printf( "Introduce una letra: " );
        fflush( stdout );
        letra = getch();
        printf("\n has introducido la letra :%c", letra );
}
```

Resultado:

```
Introduce una letra:
Has introducido la letra: a
```

Como vemos la única diferencia es que en el primer ejemplo se muestra en pantalla lo que escribimos y en el segundo no.

1.10 Ejercicios

Ejercicio 1: Busca los errores en este programa

```
int main()
{
        /* Aquí va el cuerpo del programa */
        Printf( "Hola mundo\n" );
        return 0;
}
```

Solución:

Si lo compilamos posiblemente obtendremos un error que nos indicará que no hemos definido la función 'Printf'. Esto es porque no hemos incluído la dichosa directiva '#include <stdio.h>'. (En algunos compiladores no es necesario incluir esta directiva, pero es una buena costumbre hacerlo).

Si lo corregimos y volvemos a compilar obtendremos un nuevo error. Otra vez nos dice que desconoce 'Printf'. Esta vez el problema es el de las mayúsculas que hemos indicado antes. Lo correcto es poner 'printf' con minúsculas. Parece una tontería, pero seguro que nos da más de un problema.

Ejercicio 2: Busca los errores en el programa

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

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

Solución:

- 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 incluído 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");

Ejercicio 3: Escribe un programa que borre la pantalla y escriba en la primera línea su nombre y en la segunda su apellido:

Solución:

```
#include <stdio.h>
#include <conio.h>
int main()
{
        clrscr();
        printf( "Joseph\n" );
        printf( "Paredes" );
}
```

También se podía haber hecho todo de golpe:

```
#include <stdio.h>
#include <conio.h>
int main()
{
        clrscr();
        printf( "Joseph\nParedes" );
}
```

Ejercicio 4: Escriba un programa que borre la pantalla y muestre el texto "Arriba Perú!" en la fila 10, columna 20 de la pantalla:

Solución:

```
#include <stdio.h>
#include <conio.h>
int main()
{
        clrscr();
        gotoxy( 20, 10 );
        printf( " Arriba Peru!" );
}
```

CAPITULO II:

Tipos de datos y operadores

2.1 Introducción

La mayoría de los programas realizan algo útil y generalmente para ello es necesario trabajar con grandes cantidades de datos, si queremos realizar un programa que nos calcule un sistema de ecuaciones tenemos que indicar cuales son las ecuaciones para que el programa las resuelva. Por tanto un programa estará constituido por una serie de datos y una serie de sentencias o instrucciones que le dicen lo que tiene que hacer con esos datos.

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 entero "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=1024bytes, 1byte=8bits), pero si tenemos un programa que maneja muchos datos puede no ser una cantidad despreciable. Además ahorrar memoria es una buena costumbre.

Por si alguno tiene dudas: 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 sabeís, pero me he encontrado muchas veces con gente que los confunde.

Los lenguajes de programación disponen de una serie de tipos de datos básicos, y proporcionan herramientas para crear estructuras a medida que faciliten el acceso a la información. Así en nuestro caso ficticio de resolver un sistema de ecuaciones podemos almacenar los coeficientes de cada ecuación con lo que utilizaríamos como tipo de dato los números, si planteásemos el problema desde un punto de vista matricial nos interesaría tener un tipo de datos matriz y lo ideal sería tener un tipo de datos ecuación. En este apartado describiremos los tipos básicos que proporciona el lenguaje **C** y dejaremos para temas posteriores la declaración de tipos complejos.

El **C** dispone de estos tipos básicos:

Para el sistema operativos MSDOS (En TC++3.0 y Borland **C**++ 5.0):

TIPOS	*FORMATO*	*RANGO*	*TAMAÑO*	*DESCRIPCIÓN*
char	**%c**	-128 **a** 127	**1**	Para una letra o un dígito.
unsigned char	**%u**	0 **a** 255	**1**	Letra o número positivo.
short		-32.768 **a** 32.767	**2**	Entero corto con signo
unsigned short	**%u**	0 **a** 65.535	**2**	Entero corto sin signo
int	**%d, %i**	-32.768 **a** 32.767	**2**	Entero con signo
unsigned int	**%u**	0 **a** 65.535	**2**	Entero sin signo
short int	**%d**	-32.768 **a** 32.767	**2**	Entero con signo
long	**%ld**	-2.147.483.648 **a** 2.147.483.647	**4**	Entero largo con signo
unsigned long	**%ld**	0 **a** 4.294.967.295	**4**	Entero largo sin signo
long int	**%ld**	-2.147.483.648 **a** 2.147.483.647	**4**	Entero largo con signo

unsigned long int	**%ld**	0 **a** 4.294.967.295	**4**	Entero largo sin signo
float	**%f, %g, %e, %E**	3.4E-38 **a** 3.4E+38 decimales(6)	**4**	Para números con decimales
double	**%f, %g, %e, %E**	1.7E-308 **a** 1.7E+308 decimales(10)	**8**	Para números con decimales
long double	**%f, %g, %e, %E**	3.4E-4932 **a** 1.1E+4932 decimales(10)	**10**	Para números con decimales

Para el Win32:

- En TC 3.0 y Borland **C**++5.0 todos los datos anteriores tienen el mismo tamaño en bytes excepto el tipo int que ocupa 4 bytes y su rango es igual que un long (entero largo).

- En Visual C++ 6.0 el tipo int ocupa 4 bytes y el long double 8 bytes

Todos estos tipos salvo void son tipos numéricos. Incluso el tipo char.

Además de estos tipos podemos utilizar los modificadores signed y unsigned, los cuales son aplicables a los tipos de datos enteros

Aunque el tipo char represente caracteres internamente para el computador no es más que un número comprendido entre 0 y 255 que identifica un caracter dentro del código especificado para tal propósito en el sistema en el que nos encontremos trabajando. El código más utilizado para este tipo de representación es el ASCII.

NOTA: Según la máquina, el compilador empleado y las opciones de compilación activas, "char" puede interpretarse con signo o sin signo. Esto es, de -128 a 127 o desde 0 a 255. Si se requiere una representación que no dependa de las opciones del compilador, etc., se puede poner "signed char" o "unsigned char", según el caso.

2.2 Variables

Como decíamos antes el ordenador debe de disponer de los datos necesarios para resolver el problema para el que lo queramos programar. Difícilmente se podría resolver un sistema de ecuaciones si no se dispone de éstas. Para ello podemos definir variables. Las variables almacenan valores de un tipo especificado y en ellas almacenamos los datos de nuestro problema, se denominan variables por que su valor puede cambiar a lo largo del programa.

Una variable sólo puede pertenecer a un tipo de dato. Para poder utilizar una variable, primero tiene que ser declarada. Para referenciar una variable especificada es necesario que la podamos identificar para ello se utiliza un nombre o identificador.

Nota: Las variables no son persistentes. Cuando finaliza un programa, los datos son borrados. Si quieres crear datos persistentes, debes almacenarlos en un archivo

2.2.1 Declaración De Variables

Las **variables** se utilizan para guardar datos dentro del programa. En **C** Hay que **declarar todas** las variables antes de usarlas. Cada variable tiene un **tipo**.

Es posible inicializar y declarar más de una variable del mismo tipo en la misma sentencia:

La declaración es:

tipo nombre ;

Ejemplo:

```
int pepe;                    /* tipo de variable: int y nombre de la variable: a */
float b;
double c,d;      /* tipo de las variables: double y nombre de las variables: c y d */
char x;
```

Nota: La declaracion de variables debe hacerse al principio de la función. (Aunque algunos compiladores lo admitan, no debe hacerse en cualquier lugar).

2.2.2 Asignación De Valores

Para la asignación de valores se utiliza el signo (=) igual, por ejemplo:

```
A = 5;
B = 4;
X = 43.453;
```

En **C** es posible asignar a varias variables un solo valor ejemplo:

```
a = b = c = 7;
x = y = n = 34;
p = d = m = 56.8;
k = i = 23;
```

Algunos lenguajes automáticamente inicializan variables numéricas a cero y variables tipo carácter a blanco o a carácter nulo, en **C** no es así ya que se puede declarar la variable e inicializarla al mismo tiempo, por ejemplo:

```
int x = 5, y = 1;
```

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.

Ejemplo: Suma dos valores

```
#include <stdio.h>
#include <conio.h>
int main (void)
{
        int num1=4, num2, num3=6;
        printf("El valor de num1 es %d", num1);
        printf("\nEl valor de num3 es %d", num3);
        num2=num1+num3;
        printf("\nnum1 + num3 = %d\n", num2);
        getch();
        return 0;
}
```

Nota: Si olvidas inicializar tus variables, contendrán cualquier valor (el que hubiera en es momento en su direccion de memoria). A estos se le llama “valores basura”. Por tanto ¡Inicializa tus variables…!

2.2.3 Ámbito De Variable

Según el lugar donde se declaren las variables tendrán un ámbito. Según el ámbito de las variables pueden ser utilizadas desde cualquier parte del programa o únicamente en la función donde han sido declaradas. Las variables pueden ser:

- LOCALES: Cuando se declaran dentro de una función. Las variables locales sólo pueden ser referenciadas (utilizadas) por sentencias que estén dentro de la función que han sido declaradas. No son conocidas fuera de su función. Pierden su valor cuando se sale y se entra en la función. La declaración es como siempre.

- GLOBALES: Son conocidas a lo largo de todo el programa, y se pueden usar desde cualquier parte del código. Mantienen sus valores durante toda la ejecución. Deban ser declaradas fuera de todas las funciones incluida main (void). La sintaxis de creación no cambia nada con respecto a las variables locales. Inicialmente toman el valor 0 o nulo, según el tipo.

- DE REGISTRO: Otra posibilidad es, que en vez de ser mantenidas en posiciones de memoria del ordenador, se las guarde en registros internos del microprocesador. De esta manera el acceso a ellas es más directo y rápido. Para indicar al compilador que es una variable de registro hay que añadir a la declaración la palabra ***register*** delante del tipo. Solo se puede utilizar para variables locales.

- ESTÁTICAS: Las variables locales nacen y mueren con cada llamada y finalización de una función, sería útil que mantuvieran su valor entre una llamada y otra sin por ello perder su ámbito. Para conseguir eso se añade a una variable local la palabra static delante del tipo.

- EXTERNAS: Debido a que en **C** es normal la compilación por separado de pequeños módulos que componen un programa completo, puede darse el caso que se deba utilizar una variable global que se conozca en los módulos que nos interese sin perder su valor. Añadiendo delante del tipo la palabra ***extern*** y definiéndola en los otros módulos como global ya tendremos nuestra variable global.

2.3 Tipos de Datos

2.3.1 El Tipo de dato "int"

En una variable de este tipo se almacenan números enteros (sin decimales). Una variable tipo int se almacena en 2 bytes (16 bits), aunque algunos compiladores utilizan 4 bytes (32 bits). El ANSI C no tiene esto completamente normalizado y existen diferencias entre unos compiladores y otros.

Con 16 bits se pueden almacenar 2^{16} = 65536 números enteros diferentes: de 0 al 65535 para variables sin signo, y de -32768 al 32767 para variables con signo (que pueden ser positivas y negativas), que es la opción por defecto. Este es el rango de las variables tipo 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:

```
unsigned int numero;
int nota = 10;
```

Esto hace que declaremos una variable llamada numero que va a contener un número entero sin signo y una variable nota con signo. En este caso la variable numero podrá estar entre 0 y 65535, mientras que nota deberá estar comprendida entre -32768 al 32767. Cuando a una variable int se le asigna en tiempo de ejecución un valor que queda fuera del rango permitido (situación de overflow o valor excesivo), se produce un error en el resultado de consecuencias tanto más imprevisibles cuanto que de ordinario el programa no avisa al usuario de dicha circunstancia.

Cuando el ahorro de memoria es muy importante puede asegurarse que el computador utiliza 2 bytes para cada entero declarándolo en una de las formas siguientes:

```
short numero;
short int numero;
```

Como se ha dicho antes, ANSI C no obliga a que una variable int ocupe 2 bytes, pero declarándola como short o short int sí que necesitará sólo 2 bytes (al menos en los PCs).

Mostrar variables por pantalla

Vamos a ir u 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.

Veamos un ejemplo para resumir todo:

```
#include <stdio.h>

int main()
{
```

```
        int x = 10;
        prinft( "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.

2.3.2 El Tipo de dato "char"

Las variables de tipo **char** sirven para almacenar caracteres y se almacenan en un byte de memoria (8 bits). En un bit se pueden almacenar dos valores (0 y 1); con dos bits se pueden almacenar $2^2 = 4$ valores (00, 01, 10, 11 en binario; 0, 1 2, 3 en decimal). Con 8 bits se podrán almacenar $2^8 = 256$ valores diferentes (normalmente entre 0 y 255; con ciertos compiladores entre -128 y 127).
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;
char a, b, caracter;
```

Se puede declarar más de una variable de un tipo determinado en una sola sentencia. Se puede también inicializar la variable en la declaración. Por ejemplo, para definir la variable carácter letra y asignarle el valor a, se puede escribir:

```
char letra = 'A';
```

o

```
letra = 65;
```

A partir de ese momento queda definida la variable letra con el valor correspondiente a la letra A. Recuérdese que el valor 'A' utilizado para inicializar la variable letra es una constante carácter. En realidad, letra se guarda en un solo byte como un número entero, el correspondiente a la letra A en el código ASCII, (existe un código ASCII extendido que utiliza los 256 valores y que contiene caracteres especiales y caracteres específicos de los alfabetos de diversos países, como por ejemplo las vocales acentuadas y la letra ñ para el castellano).

En una variable char sólo podemos almacenar solo una letra, no podemos almacenar ni frases ni palabras.

En el ejemplo anterior, 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=caracter 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 );
```

En este ejemplo letra comienza con el valor 'A', que es el código ASCII 65. Al sumarle 1 pasa a tener el valor 66, que equivale a la letra 'B' (código ASCII 66). La salida de este ejemplo sería:

La letra es A y su valor ASCII es 65
Ahora es B y su valor ASCII es 66

2.3.3 El Tipo de dato "long"

Existe la posibilidad de utilizar enteros con un rango mayor si se especifica como tipo long en su declaración:

```
long int numero_grande;
```

o, ya que la palabra clave int puede omitirse en este caso,

```
long numero_grande;
```

El rango de un entero long puede variar según el computador o el compilador que se utilice, pero de ordinario se utilizan 4 bytes (32 bits) para almacenarlos, por lo que se pueden representar 2^{32} = 4.294.967.296 números enteros diferentes. Si se utilizan números con signo, podrán representarse números entre -2.147.483.648 y 2.147.483.647. También se pueden declarar enteros long que sean siempre positivos con la palabra unsigned:

```
unsigned long numero_positivo_muy_grande;
```

En algunos computadores una variable int ocupa 2 bytes (coincidiendo con short) y en otros 4 bytes (coincidiendo con long). Lo que garantiza el ANSI C es que el rango de int no es nunca menor que el de short ni mayor que el de long.

2.3.4 El Tipo de dato "float"

En muchas aplicaciones hacen falta variables reales, capaces de representar magnitudes que contengan una parte entera y una parte fraccionaria o decimal. Estas variables se llaman también de punto flotante. De ordinario, en base 10 y con notación científica, estas variables se representan por medio de la mantisa, que es un número mayor o igual que 0.1 y menor que 1.0, y un exponente que representa la potencia de 10 por la que hay que multiplicar la mantisa para obtener el número considerado. Por ejemplo, π se representa como $0.3141592654 \cdot 10^{1}$. Tanto la mantisa como el exponente pueden ser positivos y negativos.

Los computadores trabajan en base 2. Por eso un número de tipo float se almacena en 4 bytes (32 bits), utilizando 24 bits para la mantisa (1 para el signo y 23 para el valor) y 8 bits para el exponente (1 para el signo y 7 para el valor). Es interesante ver qué clase de números de punto flotante pueden representarse de esta forma. En este caso hay que distinguir el rango de la precisión. La precisión hace referencia al número de cifras con las que se representa la mantisa: con 23 bits el número más grande que se puede representar es:

$$2^{23} = 8.388.608$$

lo cual quiere decir que se pueden representar todos los números decimales de 6 cifras y la mayor parte –aunque no todos– de los de 7 cifras (por ejemplo, el número 9.213.456 no se puede representar con 23 bits). Por eso se dice que las variables tipo float tienen entre 6 y 7 cifras decimales equivalentes de precisión.

Respecto al exponente de dos por el que hay que multiplicar la mantisa en base 2, con 7 bits el número más grande que se puede representar es 127. El rango vendrá definido por la potencia:

$$2^{128} = 3.4 \cdot 10^{38}$$

lo cual indica el número más grande representable de esta forma. El número más pequeño será:

$$3.4 \cdot 10^{-38}$$

Declaración de una variable de tipo float:

```
float numero;
```

Para imprimir valores tipo float Usamos %f.

```
float num=4060.80;
printf( "El valor de num es : %f", num );
```

El resultado será:

```
El valor de num es: 4060.80
```

Si queremos escribirlo en notación exponencial usamos %e:

```
float num = 4060.80;
printf( "El valor de num es: %e", num );
```

Que da como resultado:

```
El valor de num es: 4.06080e003
```

2.3.5 El Tipo de dato "double"

Las variables tipo float tienen un rango y –sobre todo– una precisión muy limitada, insuficiente para la mayor parte de los cálculos técnicos y científicos. Este problema se soluciona con el tipo ***double***, que utiliza 8 bytes (64 bits) para almacenar una variable. Se utilizan 53 bits para la mantisa (1 para el signo y 52 para el valor) y 11 para el exponente (1 para el signo y 10 para el valor). La precisión es en este caso,

$2^{52} = 4.503.599.627.370.496$

lo cual representa entre 15 y 16 cifras decimales equivalentes. Con respecto al rango, con un exponente de 10 bits el número más grande que se puede representar será del orden de 2 elevado a 2 elevado a 10 (que es 1024):

$2^{1024} = 1.7977 \cdot 10^{308}$

Las variables tipo double se declaran de forma análoga a las anteriores:

```
double numero_real_grande;
```

Por último, existe la posibilidad de declarar una variable como ***long double***, aunque el ANSI C no garantiza un rango y una precisión mayores que las de double. Eso depende del compilador y del tipo de computador. Estas variables se declaran en la forma:

```
long double real_pero_que_muy_grande;
```

cuyo rango y precisión no está normalizado. Los compiladores de Microsoft para PCs utilizan 10 bytes (64 bits para la mantisa y 16 para el exponente).

2.4 Overflow: Qué pasa cuando nos saltamos el rango

El overflow es lo que se produce cuando intentamos almacenar en una variable un número mayor del máximo permitido. El comportamiento es distinto para variables de números enteros y para variables de números en coma flotante.

2.4.1 Con números enteros

En mi ordenador y usando Visual C++6.0 bajo Dos el tamaño del tipo int es de 4bytes (4*8=32bits). El número máximo que se puede almacenar en una variable tipo int es por tanto 2.147.483.647. Si nos pasamos de este número el que se guardará será el siguiente pero empezando desde el otro extremo, es decir, el -2.147.483.648. El compilador seguramente nos dará un aviso (warning) de que nos hemos pasado.

```
#include <stdio.h>
int main(void)
{
        int num1;
        num1 = 2147483648;
        printf( "El valor de num1 es: %i\n", num1 );
}
```

El resultado que obtenemos es:

```
El valor de num1 es: -2147483648
```

Comprueba si quieres que con el número anterior (2.147.483.647) no pasa nada.

2.4.2 Con números en coma flotante

El comportamiento con números en coma flotante es distinto. Dependiendo del ordenador si nos pasamos del rango al ejecutar un programa se puede producir un error y detenerse la ejecución.
Con estos números también existe otro error que es el underflow. Este error se produce cuando almacenamos un número demasiado pequeño (3,4E-38 en float).

2.5 Conversion de tipos

Cuando escribimos una expresión aritmética a+b, en la cual hay variables o valores de distintos tipos, el compilador realiza determinadas conversiones antes de que evalúe la expresión. Estas conversiones pueden ser para 'aumentar' o 'disminuir' la precisión del tipo al que se convierten los elementos de la expresión. Un ejemplo claro, es la comparación de una variable de tipo int con una variable de tipo double. En este caso, la de tipo int es convertida a double para poder realizar la comparación.

Los tipos pequeños son convertidos de la forma siguiente: un tipo char se convierte a int, con el modificador signed si los caracteres son con signo, o unsigned si los caracteres son sin signo. Un unsigned char es convertido a int con los bits más altos puestos a cero. Un signed char es convertido a int con los bits más altos puestos a uno o cero, dependiendo del valor de la variable.

Para los tipos de mayor tamaño: si un operando es de tipo double, el otro es convertido a double. Si un operando es de tipo float, el otro es convertido a float. Si un operando es de tipo unsigned long, el otro es convertido a unsigned long. Si un operando es de tipo long, el otro es convertido a long. Si un operando es de tipo unsigned, el otro es convertido a unsigned. Si no, los operandos son de tipo int.

Otra clase de conversión implícita tiene lugar cuando el resultado de una expresión es asignado a una variable, pues dicho resultado se convierte al tipo de la variable (en este caso, ésta puede ser de menor rango que la expresión, por lo que esta conversión puede perder información y ser peligrosa). Por ejemplo, si i y j son variables enteras y x es double:

```
x = i*j – j + 1;
```

En C existe también la posibilidad de realizar conversiones explícitas de tipo (llamadas casting, en la literatura inglesa). El casting es pues una conversión de tipo, forzada por el programador. Para ello basta preceder la constante, variable o expresión que se desea convertir por el tipo al que se desea convertir, encerrado entre paréntesis. En el siguiente ejemplo,

```
k = (int) 1.7 + (int) masa;
```

la variable masa es convertida a tipo int, y la constante 1.7 (que es de tipo double) también. El casting se aplica con frecuencia a los valores de retorno de las funciones.

Ejemplo:

```
#include <stdio.h>
#include <conio.h>
int main (void)
{
        float a;
        int b;
        a = 2.8;
        b = 10;
        printf("\na (real) = %f", a); /* a = 2.8000 */
        printf("\na convertido en entero: %d", (int) a); /* a = 2 */
        printf("\nb (entero): %d", b); /* b = 10 */
        printf("\nb convertido en real: %f", (float) b); /* b = 10.0000 */
        getch();
        return 0;
}
```

En C++ tambien es valido escribir el tipo y luego poner la variable entre parentesis:

```
#include <stdio.h>
#include <conio.h>
int main (void)
{
        float a;
        int b;
        a = 2.8;
        b = 10;
        printf("\na (real) = %f", a); /* a = 2.8000 */
        printf("\na convertido en entero: %d", int (a)); /* a = 2 */
        printf("\nb (entero): %d", b); /* b = 10 */
        printf("\nb convertido en real: %f", float (b)); /* b = 10.0000 */
        getch();
        return 0;
}
```

2.6 Duración y visibilidad de las variables: Modos de almacenamiento

El tipo de una variable se refiere a la naturaleza de la información que contiene (ya se han visto los tipos char, int, long, float, double, etc.).

El modo de almacenamiento (storage class) es otra característica de las variables de C que determina cuándo se crea una variable, cuándo deja de existir y desde dónde se puede acceder a ella, es decir, desde dónde es visible.

En C existen 4 modos de almacenamiento fundamentales: auto, extern, static y register. Seguidamente se exponen las características de cada uno de estos modos.

2.6.1 auto (automático)

Es la opción por defecto para las variables que se declaran dentro de un bloque {...}, incluido el bloque que contiene el código de las funciones. En C la declaración debe estar siempre al comienzo del bloque. En C++ la declaración puede estar en cualquier lugar y hay autores que aconsejan ponerla justo antes del primer uso de la variable. No es necesario poner la palabra ***auto***. Cada variable ***auto*** es creada al comenzar a ejecutarse el bloque y deja de existir cuando el bloque se termina de ejecutar. Cada vez que se ejecuta el bloque, las variables ***auto*** se crean y se destruyen de nuevo. Las variables ***auto*** son variables locales, es decir, sólo son visibles en el bloque en el que están definidas y en otros bloques anidados en él, aunque pueden ser ocultadas por una nueva declaración de una nueva variable con el mismo nombre en un bloque anidado. No son inicializadas por defecto, y –antes de que el programa les asigne un valor– pueden contener basura informática (conjuntos aleatorios de unos y ceros, consecuencia de un uso anterior de esa zona de la memoria).

A continuación se muestra un ejemplo de uso de variables de modo auto.

```
{
        int i=1, j=2; /* se declaran e inicializan i y j */
        ...
        {
                float a=7., j=3.; /* se declara una nueva variable j */
                ...
                j=j+a; /* aqui j es float */
                ... /* la variable int j es invisible*/
                ... /* la variable i=1 es visible*/
        }
        ... /* fuera del bloque, a ya no existe*/
```

```
        ... /* la variable j=2 existe y es entera*/
}
```

2.6.2 extern

Son variables globales, que se definen fuera de cualquier bloque o función, por ejemplo antes de definir la función main(). Estas variables existen durante toda la ejecución del programa. Las variables extern son visibles por todas las funciones que están entre la definición y el fin del fichero. Para verlas desde otras funciones definidas anteriormente o desde otros ficheros, deben ser declaradas en ellos como variables ***extern***. Por defecto, son inicializadas a cero.

Una variable extern es definida o creada (una variable se crea en el momento en el que se le reserva memoria y se le asigna un valor) una sola vez, pero puede ser declarada (es decir, reconocida para poder ser utilizada) varias veces, con objeto de hacerla accesible desde diversas funciones o ficheros. También estas variables pueden ocultarse mediante la declaración de otra variable con el mismo nombre en el interior de un bloque.

La variables ***extern*** permiten transmitir valores entre distintas funciones, pero ésta es una práctica considerada como peligrosa. A continuación se presenta un ejemplo de uso de variables extern.

```
int i=1, j, k; /* se declaran antes de main()*/

main()
{
        int i=3; /* i=1 se hace invisible */
        int func1(int, int);
        ... /* j, k son visibles*/
}
int func1(int i, int m)
{
        int k=3; /* k=0 se hace invisible*/
        ... /* i=1 es invisible*/
}
```

2.6.3 static

Cuando ciertas variables son declaradas como ***static*** dentro de un bloque, estas variables conservan su valor entre distintas ejecuciones de ese bloque. Dicho de otra forma, las variables ***static*** se declaran dentro de un bloque como las ***auto***, pero permanecen en memoria durante toda la ejecución del programa como las ***extern***.

Cuando se llama varias veces sucesivas a una función (o se ejecuta varias veces un bloque) que tiene declaradas variables ***static***, los valores de dichas variables se conservan entre dichas llamadas. La inicialización sólo se realiza la primera vez. Por defecto, son inicializadas a cero.

Las variables definidas como static extern son visibles sólo para las funciones y bloques comprendidos desde su definición hasta el fin del fichero. No son visibles desde otras funciones ni aunque se declaren como extern. Ésta es una forma de restringir la visibilidad de las variables.

Por defecto, y por lo que respecta a su visibilidad, las funciones tienen modo extern.

Una función puede también ser definida como static, y entonces sólo es visible para las funciones que están definidas después de dicha función y en el mismo fichero. Con estos modos se puede controlar la visibilidad de una función, es decir, desde qué otras funciones puede ser llamada.

2.6.4 register

Este modo es una recomendación para el compilador, con objeto de que –si es posible– ciertas variables sean almacenadas en los registros de la CPU y los cálculos con ellas sean más rápidos. No existen los modos auto y register para las funciones.

2.7 Constantes

Se entiende por constantes aquel tipo de información numérica o alfanumérica que no puede cambiar más que con una nueva compilación del programa. Como ya se ha dicho anteriormente, en el código de un programa en C pueden aparecer diversos tipos de constantes que se van a explicar a continuación.

2.7.1 Constantes numéricas

2.7.1.1 Constantes Enteras

Una constante entera decimal está formada por una secuencia de dígitos del 0 al 9, constituyendo un número entero. Las constantes enteras decimales están sujetas a las mismas restricciones de rango que las variables tipo int y long, pudiendo también ser unsigned. El tipo de una constante se puede determinar automáticamente según su magnitud, o de modo explícito postponiendo ciertos caracteres, como en los ejemplos que siguen:

23484	constante tipo int
45815	constante tipo long (es mayor que 32767)
253u ó 253U	constante tipo unsigned int
739l ó 739L	constante tipo long
583ul ó 583UL	constante tipo unsigned long

En C se pueden definir también constantes enteras octales, esto es, expresadas en base 8 con dígitos del 0 al 7. Se considera que una constante está expresada en base 8 si el primer dígito por la izquierda es un cero (0). Análogamente, una secuencia de dígitos (del 0 al 9) y de letras (A, B, C, D, E, F) precedida por 0x o por 0X, se interpreta como una constante entera hexadecimal, esto es, una constante numérica expresada en base 16. Por ejemplo:

011	constante octal (igual a 9 en base 10)
11	constante entera decimal (no es igual a 011)
0xA	constante haxadecimal (igual a 10 en base 10)
0xFF	constante hexadecimal (igual a 162-1=255 en base 10)

Es probable que no haya necesidad de utilizar constantes octales y hexadecimales, pero conviene conocer su existencia y saber interpretarlas por si hiciera falta. La ventaja de los números expresados en base 8 y base 16 proviene de su estrecha relación con la base 2 (8 y 16 son potencias de 2), que es la forma en la que el ordenador almacena la información.

2.7.1.2 CONSTANTES DE PUNTO FLOTANTE

Como es natural, existen también constantes de punto flotante, que pueden ser de tipo float, double y long double. Una constante de punto flotante se almacena de la misma forma que la variable correspondiente del mismo tipo. Por defecto –si no se indica otra cosa– las constantes de punto flotante son de tipo double. Para indicar que una constante es de tipo float se le añade una f o una F; para indicar que es de tipo long double, se le añade una l o una L. En cualquier caso, el punto decimal siempre debe estar presente si se trata de representar un número real.

Estas constantes se pueden expresar de varias formas. La más sencilla es un conjunto de dígitos del 0 al 9, incluyendo un punto decimal. Para constantes muy grandes o muy pequeñas puede utilizarse la notación científica; en este caso la constante tiene una parte entera, un punto decimal, una parte fraccionaria, una e o

E, y un exponente entero (afectando a la base 10), con un signo opcional. Se puede omitir la parte entera o la fraccionaria, pero no ambas a la vez. Las constantes de punto flotante son siempre positivas. Puede anteponerse un signo (-), pero no forma parte de la constante, sino que con ésta constituye una expresión, como se verá más adelante. A continuación se presentan algunos ejemplos válidos:

1.23	constante tipo double (opción por defecto)
23.963f	constante tipo float
.00874	constante tipo double
23e2	constante tipo double (igual a 2300.0)
.874e-2	constante tipo double en notación científica (=.00874)
.874e-2f	constante tipo float en notación científica

Seguidos de otros que no son correctos:

1,23	error: la coma no esta permitida
23963f	error: no hay punto decimal ni carácter e ó E
.e4	error: no hay ni parte entera ni fraccionaria
-3.14	error: sólo el exponente puede llevar signo

2.7.2 Constantes carácter

Una constante carácter es un carácter cualquiera encerrado entre apóstrofos (tal como 'x' o 't'). El valor de una constante carácter es el valor numérico asignado a ese carácter según el código ASCII (ver Tabla 2.3). Conviene indicar que en C no existen constantes tipo char; lo que se llama aquí constantes carácter son en realidad constantes enteras.

Hay que señalar que el valor ASCII de los números del 0 al 9 no coincide con el propio valor numérico. Por ejemplo, el valor ASCII de la constante carácter '7' es 55.

Ciertos caracteres no representables gráficamente, el apóstrofo (') y la barra invertida (\) y otros caracteres, se representan mediante tabla de ***secuencias de escape*** que vimos en el capitulo anterior, con ayuda de la barra invertida (\).

Los caracteres ASCII pueden ser también representados mediante el número octal correspondiente, encerrado entre apóstrofos y precedido por la barra invertida. Por ejemplo, '\07' y '\7' representan el número 7 del código ASCII (sin embargo, '\007' es la representación octal del carácter '7'), que es el sonido de alerta. El ANSI C también admite secuencias de escape hexadecimales, por ejemplo '\x1a'.

2.7.3 Cadenas de caracteres

Una cadena de caracteres es una secuencia de caracteres delimitada por comillas ("), como por ejemplo: "Esto es una cadena de caracteres". Dentro de la cadena, pueden aparecer caracteres en blanco y se pueden emplear las mismas secuencias de escape válidas para las constantes carácter. Por ejemplo, las comillas (") deben estar precedidas por (\), para no ser interpretadas como fin de la cadena; también la propia barra invertida (\). Es muy importante señalar que el compilador sitúa siempre un byte nulo (\0) adicional al final de cada cadena de caracteres para señalar el final de la misma. Así, la cadena "mesa" no ocupa 4 bytes, sino 5 bytes. A continuación se muestran algunos ejemplos de cadenas de caracteres:

```
"Informática I"
"'A'"
" cadena con espacios en blanco "
"Esto es una \"cadena de caracteres\".\n"
```

2.7.4 Constantes de tipo Enumeración

En C existen una clase especial de constantes, llamadas constantes enumeración. Estas constantes se utilizan para definir los posibles valores de ciertos identificadores o variables que sólo deben poder tomar unos pocos valores. Por ejemplo, se puede pensar en una variable llamada dia_de_la_semana que sólo pueda tomar los 7 valores siguientes: lunes, martes, miercoles, jueves, viernes, sabado y domingo. Es muy fácil imaginar otros tipos de variables análogas, una de las cuales podría ser una variable booleana con sólo dos posibles valores: SI y NO, o TRUE y FALSE, u ON y OFF. El uso de este tipo de variables hace más claros y legibles los programas, a la par que disminuye la probabilidad de introducir errores.

En realidad, las constantes enumeración son los posibles valores de ciertas variables definidas como de ese tipo concreto.

Las enumeraciones se crean con enum:

```
enum nombre_de_la_enumeración
{
        nombres de las constantes
};
```

Ejemplo 1:

```
enum dia {lunes, martes, miercoles, jueves, viernes, sabado, domingo};
```

Esta declaración crea un nuevo tipo de variable –el tipo de variable dia– que sólo puede tomar uno de los 7 valores encerrados entre las llaves. Estos valores son en realidad constantes tipo int: lunes es un 0, martes es un 1, miercoles es un 2, etc. Ahora, es posible definir variables, llamadas dia1 y dia2, que sean de tipo dia, en la forma (obsérvese que en C deben aparecer las palabras enum dia; en C++ basta que aparezca la palabra dia)

```
enum dia dia1, dia2;  /* esto es C*/
dia dia 1, dia 2;     /* esto es C++*/
```

y a estas variables se les pueden asignar valores en la forma

```
dia1 = martes;
```

o aparecer en diversos tipos de expresiones y de sentencias que se explicarán más adelante.

Los valores enteros que se asocian con cada constante tipo enumeración pueden ser controlados por el programador. Por ejemplo, la declaración,

```
enum dia {lunes=1, martes, miercoles, jueves, viernes, sabado, domingo};
```

asocia un valor 1 a lunes, 2 a martes, 3 a miercoles, etc., mientras que la declaración,

```
enum dia {lunes=1, martes, miercoles, jueves=7, viernes, sabado, domingo};
```

asocia un valor 1 a lunes, 2 a martes, 3 a miercoles, un 7 a jueves, un 8 a viernes, un 9 a sabado y un 10 a domingo.

Se puede también hacer la definición del tipo enum y la declaración de las variables en una única sentencia, en la forma

```
enum palo {oros, copas, espadas, bastos} carta1, carta2, carta3;
```

donde carta1, carta2 y carta3 son variables que sólo pueden tomar los valores oros, copas, espadas y bastos (equivalentes respectivamente a 0, 1, 2 y 3).

Ejemplo 2:

```
enum { primero, segundo, tercero, cuarto, quinto };
```

De esta forma hemos definido las constantes primero, segundo,... quinto. Si no especificamos nada la primera constante (primero) toma el valor 0, la segunda (segunda) vale 1, la tercera 2,... Podemos cambiar estos valores predeterminados por los valores que deseemos:

```
enum { primero=1, segundo, tercero, cuarto, quinto };
```

Ahora primero vale 1, segundo vale 2, tercero vale 3,... Cada constante toma el valor de la anterior más uno. Si por ejemplo hacemos:

```
enum { primero=1, segundo, quinto=5, sexto, septimo };
```

Tendremos: primero=1, segundo=2, quinto=5, sexto=6, septimo=7.

```
#include <stdio.h>

enum { primero=1, segundo, tercero, cuarto, quinto } posicion;

int main()
{
        posicion=segundo;
        printf("posicion = %i\n", posicion);
}
```

Las constantes definidas con enum sólo pueden tomar valores enteros (pueden ser negativos). Son equivalentes a las variables de tipo int.

Un error habitual suele ser pensar que es posible imprimir el nombre de la constante:

```
#include <stdio.h>

enum { primero=1, segundo, tercero, cuarto, quinto } posicion;

int main()
{
        printf("posicion = %s\n", posicion);
}
```

Es habitual pensar que con este ejemplo tendremos como resultado: posicion = segundo. Pero no es así, en todo caso tendremos: posicion = (null).

Debemos pensar en las enumeraciones como una forma de hacer el programa más comprensible para los humanos, en nuestro ejemplo el ordenador donde vea segundo pondrá un 2 en su lugar.

2.7.5 Const

Se puede utilizar el cualificador const en la declaración de una variable para indicar que esa variable no puede cambiar de valor. Si se utiliza con un array, los elementos del array no pueden cambiar de valor. Su sinteaxis es:

```
const tipo nombre=valor_entero;
const tipo nombre=valor_entero.valor_decimal;
const tipo nombre='carácter';
```

Por ejemplo:

```
const int i=10;
const double x[] = {1, 2, 3, 4};
```

El lenguaje C no define lo que ocurre si en otra parte del programa o en tiempo de ejecución se intenta modificar una variable declarada como const. De ordinario se obtendrá un mensaje de error en la compilación si una variable const figura a la izquierda de un operador de asignación. Sin embargo, al menos con el compilador Visual C++ 6.0, se puede modificar una variable declarada como const por medio de un puntero de la forma siguiente:

```
const int i=10;
int *p;
p = &i;
*p = 1;
```

C++ es mucho más restrictivo en este sentido, y no permite de ninguna manera modificar las variables declaradas como const.

El cualificador const se suele utilizar cuando, por motivos de eficiencia, se pasan argumentos por referencia a funciones y no se desea que dichos argumentos sean modificados por éstas.

2.7.6 #define

La directiva #define es un identificador y una secuencia de caracteres que sustituirá se sustituirá cada vez que se encuentre éste en el archivo fuente. También pueden ser utilizados para definir valores numéricos constantes.

Supongamos que tenemos que hacer un programa en el que haya que escribir unas cuantas veces 3.1416 (como todos sabreis es PI). Es muy fácil que nos confundamos alguna vez al teclearlo y al compilar el programa no tendremos ningún error, sin embargo el programa no dará resultados correctos. Para evitar esto usamos las constantes con nombre. Al definir una constante con nombre estamos dando un nombre al valor, a 3.1416 le llamamos PI.

Estas constantes se definen de la manera siguiente:

```
#define nombre_de_la_constante    valor_de_la_constante
#define nombre_de_la_constante    "Cadena"
```

Ejemplo:

```
#include <stdio.h>

#define PI      3.1416

int main()
```

```
{
        int radio, perimetro;

        radio = 20;
        perimetro = 2 * PI * radio;
        printf( "El perímetro es: %i", perimetro  );
}
```

De esta forma cada vez que el compilador encuentre el nombre PI lo sustituirá por 3.1416.

A una constante no se le puede dar un valor mientras se ejecuta, no se puede hacer PI = 20;. Sólo se le puede dar un valor con #define, y sólo una vez. Tampoco podemos usar el scanf para dar valores a una constante:

```
#define CONSTANTE          14
int main()
{
        ...
        scanf( "%i", CONSTANTE );
        ...
}
```

eso sería como hacer:

```
scanf( "%i", 14 );
```

Esto es muy grave, estamos diciendo que el valor que escribamos en scanf se almacene en la posición 14 de la memoria, lo que puede bloquear el ordenador.

Las constantes se suelen escribir en mayúsculas sólo se puede definir una constante por fila. No se pone ";" al final. Las constantes nos proporcionan una mejor comprensión del código fuente. Mira el siguiente programa:

```
#include <stdio.h>

int main()
{
        int precio;
        precio = ( 4 * 25 * 100 ) * ( 1.16 );
        printf( "El precio total es: %i", precio  );
}
```

¿Quien entiende lo que quiere decir este programa? Es difícil si no imposible. Hagámoslo de otra forma.

```
#include <stdio.h>

#define CAJAS                   4
#define UNIDADES_POR_CAJA       25
#define PRECIO_POR_UNIDAD       100
#define IMPUESTOS   1.16

int main()
{
        int precio;
```

```
        precio = ( CAJAS * UNIDADES_POR_CAJA * PRECIO_POR_UNIDAD ) * (
IMPUESTOS );

        printf( "El precio total es: %i", precio  );
}
```

¿A que ahora se entiende mejor? Claro que sí. Los números no tienen mucho significado y si revisamos el programa un tiempo más tarde ya no nos acordaremos qué cosa era cada número. De la segunda forma nos enteraremos al momento.

También podemos definir una constante usando el valor de otras. Por supuesto las otras tienen que estar definidas antes:

```
#include <stdio.h>

#define CAJAS  4
#define UNIDADES_POR_CAJA  25
#define PRECIO_POR_UNIDAD  100
#define PRECIO_POR_CAJA    UNIDADES_POR_CAJA * PRECIO_POR_UNIDAD
#define IMPUESTOS  1.16

int main()
{
        int precio;

        precio = ( CAJAS * PRECIO_POR_CAJA ) * ( IMPUESTOS );

        printf( "El precio total es: %i", precio  );
}
```

#define tiene más usos aparte de éste, ya los veremos en el capítulo de directivas.

2.8 Operadores

Como ya dije en el primer capitulo, un operador sirve para manipular datos. Los hay de varios tipos: de asignación, de relación, lógicos, aritméticos y de manipulación de bits. En realidad los nombres tampoco importan mucho; aquí lo que queremos es aprender a programar, no aprender un montón de nombres.

C es un lenguaje muy rico en operadores incorporados, es decir, implementados al realizarse el compilador. Define cuatro tipos de operadores: aritméticos, relacionales, lógicos y a nivel de bits. También se definen operadores para realizar determinadas tareas, como las asignaciones.

2.8.1 ASIGNACION (=)

Los operadores de asignación atribuyen a una variable –es decir, depositan en la zona de memoria correspondiente a dicha variable– el resultado de una expresión o el valor de otra variable (en realidad, una variable es un caso particular de una expresión).

El operador de asignación más utilizado es el operador de igualdad (=), que no debe ser confundido con la igualdad matemática. Es posible realizar asignaciones múltiples, igualar variables entre si y a un valor. Su sintaxis es:

```
variable=valor;
variable=variable1;
variable=variable1=variable2=variableN=valor;
```

```
nombre_de_variable = expresion;
```

cuyo funcionamiento es como sigue: se evalúa expresion y el resultado se deposita en nombre_de_variable, sustituyendo cualquier otro valor que hubiera en esa posición de memoria anteriormente. Una posible utilización de este operador es como sigue:

```
variable = variable + 1;
```

Desde el punto de vista matemático este ejemplo no tiene sentido (¡Equivale a 0 = 1!), pero sí lo tiene considerando que en realidad el operador de asignación (=) representa una sustitución; en efecto, se toma el valor de variable contenido en la memoria, se le suma una unidad y el valor resultante vuelve a depositarse en memoria en la zona correspondiente al identificador variable, sustituyendo al valor que había anteriormente. El resultado ha sido incrementar el valor de variable en una unidad.

Así pues, una variable puede aparecer a la izquierda y a la derecha del operador (=). Sin embargo, a la izquierda del operador de asignación (=) no puede haber nunca una expresión: tiene que ser necesariamente el nombre de una variable5. Es incorrecto, por tanto, escribir algo así como:

```
a + b = c; // incorrecto
```

Existen otros cuatro operadores de asignación (+=, -=, *= y /=) formados por los 4 operadores aritméticos seguidos por el carácter de igualdad. Estos operadores simplifican algunas operaciones recurrentes sobre una misma variable. Su forma general es:

```
variable op= expresion;
```

donde op representa cualquiera de los operadores (+ - * /). La expresión anterior es equivalente a:

```
variable = variable op expresion;
```

A continuación se presentan algunos ejemplos con estos operadores de asignación:

```
distancia += 1;        equivale a:    distancia = distancia + 1;
rango /= 2.0           equivale a:    rango = rango /2.0
x *= 3.0 * y - 1.0     equivale a:     x = x * (3.0 * y - 1.0)
```

2.8.2 Operadores Aritmeticos

Los operadores aritméticos son aquellos que sirven para realizar operaciones tales como suma, resta, división y multiplicación.

Pueden aplicarse a todo tipo de expresiones. Son utilizados para realizar operaciones matemáticas sencillas, aunque uniéndolos se puede realizar cualquier tipo de operaciones.

2.8.2.1 Operador (+) : Suma

Este operador permite sumar variables:

```
#include <stdio.h>
int main()
{
        int a = 2;
        int b = 3;
        int c;
```

```
        c = a + b;
        printf ( "Resultado = %i\n", c );
}
```

El resultado será 5 obviamente.

Por supuesto se pueden sumar varias variables o variables más constantes:

```
#include <stdio.h>

int main()
{
        int a = 2;
        int b = 3;
        int c = 1;
        int d;
        d = a + b + c + 4;
        printf ( "Resultado = %i\n", c );
}
```

El resultado es 10.

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

```
x = x + 5;
```

Esto suma el valor 5 al valor que tenía la variable x. Veamos un ejemplo:

```
#include <stdio.h>
int main()
{
        int x, y;

        x = 3;
        y = 5;

        x = x + 2;
        printf( "x = %i\n", x );
        x = x+ y;
        printf( "x = %i\n", x );
}
```

Resultado:

```
x = 5
x = 10
```

2.8.2.2 Operador (-) : Resta/Negativo

Este operador tiene dos usos, uno es la resta que funciona como el operador suma y el otro es cambiar de signo.

Resta:

```
x = x - 5;
```

Para la operación resta se aplica todo lo dicho para la suma. Se puede usar también como: x -= 5;.

Pero también tiene el uso de cambiar de signo. Poniendolo delante de una variable o constante equivale a multiplicarla por -1.

```
#include <stdio.h>
int main(void)
{
        int a, b;

        a = 1;
        b = -a;
        printf( "a = %i, b = %i\n", a, b );
}
```

Resultado: a = 1, b = -1. No tiene mucho misterio.

2.8.2.3 Operador (*) : Multiplicación y punteros

Este operador sirve para multiplicar y funciona de manera parecida a los anteriores.

También sirve para definir y utilizar punteros, pero eso lo veremos más tarde.

2.8.2.4 Operador (/) : División

Este funciona también como los anteriores pero hay que tener cuidado. Si dividimos dos número en coma flotante (tipo float) tenemos las división con sus correspondientes decimales. Pero si dividimos dos enteros obtenemos un número entero. Es decir que si dividimos 4/3 tenemos como resultado 1. El redondeo se hace por truncamiento, simplemente se eliminan los decimales y se deja el entero.

Si dividimos dos enteros el resultado es un número entero, aunque luego lo saquemos por pantalla usando %f o %d no obtendremos la parte decimal.

Cuando dividimos dos enteros, si queremos saber cual es el resto (o módulo) usamos el operador %, que vemos más abajo.

2.8.2.5 Operador (%) : Resto

Si con el anterior operador obteníamos el módulo o cociente de una división entera con éste podemos tener el resto. No funciona más que con enteros, no vale para números float o double.

Cómo se usa:

```
#include <stdio.h>
int main()
{
        int a, b;
        a = 18;
        b = 5;
        printf( "Resto de la división: %d \n", a % b );
}
```

2.8.3 Operadores de Incremento y Decremento

2.8.3.1 Operador (++) : Incremento

Este operador equivale a sumar uno a la variable:

```
#include <stdio.h>
int main()
{
        int x = 5, y = 10;
        printf ( "1er Valor de x = %i\n", x );  /*Imprime X = 5*/
        x++;
        printf ( "2do Valor de x = %i\n", x ); /*imprime X = 6*/
        printf ( "3er Valor de x = %i\n", x++ );  /*Imprime X = 6, porque primero imprime y
luego incrementa el valor de X*/

        printf ( "1er Valor de y = %i\n", y );
        ++y;
        printf ( "2do Valor de y = %i\n", y );
        printf ( "3er Valor de y = %i\n", ++y ); /*Imprime Y = 12, porque primero
incrementa el valor de Y, y luego recien imprime*/
}
```

Resultado:

```
1er Valor de x = 5
2do Valor de x = 6
3er Valor de x = 6
1er Valor de y = 10
2do Valor de y = 11
3er Valor de y = 12
```

Se puede poner antes o después de la variable.

2.8.3.2 Operador (--) : Decremento

Es equivalente a ++ pero en vez de incrementar disminuye el valor de la variable. Equivale a restar uno a la variable.

2.8.4 Relacionales o de comparación

Este es un apartado especialmente importante para todas aquellas personas sin experiencia en programación. Una característica imprescindible de cualquier lenguaje de programación es la de considerar alternativas, esto es, la de proceder de un modo u otro según se cumplan o no ciertas condiciones. Los operadores relacionales permiten estudiar si se cumplen o no esas condiciones. Así pues, estos operadores producen un resultado u otro según se cumplan o no algunas condiciones que se verán a continuación.

En el lenguaje natural, existen varias palabras o formas de indicar si se cumple o no una determinada condición. En inglés estas formas son (yes, no), (on, off), (true, false), etc. En Informática se ha hecho bastante general el utilizar la última de las formas citadas: (true, false). Si una condición se cumple, el resultado es true; en caso contrario, el resultado es false.

En C un 0 representa la condición de false, y cualquier número distinto de 0 equivale a la condición true. Cuando el resultado de una expresión es true y hay que asignar un valor concreto distinto de cero, por defecto se toma un valor unidad. Los operadores relacionales de C son los siguientes:

OPERADOR	DESCRIPCIÓN
<	Menor que.
>	Mayor que.
<=	Menor o igual.
>=	Mayor o igual
= =	Igual
! =	Distinto

Todos los operadores relacionales son operadores binarios (tienen dos operandos), y su forma general es la siguiente:

expresion1 **op** expresion2

donde op es uno de los operadores (==, <, >, <=, >=, !=). El funcionamiento de estos operadores es el siguiente: se evalúan expresion1 y expresion2, y se comparan los valores resultantes. Si la condición representada por el operador relacional se cumple, el resultado es 1; si la condición no se cumple, el resultado es 0.

A continuación se incluyen algunos ejemplos de estos operadores aplicados a constantes:

```
(2==1) /*resultado=0 porque la condición no se cumple*/
(3<=3) /* resultado=1 porque la condición se cumple*/
(3<3)  /* resultado=0 porque la condición no se cumple*/
(1!=1) /* resultado=0 porque la condición no se cumple*/
```

Veremos la aplicación de estos operadores en el capítulo Sentencias. Pero ahora vamos a ver un 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 );
}
```

No sólo se pueden comparar constantes, también se pueden comparar variables.

2.8.5 Logicos

Los operadores lógicos son operadores binarios que permiten combinar los resultados de los operadores relacionales, comprobando que se cumplen simultáneamente varias condiciones, que se cumple una u otra, etc.

OPERADOR	DESCRIPCIÓN
&&	Y (AND)
||	O (OR)
!	NO (NOT)

El operador && devuelve un 1 si tanto expresion1 como expresion2 son verdaderas (o distintas de 0), y 0 en caso contrario, es decir si una de las dos expresiones o las dos son falsas (iguales a 0).

El operador || devuelve 1 si al menos una de las expresiones es cierta. Es importante tener en cuenta que los compiladores de C tratan de optimizar la ejecución de estas expresiones, lo cual puede tener a veces efectos no deseados.

Operador ! (NOT, negación): Si la condición se cumple NOT hace que no se cumpla y viceversa.

Por ejemplo: para que el resultado del operador && sea verdadero, ambas expresiones tienen que ser verdaderas; si se evalúa expresion1 y es falsa, ya no hace falta evaluar expresion2, y de hecho no se evalúa. Algo parecido pasa con el operador ||: si expresion1 es verdadera, ya no hace falta evaluar expresion2.

Los operadores && y || se pueden combinar entre sí –quizás agrupados entre paréntesis–, dando a veces un código de más difícil interpretación. Por ejemplo:

```
(2==1) || (-1==-1)      /* el resultado es 1*/
(2==2) && (3==-1)       /* el resultado es 0*/
((2==2) && (3==3)) || (4==0)          /* el resultado es 1*/
((6==6) || (8==0)) && ((5==5) && (3==2))  /* el resultado es 0*/
printf( "Resultado: %i", (10==10 && 5>2 );
```

2.8.6.1 Introducción a los bits y bytes

Supongo que todo el mundo sabe lo que son los bytes y los bits, pero por si acaso allá va. 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.

2.8.7 A Nivel De Bits

Ya hemos visto que un byte son ocho bits. Estos operadores son la herramienta más potente, pueden manipular internamente las variables, es decir bit a bit. Este tipo de operadores solo se pueden aplicar a las variables de tipo char, short, int y long. Para manejar los bits debemos conocer perfectamente el tamaño de las variables.

Los diferentes operadores de bits son:

OPERADOR	DESCRIPCIÓN
&	Y (AND) bit a bit.
|	O (OR) Inclusiva.
^	O (OR) Exclusiva.
<<	Desplazamiento a la izquierda.
>>	Desplazamiento a la derecha.
~	Intercambia 0 por 1 y viceversa.

2.8.7.1 Operador | (OR)

Toma los valores y hace con ellos la operación OR. Vamos a ver un ejemplo:

```
#include <stdio.h>
int main()
{
        printf( "El resultado de la operacion 235 | 143 es: %i\n", 235 | 143 );
}
```

Se obtiene:

```
El resultado de la operacion 235 | 143 es: 239
```

Veamos la operación a nivel de bits:

```
235 -> 11101011
143 -> 10001111 |
239 -> 11101111
```

La operación OR funciona de la siguiente manera: Tomamos los bits dos a dos y los comparamos si alguno de ellos es uno, se obtiene un uno. Si ambos son cero el resultado es cero. Primero se compara los dos primeros (el primero de cada uno de los números, 1 y 1 -> 1), luego la segunda pareja (1 y 0 -> 1) y así sucesivamente.

2.8.7.2 Operador & (AND)

Este operador compara los bits también dos a dos. Si ambos son 1 el resultado es 1. Si no, el resultado es cero.

```
#include <stdio.h>
int main()
{
        printf( "El resultado de la operación 170 & 155 es: %i\n", 170 & 155 );
}
```

Tenemos:

```
El resultado de la operación 170 & 155 es: 138
```

A nivel de bits:

```
170 -> 10101010
155 -> 10011011 &
138 -> 10001010
```

2.8.7.3 Operador ^ (XOR)

Compara los bits y los pone a unos si son distintos.

```
235 -> 11101011
143 -> 10001111 ^
100 -> 01100100
```

2.8.7.4 Operador ~ (Complemento a uno)

Este operador acepta un sólo dato (operando) y pone a 0 los 1 y a 1 los 0, es decir los invierte. Se pone delante del operando.

```
#include <stdio.h>
int main()
{
        printf( "El resultado de la operación ~152 es: %i\n", ~152 );
}
```

Tenemos:

```
El resultado de la operación ~152 es: 103
```

A nivel de bits:

```
152 -> 10011000 ~
103 -> 01100111
```

2.8.7.5 Operador >> (Desplazamiento a la derecha)

Este operador mueve cada bit a la derecha. El bit de la izquierda se pone a cero, el de la derecha se pierde. Si realizamos la operación inversa no recuperamos el número original. El formato es:

```
variable o dato >> número de posiciones a desplazar
```

El número de posiciones a desplazar indica cuantas veces hay que mover los bits hacia la derecha. Ejemplo:

```
#include <stdio.h>
int main()
{
        printf( "El resultado de la operación 150 >> 2 es: %i\n", ~152 );
}
```

Tenemos:

```
El resultado de la operación 150 >> 2 es: 37
```

Veamos la operación paso a paso. Esta operación equivale a hacer dos desplazamientos a la derecha:

```
150 -> 10010110  Número original
75 -> 01001011 Primer desplazamiento. Entra un cero por la izquierda,
                el cero de la derecha se pierde y los demás se mueven a la derecha.
37 -> 00100101 Segundo desplazamiento.
```

NOTA: Un desplazamiento a la izquierda equivale a dividir por dos. Esto es muy interesante porque el desplazamiento es más rápido que la división. Si queremos optimizar un programa esta es una buena idea. Sólo sirve para dividir entre dos. Si hacemos dos desplazamientos sería dividir por dos dos veces, no por tres.

2.8.7.6 Operador << (Desplazamiento a la izquierda)

Funciona igual que la anterior pero los bits se desplazan a la izquierda. Esta operación equivale a multiplicar por 2.

```
#include <conio.h>
#include <stdio.h>
int  main()
{
```

```
        int valor, offset,  resultado;
        puts("Intro valor:");
        scanf("%d", &valor);
        puts("Intro ofset, en decimal");
        scanf("%d", &offset);
        resultado = valor<<offset; //equivale a resultado *= offset;
        printf("\nEl desplazamiento hacia la IZQUIERDA, genera: %d", resultado);
        resultado = valor >> offset;  //equivale a resultado /= offset;
        printf("\nEl desplazamiento hacia la DERECHA, genera: %d", resultado);
        getch();
}
```

2.8.7.7 Operador Sizeof

Este es un operador muy útil. Nos permite conocer el tamaño en bytes de una variable. De esta manera no tenemos que preocuparnos en recordar o calcular cuanto ocupa. Además el tamaño de una variable cambia de un compilador a otro, es la mejor forma de asegurarse, por lo que es necesario disponer de este operador para producir código portable. Se usa poniendo el nombre de la variable después de sizeof y separado de un espacio:

```
#include <stdio.h>
int main()
{
        int variable;
        printf( "Tamaño de la variable: %i\n", sizeof variable );
}
```

Tambíen se puede usar con los especificadores de tipos de datos (char, int, float, double...). Pero en éstos se usa de manera diferente, hay que poner el especificador entre paréntesis:

```
#include <stdio.h>
int main()
{
        printf( "Las variables tipo int ocupan: %i\n", sizeof (int) );
}
```

2.8.7.8 Orden de evaluación de Operadores

El resultado de una expresión depende del orden en que se ejecutan las operaciones. El siguiente ejemplo ilustra claramente la importancia del orden. Considérese la expresión:

```
3 + 4 * 2
```

Si se realiza primero la suma (3+4) y después el producto (7*2), el resultado es 14; si se realiza primero el producto (4*2) y luego la suma (3+8), el resultado es 11. Con objeto de que el resultado de cada expresión quede claro e inequívoco, es necesario definir las reglas que definen el orden con el que se ejecutan las expresiones de C. Existe dos tipos de reglas para determinar este orden de evaluación: las reglas de precedencia y de asociatividad. Además, el orden de evaluación puede modificarse por medio de paréntesis, pues siempre se realizan primero las operaciones encerradas en los paréntesis más interiores. Los distintos operadores de C se ordenan según su distinta precedencia o prioridad; para operadores de la misma precedencia o prioridad, en algunos el orden de ejecución es de izquierda a derecha, y otros de derecha a izquierda (se dice que se asocian de izda a dcha, o de dcha a izda). A este orden se le llama asociatividad. En la siguiente tabla se muestra la precedencia –disminuyendo de arriba a abajo– y la asociatividad de los operadores de C. En dicha Tabla se incluyen también algunos operadores que no han sido vistos hasta ahora.

Precedencia	Asociatividad
() [] -> .	izda a dcha
! ~ +(unario) -(unario) ++ -- *(indir.) &(dirección) sizeof (tipo)	dcha a izda
* / %	izda a dcha
+ -	izda a dcha
<< >>	izda a dcha
< <= > >=	izda a dcha
== !=	izda a dcha
&	izda a dcha
^	izda a dcha
|	izda a dcha
&&	izda a dcha
||	izda a dcha
?:	izda a dcha
= *= /= %= += -= &= ^= |= <<= >>=	dcha a izda
,	(operador coma) izda a dcha

Por ejemplo imaginemos que tenemos la siguiente operación:

```
10 * 2 + 5
```

Si vamos a la tabla de precedencias vemos que el * tiene un orden superior al +, por lo tanto primero se hace el producto 10*2=20 y luego la suma 20+5=25. Veamos otra:

```
10 * ( 2 + 5 )
```

Ahora con el paréntesis cambia el orden de evaluación. El que tiene mayor precedencia ahora es el paréntesis, se ejecuta primero. Como dentro del paréntesis sólo hay una suma se evalúa sin más, 2+5=7. Ya solo queda la multiplicación 10*7=70. Otro caso:

```
10 * ( 5 * 2 +3 )
```

Como antes, el que mayor precedencia tiene es el paréntesis, se evalúa primero. Dentro del paréntesis tenemos producto y suma. Como sabemos ya se evalúa primero el producto, 5*2=10. Seguimos en el paréntesis, nos queda la suma 10+3=13. Hemos acabado con el paréntesis, ahora al resto de la expresión. Cogemos la multiplicación que queda: 10*13=130.

Otro detalle que debemos cuidar son los operadores ++ y --. Estos tienen mayor precedencia que las demás operaciones aritméticas (+, -, *, /, %). Por ejemplo:

```
10 * 5 ++
```

Puede parecer que primero se ejecutará la multiplicación y luego el ++. Pero si vamos a la tabla de precedencias vemos que el ++ está por encima de * (de multiplicación), por lo tanto se evaluará primero 5++=6. 10*6=60.

Es mejor no usar los operadores ++ y -- mezclados con otros, pues a veces obtenemos resultados inesperados. Por ejemplo:

```
#include <stdio.h>
int main()
{
        int a, b;
        a = 5;
        b = a++;
        printf( "a = %i, b = %i\n", a, b );
```

```
}
```

Este ejemplo en unos compiladores dará a = 6, b = 5 y en otros a = 6 y b = 6.

Para asegurarse lo mejor sería separar la línea donde se usa el ++ y el =:

```
#include <stdio.h>
int main()
{
        int a, b;
        a = 5;
        a++;
        b = a;
        printf( "a = %i, b = %i\n", a, b );
}
```

2.9 Expresiones

Ya han aparecido algunos ejemplos de expresiones del lenguaje C en las secciones precedentes. Una expresión es una combinación de variables y/o constantes, y operadores. La expresión es equivalente al resultado que proporciona al aplicar sus operadores a sus operandos. Por ejemplo, 1+5 es una expresión formada por dos operandos (1 y 5) y un operador (el +); esta expresión es equivalente al valor 6, lo cual quiere decir que allí donde esta expresión aparece en el programa, en el momento de la ejecución es evaluada y sustituida por su resultado. Una expresión puede estar formada por otras expresiones más sencillas, y puede contener paréntesis de varios niveles agrupando distintos términos. En C existen distintos tipos de expresiones.

2.9.1 Expresiones Aritméticas

Están formadas por variables y/o constantes, y distintos operadores aritméticos e incrementales (+, -, *, /, %, ++, --). Como se ha dicho, también se pueden emplear paréntesis de tantos niveles como se desee, y su interpretación sigue las normas aritméticas convencionales. Por ejemplo, la solución de la ecuación de segundo grado:

$$x = \frac{-b + \sqrt{b^2 - 4ac}}{2a}$$

se escribe, en C en la forma:

```
x = (-b+sqrt((b*b)-(4*a*c)))/(2*a);
```

donde, estrictamente hablando, sólo lo que está a la derecha del operador de asignación (=) es una expresión aritmética. El conjunto de la variable que está a la izquierda del signo (=), el operador de asignación, la expresión aritmética y el carácter (;) constituyen una sentencia. En la expresión anterior aparece la llamada a la función de librería sqrt(), que tiene como valor de retorno la raíz cuadrada de su único argumento. En las expresiones se pueden introducir espacios en blanco entre operandos y operadores; por ejemplo, la expresión anterior se puede escribir también de la forma:

```
x = (-b + sqrt((b * b) - (4 * a * c)))/(2 * a);
```

2.9.2 Expresiones Lógicas

En la evaluación de expresiones lógicas, los compiladores normalmente utilizan técnicas de evaluación rápida. Para decidir si una expresión lógica es cierta (TRUE) o falsa (FALSE) muchas veces no es necesario evaluarla completamente. Por ejemplo una expresión formada <exp1> || <exp2>, el compilador evalúa

primero <exp1> y si es cierta, no evalúa <exp2>. Por ello se deben evitar construcciones en las que se modifiquen valores de datos en la propia expresión, pues su comportamiento puede depender de la implementación del compilador o de la optimización utilizada en una compilación o en otra. Estos son errores que se pueden cometer fácilmente en **C** ya que una asignación es también una expresión.

Debemos evitar:

```
if (( x++ > 3 ) || ( x < y ))
```

y escribir en su lugar:

```
x++; if (( x > 3 ) || ( x < y ))
```

En general un valor de 0 indica que la expresión es falsa y un valor distinto de 0 indica que la expresión es verdadera.

2.9.3 Expresiones Generales

Una de las características más importantes (y en ocasiones más difíciles de manejar) del C es su flexibilidad para combinar expresiones y operadores de distintos tipos en una expresión que se podría llamar general, aunque es una expresión absolutamente ordinaria de C.

Recuérdese que el resultado de una expresión lógica es siempre un valor numérico (un 1 ó un 0); esto permite que cualquier expresión lógica pueda aparecer como sub-expresión en una expresión aritmética. Recíprocamente, cualquier valor numérico puede ser considerado como un valor lógico: true si es distinto de 0 y false si es igual a 0. Esto permite introducir cualquier expresión aritmética como sub-expresión de una expresión lógica. Por ejemplo:

```
(a - b*2.0) && (c != d)
```

A su vez, el operador de asignación (=), además de introducir un nuevo valor en la variable que figura a su izda, deja también este valor disponible para ser utilizado en una expresión más general. Por ejemplo, supóngase el siguiente código que inicializa a 1 las tres variables a, b y c:

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

que equivale a:

```
a = (b = (c = 1));
```

En realidad, lo que se ha hecho ha sido lo siguiente. En primer lugar se ha asignado un valor unidad a c; el resultado de esta asignación es también un valor unidad, que está disponible para ser asignado a b; a su vez el resultado de esta segunda asignación vuelve a quedar disponible y se puede asignar a la variable a.

2.10 Ejercicios

Ejercicio 1. Busque los errores:

```
#include <stdio.h>
int main()
{
        int número;
        número = 2;
}
```

Solución: Los nombres de variables no pueden llevar acentos, luego al compilar número dará error.

Ejercicio 2. Busque los errores:

```
#include <stdio.h>
int main()
{
        int numero;
        numero = 2;
        printf( "El valor es %i" Numero );
}
```

Solución: Falta la coma después de "El valor es %i". Además la segunda vez numero está escrito con mayúsculas.

Ejercicio 3: En este programa hay un fallo muy gordo y muy habitual en programación. A ver si lo encuentras:

```
#include <stdio.h>
int main()
{
        int a, c;
        a = 5;
        c += a +5;
}
```

Solución:

Cuando calculamos el valor de 'c' sumamos a+5 (=10) al valor de 'c'. Pero resulta que 'c' no tenía ningún valor indicado por nosotros. Estamos usando la variable 'c' sin haberle dado valor. En algunos compiladores el resultado será inesperado. Este es un fallo bastante habitual, usar variables a las que no hemos dado ningún valor.

Ejercicio 4: ¿Cual será el resultado del siguiente programa?

```
#include <conio.h>
#include <stdio.h>
int main()
{
        int a, b, c;

        a = 5;
        b = ++a;
        c = ( a + 5 * 2 ) * ( b + 6 / 2 ) + ( a * 2 );
        printf( "%i, %i, %i", a, b, c );
}
```

Solución:
El resultado es 156. En la primera a vale 5. Pero en la segunda se ejecuta b = ++a = ++5 = 6. Tenemos a = b = 6.

CAPITULO III:

Sentencias de control de flujo

3.1 Introduccion

Hasta ahora los programas que hemos visto eran lineales. Comenzaban por la primera instrucción y acababan por la última, ejecutándose todas una sola vez. . Lógico ¿no?. Pero resulta que muchas veces no es esto lo que queremos que ocurra. Lo que nos suele interesar es que dependiendo de los valores de los datos se ejecuten unas instrucciones y no otras. O también puede que queramos repetir unas instrucciones un número determinado de veces. Para esto están las sentencia de control de flujo.

3.2 Sentencias (Statements)

Las expresiones de C son unidades o componentes elementales de unas entidades de rango superior que son las sentencias. Las sentencias son unidades completas, ejecutables en sí mismas. Ya se verá que muchos tipos de sentencias incorporan expresiones aritméticas, lógicas o generales como componentes de dichas sentencias.

3.2.1 Sentencias Simples

Una sentencia simple es una expresión de algún tipo terminada con un carácter (;). Un caso típico son las declaraciones o las sentencias aritméticas. Por ejemplo:

```
sentencia_simple;
float real;
espacio = espacio_inicial + velocidad * tiempo;
x = y * 5 + sqrt(z); /* sentencia simple */
```

3.2.2 Sentencia Vacía o Nula

En algunas ocasiones es necesario introducir en el programa una sentencia que ocupe un lugar, pero que no realice ninguna tarea. A esta sentencia se le denomina sentencia vacía y consta de un simple carácter (;). Por ejemplo:

```
;
```

3.2.3 Sentencias Compuestas O Bloques

Muchas veces es necesario poner varias sentencias en un lugar del programa donde debería haber una sola. Esto se realiza por medio de sentencias compuestas. Una sentencia compuesta es un conjunto de declaraciones y de sentencias agrupadas dentro de llaves {...}. También se conocen con el nombre de bloques. Una sentencia compuesta puede incluir otras sentencias, simples y compuestas.

Ejemplos:

```
/* sentencia compuesta con llaves */
{
        a = b;
        b = x + 1;
```

```
        printf ( "hay %d productos", num_prod );
}

/* sentencias compuestas dentro de otras */
{
        {
                x=1;
                y=2;
        }
        {
                z=0;    printf("hola\n");
        }
}
```

3.3 Estructuras Condicionales

3.3.1 Condicionales Simples

3.3.1.1 Sentencia if

La construcción **if** sirve para ejecutar código sólo si una condición es cierta:

```
if ( condición )
        sentencia(s);
```

La **condición** es una expresión de cualquier clase.

- Si el resultado de la expresión es CERO, se considera una condición FALSA.
- Si el resultado de la expresión NO ES CERO, se considera una condición CIERTA.

Cuando el cuerpo del **if** está formado por una sola instrucción puede o no ir entre llaves, si existe más de una instrucción entonces si lleva obligatoriamente las llaves ({}).

Ejemplo: Determine el mayor de dos números.

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

int main (void)
{
        int numero1, numero2;
        printf("Introduzca el primer numero: ");
        scanf("%d",&numero1);
        printf("Introduzca el segundo numero: ");
        scanf("%d",&numero2);
        if (numero1 > numero2)
                printf("El mayor es %d", numero1);
        if (numero1 < numero2)
                printf("El mayor es %d", numero2);
        if (numero1 == numero2)
                printf("Los numeros son iguales");
        getch();
        return 0;
}
```

3.3.1.2 Construcción if - else

Con la construcción **if - else** se pueden definir acciones para cuando la condición del **if** sea falsa.
La sintaxis es:

```
if ( condición )
        sentencia
else
        sentencia
```

Ejemplo 1: Ingresar por teclado un numero y decir si es positivo o negativo. Considere que el cero es un número positivo.

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

int main (void)
{
        int numero1;
        printf("Introduzca un numero: ");
        scanf("%d",&numero1);
        if (numero1 >= 0)
                printf("El numero %d es positivo\n", numero1);
        else
                printf("El numero %d es negativo\n", numero1);
        getch();
        return 0;
}
```

Ejemplo 2: Realice un programa que pida dos dumeros y los divida. El número por el que dividimos no puede ser cero, esto nos daría un valor de infinito, provocando un error en el ordenador. Por tanto antes de dividir deberíamos de comprobar si el divisor es cero. El programa sería algo como esto:

```
#include <stdio.h>
#include <conio.h>
int main (void)
{
        float dividendo,divisor;
        printf ("\nDime el dividendo: ");
        scanf ("%f",&dividendo);
        printf ("\nDime el divisor: ");
        scanf ("%f",&divisor);
        if (divisor==0)
                printf ("\nNo podemos dividir un número por 0");
        else
                printf ("\nEl resultado es: %f", dividendo/divisor);
        getch();
        return 0;
}
```

Ejemplo 3: Simula una clave de acceso

```
#include <stdio.h>
#include <conio.h>
int main (void)
{
	int usuario, clave=234;
	printf("Introduce tu clave: ");
	scanf("%d",&usuario);
	if (usuario==clave)
		printf("Acceso permitido\n");
	else
		printf("Acceso denegado\n");
	getch();
	return 0;
}
```

3.3.1.3 Construcción if – else if - ...

La sintaxis es:

```
if (condición)
	sentencia1;
else
	if (condición)
		sentencia2;
	else
		if (condición)
			sentencia3;
		else
			sentencia4;
```

Con este formato el flujo del programa únicamente entra en una de las condiciones. Si una de ellas se cumple, se ejecuta la sentencia correspondiente y salta hasta el final de la estructura para continuar con el programa.

Existe la posibilidad de utilizar llaves para ejecutar más de una sentencia dentro de la misma condición.

Ejemplo 4: Escribe bebé, niño, adolescente o adulto

```
#include <stdio.h>
#include <conio.h>
int main (void)
{

	int edad;
	printf("Introduce tu edad: ");
	scanf("%d",&edad);
	if (edad<1)
		printf("Lo siento, te has equivocado.");
	else if (edad<3)
		printf("Eres un bebé");
	else if (edad<13)
		printf("Eres un niño");
	else if (edad<18)
		printf("Eres un adolescente");
```

```
        else
                printf("Eres adulto");
        getch();
        return 0;
}
```

3.3.1.4 Sentencias if Anidadas

Una sentencia if puede incluir otros if dentro de la parte correspondiente a su sentencia, A estas sentencias se les llama sentencias anidadas (una dentro de otra), por ejemplo,

```
if (a >= b)
        if (b != 0.0)
                c = a/b;
```

En ocasiones pueden aparecer dificultades de interpretación con sentencias if...else anidadas, como en el caso siguiente:

```
if (a >= b)
        if (b != 0.0)
                c = a/b;
        else
                c = 0.0;
```

En principio se podría plantear la duda de a cuál de los dos if corresponde la parte else del programa. Los espacios en blanco –las indentaciones de las líneas– parecen indicar que la sentencia que sigue a else corresponde al segundo de los if, y así es en realidad, pues la regla es que el else pertenece al if más cercano. Sin embargo, no se olvide que el compilador de C no considera los espacios en blanco (aunque sea muy conveniente introducirlos para hacer más claro y legible el programa), y que si se quisiera que el else perteneciera al primero de los if no bastaría cambiar los espacios en blanco, sino que habría que utilizar llaves, en la forma:

```
if (a >= b)
{
        if (b != 0.0)
                c = a/b;
}
else
        c = 0.0;
```

Recuérdese que todas las sentencias if e if...else, equivalen a una única sentencia por la posición que ocupan en el programa.

3.3.1.5 El Operador Condicional

C ofrece una forma abreviada de expresar la sentencia if - else se denomina expresión condicional y emplea el operador condicional (? :), la sintaxis es:

```
Variable = (condicion) ? valor si expresión es verdadera : valor si expresión es falsa
```

Ejemplo: Programa que pide un numero, y muestra su valor absoluto

```
#include <stdio.h>
#include <conio.h>
int main (void)
```

```
{

        int num, x;
        printf("Introduce un numero: ");
        scanf("%d",&num);

        x = (num<0)? num * -1 : num;

        printf("El valor absoluto de %d es: %d\n", num, x);
        getch();
        return 0;
}
```

En el ejemplo anterior la variable x toma el valor de "num * -1" si el numero introducido (num) es menor que cero o toma el valor "num" si la condición es falsa.

Como se puede observar se trata de una secuencia if else pero muy concreta, probablemente el compilador generará un código mucho más eficiente para este tipo de secuencia de ahí su inclusión en el juego de operadores del **C**.

3.3.2 Estructura Condicional Multiple

3.3.2.1 Construcción switch

Supongamos un programa que entre la nota de un alumno por el teclado y nos diga su calificación. Podríamos utilizar estructuras if de la manera siguiente:

```
Leer(Nota);
Si (Nota = 0) Entonces
        Escribir("Muy Deficiente");
Si (Nota = 1) Entonces
        Escribir("Muy Deficiente");
.
.
Si (Nota = 5) Entonces
        Escribir("Suficiente");
.
.
Si (Nota = 8) Entonces
        Escribir("Notable");
.
.
Si (Nota = 10) Entonces
        Escribir("Matrícula de Honor");
```

Este programa funcionaria correctamente. Pero, si analizamos detenidamente el proceso, nos damos cuenta que una vez cumplida una condición, las otras comparaciones son innecesarias. Por ejemplo, si Nota tiene el valor 5, a partir de la condición Si (Nota = 5) el resto de condiciones seguro que no se cumplirán. Este programa sin embargo, las evaluará con el consecuente desperdicio de tiempo. Dicho de otra manera, el programa funciona pero no está bien acabado, no está resuelto de manera elegante. Podríamos llegar a la determinación de utilizar estructures if..else anidadas, con lo cual conseguiríamos que el programa no evaluase condiciones innecesarias, y entonces nos quedaría este código:

```
Leer (Nota)
```

```
Si (Nota = 0) Entonces
        Escribir("Muy Deficiente");
Sino
        Si (Nota = 1) Entonces
                Escribir("Muy Deficiente");
        Sino
                Si (Nota = 2) Entonces
                        Escribir ("Deficiente");
                Sino
                                .
                                .

                                        Si (Nota = 5) Entonces
                                                Escribir("Suficiente");
                                        Sino
                                                .
                                                .
                                                .
                                                        Si(Nota = 10) Entonces
                                                                Escribir("Matrícula de Honor");
```

Eficiente, pero tal vez poco elegante y realmente muy liado.

En la mayoría de lenguajes de programación hay una estructura que permite evaluar una misma variable y hacer diferentes acciones según los valores que esta vaya cogiendo. Esta es la estructura Caso que:

```
Caso que (Variable)
        Inicio
                Valor 1:
                        Acción1.1;
                        Acciόm1.2;
                        .
                        Acción1.N;
                Valor 2:
                        Acción2.1;
                        Acción2.2;
                        .
                        Acción2.N;
                Valor N:
                        AcciónN.1;
                        AcciónN.2;
                        .
                        AcciónN.N
                Sino:
                        Acciones correspondientes en caso que no sea
                        ninguno de los valores anteriores;
        Fin;
```

Así el problema anterior se solucionaría,

```
Leer(Nota);
Caso que (Nota)
        Inicio
                0:Escribir('Muy Deficiente');
                1:Escribir('Muy Deficiente');
                2:Escribir('Deficientr');
                .
                5:Escribir('Suficiente');
                .
                .
                10:Escribir('Matrícula de honor');
                Sino
        Escribir('No se puede evaluar este valor');
        Fin;
```

Con esta estructura el programa evalúa los diferentes valores; cuando encuentra el que tiene la variable Nota, ejecuta las acciones correspondientes, y salta a la sentencia siguiente, a Fin. La sentencia Sino es opcional, y sirve para hacer una determinada acción, siempre y cuando la variable que se evalúe, no tenga ninguno de los valores.

En lenguaje C la sentencia correspondiente es **switch..case**,

La instrucción de selección o conmutación de clave **switch** es una instrucción de selección múltiple que permite efectuar un desvío directo hacia las acciones según el resultado obtenido, ésta expresión al ser evaluada debe proporcionar como resultado un valor entero. La sintaxis es:

La estructura **switch** comienza con la palabra reservada **switch** seguida de una expresión entre paréntesis. Luego de esto vienen las etiquetas de selección a través de la palabra reservada **case** ésta palabra debe tener como argumento obligatoriamente constantes enteras sea bajo forma numérica o simbólica.

En varios de los casos puede hacerse referencia a una misma acción para ello se disponen en secuencia y la última cláusula **case** es la que hará referencia a la secuencia de instrucciones asociada.

Por lo común la ultima instrucción antes de la siguiente etiqueta es la instrucción **break**, ésta palabra reservada provoca el abandono de la estructura **switch**. Si no existe una proposición **break** la ejecución continua con las instrucciones de la siguiente etiqueta. La ausencia de instrucciones **break** es una causa de error frecuente en un **switch**.

Por último puede haber una instrucción etiquetada como **default** y representa el caso en el que el valor de la expresión no coincida con ningún valor de las etiquetas utilizadas. No es necesario incluir la sentencia **default**.

La estructura **Caso..que** es la más indicada cuando se han de ejecutar diferentes accione que dependen del valor que tiene una única variable. Es la estructura más utilizada para bifurcar acciones en programas que utilizan menús de opciones.

La sintaxis es:

```
switch ( expresión )
{
        case valor1:
```

```
            SentenciasA
            break;

        case valor2:
            SentenciasB
            break;

        case valor3:
        case valor4:
            SentenciasC
                break;

        default:
            SentenciasD
}
```

Las SentenciasA se ejecutarán si ***expresión*** adquiere el ***valor1***.
Las SentenciasB se ejecutarán si adquiere el ***valor2***.
Las SentenciasC se ejecutarán si adquiere el ***valor3*** o el ***valor4***, indistintamente.
Cualquier otro valor de ***expresión*** conduce a la ejecución de las SentenciasD; eso viene indicado por la palabra reservada **default**.

Volviendo a nuestro ejemplo anterior sobre las notas el codigo seria el siguiente:

```
#include <stdio.h>
int main()
{
    int Nota;
    printf("Entre la nota: ");
    scanf("%d",&Nota);

    switch (Nota)
    {
        case 0:
        case 1:
        case 2:
        case 3:
        case 4:
            printf("Muy deficiente");
            break;
        case 5:
        case 6:
        case 7:
        case 8:
            printf("Suficiente");
            break;
        case 10:
            printf("Matrícula de honor");
            break;
        default:
            printf("No se puede evaluar este valor");
    }
}
```

Ejemplo 1:

```
#include <stdio.h>
#include <conio.h>
int main (void)
{
        char c;
        printf ("Escriba una letra: ");
        scanf ("%c", &c );
        switch(c)
        {
                case 'a':
                        printf("Op A\n");
                        break;
                case 'b':
                        printf("Op B\n");
                        break;
                case 'c':
                case 'd':
                        printf("Op C o D\n");
                        break;
                default:
                        printf("Op ?\n");
        }
        getch();
        return 0;
}
```

Ejemplo 2: Escribe un día de la semana

```
#include <stdio.h>
#include <conio.h>
int main (void)
{
        int dia;
        printf("Introduce el día (1 - 7): ");
        scanf("%d",&dia);
        switch(dia)
        {
                case 1:
                        printf("Lunes");        break;
                case 2:
                        printf("Martes");       break;
                case 3:
                        printf("Miércoles");break;
                case 4:
                        printf("Jueves");       break;
                case 5:
                        printf("Viernes");      break;
                case 6:
                        printf("Sábado");       break;
                case 7:
                        printf("Domingo");    break;
```

```
        }
}
```

3.4 Bucles

Supongamos que tenemos que hacer un programa para entrar las notas de una clase de 5 alumnos y después hacer la media; con las estructuras vistas hasta ahora, podriamos hacer:

```
Programa Tal;

Variables
        Nota :Entero ;
        Media :Real ;

Inicio
        Media := 0 ;
        Escribir("Entrar la nota : ") ;
        Leer(Nota) ;
        Media := Media + Nota ;

        Escribir("Entrar la nota : ") ;
        Leer(Nota) ;
        Media := Media + Nota ;

        Escribir("Entrar la nota : ") ;
        Leer(Nota) ;
        Media := Media + Nota ;

        Escribir("Entrar la nota : ") ;
        Leer(Nota) ;
        Media := Media + Nota ;

        Escribir("Entrar la nota : ") ;
        Leer(Nota) ;
        Media := Media + Nota ;

        Media := Media / 5 ;

        Escribir("La media de la clase es : ",Media) ;

Fin.
```

Fijaros que este programa repite el bloque de sentencias,

```
Escribir("Entrar la nota : ") ;
Leer(Nota) ;
Media := Media + Nota ;
```

5 veces. Para evitar esta tipo de repeticiones de código, los lenguajes de programación incorporan instrucciones que permiten la repetición de bloques de código.

Un bucle es una secuencia de instrucciones que se repite un número determinado de veces o hasta que se cumplan unas determinadas condiciones.

Los bucles son extremadamente útiles en nuestros programas, algunos ejemplos son:

- Lectura/Visualización de un número determinado de datos, como por ejemplo una matriz.
- A veces se hace necesario introducir esperas en nuestros programas ya sea por trabajar con un periférico lento o simplemente por ralentizar su ejecución. Los primeros se llaman bucles de espera activa y los segundos bucles vacíos.
- En aplicaciones gráficas como trazado de líneas o rellenado de polígonos.
- Lectura de datos de un fichero...

A continuación describiremos las opciones que nos proporciona el lenguaje de programación **C** para crear y gestionar los bucles.

3.4.1 while

Permite repetir una serie de acciones mientras se verifica una condición la sintaxis es:

```
while ( expresión )
{
        instrucciones
}
instrucción siguiente
```

Primero se evalúa la expresión si ésta es falsa el control pasa directamente a la instrucción siguiente, pero si es verdadera entonces se ejecuta el cuerpo del laso while devolviendo el control al principio del mismo.

Es evidente que una de las acciones internas del bucle while debe modificar en un momento dado una de las variables que intervienen en la expresión que condiciona el bucle para que dicha condición se haga falsa y se pueda abandonar el bucle. Si el cuerpo del laso esta formada por una sola instrucción no necesita las llaves que delimitan el cuerpo del bucle, pero si tiene más de una instrucción entonces las llaves son obligatorias.

Los bucle while son útiles en aplicaciones como; lee datos de este fichero mientras no llegues al final ó muestra estos datos en la pantalla mientras no pulse una tecla.

Una peculiaridad de esta instrucción es que puede no ejecutarse nunca la instrucción afectada por el while. Algunos ejemplos:

Ejemplo 1: Imprimir los numero desde el caro hasta el diez.

```
#include <stdio.h>
#include <conio.h>
int main(void)
{
        int i=0;
        while (i<=10)
        {
                printf ("\n%d",i);
                i++;
        }
        printf("\n%d\n",i);
        getch();
```

```
        return 0;
}
```

El primer valor que visualizaremos será el 0. Cuando i tenga el valor diez la condición i<=10 todavía se cumple por lo que entraremos en el bucle de nuevo, visualizaremos el nueve e incrementamos i con lo que pasa a tener el valor 11 momento en el cual se vuelve a evaluar la expresión i<=10 que en este caso sería falsa y no volveríamos a entrar en el bloque de instrucciones (las que están entre llaves).
La instrucción que sigue después de todo el bucle wuile visualiza el valor de i antes de abandonar el programa, el cual es 11.

Ejemplo 2: Lee números enteros hasta que se introduzca el valor hasta que se introduzca el valor 0.

```
#include <stdio.h>
#include <conio.h>
int main(void)
{
        int numero = 10;
        while (numero!=0)
        {
                printf ("\nDime un numero: ");
                scanf ("%d",&numero);
        }
        getch();
        return 0;
}
```

En este ejemplo tenemos que introducir en la variable número un valor distinto de cero antes de entrar en el bucle, puesto que en principio al declarar una variable el valor de ésta no está determinado y podría valer cero, en cuyo caso nunca se ejecutaría el bucle. Esta es la misión de la instrucción numero = 10;

Ejemplo usando el operador de predecremento:

```
#include <stdio.h>
#include <conio.h>
int main (void)
{
        int x=10;
        while ( --x )
        {
                printf("una línea\n");
        }
        getch();
        return 0;
}
```

En cada iteración se decrementa la variable **x** y se comprueba el valor devuelto por **--x**. Cuando esta expresión devuelva un cero, se abandonará el bucle. Esto ocurre después de la iteración en la que **x** vale uno.

3.4.2 do while

El laso do es muy similar al laso while solo que while verifica la expresión y luego ejecuta el cuerpo del laso en cambio el laso do ejecuta el cuerpo del laso y luego verifica la expresión. Por lo tanto el cuerpo del laso do se ejecuta al menos una vez, su sintaxis es:

```
do
```

```
{
        instrucciones
}       while (expresion)
instrucción siguiente
```

Se recomienda no utilizar esta construcción, porque las construcciones **while** y **for** bastan para diseñar cualquier clase de bucles. Muy pocos programas hoy día usan esta construcción.

Ejemplo 1: Muestra un menú si no se pulsa 4

```
#include <stdio.h>
#include <conio.h>
int main (void)
{
        char seleccion;
        do
        {
                printf("1.- Comenzar\n");
                printf("2.- Abrir\n");
                printf("3.- Grabar\n");
                printf("4.- Salir\n");
                printf("Escoge una opción: ");
                seleccion=getchar();
                switch(seleccion)
                {
                        case '1': printf("\nOpción 1\n"); break;
                        case '2': printf("\nOpción 2\n"); break;
                        case '3': printf("\nOpción 3\n"); break;
                }
        }       while(seleccion!='4');
        getch();
        return 0;
}
```

Ejemplo 2: Programa que pide un numero y suma desde el hasta el cero.

```
#include <stdio.h>
#include <stdlib.h>
void main(void)
{
        int numero, acumulador=0, contador=0;
        printf("Intro un numero positivo para sumar desde el a cero:\n");
        scanf("%d", &numero);
        do
        {
                acumulador+= numero;
                contador++;
                numero--;
        }       while(numero>0);
        printf("\nLa suma  es %d\n", acumulador);
        printf("\nEl bucle se ha ejecutado %d veces.", contador);
        system("PAUSE");
}
```

3.4.3 Bucle for

Esta estructura sirve para repetir la ejecución de una sentencia o bloque de sentencias, un número definido de veces. La estructura es la siguiente:

```
for ( expresión_inicial; condición; expresión_de_paso )
{
        instrucciones
}
instrucción siguiente
```

expresión_inicial es una expresión que sólo se ejecuta una vez al principio del bucle. El bucle for suele utilizarse en combinación con un contador. Un contador es una variable que lleva la cuenta de las veces que se han ejecutado las instrucciones sobre las que actúa el comando for. Por tanto expresión_inicial suele contener una expresión que nos permite inicializar ese contador generalmente a 0 aunque eso depende de para qué deseemos utilizar el bucle.

condición es la expresión que nos indica cuando debe finalizar el bucle, por tanto se tratará de una expresión condicional. Su interpretación sería algo como; repite la instrucción (o instrucciones) mientras se cumpla exp2. Esta expresión se evaluará en cada ciclo del bucle para determinar si se debe realizar una nueva iteración.

NOTA: Hay que recordar que condición se evalúa al principio del bucle, y no al final. Por tanto es posible no ejecutar el bucle NINGUNA vez.

expresión_de_paso es una expresión que se ejecuta en cada iteración.

Puesto que como ya indicamos el bucle for se utiliza junto a un contador, ***expresión_de_paso*** en general contiene una instrucción que actualiza nuestro contador.

Por tanto en un bucle con contador distinguimos tres partes diferenciadas:

- La inicialización del contador (***expresión_inicial***).
- La condición de fin de bucle (***condición***).
- Y la actualización del contador (***expresión_de_paso***).

El bucle **for** es (casi) equivalente a

```
expresión_inicial;
while ( condición )
{
        instrucciones
        expresión_de_paso
}
instrucción siguiente
```

El bucle for esta especialmente pensado para realizar bucles basados en contadores. Se puede utilizar en bucle del tipo "repite esto hasta que se pulse una tecla", pero para estos tenemos instrucciones más apropiadas. Veamos unos ejemplos que nos permitan comprender más fácilmente el funcionamiento del comando for.

Ejemplo 1: Contar hasta diez.

```
#include <stdio.h>
#include <conio.h>
int main (void)
{
	int i; /* Esta variable la utilizaremos como contador*/
	for (i=0;i<10;i++)
		printf ("\n%d",i);
	getch();
	return 0;
}
```

Este programa mostrará en pantalla números de 0 a 9 (diez en total). exp1 inicializa nuestro contador que en este caso es una variable de tipo entero, con el valor 0, exp2 nos dice que nuestra instrucción (la función printf) debe repetirse mientras el contador sea menor que diez y finalmente exp3 indica que el contador debe de incrementarse en una unidad en cada ciclo del bucle.

Nos podría interesar contar desde diez hasta 1, en este caso el comando for debería de ser:

```
#include <stdio.h>
#include <conio.h>
int main (void)
{
	int i; /* Esta variable la utilizaremos como contador*/
	for (i=10;i>0;i--)
		printf ("\n%d",i);
	getch();
	return 0;
}
```

Ejemplo 2: Visualizar dos tablas de multiplicar en pantalla.

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

int main (void)
{
	int i;
	int tabla1,tabla2;
	tabla1 = 2; /* Primero la tabla del dos */
	tabla2 = 5; /* y a continuación la tabla del cinco*/
	for (i=1;i<11;i++)
	{
		printf ("\n %2dx%2d=%3d",tabla1,i,tabla1*i);
		printf (" %2dx%2d=%3d",tabla2,i,tabla2*i);
	}
	printf("\n");
	getch();
	return 0;
}
```

3.4.3.1 For anidados

De la misma manera que vimos como las estructuras condicionales, un bucle for puede contener otros. Véa el programa siguiente desarrollado directamente en lenguage C.

Programa que muestra todos los resultados posibles que pueden salir al tirar dos dados. Combinaremos el resultado de un dado con los seis que pueden salir del otro.

```
#include <stdio.h>
#include <conio.h>
main()
{
        int dado2 ;
        for ( dado2=1 ; dado2 <=6 ; dado2++)
                printf(" 1, %d", dado2) ;
        for ( dado2=1 ; dado2 <=6 ; dado2++)
                printf(" 2, %d", dado2) ;
        for ( dado2=1 ; dado2 <=6 ; dado2++)
                printf(" 3, %d", dado2) ;
        for ( dado2=1 ; dado2 <=6 ; dado2++)
                printf(" 4, %d", dado2) ;
        for ( dado2=1 ; dado2 <=6 ; dado2++)
                printf(" 5, %d", dado2) ;
        for ( dado2=1 ; dado2 <=6 ; dado2++)
                printf(" 6, %d", dado2) ;
}
```

En este programa nos encontramos con el mismo caso que con la misma versión del programa de las notas (la que no utilizaba el bucle for): hay una senténcia que se repite 6 veces (la única diferéncia es el valor que representa el primer dado, valor del 1 al 6). Cerraremos esta senténcia dentro de un bucle for, y el programa quedará de la siguiente manera:

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

main()
{
        int dado1, dado2 ;
        for (dado1=1 ; dado1 <= 6 ; dado1++)
                for (dado2 = 1 ; dado2 <=6 ; dado2++)
                        printf("%d, %d", dado1, dado2) ;
}
```

Ejecute este programa, y comprueve que cada vez que se ejecuta el primer bucle (el controlado para la variable Dado1), el segundo se ejecuta 6 veces.

Programa que visualiza la cantidad de asteriscos que representa cada línea de la pantalla. Línea 1 un asterisco, línea 2 dos asteriscos,... línea 24 veinticuatro asceriscos,

```
#include <stdio.h>
#include <conio.h>
main()
{
        int numlinea, numasteriscos ;
        for (numlinea = 1 ; numlinea <= 24 ; numlinea++)
        {
```

```
                for ( numasteriscos = 1 ; numasteriscos <= numlinea ;  numasteriscos++)
                        putchar('*') ;
                printf("\n") ;          // Para hacer salto de línea
        }
}
```

3.4.3.2 Bucle for: Omisión De Expresiones

El for en **C** es mucho más general que en otros lenguajes no existe ninguna restricción sobre las expresiones que pueden intervenir en una instrucción de clave for. Los tres campos pueden no tener ningún vínculo entre ellos.

Cada una de las componentes del for es facultativa u optativa lo que permite efectuar las construcciones siguientes.

```
for (; ; )                          /* Laso infinito */
for (expresión_inicial; condición;)    /* No tiene expresión3 */
for (;expresión2;)                     /* Es equivalente a un while */
for (;expresión2;expresión3)  /* No existe valor inicial */
```

Se omite	**Resultado**
expresión_inicial	no se hace nada antes del bucle
condición	la condición es siempre cierta (1)
expresión_de_paso	no se hace nada tras cada iteración

Ejemplos:

```
for ( ; resultado!=-1 ; ) { ... }

for ( ; ; ) { /* Bucle infinito */ }
```

Ejemplo 1: Programa que muestra los valores entre un número introducido por el teclado, y visualiza los valores entre este valor y 10. Si el número entrado es > 10 no se visualiza nada.

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

main()
{
        int i ;
        clrscr() ;
        printf("Entrar el valor inicial : ") ;
        scanf("%d",&i) ;
        for ( ;i<=10 ;i++)
                printf("%d \n",i) ;
}
```

Ejemplo 2: Programa que muestra números del 1 al 10 con incrementos introducidos por el teclado.

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

```
#include <conio.h>

main()
{
        int i, incremento ;
        clrscr() ;
        for(i=1 ;i<=10 ;)
        {
                printf("%d \n", i) ;
                printf("Entrar incremento : ") ;
                scanf("%d",&incremento) ;
                i = i + incremento ;
        }
}
```

Ejemplo 3: Programa que introduce valores, hasta que se introduzca uno mayor que 10.

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

main()
{
        int i = 0; // Iniciada para que en la primera evaluación sea <= 10.
        clrscr() ;
        for(;i<=10 ;)
        {
                printf("Entrar un valor : ") ;
                scanf("%d",&i) ;
        }
}
```

3.5 Control de bucles

3.5.1 break

break provoca que se termine la ejecución de una iteración (salir directamente del bucle) o para salir de la sentencia switch, como ya hemos visto.

3.5.2 continue

Se utiliza dentro de un bucle. Cuando el programa llega a una sentencia CONTINUE no ejecuta las líneas de código que hay a continuación y salta a la siguiente iteración del bucle.

Ejemplo: Realice un programa que escriba los numeros del 1 al 10 menos el 5

```
#include <stdio.h>
int main (void)
{
        int numero=1;
        while(numero<=10)
        {
                if (numero==5)
```

```
            {
                numero++;
                continue;
            }
            printf("%d\n",numero);
            numero++;
        }
        getch();
        return 0;
}
```

3.6 Goto

La sentencia goto (ir a) nos permite hacer un salto a la parte del programa que deseemos. En el programa podemos poner etiquetas, estas etiquetas no se ejecutan. Es como poner un nombre a una parte del programa. Estas etiquetas son las que nos sirven para indicar a la sentencia goto dónde tiene que saltar. Es una sentencia muy mal vista en la programación en **C**.

El goto sólo se puede usar dentro de funciones, y no se puede saltar desde una función a otra. (Las funciones las estudiamos en otro capítulo).

```
#include <stdio.h>
int main()
{
        printf( "Línea 1\n" );
        goto linea3;    /* Le decimos al goto que busque la etiqueta linea3 */
        printf( "Línea 2\n" );
        linea3:                 /* Esta es la etiqueta */
        printf( "Línea 3\n" );
}
```

Resultado:

```
Línea 1
Línea 3
```

Como vemos no se ejecuta el printf de Línea 2 porque nos lo hemos saltado con el goto.

3.7 Nota sobre las condiciones

Las condiciones de las sentencias se evalúan al ejecutarse. De esta evaluación obtenemos un número. Las condiciones son falsas si este número es igual a cero. Son verdaderas si es distinto de cero (los números negativos son verdaderos).

Ahí van unos ejemplos:

```
a = 2;
b = 3;
if ( a == b ) ...
```

Aquí a==b sería igual a 0, luego falso.

```
if ( 0 ) ...
```

Como la condición es igual a cero, es falsa.

```
if ( 1 ) ...
```

Como la condición es distinta de cero, es verdadera.

```
if ( -100 ) ...
```

Como la condición es distinta de cero, es verdadera.
Supongamos que queremos mostrar un mensaje si una variable es distinta de cero:

```
if ( a!=0 ) printf( "Hola\n" );
```

Esto sería redundante, bastaría con poner:

```
if ( a ) printf( "Hola\n" );
```

Esto sólo vale si queremos comprobar que es distinto de cero. Si queremos comprobar que es igual a 3:

```
if ( a == 3 ) printf( "Es tres\n" );
```

Como vemos las condiciones no sólo están limitadas a comparaciones, se puede poner cualquier función que devuelva un valor. Dependiendo de si este valor es cero o no, la condición será falsa o verdadera.
También podemos probar varias condiciónes en una sola usando && (AND), || (OR).

Ejemplos de && (AND):

```
if ( a==3 && b==2 ) printf( "Hola\n" );        /* Se cumple si a es 3 Y b es dos */

if ( a>10 && a<100 ) printf( "Hola\n" );       /* Se cumple si a es mayor
                                  que 10 Y menor que 100 */

if ( a==10 && b<300 ) printf( "Hola\n" ); /* Se cumple si a es igual a 10
                                                                  Y b es menor que
300 */
```

Ejemplos de || (OR):

```
if ( a<100 || b>200 ) printf( "Hola\n" ); /*Se cumple si a menor que 100 o b mayor que 200
*/
if ( a<10 || a>100 ) printf( "Hola\n" ); /* Se cumple si a menor que 10 o a mayor que 100 */
```

Se pueden poner más de dos condiciones:

```
if ( a>10 && a<100 && b>200 && b<500 )    /* Se deben cumplir las cuatro condiciones */
```

Esto se cumple si a está entre 10 y 100 y b está entre 200 y 500.
También se pueden agrupar mediante paréntesis varias condiciones:

```
if (  ( a>10 && a<100 ) || ( b>200 && b<500 )  )
```

Esta condición se leería como sigue:

```
si a es mayor que 10 y menor que 100
        o
si b es mayor que 200 y menor que 500
```

Es decir que si se cumple el primer paréntesis o si se cumple el segundo la condición es cierta.

CAPITULO IV:

Arrays: Vectores y matrices

En este tema se describirán las herramientas que proporciona el lenguaje **C** para trabajar con tipos y estructuras de datos, flexibilizando de esta forma la creación de programas por parte del programador.

4.1 Vectores

Un vector es una porción de memoria que es utilizada para almacenar un grupo de elementos del mismo tipo y se diferencian en el índice.

Pero ¿qué quiere decir esto y para qué lo queremos?. Pues bien, supongamos que somos un metereólogo y queremos guardar en el ordenador la temperatura que ha hecho cada hora del día. Para darle cierta utilidad al final calcularemos la media de las temperaturas. Con lo que sabemos hasta ahora sería algo así (que nadie se moleste ni en probarlo):

```
#include <stdio.h>
int main()
{
        /* Declaramos 24 variables, una para cada hora del dia */
        int temp1, temp2, temp3, temp4, temp5, temp6, temp7, temp8;
        int temp9, temp10, temp11, temp12, temp13, temp14, temp15, temp16;
        int temp17, temp18, temp19, temp20, temp21, temp22, temp23, temp0;
        int media;

        /* Ahora tenemos que dar el valor de cada una */
        printf( "Introduzca las temperaturas desde las 0 hasta las 23 separadas por un
espacion: " );
        scanf( "%i %i %i ... %i", &temp0, &temp1, &temp2, ... &temp23 );
        media = ( temp0 + temp1 + temp2 + temp3 + temp4 + ... + temp23 ) / 24;
        printf( "\nLa temperatura media es %i\n", media );
}
```

¡ufff!, ¿grande verdad?. Y esto con un ejemplo que tiene tan sólo 24 variables, ¡imagínate si son más!.

Y precisamente aquí es donde nos vienen de perlas los arrays.

4.1.1 Declaración De Vectores

Un vector se declara:

```
Tipo_ de_dato  nombre_del_vector [ dimensión ];
```

Tipo_ de dato: Es el tipo de datos que contendrá la matriz. Hasta ahora sólo conocemos los tipos básicos de datos; int, float, double, char. Posteriormente veremos como definir nuestros propios tipos de datos.

nombre_del_vector: Es el nombre que le damos a la variable tipo vector y por el cual la referenciaremos en nuestro programa.

[dimension]: Indica el número de elementos de tipo tipo_de_datos contendrá la matriz identificador.

Por ejemplo:

```
int modulo[52];        /*Aquí 'modulo' es un vector de 52 elementos enteros*/
int vector [5] ;       /* Crea un vector de cinco enteros */
```

Un tipo vector muy utilizado es la cadena de caracteres (*string*). Si queremos asignar espacio para un string podemos hacer:

```
char nombre[60], direccion[80];
```

Es un vector **C** pero con la particularidad que el propio lenguaje utiliza un carácter especial como marca de final de string. Así en un vector de caracteres de tamaño N podremos almacenar una cadena de N-1 caracteres, cuyo último carácter estará en la posición N-2 y la marca de final de string en la N-1. Veamos un ejemplo:

```
char servei[6] = "SCI";
```

La posición 0 contiene el carácter 'S'; la 1 el 'C'; la 2 el 'I'; la 3 el '\0', marca de final de string y el resto de componentes no están definidas. En la inicialización de strings no se debe indicar el final; ya lo hace el compilador. Para la manipulación de cadenas de caracteres ANSI proporciona el fichero string.h que contiene las declaraciones de un conjunto de funciones proporcionadas con la librería del compilador.

4.1.2 Inicialización De Vectores

Cada elemento de un vector es accedido mediante un *número de índice* y se comporta como una variable del tipo base del vector. Los elementos de un vector son accedidos por índices que van desde **0** hasta **N-1** para un vector de **N** elementos.

- Los elementos de un vector pueden ser inicializados en la misma declaración:

```
char vocal[5] = { 'a', 'e', 'i', 'o', 'u' };

int vector [3] = {3, 5, 6};

float n_Bode[5] = { 0.4, 0.7, 1, 1.6, 2.8 };
```

- Los vectores se pueden inicializar en cualquier parte del programa (excepto para cadenas).

```
int vector [3];
vector [0] = 3;
vector [1] = 5;
vector [2] = 6;
```

- Los vectores pueden ser inicializados utilizando bucles. Ejemplo:

```
#include <stdio.h>
#include <conio.h>
int main(void)
{
        int vector [5], i;

        /*Inicializamos el vector*/
        for (i=0;i<5;i++)
                vector[i] = i;

        /*Mostramos el vector por pantalla*/
```

```
        for (i=0;i<5;i++)
                printf("\n%d", vector[i]);

        getch();
        return 0;
}
```

Ahora vamos a realizar el programa de las temperaturas con vectores:

```
#include <stdio.h>
int main(void)
{
        int temp[24]; /* Con esto ya tenemos declaradas las 24 variables */
        float media;
        int hora;

        /* Ahora tenemos que dar el valor de cada una */
        for( hora=0; hora<24; hora++ )
        {
                printf( "Temperatura de las %i: ", hora );
                scanf( "%i", &temp[hora] );
                media += temp[hora];
        }
        media = media / 24;

        printf( "\nLa temperatura media es %f\n", media );
}
```

Como ves es un programa más rápido de escribir (y es menos aburrido hacerlo), y más cómodo para el usuario que el anterior.

Como ya hemos comentado cuando declaramos una variable lo que estamos haciendo es reservar una zona de la memoria para ella. Cuando declaramos un array lo que hacemos (en este ejemplo) es reservar espacio en memoria para 24 variables de tipo int. El tamaño del array (24) lo indicamos entre corchetes al definirlo. Esta es la parte de la definición que dice: Un array es un conjunto de variables del mismo tipo que tienen el mismo nombre.

La parte final de la definición dice: y se diferencian en el índice. En ejemplo recorremos la matriz mediante un bucle for y vamos dando valores a los distintos elementos de la matriz. Para indicar a qué elemento nos referimos usamos un número entre corchetes (en este caso la variable hora), este número es lo que se llama Indice. El primer elemento de la matriz en C tiene el índice 0, El segundo tiene el 1 y así sucesivamente. De modo que si queremos dar un valor al elemento 4 (índice 3) haremos:

```
temp[ 3 ] = 20;
```

NOTA: No hay que confundirse. En la declaración del array el número entre corchetes es el número de elementos, en cambio cuando ya usamos la matriz el número entre corchetes es el índice.

¿Qué pasa si metemos más datos de los reservados?. Vamos a meter 25 datos en vez de 24. Si hacemos esto dependiendo del compilador obtendremos un error o al menos un warning (aviso). En unos compiladores el programa se creará y en otros no, pero aún así nos avisa del fallo. Debe indicarse que estamos intentando guardar un dato de más, no hemos reservado memoria para él.

Si la matriz debe tener una longitud determinada usamos este método de definir el número de elementos. En nuestro caso era conveniente, porque los dias siempre tienen 24 horas. Es conveniente definir el tamaño de la matriz para que nos avise si metemos más elementos de los necesarios.

En los demás casos podemos usar un método alternativo, dejar al ordenador que cuente los elementos que hemos metido y nos reserve espacio para ellos:

```
#include <stdio.h>
int main()
{
        int hora;
        /* Faltan los tres últimos elementos */
        int temperaturas[]={  15, 18, 20, 23, 22, 24, 22, 25, 26, 25, 24,
                              22, 21, 20, 18, 17, 16, 17, 15, 14, 14
                            };

        for ( hora=0 ; hora<24 ; hora++ )
        {
                printf( "La temperatura a las %i era de %i grados.\n", hora,
temperaturas[hora] );
        }
}
```

Vemos que no hemos especificado la dimensión del array temperaturas. Hemos dejado los corchetes en blanco. El ordenador contará los elementos que hemos puesto entre llaves y reservará espacio para ellos. De esta forma siempre habrá el espacio necesario, ni más ni menos. La pega es que si ponemos más de los que queríamos no nos daremos cuenta.

Para saber en este caso cuantos elementos tiene la matriz podemos usar el operador sizeof. Dividimos el tamaño de la matriz entre el tamaño de sus elementos y tenemos el número de elementos.

```
#include <stdio.h>
int main()
{
        int hora;
        int elementos;
        int temperaturas[] = { 15, 18, 20, 23, 22, 24, 22, 25, 26, 25,
                               24, 22, 21, 20, 18, 17, 16, 17, 15, 14, 14 };

        elementos = sizeof temperaturas / sizeof (int);
        for ( hora=0 ; hora<elementos ; hora++ )
        {
                printf( "La temperatura a las %i era de %i grados.\n", hora, temperas[hora] );
        }
        printf( "Han sido %i elementos.\n" , elementos );
}
```

¿Qué pasa si intentamos imprimir más elementos de los que hay en la matriz?

En este caso intentamos imprimir 28 elementos cuando sólo hay 24:

```
#include <stdio.h>

int main()
```

```
{
        int hora;
        int temperaturas[24] = { 15, 18, 20, 23, 22, 24, 22, 25, 26, 25, 24,
                                 22, 21, 20, 18, 17, 16, 17, 15, 14, 14, 13, 13, 12 };

        for (hora=0 ; hora<28 ; hora++ )
        {
                printf( "La temperatura a las %i era de %i grados.\n", hora,
        temperaturas[hora] );
        }
}
```

Lo que se obtiene en mi ordenador es:

La temperatura a las 22 era de 15 grados.
...
La temperatura a las 23 era de 12 grados.
La temperatura a las 24 era de **24 grados**.
La temperatura a las 25 era de **6684216 grados**.
La temperatura a las 26 era de **4199177 grados**.
La temperatura a las 27 era de **1 grados**.

Vemos que a partir del elemento 24 (incluído) tenemos resultados extraños. Esto es porque nos hemos salido de los límites del array e intenta acceder al elemento temperaturas[25] y sucesivos que no existen. Así que nos muestra el contenido de la memoria que está justo detrás de temperaturas[23] (esto es lo más probable) que puede ser cualquiera. Al contrario que otros lenguajes C no comprueba los límites de los array. Este programa no da error al compilar ni al ejecutar, tan sólo devuelve resultados extraños. Tampoco bloqueará el sistema porque no estamos escribiendo en la memoria sino leyendo de ella.

Otra cosa muy diferente es meter datos en elementos que no existen. Veamos un ejemplo (ni se te ocurra ejecutarlo):

```
#include <stdio.h>
int main()
{
        int temp[24];
        float media;
        int hora;
        for( hora=0; hora<28; hora++ )
        {
                printf( "Temperatura de las %i: ", hora );
                scanf( "%i", &temp[hora] );
                media += temp[hora];
        }
        media = media / 24;
        printf( "\nLa temperatura media es %f\n", media );
}
```

El problema ahora es que estamos intentando escribir en el elemento temp[24] que no existe y puede ser un lugar cualquiera de la memoria. Como consecuencia de esto podemos estar cambiando algún programa o dato de la memoria que no debemos y el sistema hace pluf. Así que mucho cuidado con esto.

4.2 Matrices

En **C** se puede manejar arreglos de 2, 3 o más dimensiones siendo los más comunes los de 2 dimensiones representadas por filas y columnas, según esta sintaxis:

Tipo_matriz nombre [dimensión1] [dimensión2] ... ;

Para una matriz bidimensional (Matriz), en particular su declaración será:

Tipo_matriz nombre [Tamaño_FILAS] [Tamaño_COLUMNAS] ;

Ejemplo:

```
int matriz [3][8] ;

int video[25][80][2];

static int valor [3][3] = {1,2,3,4,5,6,7,8,9};
```

4.2.1 Rellenar matrices

- Para rellenar entera una matriz se necesitaran dos bucles FOR, el primero recorrerá las filas y el segundo las columnas.

  Ejemplo: inicializar una matriz a ceros:

```
for (i = 0; i < 2; i++)
        for ( j = 0; j < 2; j++)
                matriz [i] [j] = 0;
```

- Las variables de tipo matriz como el resto de las declaraciones, se pueden inicializar en el momento de su declaración, ayudándose de las llaves ({}) para la inclusión de inicializaciones múltiples.
  Ejemplo: inicializar una matriz llamada "matriz" de 2x3 elementos con los valores deseados

```
int matriz[2][3] = { { 1,2,3 }, { 4,5,6 } };
```

Las matrices son extremadamente útiles para trabajar con multitud de problemas matemáticos que se formulan de esta forma o para mantener tablas de datos a los que se accede con frecuencia y por tanto su referencia tiene que ser muy rápida. Supongamos que estamos desarrollando un programa para dibujar objetos en tres dimensiones y más que la exactitud de la representación (aunque esto es muy relativo), nos interesa la velocidad. En la representación de objetos tridimensionales se hace continua referencia a las funciones trigonométricas seno y coseno. El cálculo de un seno y un coseno puede llevar bastante tiempo así que antes de comenzar la representación calculamos todos los senos y cosenos que necesitemos (por ejemplo con una resolución de 1 grado -360 valores-) cada vez que necesitemos uno de estos valores accedemos a la matriz en lugar de llamar a la función que nos lo calcula. Veamos como sería nuestro programa (las funciones sin y cos se encuentran en la librería estándar math.h y sus parámetros están en radianes).

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

const double PI = 3.14159;

int main(void)
{
        double senos[360]; /* Almacenamos senos */
```

```
        double cosenos[360];
        int   i;

        /* Inicializamos las matrices */
        for (i=0;i<360;i++)
        {
                senos[i]  = sin (PI*i/180);
                cosenos[i] = cos (PI*i/180);
        }
        printf ("El coseno de 30 es: %f\n",cosenos[30]);
        printf ("El seno de 30 es: %f\n",senos[30]);
        getch();
        return 0;
}
```

4.3 Sobre la dimensión de un array

Observando los programas vistos hasta ahora, podrá comprobar el hecho de que declarar un array de N elementos conlleva que en todas las estructuras repetitivas que gestionen dicho array, la variable de control tome valores que vayan de 0 hasta N elementos; por ejemplo, si tuvieramos que gestionar las notas de alumnos de una clase, si por ejemplo tuvieramos 10 notas:

```
int Notas[10];
.
.
for (i=0; i < 10; i++)
.
.
```

Si ahora en vez de tener 10 notas tuviésemos 15, ó 5, ó 7, ó... se debería redefinir el array con el número de elementos y, cambiar todos los bucles que gestionanen el array. En este programa el problema se soluciona rápidamente cambiando el valor de la declaración del array, y después el del bucle. Si se tratará de un programa mucho más largo, con muchas estructuras repetitivas, a la hora de gestionar el array no sería tan fácil modificarlo (por ejemplo, un programa con centenares de líneas en el cual se tuvieran que buscar las estructuras que gestionan el array). Para solucionar este tipo de eventualidades se puede utilizar la directiva **#define**; ésta sirve para declarar un valor constante para todo el programa. Así pues, si se decide modificar la capacidad del array, lo único que hay que hacer es cambiar el valor de este identificador. El programa de Notas quedaría así:

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

#define TAMANYO 10
main()
{
        int NotAes[TAMANYO];
        int i;
        for (i=0 ; i < TAMANYO; i++)
        {
                printf("EntrAs la Nota : %d", i+1);
                scanf("%d", &nota[i]);
        }
}
```

Si ahora queremos gestionar una clase de 200 alumno, únicamente deberemos modificar la línea #define TAMANYO 10, por #define TAMANYO 200.

Las constantes #define se suelen a escribir en mayúscula, para diferenciarlas de las variables.

Una constante no ocupa memoria como una variable, lo único que hace el compilador es sustituir la constante por el valor que ésta representa.

No se puede cambiar el valor de una constante; por ejemplo, no se puede hacer la siguiente asignación:

```
TAMANYO = 15;
```

4.4 Precauciones con los vectores

- El compilador de **C** reconoce la expresión vector[i,j], pero es un error.
- El **C** numera los elementos de un vector desde CERO.
- El **C** no detecta índices fuera de rango.
- Si **A** y **B** son vectores, la expresión **A = B** es ilegal.

4.5 Cadenas de caracteres

Para empezar diré que en **C** no existe un tipo string como en otros lenguajes. No existe un tipo de datos para almacenar texto, se utilizan arrays de chars. Funcionan igual que los demás arrays con la diferencia que ahora jugamos con letras en vez de con números.

Se les llama cadenas, strings o tiras de caracteres. A partir de ahora les llamaremos cadenas.

Para declarar una cadena se hace como un array:

```
char texto[20];
```

Para inicializar una cadena se puede utilizar cualquiera de las dos formas:

```
char saludación[5]= { 'h','o','l','a','\0'} ;
```

o bien,

```
char saludación[5] = "Hola" ;
```

En la declaraciones anteriores notamos que se añade el carácter '\0', esto es porque en **C** las cadenas terminan con este carácter, pero no nos preocupemos aun de esto porque lo veremos mas adelante.

Si alguno viene de algún otro lenguaje esto es importante: en **C** no se puede hacer esto:

```
int main()
{
        char texto[20];
        texto = "Hola";
}
```

Al igual que en los arrays no podemos meter más de 20 elementos en la cadena. Vamos a ver un ejemplo para mostrar el nombre del usuario en pantalla:

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

```
        char nombre[20];
        printf( "Introduzca su nombre (20 letras máximo): " );
        scanf( "%s", nombre );
        printf( "\nEl nombre que ha escrito es: %s\n", nombre );
}
```

Vemos cosas curiosas como por ejemplo que en el scanf no se usa el símbolo &. No hace falta porque es un array, y ya sabemos que escribir el nombre del array es equivalente a poner &nombre[0].

También puede llamar la atención la forma de imprimir el array. Con sólo usar %s ya se imprime todo el array. Ya veremos esto más adelante.

4.5.1 Las cadenas por dentro

Es interesante saber cómo funciona una cadena por dentro, por eso vamos a ver primero cómo se inicializa una cadena.

```
#include <stdio.h>
int main()
{
        char nombre[] = "Julio";
        printf( "Texto: %s\n", nombre );
        printf( "Tamaño de la cadena: %i bytes\n", sizeof nombre );
}
```

Resultado al ejecutar:

```
Texto: Julio
Tamaño de la cadena: 6 bytes
```

¡Qué curioso! La cadena es "Julio", sin embargo nos dice que ocupa 6 bytes. Como cada elemento (char) ocupa un byte eso quiere decir que la cadena tiene 6 elementos. ¡Pero si "Julio" sólo tiene 5! ¿Por qué? Muy sencillo, porque al final de una cadena se pone un símbolo '\0' que significa "Fin de cadena". De esta forma cuando queremos escribir la cadena basta con usar %s y el programa ya sabe cuántos elementos tiene que imprimir, hasta que encuentre '\0'.

El programa anterior sería equivalente a:

```
#include <stdio.h>
int main()
{
        char nombre[] = { 'J', 'u', 'l', 'i', 'o', '\0' };
        printf( "Texto: %s\n", nombre );
}
```

Aquí ya se ve que tenemos 6 elementos. Pero, ¿Qué pasaría si no pusiéramos '\0' al final?

```
#include <stdio.h>
int main()
{
        char nombre[] = { 'J', 'u', 'l', 'i', 'o', '\0' };
        printf( "Texto: %s\n", nombre );
        printf("Tamanio de la cadena: %d",sizeof (nombre));
}
```

En mi ordenador salía:

```
Texto: Julio¦¦¦8_e
Tamanio de la cadena: 5
```

Pero en el tuyo después de "Julio" puede aparecer cualquier cosa. Lo que aquí sucede es que no encuentra el símbolo '\0' y no sabe cuándo dejar de imprimir. Afortunadamente, cuando metemos una cadena se hace de la primera forma y el **C** se encarga de poner el dichoso símbolo al final.

Es importante no olvidar que la longitud de una cadena es la longitud del texto más el símbolo de fin de cadena. Por eso cuando definamos una cadena tenemos que reservarle un espacio adicional. Por ejemplo:

```
char nombre[6] = "Julio";
```

Si olvidamos esto podemos tener problemas.

4.5.2 Entrada de cadenas por teclado

4.5.2.1 scanf

Hemos visto en capítulos anteriores el uso de scanf para números, ahora es el momento de ver su uso con cadenas.

Scanf almacena en memoria (en un buffer) lo que vamos escribiendo. Cuando pulsamos ENTER (o Intro o Return, como se llame en cada teclado) lo analiza, comprueba si el formato es correcto y por último lo mete en la variable que le indicamos.

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

int main()
{
        char cadena[30];

        printf( "Escribe una palabra: " );
        fflush( stdout );
        scanf( "%s", cadena );
        printf( "He guardado: \"%s\" \n", cadena );
}
```

Ejecutamos el programa e introducimos la palabra "hola". Esto es lo que tenemos:

```
Escribe una palabra: hola
He guardado: "hola"
```

Si ahora introducimos "hola amigos" esto es lo que tenemos:

```
Escribe una palabra: hola amigos
He guardado: "hola"
```

Sólo nos ha cogido la palabra "hola" y se ha olvidado de amigos. ¿Por qué? pues porque scanf toma una palabra como cadena. Usa los espacios para separar variables.

Es importante siempre asegurarse de que no vamos a almacenar en *cadena* más letras de las que caben. Para ello debemos limitar el número de letras que le va a introducir scanf. Si por ejemplo queremos un máximo de 5 caracteres usaremos %5s:

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

int main()
{
	char cadena[6];

	printf( "Escribe una palabra: " );
	fflush( stdout );
	scanf( "%5s", cadena );
	printf( "He guardado: \"%s\" \n", cadena );
}
```

Si metemos una palabra de 5 letras (no se cuenta '\0') o menos la recoge sin problemas y la guarda en *cadena*.

```
Escribe una palabra: Julio
He guardado: "Julio"
```

Si metemos más de 5 letras nos cortará la palabra y nos dejará sólo 5.

```
Escribe una palabra: Juanjo
He guardado: "Juanj"
```

scanf tiene más posibilidades (consulta la ayuda de tu compilador), entre otras permite controlar qué caracteres entramos. Supongamos que sólo queremos coger las letras mayúsculas:

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

int main()
{
	char cadena[30];

	printf( "Escribe una palabra: " );
	fflush( stdout );
	scanf( "%[A-Z]s", cadena );
	printf( "He guardado: \"%s\" \n", cadena );
}
```

Guarda las letras mayúsculas en la variable hasta que encuentra una minúscula:

```
Escribe una palabra: Hola
He guardado: "H"
Escribe una palabra: HOLA
He guardado: "HOLA"
Escribe una palabra: AMigOS
He guardado: "AM"
```

4.5.2.2 gets

Esta función nos permite introducir frases enteras, incluyendo espacios.

```
#include <stdio.h>
char *gets(char *buffer);
```

Almacena lo que vamos tecleando en la variable *buffer* hasta que pulsamos ENTER. Si se ha almacenado algún caracter en *buffer* le añade un '\0' al final y devuelve un puntero a su dirección. Si no se ha almacenado ninguno devuelve un puntero NULL.

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

int main()
{
        char cadena[30];
        char *p;
        printf( "Escribe una palabra: " );
        fflush( stdout );
        gets( cadena );
        printf( "He guardado: \"%s\" \n", cadena );
}
```

Esta función es un poco peligrosa porque no comprueba si nos hemos pasado del espacio reservado (de 29 caracteres en este ejemplo: 29letras+'\0').

4.5.2.3 Qué son los buffer y cómo funcionan

```
#include <stdio.h>
int main()
{
        char ch;
        char nombre[20], apellido[20], telefono[10];
        printf( "Escribe tu nombre: " );
        scanf( "%[A-Z]s", nombre );
        printf( "Lo que recogemos del scanf es: %s\n", nombre );
        printf( "Lo que había quedado en el buffer: " );
        while( (ch=getchar())!='\n' )
        printf( "%c", ch );
}
```

Cuidado con scanf!!!

```
#include <stdio.h>

int main()
{
        char nombre[20], apellido[20], telefono[10];
        printf( "Escribe tu nombre: " );
        scanf( "%s", nombre );
        printf( "Escribe tu apellido: " );
        gets( apellido );
}
```

4.5.3 Funciones de manejo de cadenas

Existen unas cuantas funciones el la biblioteca estándar de C para el manejo de cadenas. Para usar estas funciones hay que añadir la directiva:

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

4.5.3.1 strlen

Esta función nos devuelve el número de caracteres que tiene la cadena (sin contar el '\0').

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

int main()
{
	char texto[]="Julio";
	int longitud;

	longitud = strlen(texto);
	printf( "La cadena \"%s\" tiene %i caracteres.\n", texto, longitud );
}
```

4.5.3.2 strcpy

```
#include <string.h>
char *strcpy(char *cadena1, const char *cadena2);
```

Copia el contenido de *cadena2* en *cadena1*. *cadena2* puede ser una variable o una cadena directa (por ejemplo "hola"). Debemos tener cuidado de que la cadena destino (cadena1) tenga espacio suficiente para albergar a la cadena origen (cadena2).

```
#include <stdio.h>
#include <string.h>
int main()
{
	char textocurso[] = "Este es un curso de C.";
	char destino[50];
	strcpy( destino, textocurso );
	printf( "Valor final: %s\n", destino );
}
```

Vamos a ver otro ejemplo en el que la cadena destino es una cadena constante ("Este es un curso de C") y no una variable. Además en este ejemplo vemos que la cadena origen es sustituida por la cadena destino totalmete. Si la cadena origen es más larga que la destino, se eliminan las letras adicionales.

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

int main()
{
	char destino[50] = "Esto no es un curso de HTML sino un curso de C.";
	printf( "%s\n", destino );
	strcpy( destino, "Este es un curso de C." );
	printf( "%s\n", destino );
```

```
}
```

4.5.3.3 strcat

```
#include <string.h>
char *strcat(char *cadena1, const char *cadena2);
```

Copia la *cadena2* al final de la *cadena1*.

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

int main()
{
	char nombre_completo[50];
	char nombre[]="Kevin";
	char apellido[]="Paredes";

	strcpy( nombre_completo, nombre );
	strcat( nombre_completo, " " );
	strcat( nombre_completo, apellido );
	printf( "El nombre completo es: %s.\n", nombre_completo );
}
```

Como siempre tenemos que asegurarnos que la variable en la que metemos las demás cadenas tenga el tamaño suficiente. Con la primera línea metemos el nombre en *nombre_completo*. Usamos strcpy para asegurarnos de que queda borrado cualquier dato anterior. Luego usamos un strcat para añadir un espacio y finalmente metemos el apellido.

4.5.3.4 Sprintf

```
#include <stdio.h>
int sprintf(char *destino, const char *format, ...);
```

Funciona de manera similar a printf, pero en vez de mostrar el texto en la pantalla lo guarda en una variable (*destino*). El valor que devuelve (int) es el número de caracteres guardados en la variable *destino*.

Con sprintf podemos repetir el ejemplo de strcat de manera más sencilla:

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

int main()
{
	char nombre_completo[50];
	char nombre[]="Kevin";
	char apellido[]="Paredes";

	sprintf( nombre_completo, "%s %s", nombre, apellido );
	printf( "El nombre completo es: %s.\n", nombre_completo );
}
```

Se puede aplicar a sprintf todo lo que valía para printf.

4.5.3.5 strcmp

```
#include <string.h>
int strcmp(const char *cadena1, const char *cadena2);
```

Compara *cadena1* y *cadena2*. Si son iguales devuelve 0. Un número negativo si *cadena1* va antes que *cadena2* y un número positivo si es al revés:

- cadena1 == cadena2 -> 0
- cadena1 > cadena2 -> número negativo
- cadena1 < cadena2 -> número positivo

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

int main()
{
        char nombre1[]="Kevin";
        char nombre2[]="Alvaro";
        printf( "%i", strcmp(nombre1,nombre2));
}
```

4.5.3.6 Otras funciones

- **strlwr:** Convierte una cadena en minúsculas.

```
char *strlwr(char *cadena)

strlwr(Nombre) ;
```

- **strupr:** Convierte una cadena en mayúsculas.

```
char *strupr(char *cadena)

strupr(Nombre) ;
```

4.5.4 Array de cadenas

Si en un programa en **C** declaramos un array de cadena de caracteres en la forma:

```
char nombres[5][15];
```

Estamos declarando un objeto capaz de almacenar 5 cadenas de 14 caracteres cada una. Recordemos que el último carácter de una cadena es el '\0'.

Si lo inicializamos en la forma:

```
char nombres[5][15] ={
                        {"JUAN"},
                        {"PEDRO"},
                        {"MARÍA JOSÉ"},
                        {"VERÓNICA"},
                        {"ANGEL"}
                };
```

tenemos la típica representación en forma de array. Podríamos representarlo como:

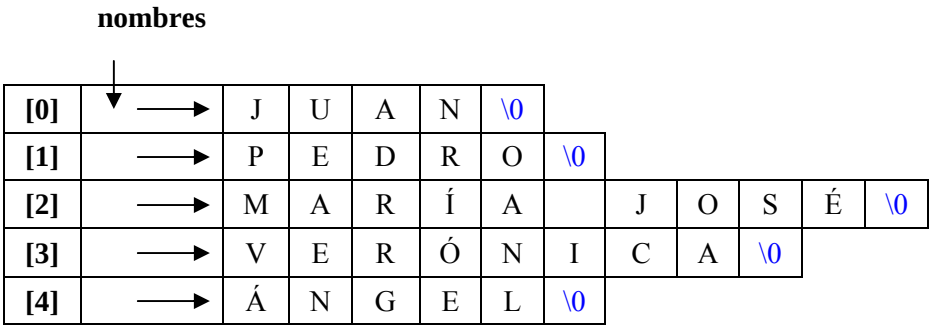
	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**
0	J	U	A	N	\0										
1	P	E	D	R	0	\0									
2	M	A	R	Í	A		J	O	S	É	\0				
3	V	E	R	Ó	N	I	C	A	\0						
4	A	N	G	E	L	\0									

En realidad la parte ocupada es mínima, forma un array de "borde irregular" (concepto propio de los procesadores de texto), justificado a la izquierda. La parte no ocupada está rellena con ceros.

Para evitar ese derroche de memoria, declaramos lo anterior como un array unidimensional de punteros a carácter:

```
char *nombres [5] ={
                        {"JUAN"},
                        {"PEDRO"},
                        {"MARÍA JOSÉ"},
                        {"VERÓNICA"},
                        {"ANGEL"}
                };
```

El asterisco delante del nombre hace que se trate de un array de punteros a dato tipo char. Cada uno de los 5 punteros guarda la dirección del primer carácter de una cadena. Cada cadena ocupa la memoria requerida, no como antes. El aspecto de esta nueva declaración podría ser algo como:



Ejemplo: arraycad.cpp: Pide tres cadenas de caracteres, las guarda en una array bidimensional y la imprime de diferentes formas (puts(), printf() y carácter a carácter

```
#include<stdio.h>
#include <conio.h>
int main(void)
{
        char cad[3][50];
        int i=0, j=0;
        puts("Intro cadenas:\n");
        for(; i<3; i++)
        {
                printf("\nCadena %d: ", i+1);
                gets(cad[i]);
        }
        puts("\nImprimimos las cadenas con puts(): ");
        for(i=0; i<3; i++)
```

```
        {
                printf("\nCadena %d: ", i+1);
                puts(cad[i]);
        }
        puts("\nImprimimos las cadenas con printf():");
        for(i=0; i<3; i++)
        {
                printf("\nCadena %d: ", i+1);
                printf("%s", cad[i]);
        }

        puts("\n\nImprimimos caracter a caracter (hasta encontrar '\\0'");
        puts("Pulsa una tecla para ir imprimiendo los caracteres:");
        i=0; j=0;
        for(i=0; i<3; i++)
        {
                printf("Cadena %d: ", i+1);
                do
                {
                        printf("%c", cad[i][j]);
                        getch();
                }       while(cad[i][j++]!='\0');
                j=0; /*me pongo al principio de cada cadena*/
                printf("\n");
        }

        puts("Imprimimos como punteros:");
        for(i=0; i<3; i++)
        {
                printf("\nCadena %d: ", i+1);
                printf("%s", *(cad + i));
        }

        puts("\nImprimimos con puts():");
        for(i=0; i<3; i++)
        {
                printf("\nCadena %d: ", i+1);
                puts(cad[i]);
        }

        puts("\nImprimimos con puts() + notacion de punteros:");
        for(i=0; i<3; i++)
        {
                printf("\nCadena %d: ", i+1);
                puts(*(cad + i));
        }
        getch();
        getch();
        return 0;
}
```

4.6 Ordenacion de Vectores

Ordenar datos es una de las más frecuentes aplicaciones informáticas. Ejemplos:

- Los bancos ordenan los cheques por el número de cuenta con el fin de actualizar el estado de cada cuenta.
- Las guías telefónicas son bases de datos con nombres de clientes clasificados por apellidos y nombres.

La clasificación de datos ha generado intensos esfuerzos de investigación en Informática.

Una correcta ordenación genera bases de datos más manejables en las que los procesos de búsqueda están optimizados.

Existen muchos métodos de ordenación , basado en diferentes algoritmos, de muy variada dificultad.

Al tratarse de datos agrupados (arrays) se va a exigir un recorrido secuencial, ascendente o descendente. Es necesario conocer perfectamente el funcionamiento de los bucles.

Métodos de ordenación:

- Burbuja
- Selección
- Inserción
- Shell
- Heap

4.6.1 Método de la BURBUJA

El método de la Burbuja es el más sencillo para clasificar una lista de números o cadenas de caracteres.
Es llamado así porque los valores "parecen ascender", como una burbuja hasta l un extremo del array.

La técnica consiste en realizar múltiples pasadas por los elementos del array:

- En cada pasada se comparan pares de elementos.
- Si estos dos elementos están ordenados, se les deja como están.
- Si están desordenados, se intercambian mediante el empleo de una variable auxiliar

```
#include <stdio.h>
#include <conio.h>
#define TAM 10
int main()
{
        int a[TAM] = { 82, 6, 4, 12, 19, 12, 89, 68, 45, 37 };
        int i, pasada, aux;
        printf( "Datos en el orden inicial:\n\n" );
        for (i=0;i<=TAM-1;i++ )
                printf("%4d",a[i]);
  for (pasada=1;pasada<=TAM-1; pasada++ )       /*pasadas*/
                for (i=0;i<=TAM-2;i++ )
                        if (a[i]>a[i+1]) /*comparación */
                        {

                                //intercambio
                                aux = a[i];
                                a[i] = a[i+1];
                                a[i+1] = aux;
                        }
```

```
        printf( "\n\nDatos ordenados en sentido ascendente:\n\n" );
        for (i=0;i<=TAM-1;i++ )
                printf("%4d", a[i]);
        printf("\n");
        getch();
        return 0;
}
```

4.6.1.1 Ventajas/Desventajas del método de la Burbuja

- La principal ventaja es que el código que genera es fácil de entender.
- La principal desventaja es que es muy lento.
- Se ha desarrollado muchísimo trabajo en informática en la búsqueda de algoritmos de ordenación eficientes. Ejemplos: Merge, Inserción, Quick.

4.7 Busqueda en vectores

4.7.1 Búsqueda Lineal

Se busca en el array mediante un valor clave.
Búsqueda lineal:
- Simple
- Compara cada elemento del array con el valor clave.

Útil para arrays pequeños y desordenados.

Veamos un ejemplo en C:

```
#include <stdio.h>
#include <conio.h>
#define TAM 100
int main()
{
        int a[TAM], x, clave, elemento= -1, n;
        for (x=0;x<=TAM-1;x++)  /* rellenamos el array */
                a[x] = 2*x;
        printf( "Intro la clave int a buscar:\n" );
        scanf("%d", &clave);
        for (n=0;n<=TAM - 1; ++n )
                if (a[n]== clave)
                        elemento = n;
        if (elemento != -1 )
                printf( "Valor encontrado en posicion %d\n", elemento);
        else
                printf( "Valor no encontrado\n" );
        getch();
        return 0;
}
```

4.7.2 Búsqueda Binaria: Características

La búsqueda lineal no es muy eficiente. En el peor de los casos, el elemento a buscar puede estar situado al final del array, requiriendo que revisemos cada uno de los elementos del array.

Búsqueda binaria:

Dividiremos el array en dos partes: Buscamos en una de ellas. Si lo encontramos, sólo hemos revisado la mitad de los elementos del array.

La Búsqueda Binaria es la más eficiente opción para buscar en arrays.

Hay dos condiciones ineludibles en este método:

- El array debe estar ordenado. Antes de usar el método de búsqueda binaria, debemos ordenarlo empleando, por ejemplo el método de la burbuja.

- La búsqueda binaria funciona dividiendo el área de búsqueda en dos mitades antes de cada comparación.

Está muy realcionada con el concepto de "Árboles de Búsqueda Binaria".

Eficiencia de la Búsqueda Binaria

Algoritmo muy eficiente.

Por ejemplo: en un array de 1024 elementos, en el peor de los casos, necesitaremos:
- Búsqueda Lineal: 1024 comparaciones
- Búsqueda Binaria: 10 comparaciones.

Si tuviéramos 1.000.000.000 de elementos

- Búsqueda Lineal: 1.000.000.000 de comparaciones
- Búsqueda Binaria: 30 comparaciones

```
#include <stdio.h>
#include <conio.h>
#define TAM 15
int main()
{
        int a[TAM], i, num, resultado = -1, mitad, alto, bajo;

        for (i=0 ; i<=TAM-1; i++ )
                a[i] =2*i;   /*El array es 0, 2, 4, 6, ..... 28*/

        printf( "Intro numero entre 0 y 28: " );
        scanf( "%d", &num);



        bajo = 0;
        alto = TAM - 1;
        while (bajo<=alto)
        {
                mitad = (bajo+alto)/2;
                if (num == a[mitad])
                        resultado = mitad;
                if (num < a[mitad])
```

```
                alto = mitad - 1;
            else
                bajo = mitad + 1;
        }
        if(resultado != -1 )
            printf("\n%d encontrado en posicion %d\n", num, resultado);
        else
            printf( "\n%d no encontrado\n", num);
        getch();
        return 0;
}
```

CAPITULO V:

Estructuras y Uniones

5.1 Estructuras

Así como los arrays son organizaciones secuenciales de variables simples, de un mismo tipo cualquiera dado, resulta necesario en múltiples aplicaciones, agrupar variables de distintos tipos, en una sola entidad. Este sería el caso, si quisiéramos generar la variable "Datos_Personales", en ella tendríamos que incluir variables del tipo: strings, para el nombre, apellido, nombre de la calle en donde vive, etc, enteros, para la edad, número de codigo postal, float (ó double, si tiene la suerte de ganar mucho) para el sueldo, y así siguiendo.

Existe en **C** en tipo de variable compuesta, para manejar ésta situación típica de las Bases de Datos, llamada ESTRUCTURA. No hay limitaciones en el tipo ni cantidad de variables que pueda contener una estructura, mientras su máquina posea memoria suficiente como para alojarla, con una sola salvedad: una estructura no puede contenerse a sí misma como miembro.

Veamos como se definen y posteriormente comentaremos todos los aspectos relevantes de ellas.

```
struct [Nombre_de_la_estructura]
{
        tipo1 campo1;
        tipo2 campo2;
        .
        .
        tipoN campoN;
}       [variable];
```

La palabra clave **struct** define una estructura. Por tratarse de un tipo de datos puede utilizarse directamente para definir una variable. La variable aparece entre corchetes puesto que puede ser omitida. Si se especifica una variable, estaremos definiendo una variable cuyo tipo será la estructura que la precede. Si la variable no es indicada definimos un nuevo tipo de datos (struct Nombre_de_la_estructura), que podremos utilizar posteriormente. Si es el nombre de la estructura lo que se omite, tendremos que especificar obligatoriamente una variable que tendrá esa estructura y no podremos definir otras variables con esa estructura sin tener que volver a especificar todos los campos. Lo que se encuentra dentro de las llaves es una definición típica de variables con su tipo y su identificador. Todo esto puede parecer un poco confuso pero lo aclararemos con unos ejemplos.

Ejemplo 1: una compañía tiene por cada empleado los siguientes datos: Nombre (cadena), Direccion (cadena), Sexo (Carácter), Edad (entero), Antigüedad (entero), los datos del registro EMPLEADO serian (observe que cada miembro es de tipo diferente):

```
struct EMPLEADO
{
        char Nombre[30], Direccion[30], Sexo;
        int Edad, Antiguedad;
}       Emp1;
```

Ejemplo 2: Colección de discos compactos

```
struct CD
{
        char Titulo[30], Artista[30];
```

```
        int Numero_de_canciones;
        float Precio;
        char Fecha_de_compra[9];
}       Comprador1;
```

Ejemplo 3: Agenda

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

5.1.1 Manejo De Variables Tipo Registro

- Se declaran de la forma siguiente:

struct **Nombre_registro Variable_registro**

```
struct punto
{
        float x;
        float y;
        int color;
}       punto_de_fuga;
```

Aquí estamos definiendo una variable llamada punto_de_fuga cuyo tipo es una estructura de datos formada por tres campos y a la que hemos llamado punto. Dos de ellos son de tipo float y representan las coordenadas del punto, el tercer valor es un entero que indica el color de ese punto. En este caso hemos definido una variable y una estructura. Al disponer de un identificador para esta última podemos definir nuevas variables de esta estructura.

```
struct punto origen1;
struct punto final1;
```

Donde *origen1* y *final1* son variables de tipo **struct punto** que hemos definido anteriormente. Si en la definición de punto_de_fuga no se hubiese incluído un identificador para la estructura (en este caso el identificador es punto), no podríamos definir nuevas variables con esa estructura ya que no estaría identificada por ningún nombre.

También podríamos haber excluido el nombre de la variable (punto_de_fuga). En este caso lo que definiríamos sería una estructura llamada punto que pasaría a ser un nuevo tipo disponible por el usuario.

NOTA: En algunos compiladores es posible que se exija poner antes de struct la palabra *static*.

5.1.2 Definición Y Declaración A La Vez

Las variables del tipo de una estructura, pueden ser inicializadas en su definición, así por ejemplo se podría escribir:

```
struct Datos_Personales
{
```

```
        int edad;
        char nombre[50];
        float sueldo;
        char observaciones[500];
}       Vendedor1 = {40, "Juan Eneene", 1200.50,"Asignado a zona A"};

struct Datos_Personales Vendedor2 = {23, "Jose Perez",2000.0 , "Asignado a zona B" } ;
```

Acá se utilizaron las dos modalidades de definición de variables, inicializándolas a ambas.

5.1.3 Acceso A Los Campos De Una Variable De Tipo Registro

Lo que ahora nos interesa es saber como referenciar, acceder o modificar los campos de una variable tipo registro, por tanto la información que contienen. Para ello existe un operador que relaciona al nombre de ella con el de un miembro, este operador se representa con el punto (**.**), así se podrá referenciar a cada uno de los miembros como variables individuales, con las particularidades que les otorgan sus propias declaraciones, internas a la estructura.

La sintaxis para realizar ésta referencia es:

***Variable_registro . nombre_del_miembro*;**

Así podremos escribir por ejemplo, las siguientes sentencias:

```
struct posicion_de
{
        float eje_x ;
        float eje_y ;
        float eje_z ;
}       fin_recta, inicio_recta = { 1.0 , 2.0 , 3.0 );

fin_recta.eje_x = 10.0 ;
fin_recta.eje_y = 50.0 ;
fin_recta.eje_z = 90.0 ;
```

Es muy importante recalcar que dos estructuras, aunque sean del mismo tipo, no pueden ser asignadas ó comparadas la una con la otra, en forma directa, sino asignando ó comparándolas miembro a miembro. Esto se ve claramente explicitado en las líneas siguientes, basadas en las declaraciones anteriores:

```
fin_recta = inicio_recta ;          /* ERROR */

if( fin_recta >>= inicio_recta );      /* ERROR */

fin_recta.eje_x = inicio_recta.eje_x ;    /* FORMA CORRECTA DE ASIGNAR */

fin_recta.eje_y = inicio_recta.eje_y ;    /* UNA ESTRUCTURA A OTRA    */

fin_recta.eje_z = inicio_recta.eje_z ;

if( (fin_recta.eje_x >>= inicio_recta.eje_x) &&    /* FORMA CORRECTA DE  */

   (fin_recta.eje_y >>= inicio_recta.eje_y) &&    /* COMPARAR UNA      */

   (fin_recta.eje_z >>= inicio_recta.eje_z) )     /* ESTRUCTURA CON OTRA */
```

Las estructuras pueden anidarse, es decir que una ó más de ellas pueden ser miembro de otra. Las estructuras también pueden ser pasadas a las funciones como parámetros, y ser retornadas por éstas, como resultados.

Ejemplo: Realice un programa utilice un registro AGENDA para una agenda personal:

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

struct AGENDA
{
        char Nombre[20], Direccion[30];
        int Num_Telefono;
        int Edad;
}       Amigo1, Amigo2={"Juan", "Av. Lima  234", 96, 21};


int main(void)
{
        struct AGENDA Amigo3;

        /*Ingrese los datos de Amigo1 en el programa*/
        strcpy(Amigo1.Nombre,"Julio");
        strcpy(Amigo1.Direccion,"Av Arequipa 113");
        Amigo1.Num_Telefono = 571891;
        Amigo1.Edad = 23;

        /*Ingrese los datos de Amigo3 por teclado*/
        printf("Nombre: ");
        gets(Amigo3.Nombre);
        fflush( stdout );
        printf("Direccion: ");
        gets(Amigo3.Direccion);
        fflush( stdout );
        printf("Numero de telefono: ");
        scanf("%d",&Amigo3.Num_Telefono);
        printf("Edad: ");
        scanf("%d",&Amigo3.Edad);
        fflush( stdout );

        /*Imprimo en pantalla los datos que introduje por teclado*/
        printf("%s\n", Amigo3.Nombre);
        printf("%s\n",Amigo3.Direccion);
        printf("%d\n",Amigo3.Num_Telefono);
        printf("%d\n",Amigo3.Edad);

        getch();
        return 0;
}
```

Como vemos en el ejemplo anterior no podemos acceder a un campo tipo cadena como lo hacemos con por ejemplo con un campo tipo entero, para ello tenemos que utilizar la funcion ***strcpy*** que no hace otra cosa que copiar lo que nosotros queramos en el campo tipo cadena.

5.2 ARRAYS DE ESTRUCTURAS

Cuando hablamos de arrays dijimos que se podían agrupar para formar cualquier tipo de variables, esto es extensible a las estructuras y podemos entonces agruparlas ordenadamente, como elementos de un array. Veamos dos ejemplos:

```
struct Punto
{
        int x, y;
}       Array[10];
```

Aquí hemos definido un arreglo de 10 elementos, donde cada una de ellos es una estructura del tipo Punto, compuesta por dos variables enteras.

```
typedef struct
{
        char material[50] ;
        int existencia ;
        double costo_unitario ;
}       Item;

Item stock[100];
```

Hemos definido aquí un array de 100 elementos, donde cada uno de ellos es una estructura del tipo Item compuesta por tres variables, un int, un double y un string ó array de 50 caracteres.

Los arrays de estructuras pueden inicializarse de la manera habitual, así en una definición de stock, podríamos haber escrito:

```
Item stock1[100] =
        {
                "tornillos" , 120, .15 ,
                "tuercas"   , 200, .09 ,
                "arandelas" ,  90, .01
        };

Item stock2[] =
        {
                {{ 'i','t','e','m','1','\0' }, 10, 1.5 },
                {{ 'i','t','e','m','2','\0' }, 20, 1.0 },
                {{ 'i','t','e','m','3','\0' }, 60, 2.5 },
                {{ 'i','t','e','m','4','\0' }, 40, 4.6 },
                {{ 'i','t','e','m','5','\0' }, 10, 1.2 },
        } ;
```

Analicemos un poco las diferencias entre la dos inicializaciones dadas, en la primera, el array material[] es inicializado como un string, por medio de las comillas y luego en forma ordenada, se van inicializando cada uno de los miembros de los elementos del array stock1[], en la segunda se ha preferido dar valores individuales a cada uno de los elementos del array material, por lo que es necesario encerrarlos entre llaves.

Sin embargo hay una diferencia mucho mayor entre las dos sentencias, en la primera explicitamos el tamaño del array, [100], y sólo inicializamos los tres primeros elementos, los restantes quedarán cargados de basura si la definición es local a alguna función, ó de cero si es global, pero de cualquier manera están alojados en la memoria, en cambio en la segunda dejamos implícito el número de elementos, por lo que será el compilador el que calcule la cantidad de ellos, basándose en cuantos se han inicializado, por lo tanto este array sólo tendrá ubicados en memoria cuatro elementos, sin posibilidad de agregar nuevos datos posteriormente.

Veremos más adelante que en muchos casos es usual realizar un alojamiento dinámico de las estructuras en la memoria, en razón de ello, y para evitar además el saturación de stack por el pasaje ó retorno desde funciones, es necesario conocer el tamaño, ó espacio en bytes ocupados por ella. Podemos aplicar el operador sizeof, de la siguiente manera:

```
longitud_base_de_datos = sizeof ( stock1 ) ;

longitud_de_dato = sizeof ( Item ) ;

cantidad_de_datos = sizeof ( stock1 ) / sizeof ( Item ) ;
```

Con la primera calculamos el tamaño necesario de memoria para albergar a todos datos, en la segunda la longitud de un sólo elemento (record) y por supuesto dividiendo ambas, se obtiene la cantidad de records.

En el caso de la matriz tenemos tantas variables de tipo struct punto como las indicadas, puesto que el punto separa el nombre de la variable del campo al que queremos acceder, la forma de modificar una entrada de la matriz sería:

```
matriz_de_puntos[4].x = 6;

matriz_de_puntos.x[4] = 6; /* No sería correcto */
```

Esta última declaración se podría utilizar con una estructura de un tipo como:

```
struct otra
{
        float x[10];
}       matriz_de_puntos;
```

Con lo cual accederíamos al cuarto elemento del campo x de matriz_de_puntos que es una variable de tipo struct otra constituida por una matriz de diez floats.

EJEMPLO 1: Hacer un programa que lea dos puntos (Arreglo) representados como registros, calcular la longitud del segmento que los une y la pendiente de la recta que pasa por dichos puntos $P_1\ (x_1, y_1)$ y $P_2(x_2, y_2)$

$$Longitud = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

$$Pendiente = \frac{y_2 - y_1}{x_2 - x_1}$$

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

struct Punto
{
        int x, y;
}       A[2];

int main(void)
{
        int i;
        double Longitud, Pendiente;
```

```
        printf("Ingrese los dos puntos:\n");
        for(i=0; i<2;i++)
        {
                printf("\nPunto %d",i);
                printf("\nX: ");
                scanf("%d",&A[i].x);
                printf("Y: ");
                scanf("%d",&A[i].y);
        };
        Longitud = sqrt( pow((A[0].x - A[1].x),2) + pow((A[0].y - A[1].y),2)) ;
        Pendiente = (A[0].y - A[1].y) / (A[0].x - A[1].x);
        printf("La longitud de la recta es: %lf\n",Longitud);
        printf("La pendiente de la recta es: %lf\n",Pendiente);
        getch();
        return 0;
}
```

Ejemplo 2: Realizar un programa que almacene los datos de personas en una estructura llamada AGENDA.

```
#include <stdio.h>
#define ELEMENTOS 3

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

int main()
{
        int num_amigo;
        for( num_amigo=0; num_amigo<ELEMENTOS; num_amigo++ )
        {
                printf( "\nDatos del amigo número %i:\n", num_amigo+1 );
                printf( "Nombre: " ); fflush( stdout );
                gets(amigo[num_amigo].nombre);
                printf( "Apellido: " ); fflush( stdout );
                gets(amigo[num_amigo].apellido);
                printf( "Teléfono: " ); fflush( stdout );
                gets(amigo[num_amigo].telefono);
                printf( "Edad: " ); fflush( stdout );
                scanf( "%i", &amigo[num_amigo].edad );
                while(getchar()!='\n');
        }
        /* Ahora imprimimos sus datos */
        for( num_amigo=0; num_amigo<ELEMENTOS; num_amigo++ )
        {
                printf( "El amigo %s ", amigo[num_amigo].nombre );
                printf( "%s tiene ", amigo[num_amigo].apellido );
                printf( "%i años ", amigo[num_amigo].edad );
                printf( "y su teléfono es el %s.\n" , amigo[num_amigo].telefono );
```

```
        }
}
```

IMPORTANTE: Quizás alguien se pregunte qué pinta la línea esa de *while(getchar()!='\n');*. Esta línea se usa para vaciar el buffer de entrada.

5.3 Estructuras Anidadas

Es posible crear estructuras que tengan como miembros otras estructuras. Esto tiene diversas utilidades, por ejemplo tener la estructura de datos más ordenada.

Ejemplo 1: Una tienda de música quiere hacer un programa para el inventario de los discos, cintas y cd's que tienen. Para cada título quiere conocer las existencias en cada soporte (cinta, disco, cd), y los datos del proveedor (el que le vende ese disco). Podría pensar en una estructura así:

```
struct inventario
{
        char titulo[30];
        char autor[40];
        int existencias_discos;
        int existencias_cintas;
        int existencias_cd;
        char nombre_proveedor[40];
        char telefono_proveedor[10];
        char direccion_proveedor[100];
};
```

Sin embargo utilizando estructuras anidadas se podría hacer de esta otra forma más ordenada:

```
struct estruc_existencias
{
        int discos;
        int cintas;
        int cd;
        };

struct estruc_proveedor
{
        char nombre_proveedor[40];
        char telefono_proveedor[10];
        char direccion_proveedor[100];
        };

struct estruc_inventario
{
        char titulo[30];
        char autor[40];
        struct estruc_existencias existencias;
        struct estruc_proveedor proveedor;
}       inventario;
```

Ahora para acceder al número de cd de cierto título usaríamos lo siguiente:

```
inventario.existencias.cd
```

y para acceder al nombre del proveedor:

```
inventario.proveedor.nombre
```

Ejemplo 2: Crear registros anidados para trabajar con alguna figura geométrica.

```
struct vector
{
        float x;
        float y;
        float z;
};

struct poligono_cuadrado
{
        struct vector p1;
        struct vector p2;
        struct vector p3;
        struct vecto p4;
};

struct cubo
{
        struct poligono_cuadrado cara[6];
        int  color;
        struct vector  posicion;
};

struct cubo mi_cubo;
```

Hemos declarado una variable (mi_cubo) de tipo struct cubo que es una estructura conteniendo un valor entero que nos indica el color de nuestro objeto, una variable de tipo struct vector (posición) indicando la posición del objeto en un espacio de tres dimensiones (posición tiene tres campos x,y,z por tratarse de una struct vector) y una matriz de seis elemento en la que cada elemento es un struct poligono_cuadrado, el cual está formado por cuadro vectores que indican los cuatro vértices del cuadrado en 3D. Para acceder a todos los campos de esta variable necesitaríamos sentencias del tipo.

```
mi_cubo.color = 0;
mi_cubo.posicion.x = 3;
mi_cubo.posicion.y = 2;
mi_cubo.posicion.z = 6;
mi_cubo.cara[0].p1.x = 5;
/* Ahora acedemos a la coordenada 0 del tercer polígono de la cara 0 de mi_cubo*/
mi_cubo.cara[0].p3.z = 6;
 ....
```

Ejemplo 3: Realice un programa que saque por pantalla los datos de los empleados de una compañía sabiendo que tiene por cada empleado los siguientes campos:

CAMPO	TIPO DE DATO
Nombre	Cadena
Domicilio	Registro DIRECCION
Edad	Entero

Telefono	Entero

CAMPOS DE REGICTRO DIRECCION	TIPO DE DATO
Calle	Cadena
Numero	Entero
Ciudad	Cadena
Pais	Cadena

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

struct DIRECCION
{
	char Calle[20];
	int Numero;
	char Ciudad[20];
	char Pais[20];
};


struct EMPLEADO
{
	char Nombre[30];
	struct DIRECCION Domicilio;
	int Edad;
	int Telefono;
} EM1;

int main(void)
{
	strcpy(EM1.Nombre,"Joseph Paredes");
	strcpy(EM1.Domicilio.Calle,"Jr Arequipa");
	EM1.Domicilio.Numero=123;
	strcpy(EM1.Domicilio.Ciudad,"Arequipa");
	strcpy(EM1.Domicilio.Pais,"Peru");
	EM1.Edad=23;
	EM1.Telefono=571891;

	/*Imprimo en pantalla los datos del empleado EM1*/

	printf("\nNombre: %s",EM1.Nombre);
	printf("\nDomicilio:");
	printf("\n\tCalle: %s",EM1.Domicilio.Calle);
	printf("\n\tCalle: %d",EM1.Domicilio.Numero);
	printf("\n\tCalle: %s",EM1.Domicilio.Ciudad);
	printf("\n\tCalle: %s",EM1.Domicilio.Pais);
	printf("\nEdad: %d",EM1.Edad);
	printf("\nTelefono: %d",EM1.Telefono);
	getch();
}
```

5.4 UNIONES

La definición de una union es analoga a la definición de una estructura. La diferencia entre ambas es que los campos que especifiquemos en una union ocupan todos la misma posicion de memoria. Cuando se declara una union se reserva espacio para poder almacenar el campo de mayor tamaño de los declarados y como ya se dijo todos los campos ocupan la misma posición en la memoria. Veamos un ejemplo.

```
union ejemplo
{
        char   caracter;
        int    entero;
}       mi_var;
```

mi_var es una variable cuyo tipo es union ejemplo, y el acceso a cada campo de los definidos se realiza igual que en las struct mediante la utilización de un punto. Hasta aquí nada nuevo lo que sucede es que carácter y entero (los dos campos) ocupan la misma posición de memoria. Así:

```
mi_var.entero = 0; /* Como el tipo int ocupa más que el tipo char ponemos a 0 toda la
union */
mi_var.caracter = 'A'; /* El código ASCII de A es 65, por tanto ahora mi_var.entero = 65 */
mi_var.entero = 0x00f10;
```

Esta última instrucción introduce un valor en hexadecimal en la variable mi_var.entero. El código hexadecimal se representa en **C** anteponiendo al número los caracteres 0x. Para comprender lo que realiza esta instrucción veamos un poco como el ordenador representa los números internamente.

Todos hemos oido alguna vez que el ordenador sólo entiende ceros y unos, pues bien, lo único que significa ésto es que el ordenador cuenta en base dos en lugar de hacerlo en base diez como nosotros. Cuando contamos en base diez comenzamos en 0 y al llegar a nueve añadimos una unidad a la izquierda para indicar que llegamos a las centenas y así consecutivamente. Cada cifra de un número en base diez representa esa cifra multiplicada por una potencia de diez que depende de la posición del dígito. Es lo que se llama descomposición factorial de un número.

63452 = 6*10^4+3*10^3+4*10^2+5*10^1+2*10^0 = 60000+3000+400+50+2

Como nuestro ordenador en lugar de contar de diez en diez cuenta de dos en dos cada cifra es una potencia de dos. El sistema de numeración en base dos se denomina sistema binario.

b100101 = 1*2^5+0*2^4+0*2^3+1*2^2+0*2^1+1*2^0 = 32 + 0 + 0 + 4 + 1 = 37

Así es como representa el ordenador el número 37 en su sistema binario. Cada una de las cifras de un número binario se denomina BIT (BInary digiT) y los ordenadores los suelen agrupar el grupos de 8. Así 8 bits se denomina un byte, 16bits serían 2 bytes y se denomina word o palabra y así sucesivamente.

El mayor número que podríamos representar en binario con 1 byte (8bits) sería:

b11111111 = 255

Este es el tamaño que el lenguaje **C** asigna al tipo char, que sólo puede representar 256 valores distintos, desde 0 a 255. El tipo int short suele ocupar una palabra es decir, 16 bits. Así con 16 bits el mayor número que podemos representar es:

b1111111111111111 = 65535

NOTA: El tamaño asociado a cada tipo de datos es muy específico de cada compilador/ordenador. No debería darse nada por supuesto...

Los números en binario rápidamente se hacen muy largos por ello se utilizan otros sistemas de numeración que permitan una escritura más compacta sin perter la información binaria en gran medida. Esto sistemas son en general sistemas con bases que son potencias de dos. Así tenemos el sistema octal (base 8) y el sistema hexadecimal (base 16). Este último es el más ampliamente usado, disponemos de 16 cifras de 0 a F(15) y la característica más importante de este sistema es que cada cifra hexadecimal, representa cuatro bits binarios, con lo cual el paso de un sistema al otro es extremadamente fácil.

Volvamos ahora a la instrucción anteriormente indicada

```
mi_var.entero = 0x00f10;
```

Si pasamos este número a binario obtenemos:

```
0 -> 0000
f -> 1111 -> 15 en decimal
1 -> 0001 -> 1 en decimal
0f10 <-> 0000111100010000 -> 3856 en decimal
```

Como dijimos anteriormente un char ocupa 8 bits y un int ocupa 16, como la union los solapa tendríamos un esquema en la memoria del ordenador como éste:

```
int     0000111100010000    -> 3856
char        00010000    -> 65 ('A')
```

Así mi_var.caracter contendrá el valor A, pero mi_var.entero contendrá el valor 3856.

NOTA: Como ya se indicó en la nota anterior, el tamaño asignado a cada tipo depende del ordenador y del compilador. Además, algunos ordenadores almacenan los números en formato Bajo/Alto (los 8 bits e Intel) y otros en formato Alto/Bajo (Motorola, Sparc, etc.).

Este tipo de estructura se suele utilizar en aplicaciones a bajo nivel en la que es necesario poder utilizar este tipo de solapamiento de bits. Como ya se habrá podido comprobar para comprender mínimamente como funciona esto es necesario bajar mucho al nivel de la máquina con la consiguiente complicación de la explicación.

5.5 Creacion de tipos: typedef

Esta palabra reservada del lenguaje C sirve para la creación de nuevos nombres de tipos de datos. Mediante esta declaración es posible que el usuario defina una serie de tipos de variables propios, no incorporados en el lenguaje y que se forman a partir de tipos de datos ya existentes. Por ejemplo, la declaración:

```
typedef int ENTERO;
```

define un tipo de variable llamado ENTERO que corresponde a int.

Como ejemplo más completo, se pueden declarar mediante typedef las siguientes estructuras:

```
#define MAX_NOM 30
#define MAX_ALUMNOS 400
struct s_alumno
{       /* se define la estructura s_alumno*/
        char nombre[MAX_NOM];
        short edad;
};
typedef struct s_alumno ALUMNO; /* ALUMNO es un nuevo tipo de variable*/
typedef struct s_alumno *ALUMNOPTR;
```

```
struct clase
{
        ALUMNO alumnos[MAX_ALUMNOS];
        char nom_profesor[MAX_NOM];
};
typedef struct clase CLASE;
typedef struct clase *CLASEPTR;
```

Con esta definición se crean las cuatro palabras reservadas para tipos, denominadas ALUMNO (una estructura), ALUMNOPTR (un puntero a una estructura), CLASE y CLASEPTR. Ahora podría definirse una función del siguiente modo:

```
int anade_a_clase(ALUMNO un_alumno, CLASEPTR clase)
{
        ALUMNOPTR otro_alumno;
        otro_alumno = (ALUMNOPTR) malloc(sizeof(ALUMNO));
        otro_alumno->edad = 23;
        ...
        clase->alumnos[0]=alumno;
        ...
        return 0;
}
```

El comando typedef ayuda a parametrizar un programa contra problemas de portabilidad. Generalmente se utiliza typedef para los tipos de datos que pueden ser dependientes de la instalación. También puede ayudar a documentar el programa (es mucho más claro para el programador el tipo ALUMNOPTR, que un tipo declarado como un puntero a una estructura complicada), haciéndolo más legible.

CAPITULO VI:

Funciones

6.1 Introduccion

Hasta el momento hemos utilizado ya numerosas funciones, como printf o scanf, las cuales forman parte de la librería estándar de entrada/salida (stdio.h). Sin embargo el lenguaje C nos permite definir nuestras propias funciones, es decir, podemos añadir al lenguaje tantos comandos como deseemos.

Las funciones son básicas en el desarrollo de un programa cuyo tamaño sea considerable, puesto que en este tipo de programas es común que se repitan fragmentos de código, los cuales se pueden incluir en una función con el consiguiente ahorro de memoria. Por otra parte el uso de funciones divide un programa de gran tamaño en subprogramas más pequeños (las funciones), facilitando su comprensión, así como la corrección de errores.

6.1.1 Programacion estructurada

La Programación Estructurada pretende evitar la edición de programas en un solo bloque (monolíticos), dificiles de manejar por su longitud. Algunos autores aconsejan que el cuerpo de una función debería ser visible en una sola pantalla.

La Programación Estructurada es una estrategia de resolución de problemas y una metodología de programación que incluye dos grandes líneas:

- El control de flujo en un programa debería ser tan simple como fuera posible.
- La construcción de un programa debería hacerse empleando la técnica del diseño descendente (top-down).

6.1.2 Diseño Top-down

El método de diseño Top-Down, también llamado método de refinamientos sucesivos o divide y vencerás, consiste en descomponer repetidamente un problema en problemas más pequeños. Es decir:

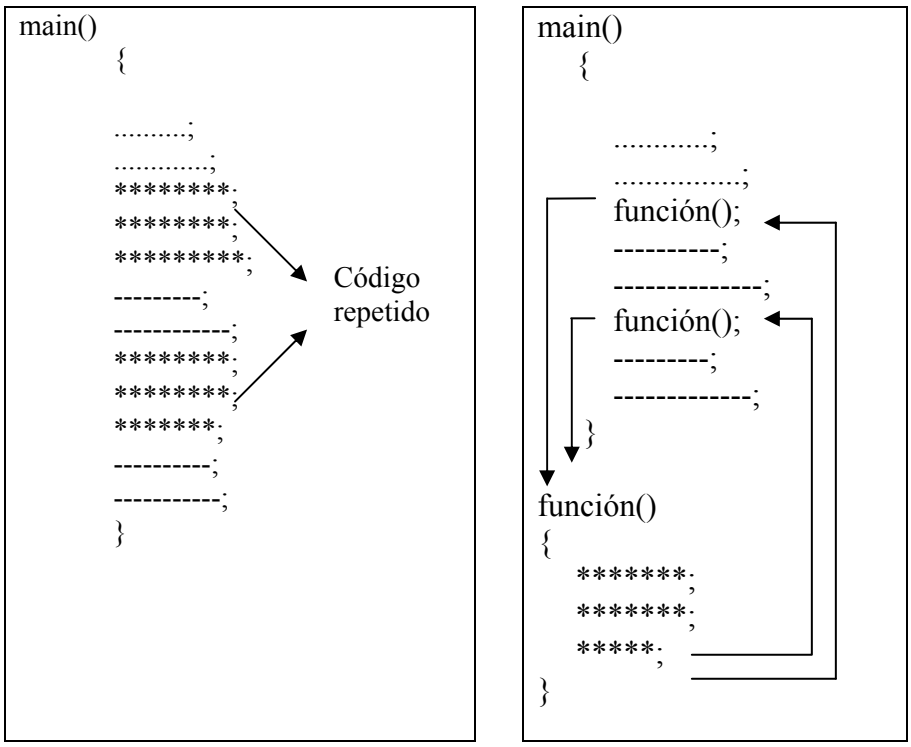
- Construir un programa a partir de pequeñas piezas o componentes.
- Esas pequeñas piezas son llamadas módulos.
- Cada pieza es más manejable que el programa original

6.2 Definición de funciones

Una función es una entidad formada por un grupo de sentencias destinadas a hacer un trabajo concreto. Los objetivos básicos de la utilización de funciones son tres:

a. Evitar repeticiones de código

Supongamos que en uno de sus programas tiene una sección de código destinado a calcular la potencia de un número. Si después, en el mismo programa necesita calcular la potencia de otro número, seria deseable, en lugar de escribir otra vez las sentencias que calculan la potencia, poder saltar hacia la sección que calcula la potencia y volver, una vez hecho el cálculo, hacia el flujo normal del programa,. Observe el esquema siguiente, en el cuadro de su izquierda se observa que hay código repetido (podría ser el código que calcula la potencia de un número), mientras que en el cuadro de su izquierda el código solo aparece una vez, cuando este es requerido se llama y punto.



b. Organización del programa

La utilización de funciones permite dividir el programa en actividades. Cada función realiza una tarea determinada, y cada función puede comprobarse, más o menos, de forma independiente, facilitando así el diseño, la comprensión de los programas, y la detección de errores.

Independencia

Utilizar funciones independientes del programa principal y entre si, permite su reutilización en diferentes programas. En el ejemplo de calcular la potencia de un número, se podría coger la función que hace esta operación, y incluirla en cualquier programa que la necesite sin tenerla que escribir cada vez.

6.3 Estructura de las funciones

Para poder utilizar una función, se han de definir tres elementos en el programa:

- Definición de la función.
- Prototipo de la función.
- Llamada a la función.

a. Definición de función

Se llama definición de función a la función misma. La función siempre empieza con una línea que incluye, entre otros elementos, el nombre de la función, en el ejemplo anterior,

```
void borrar (void)
```

Esta línea se llama declarador de la función. El void inicial significa que la función no devuelve nada, y el void que va entre el paréntesis, que la función no recibe ningún parámetro; es una función sin argumentos. Se verán las funciones que devuelven valores y los argumentos, un poco más adelante.

Fíjese que el declarador de función no acaba en punto y coma, no es una instrucción, sirve para indicar al compilador que en este punto empieza la definición de la función. A continuación del declarador de la función se encuentra el cuerpo de la función, que son todas las líneas de código que incluye.

b. Prototipo de la función

Es la línea que hay antes de la función main()

```
void borrar (void);
```

El prototipo (también llamado declaración de función), sirve para declarar la función, de la misma manera que se han de declarar les variables antes de poder ser utilizadas. Esta línea le indica al compilador el nombre de la función, el tipo de dato que devuelve (ninguno en el ejemplo), y el tipo de argumentos que se le pasan (ninguno en el ejemplo), de esta manera cuando el compilador llegue a las líneas que llaman a la función, comprobará que esta llamada sea sintácticamente correcta (que el tipo de función y el tipo y número de argumentos coincidan con la declaración del prototipo).

El prototipo está escrito antes de la función main(), esto provoca que esta función pueda ser llamada desde todas les funciones del programa.

Observe que la línea del prototipo acaba con punto y coma.

c. Llamar a la función

Para llamar a la función sonido se utiliza la misma sintaxis que cuando es llama cualquier función de C, como por ejemplo printf(), getche() o clrscr(), es decir, el nombre de la función seguido de paréntesis. Estos son necesarios para que el compilador sepa que nos referimos a una función. La llamada a una función es una instrucción o sentencia, por tanto ha de acabar en punto y coma.

6.4 Declaracion de funciones

Una función se declara con el nombre de la función precedido del tipo de valor que retorna y una lista de argumentos encerrados entre paréntesis. El cuerpo de la función está formado por un conjunto de declaraciones y de sentencias comprendidas entre llaves.

```
tipo_a_devolver identificador (tipo1 parámetro1, tipo2 ...)
{
        tipo1 Variable_Local1;
        tipo2 Variable_Local2;
        ...

        Código de la función

        return valor del tipo valor a devolver;
}
```

Lo primero con lo que nos encontramos es la cabecera de la función. Esta cabecera está formada por una serie de declaraciones. En primer lugar el tipo_a_devolver.

Todas las funciones tienen la posibilidad de devolver un valor, aunque pueden no hacerlo. Si definimos una función que nos calcula el coseno de un cierto ángulo nos interesaría que nuestra función devolviese ese valor. Si por el contrario nuestra función realiza el proceso de borrar la pantalla no existiría ningún valor que nos interesase conocer sobre esa función. Si no se especifica ningún parámetro el compilador supondrá que nuestra función devuelve un valor entero (int).

A continuación nos encontramos con el identificador de la función, es decir, el nombre con el que la vamos a referenciar en nuestro programa, seguido de una ***lista de parámetros formales*** entre paréntesis y separados por comas sobre los que actuará el código que escribamos para esa función. En el caso de la función coseno a la que antes aludíamos, el parámetro sería el ángulo calculamos el coseno de un cierto ángulo que en cada llamada a la función probablemente sea distinto. Véase la importancia de los parámetros, si no pudiésemos definir un parámetro para nuestra función coseno, tendríamos que definir una función para cada ángulo, en la que obviamente no indicaríamos ningún parámetro.

A continuación nos encontramos el cuerpo de la función. En primer lugar declaramos las variables locales de esa función. Estas variables solamente podrán ser accedidas dentro de la función, esto es, entre las llaves ({}). Los nombres de las variables locales pueden ser los mismos en distintas funciones puesto que sólo son accesibles dentro de ellas. Así si estamos acostumbrados a utilizar una variable entera llamada i como contador en nuestros bucles, podemos definir en distintas funciones esta variable y utilizarla dentro de cada función sin que haya interferencias entre las distintas funciones.

Con respecto al código de la función, pues simplemente se trata de un programa como todos los que hemos estado haciendo hasta ahora.

La instrucción return del final puede omitirse si la función no devuelve ningún valor, su cometido es simplemente indicar que valor tomaría esa función con los parámetros que le hemos pasado. Debe ser del mismo tipo que tipo_a_devolver. En otros lenguajes las funciones que no devuelven valores se conocen como procedimientos.

6.5 Dónde se definen las funciones

Las funciones deben definirse antes de ser llamadas. Las funciones deben definirse siempre antes de donde se usan. Lo habitual en un programa es:

Sección	Descripción
Includes	Aquí se indican qué ficheros externos se usan
Definiciones	Aquí se definen las constantes que se usan en el programa
Definición de variables	Aquí se definen las variables globales (las que se pueden usar en TODAS las funciones)
Definición de funciones	Aquí es donde se definen las funciones
Función main	Aquí se define la función main.

Esta es una forma muy habitual de estructurar un programa. Sin embargo esto no es algo rígido, no tiene por qué hacerse así, pero es recomendable.

Se puede hacer de otra forma, también aconsejable. Consiste en definir después de las variables las cabeceras de las funciones, sin escribir su código. Esto nos permite luego poner las funciones en cualquier orden.

Ejemplos de declaración de funciones:

```
int fact_i ( int v );
int mayor ( int a, int b );
int cero ( double a );
long raiz ( long valor );
void final_countdown ( void );
int main ( int argc, char **argv );
```

Observando el prototipo de una función podemos decir exactamente que tipo de parámetros necesita y que resultado devuelve. Si una función tiene como argumento void, quiere decir que no tiene argumentos, al igual que si el resultado es void, no devuelve ningún valor.

En la vieja definición de Kernighan y Ritchie el tipo que devolvía una función se declaraba únicamente si era distinto de int. Similarmente, los parámetros eran declarados en el cuerpo de la función, en lugar de utilizar la lista de parámetros. Por ejemplo:

```
mayor ( a, b )
int a;
int b;

{
        Código de la función
}
```

Las funciones al viejo estilo se compilan correctamente en muchos compiladores actuales. Por contra, proporcionan menos información sobre sus parámetros y errores que afecten al tipo de parámetros de llamada a las funciones no pueden ser detectados automáticamente.

6.6 Parametros, Argumentos

Una función puede ser llamada con cero o más argumentos.

En el prototipo de función:

```
int func(int x, double y, char c);
```

La lista de parámetros formales (variables y sus tipos)

En la llamada a la función:

```
valor = func(edad, puntuacion, letra);
```

La lista de parámetros actuales (sin su tipo)

6.6.1 Normas para escribir la lista de argumentos formales

- El número de parámetros formales y actuales debe ser coincidente.
- La asociación de argumentos es posicional: el primer parametro actual se corresponde con el primer parámetro formal, el segundo con el segundo, etc…
- Los parámetros actuales y formales deben ser tipos de datos compatibles.
- Los argumentos actuales pueden ser variables, constantes o cualquier expresión del tipo de dato correspondiente al parámetro formal.
- Las llamadas pueden ser por valor o por referencia.

6.6.2 Paso de parámetros por valor

- Cada argumento es evaluado, y su valor es usado localmente en el lugar del correspondiente parámetro formal.
- Si se pasa una variable a una función, el valor de esta variable almacenado en el entorno de la llamada no cambia. (la variable transferida si es alterada en la funcion llamada, no afecta al valor original Al llamar por valor, el valor de la variable que queremos transferir es copiado en el parámetro actual.

Ejemplo 1:

```
#include <stdio.h>

void imprime_mensaje (int k);        /*funcion  prototipo */

int main (void)
{
        int n;
        printf("Tengo un mensaje para tí.\n");
        printf("¿Cuantas veces quieres verlo?: ");
        scanf("%d", &n);
        imprime_mensaje( n );        /*Llamada a la funcion*/
        return 0;
}

void  imprime_mensaje(int k) /* definición de funcion */
{
        int i;
        printf("\nEste es tu mensaje:\n");
        for (i=0; i < k; ++i)
                printf("En el fondo del mar...\n");

}
```

Parámetros, Argumentos formales

Parámetros, Argumentos actuales

Ejemplo 2: Función sin argumentos que no devuelve nada: Este programa llama a la función prepara pantalla que borra la pantalla y muestra el mensaje "la pantalla está limpia". Por supuesto es de nula utilidad pero nos sirve para empezar.

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

void prepara_pantalla()    /* No se debe poner punto y coma aquí */
{
        clrscr();
        printf( "La pantalla está limpia\n" );
        return;    /* No hace falta devolver ningún valor, mucha gente ni siquiera pone este
return */
        }

int main()
{
        prepara_pantalla();/* Llamamos a la función */
}
```

Ejemplo 3: Función con argumentos, no devuelve ningún valor: En este ejemplo la función compara toma dos números, los compara y nos dice cual es mayor.

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

```
void compara( int a, int b )     /* Metemos los parámetros a y b a la función */
{
        if ( a>b )
                printf( "%i es mayor que %i\n" , a, b );
        else
                printf( "%i es mayor que %i\n", b, a );
        return;
}

int main()
{
        int num1, num2;

        printf( "Introduzca dos números: " );
        scanf( "%i %i", &num1, &num2 );

        compara( num1, num2 );/* Llamamos a la función con sus dos argumentos */
}
```

Ejemplo 4: Función con argumentos que devuelve un valor. Este ejemplo es como el anterior pero devuelve como resultado el mayor de los dos números.

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

int compara( int a, int b )     /* Metemos los parámetros a y b a la función */
{
        int mayor;     /* Esta función define su propia variable, esta variable sólo se puede
usar aquí */
        if ( a>b )
                mayor = a;
        else
                mayor = b;
        return mayor;
}

int main()
{
        int num1, num2;
        int resultado;

        printf( "Introduzca dos números: " );
        scanf( "%i %i", num1, num2 );

        resultado = compara( num1, num2 ); /* Recogemos el valor que devuelve la función
en resultado */
        printf( "El mayor de los dos es %i\n", resultado );
}
```

En este ejemplo podíamos haber hecho también:

```
printf( "El mayor de los dos es %i\n", compara( num1, num2 ) );
```

De esta forma nos ahorramos tener que definir la variable 'resultado'.

Cuando se define la cabecera de la función sin su cuerpo (o código) debemos poner un ';' al final. Cuando definamos el cuerpo más tarde no debemos poner el ';', se hace como una función normal.

La definición debe ser igual cuando definimos sólo la cabecera y cuando definimos el cuerpo. Mismo nombre, mismo número y tipo de parámetros y mismo tipo de valor devuelto.

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

void compara( int a, int b );     /* Definimos la cabecera de la función */

int main()
{
        int num1, num2;
        int resultado;

        printf( "Introduzca dos números: " );
        scanf( "%i %i", num1, num2 );

        resultado = compara( num1, num2 );
        printf( "El mayor de los dos es %i\n", resultado );
}

int compara( int a, int b )     /* Ahora podemos poner el cuerpo de la función donde
queramos. */
          /* Incluso después de donde la llamamos (main) */
{
        int mayor;
        if ( a>b )
                mayor = a;
        else
                mayor = b;
        return mayor;
}
```

6.7 Paso de un array a una función

6.7.1 Paso de vectores

Como es sabido mediante parámetros se pueden pasar valores de distintas variables a funciones. se puede utilizar un vector como parámetro de una función, pero... copie el programa siguiente, ejecúteo y vea los resultados:

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

void sumardos(int tabla[],int Var) ;

main()
```

```
{
        int tabla[10] ;
        int Var;
        int i ;
        clrscr() ;
        for (i=0 ; i < 10 ; i++)
                tabla[i] = i+1 ;
        Var=10;
        sumardos(tabla,Var) ;

        printf("\n\n %d",a);

        for (i=0 ; i <= 10 ; i++)
                printf("Elemento %d és %d ", i, tabla[i]) ;
}

void sumardos(int tabla[],int Var)
{
        int i ;
        for (i=0 ; i < 10 ; i++)
                tabla[i] = tabla[i] + 2 ;
        Var=Var+2;
}
```

El programa entra los valores del 1 al 10 en array tabla en el primer bucle for.

El programa asigna el valor 10 a la variable Var.

Se pasa array tabla y variable Var a función sumados. En esta función se suma 2 a cada elemento del array y a la variable Var.

Una vez que el programa ha vuelto a la función main, imprime el valor 10 de la variable, la cual cosa no es sorprendente puesto que, tal y como recordamos, la variable Var de función main, pasa el valor a la variable Var de función sumardos(). Lo que si ha cambiado son los valores del array; cada elemento vale dos unidades más que el valor establecido antes de llamar a la función suma.

Al pasar un array a función, no se crea otro array en memoria , tal y como pasaba con las variables, sino que se trabaja con el mismo array. Para comprender qué pasa en realidad, primero se deben conocer mejor los punteros. Por el momento basta con tener en cuenta que siempre que se pasa un array a función, se trabaja sobre el array pasado.

Tanto en la declaración de la función, como en su prototipo, el argumento array se identifica con [], sin ningún valor dentro.

6.7.2 Paso de matrices

Se especifican de la manera siguiente:

```
tipo_función nombre(tipo_datos nombre_array[][Num Columnas]) ;
```

Haciendo una función para entrar valores en el array Alumnos:

```
void entrar_valores(int Alumnos[][3]) ;
```

6.8 Paso de estructuras a funciones

Las estructuras se pueden pasar directamente a una función igual que hacíamos con las variables. Por supuesto en la definición de la función debemos indicar el tipo de argumento que usamos:

```
int nombre_función ( struct nombre_de_la_estructura nombre_de_la variable_estructura )
```

En el ejemplo siguiente se usa una función llamada *suma* que calcula cual será la edad 20 años más tarde (simplemente suma 20 a la edad). Esta función toma como argumento la variable estructura *arg_amigo*. Cuando se ejecuta el programa llamamos a *suma* desde *main* y en esta variable se copia el contenido de la variable *amigo*.

Esta función devuelve un valor entero (porque está declarada como int) y el valor que devuelve (mediante return) es la suma.

```
#include <stdio.h>

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

struct estructura_amigo amigo =
        {
                "Juanjo",
                "Lopez",
                "592-0483",
                30
        };

int suma( struct estructura_amigo arg_amigo )
{
        return arg_amigo.edad+20;
}

int main()
{
        printf( "%s tiene ", amigo.apellido );
        printf( "%i años ", amigo.edad );
        printf( "y dentro de 20 años tendrá %i.\n", suma(amigo) );
        return 0;
}
```

Si dentro de la función *suma* hubiésemos cambiado algún valor de la estructura, dado que es una copia no hubiera afectado a la variable *amigo* de main. Es decir, si dentro de 'suma' hacemos arg_amigo.edad = 20; el valor de arg_amigo cambiará, pero el de *amigo* seguirá siendo 30.

Ejemplo: Programa de gestion de alumnos de una clase

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

```
#define ALUMNOS 20

struct Alumno
{
	char Nombre[20];
	int Edad;
	int Nota;
};

void EntrarDatos(struct Alumno Clase[]);
void OrdenarPorNotas(struct Alumno Clase[]);
void MostrarDatos(struct Alumno Clase[]);

int main()
{
	struct Alumno Clase[ALUMNOS];
	EntrarDatos(Clase);
	OrdenarPorNotas(Clase);
	MostrarDatos(Clase);
	return 0;
}


void EntrarDatos(struct Alumno Clase[])
{
	int y;
	for (y=0; y < ALUMNOS; y++)
	{
		flushall();
		printf("Alumno %d\n", y+1);
		puts("Nombre : ");
		gets(Clase[y].Nombre);
		puts("Edad : ");
		scanf("%d", &Clase[y].Edad);
		puts("Nota: ");
		scanf("%d", &Clase[y].Nota);
	}
}

void OrdenarPorNotas(struct Alumno Clase[])
{
	struct Alumno ClaseAux;
	int y, k, posmenor;
	for (y=0; y< ALUMNOS; y++)
	{
		posmenor = y;
		ClaseAux = Clase[y];
		for (k=y+1; k < ALUMNOS; k++)
		{
			if (Clase[k].Nota < ClaseAux.Nota)
			{
				ClaseAux = Clase[k];
```

```
                                posmenor = k;
                        }
                }
                Clase[posmenor] = Clase[y];
                Clase[y] = ClaseAux;
        }
}

void MostrarDatos(struct Alumno Clase[])
{
        int y;
        for (y = 0; y < ALUMNOS; y++)
        {
                printf("Alumno %d \n ", y + 1);
                puts("Nombre : ");
                printf("%s \n",Clase[y].Nombre);
                puts("Edad : ");
                printf("%d \n ", Clase[y].Edad);
                puts("Nota : ");
                printf("%d \n", Clase[y].Nota);
        }
}
```

6.9 Pasar sólo miembros de la estructura

Otra posibilidad es no pasar toda la estructura a la función sino tan sólo los miembros que sean necesarios. Los ejemplos anteriores serían más correctos usando esta tercera opción, ya que sólo usamos el miembro 'edad':

```
int suma( char edad )
{
        return edad+20;
}

int main()
{
        printf( "%s tiene ", amigo.apellido );
        printf( "%i años ", amigo.edad );
        printf( "y dentro de 20 años tendrá %i.\n", suma(amigo.edad) );
}
```

Por supuesto a la función suma hay que indicarle que va a recibir una variable tipo char (amigo->edad es de tipo char).

6.10 Ambito De Funciones Y Variables

El ámbito, o visibilidad, de una variable nos indica en que lugares del programa está activa esa variable. Hasta ahora, en los ejemplos que hemos visto, se han utilizado variables definidas en el cuerpo de funciones. Estas variables se crean en la memoria del ordenador cuando se llama a la función y se destruyen cuando se sale. Es necesario a veces, que una variable tenga un valor que pueda ser accesible desde todas las funciones de un mismo fuente, e incluso desde otros fuentes.

En C, el ámbito de las variables depende de dónde han sido declaradas y si se les ha aplicado algún modificador. Una variable definida en una función es, por defecto, una variable local. Esto es, que sólo existe

y puede ser accedida dentro de la función. Para que una variable sea visible desde una función cualquiera del mismo fuente debe declararse fuera de cualquier función. Esta variable sólo será visible en las funciones definidas después de su declaración. Por esto, el lugar más común para definir las variables globales es antes de la definición de ninguna función. Por defecto, una variable global es visible desde otro fuente. Para definir que existe una variable global que está definida en otro fuente tenemos que anteponer la palabra extern a su declaración. Esta declaración únicamente indica al compilador que se hará referencia a una variable externa al módulo que se compila.

Las variables locales llevan implícito el modificador auto. Esto es que se crean al inicio de la ejecución de la función y se destruyen al final. En un programa sería muy ineficiente en términos de almacenamiento que se crearan todas las variables al inicio de la ejecución. Por contra, en algunos casos es deseable. Esto se consigue anteponiendo el modificador static a una variable local. Si una función necesita una variable que únicamente sea accedida por la misma función y que conserve su valor a través de sucesivas llamadas, es el caso adecuado para que sea declarada local a la función con el modificador static. El modificador static se puede aplicar también a variables globales. Una variable global es por defecto accesible desde cualquier fuente del programa. Si, por cualquier motivo, se desea que una de estas variables no se visible desde otro fuente se le debe aplicar el modificador static. Lo mismo ocurre con las funciones. Las funciones definidas en un fuente son utilizables desde cualquier otro. En este caso conviene incluir los prototipos de las funciones del otro fuente. Si no se desea que alguna función pueda ser llamada desde fuera del fuente en la que está definida se le debe anteponer el modificador static.

```
void contar ( void )
{
        static long     cuenta = 0;
        cuenta++;
        printf("Llamada %ld veces\n", cuenta );
}
```

Otro modificador muy importante es const. Con él se pueden definir variables cuyo valor debe permanecer constante durante toda la ejecución del programa. También se puede utilizar con argumentos de funciones. En esta caso se indica que el argumento en cuestión es un parámetro y su valor no debe ser modificado. En el caso que por error modifiquemos ese parámetro, el compilador nos indicará el error.

```
#define EULER       2.71828
const double   pi = 3.14159;

double lcercle ( const double r )
{
        return 2 * pi * r;
}

double EXP ( const double x )
{
        return pow ( EULER, x );
}

double sinh ( const double x )
{
        return (exp(x) - exp(-x)) / 2;
}
```

Debemos fijarnos que en el ejemplo anterior pi es una variable, la cual no podemos modificar. Por ello pi sólo puede aparecer en un único fuente. Si la definimos en varios, al linkar el programa se nos generará un error por tener una variable duplicada.

Otro modificador utilizado algunas veces es el register. Este modificador es aplicable únicamente a variables locales e indica al compilador que esta debe ser almacenada permanentemente en un registro del procesador del ordenador. Este modificador es herencia de los viejos tiempos, cuando las tecnologías de optimización de código no estaban muy desarrolladas y se debía decir qué variable era muy utilizada en la función. Hoy en día casi todos los compiladores realizan un estudio de qué variables locales son las más adecuadas para ser almacenadas en registros, y las asignan automáticamente. Con los compiladores modernos se puede dar el caso de que una declaración register inadecuada disminuya la velocidad de ejecución de la función, en lugar de aumentarla. Por ello, hoy en día, la utilización de este modificador está en desuso, hasta el punto de que algunos compiladores lo ignoran. Se debe tener en cuenta que de una variable declarada como register no se puede obtener su dirección, ya que está almacenada en un registro y no en memoria.

6.10.1 Variables locales

Veamo el siguiente ejemplo:

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

void sonido(int );

main()
{
        int veces;
        clrscr();
        sonido(2);
        printf("Entre el número de veces que ha de pitar : ");
        scanf("%d",&veces);
        sonido(veces);
}

void sonido(int veces)
{
        int i;
        for (i=1 ; y <= veces ; i++)
                printf("\x7");
}
```

La variable i definida dentro de la función sonido sólo se conoce dentro de esta función, no se puede utilizar en la función main().

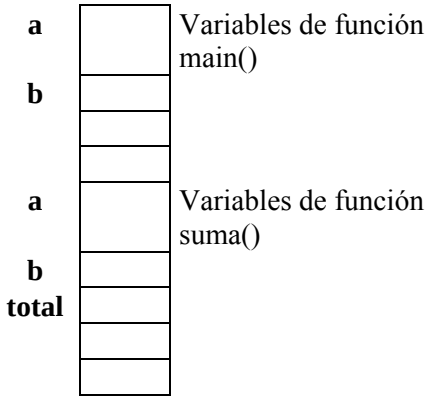
La variable veces definida en main() sólo es conocida en main(). La variable veces definida como argumento de la función sonido, sólo es conocida dentro de la función sonido. Evidentemente, la variable veces declarada en main() no es la misma variable que veces declarada como argumento de función sonido.

Una variable sólo se conoce en la función donde se declara, y sólo puede utilizarse en esta función. Vea el siguiente ejemplo donde se utiliza una función para sumar dos números:

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

void suma (int a, int b);

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

```
        clrscr();
        suma(3, 5);
        printf("Entre el primer valor a sumar : ");
        scanf("%d",&a);
        printf("Entre el segundo valor a sumar : ");
        scanf("%d",&b);
        suma(a,b);
}

void suma (int a, int b)
{
        int total;
        total = a + b;
        printf("El valor de la suma es : %d \n ", total);
}
```

Esquema de memoria



En la función main() se han declarado dos variables a y b, y en la función suma se han declarado tres a, b y total; a y b son además variables que reciben valores externos. Insistimos en que a y b declaradas en main() no son las mismas variables que a y b declaradas en suma, vea el esquema de memoria.

Funcionamiento del programa

Cuando el programa llega a la sentencia suma(3,5), se pasa a ejecutar la función suma.

La variable a de función suma recibe el valor 3.
La variable b de función suma recibe el valor 5.
Se ejecuta la función suma y se vuelve a main().
Se ejecutan las sentencias printf y scanf en las cuales se entran valores desde teclado, que se guardan en les variables a y b de función main().

El programa llega a la sentencia suma(a, b), y se pasa otra vez a ejecutar la función suma.

La variable a de función suma recibe el valor de la variable a de función main.
La variable b de función suma recibe el valor de la variable b de función main.

Se ejecuta la función suma y se vuelve a main().
Finaliza el programa.

6.10.2 Variables externas o globales. Ámbito de una variable.

Ya se ha comentado que las variables declaradas dentro de una función, solo se pueden utilizar dentro de esta función. EL ámbito de las variables locales es la función donde se declaran.

Una variable externa o global, es aquella que se declara fuera de cualquier función (normalmente antes de la sentencia main()); esta variable es visible desde todas las funciones, todas pueden utilizarla. El ámbito de estas variables es todo el programa.

Copie el programa siguiente y ejecútelo para acabar de comprender los conceptos global, local y ámbito de una variable.

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

void prueba(int a, int b);

int global;                    // Esta variable la conocen todas las funciones

main()
{
        int a, b;          // Variables locales a main, solo conocida en función  main.
        a = 10;
        b = 20;
        global = 20;
        prueba(a, b);
        printf("%d", a);        // Escribe 10
        printf("%d", b);        // Escribe 20
        printf("%d", global); // Escribe 31, el valor que se le ha dado en función prueba.
}

void prueba(int Pepe, int Paco)
{
        printf("%d", Pepe);   // Escribe 10 que es el valor que recibe de variable a de main
        printf("%d", Paco);   // Escribe 20 que es el valor que recibe de variable b de main
        Pepe = 5;             // Asigna 5 a variable local Pepe
        Paco = 5;             // Asigna 5 a variable local Paco
        global = 31;          // Asigna 31 a la variable global
}
```

6.11 Recursividad

Es el proceso de definir algo en términos de si mismo, es decir que las funciones pueden llamarse a si mismas, esto se consigue cuando en el cuerpo de la función hay una llamada a la propia función, se dice que es recursiva. Una función recursiva no hace una nueva copia de la función, solo son nuevos los argumentos.

La principal ventaja de las funciones recursivas es que se pueden usar para crear versiones de algoritmos más claras y sencillas. Cuando se escriben funciones recursivas, se debe tener una sentencia if para forzar a la función a volver sin que se ejecute la llamada recursiva.

Ejemplo:

```
#include <stdio.h>

int fact(int numero)
{
```

```
        int resp;
        if(numero==1)
                return 1;
        resp=fact(numero-1)*numero;
        return(resp);
}


int main()
{

        int num;
        printf("Deme un numero y le dare el factorial: ");
        scanf("%d",&num);
        printf("%d\n", fact(num));
        return 0;
}
```

6.12 La función main

Método para pasar información a la función main mediante el uso de argumentos, el lugar desde donde se pasan esos valores es la línea de ordenes del sistema operativo. Se colocan detrás del nombre del programa. Veamos cuáles son sus argumentos y cómo se declaran:

```
int main( int argc, char *argv[] )
int main ( int argc, char ** argv)
int main ( int argc, char argv[][])
```

El primer argumento es argc (argument count). Es de tipo int e indica el número de argumentos que se le han pasado al programa.

El segundo es argv (argument values). Es un array de strings (o puntero a puntero a char). En el se almacenan los parámetros. Normalmente (como siempre depende del compilador) el primer elemento (argv[0]) es el nombre del programa con su ruta. El segundo (argv[1]) es el primer parámetro, el tercero (argv[2]) el segundo parámetro y así hasta el final.

Si la función main espera argumentos y no se le pasa ninguno desde la línea de ordenes, es muy posible que de un error de ejecución cuando se intenten utilizar esos argumentos. Por tanto lo mejor es siempre controlar que el número de argumentos es correcto.

Otro aspecto a tener en cuenta es que el nombre de los dos argumentos (argc y argv) son tradicionales pero arbitrarios, es decir que se les puede dar el nombre que a nosotros nos interese, manteniendo eso si el tipo y el número de argumentos que recibe.

Ejemplo 1: Muestra todos los parámetros de un programa:

```
#include<stdio.h>

int main(int argc,char *argv[])
{
        int i;
        for( i=0 ; i<argc ; i++ )
                printf( "Argumento %i: %s\n", i, argv[i] );
}
```

Si por ejemplo llamamos al programa argumentos.c y lo compilamos (argumentos.exe) podríamos teclear (lo que está en negrita es lo que tecleamos):

```
c:\programas> argumentos hola amigos
```

Tendríamos como salida:

```
Argumento 0: c:\programas\argumentos.exe
Argumento 1: hola
Argumento 2: amigos
```

Pero si en vez de eso tecleamos:

```
c:\programas> argumentos "hola amigos"
```

Lo que tendremos será:

```
Argumento 0: c:\programas\argumentos.exe
Argumento 1: hola amigos
```

CAPITULO VII:

Punteros

7.1 Introduccion

Cada variable de un programa tiene una dirección en la memoria del ordenador. Esta dirección indica la posición del primer byte que la variable ocupa. En el caso de una estructura es la suma del tamaño de cada uno de sus campos. Como en cualquier caso las variables son almacenadas ordenadamente y de una forma predecible, es posible acceder a estas y manipularlas mediante otras variables que contenga su dirección. A este tipo de variables se les denomina punteros.

En C, los punteros son el tipo más potente y seguramente la otra clave del éxito del lenguaje. La primera ventaja que obtenemos de los punteros es la posibilidad que nos dan de poder tratar con datos de un tamaño arbitrario sin tener que moverlos por la memoria. Esto puede ahorrar un tiempo de computación muy importante en algunos tipos de aplicaciones. También permiten que una función reciba y cambie el valor de una variable. Recordemos que todas las funciones C únicamente aceptan parámetros por valor. Mediante un puntero a una variable podemos modificarla indirectamente desde una función cualquiera.

Los punteros nos permites acceder directamente a cualquier parte de la memoria, esto da a los programas **C** una gran potencia. Sin embargo son una fuente ilimitada de errores. Un error usando un puntero puede bloquear el sistema (si usamos ms-dos o win95, no en Linux) y a veces puede ser difícil detectarlo. Otros lenguajes no nos dejan usar punteros para evitar estos problemas, pero a la vez nos quitan parte del control que tenemos en C. A pesar de todo esto no hay que tenerles miedo. Casi todos los programas C usan punteros. Si aprendemos a usarlos bien no tendremos más que algún problema esporádico.

7.2 La memoria del ordenador

Cuando hablamos de memoria nos estamos refiriendo a la memoria RAM del ordenador. Son unas pastillas que se conectan a la placa base y nada tienen que ver con el disco duro. El disco duro guarda los datos permanentemente (hasta que se rompe) y la información se almacena como ficheros. Nosotros podemos decirle al ordenador cuándo grabar, borrar, abrir un documento, etc. La memoria Ram en cambio, se borra al apagar el ordenador. La memoria Ram la usan los programas sin que el usuario de éstos se de cuenta.

Para hacernos una idea, hoy en dia la memoria se mide en MegaBytes (suelen ser 16, 32, 64, 128Mb) y los discos duros en GigaBytes (entre 3,4 y 70Gb, o mucho más).

Hay otras memorias en el ordenador aparte de la mencionada. La memoria de video (que está en la tarjeta gráfica), las memorias caché (del procesador, de la placa...) y quizás alguna más que ahora se me olvida.

7.3 Direcciones de variables

Vamos a ir como siempre por partes. Primero vamos a ver qué pasa cuando declaramos una variable.

Al declarar una variable estamos diciendo al ordenador que nos reserve una parte de la memoria para almacenarla. Cada vez que ejecutemos el programa la variable se almacenará en un sitio diferente, eso no lo podemos controlar, depende de la memoria disponible y otros factores misteriosos. Puede que se almacene en el mismo sitio, pero es mejor no fiarse. Dependiendo del tipo de variable que declaremos el ordenador nos reservará más o menos memoria. Como vimos en el capítulo de tipos de datos cada tipo de variable ocupa más o menos bytes. Por ejemplo si declaramos un char, el ordenador nos reserva 1 byte (8 bits). Cuando finaliza el programa todo el espacio reservado queda libre.

Existe una forma de saber qué direcciones nos ha reservado el ordenador. Se trata de usar el operador & (operador de dirección). Vamos a ver un ejemplo: Declaramos la variable 'a' y obtenemos su valor y dirección.

```
#include <stdio.h>

int main()
{
        int a;

        a = 10;
        printf( "Dirección de a = %p, valor de a = %i\n", &a, a );
}
```

Para mostrar la dirección de la variable usamos %p en lugar de %i, sirve para escribir direcciones de punteros y variables. El valor se muestra en hexadecimal.

No hay que confundir el valor de la variable con la dirección donde está almacenada la variable. La variable 'a' está almacenada en un lugar determinado de la memoria, ese lugar no cambia mientras se ejecuta el programa. El valor de la variable puede cambiar a lo largo del programa, lo cambiamos nosotros. Ese valor está almacenado en la dirección de la variable. El nombre de la variable es equivalente a poner un nombre a una zona de la memoria. Cuando en el programa escribimos 'a', en realidad estamos diciendo, "el valor que está almacenado en la dirección de memoria a la que llamamos 'a'".

7.4 Punteros

Básicamente, un puntero en C es una variable numérica, que ocupa 4 bytes (en entornos de sistemas de 32 bits, tales como Windows o Linux). Su valor es tan solo un valor, igual que el valor de cualquier otra variable entera que se haya declarado. La diferencia fundamental entre una variable "normal" y un puntero es el uso que hace de ella el programador y el compilador cuando traduce las expresiones en las que se usan.

Los punteros en el Lenguaje C, son variables que "apuntan", es decir que poseen la dirección de las ubicaciones en memoria de otras variables, y por medio de ellos tendremos un poderoso método de acceso a todas ellas.

Quizás este punto es el más conflictivo del lenguaje, ya que muchos programadores en otros idiomas, y novatos en C, lo ven como un método extraño ó al menos desacostumbrado, lo que les produce un cierto rechazo. Sin embargo, y en la medida que uno se va familiarizando con ellos, se convierten en la herramienta más cómoda y directa para el manejo de variables complejas, argumentos, parámetros, etc, y se empieza a preguntar como es que hizo para programar hasta aquí, sin ellos. La respuesta es que no lo ha hecho, ya que los hemos usado en forma encubierta, sin decir lo que eran.

Un puntero se declara de la forma:

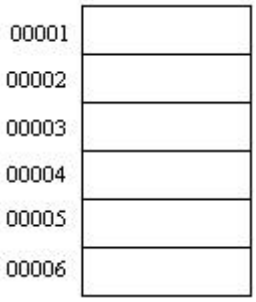
```
Tipo_de_dato  *nombre_del_puntero;
```

Ejemplos:

```
int *p
float    *pf;
PLANETA     *pp;
char     *pc;
```

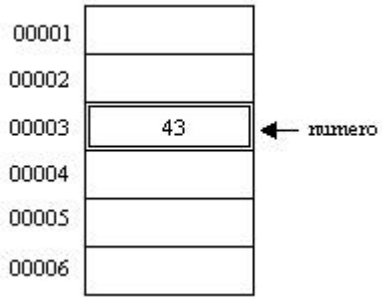
NOTA: Es muy importante entender **p** no es una varaible puntero de tipo integer, sino una variable de tipo puntero a integer.

Vamos a ver si cogemos bien el concepto de puntero y la diferencia entre éstos y las variables normales.



En el dibujo anterior tenemos una representación de lo que sería la memoria del ordenador. Cada casilla representa un byte de la memoria. Y cada número es su dirección de memoria. La primera casilla es la posición 00001 de la memoria. La segunda casilla la posición 00002 y así sucesivamente.

Supongamos que ahora declaramos una variable char: char numero = 43. El ordenador nos guardaría por ejemplo la posición 00003 para esta variable. Esta posición de la memoria queda reservada y ya no la puede usar nadie más. Además esta posición a partir de ahora se le llama numero. Como le hemos dado el valor 43 a numero, el valor 43 se almacena en numero, es decir, en la posición 00003.



Si ahora usáramos el programa anterior:

```
#include <stdio.h>

int main()
{
        int numero;
        numero = 43;
        printf( "Dirección de numero = %p, valor de numero = %i\n", &numero, numero );
}
```

El resultado sería:

Dirección de numero = 00003, valor de numero = 43

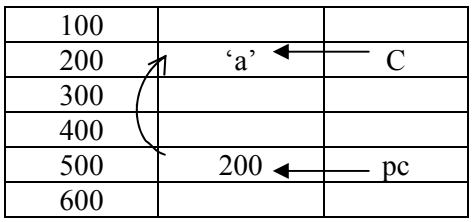
Creo que así ya está clara la diferencia entre el valor de una variable (43) y su dirección (00003).

Si declaramos un puntero:

```
char    *pc;
```

'pc' es una variable que contendrá la dirección de memoria en la que se aloja un char;

dirección	Contenido	Variable



Lo que representa el esquema anterior, bien podría ser el siguiente caso:

```
{
        char c = 'a';
        char *p;
        p = &c;
}
```

Para manipular un puntero, como variable que es, se utiliza su nombre; pero para acceder a la variable a la que apunta se le debe preceder de *. A este proceso se le llama indirección. Accedemos indirectamente a una variable. Para trabajar con punteros existe un operador, &, que indica 'dirección de'. Con él se puede asignar a un puntero la dirección de una variable, o pasar como parámetro a una función.

El operador &, es un operador unario, es decir, actúa sobre un sólo operando. Este operando tiene que ser obligatoriamente una estructura direccionable, es decir, que se encuentre en la memoria del ordenador. Estas estructuras son fundamentalmente las variables y las funciones, de las que hablaremos posteriormente. Decimos que sólo se puede aplicar sobre estructuras direccionables porque su función es devolver la posición de memoria en la que se encuentra dicha estructura. En nuestro ejemplo nos indicaría cual sería el compartimiento en el que se encuentra el informe que le indiquemos.

El segundo operador es el *. También se trata de un operador unario como el anterior y su función en este caso es la de permitir el acceso al contenido de la posición indicada por un puntero. En nuestro ejemplo el operador * nos permitiría leer o escribir el informe al que apunta uno de nuestros compartimientos punteros. Además el carácter * se utiliza para declarar punteros.

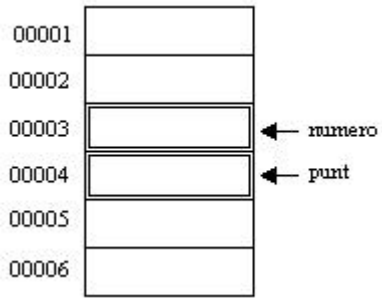
Por supuesto el operador * debe ser aplicado sobre un puntero, mientras que el operador & sobre una estructura direccionable (variable o función).

Ahora vamos un poco más allá, vamos a declarar un puntero. Hemos dicho que un puntero sirve para almacenar la direcciones de memoria. Muchas veces los punteros se usan para guardar las direcciones de variables. Vimos en el capítulo Tipos de Datos que cada tipo de variable ocupaba un espacio distinto. Por eso cuando declaramos un puntero debemos especificar el tipo de datos cuya dirección almacenará. En nuestro ejemplo queremos que almacene la dirección de una variable int. Así que para declarar el puntero punt debemos hacer:

```
#include <stdio.h>

int main()
{
        int numero;
        int *punt;
        numero = 43;
        punt = &numero;
        printf( "Dirección de numero = %p, valor de numero = %i\n", &numero, numero );
}
```

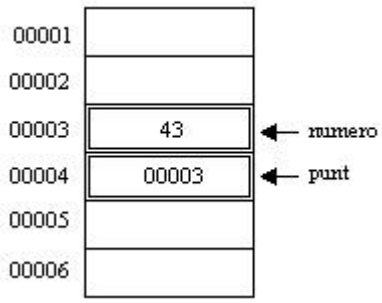
Vamos a ir línea a línea:

- En la primera int numero reservamos memoria para numero (supongamos que queda como antes, posición 00003). Por ahora numero no tiene ningún valor.
- Siguiente línea: int *punt;. Reservamos una posición de memoria para almacenar el puntero. Lo normal es que según se declaran variables se guarden en posiciones contiguas. De modo que quedaría en la posición 00004. Por ahora punt no tiene ningún valor, es decir, no apunta a ninguna variable. Esto es lo que tenemos por ahora:



- Tercera línea: numero = 43;. Aquí ya estamos dando el valor 43 a numero. Se almacena 43 en la dirección 00003, que es la de numero.

- Cuarta línea: punt = №. Por fin damos un valor a punt. El valor que le damos es la dirección de numero (ya hemos visto que & devuelve la dirección de una variable). Así que punt tendrá como valor la dirección de numero, 00003. Por lo tanto ya tenemos:



Cuando un puntero tiene la dirección de una variable se dice que ese puntero apunta a esa variable.

EJEMPLO 2:

```
main ()
{
        int x,y;  /* Variables de tipo entero */
        int *px;    /* Puntero a una variable de tipo entero */

        /* Leemos la dirección -compartimiento- de la variable
          -informe- x mediante & y lo almacenamos en la variable puntero
          px */

        px = &x;

        /* px contiene la dirección en la que se encuentra x */
        /* Utilizando el operador *, podemos acceder a su información.
          *px representa ahora el valor de la variable x */

        *px = 10;          /* Ahora x contiene el valor 10 */
        y = 15;
```

```
        /* Si ahora hacemos que nuestro puntero apunte a la
        variable y utilizando de nuevo el operador & */

        px = &y;
        /* El valor que ahora toma *px será el valor de y puesto
        que es el compartimiento al que ahora estamos apuntando */

        *px = 125; /* Ahora y contiene el valor 125 */
        x = *px    /* Ahora x contiene también 125 */
}
```

Como hemos visto en este ejemplo es exactamente igual acceder a una variable que utilizar un puntero que apunte a ella (hacemos que apunte a ella mediante el operador &) junto con el operador *.

7.5 Para qué sirve un puntero y cómo se usa

Los punteros tienen muchas utilidades, por ejemplo nos permiten pasar argumentos (o parámetros) a una función y modificarlos. También permiten el manejo de cadenas y de arrays. Otro uso importante es que nos permiten acceder directamente a la pantalla, al teclado y a todos los componenetes del ordenador. Pero esto ya lo veremos más adelante.

Pero si sólo sirvieran para almacenar direcciones de memoria no servirían para mucho. Nos deben dejar también la posibilidad de acceder a esas posiciones de memoria. Para acceder a ellas se usa el operador *, que no hay que confundir con el de la multiplicación.

```
#include <stdio.h>
int main()
{
        int numero;
        int *punt;
        numero = 43;
        punt = &numero;
        printf( "Dirección de numero = %p, valor de numero = %i\n", &numero, *punt );
}
```

Si nos fijamos en lo que ha cambiado con respecto al ejemplo anterior, vemos que para acceder al valor de número usamos *punt en vez de numero. Esto es así porque punt apunta a numero y *punt nos permite acceder al valor al que apunta punt.

```
#include <stdio.h>
int main()
{
        int numero;
        int *punt;
        numero = 43;
        punt = &numero;
        *punt = 30;
        printf( "Dirección de numero = %p, valor de numero = %i\n", &numero, numero );
}
```

Ahora hemos cambiado el valor de numero a través de *punt.

En resumen, usando punt podemos apuntar a una variable y con *punt vemos o cambiamos el contenido de esa variable.

Un puntero no sólo sirve para apunta a una variable, también sirve para apuntar una dirección de memoria determinada. Esto tiene muchas aplicaciones, por ejemplo nos permite controlar el hardware directamente (en MS-Dos y Windows, no en Linux). Podemos escribir directamente sobre la memoria de video y así escribir directamente en la pantalla sin usar printf.

7.6 Usando punteros en una comparación

Veamos el siguiente ejemplo. Queremos comprobar si dos variables son iguales usando punteros:

```
#include <stdio.h>
int main()
{
        int a, b;
        int *punt1, *punt2;
        a = 5; b = 5;
        punt1 = &a; punt2 = &b;
        if ( punt1 == punt2 )
        printf( "Son iguales\n" );
}
```

Alguien podría pensar que el if se cumple y se mostraría el mensaje Son iguales en pantalla. Pues no es así, el programa es erróneo. Es cierto que a y b son iguales. También es cierto que punt1 apunta a 'a' y punt2 a 'b'. Lo que queríamos comprobar era si a y b son iguales. Sin embargo con la condición estamos comprobando si punt1 apunta al mismo sitio que punt2, estamos comparando las direcciones donde apuntan. Por supuesto a y b están en distinto sitio en la memoria así que la condición es falsa. Para que el programa funcionara deberíamos usar los asteriscos:

```
#include <stdio.h>
int main()
{
        int a, b;
        int *punt1, *punt2;
        a = 5; b = 5;
        punt1 = &a; punt2 = &b;
        if ( *punt1 == *punt2 )
        printf( "Son iguales\n" );
}
```

Ahora sí. Estamos comparando el contenido de las variables a las que apuntan punt1 y punt2. Debemos tener mucho cuidado con esto porque es un error que se cuela con mucha facilidad.

Vamos a cambiar un poco el ejemplo. Ahora 'b' no existe y punt1 y punt2 apuntan a 'a'. La condición se cumplirá porque apuntan al mismo sitio.

```
#include <stdio.h>

int main()
{
        int a;
        int *punt1, *punt2;

        a = 5;
```

```
        punt1 = &a; punt2 = &a;

        if ( punt1 == punt2 )
        printf( "punt1 y punt2 apuntan al mismo sitio\n" );
}
```

7.7 Punteros a vectores

Ya hemos hablado de los vectores en temas anteriores. Se trataba de un conjunto de un número de terminado de variables de un mismo tipo que se referenciaban con un nombre común seguido de su posición entre corchetes con relación al primer elemento. Todas las entradas de un vetor están consecutivas en memoria, por eso es muy sencillo acceder al elemento que queramos en cada momento simplemente indicando su posición. Sólo se le suma a la posición inicial ese índice que indicamos.

Un array y un puntero son básicamente la misma cosa. Un array es un puntero allocado de forma estática en su propia declaración. De hecho, una variable no es más que un array de dimensión 1.

Los vectores son realmente punteros al inicio de una zona consecutiva de los elementos indicados en su declaración, por lo cual podemos acceder a la matriz utilizando los corchetes como ya vimos o utilizando el operador *.

```
elemento[i] <=> *(elemento +i)
```

Como ya se ha comentado todos los punteros ocupan lo mismo en memoria, el espacio suficiente para contener una dirección, sin embargo cuando se declaran es necesario indicar cual es el tipo de datos al que van a apuntar (entero, real, alguna estructura definida por el usuario). En nuestro ejemplo tendríamos un tipo de puntero por cada tipo de informe distinto, un puntero para informes de una página, otro puntero para informes de 2 páginas y así sucesivamente. En principio esto es irrelevante por que una dirección de memoria es una dirección de memoria, independientemente de lo que contenga con lo cual no sería necesario declarar ningún tipo, pero esta información es necesaria para implementar la aritmética de punteros que ejemplificaremos a continuación.

Para que un puntero apunte a un array se puede hacer de dos formas, una es apuntando al primer elemento del array:

```
int *puntero;
int temperaturas[24];
puntero = &temperaturas[0];
```

El puntero apunta a la dirección del primer elemento. Otra forma equivalente, pero mucho más usada es:

```
puntero = temperaturas;
```

Con esto también apuntamos al primer elemento del array. Fijate que el puntero tiene que ser del mismo tipo que el array (en este caso int).

Ahora vamos a ver cómo acceder al resto de los elementos. Para ello empezamos por cómo funciona un array: Un array se guarda en posiciones seguidas en memoria, de tal forma que el segundo elemento va inmediatamente después del primero en la memoria. En un ordenador en el que el tamaño del tipo int es de 32 bits (4 bytes) cada elemento del array ocupará 4 bytes. Veamos un ejemplo:

```
#include <stdio.h>

int main()
{
        int i;
```

```
        int temp[24];

        for( i=0; i<24; i++ )
        {
                printf( "La dirección del elemento %i es %p.\n", i, &temp[i] );
        }
}
```

NOTA: Recuerda que %p sirve para imprimir en pantalla la dirección de una variable en hexadecimal.

El resultado es (en mi ordenador):

```
La dirección del elemento 0 es 4c430.
La dirección del elemento 1 es 4c434.
La dirección del elemento 2 es 4c438.
La dirección del elemento 3 es 4c43c.
...
La dirección del elemento 21 es 4c484.
La dirección del elemento 22 es 4c488.
La dirección del elemento 23 es 4c48c.
```

(Las direcciones están en hexadecimal). Vemos aquí que efectivamente ocupan posiciones consecutivas y que cada una ocupa 4 bytes. Si lo representamos en una tabla:

4C430	4C434	4C438	4C43C
temp[0]	**temp[1]**	**temp[2]**	**temp[3]**

Ya hemos visto cómo funcionas los arrays por dentro, ahora vamos a verlo con punteros. Voy a poner un ejemplo:

```
#include <stdio.h>

int main()
{
        int i;
        int temp[24];
        int *punt;

        punt = temp;

        for( i=0; i<24; i++ )
        {
        printf( "La dirección de temp[%i] es %p y la de punt es %p.\n",
            i, &temp[i], punt );
        punt++;
        }
}
```

Cuyo resultado es:

```
La dirección de temp[0] es 4c430 y la de punt es 4c430.
La dirección de temp[1] es 4c434 y la de punt es 4c434.
La dirección de temp[2] es 4c438 y la de punt es 4c438.
```

...
La dirección de temp[21] es 4c484 y la de punt es 4c484.
La dirección de temp[22] es 4c488 y la de punt es 4c488.
La dirección de temp[23] es 4c48c y la de punt es 4c48c.

En este ejemplo hay dos líneas importantes (en negrita). La primera es punt = temp. Con esta hacemos que el punt apunte al primer elemento de la matriz. Si no hacemos esto punt apunta a un sitio cualquiera de la memoria y debemos recordar que no es conveniente dejar los punteros así, puede ser desastroso.

La segunda línea importante es punt++. Con esto incrementamos el valor de punt, pero curiosamente aunque incrementamos una unidad (punt++ equivale a punt=punt+1) el valor aumenta en 4. Aquí se muestra una de las características especiales de los punteros. Recordemos que en un puntero se guarda una dirección. También sabemos que un puntero apunta a un tipo de datos determinado (en este caso int). Cuando sumamos 1 a un puntero sumamos el tamaño del tipo al que apunta. En el ejemplo el puntero apunta a una variable de tipo int que es de 4 bytes, entonces al sumar 1 lo que hacemos es sumar 4 bytes. Con esto lo que se consigue es apuntar a la siguiente posición int de la memoria, en este caso es el siguiente elemento de la matriz.

Operación	Equivalente	Valor de punt
punt = temp;	punt = &temp[0];	4c430
punt++;	sumar 4 al contenido de punt (4c430+4)	4c434
punt++;	sumar 4 al contenido de punt (4c434+4)	4c438

Cuando hemos acabado estamos en temp[24] que no existe. Si queremos recuperar el elemento 1 podemos hacer punt = temp otra vez o restar 24 a punt:

```
punt -= 24;
```

con esto hemos restado 24 posiciones a punt (24 posiciones int*4 bytes por cada int= 96 posiciones).

Vamos a ver ahora un ejemplo de cómo recorrer la matriz entera con punteros y cómo imprimirla:

```
#include <stdio.h>
int main(int argc,char *argv[])
{
        int temperaturas[24] = { 15, 18, 20, 23, 22, 24, 22, 25, 26, 25, 24,
                        22, 21, 20, 18, 17, 16, 17, 15, 14, 14, 13, 12, 12 };

        int *punt;
        int i;
        punt = temperaturas;

        for( i=0 ; i<24; i++ )
        {
                printf( "Elemento %i: %i\n", i, *punt );
                punt++;
        }
}
```

Cuando acabamos punt apunta a temperaturas[24], y no al primer elemento, si queremos volver a recorrer la matriz debemos volver como antes al comienzo. Para evitar perder la referencia al primer elemento de la matriz (temperaturas[0]) se puede usar otra forma de recorrer la matriz con punteros:

```
#include <stdio.h>

int main(int argc,char *argv[])
```

```
{
        int temperaturas[24] = { 15, 18, 20, 23, 22, 24, 22, 25, 26, 25, 24,
                        22, 21, 20, 18, 17, 16, 17, 15, 14, 14, 13, 12, 12 };

        int *punt;
        int i;

        punt = temperaturas;

        for( i=0 ; i<24; i++ )
                printf( "Elemento %i: %i\n", i, *(punt+i) );
        return 0;
}
```

Con *(punt+i) lo que hacemos es tomar la dirección a la que apunta punt (la dirección del primer elemento de la matriz) y le sumamos i posiciones. De esta forma tenemos la dirección del elemento i. No estamos sumando un valor a punt, para sumarle un valor habría que hacer punt++ o punt+=algo, así que punt siempre apunta al principio de la matriz.

Se podría hacer este programa sin usar punt. Sustituyendolo por temperaturas y dejar *(temperaturas+i). Lo que no se puede hacer es: temperaturas++;.

Importante: Como final debo comentar que el uso de índices es una forma de maquillar el uso de punteros. El ordenador convierte los índices a punteros. Cuando al ordenador le decimos temp[5] en realidad le estamos diciendo *(temp+5). Así que usar índices es equivalente a usar punteros de una forma más cómoda.

7.7.1 Inicializacion de vectores con notacion de punteros

El lenguaje C, permite hacer asignaciones como las siguientes:

```
int Notas[]={10, 12, 21, 44, 56};
int *Notas = {10, 12, 21, 44, 56};
```

Inicia el array Notas con 5 elementos. En el primer caso, Notas es una constante puntero, en el segundo caso, es una variable puntero.

```
char *Palabra="Hola";
```

Inicia la variable puntero Palabra con cinco caracters, los corresponents a Hola más el '\0' del final.

7.8 Punteros a matrices

¿Es, lo anterior, generalizable a arrays de más dimensiones?. La respuesta es SI.

Un array bidimensional es, en realidad, un conjunto de dos arrays monodimensionales (vectores). Por esto, podemos definir un array bidimensional como un puntero a un grupo de arrays monodimensionales contiguos. Veamos esto:

El array:

```
tipo_dato array[indice1][indice2];
```

puede ser declarado como:

```
tipo_dato (*array)[indice2]; /*puntero a un grupo de arrays */
```

Importante: en la declaración anterior desaparece la primera dimensión. Los paréntesis de (*array)[indice2]; deben estar presentes. Sin ellos definiríamos un array de punteros (Ver el siguiente punto) en lugar de un puntero a un grupo de arrays.

Ejemplo:

```
int x[5][10];  /* array bidimensional con 5 filas y 10 columnas*/
```

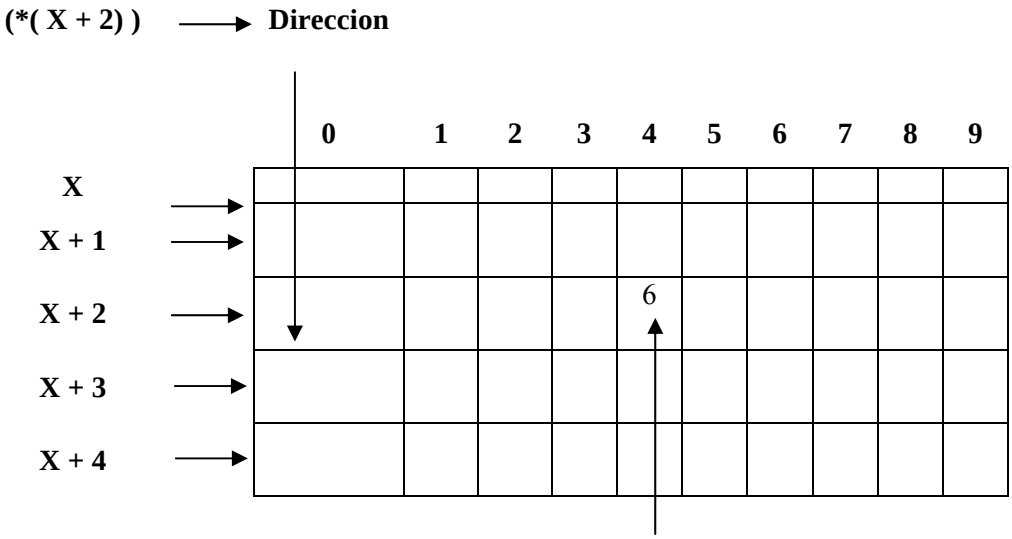
Equivale a:

```
int (*x)[10];  /*Puntero a un grupo de vectores de 10 elementos cada uno*/
```

Realmente, x apunta al primero de los vectores de 10 elementos (primera fila (fila 0) del array bidimensional original)

Del mismo modo (x + 1) apunta al segundo vector de 10 elementos (segunda fila del array bidimensional original). Entonces generalizando **(X + i)** representa la dirección de memoria de cada una de las filas

La expresión (*(x+0) + 0) es lo mismo que &x[0][0] (dirección).
La expresión *(*(x+0) +0) es lo mismo que x[0][0] (contenido).



CONTENIDO	DIRECCION
(*(x+2)[4]) vale 6	(*(x+2)+4)
X[2][4] vale 6	&X[2][4]
((x+2)+4) vale 6	

Ejemplo1: Declara un array bidimensional de 2x5. Tratandolo como un puntero a dos arrays monodimensionales de tamaño 5, imprime la dirección de memoria del inicio de cada fila (cada array monodimensional) de cada elemento y los contenidos con notación de puntero doble y como puntero a un grupo de arrays contiguos: valor[2][5] se puede reepresentar como (*valor)[5] en donde valor es un puntero a dos arrays monodimensionales de 5 elemntos cada uno (*valor+0) es un puntero a la primera fila de 5 elementos (*valor+1) es un puntero a la segunda fila de 5 elementos

```
#include <conio.h>
#include <stdio.h>
int main()
{
	int valor[][5]= {3, -6, 0, 9, 15, 4, 12, 7, 23, 1}, i ,j;
	/*direcciones de las filas:    */
	puts("Direcciones de las filas:");
	for(i=0; i<2; i++)
		printf("%p  ", *(valor+i));

	/*direcciones de cada elemento  */
	puts("\nDirecciones de cada elemento:");
	for(i=0; i<2; i++)
	{
		for(j=0; j<5; j++)
		printf("%p  ", (*(valor+i)+j));
		printf("\n");
	}

	/*Contenido de cada elemento*/

	puts("\nContenidos de cada elemento (como puntero doble):");
	for(i=0; i<2; i++)
	{
		for(j=0; j<5; j++)
			printf("%d  ", *(*(valor+i)+j));
		printf("\n");
	}

	/*Contenidos como puntero a grupo de arrays contiguos*/

	puts("\nContenidos de cada elemento (como puntero a un grupo de arrays
contiguos:");
	for(i=0; i<2; i++)
	{
		for(j=0; j<5; j++)
			printf("%d  ", (*(valor+i))[j]);
		printf("\n");
	}
	getch();
}
```

Ejemplo 2:

```
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
void main()
{
	static int i, j;
	int mat[2][5] ={10, 9, 8, 7, 6, 5, 4, 3, 2, 1};
```

```
        puts("Imprimimos como array:\n");
        for(i=0; i<2; i++)
                for(j=0; j<5; j++)
                        printf("\nmat[%d][%d] = %d", i, j, mat[i][j]);
        getch();

        puts("\nImprimimos como array de punteros:");
        for(i=0; i<2; i++)
                for(j=0; j<5; j++)
                        printf("\n(*(mat+%d))[%d] = %d", i, j, (*(mat+i))[j]);

        puts("\nImprimimos de otro modo (doble puntero):");
        for(i=0; i<2; i++)
                for(j=0; j<5; j++)
                        printf("\n*(*(mat+%d)+%d) = %d", i, j, *(*(mat+i) + j));
        getch();
}
```

7.8.1 Inicializar un array de punteros a cadena

Con matrices iniciaba un array de la manera siguiente:

```
char Nombres[5][10] =
                {
                        "Pere",
                        "Maria",
                        "Lluís",
                        "Antoni",
                        "Anna"
                };
```

También se puede iniciar un array de cadenas así:

```
char *Nombres[5] =
                {
                        "Pere",
                        "Maria",
                        "Lluís",
                        "Antoni",
                        "Anna"
                };
```

7.9 Arrays de Punteros

Un array, de cualquier tipo, multidimensional (n), pude ser representado mediante un array de punteros que contienen la dirección de memoria en la que reside un valor. El nuevo array será de una dimensión menos que el original (es decir, tendrá n-1 dimensiones):

```
float mat[10][20];
```

puede ser representado mediante la expresión:

```
float   *mat[10];
```

De modo general, podemos afirmar que:

```
tipo_dato  nombre[indice 1][indice 2][indice 3]. . . [indice n-1][indice n];
```

puede ser sustituido por

```
tipo_dato  *nombre[indice 1][indice 2][indice 3]. . . [indice n-1];
```

Ejemplo 1:

```
int x[10][20];
```

es equivalente a

```
int  *x[10];
```

Si queremos acceder a un elemento individual, para cargar datos por inicialización, podemos poner, como formas equivalentes:

```
X[2][5] = 27;
```

o bien:

```
*(x[2] + 5) = 27;
```

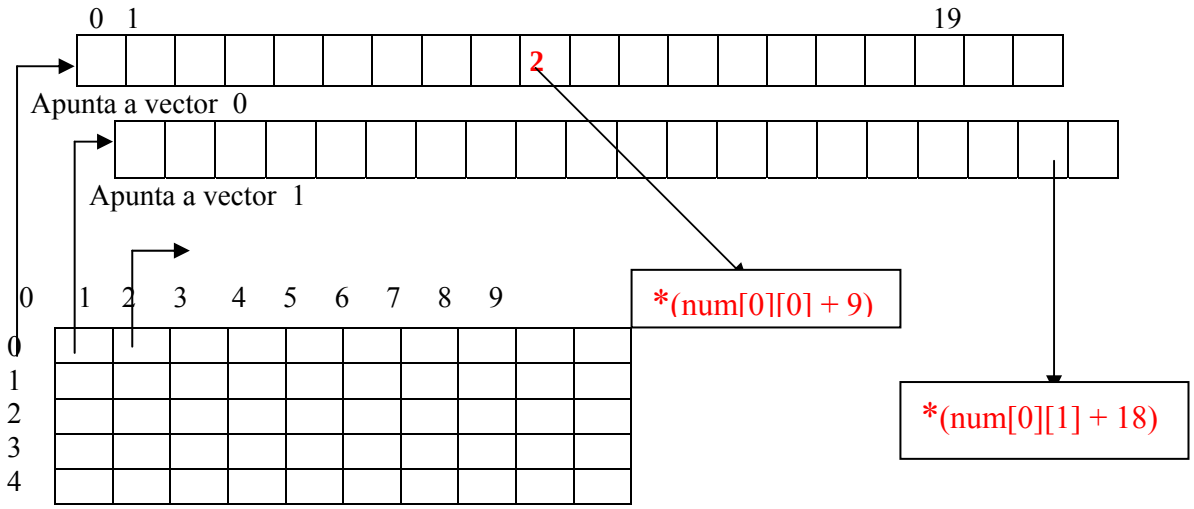
Ejemplo 2:

```
float  num[5][10][20];
```

guarda 1000 valores float. Con notación de array de punteros esa declaración es equivalente a:

```
float  *num[5][10];
```

que representa un array de 50 punteros, cada uno de los cuales apunta (guarda la dirección de) a un array monodimensional (vector) de 20 valores float.



Ejemplo: Declara un array bidimensional de 2x5. Tratandolo como un array de punteros valor[2][5] se puede representar como *valor[10] en donde valor es un array de 10 punteros a entero.

```
#include <conio.h>
#include <stdio.h>
int main()
{
	int valor[][5]= {3, -6, 0, 9, 15, 4, 12, 7, 23, 1}, i, j;

	/*direcciones de las filas:*/

	puts("Direcciones de las filas:");
	for(i=0; i<2; i++)
		printf("%d  ", valor[i]);

	puts("\nDirecciones de cada elemento:");
	for(i=0; i<2; i++)
	{
		for(j=0; j<5; j++)
			printf("%d  ", (valor[i]+j));
		printf("\n");
	}

	puts("\nContenidos de cada elemento (como puntero doble):");
	for(i=0; i<2; i++)
	{
		for(j=0; j<5; j++)
			printf("%d  ", *(*(valor+i)+j));
		printf("\n");
	}
	puts("\nContenidos de cada elemento (como array de punteros:");
	for(i=0; i<2; i++)
	{
		for(j=0; j<5; j++)
			printf("%d  ", *(valor[i]+j));
		printf("\n");
	}
	getch();
}
```

Nota: en la impresión de direcciones de memoria he empleado la notación

```
printf("%d ", (valor[i]+j));
```

que imprime las direcciones como números enteros. Si se desea trabajar con formatos hexadecimales, emplear la notación:

```
printf("%#X ", (valor[i]+j));
```

O bien:

```
printf("%p ", (valor[i]+j));
```

7.10 Punteros y cadenas de caracteres

Como su propio nombre indica una cadena de caracteres es precisamente eso un conjunto consecutivo de caracteres. Como ya habíamos comentado los caracteres se codifican utilizando el código ASCII que asigna un número desde 0 hasta 255 a cada uno de los símbolos representables en nuestro ordenador. Las cadenas de caracteres utilizan el valor 0 ('\0') para indicar su final. A este tipo de codificación se le ha llamado alguna vez ASCIIZ (la Z es de zero).

Las cadenas de caracteres se representan entre comillas dobles (") y los caracteres simples, como ya habíamos indicado con comillas simples ('). Puesto que son un conjunto consecutivo de caracteres la forma de definirlas es como una matriz de caracteres.

```
char identificador[tamaño_de_la_cadena];
```

Y por ser en esencia una matriz todo lo comentado anteriormente para matrices y punteros puede ser aplicado a ellas. Así la siguiente definición constituye también una cadena de caracteres:

```
char   *identificador;
```

La diferencia entre ambas declaraciones es que la primera reserva una zona de memoria de tamaño_de_la_cadena para almacenar el mensaje que deseemos mientras que la segunda sólo genera un puntero. La primer por tratarse de una matriz siempre tiene un puntero asociado al inicio del bloque del tamaño especificado. Podemos tratar a las cadenas como punteros a caracteres (char *) pero tenemos que recordar siempre que un puntero no contiene información sólo nos indica dónde se encuentra ésta, por tanto con la segunda definición no podríamos hacer gran cosa puesto que no tenemos memoria reservada para ninguna información. Veamos un ejemplo para comprender mejor la diferencia entra ambas declaraciones. Utilizaremos dos funciones especiales de stdio.h para trabajar con cadenas. Estas son puts y gets que definiríamos como un printf y un scanf exclusivo para cadenas.

```
#include <stdio.h>
main()
{
        char    cadena1[10];
        char    cadena2[10];
        char    *cadena;

        gets(cadena1); /* Leemos un texto por teclado y lo almacenamos en cadena 1 */
        gets(cadena2); /* Idem cadena2 */

        puts (cadena1); /* Lo mostramos en pantalla */
        puts (cadena2);

        cadena = cadena1; /* cadena que sólo es un puntero ahora apunta a cadena1 en
donde tenemos 10 caracteres reservados por la definición */

        puts (cadena);  /* Mostrara en pantalla el mensaje contenido en cadena1 */
        cadena = cadena2; /* Ahora cadena apunta a la segunda matriz de caracteres */
        gets(cadena);   /* Cuando llenos sobre cadena ahora estamos leyendo sobre
cadena2, debido al efecto de la instrucción anterior */
        puts(cadena2); /* SI imprimimos ahora cadena2 la pantalla nos mostrará la cadena
que acabamos de leer por teclado */
}
```

En el programa vemos como utilizamos cadena que solamente es un puntero para apuntar a distintas zonas de memoria y utilizar cadena1 o cadena2 como destino de nuestras operaciones. Como podemos ver cuando

cambiamos el valor de cadena a cadena1 o cadena2 no utilizamos el operador de dirección &, puesto que como ya hemos dicho una matriz es en sí un puntero (si sólo indicamos su nombre) y por tanto una matriz o cadena de caracteres sigue siendo un puntero, con lo cual los dos miembros de la igualdad son del mismo tipo y por tanto no hay ningún problema.

Ejemplo 2: Este sencillo programa cuenta los espacios y las letras 'e' que hay en una cadena.

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

int main()
{
	char cadena[]=Arriba Alianza Lima!";
	char *p;
	int espacios=0, letras_e=0;
	p = cadena;
	while (*p!='\0')
	{
		if (*p==' ') espacios++;
		if (*p=='e') letras_e++;
		p++;
	}
	printf( "En la cadena \"%s\" hay:\n", cadena );
	printf( "  %i espacios\n", espacios );
	printf( "  %i letras e\n", letras_e );
}
```

Para recorrer la cadena necesitamos un puntero *p* que sea de tipo char. Debemos hacer que *p* apunte a la cadena (p=cadena). Así *p* apunta a la dirección del primer elemento de la cadena. El valor de **p* sería por tanto 'A'. Comenzamos el bucle. La condición comprueba que no se ha llegado al final de la cadena (*p!='\0'), recordemos que '\0' es quien marca el final de ésta. Entonces comprobamos si en la dirección a la que apunta *p* hay un espacio o una letra e. Si es así incrementamos las variables correspondientes. Una vez comprobado esto pasamos a la siguiente letra (*p++*).

Dos cosas muy importantes: primero no debemos olvidarnos nunca de inicializar un puntero, en este caso hacer que apunte a *cadena*. Segundo no debemos olvidarnos de incrementar el puntero dentro del bucle (p++), sino estaríamos en un bucle infinito siempre comprobando el primer elemento.

En la condición del bucle podíamos usar símplemente: while (!*p), que es equivalente a (*p!='\0').

En este otro ejemplo sustituímos los espacios por guiones:

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

int main()
{
	char cadena[]="Arriba Alianza Lima!";
	char *p;

	p = cadena;
	while (*p!='\0')
	{
		if (*p==' ')
			*p = '-';
```

```
            p++;
        }
        printf( "La cadena queda: \"%s\" \n", cadena );
}
```

y se obtiene:

La cadena queda: "Arriba-Alianza-Lima!"

7.11 Paso y devolucion de punteros

7.11.1 Paso punteros a funciones

Veamos el siguiente programa:

```
#include <stdio.h>
void Sumar (int a, int b, int c)
{
        int Resultado;
        Resultado = a + b;
        c = Resultado;
}

int main (void)
{
        int x;
        int y;
        int z;

        x = 4;
        y = 2;
        Sumar (x,y,z);
        printf ("La z vale: %d\n",z);
        return 0;
}
```

¿Que imprime este programa?. UN VALOR INDETERMINADO.
Para llegar a esta conclusión es necesario entender como se crean y se destruyen las variables locales de una función, y como funciona el paso de parámetros.

Las variables globales de un programa son aquellas que se declaran fuera del ámbito de las funciones. Es decir, en un programa de un solo módulo corresponde a las que se declaran antes de la especificación de sus funciones. Existe un área de memoria usada por el programa donde se encuentran almacenadas estas variables. Son conocidas y pueden ser usadas por todas las funciones del programa.

Cada función de C dispone de un "área propia" donde se almacenan sus variables locales y sus parámetros mientras que se encuentra activa su ejecución. Esta área se crea cuando la función es llamada, y se destruye cuando la función termina.

¿Qué ocurre entonces al ejecutarse la instrucción [Sumar (x,y,z)]?.

- Primero se activa el bloque de la función, que reserva espacio para una variable local Resultado, y que también reserva espacio para los parámetros a, b y c.
- A continuación, el valor de x es copiado a a, el valor de y es copiado a b, y el valor de z es copiado a c.

- Se ejecuta la función, que en este caso solo accede a los datos alojados en esta área temporal. Resultado valdrá 6, c valdrá 6, a valdrá 4 y b valdrá 2.
- Cuando termina la ejecución de la función Sumar, el bloque de activación creado (donde residen Resultado, a, b y c) es destruido, y sus valores se pierden.
- Por lo tanto, el programa imprimirá el valor de z, que no es 6, ya que en ningún momento se le ha establecido este valor. El valor de z es indeterminado porque es una variable no inicializada, y el contenido de la memoria donde le ha tocado vivir puede ser cualquiera.

Si lo que queremos es que la variable z valga 6 después de retornar de la función Sumar, deberíamos programarlo de la siguiente manera, utilizando punteros:

```
#include <stdio.h>
void Sumar (int a, int b, int *c)
{
        int Resultado;
        Resultado = a + b;
        *c = Resultado;
}

int main (void)
{
        int x = 4;
        int y = 2;
        int z;

        Sumar (x,y,&z);
        printf ("La z vale: %d\n",z);
        return 0;
}
```

Ahora vemos que la función Sumar no recibe un integer como tercer parámetro, sino un puntero a integer.

Una cuestión importante, antes de nada: EN C NO EXISTE EL PASO DE PARAMETROS POR REFERENCIA. Se puede pasar por valor un puntero, pero no es mas que eso, una variable de tipo puntero a algo pasada por valor, no un paso por referencia.

¿Qué está haciendo este programa ahora?

- Crea el bloque de activación, exactamente igual que antes, para alojar a la variable local Resultado, y a los parámetros a, b y c.

- Copia el valor de x en a, el valor de y en b, y el valor de &z en c. Dependiendo del bloque de activación previo de esta función, z vivirá en una posición de memoria determinada, que podría ser por ejemplo la 5000. Se está pasando el valor 5000 a c, y por tanto, la variable c vale 5000.
- Al ejecutar *c = Resultado, se está poniendo en la posición de memoria indicada por c el valor 6. Esta posición es la 5000, que fue el valor pasado a la función y como en la posición 5000 se encuentra la variable z, ahora (y justo ahora) es cuando la variable z toma por valor 6.

Hemos encontrado la forma de "pasar parámetros por referencia", gracias a los punteros.

Por qué es importante el paso de parámetros por referencia?

- En primer lugar, si no existieran los parámetros por referencia, todas las variables del programa deberían ser públicas (globales), al menos las que se fueran a compartir entre funciones.

- No podríamos declarar las variables deseadas como locales en las funciones (con la seguridad de código que esto ofrece), ya que otra función no podría acceder a ellas, como en el caso del ejemplo anterior.

Ejemplo : Hacer un programa para entrar dos valores desde el teclado y guardarlos respectivamente en les variables A y B. Estos valores se han de pasar a una función de manera que cuando se vuelva a main() se ha de cumplir que A > B.

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

void cambiar (int *pA, int *pB);
main()
{
        int A, B;
        clrscr();
        puts("Entre el valor para variable A : ");
        scanf("%d", &A);
        puts("Entre el valor para la variable B : ");
        scanf("%d", &B);
        cambiar(&A, &B);
        printf("%d, %d ", A, B);
}

void cambiar(int *pA, int *pB)
{
        int Auxiliar;
        if (*pB > *pA)
        {
                Auxiliar = *pA;
                *pA = *pB;
                *pB = Auxiliar;
        }
}
```

7.11.2 Devolucion de punteros

Para devolver un puntero, una función debe ser declarada como si tuviera un tipo puntero de vuelta, si no hay coincidencia en el tipo de vuelta la función devuelve un puntero nulo. En el prototipo de la función antes del nombre de esta hay que poner un asterisco y en la llamada hay que igualar la función a un puntero del mismo tipo que el valor que devuelve. El valor que se retorna debe ser también un puntero.

PROTOTIPO:

```
valor_devuelto *nombre(lista_parametros);
```

LLAMADA:

```
puntero=nombre_funcion(lista_parametros);
```

DESARROLLO:

```
valor_devuelto *nombre_funcion(lista_parametros)
```

```
{
        cuerpo;
        return puntero;
}
```

Ejemplo: Busca la letra que se le pasa en la cadena y si la encuentra muestra la cadena a partir de esa letra.

```
#include<stdio.h>
void *encuentra(char letra, char *cadena);

void main(void)
{
        char frase[80], *p, letra_busca;
        clrscr();

        printf("Introducir cadena: ");
        gets(frase);
        fflush(stdin);
        printf("Letra a buscar: ");
        letra_busca=getchar();
        p=encuentra(letra_busca,frase);
        if(p)
        printf("\n %s",p);

        getch();
}

void *encuentra(char letra, char *cadena)
{
        while(letra!=*cadena && *cadena)
                cadena++;
        return cadena;
}
```

7.11.3 Punteros a array en funciones

No le tendría que sorprender; para indicar que a una función se le pasa un array, se puede utilizar la notación puntero. Así por ejemplo, en una funció par entrar datos en el array Notas, se puede definir el parámetro de la manera siguiente:

```
void EntrarNotas(int *Notas)
```

que equivale a:

```
void EntrarNotas(int Notas[]);
```

7.11.4 Paso de parámetros por referencia en C++

C no tiene paso de parámetros por referencia (se emulan mediante punteros), pero C++ si tiene paso de parámetros por referencia.

Veamos el fragmento de código anterior escrito de otra manera:

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

```
void Sumar (int a, int b, int &c)
{
        int Resultado;
        Resultado = a + b;
        c = Resultado;
}

int main (void)
{
        int x;
        int y;
        int z;

        x = 4;
        y = 2;
        Sumar (x,y,z);
        printf ("La z vale: %d\n",z);
        return 0;
}
```

El único cambio que se ha hecho en el programa está en la declaración de la función Sumar, se ha declarado la variable c como referencia a integer (int& c).

En este caso, el compilador de C++ se encarga de generar un codigo semejante al que hemos diseñado antes con los punteros, pero lo hace él solo, internamente. Realmente se está pasando por valor la dirección de z, y se está usando un puntero, pero la sintaxis es bastante mas clara. Aunque el uso de punteros no queda suplantado completamente con las referencias, en la mayoría de los casos (siempre que se utilice un compilador de C++ y no de C), los pasos de parámetros por referencia son mucho mas claros hechos utilizando esta herramienta.

7.12 Puntero Y Estructuras

Cómo no, también se pueden usar punteros con estructuras. Vamos a ver como funciona esto de los punteros con estructuras. Primero de todo hay que definir la estructura de igual forma que hacíamos antes. La diferencia está en que al declara la variable de tipo estructura debemos ponerle el operador '*' para indicarle que es un puntero.

Creo que es importante recordar que un puntero no debe apuntar a un lugar cualquiera, debemos darle una dirección válida donde apuntar. No podemos por ejemplo crear un puntero a estructura y meter los datos directamente mediante ese puntero, no sabemos dónde apunta el puntero y los datos se almacenarían en un lugar cualquiera.

Y para comprender cómo funcionan nada mejor que un ejemplo. Este programa utiliza un puntero para acceder a la información de la estructura:

```
#include <stdio.h>

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

```
struct estructura_amigo amigo =
	{
		"Kevin",
		"Paredes",
		"592-0483",
		30
	};

struct estructura_amigo *p_amigo;

int main()
{
	p_amigo = &amigo;
	printf( "%s tiene ", p_amigo->apellido );
	printf( "%i años ", p_amigo->edad );
	printf( "y su teléfono es el %s.\n" , p_amigo->telefono );
}
```

Has la definición del puntero *p_amigo* vemos que todo era igual que antes. *p_amigo* es un puntero a la estructura *estructura_amigo*. Dado que es un puntero tenemos que indicarle dónde debe apuntar, en este caso vamos a hacer que apunte a la variable *amigo*:

```
p_amigo = &amigo;
```

No debemos olvidar el operador & que significa 'dame la dirección donde está almacenado...'.

Ahora queremos acceder a cada campo de la estructura. Antes lo hacíamos usando el operador '.', pero, como muestra el ejemplo, si se trabaja con punteros se debe usar el operador '->'. Este operador viene a significar algo así como: "dame acceso al miembro ... del puntero ...".

Ya sólo nos queda saber cómo podemos utilizar los punteros para introducir datos en las estructuras. Lo vamos a ver un ejemplo:

```
#include <stdio.h>

struct estructura_amigo
{
	char nombre[30];
	char apellido[40];
	int edad;
	};

struct estructura_amigo amigo, *p_amigo;

int main()
{
	p_amigo = &amigo;

	/* Introducimos los datos mediante punteros */
	printf("Nombre: ");fflush(stdout);
	gets(p_amigo->nombre);
	printf("Apellido: ");fflush(stdout);
	gets(p_amigo->apellido);
```

```
        printf("Edad: ");fflush(stdout);
        scanf( "%i", &p_amigo->edad );

        /* Mostramos los datos */
        printf( "El amigo %s ", p_amigo->nombre );
        printf( "%s tiene ", p_amigo->apellido );
        printf( "%i años.\n", p_amigo->edad );
}
```

NOTA: *p_amigo* es un puntero que apunta a la estructura *amigo*. Sin embargo *p_amigo->edad* es una variable de tipo int. Por eso al usar el scanf tenemos que poner el &.

Ejemplo: Programa que introduce datos de un alumno, utilizando punteros a estructuras

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

struct Alumno
{
        char Nombre[10];
        int Edat;
        int Nota;
};

main()
{
        struct Alumno Estudiante;
        struct Alumno *pAlumno;
        pAlumno = &Estudiante;
        puts("Nombres : ");
        gets(pAlumno->Nombre);
        puts("Edat : ");
        scanf("%d ", pAlumno->Edad);
        puts("Nota : ");
        scanf("%d ", pAlumno->Nota);
}
```

7.13 Punteros a arrays de estructuras

Por supuesto también podemos usar punteros con arrays de estructuras. La forma de trabajar es la misma, lo único que tenemos que hacer es asegurarnos que el puntero inicialmente apunte al primer elemento, luego saltar al siguiente hasta llegar al último.

```
#include <stdio.h>
#define ELEMENTOS 3

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

```
struct estructura_amigo amigo[] =
                {
                        "Juanjo", "Lopez", "504-4342", 30,
                        "Marcos", "Gamindez", "405-4823", 42,
                        "Ana", "Martinez", "533-5694", 20
                };

struct estructura_amigo *p_amigo;

int main()
{
        int num_amigo;
        p_amigo = amigo;  /* apuntamos al primer elemento del array */

        /* Ahora imprimimos sus datos */
        for( num_amigo=0; num_amigo<ELEMENTOS; num_amigo++ )
        {
                printf( "El amigo %s ", p_amigo->nombre );
                printf( "%s tiene ", p_amigo->apellido );
                printf( "%i años ", p_amigo->edad );
                printf( "y su teléfono es el %s.\n" , p_amigo->telefono );

                /* y ahora saltamos al siguiente elemento */
                p_amigo++;
        }
}
```

En vez de *p_amigo = amigo;* se podía usar la forma *p_amigo = &amigo[0];*, es decir que apunte al primer elemento (el elemento 0) del array. La primera forma creo que es más usada pero la segunda quizás indica más claramente al lector principiante lo que se pretende.

Ahora veamos el ejemplo anterior de cómo introducir datos en un array de estructuras mediante punteros:

```
#include <stdio.h>
#define ELEMENTOS 3

struct estructura_amigo
{
        char nombre[30];
        char apellido[40];
        int edad;
};

struct estructura_amigo amigo[ELEMENTOS], *p_amigo;

int main()
{
        int num_amigo;
        /* apuntamos al primer elemento */
        p_amigo = amigo;

        /* Introducimos los datos mediante punteros */
        for ( num_amigo=0; num_amigo<ELEMENTOS; num_amigo++ )
```

```
        {
                printf("Datos amigo %i\n",num_amigo);
                printf("Nombre: ");fflush(stdout);
                gets(p_amigo->nombre);
                printf("Apellido: ");fflush(stdout);
                gets(p_amigo->apellido);
                printf("Edad: ");fflush(stdout);
                scanf( "%i", &p_amigo->edad );
                /* vaciamos el buffer de entrada */
                while(getchar()!='\n');
                /* saltamos al siguiente elemento */
                p_amigo++;
        }

        /* Ahora imprimimos sus datos */
        p_amigo = amigo;
        for( num_amigo=0; num_amigo<ELEMENTOS; num_amigo++ )
        {
                printf( "El amigo %s ", p_amigo->nombre );
                printf( "%s tiene ", p_amigo->apellido );
                printf( "%i años.\n", p_amigo->edad );
                p_amigo++;
        }
}
```

Es importante no olvidar que al terminar el primer bucle for el puntero p_amigo apunta al último elemento del array de estructuras. Para mostrar los datos tenemos que hacer que vuelva a apuntar al primer elemento y por eso usamos de nuevo p_amigo=amigo; (en negrita).

7.14 Paso de estructuras a funciones utilizando punteros

```
#include <stdio.h>
#include <conio.h>
typedef struct
{
        int  hora;
        int min;
        int seg;
}       tipodato;

tipodato LLAMADA[2];  //GLOBAL------

void llenar(void); //al ser global, las funciones "ven" la estructura
void imprimir(void);

int main()
{
        llenar(); //llamda a la funci¢n
        puts("\n\n*************************FIN*************************");
        getch();
}

//cuerpo de la funci¢n
```

```
void llenar(void)
{
        int i;
        for(i=0;i<2; i++)
        {
    //Empleo notaci¢n de arrays

                printf("\nHora %d: ", i+1);
                scanf("%d", &LLAMADA[i].hora);
                printf("Minuto %d: ", i+1);
                scanf("%d", &LLAMADA[i].min);
                printf("Segundo %d: ", i+1);
                scanf("%d", &LLAMADA[i].seg);
        }
        imprimir(); //llamada a la funci¢n que imprime los datos

        return;
}

//cuerpo de la funci¢n a la  que llama la funci¢n llenar
void imprimir()
{
        int j=0;
        puts("Desde la funci¢n imprimimos los datos alterados,");
        puts("empleando notaci¢n de punteros:");
        for(; j<2; j++)
        {
                printf("\n\nHora %d: %d",j+1, (LLAMADA+j)->hora+1);
                printf("\nMinuto %d: %d",j+1, (LLAMADA+j)->min+1);
                printf("\nSegundo %d: %d",j+1,(LLAMADA+j)->seg+1);
        }
}
```

7.15 Indirección Múltiple

El lenguaje de programación C permite referenciar un puntero con otro puntero. Son declaracions ciertamente complejas, que algunas veces la notación C las hace todavia más difíciles de interpretar. Por ejemplo, una declaración como la siguiente:

```
int **ppInt;
```

es un puntero a un puntero que apunta a un puntero que apunta a un dato del tipo int. Vea el siguiente fragmento de código:

```
int var;
int *pVar;              /* Puntero a posición de memoria ocupada por un int*/
int **ppVar;            /* Puntero a posición de memoria ocupada por un puntero a int.*/

Var = 10;
*pVar = Var;   /* pVar guarda la dirección de memoria de variable Var*/

*pVar = 20;     /* Dirección de memoria apuntada por pVar (variable Var), poner un 20*/
```

```
printf("%d", Var);      /*Imprimirá 20;*/

*ppVar = &pVar;        /* ppVar guarda dirección de memoria de variable pVar*/

**ppVar = 30;          /* en la dirección de memoria que apunta la variable que hay en la
dirección apuntada per ppVar (variable pVar), poner un 30*/

printf("%d", Var);      /* Imprimirà un 30*/
```

Es realmente rebuscado, pero hay situaciones (es posible que nunca las encuentre, y si encuentra una deseará no haberla encontrado) que sólo pueden resolverse utilizando este tipo de notación.

Vea a continuación un ejemplo de utilización de punteros a punteros, "Los arrays bidimensionales".

Primeramente recuerde que un array bidimensional no es más que un array de arrays. Suponga un array de int declarado como sigue:

```
int Tabla[5][5];
```

1	2	3	4	5
6	7	8	9	10
11	12	13	14	15
16	17	18	19	20
21	22	23	24	25

Tabla[5][5], significa un array formado por 5 arrays, cada uno de ellos con 5 datos del tipo int..

Tabla[0]	1	2	3	4	5
Tabla[1]	6	7	8	9	10
Tabla[2]	11	12	13	14	15
Tabla[3]	16	17	18	19	20
Tabla[4]	21	22	23	24	25

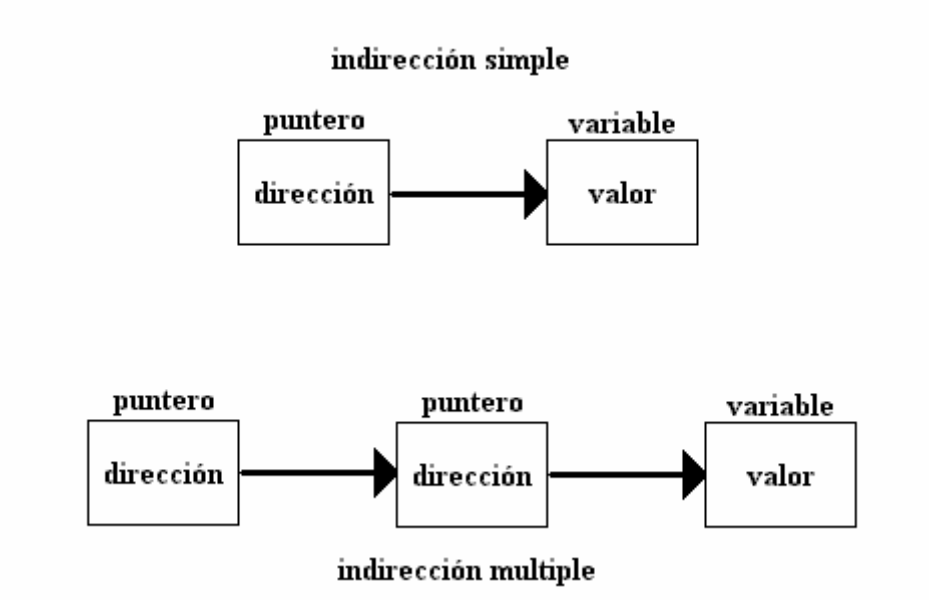
Tabla es la dirección de todo el array, supoga que comienza en la dirección 1000. En un array de enteros cada elemento ocupa dos bytes, cada fila entonces, empezará 10 bytes después de la anterior (5 elementos de cada fila por dos bytes que tiene cada elemento). El compilador sabe cuantas columnas tiene cada fila del array ya que se le ha indicado en la declaración. Así, Tabla+1 es la dirección de tabla 1000, más los bytes que ocupa una fila, 10; Tabla+1 comenzará entonces en la dirección 1010, Tabla+2 en la dirección 1020, Tabla+3 en la dirección 1040, y Tabla+5 en la dirección 1050.

Ahora que ha visto como referenciar cada una de las filas; vea como se referencia con notación puntero, un elemento individual de alguna de las filas.

Cogeremos por ejemplo el primer elemento de la segunda fila. Con notación array la dirección de este elemento se indica con &Tabla[1][0], con notación puntero se indica con *(Tabla+1). Observe que tanto Tabla+1 como *(Tabla+1) hacen referencia a la misma dirección. Si se suma 1 a Tabla+1, se añaden 10 bytes, y se encuentra Tabla+2, la dirección del primer elemento de la tercera fila, pero si se suma 1 a *(Tabla+1), se está sumando sólo dos bytes encontrando entonces la dirección del segundo elemento de la segunda fila. La expresión quedaria *(Tabla+2)+1, y para referenciar el contenido de esta dirección sólo hace

falta referenciar con notación puntero la expresión anterior, o sea Tabla[1][1] = *(*(Tabla+1)+1). Generalizando :

Tabla[y][j] = *(*(Tabla+y)+j)



7.16 Acceso a memoria física.

C es el lenguaje comúnmente utilizado a la hora de programar módulos de sistemas operativos, drivers, etc. Este tipo de programas deben acceder con frecuencia al hardware de la maquina, para poder leer datos del exterior (movimientos de un ratón, temperatura de un sensor…) o bien para escribir sobre ellos (dibujar en una pantalla, mover electroválvulas…). Una de las maneras de hacer todo esto consiste en mapear los dispositivos de entrada/salida sobre un espacio real de memoria. De este modo, cuando se escribe en cierta posición de memoria lo que se está haciendo realmente es actuar sobre una salida, y cuando se lee de cierta posición de memoria se está leyendo una entrada.

Por ejemplo, en un PC, la memoria de la pantalla en modo texto se encuentra en la posición 0B8000000H (esto no funcionará en Windows, solo en MS-DOS, ya que Windows protege a la máquina contra los programas de usuario bajo una capa de memoria virtual). A partir de esta posición, cada dos bytes codifican una letra de pantalla: el primer byte corresponde al código ASCII de la misma, y el segundo byte corresponde al código del color a aplicarle.

Examinemos el siguiente fragmento de programa:

```
void PintarLetra (int x, int y, char z)
{
        int  Sitio;
        int* Pantalla;

        Pantalla = 0xB8000000;
        Sitio = x + y*80;
        Pantalla += Sitio;
        *Pantalla = 0x0F00 + z;
}
```

El programa anterior utiliza un puntero para poder acceder al área física de pantalla. En este caso no se le asigna la dirección de ninguna variable, sino que se le asigna un valor concreto: el lugar donde queremos acceder.

Luego se calcula la posición del dato (cada fila tiene 80 caracteres en modo texto).

A la dirección base de la pantalla se le suma el desplazamiento calculado.

Se asigna a esa posición de memoria el valor deseado, que contiene en la parte baja el valor de la letra (su código ASCII) y en la parte alta 0x0F (15), que es el código del color blanco.

Vemos que mediante el uso de punteros es posible acceder a zonas concretas de memoria, y gracias a ello acceder al hardware mapeado en la máquina.

CAPITULO VIII:

Asignación dinamica de memoria

8.1 Introducción

En este tema estudiaremos las posibilidades que ofrece el lenguaje C a la hora de trabajar dinámicamente con la memoria dentro de nuestros programas, esto es, reservar y liberar bloques de memoria dentro de un programa.

Además en este tema se introducirá el concepto de tipo abstracto de dato y la forma de dividir un gran programa en otros más pequeños.

8.2 Asignación dinámica y estática de memoria.

Hasta este momento solamente hemos realizado asignaciones estáticas del programa, y más concretamente estas asignaciones estáticas no eran otras que las declaraciones de variables en nuestro programa. Cuando declaramos una variable se reserva la memoria suficiente para contener la información que debe almacenar.

Esta memoria permanece asignada a la variable hasta que termine la ejecución del programa (función main). Realmente las variables locales de las funciones se crean cuando éstas son llamadas pero nosotros no tenemos control sobre esa memoria, el compilador genera el código para esta operación automáticamente. En este sentido las variables locales están asociadas a asignaciones de memoria dinámicas, puesto que se crean y destruyen durante la ejecución del programa.

Así entendemos por asignaciones de memoria dinámica, aquellas que son creadas por nuestro programa mientras se están ejecutando y que por tanto, cuya gestión debe ser realizada por el programador.

8.3 ¿Cómo se reserva memoria dinámicamente?

El lenguaje C dispone, como ya indicamos con anterioridad, de una serie de librerías de funciones estándar.

El fichero de cabeceras stdlib.h contiene las declaraciones de dos funciones que nos permiten reservar memoria, así como otra función que nos permite liberarla.

8.3.1 Reserva de memoria

Las dos funciones que nos permiten reservar memoria son:

8.3.1.1 Malloc

La función malloc (**M**emory **alloc**ate - asignar memoria) reservar una parte de la memoria y devuelve la dirección del comienzo de esa parte. Esta dirección podemos almacenarla en un puntero y así podemos acceder a la memoria reservada. La función malloc tiene el siguiente formato:

```
#include <stdlib.h>
void *malloc ( size_t size );
```

El tipo `size_t` está definido normalmente como `unsigned` y se utiliza en todas las funciones que necesitan un tamaño en bytes.

La function malloc se utilize mediante el siguiente formato:

```
puntero = (tipo_de_variable *) malloc( número de bytes a reservar );
```

Donde:

- puntero: es una variable tipo puntero que almacena la dirección del bloque de memoria reservado. Puede ser un puntero a char, int, float,...

- (tipo_de_variable *): es lo que se llama un molde. La función malloc nos reserva una cierta cantidad de bytes y devuelve un puntero del tipo void (que es uno genérico). Con el molde le indicamos al compilador que lo convierta en un puntero del mismo tipo que la variable puntero. Esto no es necesario en C, ya que lo hace automáticamente, aunque es aconsejable acostumbrarse a usarlo.

8.3.1.2 Calloc

La función calloc (**C**lear **Alloc**) tiene el siguiente formato:

```
#include <stdlib.h>
void *calloc ( numero_de_elementos, tamaño_del_elemento );
```

En este caso se le pasa como parámetro el número de elementos consecutivos que se desean. A diferencia de malloc, calloc inicializa el contenido de la memoria asignada a 0.

Hay que tener en cuenta, que un calloc no funciona exáctamente igual que el malloc, ya que en malloc, generalmente, asignábamos un valor a size_t igual al número de elementos multiplicado por el tamaño de los mismos, nótese que con calloc esto no es necesario, ya que el tamaño de los elementos se lo pasamos en tamaño_del_elemento.

Ejemplo:

```
long *ptr;
/* Dos asignaciones idénticas */
/* 1 */
ptr = malloc( sizeof ( long ) * 10 );
memset( ptr, 0, 10 );

/* 2 */
ptr = calloc( 10, sizeof ( long ) );
```

8.3.1.3 realloc

Reasigna una zona de memoria previamente dimensionada (reduce o ezpande el tamaño de un bloque de memoria), o asigna una nueva en caso de que la zona que queramos redimensionar sea NULL. El nuevo bloque no tiene porque estar en el mismo lugar. Formato:

```
void *realloc(void *bloque, nuevo_tamaño);
```

Realloc retorna un puntero a la nueva zona de memoria, hay que tener en cuenta que no tendrá por qué ser la misma que antes, así que siempre tenemos que igualar el puntero al valor de esta función. También hay que pensar que realloc lo que hace realmente es liberar esa zona de memoria y meter los datos en otra (si es menor no pasa nada y si es menor, evidentemente, los trunca).

Las funciones antes mencionadas reservan la memoria especificada y nos devuelven un puntero a la zona en cuestión. Si no se ha podido reservar el tamaño de la memoria especificado devuelve un puntero con el valor 0 o NULL. El tipo del puntero es, en principio void, es decir, un puntero a cualquier cosa. Por tanto, a la hora de ejecutar estás funciones es aconsejable realizar una operación cast (de conversión de tipo) de cara a la utilización de la aritmética de punteros a la que aludíamos anteriormente. Los compiladores modernos suelen realizar esta conversión automáticamente.

```
char *str;
str = (char *) malloc(10*sizeof (char)); /*eserva para 10 caracteres*/
```

O bien:

```
str = (char *) calloc(10, sizeof (char));
str = (char *) realloc(str, 50*sizeof (char)); /*realojamos*/
```

Antes de indicar como deben utilizarse las susodichas funciones tenemos que comentar el operador sizeof. Este operador es imprescindible a la hora de realizar programas portables, es decir, programas que puedan ejecutarse en cualquier máquina que disponga de un compilador de C.

El operador sizeof(tipo_de_dato), nos devuelve el tamaño que ocupa en memoria un cierto tipo de dato, de esta manera, podemos escribir programas independientes del tamaño de los datos y de la longitud de palabra de la máquina. En resumen si no utilizamos este operador en conjunción con las conversiones de tipo cast probablemente nuestro programa sólo funciones en el ordenador sobre el que lo hemos programado.

Por ejemplo, el los sistemas PC, la memoria está orientada a bytes y un entero ocupa 2 posiciones de memoria, sin embargo puede que en otro sistema la máquina esté orientada a palabras (conjuntos de 2 bytes, aunque en general una máquina orientada a palabras también puede acceder a bytes) y por tanto el tamaño de un entero sería de 1 posición de memoria, suponiendo que ambas máquinas definan la misma precisión para este tipo.

Con todo lo mencionado anteriormente veamos un ejemplo de un programa que reserva dinámicamente memoria para algún dato.

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

main()
{
        int  *p_int;
        float *mat;
        p_int = (int *) malloc(sizeof (int));
        mat = (float *)calloc(20,sizeof (float));
        if ((p_int==NULL)||(mat==NULL))
        {
                printf ("\nNo hay memoria");
                exit(1);
        }

        /* Aquí irían las operaciones sobre los datos */
        /* Aquí iría el código que libera la memoria */

}
```

Este programa declara dos variables que son punteros a un entero y a un float. A estos punteros se le asigna una zona de memoria, para el primero se reserva memoria para almacenar una variable entera y en el segundo se crea una matriz de veinte elementos cada uno de ellos un float. Obsérvese el uso de los operadores cast para modificar el tipo del puntero devuelto por malloc y calloc, así como la utilización del operador sizeof.

Como se puede observar no resulta rentable la declaración de una variable simple (un entero, por ejemplo, como en el programa anterior) dinámicamente, en primer lugar por que aunque la variable sólo se utilice en una pequeña parte del programa, compensa tener menos memoria (2 bytes para un entero) que incluir todo el código de llamada a malloc y comprobación de que la asignación fue correcta (ésto seguro que ocupa más de dos bytes). En segundo lugar tenemos que trabajar con un puntero con lo cual el programa ya aparece un poco más engorroso puesto que para las lecturas y asignaciones de las variables tenemos que utilizar el operador *.

Para termina un breve comentario sobre las funciones anteriormente descritas. Básicamente da lo mismo utilizar malloc y calloc para reservar memoria es equivalente:

```
mat = (float *)calloc (20,sizeof (float));
mat = (float *)malloc (20*sizeof (float));
```

La diferencia fundamental es que, a la hora de definir matrices dinámicas calloc es mucho más claro y además inicializa todos los elementos de la matriz a cero. Nótese también que puesto que las matrices se referencian como un puntero la asignación dinámica de una matriz nos permite acceder a sus elementos con instrucciones de la forma:

NOTA: En realidad existen algunas diferencias al trabajar sobre máquinas con alineamiento de palabras.

```
mat[0] = 5;
mat[2] = mat[1]*mat[6]/67;
```

Con lo cual el comentario sobre lo engorroso que resultaba trabajar con un puntero a una variable simple, en el caso de las matrices dinámicas no existe diferencia alguna con una declaración normal de matrices.

8.3.2 Liberación de la memoria.

Cuando ya no necesitemos más el espacio reservado debemos liberarlo, es decir, indicar al ordenador que puede destinarlo a otros fines. Si no liberamos el espacio que ya no necesitamos corremos el peligro de agotar la memoria del ordenador.

La función que nos permite liberar la memoria asignada con malloc y calloc es **free**(puntero), donde puntero es el puntero devuelto por malloc o calloc. Es muy importante no perder la dirección del comienzo del bloque, pues de otra forma no podremos liberarlo.

```
#include <stdlib.h>
void free ( void *block );
```

En nuestro ejemplo anterior, podemos ahora escribir el código etiquetado como:

```
/* Ahora iría el código que libera la memoria */
free (p_int);
free(mat);
```

Hay que tener cuidado a la hora de liberar la memoria. Tenemos que liberar todos los bloques que hemos asignado, con lo cual siempre debemos tener almacenados los punteros al principio de la zona que reservamos. Si mientras actuamos sobre los datos modificamos el valor del puntero al inicio de la zona reservada, la función free probablemente no podrá liberar el bloque de memoria.

EJEMPLO 1: Vamos a ver un sencillo ejemplo del manejo de malloc y free:

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

```
int main()
{
	int bytes;
	char *texto;

	printf("Cuantos bytes quieres reservar: ");
	scanf("%i",&bytes);
	texto = (char *) malloc(bytes);

	/* Comprobamos si ha tenido éxito la operación */
	if (texto)
	{
		printf("Memoria reservada: %i bytes = %i kbytes = %i Mbytes\n",
	 bytes, bytes/1024, bytes/(1024*1024));

		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");
}
```

En este ejemplo se pregunta cuánta memoria se quiere reservar, si se consigue se indica cuánta memoria se ha reservado y dónde comienza el bloque. Si no se consigue se indica meniante el mensaje: "No se ha podido reservar memoria".

Probemos este ejemplo reservando 1000 bytes:

```
Cuantos bytes quieres reservar: 1000
Memoria reservada: 1000 bytes = 0 kbytes = 0 Mbytes
El bloque comienza en la dirección: 90868
```

Pruébalo unas cuantas veces incrementando el tamaño de la memoria que quieres reservar. Si tienes 32Mbytes de memoria RAM teclea lo siguiente:

```
Cuantos bytes quieres reservar: 32000000
No se ha podido reservar memoria
```

Malloc no ha podido reservar tanta memoria y devuelve (null) por lo que se nos avisa de que la operación no se ha podido realizar.

Ejemplo 2: Utilización de malloc

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

int main(void)
{
	char * ptr = NULL;
	unsigned int n = 0;
```

```
        printf("\n Número de caracteres: ");
        scanf("%d", &n);

        if (n)
        {
                if ( (ptr = malloc( (sizeof ( char ) * n) + 1 )) != NULL )
                {
                        int i;
                        memset(ptr, 0, (n + 1) );
                        for( i = 0 ; i < n ; i++ )
                        {
                                *(ptr + i) = getche();
                        }
                        free(ptr);
                }
                else
                        printf( "Error asignando memoria\n" );
        }
        else
                printf( "Número de caracteres no válido\n");
        }
        return (0);
}
```

Ejemplo 3: Utilización de malloc y realloc

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

int main(void)
{
        char *str;

        /* reserva memoria para cadena de caracteres */
        if((str = (char *) malloc(10*sizeof (char)))==NULL)
        {
                puts("Fallo en la reserva de memoria");
                exit(0);  /*salida forzada a S.O.*/
        }
        /* copia "Hola X" en la cadena */
        strcpy(str, "Hola X");

        printf("La cadema es:  %s\n  La direccion es: %p\n", str, str);

        /*realojamos, expandiendo a 30 caracteres*/
        str = (char *) realloc(str, 30*sizeof (char));
        printf("La cadena es: %s\n  La nueva direccion: %p\n", str, str);

        /* Liberamo memoria*/
        free(str);
```

```
        getch();
        return 0;
}
```

8.4 Problemas a evitar en el uso de punteros y el uso dinámico de memorio

El uso descuidado de punteros lleva a programas incomprensibles y particularmente a problemas difíciles de encontrar y reparar.

Ausencia de inicialización de punteros

Un problema común es el uso de punteros no inicializados. Mientras las variables externas (a funciones) o estáticas son inicializadas en cero por omisión, las variables locales (pertenecientes a funciones) no son inicializadas automáticamente. Por lo tanto punteros definidos dentro de funciones que no sean inicializados pueden contener cualquier valor ("basura").

Es un error desreferenciar un puntero con valor NULL,

```
int *ip = NULL;
cout << *ip << endl;
```

provoca la excepción de violación de acceso de memoria (segmentation fault) y detiene el programa.

Desreferenciar un puntero no inicializado

```
int *ip;
cout << *ip << endl;
```

Los compiladores en general informan (warning) que se esta usando una variable no inicializada. Nota: Lamentablemente el ambiente cygwin no informa al respecto, por lo tanto se recomienda prestar atención a las inicializaciones.

Su ejecución tiene un comportamiento aleatorio, dependiendo del valor que casualmente exista en el área de memoria asignada a la variable puntero. Puede llegar a funcionar incorrectamente si la información que hay en memoria casualmente es un entero válido; pero probablemente provocará la detención del programa. En estructuras de datos dinámicas la ejecución del programa en estas condiciones es errática, luego de ejecutarse incorrectamente por cierto tiempo se detiene ("cuelga") el programa.

Modificar área de memoria que no fue definida explícitamente

```
int *ip;
*ip = 10;
```

Potencialmente mas grave es modificar área de memoria que no fue definida explícitamente, no solo modifica un espacio de memoria que no le fue asignado; sino que, al no estar inicializado el puntero, se modifica el espacio de memoria al que casualmente direccione el puntero. La declaración int *ip solo asigna memoria para la variable puntero, no para los datos apuntados por éste. En general (lo cual no quiere decir que ocurra siempre) provoca la excepción de violación de acceso de memoria (segmentation fault) y detiene el programa. Una forma adecuada de hacerlo es:

```
int *ip;
ip = new int; /* (1) */
*ip = 10;
```

donde la función new establece memoria para el entero y la asignación inicializa el puntero. En caso de querer utilizar estrictamente el lenguaje C (el operador new es propio de C++), la sentencia (1) pasaría a ser la siguiente:

```
ip = (int*) malloc (sizeof (int)); /* (1) */
```

La asignación de punteros se puede realizar adecuadamente al:

- Establecer su valor inicial en NULL, y luego asignar memoria previo a su desreferenciamiento.
- asignar un puntero a otro que a sido adecuadamente inicializado.
- usar las funciones new o malloc para reservar memoria.

Problemas de alias

La flexibilidad de tener más de un puntero para una misma área de memoria tiene ciertos riesgos.
El manejo de memoria dinámica (heap) es restringido a las funciones new (o malloc) y delete (o free), en

```
int *ip;
int x = 1;
ip = &x;
delete ip;
```

es erróneo que delete desasigne memoria que no fue asignada con new.

En general provoca la excepción de violación de acceso de memoria (segmentation fault) y detiene el programa.

El manejo de alias puede dejar punteros colgados (dangling), en:

```
int *ip, *jp;
ip = new int;
*ip = 1;
jp = ip;
delete ip;
cout << *jp << endl;
```

jp queda apuntando a un área de memoria que conceptualmente no esta disponible, más aun el área de memoria a la cual jp quedó apuntando puede ser reasignada por ejecuciones subsiguientes de la función new. Notar el error en que se podría incurrir si luego de reasignar el área de memoria en cuestión para otro uso (nueva invocación a new), se desasigna el área de memoria a la que apunta jp (delete jp).

Pérdida de memoria asignada

```
int *ip = new int;
int *jp = new int;
*ip = 1;
*jp = 2;
jp = ip;
```

Otro problema potencial es la pérdida de memoria asignada, en el ejemplo el área de memoria asignada a jp se perdió, no puede ser retornada al heap para su posterior reasignación. Errores similares son reasignar memoria a un puntero que ya tiene memoria asignada o asignarle el valor NULL, sin previamente desasignar la memoria inicial. Esto es un mal manejo de la memoria y puede llevar al agotamiento prematuro de la misma.

8.5 Introduccion a las Estructuras Dinámicas De Datos.

Supongamos que tenemos que implementar una aplicación que permita mantener una agenda de clientes. Una forma de implementarlo consistiría en crear una estructura de datos adecuada para ello, que recuperara de un fichero del disco la información al iniciarse la ejecución del programa, y que guardara dicha información al terminar. Tenemos varias maneras de crear una estructura de datos capaz de albergar esta información:

Supondremos que la estructura de datos de un cliente es la siguiente, para examinar algunas de las posibilidades a continuación:

```
typedef struct
{
        int  Codigo;
        char Nombre[50];
        char NIF[13];
        char Telefono[25];
        char Direccion[100];
        char CodigoPostal[6];
        int  CodProvincia;
}       Cliente;
```

8.5.1 Declarar una estructura estática de clientes:

```
#define MAX_CLIENTES 1000

Cliente Clientes[MAX_CLIENTES];
int    NumClientes = 0;
```

Estamos declarando un array estático de clientes, en concreto de MAX_CLIENTES (1000). Tenemos además una variable que indica cuantos de estos clientes hay válidos hasta el momento.
Esta estructura de datos utiliza siempre la misma cantidad de memoria. Cuando hay pocos clientes despilfarra los recursos, y puede darse el caso de que no se puedan cargar todos los clientes deseados aunque la máquina tenga memoria libre.

Info Cliente 0	Info Cliente 1	Info Cliente 2	Info Cliente 3	…	Info Cliente 1000

8.5.2 Declarar una estructura estática de punteros, con clientes dinámicos:

```
#define MAX_CLIENTES 1000
Cliente* Clientes[MAX_CLIENTES];
int     NumClientes = 0;
```

Estamos declarando un array estático de punteros a clientes, en concreto de MAX_CLIENTES (1000).

Para añadir un cliente, procederíamos de la siguiente manera:

```
void MeterCliente (int Cod, char* Nom, char* Nif, char* Tel, char* Dr, char* cp, int Prv)
{
        Cliente* Cl;

        Cl = (Cliente*) malloc (sizeof (Cliente));
```

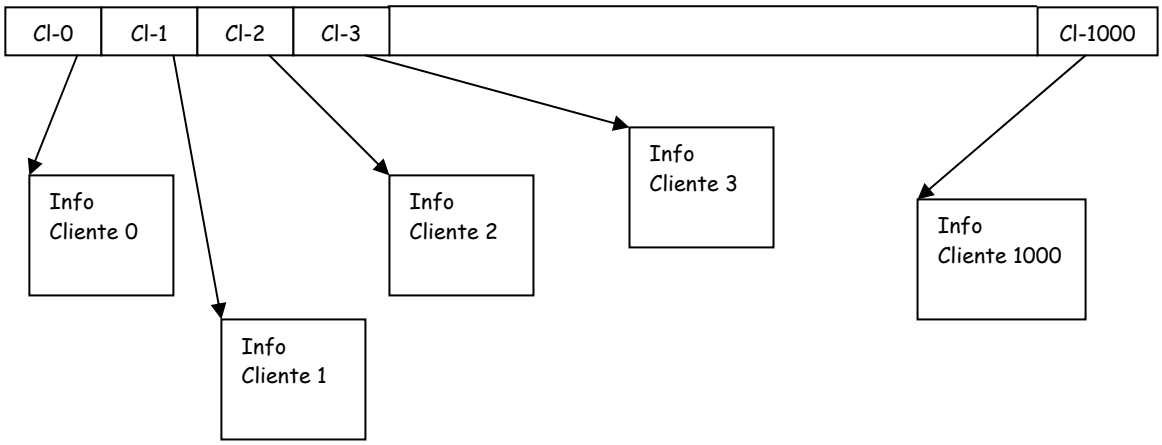
```
        Cl->Codigo = Cod;
        strcpy (Cl->Nombre,Nom);
        strcpy (Cl->NIF,Nif);
        strcpy (Cl->Telefono,Tel);
        strcpy (Cl->Direccion,Dr);
        strcpy (Cl->CodigoPostal,cp);
        Cl->CodProvincia = Prv;
        Clientes[NumClientes++] = Cl;
}
```

Vemos que la función anterior recibe por parámetro los datos del nuevo cliente (no quiere decir que esta sea la mejor manera de hacerlo, solo es un ejemplo), crea una estructura dinámica de tipo Cliente, la llena, y la engancha en el array de punteros a clientes. La nueva estructura no es liberada, ha sido creada y se ha enlazado con el array principal.

Para eliminar un cliente determinado (por su posición), procederíamos de la siguiente manera:

```
void EliminarCliente (int Cual)
{
        int i;
        free (Clientes[Cual]);
        for (i=Cual ; i < NumClientes-1 ; i++) Clientes[i] = Clientes[i+1];
        NumClientes--;
}
```

Para eliminar un cliente, primero liberamos el espacio apuntado por el array de punteros, y luego desplazamos el contenido de los punteros a los siguientes clientes. Los demás clientes no se mueven de sitio, solo se mueven sus referencias dentro del array.
En cuanto a capacidad de almacenamiento de clientes, esta solución es la misma que la anterior, pero en este caso no se usa la memoria que no es necesaria si no existen los clientes. La eliminación de clientes también es más rápida.



8.5.3 Utilizar una estructura dinámica de clientes:

En este ejemplo no utilizaremos un array estático de punteros. Añadiremos un campo a la estructura de clientes que nos permitirá ir encadenadamente de uno a otro.

```
typedef struct
```

```
{
        int  Codigo;
        char Nombre[50];
        char NIF[13];
        char Telefono[25];
        char Direccion[100];
        char CodigoPostal[6];
        int  CodProvincia;
        Cliente* Siguiente;       /*No es correcto, pero para el ejemplo es irrelevante.*/
}       Cliente;
```

La estructura de datos necesaria sería ahora:

```
Cliente* Clientes = NULL;
```

Únicamente se declara un puntero a cliente, que es inicializado a NULL (puntero nulo) porque no hay clientes disponibles.

¿Cómo añadiremos un cliente a la estructura actual?

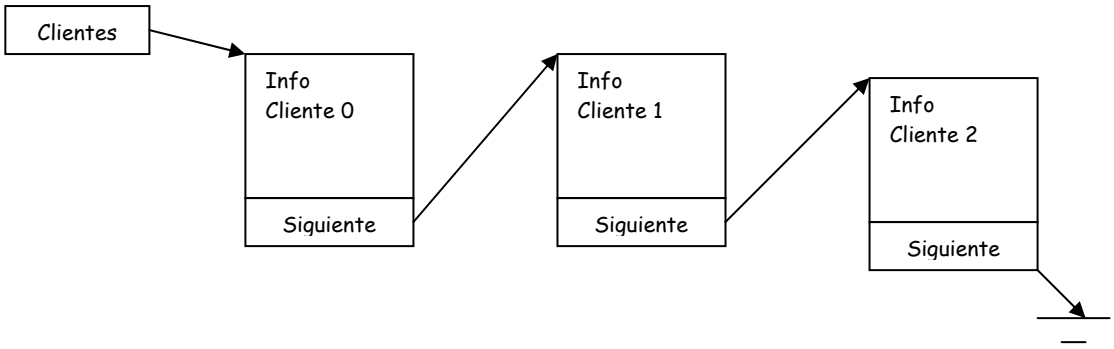
```
void MeterCliente (int Cod, char* Nom, char* Nif, char* Tel, char* Dr, char* cp, int Prv)
{
        Cliente* Cl;

        Cl = (Cliente*) malloc (sizeof (Cliente));
        Cl->Codigo = Cod;
        strcpy (Cl->Nombre,Nom);
        strcpy (Cl->NIF,Nif);
        strcpy (Cl->Telefono,Tel);
        strcpy (Cl->Direccion,Dr);
        strcpy (Cl->CodigoPostal,cp);
        Cl->CodProvincia = Prv;
        Cl->Siguiente = NULL;

        Cliente* ClAux = Clientes;
        if (ClAux == NULL)
        {
                Clientes = Cl;
        }
        else
        {
                while (ClAux->Siguiente != NULL)
                        ClAux = ClAux->Siguiente;
                ClAux->Siguiente = Cl;
        }
}
```

Vemos que en este caso, al añadir un cliente lo que hacemos es allocar espacio para el nuevo cliente exactamente igual que en el ejemplo anterior, pero ahora lo encadenamos a la lista de clientes ya existente. Si la lista está vacia (Clientes = NULL), se asigna el nuevo cliente a este puntero, indicando que es el único y primero. Si ya existen clientes en la lista, buscamos el ultimo (el que tiene como ->Siguiente = NULL), y lo enganchamos allí.

¿Como eliminaremos un cliente de la estructura actual?

```
void EliminarCliente (int Cual)
{
        Cliente* ClAux;
        Cliente* ClBorrar;
        int i;

        if (Cual == 0)
        {
                Clientes = Clientes->Siguiente;
                free (Clientes);
                return;
        }

        ClAux = Clientes;
        for (i=0 ; i < Cual ; i++) ClAux = ClAux->Siguiente;
        ClBorrar = ClAux->Siguiente;
        ClAux->Siguiente = ClAux->Siguiente->Siguiente;
        free (ClBorrar);
}
```

En este planteamiento se usa únicamente la memoria estrictamente necesaria. No se asigna ni espacio para los punteros de forma estática, y la cantidad de información puede crecer tanto como permitan los recursos de la máquina. Tiene el problema del acceso directo a un elemento concreto, pero esto es una cosa que se puede arreglar con otro tipo de estructura, no ha de preocuparnos excesivamente.



En el ejemplo anterior vemos que hay parte de código que podríamos llegar a repetir si precisáramos de una lista de clientes, otra de proveedores y otra de artículos. El siguiente paso en la abstracción de los datos consistiría en crear una estructura lista genérica que encapsulara las operaciones de encadenamiento de elementos, pero es un nivel de complejidad al cual no es conveniente acceder por ahora. C++ implementa una serie de herramientas (mediante el concepto de clases) que facilitan sobremanera el manejo de este tipo de estructuras, que veremos en su momento.

Algo para pensar: ¿Qué pasaría si tuviéramos que mantener la lista de clientes ordenada a la vez por código y por nombre, por ejemplo? ¿Es posible mantener un conjunto de datos ordenados por dos criterios diferentes sin duplicar la información? La respuesta a todo ello se encuentra en los tipos abstractos de datos, en la encapsulación de la información, y C++ dispone de los elementos necesarios para hacerlo de forma segura, elegante y eficiente. (El concepto anterior es la semilla de la cual nacen las bases de datos jerárquicas).

8.6 Ventajas de la asignación dinámica.

Vamos a exponer un ejemplo en el que se aprecie claramente la utilidad de la asignación dinámica de memoria.

Supongamos que deseamos programar una serie de funciones para trabajar con matrices. Una primera solución sería definir una estructura de datos matriz, compuesta por una matriz y sus dimensiones puesto que nos interesa que nuestras funciones trabajen con matrices de cualquier tamaño. Por tanto la matriz dentro de la estructura tendrá el tamaño máximo de todas las matrices con las que queramos trabajar y como tenemos almacenadas las dimensiones trabajaremos con una matriz de cualquier tamaño pero tendremos reservada memoria para una matriz de tamaño máximo.

Estamos desperdiciando memoria. Una definición de este tipo sería:

```
typedef struct
{
        float mat[1000][1000];
        int  ancho,alto;
} MATRIZ;
```

En principio esta es la única forma de definir nuestro tipo de datos. Con esta definición una matriz 3x3 ocupará 1000x1000 floats al igual que una matriz 50x50.

Sin embargo podemos asignar memoria dinámicamente a la matriz y reservar sólo el tamaño que nos hace falta. La estructura sería ésta.

```
struct mat
{
        float *datos;
        int  ancho,alto;
};

typedef struct mat *MATRIZ;
```

El tipo MATRIZ ahora debe ser un puntero puesto que tenemos que reservar memoria para la estructura que contiene la matriz en sí y las dimensiones. Una función que nos crease una matriz sería algo así:

```
MATRIZ inic_matriz (int x,int y)
{
        MATRIZ temp;

        temp = (MATRIZ) malloc (sizeof (struct mat));
        temp->ancho = x;
        temp->alto = y;
        temp->datos = (float *) malloc (sizeof (float)*x*y);

        return temp;
}
```

En esta función se ha obviado el código que comprueba si la asignación ha sido correcta. Veamos como se organizan los datos con estas estructuras.

```
temp------->datos---------->x*x elementos
   ancho,alto
```

En nuestro programa principal, para utilizar la matriz declararíamos algo así:

```
main()
{
```

```
        MATRIZ a;

        a = inic_matriz (3,3);
        ...
        borrar_matriz(a);
}
```

Dónde borrar_matriz(a) libera la memoria reservada en inic_matriz, teniendo en cuenta que se realizaron dos asignaciones, una para la estructura mat y otra para la matriz en sí.

Otra definición posible del problema podría ser así.

```
typedef struct mat MATRIZ;

void inic_matriz (MATRIZ *p,int x,int y)
{

        p->ancho = x;
        p->alto = y;
        p->datos = (float *)malloc(sizeof (float)*x*y);
}
```

Con este esquema el programa principal sería algo como ésto:

```
main()
{
        MATRIZ a;
        inic_matriz (&a,3,3);
        .....
        borrar_matriz (&a);
}
```

Con este esquema el acceso a la matriz sería *(a.datos+x+y*a.ancho), idéntico al anterior sustituyendo los puntos por flechas ->. En el capítulo de estructuras de datos avanzados se justificará la utilización de la primera forma de implementación frente a esta segunda. En realidad se trata de la misma implementación salvo que en la primera el tipo MATRIZ es un puntero a una estructura, por tanto es necesario primero reservar memoria para poder utilizar la estructura, mientras que en la segunda implementación, el tipo MATRIZ es ya en si la estructura, con lo cual el compilador ya reserva la memoria necesaria. En el primer ejemplo MATRIZ a; define a como un puntero a una estructura struct mat, mientras que en la segunda MATRIZ a; define a como una variable cuyo tipo es struct mat.

8.7 Notas sobre la asignación de memoria:

Suele ser interesante moldear el resultado devuelto por estas funciones, además así nos evitaremos warnings. También hay que pensar que si se fragmenta mucho la memoria, es posible que nos fallen las llamadas, sobre todo si trabajamos en modo MS-DOS (640K como máximo). También suele quedar más curioso si en vez de presuponer que van a funcionar las llamadas a estas funciones, evaluar el resultado con un if, si es NULL ha fallado y podemos informar al usuario y si no continuamos, ya que un intento de escritura en una zona de memoria "extraña" y se nos puede colgar el programa (o algo peor).
Otro problemilla, referente esta vez a realloc, es que si una llamada a realloc falla nos quedaremos con una zona de memoria asignada a la que nadie apunta (Error: Null Pointer Assignement), para evitar esto, podemos por ejemplo realizar la llamada a realloc con una variable temporal.

Ejemplo:

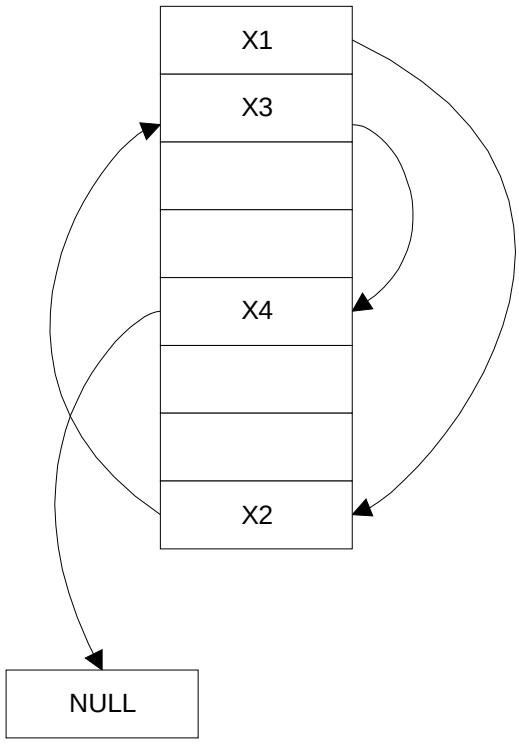
```
char *ptr = NULL;
char *tmp;

ptr = ( char * ) malloc( sizeof ( char ) * 10 );
tmp = ( char * ) realloc( ptr, sizeof ( char ) * 20 );

/* Comprobamos que no ha fallado */
if ( tmp )
        ptr = tmp;
```

8.8 Memoria dispersa:

Como hemos visto en el apartado anterior, el principal problema de la asignación de memoria dinámica, es que si empleamos varias llamadas, la memoria se queda fragmentada con lo que llegará un momento en el que llamadas fallen.

Para evitar esto emplearemos las técnicas de memoria dispersa.Lo que crearemos, será una serie de elementos en los que cada uno de ellos es capaz de direccionar al elemento que le sucede y/o al que le precede.

Ejemplo de estructura auto-referenciada.



En el ejemplo de arriba, vemos que el primer elemento (X1), apunta al siguiente (X2) y así sucesivamente hasta el último (X4) que apunta a NULL.

CAPITULO IX:

Redireccionamiento

9.1 ¿Qué es la redirección?

Cuando ejecutamos el comando dir se nos muestra el resultado en la pantalla. Sin embargo podemos hacer que el resultado se guarde en un fichero haciendo:

```
dir > resultado.txt
```

De esta forma el listado de directorios quedará guardado en el fichero resultado.txt y no se mostrará en la pantalla.

Esto es lo que se llama **redirección**. En este ejemplo lo que hemos hecho ha sido redireccionar la salida utilizando '>'.

La salida del programa se dirige hacia la salida estándar (stdout, standard output). Normalmente, si no se especifica nada, la salida standard es la pantalla.

La entrada de datos al programa se hace desde la entrada estándar (stdin, standard input). Normalmente, por defecto es el teclado.

Existe una tercera opción que es la salida de error estándar (stderr, standard error). Aquí es donde se muestran los mensajes de error del programa al usuario. Normalmente suele ser la pantalla, al igual que la de salida. ¿Por qué tenemos stderr y stdout? Porque de esta forma podemos redireccionar la salida a un fichero y aún así podemos ver los mensajes de error en pantalla.

Stdin, stdout y stderr en realidad hay que verlos como ficheros:

- Si stdout es la pantalla, cuando usemos printf, el resultado se muestra en el monitor de nuestro ordenador. Podemos imaginar que stdout es un fichero cuyo contenido se muestra en la pantalla.
- Si stdin es el teclado imaginemos que lo que tecleamos va a un fichero cuyo contenido lee el programa.

9.2 Redireccionar la salida

Ya hemos visto cómo se redirecciona la salida con el dir. Ahora vamos a aplicarlo a nuestro curso con un sencillo ejemplo. Vamos a ver un programa que toma los datos del teclado y los muestra en la pantalla. Si el usuario teclea 'salir' el programa termina:

```
#include <stdio.h>
int main()
{
        char texto[100];

        gets(texto);
        do
        {
                printf( "%s\n",texto );
                gets(texto);
        }       while ( strcmp(texto, "salir") != 0 );
        fprintf( stderr, "El usuario ha tecleado 'salir'" );
}
```

En este programa tenemos una función nueva: **fprintf**. Esta función permite escribir en el fichero indicado. Su formato es el siguiente:

```
int fprintf(FILE *fichero, const char *formato, ...);
```

Ya veremos esta función más adelante. Por ahora nos basta con saber que funciona como printf pero nos permite indicar en qué fichero queremos que se escriba el texto. En nuestro ejemplo fichero=stderr.

Como ya hemos visto antes stderr es un fichero. Stderr es la pantalla, así que, si escribimos el texto en el fichero stderr lo podremos ver en el monitor.

Si ejecutamos el programa como hemos hecho hasta ahora tendremos el siguiente resultado (en negrita lo que tecleamos nosotros):

```
primera línea
primera línea
segunda
segunda
esto es la monda
esto es la monda
salir
El usuario ha tecleado 'salir'
```

Como vemos el programa repite lo que tecleamos.

Si ahora utilizamos la redirección '> resultado.txt' el resultado por pantalla será (en negrita lo que tecleamos nosotros):

```
primera línea
segunda
esto es la monda
salir
El usuario ha tecleado 'salir'
```

NOTA: Seguimos viendo el mensaje final (El usuario ha tecleado 'salir') porque stderr sigue siendo la pantalla.

y se habrá creado un fichero llamado resultado.txt cuyo contenido será:

```
primera línea
segunda
esto es la monda
```

Si no hubiésemos usado stderr el mensaje final hubiese ido a stdout (y por lo tanto al fichero), así que no podríamos haberlo visto en la pantalla. Hubiera quedado en el fichero resultado.txt.

9.3 Redireccionar la salida con >>

Existe una forma de redireccionar la salida de forma que se añada a un fichero en vez de sobreescribirlo. Para ello debemos usar '>>' en vez de '>'. Haz la siguiente prueba:

```
dir > resultado.txt
dir >> resultado.txt
```

Tendrás el listado del directorio dos veces en el mismo fichero. La segunda vez que llamamos al comando dir, si usamos >>, se añade al final del fichero.

Nota: Todo esto nos sirve como una introducción al mundo de los ficheros, pero puede no ser la forma más cómoda para trabajar con ella (aunque como hemos visto es muy sencilla). El tema de los ficheros lo veremos más a fondo en el siguiente capítulo.

9.4 Redireccionar la entrada

Ahora vamos a hacer algo curioso. Vamos a crear un fichero llamado entrada.txt y vamos a usarlo como entrada de nuestro programa. Vamos a redireccionar la entrada al fichero entrada.txt. El fichero entrada.txt debe contener lo siguiente:

```
Esto no lo he tecleado yo.
Se escribe sólo.
Qué curioso.
salir
```

Es importante la última línea 'salir' porque si no podemos tener unos resultados curiosos (en mi caso una sucesión infinita de 'Qué curioso.'.

Para cambiar la entrada utilizamos el símbolo '<'. Si nuestro programa lo hemos llamado stdout.c haremos:

```
# stdout < entrada.txt
```

y tendremos:

```
primera línea
segunda
esto es la monda
El usuario ha tecleado 'salir'
```

Increíble, hemos escrito todo eso sin tocar el teclado. Lo que sucede es que el programa toma los datos del fichero entrada.txt en vez del teclado. Cada vez que encuentra un salto de línea (el final de una línea) es equivalente a cuando pulsamos el 'Enter'.

Podemos incluso hacer una doble redirección: Tomaremos como entrada entrada.txt y como salida resultado.txt:

```
stdout < entrada.txt > resultado.txt
```

NOTA: Cambiando el orden también funciona:

```
stdout > resultado.txt < entrada.txt
```

9.5 Redireccionar desde el programa - freopen

Existe una forma de cambiar la salida estándar desde el propio programa. Esto se puede conseguir utilizando la función **freopen**:

```
FILE *freopen(const char *nombre_fichero, const char *modo, FILE *fichero);
```

Esta función hace que el fichero al que apunta fichero se cierre y se abra pero apuntando a un nuevo fichero llamado nombre_fichero. Para redireccionar la salida con este método podemos hacer:

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

```
int main()
{
	char texto[100];
	freopen( "resultado.txt","wb",stdout );
	gets(texto);
	do
	{
		printf( "%s\n",texto );
		gets(texto);
	}	while ( strcmp(texto, "salir") != 0 );
	fprintf( stderr, "El usuario ha tecleado 'salir'" );
}
```

En este programa la función freopen cierra el fichero stdout, que es la pantalla, y lo abre apuntando al fichero resultado.txt. Ya veremos esta función más adelante.

CAPITULO X:

Operaciones con ficheros

10.1 Introduccion

Hasta el capítulo anterior no habíamos visto ninguna forma de guardar permanentemente los datos y resultados de nuestros programas. En este capítulo vamos a verlo mediante el manejo de ficheros.

En el capítulo anterior usábamos la redirección para crear ficheros. Este es un sistema poco flexible para manejar ficheros. Ahora vamos a crear y modificar ficheros usando las funciones estándar del C.

Es importante indicar que los ficheros no son únicamente los archivos que guardamos en el disco duro, en C todos los dispositivos del ordenador se tratan como ficheros: la impresora, el teclado, la pantalla,...

Un fichero no es más que una serie de datos seguidos, almacenados en un soporte, que se referencia por un nombre.

La entrada y salida a ficheros es uno de los aspectos más delicados de cualquier lenguaje de programación, pues suelen estar estrechamente integradas con el sistema operativo. Los servicios ofrecidos por los sistemas operativos varían enormemente de un sistema a otro. Las librerías del C proporcionan un gran conjunto de funciones, muchas de ellas descritas en el libro de Kernighan y Ritchie y otras derivadas de los servicios que ofrece el Unix.

En C hay dos tipos de funciones de entrada/salida a ficheros. Las primeras son derivadas del SO Unix y trabajan sin buffer. Las segundas son las que fueron estandarizadas por ANSI y utilizan un buffer intermedio. Además, hacen distinciones si trabajan con ficheros binarios o de texto.

- **Funciones de alto nivel o estándar.** Son las más utilizadas, por su fácil implementación. Al programador no le hace falta controlar los buffers ni las conversiones de datos (más se verá qué son los buffers, y la problemática de la conversión de datos).

- **Funciones de bajo nivel.** Son mucho más próximas a las funciones que utiliza el sistema operativo para gestionar archivos; requieren más control por parte del programador, el código, por decirlo de alguna manera, se hace más duro. Pero a veces, es más conveniente la utilización de estas funciones ya que permiten un grado de control más alto, en cuanto a como y cuando se han de guardar los datos, además la velocidad de los accesos a disco es más rápida.

Las funciones del C no hacen distinción si trabajan con un terminal, cinta o ficheros situados en un disco. Todas las operaciones se realizan a través de streams. Un stream está formado por una serie ordenada de bytes. Leer o escribir de un fichero implica leer o escribir del stream. Para realizar operaciones se debe asociar un stream con un fichero, mediante la declaración de un puntero a una estructura FILE. En esta estructura se almacena toda la información para interactuar con el SO. Este puntero es inicializado mediante la llamada a la función fopen(), para abrir un fichero.

Cuando se ejecuta todo programa desarrollado en C hay tres streams abiertos automáticamente. Estos son:

- stdin: dispositivo de entrada estándar
- stdout: dispositivo de salida estándar
- stderr: dispositivo de salidad de error estándar

Si arrancamos un programa en C bajo MS-DOS abre 5 ficheros como mínimo:

- stdin Entrada estandar (teclado)
- stdout Salida estandar (pantalla)
- stderr Salida de errores (pantalla)
- stdaux Salida auxiliar (com)
- stdprn salida impresora estandar (lpt1)

De estos 5, los 3 primeros son estándar y los 2 últimos son del MS-DOS.

Al finalizar el programa, bien volviendo de la función main al sistema operativo o bien por una llamada a exit(), todos los ficheros se cierran automáticamente. No se cerrarán si el programa termina a través de una llamada a abort() o abortando el programa. Estos tres ficheros no pueden abrirse ni cerrarse explicítamente.

Normalmente estos streams trabajan con el terminal, aunque el sistema operativo permite redireccionarlos. Las funciones printf() y scanf() que hemos visto, utilizan stdout y stdin respectivamente.

Los archivos se pueden abrir en modo texto, o en modo binario. Esta distinción tiene origen en las primeras versiones de C desarrolladas para el sistema operativo UNIX, que utilizaba el modo texto; después MS-DOS utilizó el modo binario. Con el lenguaje C se puede abrir un archivo de las dos formas. Las diferencias entre el modo texto y el modo binario son básicamente dos.

- text stream (Flujos de texto): son una sucesión de caracteres originado en líneas que finalizan con un carácter de nueva-línea (newline). En estos flujos puede no haber una relación de uno a uno entre los caracteres que son escritos (leídos) y los del dispositivo externo, por ejemplo, una nueva-línea puede transformarse en un par de caracteres (un retorno de carro y un carácter de salto de línea).
  Como en modo texto, los números se guardan en disco como cadenas de caracteres; por ejemplo, el número 6590 se guarda dígito a dígito, esto quiere decir el carácter 6, el carácter 5, el carácter 9 y el carácter 0, cada dígito se guarda en un byte, ocupando, en este caso concreto, cuatro bytes. Los archivos de texto se caracterizan por ser planos, es decir, todas las letras tienen el mismo formato y no hay palabras subrayadas, en negrita, o letras de istinto tamaño o ancho.

- Flujos binarios: son flujos de bytes que tienen una correspondencia uno a uno con los que están almacenados en el dispositivo externo. Esto es, no se presentan desplazamientos de caracteres. Además el número de bytes escritos (leídos) es el mismo que los almacenados en el dispositivo externo.
  Mejor dicho, en modo binario, un número se guarda tal y como está en memoria. Un int, por ejemplo el número visto anteriormente (6590), se guarda tal y como se representa en memoria, y con dos bytes (que es lo que ocupa un dato tipo int).

- Esta diferencia de flujos es importante tenerla en cuenta al leer ficheros de discos. Supongamos que tenemos un fichero de disco con 7 caracteres donde el cuarto carácter es el carácter fin de fichero (en sistema operativo DOS es el carácter con código ASCII 26). Si abrimos el fichero en modo texto, sólo podemos leer los 3 primeros caracteres, sin embargo, si lo abrimos en modo binario, leeremos los 7 caracteres ya que el carácter con código ASCII 26 es un carácter como cualquier otro. Ejemplos de estos archivos son Fotografías, imágenes, texto con formatos, archivos ejecutables (aplicaciones), etc.

Respecto a la velocidad de la memoria y de la CPU, los dispositivos de entrada y salida son muy lentos. Puede haber tres, cuatro y hasta cinco órdenes de magnitud entre la velocidad de la CPU y la de un disco duro. Además una operación de entrada y salida puede consumir una cantidad importante de recursos del sistema. Por ello, conviene reducir en número de lecturas y escrituras a disco. La mejor forma de realizar esto es mediante un *buffer*. Un buffer es una área de memoria en la cual los datos son almacenados temporalmente, antes de ser enviados a su destino. Por ejemplo, las operaciones de escritura de caracteres a un fichero se realizan sobre el *buffer* del *stream*. Unicamente cuando se llena el *buffer* se escriben todos los caracteres sobre el disco de una vez. Esto ahorra un buen número de operaciones sobre el disco. Las funciones del C nos permiten modificar el tamaño y comportamiento del *buffer* de un *stream*.

Resumen de lo anterior

Como todo lo que acabamos de decir puede resultar un poco confuso a las personas que tienen poca experiencia en C, vamos a hacer un pequeño resumen en términos generales:

- En C, cualquier cosa externa de la que podemos leer o en la que podemos escribir datos es un fichero.
- El programador escribe (lee) datos en estos ficheros a través de los flujos de cada fichero. De esta forma el programador escribe (lee) los datos de la misma forma en todos los tipos de ficheros independientemente del tipo de fichero que sea.
- Aunque conceptualmente todos los flujos son iguales, en realidad hay dos tipos: flujos de texto y flujos binarios.
- Hay tres flujos de texto predefinidos que se abren automáticamente al principio del programa: stdin, stdout y stderr. Estos tres flujos se cierran automáticamente al final del programa.

10.2 Pasos para operar con un fichero

Para utilizar las funciones de ficheros se debe incluir el fichero stdio.h. Este define los prototipos de todas las funciones, la declaración de la estructura FILE y algunas macros. Una macro importante es EOF, que es el valor devuelto por muchas funciones cuando se llega al final de fichero.

Los pasos a realizar para realizar operaciones con un fichero son los siguientes:

10.2.1 CREAR FICHERO

Crear un nombre interno de fichero. Esto se hace en C declarando un puntero de fichero (o puntero a fichero). Un puntero de fichero es una variable puntero que apunta a una estructura llamada FILE.

Esta estructura incluye entre otras cosas información sobre el nombre del archivo, la dirección de la zona de memoria donde se almacena el fichero, tamaño del buffer, toda la información necesaria para poder trabajar con un fichero. El contenido de esta estructura es dependiente de la implementación de C y del sistema

Definición de la estructura FILE:

```
typedef struct
{
        int   _cnt;
        char *_ptr;
        char *_base;
        int   _bufsiz;
        int   _flag;
        int   _file;
        char *_name_to_remove;
        int   _fillsize;
}       FILE;
```

Ejemplo:

```
FILE *pf; /* pf es un puntero de fichero */
```

10.2.2 Abrir Fichero

En lenguaje C, la funcion para abrir un archivo es fopen(). Esta toma dos *strings* como parámetros. El prototipo se encuentra en el fichero stdio.h y es:

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

Nombre_fichero: es el nombre que tiene el archivo en el disco; se puede poner la vía de acceso completa, pero teniendo en cuenta que la barra invertida (\) hay que repetirla en una cadena de caracteres.
Modo: es la manera como se abre el archivo.

Si el fichero con nombre nombre_fichero no se puede abrir devuelve NULL.

Modo	Interpretación
r	Abre un fichero de texto existente para lectura. La lectura se realiza al inicio del fichero.
w	Crear un fichero texto para escritura. Si el fichero existe se inicializa y sobreescribe
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 sin borrar los existentes.
rb	Abrir un fichero binario para lectura
wb	Crear un fichero binario para escritura
ab	Añadir información a un fichero binario
r+	Abrir un fichero texto para lectura/escritura
w+	Crear un fichero texto para lectura/escritura. Si existe, lo sobreescribe.
a+	Abrir un fichero texto para lectura/escritura. Si no existe lo crea.
rb+	Abrir un fichero binario para lectura/escritura
wb+	Crear un fichero binario para lectura/escritura
ab+	Abrir un fichero binario para lectura/escritura
r+b	Abre un archivo binario para lectura / escritura.
w+b	Crea un archivo binaro para lectura / escritura.
a+b	Abre o Crea un archivo binario para añadir información

Ejemplo:

```
FILE *fichero;
fichero = fopen ("c:\\autoexec.bat", "r");
```

Abrir un archivo, significa indicarle al sistema que establezca una conexión entre el lugar donde se encuentran los datos en el disco, y una zona de memoria; esta zona de memoria es el famoso buffer. Cuando se lee un archivo, lo que está pasando realmente es que se lee del buffer, de la misma manera que cuando se escribe en un archivo, se está escribiendo en el buffer. Y todo esto para qué? Imagine que cada vez que se quisiera guardar un carácter, este se guardara directamente en el disco; el proceso seria realmente lento. Los datos, no se guardan directamente en el disco, se guardan en un buffer (recuerde, zona de memoria), y cuando el buffer está lleno, es cuando los datos pasan al disco. De la misma manera, cuando se lee un archivo, todo un bloque de datos del disco pasan al buffer, y desde el buffer, es de donde se leen.

Quien pasa los datos del buffer al archivo de disco y viceversa?.
El sistema operativo. En operaciones de escritura a archivos, cuando el sistema detecta que un buffer está lleno, pasa los datos que contiene hacia el disco. De la misma manera, cuando el sistema detecta que se han leído todos los datos contenido en un buffer, procede a la lectura del disco para llenar el buffer con nuevos datos.

Cual es el tamaño del buffer ?
El tamaño del buffer se establece al abrir un archivo; cuando se utilizan las funciones E/S a bajo nivel el tamaño lo puede establecer el programador.

Los valores válidos para el parámetro modo son:

10.2.3 Comprobar si está abierto

Una cosa muy importante después de abrir un fichero es comprobar si realmente está abierto. El sistema no es infalible y pueden producirse fallos: el fichero puede no existir, estar dañado o no tener permisos de lectura.

Si intentamos realizar operaciones sobre un puntero tipo FILE cuando no se ha conseguido abrir el fichero puede haber problemas. Por eso es importante comprobar si se ha abierto con éxito.

Si el fichero no se ha abierto el puntero fichero (puntero a FILE) tendrá el valor NULL, si se ha abierto con éxito tendrá un valor distinto de NULL. Por lo tanto para comprobar si ha habido errores nos fijamos en el valor del puntero:

```
if (pf == NULL) /* siempre se debe hacer esta comprobación*/
{
        puts ("No se puede abrir fichero.");
        exit (1);
}
```

Si fichero==NULL significa que no se ha podido abrir por algún error. Lo más conveniente es salir del programa. Para salir utilizamos la función exit(1), el 1 indica al sistema operativo que se han producido errores.

10.2.4 OPERACIONES

Realizar las operaciones deseadas con el fichero como pueden ser la escritura en él y la lectura de él. Las funciones que disponemos para hacer esto las veremos un poco más adelante.

10.2.5 CERRAR EL FICHERO

Una vez realizadas todas las operaciones deseadas sobre el fichero hay que cerrarlo.
Aunque el C cierra automáticamente todos los ficheros abiertos al terminar el programa, es muy aconsejable cerrarlos explícitamente.

Se realiza mediante la llamada a la función fclose() que cierra un fichero determinado o fcloseall() que cierra todos los archivos abiertos. Estas funciones escribe la información que todavía se encuentre en el buffer y cierra el archivo a nivel de MS-DOS.

```
#include <stdio.h>
int fclose (FILE *pf);
int fcloseall();
```

La función fclose() cierra el fichero asociado con el flujo pf y vuelca su buffer.

Si fclose() se ejecuta correctamente devuelve el valor 0, EOF si se produce algún error, estos problemas se pueden producir si el disco está lleno, por ejemplo. La comprobación del valor devuelto no se hace muchas veces porque no suele fallar, pero seria de la siguiente manera:

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

Cuando se cierra un archivo, por decirlo de alguna manera, se corta la conexión entre el buffer de memoria, y el archivo de disco. Es muy importante cerrar un archivo, sobretodo si se está escribiendo información, el sistema guarda en el disco toda la información contenida en el buffer antes de cerrar. Si se finaliza un programa antes de cerrar un archivo, es posible que parte de la información no sea guardada o actualizada.

10.3 Comprobar fin de fichero - feof

Cuando entramos en el bucle while, la lectura se realiza hasta que se encuentre el final del fichero. Para detectar el final del fichero se pueden usar dos formas:

- con la función feof()
- comprobando si el valor de letra es EOF.

La función feof tiene la siguiente forma:

```
int feof(FILE *fichero);
```

Esta función comprueba si se ha llegado al final de fichero en cuyo caso devuelve un valor distinto de 0. Si no se ha llegado al final de fichero devuelve un cero. Por eso lo usamos del siguiente modo:

```
while ( feof(fichero)==0 )
```

o

```
while ( !feof(fichero) )
```

La segunda forma que comentaba arriba consiste en comprobar si el carácter leído es el de fin de fichero EOF:

```
while ( letra!=EOF )
```

Cuando trabajamos con ficheros de texto no hay ningún problema, pero si estamos manejando un fichero binario podemos encontrarnos EOF antes del fin de fichero. Por eso es mejor usar feof.

En el final de un archivo modo texto siempre se grava el carácter ASCII 26 (llamado EOF de End Of File). Si se leen datos de un archivo en modo texto, se puede determinar el final cuando se encuentre este carácter; si el archivo está en modo binario, no se puede utilizar esta técnica para determinar el final. Piense que en un archivo binario el número 26 se guardará como tal, no como los caracteres 2 y 6, como pasa en modo texto.

Hay otra diferencia con el carácter final de línea. Cuando se detecta un final de línea en modo texto, realmente, se guardan dos caracteres el 13 (CR), y el 10 (LF); en modo binario, esta transformación no se realiza.

10.4 Lectura Y Escritura De Ficheros – E/S Estandar

En la E/S estándar se utilizaran funciones de alto nivel; recuerde que son más sencillas de utilizar que las de bajo nivel, pero que también son menos flexibles.

10.4.1 Archivos de texto

10.4.1.1 Escritura de caracteres en el fichero – putc y fputc

Estas dos funciones escriben un carácter en el fichero abierto por el puntero que se pone como parámetro. Si todo se produce correctamente la función devuelve el propio carácter, si hay algún error devuelve EOF. La cabecera que utiliza es <stdio.h>.

```
int putc   (int c , FILE *fichero);
int putc   ('caracter' , FILE *fichero);
```

```
int fputc ( int c , FILE *fichero );
int putc   (‘caracter’ , FILE *fichero);
```

donde c contiene el carácter que queremos escribir en el fichero y el puntero fichero es el fichero sobre el que trabajamos.

Ejemplo 1:

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

main()
{
	FILE *fichero=fopen("C:\\HOLA.txt", "w");
	int i=0;
	char ch[]= "Yo soy peruano carajo!";

	/* Compruebo si el archivo se puede crear*/
	if(fopen==NULL)
	{
		printf("No se pudo abrir el fichero");
		exit (0);
	}

	/*Escribo en el fichero con putc*/
	while(ch[i]!='\0')
	{
		putc(ch[i],fichero);
		i++;
	}
	fclose(fichero);
}
```

Ejemplo 2:

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

main()
{

	char Car;
	FILE *pArchivo;

	pArchivo=fopen("C:\\Texto.Txt","w"); /* Crear archivo*/

	if (pArchivo != NULL)		/* Si no se ha producido error*/
	{
		Car=getche();
		while (Car != '\r')	/*Mientras no se pulse ENTER*/
		{
			putc(Car, pArchivo);  /* Guardar carácter.*/
```

```
                Car=getche();
            }
            fclose(pArchivo);       /* Cerrar archivo*/
        }
        else    /* Se ha producido un error */
            puts("Problemas en la creación del archivo");

}
```

10.4.1.2 Lectura de caracteres del fichero – getc y fgetc

Devuelve el carácter leído del fichero e incrementa el indicador de posición del archivo. Si se llega al final del fichero la función devuelve EOF. Todos los valores que lee los transforma a carácter. Cabecera es <stdio.h>. La función fgetc es equivalente a getc, la diferencia es que getc está implementada como macro)

El formato de la función getc y de fgetc es:

```
int getc(FILE *fichero);
int fgetc(FILE *fichero);
```

En este caso lo usamos como:

```
letra = getc( fichero );
letra = fgetc( fichero );
```

Tomamos un carácter de fichero, lo almacenamos en letra y el puntero se coloca en el siguiente carácter.

Ejemplo 1:

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

main()
{
    FILE*fichero=fopen("C:\\HOLA.txt", "r");
    char ch;
    do
    {
        /* Leemos caracteres desde el archivo */
        ch = fgetc(fichero);

        /* Escribimos en pantalla lo que leimos del archivo*/

        printf("%c",ch);
    }   while (ch != EOF);
    fclose(fichero);
}
```

Ejemplo 2:

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

main()
```

```
{
        char  Car;
        FILE *pArchivo;

        pArchivo=fopen("C:\\Hola.Txt","r");          //Abrir para leer

        if (pArchivo != NULL)                    // Si se ha podido abrir
        {
                Car=getc(pArchivo);  // Leer carácter del archivo
                while (Car != EOF)     // Mientras no se lea carácter EOF(fin archivo).
                {
                        putchar(Car);
                        Car=getc(pArchivo);
                }
                fclose(pArchivo);
        }
        else
                puts("Error, no se ha podido abrir el archivo");
}
```

Ejemplo 3:

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

int main()
{
        FILE *fichero;
        char letra;

        fichero = fopen("C:\\Hola.txt","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)==0)
        {
                printf( "%c",letra );
                letra=getc(fichero);
        }
        if (fclose(fichero)!=0)
                printf( "Problemas al cerrar el fichero\n" );
}
```

Ejemplo 4: En este ejemplo abrimos un fichero 'C:\Hola.txt' y lo copiamos en otro fichero 'destino.txt'. Además el fichero se muestra en pantalla.

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

```
{
        FILE *origen, *destino;
        char letra;

        origen=fopen("C:\\HOLA.txt","r");
        destino=fopen("C:\\destino.txt","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" );
}
```

10.4.1.3 Escritura de cadenas en el fichero – fputs

Escribe el contenido de la cadena puesta como primer parámetro de la función. El carácter nulo no se escribe en el fichero. Si se produce algún error devuelve EOF y si todo va bien devuelve un valor no negativo. La cabecera <stdio.h>. La función **fputs** trabaja junto con la función fgets que vimos en el item anterior:

```
int fputs(const char *cadena , FILE *fichero);
int fputs("texto" , FILE *fichero);
```

Ejemplo:

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

main()
{
        char cadena[80];
        FILE *pArchivo;

        pArchivo=fopen("C:\\Texto.txt","w");

        if (pArchivo != NULL)
        {
                puts("Ingrese cadenas y las guardare en un archivo");
                gets(cadena);
                while( strlen(cadena) > 0)   /* Mientras no se entre una cadena vacía.*/
                {
                        fputs(cadena,pArchivo);
```

```
                gets(cadena);
            }
            fclose(pArchivo);
        }
        else
            puts("Error, no se ha podido crear el archivo");
}
```

10.4.1.4 Lectura de cadenas de un fichero – fgets

La función fgets es muy útil para leer líneas completas desde un fichero. El formato de esta función es:

```
char *fgets(char *buffer , int longitud_max , FILE *fichero);
```

Lee un determinado número de caracteres de un fichero y los pasa a una variable de tipo cadena. Lee caracteres hasta que encuentra un carácter un salto de linea ('\n'), un EOF o hasta que lee longitud_max-1 caracteres y añade '\0' al final de la cadena. La cadena leída la almacena en buffer.

Si se encuentra EOF antes de leer ningún carácter o si se produce un error la función devuelve NULL, en caso contrario devuelve la dirección de buffer.

Ejemplo 1:

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

int main()
{
        FILE *fichero;
        char texto[100];

        fichero=fopen("C:\\HOLA.txt","r");
        if (fichero==NULL)
        {
                printf( "No se puede abrir el fichero.\n" );
                exit( 1 );
        }
        printf( "Contenido del fichero:\n" );
        fgets(texto,100,fichero);
        printf( "%s",texto );

        while (feof(fichero) ==0)
        {
                fgets(texto,100,fichero);
                printf( "%s",texto );
        }
        printf("\n\n");
        if (fclose(fichero)!=0)
                printf( "Problemas al cerrar el fichero\n" );
}
```

Ejemplo 2 :

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

```c
#include <stdlib.h>
#include <conio.h>

void letra(void);
void frase(void);

FILE *punt_fich;

void main(void)
{
	int opt;
	if((punt_fich=fopen("C:\\hola.txt","r"))==NULL)
	{
		printf("Error en la apertura");
		exit(0);
	}
	printf("1.LEER LETRA A LETRA\n");
	printf("2.LEER CADENAS\n\n");
	printf("Elegir opcion: ");
	scanf("%d",&opt);
	fflush(stdin);
	switch(opt)
	{
		case 1:
			letra();
			break;
		case 2:
			frase();
			break;
	}
	getch();
	fclose(punt_fich);
}

void letra(void)
{
	char t=0;

	for(;t!=EOF;)
	{
		t=getc(punt_fich);
		printf("%c",t);
	}
}

void frase(void)
{
	char frase[31];
	fgets(frase,30,punt_fich);
	printf("%s",frase);
}
```

Ejemplo 3:

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

#define NUM_CARACTERES 80

main()
{

        char cadena[NUM_CARACTERES];
        FILE *pArchivo;

        pArchivo=fopen("C:\\Hola.Txt","r");

        if (pArchivo != NULL)
        {
                fgets(cadena, NUM_CARACTERES, pArchivo);
                while (!feof(pArchivo))  // Mientras se hayan leídocaracteres.
                {
                        puts(cadena);
                        fgets(cadena,NUM_CARACTERES, pArchivo);
                }
                fclose(pArchivo);
        }
        else
                puts("Error, no se ha podido abrir el archivo");
}
```

10.4.1.5 Entrada/Salida con formato – fprintf, fscanf

Estas funciones son muy útiles cuando se han de guardar datos de diferentes tipos en un archivo de texto. Suponga que quiere guardar el nombre, la edad y la nota de un individuo; el nombre es una cadena de caracteres, pero la edad y la nota son datos del tipo int. Utilizando estas funciones, puede guardar y leer datos de diferentes tipos. Piense que los datos siempre se guardan en formato carácter, tanto si son del tipo char, como si son del tipo int, float, o cualquier otro tipo numérico.

Estas dos funciones trabajan igual que sus equivalentes printf y scanf. La única diferencia es que podemos especificar el fichero sobre el que operar (si se desea puede ser la pantalla para fprintf o el teclado para fscanf).

La función fprintf se comporta exactamente a printf, excepto en que toma un argumento más que indica el stream por el que se debe realizar la salida. De hecho, la llamada printf("x") es equivalente a fprintf (stdout, "x").

```
fprintf(puntero_fichero,“texto”);
fprintf(puntero_fichero“identificador”,var);
fprintf(puntero_fich“ident(es)_formato”,variable(s));

int fscanf (FILE *pfichero, const char *formato, ...);
int fscanf(pArchivo, “especificadores de formato”, variables);
```

Ejemplo 1:

fprintf(stdout, “Hola peña! %d\n”, numeraco)

Imprimirá por la salida estándar (normalmente la pantalla) el mensaje y la variable numeraco.

Hay que tener en cuenta, que en C, (al igual que UNIX ya que C se diseñó para crear el S.O. UNIX) todo dispositivo E/S es tratado como un fichero, luego podremos usar esta función para trabajar con cualquier dispositivo de este tipo.

Ejemplo 2:

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

void letra(void);
void frase(void);

FILE *punt_fich;

void main(void)
{
	int opt;
	if((punt_fich=fopen("C:\\HOLA2.txt","w"))==NULL)
	{
		printf("Error en la apertura");
		exit(0);
	}

	printf("1.INTRODUCIR LETRA A LETRA\n");
	printf("2.INTRODUCIR CADENA A CADENA\n\n");
	printf("Elegir opcion: ");
	scanf("%d",&opt);
	fflush(stdin);
	switch(opt)
	{
		case 1:
			letra();
			break;
		case 2:
			frase();
			break;
	}
	fclose(punt_fich);
}

void letra(void)
{
	char t;

	for(;t!='$';)
	{
		printf(":");
		t=getchar();
```

```
                fputc(t,punt_fich);
                fflush(stdin);
        }
}

void frase(void)
{
        char *frase;

        do
        {
                printf(":");
                gets(frase);
                fprintf(punt_fich,"%s\n",frase);
                // fputs(frase,punt_fich);
                fflush(stdin);
        }       while(strcmp(frase,"$"));
}
```

Ejemplo 3:

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

main()
{
        char Nombre[10];
        char mas;
        int Edad, Nota;
        FILE *pArchivo;

        pArchivo=fopen("C:\\Datos.Txt","w");

        if (pArchivo != NULL)
        {
                do
                {
                        flushall() ;
                        puts("Entre el Nombre : ");
                        gets(Nombre);
                        puts("Entre Edad : ");
                        scanf("%d", &Edad);
                        puts("Entre la Nota : ");
                        scanf("%d", &Nota);
                        fprintf(pArchivo,"%s %d %d",Nombre, Edad, Nota);
                        puts("Quiere entrar más datos : ");
                        mas=getch();
                }       while (mas == 's' || mas == 'S');
                fclose(pArchivo);
        }
        else
                puts("Error, no se ha podido crear el archivo");
}
```

Ejemplo 4:

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

main()
{
        char Nombre[10];
        int Edad, Nota;
        FILE *pArchivo;

        pArchivo=fopen("C:\\Datos.Txt","r");

        if (pArchivo != NULL)
        {
                fscanf(pArchivo,"%s %d %d", Nombre, &Nota, &Edad);
                while (!feof(pArchivo))
                {
                        printf("Nombre : %s, Edad : %d, Nota : %d \n", Nombre, Nota,Edad);
                        fscanf(pArchivo,"%s %d %d", Nombre, &Nota, &Edad);
                }
                fclose(pArchivo);
        }
        else
                puts("Error, no se ha podido abrir el archivo");
}
```

10.4.2 Archivos Binarios - Entrada/Salida de registros

Todas les funciones vistas hasta ahora, guardan los números en forma de cadenas de caracteres ocupando cada dígito un byte. Esta es una manera, no demasiado eficiente. Si se quisiera guardar arrays numéricos, no quedaría otro remedio que tratar cada elemento de manera individual; si se quisiera guardar estructuras, se tendría que hacer por partes. La solución a estos problemas está en utilizar las funciones de E/S de registros. Estas funciones guardan los valores en formato binario, así, un entero se guardará siempre en dos bytes, un float en cuatro bytes, etc. La E/S de registros permite escribir o leer cualquier cantidad de bytes a la vez, no está limitada a hacerlo carácter a carácter, ni a hacerlo en una cadena, ni en grupos de valores formateados utilizando las funciones fprintf() y fscanf().

En los dos últimos programas estudiados se utilizaban las funciones fprintf() y fscanf() para gestionar los datos correspondientes a alumnos de una clase. Se repetirán estos programas, pero utilizando una estructura para agrupar los datos de cada alumno; las funciones que se utilizaran permiten leer y escribir en el disco una estructura (o más) cada vez; también permiten guardar o leer todo un array a la vez. Recuerde que estas funciones, guardan los datos en formato binario.

10.4.2.1 Escritura de registro - fwrite

Escribe los datos de una estructura a un fichero binario e incrementa la posición del archivo en el número de caracteres escritos. Hay que tener en cuenta que el modo de apertura del fichero debe ser binario.
Esta función tiene el formato siguiente:

```
size_t  fwrite (&Var Estructura, Número de Bytes, Número de registros, pArchivo);
```

El primer argumento es la dirección de la estructura que contiene los datos que vamos a escribir en el fichero.
El segundo es el número de bytes que se han de guardar a la vez.
El tercero es el número de registros a guardar.
El último argumento es un puntero hacia el buffer donde el sistema pone los datos del archivo.

Ejemplo 1:

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

struct Alumno
{
        char Nombre[10];
        int Edad;
        int Nota;
};

int main()
{
        struct Alumno Estudiante;
        FILE *pArchivo;
        char mas;

        pArchivo = fopen("C:\\Clase.Dat","ab");

        if (pArchivo != NULL)
        {
                do
                {
                        flushall();
                        puts("Nombre Alumno:");
                        gets(Estudiante.Nombre);
                        puts("Edad : ");
                        scanf("%d", &Estudiante.Edad);
                        puts("Nota : ");
                        scanf("%d", &Estudiante.Nota);
                        fwrite (&Estudiante, sizeof (struct Alumno), 1, pArchivo);
                        puts("Mas Alumnos : ");
                        mas = getch();
                }       while (mas == 's' || mas == 'S');
                fclose(pArchivo);
        }
        else
                puts("Error, no se ha podido crear el archivo");
        return 0;
}
```

La función sizeof() retorna el número de bytes del argumento que se le pasa, por ejemplo:

```
printf("%d", sizeof (int ));     /*retorna 2 o 4 dependiendo del SO*/
printf("%d", sizeof (char));     /*retorna 1*/
```

```
printf("%d", sizeof (struct Alumnos)),  /*retorna 14 (10 del array Nombre más dos de
                                          Nota (int) y más dos de Edad (int)).*/
```

Ejemplo 2:
Un programa de agenda que guarda el nombre, apellido y teléfono de cada persona mediante fwrite usando la estructura registro. Abrimos el fichero en modo 'a' (append, añadir), para que los datos que introducimos se añadan al final del fichero. Una vez abierto abrimos entramos en un bucle do-while mediante el cual introducimos los datos. Los datos se van almacenando en la variable registro (que es una estructura). Una vez que tenemos todos los datos de la persona los metemos en el fichero con fwrite: NOTA: El bucle termina cuando el 'nombre' se deja en blanco.

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

struct
{
        char nombre[20];
        char apellido[20];
        char telefono[15];
} registro;

int main()
{
        FILE *fichero;

        fichero = fopen( "C:\\nombres.txt", "a" );
   do
        {
                printf( "Nombre: " ); fflush(stdout);
                gets(registro.nombre);
                if (strcmp(registro.nombre,""))
                {
                        printf( "Apellido: " ); fflush(stdout);
                        gets(registro.apellido);
                        printf( "Telefono: " ); fflush(stdout);
                        gets(registro.telefono);
                        fwrite(&registro,sizeof (registro),1,fichero);
                }
        }       while (strcmp(registro.nombre,"")!=0);
        fclose(fichero);
}
```

10.4.2.2 Lectura de registros - fread()

Lee registros de un fichero binario, cada uno del tamaño especificado en la función y los sitúa en la estructura indicada en primer termino de la función. Además de leer los registros incrementa la posición del fichero. Hay que tener en cuenta el modo de apertura del fichero. Cabecera <stdio.h>.

```
size_t  fread  (&Var Estructura, Número de Bytes, Número de registros, pArchivo);
```

El primer argumento es la dirección de la estructura. donde se van a escribir los datos leídos del fichero pArchivo.

El segundo es el número de bytes que se han de leer.
El tercero es el número de registros a leer.
El último argumento es un puntero hacia el buffer donde el sistema pone los datos del archivo.
La función fread() retorna el número de registros leídos cada vez; normalmente coincide con el tercer argumento de la función. Si se ha llegado al final de archivo, este número será menor (0 en nuestro caso, ya que leemos un único registro cada vez). Comprobando esta situación podremos determinar cuando se ha de parar la lectura.

Ejemplo 1:

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

struct Alumno
{
        char Nombre[10];
        int Edad;
        int Nota;
};

int main()
{
        struct Alumno Estudiante;
        FILE *pArchivo;
        int NumRegistros;
        pArchivo = fopen("C:\\Clase.Dat","rb");
        if (pArchivo != NULL)
        {
                NumRegistros = fread(&Estudiante,sizeof (struct Alumno),1,pArchivo);
                while (NumRegistros == 1)  // Mientras se haya leido un registro
                {
                        printf("Nom : %s \n", Estudiante.Nombre);
                        printf("Edad : %d \n", Estudiante.Edad);
                        printf("Nota : %d \n ", Estudiante.Nota);
                        NumRegistros = fread(&Estudiante, sizeof (struct Alumno), 1,
pArchivo);
                }
                fclose(pArchivo);
        }
        else
                puts("Error, no se ha podido abrir el archivo");
        return 0;
}
```

TC incorpora la función feof(pArchivo) para determinar el final de archivo. Esta función retorna 0 si se ha llegado al final, en caso contrario retorna 1. El programa utilizando esta función seria:

```
fread(&Estudiante, sizeof (struct Alumno),1, pArchivo);
while (! feof(pArchivo))
{
        .
        fread(&Estudiante, sizeof (struct Alumno), 1, pArchivo);
}
```

Ejemplo 2:

Ahora vamos a leer el archivo que creamos anteriormente con fwrite. Abrimos el fichero C:\nombres.txt en modo lectura. Con el bucle while nos aseguramos que recorremos el fichero hasta el final (y que no nos pasamos). La función fread lee un registro (numero=1) del tamaño de la estructura registro. Si realmente ha conseguido leer un registro la función devolverá un 1, en cuyo caso la condición del 'if' será verdadera y se imprimirá el registro en la pantalla. En caso de que no queden más registros en el fichero, fread devolverá 0 y no se mostrará nada en la pantalla.

```
#include <stdio.h>
struct
{
        char nombre[20];
        char apellido[20];
        char telefono[15];
}       registro;

int main()
{
        FILE *fichero;

        fichero = fopen( "C:\\nombres.txt", "r" );
        while (!feof(fichero))
        {
                if (fread(&registro, sizeof (registro), 1, fichero ))
                {
                        printf( "Nombre: %s\n", registro.nombre );
                        printf( "Apellido: %s\n", registro.apellido);
                        printf( "Teléfono: %s\n", registro.telefono);
                }
        }
        fclose( fichero );
}
```

Ejemplo 3:

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

void altas(void);
void muestra(void);
FILE *fich;
struct ficha
{
        int  código;
        char nombre[25];
        char direcion[40];
        int  edad;
}cliente;

void main(void)
{
        char opcion;
```

```
	if((fich=fopen("gestion.dat","a+b"))==NULL)
	{
		printf("Error al crear fichero");
		exit(0);
	}

	do
	{
		clrscr();
		printf("Altas\n");
		printf("Consulta\n");
		printf("Salir\n\n");
		printf("Elegir opcion: ");
		scanf("%c",&opcion);
		fflush(stdin);
		switch(toupper(opcion))
		{
			case 'A':
				altas();
				break;
			case 'C':
				muestra();
				break;
		}
	}	while(toupper(opcion)!='S');
	fclose(fich);
}

void altas(void)
{
	clrscr();
	printf("Código: ");
	scanf("%d",&cliente.codigo);
	fflush(stdin);

	printf("Nombre: ");
	gets(cliente.nombre);
	fflush(stdin);

	printf("Direccion: ");
	gets(cliente.direcion);
	fflush(stdin);

	printf("Edad: ");
	scanf("%d",&cliente.edad);
	fflush(stdin);

	fwrite(&cliente,sizeof (cliente),1,fich);
}

void muestra(void)
{
	int cod_temp;
```

```
        clrscr();

        rewind(fich);  /*Nos posicionamos al comienzo del fichero*/
        printf("Código a mostrar:");
        scanf("%d",&cod_temp);

        while(!feof(fich))
        {
                fread(&cliente,sizeof (cliente),1,fich);
                if(cod_temp==cliente.codigo)
                {
                        printf("Código: %d\n",cliente.codigo);
                        printf("Nombre: %s\n",cliente.nombre);
                        printf("Direc: %s\n",cliente.direcion);
                        printf("Edad: %d\n",cliente.edad);
                        getch();
                        break;
                }
        }
}
```

10.4.2.3 Ejemplo de gestión de un archivo de registros.

Se utilizará una estructura para guardar los datos siguientes: Nombre, Edad y Población. El programa ha de realizar las accione siguientes:

- Añadir datos.
- Listar todo el archivo.
- Mostrar un nombre.
- Listar todos los registros de una población.
- Listar todos los individuos que tengan más de una determinada edad.
- El programa muestra un menú de opciones, línea opción=Menu() para qué el usuario pueda escoger una determinada acción.

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

#define NOM_ARCHIVO "C:\\Datos.Dat"

struct Registro
{
        char Nombre[10];
        char Poblacion[15];
        int Edad;
};

char Menu(void);
void Anyadir(void);
void Listar (void);
void Listar_Poblacion(void);
void Listar_Edad(void);
```

```
main()
{
	char opcion;

	do
	{
		opcion = Menu();
		switch(opcion)
		{
			case '1' : Anyadir();
				break;
			case '2': Listar();
				break;
			case '3': Listar_Poblacion();
				break;
			case '4': Listar_Edad();
				break;
			default:
				if (opcion != '5')
					puts ("Numero de opción incorrecto");
		}
	}	while (opcion != '5');
	getch();
}

char Menu(void)
{
	char opcion;
	system("cls");
	puts("\n\t\t\tMENU PRINCIPAL\n\n");
	puts("1.- Añadir Registros");
	puts("2.- Listar Registros");
	puts("3.- Listar una Poblacion");
	puts("4.- Listar a partir de una edad");
	puts("5.- Acabar");
	opcion=getche();
	return(opcion);
}

void Anyadir(void)
{

	FILE *pArchivo;
	struct Registro Ficha;
	char continuar;

	pArchivo = fopen(NOM_ARCHIVO,"ab");

	if (pArchivo != NULL)
	{
		puts ("Entrar Datos ");
```

```
                do
                {
                        flushall();
                        puts("Nombre : ");
                        gets(Ficha.Nombre);
                        puts("Poblacion : ");
                        gets(Ficha.Poblacion);
                        puts("Edad : ");
                        scanf("%d", &Ficha.Edad);
                        fwrite(&Ficha, sizeof (struct Registro), 1, pArchivo);
                        puts("Más registros <S/N>: ");
                        continuar = getche();
                }       while(continuar == 'S' || continuar == 's');
                fclose(pArchivo);
        }
        else
                puts ("Error, no se ha podido abrir o crear el archivo ");
        getch();
}

void Listar (void)
{
        FILE *pArchivo;
        struct Registro Ficha;
        pArchivo=fopen(NOM_ARCHIVO,"rb");
        if (pArchivo != NULL)
        {
                fread(&Ficha, sizeof (struct Registro), 1, pArchivo);
                while (!feof(pArchivo))
                {
                        printf("%s, %s, %d \n", Ficha.Nombre, Ficha.Poblacion,
                                                        Ficha.Edad);
                        fread(&Ficha, sizeof (struct Registro), 1, pArchivo);
                }
                fclose(pArchivo);
        }
        else
                puts("Error, no se ha podido abrir el archivo");
        getch();
}

void Listar_Poblacion(void)
{
        FILE *pArchivo;
        struct Registro Ficha;
        char B_Poblacion[15];
        pArchivo=fopen(NOM_ARCHIVO,"rb");
        if (pArchivo != NULL)
        {
                puts("Entre la Poblacion a listar : ");
                gets(B_Poblacion);
                fread(&Ficha, sizeof (struct Registro), 1, pArchivo);
                while (!feof(pArchivo))
```

```
                {
                        if (strcmp(B_Poblacion, Ficha.Poblacion) == 0)
                        printf("%s, %d \n ", Ficha.Nombre, Ficha. Edad);
                        fread( &Ficha, sizeof (struct Registro), 1, pArchivo);
                }
                fclose(pArchivo);
        }
        else
                puts("Error, no se ha podido abrir el archivo");
        getch();
}

void Listar_Edad(void)
{
        FILE *pArchivo;
        struct Registro Ficha;
        int B_Edad;
                pArchivo=fopen(NOM_ARCHIVO,"rb");
        if (pArchivo != NULL)
        {
                puts("Entrar edad a partir de la cual se lista : ");
                scanf("%d",&B_Edad);
                fread(&Ficha, sizeof (struct Registro), 1, pArchivo);
                while( !feof(pArchivo))
                {
                        if (Ficha.Edad >= B_Edad)
                                printf("%s, %s, %d \n ",Ficha.Nombre, Ficha.Poblacion,
Ficha.Edad);
                        fread(&Ficha, sizeof (struct Registro), 1, pArchivo);
                }
                fclose(pArchivo);
        }
        else
                puts("Error, No se ha podido abrir el archivo");
        getch();
}
```

10.4.3 Definicion de algunas funciones

10.4.3.1 REMOVE

Borra el archivo especificado por el Nombre_Archivo. Si su archivo está abierto, entonces primero cierrelo para luego borrarlo.

Retorna 0 si la operación se relizo con éxito, y devuelve -1 si no se pudo borrar.

La sintaxis

```
#include <stdio.h>
int remove(const char *Nombre_Archivo);
```

Ejemplo:

```
#include<stdio.h>
int main()
{
	char NOMBRE[]="C:\\Datos.txt", TEMPORAL[]="C:\\TEMP.txt";
	char texto[100];
	FILE *fichero = fopen(NOMBRE, "r");
	FILE *tempo = fopen(TEMPORAL, "w");
	if(fichero==NULL)
	{
		printf("Nose oudo abrir");
	}
	fgets(texto,100,fichero);
	fputs(texto,tempo);

	while (!feof(fichero))
	{
		fgets(texto,100,fichero);
		fputs(texto,tempo);
	}
	fclose(fichero);
	if(remove(NOMBRE)==0)
		printf("Borrado");
	fcloseall();
	return 0;
}
```

10.4.3.2 RENAME

Esta funcion permite renombrar un archivo. Retornra 0 si la operaron se realizo con éxito, y -1 si hubo algun error. La sintaxis

```
#include <stdio.h>
int rename(const char *oldname, const char *newname);
```

Ejemplo 1:

```
#include<stdio.h>
#include<stdlib.h>
int main()
{
	char NOMBRE[]="C:\\Datos.txt", TEMPORAL[]="C:\\TEMP.txt";
	char texto[100];
	FILE *fichero = fopen(NOMBRE, "r");
	FILE *tempo = fopen(TEMPORAL, "w");

	/*Verificamos si se puede abrir el archivo*/

	if(fichero==NULL)
	{
		printf("No se pudo abrir el archivo");
		exit (0);
	}
```

```
        /*Copiamos de un fichero a otro*/
        fgets(texto,100,fichero);
        fputs(texto,tempo);

        while (!feof(fichero))
        {
                fgets(texto,100,fichero);
                fputs(texto,tempo);
        }

        /*Cerramos todos los ficheros*/

        fcloseall();

        /*Borramos el archivo original*/
        remove(NOMBRE);

        /*Renombramos el archivo temporal*/
        rename(TEMPORAL,"C:\\Datos2.txt");
        return 0;
}
```

10.4.4 Acceso directo o aleatorio

En todos los ejemplos vistos hasta el momento se ha gestionado el acceso a los archivos de manera secuencial,, cuando se guardaba nueva información (caracteres, cadenas o registros) se hacia siempre al final del archivo, cuando se accedía a la información guardada en un archivo, se empezaba desde el principio, y se iba recorriendo o leyendo el archivo hasta el final.

Además del acceso secuencial también existe la posibilidad de leer o guardar directamente información en una posición determinada de un archivo. Así por ejemplo, se puede acceder directamente a la información que se encuentra en la mitad del archivo sin necesidad de leer la información anterior a esta posición.

Vamos a ver un ejemplo de cada caso. Suponga un archivo de estructuras, si quiere leer el registro número 10, en un acceso secuencial antes tendrá que leer los nueve primeros registros, mientras que en un acceso directo, puede leer directamente el registro que ocupa la posición 10.

10.4.4.1 Punteros de archivos

No confunda el puntero de archivo con el puntero que se utiliza para referenciar un archivo (la variable tipo FILE). Este último sirve para acceder al buffer donde se guarda temporalmente la información leída, o la que se tiene que escribir a disco (entre otras coses). El puntero de archivo sirve para referenciar un byte de información contenido en el disco. Cuando se abre un archivo en modo lectura, el puntero de archivo se sitúa apuntando el primer byte del archivo; cuando se abre un archivo para escritura (con el modo “a”), el puntero se sitúa apuntando al último byte.

a. REWIND

Lleva el indicador de posición al principio del archivo. No devuelve ningún valor. La cabecera que utiliza es <stdio.h>.

```
rewind(puntero_fichero);
```

b. FGETPOS

Guarda el valor actual del indicador de posición del archivo. El segundo termino es un objeto del tipo fpos_t que guarda la posición. El valor almacenado sólo es valido para posteriores llamadas a fsetpos. Devuelve DISTINTO DE CERO si se produce algún error y CERO si todo va bien. Cabecera <stdio.h>.

```
int fgetpos(puntero_fichero,&objeto_fpos_t);
```

c. FSETPOS:

Desplaza el indicador de posición del archivo al lugar especificado por el segundo termino que es un objeto fpos_t. Este valor tiene que haber sido obtenido por una llamada a fgetpos. Devuelve DISTINTO DE CERO si hay errores y CERO si va todo bien. Cabecera <stdio.h>.

```
int fsetpos(puntero_fichero,&objeto_fpos_t);
```

d. TELL

Devuelve el valor actual del indicador de posición del archivo. Este valor es el número de bytes que hay entre el comienzo del archivo y el indicador. Devuelve –1 si se produce un error. Cabecera <io.h>.

```
var_long =tell(puntero_fichero);
```

e. FSEEK

La función fseek nos permite situarnos en la posición que queramos de un fichero abierto. Cuando leemos un fichero hay un 'puntero' que indica en qué lugar del fichero nos encontramos. Cada vez que leemos datos del fichero este puntero se desplaza. Con la función fseek podemos situar este puntero en el lugar que deseemos. La Cabecera es <io.h>. Si se produce algún error al intentar posicionar el puntero, la función devuelve un valor distinto de 0. Si todo ha ido bien el valor devuleto es un 0. Su sintaxis es la siguiente:

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

pArchivo: es el puntero a la estructura FILE del archivo.

Desplazamiento: es el número de bytes desde una posición, hasta la posición donde se quiere situar el puntero de archivo. Normalmente será el número de bytes desde principio de archivo, hasta la nueva posición. Despl es un valor tipo long.

Modo: Sirve para determinar la posición desde donde se ha de contar el desplazamiento. Si pone 0 se empezará a contar desde el principio de archivo, 1 desde la posición actual, y 2 desde la posición final de archivo. También se pueden utilizar las constantes definidas a stdio.h SET_SEEK, SET_CUR y SET_POS.

CONSTANTE	VALOR	DESCRIPCIÓN
SEEK_SET	0	El puntero se desplaza desde el principio del fichero.
SEEK_CUR	1	El puntero se desplaza desde la posición actual del fichero.
SEEK_END	2	El puntero se desplaza desde el final del fichero.

Ejemplo 1: Vea la función siguiente. Pide que registro se quiere visualizar, desplaza el puntero a la posición que ocupa, y hace la lectura:

```
void Leer_Registro(void)
{
        FILE *pArchivo;
```

```
        long int Num_Reg;
        struct Alumno Estudiante;


        pArchivo=fopen(NOM_ARCHIVO, "rb");
        if (pArchivo != NULL)
        {
                puts("Entre el número de registro a leer ");
                scanf("%ld", &Num_Reg);
                if( fseek(pArchivo, Num_Reg, 0) != 0) // Si se ha podido situar el puntero
                {
                        fread(&Estudiante, sizeof (struct Alumno), 1, pArchivo);
                        printf("Nombre : %s \n", Estudiante.Nombre);
                        printf("Edad : %d \n ", Estudiante.Edad);
                        printf("Nota : %d \n", Estudiante.Nota);
                }
                else
                        puts("Error, no se ha podido desplazar a la posición indicada ");
        }
        else
                puts("Error, no se ha podido abrir el archivo");
}
```

Ejemplo 2:

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

void main(void)
{
        FILE *punte;
        clrscr();
        if((punte=fopen("hla.txt","r"))==NULL)
        {
                printf("Error de lectura");
                exit(0);
        }
        fseek(punte,7,SEEK_SET);
        printf("%c",fgetc(punte));
        getch();
        fclose(punte);
}
```

Ejemplo 3: En el siguiente ejemplo se muestra el funcionamiento de fseek. Se trata de un programa que lee la letra que hay en la posición que especifica el usuario.

```
#include <stdio.h>

int main()
{
        FILE *fichero;
        long posicion;
        int resultado;
```

```
        fichero = fopen( "origen.txt", "r" );
        printf( "¿Qué posición quieres leer? " ); fflush(stdout);
        scanf( "%D", &posicion );
        resultado = fseek( fichero, posicion, SEEK_SET );
    if (!resultado)
            printf( "En la posición %D está la letra %c.\n", posicion, getc(fichero) );
        else
            printf( "Problemas posicionando el cursor.\n" );
        fclose( fichero );
}
```

f. ftell

Esta función es complementaria a fseek, devuelve la posición actual dentro del fichero. Su formato es el siguiente:

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

El valor que nos da ftell puede ser usado por fseek para volver a la posición actual.

10.4.4.2 Opción de modificar en el programa ejemplo de gestión

El problema de modificar un registro, es que una vez encontrado, el puntero de archivo marcará el principio del registro siguiente. Vea el esquema siguiente donde se ha buscado el registro con el Nombre "Pepito":

reg1	
reg2	
Pepito	Registro Buscado
reg4	Registro actual después de encontrar "Pepito"
reg5	

Si se guardan los datos inmediatamente después de la localización del registro, será incorrecto ya que el puntero de archivo estará apuntando ya al registro siguiente, y por tanto los nuevos datos se guardaran en este y no en el localizado. En este caso se ha de resituar el puntero en el registro indicado, antes de guardar los datos modificados.

```
void Modificar (void)
{
        FILE *pArchivo;
        struct Alumno Estudiante;
        long int NumReg = 0;
        char B_Nombre[10];
        pArchivo=fopen(NOM_ARCHIVO,"rb+");  // Abrir en modo lectura escritura
        if (pArchivo != NULL)
        {
                puts("Entre Nombre a Buscar : ");
                gets(B_Nombre);
                fread( &Estudiante, sizeof (struct Alumno), 1, pArchivo);
                while (!feof(pArchivo) && strcmp(Estudiante.Nom, B_Nom) == 0)
                {
                        fread(&Estudiante, sizeof (struct Alumno), 1, pArchivo);
                        Num_Reg = Num_Reg + 1;   // Contar los registros
```

```
                }
                if (strcmp (Estudiante.Nom, B_Nom) == 0) // Nombre encontrado
                {
                        puts("Nom : "); gets(Estudiante.Nombre);
                        puts("Edad : "); scanf("%d", &Estudiante.Edad);
                        puts("Nota : ");scanf("%d", &Estudiante.Nota);
                        Num_Reg = Num_Reg - 1;    // Ajustar num de registro
                        fseek(pArchivo, Num_Reg * sizeof (struct Alumno), 0); // Resituar

                // punter
                        fwrite(&Estudiante, sizeof (struct Alumno), 1, pArchivo)
                }
                else
                        puts ("Nombre no encontrado ");
        }
        else
                puts("Error, no se ha podido abrir el archivo ");
}
```

10.4.4.3 Opción de borrar en el programa ejemplo de gestión

El problema de borrar datos en cualquier sistema de gestión de archivos (tanto si está construido en C, como en cualquier otro lenguaje o aplicación) es el de elimminarlas físicamente del disco, no por el simple hecho de eliminarlas, sino por la lentitud del proceso. Imagine por ejemplo, un archivo de 1000 registros de información de uno o más bytes cada registro, si se quiere eliminar físicamente el segundo registro de información, supone que los 998 restantes tendrán que actualizar su posición: el tercer hacia al segundo, el cuarto hacia el tercero, el quinto hacia el cuarto, y así hasta al final. Observe que cada actualización requiere una lectura y una escritura, y sólo que cada proceso lectura-escritura tarde medio segundo en realizarse, todo el proceso de eliminación tardará 500 segundos (unos ocho minutos). Para solucionar estas situaciones hay diferentes métodos. A continuación se verá uno que resulta práctico en archivos de registros.

El método consiste en definir una estructura a la cual se le añade un campo de un byte, que determinará si el registro está borrado o no. 1 significa que está borrado, y 0 que no lo está; así por ejemplo la estructura que hemos utilizado hasta ahora, quedaría:

```
struct Alumno
{
        char Nombre[10];
        int Edad;
        int Nota;
        unsigned char Borrado;
};
```

Codificaremos dos funciones, una que sirva para localizar el registro y poner el campo borrado 1, y otra que recorre todo el archivo y elimina físicamente los registros que tienen este campo a 1. Una manera fácil de implementar el segundo proceso es leer todos los registros del archivo original, y guardar en un archivo auxiliar solo los que tienen el campo borrado igual a 0, después se elimina el archivo original y se cambia el nombre del archivo auxiliar. Vea el siguiente algoritmo:

```
pOriginal:=Abrir(Original, “Lectura”);
PAuxiliar:=Abrir(Auxiliar, “Escritura”);      // el archivo se crea
Leer(Registro, pOriginal);
Mientras (No final Archivo Original)
        Inicio
```

```
                Si Registro.Borrado = 0 Entonces
                        Escribir(Registro,pAuxiliar);
                Leer(Registro, pOriginal);
        Fin;
Cerrar(pOriginal);
Cerrar(pAuxiliar);
Borrar(Original);
CanviarNombre(Auxiliar,Original);
```

Al proceso de eliminar físicamente los registros marcados se le denomina purgar.

a. Observaciones:

- Este último proceso no se ha de ejecutar cada vez que se borra un registro, tiene que haber una opción que lo active a petición del usuario, o bien se ha de codificar de tal manera que se ejecute de tanto en tanto.
- Todos los otros procesos han de gestionar la posibilidad de encontrar registros marcados como borrados de manera que estos no sean tratados. Por ejemplo, un listado no tendría que mostrar los registros que tengan el campo borrado igual a 1.

Vea a continuación las funciones codificadas en lenguaje C para borrar y purgar:

```
void Borrar (void)
{
        FILE *pArchivo;
        struct Alumno Estudiante;
        long int NumReg = 0;
        char B_Nombre[10];
        unsigned char Encontrado = 0;// Var que se pone a 1 cuando se encuentra el
                                                        //registro buscado
        pArchivo=fopen(NOM_ARCHIVO,"rb+");  // Abrir para lectura escritura
        if (pArchivo != NULL)
        {
                puts("Entre Nombre a Borrar : ");
                gets(B_Nombre);
                fread( &Estudiante, sizeof (struct Alumno), 1, pArchivo);
                if (strcmp(Estudiante.Nombre, B_Nombre)==0 && Estudiante.Borrado==0)
                        Encontrado = 1;
                while (!feof(pArchivo) && !Encontrado)
                {
                        Num_Reg = Num_Reg + 1;          // Contar los registros
                        fread(&Estudiante, sizeof (struct Alumno), 1, pArchivo);
                        if (strcmp(Estudiante.Nombre, B_Nombre) = = 0 &&
                                                Estudiante.Borrado == 0)
                                Encontrado = 1;
                }
                if (strcmp (Estudiante.Nombre, B_Nombre) == 0 && Encontrado)
                {
                        Estudiante.Borrado = 1;         // Activar marca de borrado
                        Num_Reg = Num_Reg - 1;    // Ajustar num de registro
                        fseek(pArchivo, Num_Reg * sizeof (struct Alumno), 0);
                        fwrite(&Estudiante, sizeof (struct Alumno), 1, pArchivo)
                }
                else
```

```
                    puts ("Nombre no Encontrado ");
        }
        else
                puts("Error, no se ha podido abrir el archivo ");
}
```

Observe que cuando se busca el registro, también ha de incluir la condición que el registro tratado no esté borrado:

```
if (strcmp(Estudiante.Nombre,B_Nombre) ==0  &&  Estudiante.Borrado == 0)
        Encontrado = 1;
```

Otra manera más elegante de codificar la sentencia anterior seria:

```
Encontrado = (strcmp(Estudiante.Nom,B_Nom) == 0 && Estudiante.Borrado ==0);
```

Función para eliminar físicamente los registros marcados:

```
void Purgar(void)
{
        FILE *pOriginal, *pAuxiliar;
        struct Alumno Estudiante;
        pOriginal=fopen(NOM_ARCHIVO,"rb");
        pAuxiliar=fopen("Temporal","wb");
        if (pOriginal != NULL && pAuxiliar != NULL)
        {
                fread( &Estudiante, sizeof (struct Alumno), 1, pOriginal);
                while( !feof(pOriginal))
                {
                        if (Estudiante.Borrado == 0)
                        fwrite( &Estudiante, sizeof (struct Alumno), 1, pAuxiliar);
                        fread( &Estudiante, sizeof (struct Alumno), 1, pAuxiliar);
                }
                fclose(pOriginal);
                fclose(pAuxiliar);
                remove(NOM_ARCHIVO);
                rename("Temporal",NOM_ARCHIVO);
        }
        else
                puts ("Error, no se ha podido abrir alguno de los archivos");
}
```

Para acabar, vea como quedaría la función listar:

```
void Listar (void)
{
        FILE *pArchivo;
        struct Alumno Estudiante;
        pArchivo = fopen(NOM_ARCHIVO,"rb");
        if (pArchivo != NULL)
        {
                fread( &Estudiante, sizeof (struct Alumno), 1, pArchivo);
                while (!feof(pArchivo))
                {
```

```
                    if (Estudiante .Borrado == 0)          /* Si registro no borrado*/
                    printf("%s, %d, %d \n",Estudiante.Nombre, Estudiante.Edad,
Estudiante.Nota);
                    fread( &Estudiante, sizeof (struct Alumno), 1, pArchivo);
                }
                fclose(pArchivo);
        }
        else
                puts("Error, no se ha podido abrir el archivo ");
}
```

10.4.5 Tratamiento de errores

Para dejar bien acabado un programa de mantenimiento de archivos conviene que este gestione todos los errores que se puedan producir. Puede utilizar la función ferror(pArchivo) cada vez que haga un acceso a un archivo para determinar si se ha producido un error. Si no se ha producido ningún error esta función retorna 0, y si se ha producido un error retorna un valor mayor que cero. Junto con ferror() se suele utilizar la función perror("Mensaje del programa"); esta función muestra la cadena pasada, y después otro mensaje suministrado por el sistema.

```
fwrite( &Estudiante, sizeof (struct Alumno), 1, pArchivo);
if (ferror(pArchivo))
{
        perror("Error al intentar escribir en el archivo ");
        fclose(pArchivo);
        exit();          // Salir del programa
}
```

Si cuando quiera guardar el registro en el disco ya está lleno, este código muestra el mensaje:

```
Error al intentar escribir en el archivo : No space left on device
```

Seguidamente se cierra el archivo (fclose), y se sale del programa (exit);

10.5 Entrada/ Salida a nivel sistema.

También denominada Entrada/ Salida a bajo nivel porqué se parece mucho a los métodos utilizados por el sistema operativo, para leer y guardar información en archivos.

Con estos métodos no es posible escribir ni leer caracteres, cadenas o estructuras; los datos se tienen que tratar como series de bytes gestionados a través de un buffer. Antes de escribir en un disco se ha de llenar este buffer con los valores que se quieran guardar. Cuando se lee del disco, los datos pasan al buffer desde donde el programa los tendrá que coger.

Utilizando las funciones de entrada/salida a nivel sistema se consigue más velocidad en los accesos a disco, y el código compilado también es menor.

10.5.1 Abrir un archivo

De la misma manera que en E/S estándar, lo primero que se tiene que hacer es abrir el archivo para establecer la comunicación con el sistema operativo. La función para abrir el archivo es open().

```
open(Nombre_Archivo, indicadores);
```

Nom_Archivo, es el nombre de el archivo que se quiere abrir; se puede especificar la trayectoria entera.

Indicadores, sirven para indicar si el archivo se abrirá para lectura, escritura, modo binario, o bien modo texto. Estos indicadores (flags en inglés) son diferentes valores numéricos, TC incorpora una serie de constantes que se pueden utilizar para establecer los diferentes modos:

O_APPEND	sitúa el puntero al final del archivo.
O_CREAT	crea un nuevo archivo para escritura; si el archivo existe, no tiene efecto.
O_EXCL	retorna un valor de error si el archivo ya existe, sólo se utiliza con O_CREAT.
O_RDONLY	abre un archivo sólo para lectura.
O_RDWR	abre un archivo par lectura-escritura.
O_TRUNC	abre un archivo y trunca su longitud a 0.
O_WRONLY	abre un archivo sólo para escritura.
O_BINARY	abre un archivo en modo binario.
O_TEXT	abre un archivo en modo texto.

A la hora de abrir un archivo se pueden combinar estos valores; por ejemplo si se pretende abrir un archivo para lectura en modo binario se combinaran O_RDONLY | O_BINARY. Hay posibilidades que se excluyen mútuamente; por ejemplo, no se puede abrir un archivo para lectura y para lectura-escritura al mismo tiempo.

Todas estas constantes están definidas en el archivo de cabecera FCNTL.H que ha de estar incluido con la directiva #include.

10.5.1.1 Handles de archivos

De la misma manera que la forma fopen() retorna un puntero que sirve para referenciar o indicar sobre que archivo se trabaja, la forma open() retorna un valor entero llamado handle de archivo (se podría traducir por manejador de archivo). El handle es un número que después se utiliza por referenciar el archivo; si la función open() retorna un valor < 0 es porqué se ha producido un error. Para abrir un archivo y comprobar si se ha hecho correctamente, se podría utilizar un código similar al siguiente:

```
Num_Archivo = open(Nombre_Archivo, O_Indicadores);
if (Num_Archivo > 0)
        Acciones;
else
        Error;
```

10.5.2 Cerrar un archivo.

Para cerrar un archivo se ha de utilizar la siguiente función:

```
close(Num_Archivo);
```

10.5.3 Establecer el buffer

Tal y como se ha indicado anteriormente, todas las operaciones que se hacen con archivos a nivel sistema requieren la utilización de un buffer para gestionar los datos. Recuerde que un buffer no es más que una zona de memoria. La manera más práctica de definir el buffer, es dimensionar un array de datos tipo char (el tipo char ocupa un byte). El tamaño del buffer puede ser el que se quiera, pero hay tamaños con los que se trabaja de forma más óptima. Por ejemplo, en sistema operativo MS-DOS el tamaño óptimo de un buffer son múltiplos de 512. Para declarar un buffer se hace de la forma siguiente:

```
#define G_BUFFER
char buffer[G_BUFFER];
```

10.5.4 Lectura de un archivo

La función para leer un archivo es read(),

```
read(Num_Archivo, buffer, Bytes_a_Leer);
```

Esta función retorna el número de bytes leídos, es importante controlarlos ya que en la última lectura posiblemente se lean menos bytes que los indicados en Bytes_a_Leer, y esto significará que se ha llegado a final de archivo.

Ejemplo de lectura de un archivo utilizando funciones de bajo nivel:

```
#include <stdio.h>
#include <io.h>
#include <conio.h>
#include <fcntl.h>
#define G_BUFFER 512
#define NOM_ARCHIVO "C:\\Datos.Dat"

main()
{
        int Num_Archivo, Bytes_Leidos, i;
        char buffer[G_BUFFER];

        Num_Archivo = open(NOM_ARCHIVO, O_RDONLY | O_BINARY);

        if (Num_Archivo > 0)          // Si se puede  abrir el archivo
        {
                Bytes_Leidos = read(Num_Archivo, buffer, G_BUFFER);
                while (Bytes_Leidos > 0)       // Mientras se han leido bytes
                {
                        for (i=0; i < Bytes_Leidos ; i++)
                        putchar(buffer[i]);
                        Bytes_Leidos=read(Num_Archivo, buffer, G_BUFFER);
                }
                close(Num_Archivo);
        }
        else
                puts("Error, no se ha podido abrir el archivo");
}
```

10.5.5 Escribir en un archivo

La función utilizada para escribir en un archivo es write():

```
write(Num_Archivo, buffer, G_Buffer);
```

Recuerde que los datos a escribir han de estar colocados en el buffer.

Como ejemplo de utilización de funciones de acceso a archivos a bajo nivel, haremos un programa que sirva para copiar el contenido de un archivo a otro. La función del programa es similar a la orden COPY de MS-DOS:

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

```
#include <io.h>
#include <conio.h>
#include <fcntl.h>
#include "sys\stat.h"
#define G_BUFFER 2048
main()
{
        int Num_Archivo_E, Num_Archivo_L;
        int Bytes_Leidos;
        char buffer[G_BUFFER];
        char Archivo_Origen[12],Archivo_Destino[12];
        puts("Nombre Archivo origen : ");
        gets(Archivo_Origen);
        puts("Nombre Archivo copia : ");
        gets(Archivo_Destino);
        Num_Archivo_L=open(Archivo_Origen, O_RDONLY | O_BINARY);
        if (Num_Archivo_L > 0)
        {
                Num_Archivo_E = open(Archivo_Destino, O_CREAT | O_WRONLY |
O_BINARY, S_IWRITE);
                if (Num_Archivo_E > 0)
                {
                        Bytes_Leidos = read(Num_Archivo_L, buffer, G_BUFFER);
                        while (Bytes_Leidos > 0)
                        {
                                write(Num_Archivo_E, buffer, Bytes_Leidos);
                                Bytes_Leidos = read(Num_Archivo_L, buffer, G_BUFFER);
                        }
                        close(Num_Archivo_L);
                        close(Num_Archivo_E);
                }
                else
                        puts("Error, no se ha podido crear el archivo copia ");
        }
        else
                puts("Error, no se ha podido abrir el archivo origen");
}
```

Observe la función open() utilizada para abrir el archivo para escritura:

```
Num_Archivo_E = open(Archivo_Desti , O_CREAT | O_WRONLY | O_BINARY,
S_IWRITE);
```

S_IWRITE es un indicador de permiso. Cuando se utiliza O_CREAT siempre se ha de añadir otra variable a la función open() para indicar el estado de lectura-escritura en el archivo que se creará. Hay tres valores posibles:

S_IWRITE	escritura permitida.
S_IREAD	lectura permitida.
S_IREAD | S_IWRITE	escritura y lectura permitidas.

En MS-DOS todos los archivo son leíbles, por tanto el único modo necesario en este sistema operativo es **S_IWRITE.**

Los indicadores **S_IWRITE** y **S_IREAD** están definidos en el archivo de cabecera stat.h. Este archivo no se encuentra en el directorio include, por tanto se tendrá que definir toda la trayectoria en la línea #include

```
#include "sys\stat.h"
```

El archivo types.h es necesaria para stat.h.

Observe que se ha definido el tamaño del buffer a 2048; a mayor tamaño del buffer, menos accesos se harán a disco, y más rápida irá el programa. Pero Cuidado con no pasarse ya que limitaria otras áreas de memoria.

10.6 Ejercicios

Ejercicio 1: Escribe un programa que lea un fichero y le suprima todas las vocales. Es decir que siendo el fichero Hola.txt:

El alegre campesino
pasea por el campo
ajeno a que el toro
se acerca por detrás

El fichero destino.txt sea:

l lgr cmpsn
ps pr l cmp
jn q l tr
s crc pr dtrás

Solucion:

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

int main()
{
        FILE *origen, *destino;
        char letra;

        origen=fopen("C:\\Hola.txt","r");
        destino=fopen("C:\\destino.txt","w");
        if (origen==NULL || destino==NULL)
        {
                printf( "Problemas con los ficheros.\n" );
                exit( 1 );
        }
        letra=getc(origen);
        while (feof(origen)==0)
        {
                if (!strchr("AEIOUaeiou",letra)) putc( letra, destino );
                        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" );
        }
```

Ejercicio 2:

La solución propuesta no elimina las vocales acentuadas, modifica el programa para conseguirlo.

Solución:

Lo único que hay que hacer es añadir "áéíóúÁÉÍÓÚ" a la línea:

```
        if (!strchr("AEIOUaeiouáéíóúÁÉÍÓÚ",letra)) putc( letra, destino );
```

CAPITULO IX:

El preprocesador

11.1 Introduccion

El lenguaje C se desarrolló con el fin de cubrir las necesidades de los programadores atareados, y a este tipo de personas les gusta disponer de un preprocesador. Cuando escribo un programa en C, no es necesario que lo haga en detalle, sino que parte del trabajo pesado se lo puedo dejar a este "colaborador": él se encargará de leer mi programa antes de que caiga en las manos del compilador (de ahí el nombre de preprocesador), y, siguiendo las indicaciones que le haya dejado a lo largo del programa fuente, sustituirá abreviaturas simbólicas por las direcciones que representan, buscará otros ficheros donde puede tener escritos trozos de programa e incluso puede tomar decisiones sobre qué partes enviar al compilador y qué partes no. Esta breve descripción no hace justicia a la gran ayuda que representa este intermediario.

Una posible definición de preprocesador sería la siguiente:

- Es un programa procesador de textos, independiente del lenguaje C. El compilador invoca al preprocesador como primera acción sobre el código fuente. El preprocesador recibe ficheros de texto como entrada (programas fuente o simplemente texto) y produce un fichero de texto expandido como salida.

- Tiene comandos que permiten al programador incluir ficheros externos, sustituir texto, expandir macros y excluir condicionalmente líneas de texto.

11.2 Sintaxis Del Preprocesador

La sintaxis del preprocesador está bien estandarizada entre los numerosos compiladores del C. El compilador de C de Microsoft también incorpora algunos mejoras de ANSI que aumentan la legibilidad y utilidad de los comandos de su preprocesador lo que provoca que sean más parecidos a las sentencias de control de flujo de C.

Todos los comandos del preprocesador empiezan por el símbolo # , seguido directamente de un nombre en minúsculas.

Sólo puede aparecer un comando por línea (como con las sentencias del C, la barra invertida indica sentencias que continúan en lo línea siguiente).

No nos podemos olvidar ni de las directrices ni del símbolo #. Las primeras son utilizadas para hacer programas fáciles de cambiar y fáciles de compilar en diferentes entornos de ejecución. Las directrices le indican al preprocesador qué acciones específicas tiene que ejecutar. Por ejemplo, reemplazar elementos en el texto, insertar el contenido de otros ficheros en el fichero fuente, o suprimir la compilación de partes del fichero fuente.

El preprocesador de C reconoce las siguientes directrices:

#define	#ifdef	#endif	#line
#elif	#error	#ifndef	#pragma
#else	#if	#include	#undef

El símbolo # debe ser el primer carácter, distinto de espacio en blanco, en la línea que contiene o la directriz; entre el símbolo # y la primera letra de lo directriz pueden aparecer caracteres espacio en blanco. Una

directriz que ocupe más de una línea física puede ser continuado en la línea siguiente, colocando el carácter \ inmediatamente antes de cambiar a la línea siguiente.

Cualquier texto que siga a una directriz, excepto un argumento o un valor que forma parte de la directriz, tiene que ser incluido como un comentario (/* */).

Una directriz puede escribirse en cualquier parte del fichero fuente, pero sólamente se aplica desde su punto de definición hasta el final del programa fuente.

11.3 El comando #define

El comando #define hace que un identificador sea reemplazado en todas sus apariciones en el código fuente por un texto definido en el comando del preprocesador. Su sintaxis es:

```
#define IDENTTFICADOR texto_definición
```

El texto_definición puede ser una expresión, parte de una sentencia o uno o más sentencias completas.

Siempre es preferible usar constantes definidas en el código fuente de C, en lugar de constantes o variables normales cuyos valores no cambian nunca. La utilización de símbolos definidos con el comando #define puede aumentar la legibilidad, la independencia o la facilidad de mantenimiento. No es necesario incluir el código expandido cuando se usa un símbolo definido en lugar del código, y se puede cambiar el valor de todas las apariciones del símbolo definido en todo el programa con un solo cambio en el nivel del preprocesador. Además, nombres de identificadores cuidadosamente escogidos mejoran la claridad del código al reflejar la función de cada constante definida.

A continuación del comando #define se escribe el nombre del identificador que se está definiendo y el texto al que va a reemplazar. Es una sustitución directa de texto por texto, no se realiza ninguna evaluación de expresiones.

Observe que el preprocesador no reemplaza los símbolos definidos en el programa en que se encuentran entre comillas, en constantes de caracteres o son parte de un nombre más largo. Paro mayor claridad, un buen estilo de programación aconseja el uso de mayúsculas para símbolos definidos, y minúsculas, paro nombres de variables.

EJEMPLO:

```
#define MES 30
```

Está definición reemplaza todas las apariciones del símbolo MES en el programa de entrada por su valor, que en esto caso es 30. En caso de querer modificar el número de días sólo haría falta ir a la línea donde está definido y cambiar dicho número.

11.3.1 Definiciones A Evitar

Algunos tipos de definiciones parecen adecuados o primera vista, pero pueden ocasionar serios problemas. Veamos o continuación unos cuantos casos:

- Evite definiciones circulares en las cuales los símbolos que se definen aparecen en su propia definición (definición recursiva). En el ejemplo siguiente, el símbolo SINFIN se expande sin cesar y como poco provocará en el preprocesador en un tremendo embrollo.

  ```
  #define SINFIN (SINFIN +1)
  var = SINFIN;
  ```

  Se expande indefinidamente para convertirse en:

var = ... (((SINFÍN+1)+1)+1) ....

- Omita definiciones que no aporten claridad. Fuerzan al lector a tener que buscar en otro sitio para conocer el significado del símbolo. Por ejemplo, la siguiente definición no hace absolutamente nada:

```
#define DOCE 12 /*¡Vaya novedad!*/
```

  Siguiendo en la misma línea, evite definiciones que puedan prestarse a confusión. Por ejemplo, usted puede saber lo que significa dicha definición, pero otras personas pueden llegar a lo confusión:

```
#define DOCE 14 /* Interesante...*/
```

Es recomendable tener presente que el comando #define:

- No define variables.
- Sus parámetros no son variables.
- En el preprocesamiento no se realiza una revisión de tipos, ni de sintaxis.
- Sólo se realizan sustituciones de código.

11.4 ¿Macros O Funciones?

En muchos casos podemos encontrarnos en la incertidumbre de emplear una macro con argumentos o una función. En general, no hoy una línea divisoria nítida y sencilla, pero conviene hacer algunas consideraciones.

Es preciso estar mucho más atento cuando se usan macros que cuando se emplean funciones. Algunos compiladores limitan la definición de uno macro a una línea. Si nuestro compilador permite emplear más de una línea, actuemos como si nuestro compilador no permitiera definiciones en varios líneas.

La elección entre macros y funciones es otra forma de la lucha entre tiempo y espacio. Las macros producen programas de mayor extensión, ya que añadimos una sentencia al programa. Si empleo la macro 20 veces, cuando el preprocesador termine su trabajo, su programa contendrá 20 nuevas sentencias. Si, por el contrario, emplea uno función 20 veces, su programa solamente tendrá uno vez el cuerpo de la función; por tanto, ocupará un menor espacio.

Como contrapartida, el control del programa deberá saltar al punto de comienzo de la función y, una vez terminado, retornar al punto de donde salió; este proceso es más lento que seguir el curso normal sin saltos.
Una ventaja adicional de los macros es que, debido a que actúan sobre caracteres y no sobre las valores que representa, son independientes del tipo que tengan las variables.

11.4.1 Diferencias Entre Macros Y Funciones

Un macro es genérica y aceptará tipos de datos distintos para los argumentos, al contrario que uno función, que es menos flexible. Por ejemplo, un macro puede tomar argumentos char, short, long. float o double.

Siguiendo en lo mismo línea, evite definiciones que puedan prestarse o confusión. Por ejemplo, usted puede saber lo que significo dicho definición, pero otros personas pueden llegar a lo confusión:

```
#define DOCE 14       /* esto es nuevo */
```

Una macro se ejecuta más rápidamente que su función equivalente, porque no son necesarios los procesos de llamada y retorno.

Una macro es más difícil de depurar que una función. En realidad, se debería escribir en primer lugar como una función, depurarla y luego convertirla en macro.

Una macro puede contener muchas sentencias, pero si usamos una de 10 sentencias, añadimos 1000 sentencias a nuestro programa en un bucle de 100 iteraciones. Si empleamos una función de 10 líneas a la que llamamos 100 veces, el programa aumenta en 110 líneas.

Una macro, al contrario que una función, debe recibir exactamente el número de argumentos que espera. Hay que tener cuidado con los efectos secundarios en los argumentos de las macros y no llamar a una función para utilizarla como argumento y usar su valor de retorno: el efecto secundario o función podría ejecutarse dos veces y crear problemas.

11.4.2 Tipos De Macros

Las macros se pueden agrupar en tres categorías, según la naturaleza de su sustitución:

- Macros de expresión.
- Macros de sentencias.
- Macros de bloques.

El tipo de macro determina el contexto en el que puede aparecer. Por ejemplo: una macro de expresión se puede llamara varias veces en una sentencia, pero una macro de sentencia no se puede emplear como parte de una expresión, porque las sentencias no equivalen a un valor. Cuando se pueda elegir, utilizar macros de expresión ya que tiene un valor y se pueden usar como funciones que devuelven un valor.

11.4.3 Macros Predefinidas

ANSI C reconoce cinco macros predefinidas. El nombre de cada una de ellas va precedido y seguido por dos guiones de subrayado. Estas macros son:

__DATE__
Es sustituida por una cadena de caracteres de la forma mm dd aaa (mes, día, año). Ejemplo:

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

__TIME__
Esta macro es sustituida por una cadena de caracteres de la forma hh:mtn:ss (hora, minutos, segundos). Por ejemplo:

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

__FILE__
Esta macro es sustituida por el nombre del fichero fuente. Por ejemplo:

```
príntf( "%s"\n", __FILE__);
```

__LINE__
Esta macro es sustituida por el entero correspondiente al número de línea actual. Por ejemplo:

```
int linea = __LINE__;
```

__STDC__
Esta macro es sustituida por el valor 1 si el código esté conforme o las normas ANSI C. Por ejemplo:

```
int ANSI = __STDC__ ;
```

11.5 Operadores

11.5.1 El operador

Es utilizado solamente con macros que reciben argumentos. Cuando este operador precede al nombre de un parámetro formal en la definición de la macro, el parámetro actual correspondiente es incluido entre comillas dobles y tratado como un literal.

Por ejemplo:

```
#define IMPRIME(s) printf(#s "\n")
```

Una ocurrencia como

```
IMPRIME(Pulse Intro para continuar);
```

Será sustituida por:

```
printf("Pulse una tecla para continuar" "\n");
```

equivalente a:

```
printf("Pulse una tecla para continuar\n");
```

11.5.2 El operador #@

Es utilizado solamente con macros que reciben argumentos. Cuando este operador precede al nombre de un parámetro formal en la definición de la macro, el parámetro actual correspondiente es incluido entre comillas simples y tratado como un carácter.

Por ejemplo:

```
#de fine CARACTER (c) #@c
```

Una ocurrencia como:

```
car = CARACTER (b);
```

Sería sustituida por:

```
car = 'b';
```

11.5.3 El operador

Al igual que el anterior, también es utilizado con macros que reciben argumentos. Este operador permite la concatenación de dos cadenas.

Por ejemplo:

```
#define SALIDA(n) printf("elemento "#n" = %d\n", elemento##n);
```

Una ocurrencia como:

```
SALIDA (1);
```

Sería sustituida por:

```
printf("elemento " "1" = %d\n", elemento1);
```

equivalente a:

```
printf("elemento 1 = %d\n", elemento1);
```

11.6 El comando #include

Este comando pone los contenidos de los ficheros cabecera o disposición de los ficheros fuentes en C, simplemente reemplazando cada comando #include de un programa fuente por el contenido del fichero de cabecera que nombra.

Para un buen estilo los comandos #include se deberían colocar al principio del fichero fuente en C, después de los comentarios iniciales que describen el fichero fuente, pero antes de la primera función.

Hay dos sintaxis paro el comando #include. Su elección depende de dónde se vaya a buscar el fichero incluido (si la vía de acceso (path) está completa y sin ambigüedades, no se requiere búsqueda y las sintaxis son equivalentes):

- Busca en el directorio actual primero y luego en los sitios estándar si es necesario.

```
#include "vía de acceso"
```

- Busca sólo en los sitios estándar.

```
#include < vía de acceso>
```

Utilice la sintaxis < > para ficheros de cabecera que se vayan a usar en todo el sistema; pueden incluirse desde muchos directorios distintos para compilar una gran variedad de aplicaciones. Utilice la sintaxis con comillas (" ") para ficheros que se vayan a necesitar exclusivamente para un programa determinado o para uso personal.

Si estamos trabajando en UNIX, los paréntesis de ángulo le indican al preprocesador que busque el fichero en un (también puede haber varios) directorio estándar del sistema. Las comillas le dicen que lo busque en su directorio (o en algún otro si se le añade el nombre del fichero) en primer lugar; si no lo encuentra, en el directorio estándar.

- Busca en los directorios del sistema.

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

- Busca en el directorio en el que estemos trabajando.

```
#include "mi_archivo.h"
```

- Busca en el directorio indicado.

```
#include "/hoy/ayer/usuario/cabera.h"
```

- Busca en la unidad a:

```
#include <a:file.h>
```

Hay que tener cuidado con / y \ si estamos en Windows o en Linux.

Si estamos trabajando en un ordenador típico, las formas son iguales, y el preprocesador buscará en el lugar que le le indiquemos

```
#include "stdio.h"
#include <stdio.h>
#include "a:conio.h"
```

busca en el disco de trabajo a:

11.6.1 ¿Paro qué incluir ficheros?

Porque tienen la información que necesita. El fichero stdio.h, por ejemplo, contiene generalmente las definiciones de EOF, getchar() y putchar(), entre otras muchas… Las dos últimas, definidas como macros con argumentos.

El sufijo .h se suele emplear para ficheros de encabezamiento (header), es decir, con información que debe ir al principio del programa. Los ficheros cabecera consisten, generalmente, en sentencias para el preprocesador. Algunos, como stdio.h, vienen con el sistema, pero pueden crearse los que quiera para darle un toque personal a sus programas.

Para crear una ruta de acceso (path) para los directorios por defecto o los "sitios de búsqueda estándar" para los ficheros de cabecera, use el comando del MS-DOS SET para establecer la variable de entorno INCLUDE. (Puede usar este comando desde la línea de comandos o en su fichero autoexec.bat.) Por ejemplo, el siguiente

comando hace de INCLUDE en la unidad C, el directorio en el que el compilador buscará los ficheros de cabecera:

```
SET INCLUDE = C:\INCLUDE
```

11.7 El comando #undef

Como su nombre indica, la directriz #undef borra la definición de un identificador previamente creado con #define. La sintaxis es:

```
#undef identificador
```

La directriz #undef borra la definición actual de identificador Consecuentemente, cualquier ocurrencia de identificador es ignorada por el preprocesador.

```
#define SUMA(a, b) (a) + (b)
. . .
. . .
#undef SUMA /* la definición de SUMA es borrada */
```

Un ejemplo podría ser el siguiente: Suponga que esta trabajando en un conjunto de programas en colaboración con otros programadores. Quiere definir uno macro, pero no está seguro si su definición será compatible con otros realizados por sus compañeros. Paro evitar problemas deje sin efecto sus definiciones en cuanto termine su zona de utilidad, y, si estaban definidos anteriormente, volverán o recuperar su valor.

Los comandos restantes que mencionamos te permiten realizar una compilación bajo determinados condiciones. Aquí va un ejemplo:

```
#ifdef EQUIPO
        #include "portero.h" /*se realiza si EQUIPO está definido*/
        #define GOL 1
else
        #include <defensa.h> /*se realiza si EQUPO no está definido */
```

```
#define GOL 2
```

11.8 Los comandos #ifdef e #ifndef

El comando #ifdef indica que si el identificador que le sigue (en el ejemplo anterior EQUIPO) ha sido definido por el preprocesador, entonces se ejecutan todos los comandos hasta el siguiente #else o #endíf. Si encuentro primero un #else, entonces se ejecutan los que se encuentren desde #else hasta #endif si el identificador no está definido.

La estructura es similar a la if ... else de C. La principal diferencia reside en que el preprocesador no reconoce el método { } de limitar un bloque; por tanto, emplea #else (si lo hay) y #endif (que debe estar) para delimitar los bloques de comandos.

Estas estructuras condicionales pueden anidarse en más de un nivel.

Los comandos #ifndef e #if pueden emplearse junto con #else y #endif de la misma forma. #ifndef pregunta si el identificador no está definido; es el complementario de #ifndef. El comando #if se parece más al if de C; es seguido por una expresión constante que se considera cierta si no es 0:

```
#if MICRO == PENTIUM'
        #include "cuidado.h"
#endif
```

El uso de esta "compilación condicional" es hacer que un programa sea más portátiles.

La sintaxis correspondiente a estas directrices es:

```
#ífdef ídentificador
#ifndef ídentífícador
```

La directriz #ifdef comprueba si el identificador está definido e #ifndef comprueba si el identificador no está definido. La directriz #ifdef ídentificador es equivalente a #if defined identificador, y la línea #ifndef es equivalente a #if ! defined identificador (if not defined)

Estas directrices simplemente garantizan la compatibilidad con versiones anteriores de C, ya que su función es ejecutada perfectamente por el operador defined(identificador).

11.9 El comando #line

La sintaxis de la directriz #line es la siguiente:

```
#line constante_entera ["identificador"]
```

Una línea de la forma indicada pone las macros predefinidas __LINE__ y __FILE__ a los valores indicados por constante _entera e identificador, respectivamente, lo cual hace que el compilador cambie su contador interno y su nombre de fichero de trabajo, por los valores especificados en estos constantes. Si se omite el identíficador (normalmente un nombre de fichero) se utiliza el que tenga lo constante __FILE __ por defecto (el nombre del fichero fuente).

La información proporcionado por la directriz #line se utiliza simplemente con el objeto de dar mensajes de error más informativos. El compilador utiliza está información paro referirse a los errores que encuentra durante la compilación.

11.10 El comando #error

La directriz #error es utilizado paro abortar una compilación, al mismo tiempo que se visualiza el mensaje de error especificado a continuación de lo misma. Su sintaxis es:

```
#error mensaje
```

Esta directriz tiene utilidad cuando en un programa incluimos un proceso de compilación condicional. Si se detecta una condición anormal, podemos abortar lo compilación utilizando esta directriz, al mismo tiempo que se visualiza el mensaje de error especificado. Por ejemplo:

```
#ifn ! defined (MONEDA)
#error MONEDA no definida.
#endif
```

Cuando se compile el programa fuente que contiene las directrices anteriores, si la macro o constante MONEDA no está definida, se emitirá el mensaje de error especificado.

11.11 Directriz #pragma:

Si un código fuente contiene una determinada instrucción que el preprocesador no soporta, mediante #pragma será ignorada.

```
#pragma nombre_directiva
```

Si nombre_directiva no es reconocido, mediante #pragma será ignorada.

Turbo C soporta dos pragmas: inline y warn.

La primera se emplea para indicar el lugar en el que se inserta código en ensamblador. La segunda activa, desactiva o restablece un mensaje de advertencia.

11.12 La compilación condicional

Los comandos de compilación condicional del C se pueden usar para controlar los comandos del preprocesador o del código fuente del C. Los comandos condicionales hacen que determinadas líneas de código fuente sean compiladas o ignoradas de forma selectiva, dependiendo del valor o la existencia de un símbolo o macro definidos por medio del comando #define.

El comando #if controla el texto del C y de los comandos del preprocesador que se encuentran entre él mismo y el comando #endif asociado, de la forma siguiente:

```
#if expresión_ constante_restringida
/* ...texto compilado condicionalmente... */
#endif
```

Si la expresión_ constante_restringida después del #if es distinta de cero, las líneas que sigan al comando #if serán compiladas; si no es así, serán ignoradas y el preprocesador seguirá con la línea siguiente a #endif.

Una expresión_ constante_restringida es un tipo especial de expresión constante. Puede contener cualquier número y tipo de constantes (excepto constantes enum) combinadas por operadores (excepto sizeof adaptación de tipo (casting), coma y asignación) para componer una expresión que será evaluada durante la compilación.

La expresión_constante_restringida también puede contener la expresión defined (IDENTIFICADOR), que se evalúa como verdadero si IDENTIFICADOR ha sido definido como constante o macro.

Por ejemplo, estos tres líneas harán que lo sentencia:

```
dibujar (10, 20, 30, 40);
```

será compilada únicamente si el símbolo GRAFICOS ha sido definido previamente:

```
#íf defíned (GRAFICOS)
dibujar(10, 20, 30, 40);
#endif
```

Ya hemos mencionado que se pueden inhabilitar, en lugar de borrar, líneas de código en C o comandos del preprocesador que han sido introducidos en un programa, pero que todavía no son totalmente operativos y tendrán que ser ignorados mientras se depura otra sección del código. Desafortunadamente, los delimitadores de comentarios /* y */ no pueden comentar código que o su vez contiene comentarios, porque el primer */ que encuentre el preprocesador será considerado como el final del comentario y todo lo que le sigue se tratará como código potencialmente ejecutable.

Pero #if ofrece una solución fácil a este problema: utiliza simplemente #if 0 para asegurar que los líneas englobadas no serán compiladas.

```
#if 0
/*... líneas que el compilador siempre ignorará...*/
#endif
```

Entre los comandos del preprocesador que necesitarían ser ignorados en algunas situaciones están las palabras clave near, far y huge relacionadas con la memoria. Estas palabras clave son útiles porque permiten aprovechar todo la memoria instalada en un PC, en lugar de los 64 Kb de programa y 64 Kb de datos de muchos compiladores más antiguos. Sin embargo, no son estándar, por lo que se deben eliminar para usar algunos compiladores que funcionan en sistemas de 32 bits, especialmente sistemas con capacidad para memoria virtual. Puede usar la siguiente secuencia de comandos para inhabilitar globalmente esas palabras clave, reemplazándolas por cadenas vacías (siempre que el nombre GRANMEMORIA haya sido definido previamente):

```
#if defined (CRANMEMORIA)
#define near
#define far
#define huge
#endif
```

Las directrices #if, #elif, #else y #endif permiten compilar o no partes seleccionados del fichero fuente. Su sintaxis es lo siguiente:

```
#if expresión
[grupo de líneas]
[#elif expresión 1
grupo de líneas]
[#elif expresión 2
grupo de líneas]
. . .
. . .
[#elif expresión N
grupo de líneas]
[#else
grupo de líneas]
#endif
```

donde grupo de líneas representa cualquier número de líneas de texto de cualquier tipo.

El preprocesador selecciona un único grupo de líneas para pasarlo al compilador. El grupo de líneas seleccionado será aquel que se corresponda con un valor verdadero de la expresión que sigue a #if o #elif. Si todas las expresiones son falsas, entonces se ejecutará el grupo de líneas a continuación de #else.

11.13 El Operador Defined

El operador defined puede ser utilizado en una expresión de constantes enteras, de acuerdo con la siguiente sintaxis:

```
defíned ( identificador)
```

La expresión constante a que da lugar este operador es considerada verdadera (distinta de cero) si el ídentificador está actualmente definido y es considerada falso, en caso contrario. Por ejemplo:

```
#if defined (SUMA)
suma();
#elif defíned (RESTA)
resta();
#else
fnerrorQ;
#endif
```

En este ejemplo, se compila la llamada o lo función suma( ) si el identificador SUMA está definido; se compila la llamada o la función resta() si el identificador RESTA está definido; y se compila la llamada o la función fnerror() si no está definido ninguno de los identificadores anteriores.

11.14 Ficheros De Cabecera

Las definiciones incluidas en unos ficheros fuente especiales llamados ficheros de cabecera se pueden usar en funciones de C contenidas en diversos ficheros fuente. Esto significa que no tendrá que duplicar definiciones comunes para cada fichero de cada aplicación. Algunos ficheros de cabecera, como stdio.h o stdtlib.h, vienen con su compilador, junto con la librería estándar (en un buen estilo, los nombres de los ficheros de cabecera deberían terminar con la expresión .h. Otras se pueden asociar con librerías corporativas o de proyecto. También puede desarrollar las suyas propias de usuario.

11.14.1 Cabeceras Vs funciones de librería

Un fichero de cabecera es un fichero de texto que contiene código fuente en C y comandos del preprocesador que serán compilados cuando se incluya el fichero de cabecera en un fichero fuente en C y comandos del preprocesador que serán compilados cuando se incluya el fichero de cabecera en un fichero fuente en C mediante el comando #include.

Un fichero de librería es un fichero objeto que contiene uno colección de módulos yo compilados con funciones de uso frecuente. En las librerías es donde el linker busca las funciones que usted llama, pero que no ha escrito. Recordemos que las librerías las usa el linker y las ficheros de cabecera el preprocesador.
Paro contribuir a la confusión, lo mayoría de las librerías también tienen ficheros de cabecera asociados que incluyen comandos #define, declaraciones de tipos de parámetros o valores devueltos por las funciones de librería y más información relacionada requerido por las funciones de la librería.

11.15 Ejemplos De Programas

Ejemplo 1:

```
#include <stdio.h>

/* Ejemplos sencillos de preprocesador */
#define DOS 2 /* Se pueden usar comentarios */
#define MSJ "Hola programador..."
#define CUATRO DOS*DOS /* DOS ya está definido*/
#define PX printf(" X es %d.\n", x)
#define FMT " X es %d.\n"
void main()
{
        int x = DOS;
        PX;
        x = CUATRO;
        printf( FMT, x);
        printf("DOS: MSJ\n");
}
```

Ejemplo 2:

```
/* Macros con argumentos */
#include <stdio.h>
#define CUADRADO(x) x*x
#define PR(x) printf("x es %d.\n", x)

void main ()
{
        int x= 4, z;
        z = CUADRADO (x);
        PR(z);
        z = CUADRADO(2);
        PR(z);
        PR(CUADRADO(x));
        PR(CUADRADO(x+2));
        PR(100/CUADRADO(2));
        PR(CUADRADO(++x));
}
```

Ejemplo 3:

```
/* Contador de espacios en blanco*/

#include <stdio.h>

bool espacio (char c)
{
        if(c == ' ' || c =='\n' || c == '\t')
        return (true);
        else
        return (false);
}

void main()
{
```

```
        char caracter;
        int contador = 0;
        while ((caracter = getchar()) != 'F')
        if (espacio(caracter))
                contador++;
        printf(" HE CONTADO %d ESPACIOS EN BLANCO.\n", contador);
}
```

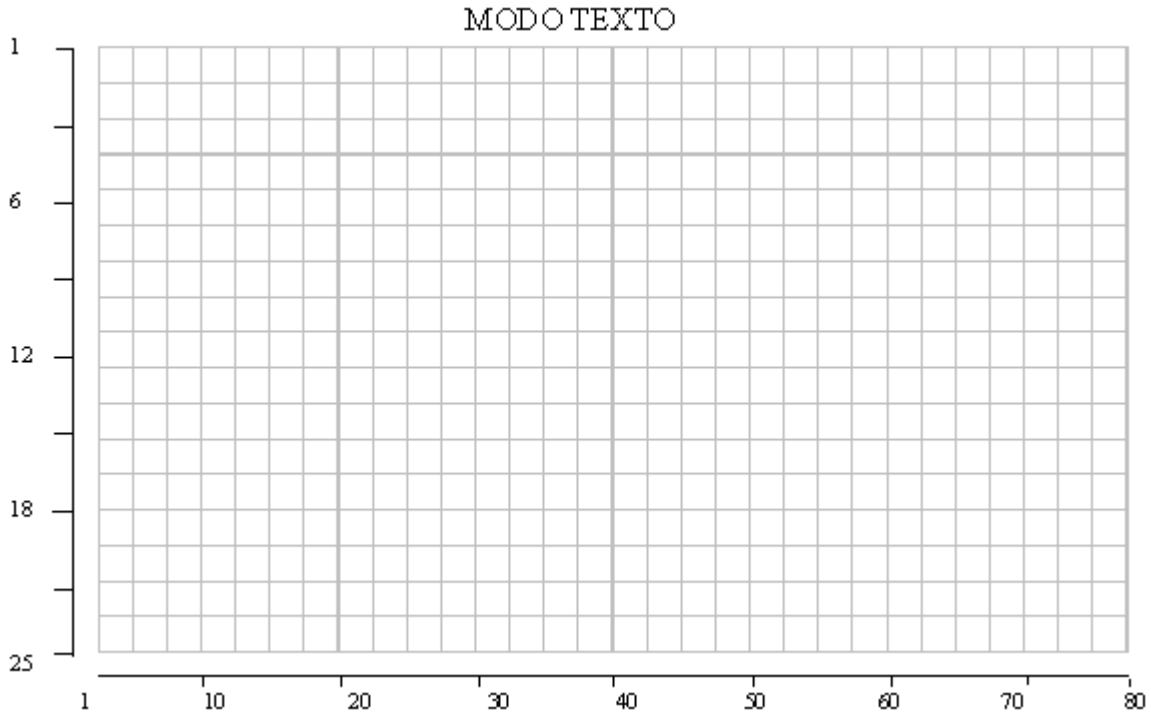
Ejemplo 4:

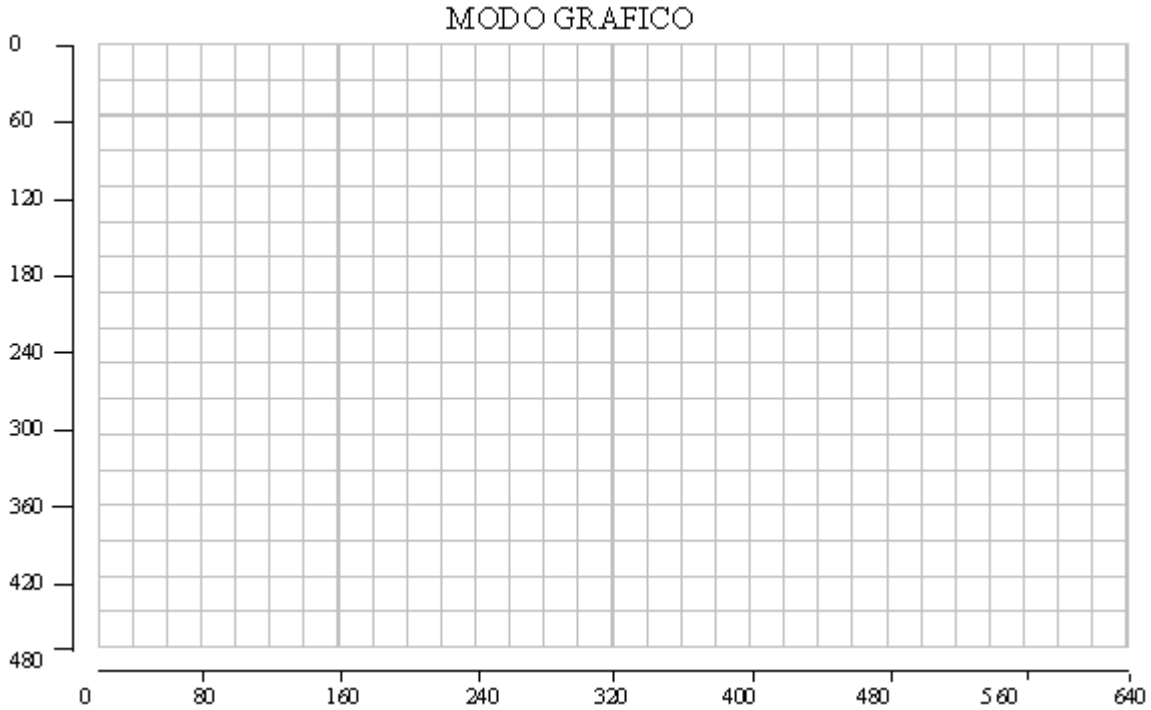
```
/* Compara la comprobación de definiciones de los preprocesadores nuevo y antiguo*/
#include <stdio.h>
void main()
{
        /* Uso de los comandoss del preprocesador antiguo para ver si NOTAL fue
admitido. */

        #ifdef NOTAL
        printf("#ifdef NOTAL es verdadero\n");
        #else
        printf("#ifdef NOTAL es falso\n");
        #endif
        #ifndef NOTAL
        printf("#ifndef NOTAL es verdadero\n");
        #endif
        /* Uso de los comandos del preprocesador nuevo paro ver si NOTAL fue
admitido*/
        #if defined NOTAL
        printf("#if defined (NOTAL) es verdadero\n");
        #else
        printf("#if defined (NOTAL) es falso\n");
        #endif
        #if !defined (NOTAL)
        printf("#if !defined (NOTAL) es verdadero\n");
        #endif
        /* Se pueden usar operadores lógicos para combinar tests.*/
        #if (!defined (NOTAL) && defined (SITAL) | | !defined( NODEF))
        printf ("LA COMBINACIÓN LÓGICA ES VERDADERA\n");
        #endif
}
```

CAPITULO X:

Resumen del lenguaje

INSTRUCCIONES Y COMANDOS DE C/C++





NORMAL	**FLUORESCENTE**	**NORMAL**	FLUORESCENTE
0 - Negro 1 - Azul 2 - Verde	9 - Azul 10 - Verde 11 - Azul Claro	17 - Azul 18 - Verde 19 - Azul Claro	25 - Azul 26 - Verde 27 - Azul Claro

3 - Azul Claro 4 - Rojo 5 - Fushia 6 - Naranja 7 - Blanco 8 - Gris	12 - Rojo 13 - Fushia 14 - Amarillo 15 - Blanco 16 - Negro	20 - Rojo 21 - Fushia 22 - Amarillo 23 - Blanco 24 - Gris	28 - Rojo 29 - Fushia 30 - Naranja 31 - Blanco 32 - Gris

Operandos Aritmeticos	**Operandos Relacionales**	Operandos Logicos
- Resta + Suma * Multiplicacion / Division real % Residuo = Asignacion	< Menor que > Mayor que <= Menor igual que >= Mayor igual que != Desigualdad = = Igualdad	&& Y || O Xor Bicondicional ! Negacion ++ Incremento -- Decremento

Declaracion de Variables

Tipo	Declaración	Limite Inferior	Limite Superior
Entero	Int A;	-32768	32767
Entero Corto	Short Int A;	-128	127
Entero Largo	Long Int A;	2E -6	2E 6
Entero sin Signo	Unsigned Int A;	0	65535
Entero con Signo	Signed Int A;	-65000	65000
Real	Float A;	-3.4E37	3.4E 38
Real Doble	Double A;	-1.7E -304	1.7E 308
Real Largo	Long DoubleA;	3.4E -4932	1.1E 4932
Caracter	Char A;	-128	127
Caracter sin signo	Unsigned Char A;		
Caracter con signo	Signed Char A;		
Palabra	Char[ ] A;		
Valor Nulo	Void	0	0
Arreglo	Int A[N]		
Texto	Text A;		
ante	A;		
Apuntador	*A		

for(Contador = 1;Contador <= N;Contador++)
{
Sentencia;
}
	De Contador=1 hasta Contador<=N Incremento

If(Condicion = = 1)
{
Sentencia;
}
Else
{
Sentencia;
}
	Si Condicion = 1 Entonces
	
	Si no

While(Condicion= =1)
{
Sentencia;
}
	Mientras Condicion = 1 haz

Do{ Sentencia; }(Condicion = = 1);	Haz Hasta Condicion = 1
Switch(Opcion) { 1: Sentencia1; break; 2: Sentencia2; break;	En caso de Opcion Si Opcion = 1 Sentencia1 rompe Si Opcion = 2 Sentencia2 rompe

Default: Sentencia3; break; }	Si no Sentencia3 rompe

Procedimiento	Función	Unidad

Void Nombre(VariableInt)
{
Sentencia;
}
void main(void)
{
Nombre(Variable)
}
	Float Nombre (Variable Int)
	{
	Sentencia;
	VarReal = Variable;
	Return (VarReal);
	}
	void main(void)
	{
	A = Nombre (X);
	}
		PROG.H
		#Ifndef prog__H
		#Define prog__H
		void pausa(void);
		PROG.C
		void pausa(void)
		{
		char A;
		A=Lee;
		}

OPERACIONES	
malloc(sizeof(Apuntador), Apuntador);	
Inicio->Info:='Hola';	
Inicio->Siguiente:=NIl;	
Aux:=NIL;	
malloc(sizeof(Apuntador), Apuntador);	
Inicio->Siguiente:=Aux;	
Dispose(Inicio);	

Usos de printf y scanf	

printf("Mensaje");
printf("Mensaje %d",Variable);
printf("Mensaje %d",Variable:2:3);
cprintf("Mensaje");
	Escribe Mensaje en la pantalla
	Escribe Mensaje y el valor de la Variable en pantalla
	Escribe Mensaje y el valor de la Variable con 2 enteros y 3 decimales
	Escribe Mensaje en color especificado

scanf("%d",&Variable); scanf("%d %f",&Variable1,&Variable2);	Asigna valor entero a Variable Asigna valor entero a Variable1 y valor real a Variable2

Formato para Variable		Formato de Barra Invertida	
Formato	Descripción	Formato	Descripción
%c	Un caracter	\b	Retroceso
%d	Real o Entero	\f	Salto de linea
%i	Real	\n	Nueva linea
%e	Notacion con "e"	\r	Retorno de carro
%E	Notacion con "E"	\t	Tabulador horizontal
%f	Real	\"	Comillas
%g	Real	\'	Apostrofo
%G	Real con notacion "E"	\n	Caracter nulo
%o	Octal	\\	Barra invertida
%s	Cadena	\v	Tabulador Vetical
%u	Real sin signo	\a	Campanilla
%x	Hexadecimal en minusculas	\N	ante octal
%X	Hexadecimal en mayusculas	\xN	ante hexadecimal

%%	Imprime %		
%p	Apuntador		
%n	Argumento de apuntador		

Funciones de Caracteres	
Función	Descripción
gets	Lee un string
getchar	Lee un caracter y espera enter
getche	Lee un caracter del teclado y lo refleja en pantalla
getch	Lee un caracter del teclado sin reflejarlo en pantalla
getc	Lee un caracter del un flujo
getpass	Lee un password
gettexto	Copia texto de la pantalla en modo texto a memoria
cgets	Lee string del teclado sin reflejar en pantalla
cgetc	Lee un string del teclado y lo refleja en pantalla
ungetch	Empuja un caracter al teclado
ungetc	Empuja un caracter a la salida de un flujo
puts	Escribe un string y retorna
putw	Escribe un entero en un stream
putc	Escribe un caracter para un stream
putch	Escribe caracter en la ventana de texto
putchar	Escribe un caracter
putenv	Adiciona un string en el ambiente actual
puttext	Copia texto de la memoria en la pantalla
fputc	Lee un caracter desde un flujo
fputs	Pone un string a un flujo
fputchar	Pone un caracter a un flujo
cputs	Escribe a string a una ventana de texto
kbhit	Verifica actividad teclas de función
gettextoenfo	Lee informacion de modo texto de video
movetexto	Copia texto de un rectangulo a otro
enp	Lee un byte del un puerto de el hardware
enpw	Lee un word del un puerto de el hardware
outp	Pone un byte en un puerto del hardware
outpw	Pone un word en un puerto del hardware
gotoxy	Posiciona el cursor en ventana de texto
swab	Copia n bytes de un string origen a destino
memset	Pone n bytes de orige a destino
memccpy	Copia un bloque de n bytes de origen a destino
memcmp	Compara los primeros n bytes de strings s1 y s2
memcpy	Copia un bloque de n bytes de origen a destino
memicmp	Compara los primeros n bytes de strings s1 y s2
memmove	Copia un bloque de n bytes de origen a destino
memchr	Busca n bytes en caracter c

Fucniones para Cadenas	
Función	Descripción
strcpy	Copia un string origen a un destino
strncpy	Copia hasta n caracteres de origen a destino
stpcpy	Copia uno string en otro
strdup	Copia un string dentro una locacion nuevamente creada
strstr	Busca la primera ocurrencia de un subcadena en otro string
strrchr	Busca la ultima ocurrencia de un caracter en un string
strchr	Busca un string por la primera ocurrencia de un caracter dado
strspn	Busca un string por un segmento que no contiene
strcspn	Busca un string por un segmento que no contiene
strpbrk	Busca un string1 la primera ocurrencia de cualquier caracter que esta string2
strtok	Busca s1 por el primera señal no contenida en s2
strcmp	Compara dos strings
stricmp	Compara dos strings sin caso sensitivo
strcmpi	Compara dos strings sin caso sensitivo

strcoll	Compara dos strings
strncmp	Compara porciones de dos strings
strnicmp	Compara porciones de dos strings
strncmpi	Comparas porciones de dos strings
strcat	Añade un string a otro
strlen	Calcula la longitud de un string
strncat	Añade un string a otro
strrev	Revierte todo caracteres en string(excepto el nulo)
strset	Pone todos caracteres en s a ch
strnset	Pone los primeros n caracteres de origen a destino

Funciones de Conversión	
Función	Descripción
fcvt	Convierte un real a string
ecvt	Convierte un real a string
gcvt	Convierte un real a string
itoa	Convierte un entero a string
ltoa	Converts a long to a string
ultoa	Convierte un unsigned long a string
ctime	Convierte fecha y hora a un string
atoi	Convierte un string a entero.
atol	Convierte un string a un long
_atold	Convierte un string to un long double
atof	Convierte un string a real
strtol	Convierte un string a long
strtoul	Convierte un string a unsigned long
strtod	Convierte un string a double real
asctime	Convierte fecha y hora a ASCII
strlwr	Convierte el contenido de un apuntador a caracteres a minusculas
strupr	Convierte el contenido de un apuntador a caracteres a mayusculas
strxfrm	Transforma una porcion de un string
toupper	Translada caracteres a mayusculas
tolower	Translada caracteres a minusculas
toascii	Translada caracteres a ASCII formato

Funciones de Comparacion	
Función	Descripción
isupper	Es una letra mayuscula (A-Z)
isxdigit	Es un hexadecimal digito (0-9,A-F,a-f)
isspace	Es un espacio,tab,acarreo,retorno,nueva linea
islower	Es un letra minuscula(a-z)
isprent	Es un prenteng caracter(0x20-0x7E)
ispunct	Es un signo puntuacion(cntrlo space)
isgraph	Es un caracter imprimible
iscntrl	Es un caracter delete o caracter de control(0x7F,0x00-0x1F)
isdigit	Es un digito(0-9)
isalpha	Es una letra(A-Z o a-z)
isascii	Es el byte mas bajo en el rango 0 a 127(0x00 - 0x7F)
isalnum	Es alfanumerico

Funciones de Creación de Archivo	
Función	Descripción
int creattemp(char *path, int attrib);	Crea un archivo unico en directorio dado por el path
int creatnew(char *path, int modo);	Crea y Abre un archivo nuevo para e/s en modo binario
int _dos_creatnew(char *path, int attrib, int *handlep);	Crea y Abre un nuevo archivo para e/s en modo binario
int creat(char *path, int amode);	Crea un nuevo archivo o sobreescribe en uno existente
int _creat(char *path, int attrib);	Crea un nuevo archivo o sobreescribe en uno existente
int _dos_creat(char *path, int attrib, int *handlep);	Crea un nuevo archivo o sobreescribe en uno existente
int dup(int handle);	Duplica un archivo handle
int dup2(int oldhandle, int newhandle);	Duplica un archivo handle en uno existente archivo handle

Funciones de Apertura y Cierre de Archivos	
Función	Descripción
FILE *fopen(char *nombre, char *mode);	Abre un flujo
FILE *freopen(char *nombre,char *modo, FILE *flujo);	Conecta nuevo archivo con un flujo abierto
FILE *_fsopen(char *nombre,char *modo, int shflg);	Abre un flujo con archivo sharing
int _dos_open(char *nombre, unsigned flags,int *handlep);	Abre un archivo para escritura o lectura
int _open(char *nombre, int flags);	Abre un archivo para escritura or lectura
int close(int handle);	Cierra archivo asociado con un handle
int _close(int handle);	Cierra archivo asociado con un handle
int _dos_close(int handle);	Cierra archivo asociado con un handle
int fclose(FILE *flujo);	Cierra flujo
int fcloseall(void);	Cierra todos flujos abiertos

Funciones de Posicionamiento en Archivos	
Función	Descripción
long ftell(FILE *flujo);	Devuelve el apuntador actual de el archivo
int fstat(int handle, struct stat *statbuf);	Toma informacion acerca un archivo abierto
int stat(char *path, struct stat *statbuf);	Toma informacion acerca un archivo
long tell(int handle);	Toma posicion actual de apuntador a archivo
void rewind(FILE *flujo);	Reposiciona apuntador de flujo para archivo
long lseek(int handle, long offset, int fromwhere);	Mueve apuntador lee/escribe archivo
int fseek(FILE *flujo, long offset, int whence);	Reposiciona el apuntador de flujo a un archivo
int fgetpos(FILE *flujo, fpos_t *pos);	Toma posicion actual de apuntador de archivo
int fsetpos(FILE *flujo, fpos_t *pos);	Posiciona el apuntador de flujo a un archivo

Funciones de Archivos Unicos	
Función	Descripción
FILE *tmpfile(void);	Abre a "scratch" archivo en modo binario
char *mktemp(char *template);	Hace un archivo de nombre unico
char *tmpnam(char *sptr);	Crea un archivo de nombre unico
char *tempnam(char *dir, char *prefix);	Crea archivo de nombre unico en el especificado directorio.
int rmtmp(void);	Remueve temporalmente archivos.

Fuciones Varias de Archivos	
Función	Descripción
fpos_t	Un archivo posicion type
FILE	Estructura de control de archivo para flujos
EOF,#define	Ante indicando que fin de archivo alcanzado
int fd();	Devuelve el archivo descriptor o EOF
int eof();	Devuelve nonzero si es fin de archivo
int eof(int handle);	Verifica para fin de archivo
int feof(FILE *flujo);	Ensaya si fin de archivo ha sido alcanzado en un flujo
int flushall(void);	Refresca todos los flujos abiertos
int fflush(FILE *flujo);	Refresca un flujo
int fileno(FILE *flujo);	Devuelve el archivo handle para el flujo
long filelength(int handle);	Toma tamaño de archivo en bytes
int ferror(FILE *flujo);	Ensaya si un error ha ocurrido en un flujo
char *strerror(int errnum);	Devuelve un apuntador a un mensaje de error
void clearerr(FILE *flujo);	Limpia indicador de error
int _chmod(char *path, int func [int attrib]);	Pone atributos de archivo
int _dos_getfileattr(char *path, unsigned *attribp);	Toma el atributos
int _dos_setfileattr(char *path, unsigned attrib);	Pone el atributos
int renom(char *oldnom, char *newnom);	Renombra un archivo
int remove(char *nombre);	Remueve un archivo
struct dirent readdir(DIR *dirp);	Lee la actual entrada desde un directorio flujo
fcb	Archivo bloques de control
int chmod(char *path,int amode);	Pone permiso de acceso a archivo
int chsize(int handle,long size);	Cambia archivo tamaño
int access(char *nombre,int amode);	Determina accessabilidad de un archivo

char *_searchenv(char *archivo,char *varnom,char *buf); unsigned umask (unsigned modoMask);	Busca un ambiente path para un archivo Pone archivo lee/escribe permiso mask

Funciones de manejo de bytes en Archivos	
Función	Descripción

int write(int handle,void *buf,int len);
int _write(int handle,void *buf,int len);
int _dos_write(int handle,void *buf,int len,int *nwritten);
int read(int handle,void *buf,int len);
int _read(int handle,void *buf,int len);
int _dos_read(int handle,void far *buf,int len,int *nread);
void setbuf(FILE *flujo, char *buf);
int setvbuf(FILE *flujo, char *buf, int type, size_t size);
int setmode(int handle, int amode);
	Escribe un buffer de datos a un archivo o dispositivo
	Usa funcion 0x40 para escribir bytes de un buffer a un archivo
	Usa funcion 0x40 para escribir bytes de un buffer a un archivo
	Intenta para leer bytes desde un archivo dentro un buffer
	Usa funcion 0x3F (lectura) para leer bytes de un archivo a un buffer
	Usa funcion 0x3F (lectura) para leer bytes de un archivo a un buffer
	Asigna buffering hasta un flujo
	Asigna buffering hasta un flujo
	Pone modo de apertura de un archivo

Funciones de Entrada y Salida de Archivos	
Función	Descripción

int ungetc(int c, FILE *flujo);
int getw(FILE *flujo);
int fgetchar(void);
int fgetc(FILE *flujo);
char *fgets(char *s, int n, FILE *flujo);
int putw(int w, FILE *flujo);
int fputc(int c, FILE *flujo);
int fputs(char *s, FILE *flujo);
int fputchar(int c);
int fprintf (FILE *flujo,char *format [arg]);
int fscanf (FILE *flujo,char *format [address]);
	Empuja un caracter a la entrada de un flujo
	Toma un entero de un flujo
	Toma un caracter de un flujo
	Toma un caracter de un flujo
	Toma un string de un flujo
	Salida de un entero en un flujo
	Salida de un caracter a un flujo
	Salida de a string a un flujo
	Salida de un caracter a un flujo
	Manda formato de salida a un flujo
	Recibe formato de entrada de un flujo

O_RDONLY	Abre para escritura solamente
O_WRONLY	Abre para lectura solamente
O_RDWR	Abre para escritura y lectura
O_APPEND	Si pone, el apuntador de archivo en fin de archivo anterior a cualquiera escribe.
O_CREAT	Crea y abre archivo Si el archivo existe no tiene efecto si no, el archivo es creado.
O_EXCL	Apertura exclusiva: Usado con O_CREAT.Si el archivo ya existe, un error es devuelto.
O_TRUNC	Abre con truncacion Si el archivo ya existe,es truncado en 0 los atributos no cambian
O_BINARY	No translacion Explicitamente Abre el archivo en binario modo
O_TEXT	CR-LF translacion Explicitamente Abre el archivo en modo texto
O_NOINHERIT	Proceso hijo heredando archivo
O_DENYALL	Error si abierto para lectura/escritura
O_DENYWRITE	Error si abierto para escritura
O_DENYREAD	Error si abierto para lectura
O_DENYNONE	Permite acceso concurrente
O_CHANGED	Especial DOS lee-solamente bit
O_dispòsitivo	Especial DOS lee-solamente bit

stdin	Standard dispositivo entrada
stdout	Standard dispositivo salida
stderr	Standard error salida dispòsitivo
stdaux	Standard dispòsitivo auxiliar
stdprn	Standard impresora

SEEK_SET	0 Busqueda desde principio de archivo
SEEK_CUR	1 Busqueda desde actual posicion
SEEK_END	2 Busqueda desde fin de archivo

_F_RDWR	Lee y escribe
_F_READ	Lee-solamente archivo
_F_WRIT	Escribe-solamente archivo
_F_LBUF	Linea-buffered archivo
_F_ERR	Indicator de error
_F_EOF	Indicator EOF
_F_BIN	Indicador binario archivo

_F_IN	Datos por llegar
_F_OUT	Datos por salir
_F_TERM	Archivo es un terminal

Funciones de Division	
Función	Descripción
double frexp(double x, int *exponent);	Divide a double numero en mantisa y exponente
double fmod(double x, double y);	Calcula x modulo y, el residuo de x/y
long double frexp(long double (x), int *(exponent));	Divide un long double numero en mantisa y exponente
long double fmod(long double (x), long double (y));	Calcula x modulo y,el residuo de x/y
double modf(double x, double *ipart);	Divide double en entero y fraccion
long double modfl(long double (x), long double *(ipart));	Divide long double en entero y fraccion
div_t div(int numer, int denom);	Divide dos enteros
ldiv_t ldiv(int numer,int denom);	Divide dos longs

Funciones de Potenciacion	
Función	Descripción
double pow(double x, double y);	Funcion Potenciacion , x a la y (x**y)
double pow10(int p);	Funcion Potenciacion , 10 a el p(10**p)
long double pow(long double (x), long double (y));	Funcion Potenciacion , x a la y (x**y)
long double pow10l(int (p));	Funcion Potenciacion , 10 a el p (10**p)
long double ldexpl(long double (x),int (expon));	Calcula x times (2 raiz a exp)
double ldexp(double x,int expon);	Calcula x times (2 raised to exp)
double exp(double x);	Calcula Potenciacion e a el xth
long double exp(long double (x));	Calcula Potenciacion e a el xth
double sqrt(double x);	Calcula raiz cuadrada
long double sqrtl(long double @E(x));	Calcula raiz cuadrada

Funciones de Logaritmos	
Función	Descripción
double log(double x);	Funcion logaritmo natural
double log10(double x);	Funcion logaritmo comun
long double logl(long double (x));	Funcion logaritmo natural
long double log10l(long double (x));	Funcion logaritmo comun
double hypot(double x,double y);	Calcula hipotenusa de derecha triangulo
long double hypotl(long double (x),long double (y));	Calcula hipotenusa de derecha triangulo

Funciones Trigonometricas	
Función	Descripción
double cos(double x);	Funcion Coseno
double sin(double x);	Funcion Seno
double tan(double x);	Funcion Tangente
long double sinl(long double x);	Funcion Seno
long double tanl(long double x);	Funcion Tangente
long double coshl(long double (x));	Funcion Coseno hiperbolico
double cosh(double x);	Funcion Coseno hiperbolico
double sinh(double x);	Funcion Seno hiperbolico
double tanh(double x);	Funcion Tangente hiperbolico
long double cosl(long double x);	Funcion Coseno
long double sinhl(long double (x));	Funcion Seno hiperbolico
long double tanhl(long double (x));	Funcion Tangente hiperbolico
double acos(double x);	Funcion inversa de conseno
double asin(double x);	Funcion inversa de seno
double atan(double x);	Funcion inversa de tangente
double atan2(double y, double x);	Funcion inversa de tangente2
long double acosl(long double (x));	Funcion inversa de Coseno
long double asinl(long double (x));	Funcion inversa de Seno
long double atanl(long double (x));	Funcion inversa de Tangente

Funciones para Manejo de Bytes	
Función	Descripción
void swab(char *de,char *to,int nbytes);	Swaps bytes
void *memset (void *s,int c,size_t n);	Pone n bytes de s to byte c
void far *far _fmemset(void far *s,int c,size_t n);	Pone n bytes de s a byte c
void movedata(int srcseg,int srcoff,int destseg,int destoff,size_t n);	Copia n bytes
void *memccpy(void *dest,void *origen,int c,size_t n);	Copia un bloque de n bytes de origen a destino
void *memcpy (void *dest,void *origen,size_t n);	Copia un bloque de n bytes de origen a destino
void *memmove(void *dest,void *origen,size_t n);	Copia un bloque de n bytes de origen a destino
void *memchr (void *s,int c, size_t n);	Searches n bytes for caracter c
int far _fmemcmp (void far *s1,void far *s2,ize_t n);	Compara el primer n bytes de strings s1 y s2
int far _fmemicmp(void far *s1,void far *s2, size_t n);	Compara el primer n bytes de strings s1 y s2
void far * far _fmemchr(void far *s,int c, size_t n);	Busca n bytes para caracter c
bcd bcd(int x);	Convierte numero decimal a binario
bcd bcd(double x);	Convierte numero decimal a binario
bcd bcd(double x,int decimals);	Convierte numero decimal a binario

Funciones de Manejo de Bytes	
Función	Descripción
int _control87(int int newcw,int int mask);	Cambia real control word
int memcmp(void *s1,void *s2, size_t n);	Compara el primer n bytes de strings s1 y s2
int memicmp(void *s1,void *s2, size_t n);	Compara el primer n bytes de strings s1 y s2, ignoring case
int mbtowc(wchar_t *pwc,char *s, size_t n);	Convierte un multibyte caracter a wchar_t code
int mblen(char *s, size_t n);	Determina la longitud de un multibyte caracter
int matherr(struct exception *e);	User-modifiable math error handler
int _matherrl(struct _exceptionl *(e));	User-modifiable math error handler
int matherr(struct exception *e);	User-modifiable math error handler
int _matherrl(struct _exceptionl *(e));	User-modifiable math error handler
size_t mbstowcs(wchar_t *pwcs,char *s, size_t n);	Convierte un multibyte string aar_t array

Funciones de Valor Numerico	
Función	Descripción
int abs(int x);	Obtiene el absoluto valor de un entero
double fabs(double x);	Calcula el absoluto valor de un real
long double fabsl(long double @E(x));	Calcula el absoluto valor de un real
long int labs(long int x);	Calcula el absoluto valor de un long
double ceil(double x);	Redondear hacia arriba
double floor(double x);	Redondear hacia abajo
long double ceill(long double (x));	Redondear hacia arriba
long double floorl(long double (x));	Redondear hacia abajo;
int _rotl(unsigned val, int count);	Rota un entero valor a la izquierda
int _rotr(unsigned val, int count);	Rota un entero valor a la derecha
long _lrotr(unsigned long val, int count);	Rota un long valor a la derecha
long _lrotl(unsigned long val, int count);	Rota un long valor a la izquierda
max	Devuelve el alto de dos valores
min	Devuelve el bajo de dos valores TLOSS

Funciones de Numeros Complejos	
Función	Descripción
complex cos(complex z);	Funcion Coseno
complex sin(complex z);	Funcion Seno
complex tan(complex x);	Funcion Tangente
complex cosh(complex z);	Funcion Coseno hiperbolico
complex sinh(complex z);	Funcion Seno hiperbolico
complex tanh(complex x);	Funcion Tangente hiperbolico
complex acos(complex z);	Funcion Inversa de Coseno
complex asin(complex z);	Funcion Inversa de Seno
complex atan(complex x);	Funcion Inversa de Tangente
double abs(complex x);	Obtiene el valor absoluto de un entero

double cabs(struct complex z);	Calcula el valor absoluto de un numero complex
long double cabsl(struct _complexl (z));	Calcula el valor absoluto de un numero complex
complex pow(complex x, complex y);	Funcion Potenciacion , x to the y (x**y)
complex pow(complex x, double y);	Funcion Potenciacion , x to the y (x**y)
complex pow(double x, double y);	Funcion Potenciacion , x to the y (x**y)
complex log(complex x);	Funcion logaritmo natural
complex log10(complex x);	Funcion logaritmo comun
complex exp(complex z);	Calcula e a el zth potenciacion(z is a complex numero)
double imag(complex x);	Devuelve la imaginaria parte de un numero complex
complex sqrt(complex x);	Calcula raiz cuadrada
double conj(complex z);	Conjuga de un numero complex
double arg(complex z);	Obtiene el angulo de un numero en el plano complex
double real(complex x);	Devuelve parte real de numero complex
complex complex(double real,double imag);	Crea numeros complex
complex polar(double mag, double angulo);	Calcula numero complex
double poly(double x, int degree, double coeffs[]);	Calcula numero complex
struct complex {double x, y;};	Crea numeros complex

Constante	Descripción
EDOM	Codigo error para math dominio
ERANGE	Codigo error para resultado fuera de rango
HUGE_VAL	Overflow valor para math funciones
DOMAIN	Argumento no fue en dominio de funcion log(-1)
SING	Argumento debe ser resultado en una singularidad pow(0, -2)
OVERFLOW	Argumento debe tener un funcion result > MAXDOUBLE exp(1000)
UNDERFLOW	Argumento debe tener un funcion result < MINDOUBLE exp(-1000)
TLOSS	Argumento debe tener el total de digitos significantes perdidos in(10e70)
CW_DEFAULT	Default control word for 8087/80287 math coprocessor.
BITSPERBYTE	Numero de bits en un byte.
M_PI	ã
M_PI_2	Uno-half ã (ã/2)
M_PI_4	Uno-cuatro ã (ã/4)
M_1_PI	Uno dividido por ã (1/ã)
M_2_PI	Dos dividido por ã (2/ã)
M_1_SQRTPI	Uno dividido por raiz cuadrada de ã (1/ûã)
M_2_SQRTPI	Dos dividido por raiz cuadrada de ã (2/ûã)
M_E	El valor de e
M_LOG2E	El valor de log(e)
M_LOG10E	El valor de log10(e)
M_LN2	El valor de ln(2)
M_LN10	El valor de ln(10)
M_SQRT2	Raiz Cuadrada de 2 (û2)
M_SQRT_2	1/2 la raiz cuadrada de 2 (û2)/2

Funciones varias para Disco

Función	Descripción
int system(char *command);	Se ejecuta un comando de el DOS
void sound(int freq);	Pone una frequencia en la bocina
void nosound(void);	Apaga el sonido de la bocina
void delay(int milseg);	Suspende ejecucion para el intervalo
void sleep(int seg);	Suspende ejecucion para intervalo
void setverify(int valor);	Pone verificacion estado
void exit(int status);	Terminacion de el programa
void _cexit(void);	Ejecuta la salida limpia sin salir de el programa
void _dos_keep(char status,int n);	Sale y permanece residente
void keep(char status,int n);	Sale y permanece residente
char *_strerror(char *s);	Construye un mensaje de error ordinario
char *strerror(int errnum);	Devuelve un apuntador a un mensaje de error
int fail();	Devuelve nonzero si una fallo la operacion
int dosexterr(struct DOSERROR *eblkp);	Obtiene informacion DOS de error extendido
int getcbrk(void);	Obtiene la verificacion de el control-break

int setcbrk(int valor);	Pone verificacion ctrl-break
void ctrlbrk(int (*handler)(void));	Pone manejador de control-break
void getfat(int char dis,struct fatinfo *table);	Obtiene informacion de FAT
void getfatd(struct fatinfo *dtable);	Obtiene informacion de FAT
int getpid(void);	Obtiene el proceso ID de el programa
int getverify(void);	Obtiene el verificacion de estado
int isatty(int handle);	Verifica para un tipo de dispositivo
int mbtowc(wchar_t *O,char *D, size_t n);	Convierte un multibyte caracter a un wchar_t
int mblen(char *s,size_t n);	Determina la longitud de un multibyte caracter
void __emit__(argument,);	Inserta literal valor directamente en el codigo
void poke(int seg,int offset,int valor);	Obtiene un valor entero de la memoria
void pokeb(int seg,int offset,char valor);	Obtiene un byte valor de locacion de memoria
char *getenv(char *nom);	Obtiene un string desde el ambiente
int putenv(char *nom);	Añade un string a el actual ambiente
int freemem(int segx);	Libera un bloque de la memoria
int setblock(int segx,int n);	Modifica el tamaño de un bloque
int randbrd(struct fcb *fcb,int rcnt);	Ejecuta una lectura aleatoria usando FCB
int randbwr(struct fcb *fcb,int rcnt);	Ejecuta una escritura aleatoria usando FCB
void movmem(void *O,void *D,int n);	Mueve un bloque de n bytes de origen a dest
int execl(char *path,char *arg0,,NULL);	Carga un programa y corre otro(proceso hijo)
int fnsplit(char *path,char *dis,char *dir,char *nom,char *ext);	Divide un path en sus componentes
void fnmerge (char *path,char *dis,char *dir,char *nom,char *ext);	Construye un path

Funciones de Manejo de Archivos

Función	Descripción
int umask (int modeMask);	Pone permiso aparente de e/s al archivo
char *tempnam(char *dir,char *prefix);	Crea un nombre de archivo unico en directorio especificado
char *searchpath(char *arch);	Busca el path de el DOS para un archivo
int renom(char *oldnom,char *newnom);	Renombra un archivo
int remove(char *nom);	Remueve un archivo
char *parsfnm(char *cmdline,struct fcb *fcb,int opt);	Analiza archivo y construye un archivo de control bloques

Funciones de Interrupciones

Función	Descripción
void disable(void);	Deshabilita interrupciones
void _disable(void);	Deshabilita interrupciones
void enable(void);	Habilita interrupciones
void _enable(void);	Habilita interrupciones
void setvect(int interruptno,void interrupt (*isr) ( ));	Pone vector de interrupcion
void interrupt(*_dos_getvect(int intnum)) ();	Obtiene el vector de interrupcion
void interrupt (*getvect(int interruptno))();	Obtiene el vector de interrupcion
void geninterrupt(int intr_num);	Genera una software interrupcion
void _dos_setvect(int interruptno,void interrupt (*isr) ());	Pone vector de interrupcion
void _chain_intr(void (interrupt far *newhandler)());	Cambia a otro manejador interrupcion
void intr(int intno,struct REGPACK *preg);	Alterna Interrupciones del 8086
int int86(int intno,union REGS *inregs,union REGS *outregs);	Interrupciones del 8086
int int86x(int intno,union REGS *inregs,union REGS *outregs,struct SREGS *segregs);	Interrupciones del 8086
int intdos(union REGS *inregs,union REGS *outregs);	Interrupciones del DOS
int intdosx(union REGS *inregs,union REGS *outregs,struct SREGS *segregs);	Interrupciones del DOS

Funciones del BIOS

Función	Descripción
long biostime(int cmd,long newtime);	Lee o pone la hora del BIOS
int biosequip(void);	Verifica equipo
int bioskey(int cmd);	Interface con teclado,usando servicios del BIOS
int biosmemory(void);	Devuelve tamaño de memoria
int biosprint(int cmd,int abyte,int puerto);	Impresora e/s usando servicios del BIOS.
int bioscom(int cmd,char abyte,int puerto);	RS-232 communicaciones (serial e/s)
int _bios_timededia(int cmd,long *timep);	Lee o pone la BIOS hora
int _bios_equiplist(void);	Verifica equipo

int _bios_keybrd(int cmd);	Interface con teclado,usando servicios BIOS
int _bios_memsize(void);	Devuelve tamaño de memoria
int _bios_serialcom(int cmd,int puerto,char abyte);	RS-232 communicaciones (serial e/s)
int _bios_printer(int cmd,int puerto,int abyte);	Impresora e/s usando servicios BIOS
int _bios_disco(int cmd,struct discoinfo_t *dinfo);	Servicios BIOS disco disco

Funciones de Direccionamiento	
Función	Descripción
int inp(int portid);	Lee un byte desde un puerto del hardware
int inpw(int portid);	Lee un word desde un puerto del hardware
int inport(int portid);	Lee a word desde un puerto del hardware
int char inportb(int portid);	Lee a byte desde un puerto del hardware
int outp(int portid,int value);	Sale un byte a un puerto del hardware
int outpw(int portid,int value);	Sale un word a un puerto del hardware
void outport (int portid,int value);	Sale un word a un puerto del hardware
void outportb(int portid,int char value);	Sale un byte a un puerto del hardware
void segread(struct SREGS *segp);	Lee segmento de registros
int FP_OFF(void far *p);	Pone una direccion de desplazamiento
int FP_SEG(void far *p);	Pone una direccion de segmento
void far *MK_FP(int seg,int ofs);	Hace un apuntador far
int bdos(int dosfun,int dosdx,int dosal);	Accesos a llamadas del sistema DOS
int bdosptr(int dosfun,void *argument,int dosal);	Accesos a llamadas del sistema MS-DOS
void hardresume(int axret);	Error de hardware manejador rutinas
void hardretn(int retn);	Error de hardware manejador rutinas
int getpsp(void);	Obtiene el prefijo de segmento del programa

Funciones de Disco	
Funcion	Descripcion
int setdisk(int dis);	Pone el actual numero de disco
int _chdrive(int dis);	Pone el actual numero de disco
int _dos_setdrive(int dis,int *ndis);	Pone el actual numero de disco
int getdisk(void);	Obtiene el actual numero de disco
int _getdrive(void);	Obtiene el actual numero de disco
void _dos_getdrive(int *dis);	Obtiene el actual numero de disco
int absread(int dis,int nsec,long lsect,void *buf);	Lee sectores absoluto disco
int abswrite(int dis,int nsec,long lsect,void *buf);	Escribe sectores absoluto disco
void setdta(char far *dta);	Pone direccion transferencia de disco
char far *getdta(void);	Obtiene direccion transferencia de disco
int _dos_getdiscofree(char dis,struct disfree_t *table);	Obtiene espacio libre del disco
void getdfree(char dis,struct dfree *table);	Obtiene espacio libre del disco

Funciones de Directorio	
Funcion	Descripcion
int mkdir(char *path);	Crea un directorio
int chdir(char *path);	Cambia directorio actual
int rmdir(char *path);	Remueve un archivo DOS directorio
char *getcwd(char *buf,int buflen);	Obtiene el directorio actual trabajo
int getcurdir(int drive,char *directorio);	Obtiene directorio actual para disco especificado
char *_getdcwd(int drive,char *buffer, int buflen);	Obtiene el directorio actual para disco especificado
DIR *opendir(char *dirnom);	Abre un directorio stream para lectura
void closedir(DIR *dirp);	Cierra un directorio stream
struct dirent readdir(DIR *dirp);	Lee la entrada actual desde un directorio stream
int _dos_findnext(struct find_t *ffblk);	Continua la busqueda un disco directorio para archivos
int findnext(struct ffblk *ffblk);	Continua busqueda un disco directorio para archivos

Funciones de Fecha y Hora	
Funcion	Descripcion
time_t time(time_t *timer);	Pone hora de dia
void settime(struct time *timep);	Pone hora sistema

int _dos_settime(struct dostime_t *timep);	Pone hora sistema
void gettime(struct time *timep);	Obtiene hora sistema
void _dos_gettime(struct dostime_t *timep);	Obtiene hora sistema
int stime(time_t *tp);	Pone fecha y hora de el sistema
void getdate(struct date *datep);	Obtiene fecha sistema DOS
void _dos_getdate(struct dosdate_t *datep);	Obtiene fecha sistema DOS
void setdate(struct date *datep);	Pone fecha sistema DOS
void _dos_setdate(struct dosdate_t *datep);	Pone fecha sistema DOS
time_t mktime(struct tm *t);	Convierte hora a formato calendario
long dostounix(struct date *d,struct dostime *t);	Convierte fecha y hora a formato UNIX
clock_t clock(void);	Devuelve n de ticks reloj desde inicio del programa
size_t _cdecl strftime(char *s, size_t maxsize,char *fmt,struct tm *t);	Formato hora para salida

Constante	Descripcion
MAXPATH	Completo archivo nombre con path
MAXDRIVE	Disk drive (e.g., "A:")
MAXDIR	Archivo subdirectorio especificacion
MAXFILE	Archivo nombre sin extension
MAXEXT	Archivo extension
FA_RDONLY	Lectura solamente atributo
FA_HIDDEN	Hidden archivo
FA_SYSTEM	Sistema archivo
FA_LABEL	Volumen Etiqueta
FA_DIREC	Directorio
FA_ARCH	Archivo
EZERO	Error 0
EINVFNC	Invalido funcion numero
ENOFILE	Archivo no encontrado
ENOPATH	Path no encontrado
ECONTR	Memoria bloques destruido
EINVMEM	Invalido memoria bloque direccion
EINVENV	Invalido ambiente
EINVFMT	Invalido formato
EINVACC	Invalido acceso codigo
EINVDAT	Invalido dato
EINVDRV	Invalido disco especificado
ECURDIR	Attempt a remover CurDir
ENOTSAM	No mismo dispositivo
ENMFILE	No mas archivos
ENOENT	No such archivo o directorio
EMFILE	Muchos archivos abiertos
EACCES	Permiso denegado
EBADF	Malo archivo numero
ENOMEM	No suficiente memoria
ENODEV	No hay dispositivo
EINVAL	Invalido argumento
E2BIG	Argumento muy largo
ENOEXEC	Ejecucion formato error
EXDEV	Dispositivo enlazador
EDOM	Matematico argumento
ERANGE	Resultado muy largo
EFAULT	Error desconocido
EEXIST	Archivo ya existe
EXIT_SUCCESS	Normal programa terminacion
EXIT_FAILURE	Anormal programa terminacion

Funciones Varias de Graficos	
Funcion	Descripcion
int registerbgidriver(void (*dis)(void));	Registros enlazado en disco graficos
int register bgidisco(void *dis);	Registros enlazado en disco graficos
int registerbgifont(void (*font)(void));	Registros enlazado en font

int register bgifont(void *font);	Registros enlazado en font
int installuserdisco(char *nom,int huge (*detect)(void));	Instala tabla BGI a disco
int installuserfont(char *nom);	Instala un archivo font
int getgraphmode(void);	Devuelve el modo grafico actual
char* getmodenom(int mod_num);	Devuelve el nombre del modo grafico especificado
void setgraphmode(int mod);	Pone el modo graficos del sistema ,limpia la pantalla
void getmodorange(int disco,int *lmod,int *hmod);	Obtiene el rango de modos graficos para un disco
char* getdisconom(void);	Devuelve un apuntador al nombre del disco grafico actual
void setwritemode(int modo);	Pone el modo escritura para dibujar lineas
void _graphfreemem(void *ptr,int size);	Pone memoria de graficos
void* _graphgetmem(int size);	Devuelve memoria de graficos
void setactivepage(int page);	Pone la pagina activa para salida de graficos
void setvisualpage(int page);	Pone el numero de la pagina visual grafica
void detectgraph(int *dis, int *mod);	Determina y verifica el hardware para graficos
void initgraph(int *dis,int *mod,char *pathdis);	Inicializa el graficos del sistema
void closegraph(void);	Cierra el graficos del sistema
void graphdefaults(void);	Limpia todos los settings graficos
char* grapherrormsg(int error);	Devuelve un apuntador a un mensaje de error
int graphresult(void);	Devuelve un codigo de error de operacion grafica

Control	Valor
Detected	0
CGA	1
MCGA	2
EGA	3
EGA64	4
EGAMONO	5
IBM8514	6
HERCMONO	7
ATT400	8
VGA	9
PC3270	10

Tipos de Error	
Valor	Descripcion
0 grOk	No hay error
-1 grNoInitGraph	Graficos no instalados(usa initgraph)
-2 grNotDetected	Graficos de hardware no detectado
-3 grFileNotFound	Archivo no encontrado
-4 grInvalidDriver	Disco invalido
-5 grNoLoadMem	No hay memoria para cargar el disco
-6 grNoScanMem	NO RAM para exploracion de fondo
-7 grNoFloodMem	No RAM para exploracion de flujo
-8 grFontNotFound	Archivo font no encontrado
-9 grNoFontMem	No hay memoria para cargar font
-10 grInvalidMode	Invalido modo grafico
-11 grError	Error graficos
-12 grIOerror	Error graficos e/s
-13 grInvalidFont	Invalido archivo de font
-14 grInvalidFontNum	Invalido numero de font
-15 grInvalidDeviceNum	Invalido numero de dispositivo
-18 grInvalidVersion	Invalido numero de version

Funciones de Lineas	
Funcion	Descripcion
void getlinesettings(struct linesettingstype *lineinfo);	Obtiene el actual linea estilo, modelo, y espesor
void setlinestyle(int estilo,int modelo,int espesor);	Pone el actual linea estilo y anchura y modelo

Tipos de Lineas		Modos de Lineas	
Funcion	Descripcion	Funcion	Descripcion

SOLID_LINE	0 Solida linea	COPY_PUT0	Copia origen bitmap onto pantalla
DOTTED_LINE	1 Dotted linea	XOR_PUT 1	Excluye ORs origen imagen con que ya en pantalla
CENTER_LINE	2 Centered linea	OR_PUT 2	Incluye ORs imagen with que ya en pantalla
DASHED_LINE	3 Dashed linea	AND_PUT 3	ANDs imagen con que ya en pantalla
USERBIT_LINE	4 Usuario-definido linea estilo	NOT_PUT 4	Copia el inverso de el origen

Funciones de Texto

Funcion	Descripcion
void outtext(char *textstring);	Despliega un string en la ventana grafica
void outtextxy(int x,int y,char *textstring);	Despliega un string en la posicion especificada (graficos modo)
void textbackground(int newcolor);	Selecciona un nuevo color background para texto
void textcolor(int newcolor);	Selecciona un nuevo color caracter en texto modo
int textheight(char *textstring);	Devuelve la altura de un string en pixeles
int textwidth(char *textstring);	Devuelve la achura de un string en pixeles
void textmodo(int newmodo);	Cambia pantalla modo (en modo texto)
void setusercharsize(int multx,int divx,int multy,int divy);	Coloca el tamaño para un caracter definido por usuario
void settextjustify(int horiz,int vert);	Pone justificacion de texto para modo grafico
void settextstyle(int font,int direction,int charsize);	Pone las caracteristicas de el texto actual
void gettextsettings(struct textsettingstype *texttypeinfo);	Obtiene informacion acerca de el actual texto grafico

Tipografia	Valor
Default	0
Triplex.chr	1
Litt.chr	2
Sansheirf.chr	3
Gothic.chr	4
Script.chr	5
Simplex.chr	6
European.chr	7
Double.chr	8
Bold.chr	9

Justificacion de Texto

Funcion	Descripcion
horizontal	
LEFT_TEXT	0 Justificacion izquierda
CENTER_TEXT	1 Justificacion centrada
RIGHT_TEXT	2 Justificacion derecha
vertical	
BOTTOM_TEXT	0 Justificacion abajo
CENTER_TEXT	1 Justificacion centrada
TOP_TEXT	2 Justificacion alta

Figuras Geometricas

Figura	Descripcion
Putpixel(x,y,color);	Dibuja un pixel en x,y
Line(x,y,x2,y2);	Traza una linea de x,y hasta x2,y2
Linerel(x,y);	Traza una linea de 0,0 hasta x2,y2
Lineto(x,y);	Traza una linea de la ultima linea hasta x,y
Arc(x,y,Grado1,Grado2,Radio);	Traza un arco de centro x,y de Grado1 a Grado2
Bar(x,y,x2,y2);	Traza una barra de ESI x,y hasta EID x2,y2
Rectangle(x,y,x2,y2);	Traza un Rectangulo de ESI x,y hasta EID x2,y2
Pieslice(x,y,Grado1,Grado2,Radio);	Traza un Pay de centro x,y con radio de Grado1 a Grado 2
Bar3d(x,y,x2,y2,Profundidad,topon/topoff);	Traza una barra en 3D de ESI x,y hasta EID x2,y2
Circle(x,y,Radio);	Traza un circulo de centro x,y con Radio
Sector(x,y,Grado1,Grado2,Radiox,Radioy);	Traza el sector de una elipse de centro x,y con dos Radios
Ellipse(x,y,Grado1,Grado2,Radiox,Radioy);	Traza una Elipse de centro x,y con dos Radios
Drawpoly(Sizeof(arreglo) div size(arreglo));	Traza un Poligono contenido en un arreglo
Fillellipse(x,y,Radiox,Radioy);	Rellena una Elipse de x,y con dos Radios
Fillpoly(Sizeof(arreglo) div size(arreglo)());	Rellena un poligono contenido en un arreglo

Floodfill(x,y,Borde));	Rellena una figura cerrada que tiene color de borde
Setfillstyle(Estilo,Color);	Coloca estilo de relleno

Funciones de Ventanas

Funcion	Descripcion
void setviewport(int A,int B,int C,int D,int clip);	Delimita una ventana grafica
void clearviewport(void);	Limpia la ventana grafica
void cleardevice(void);	Limpia la graficos pantalla
void getfillpattern(char *pattern);	Copia un modelo definido por usuario en la memoria
void getfillsettings (struct fillsettingstype *fillinfo);	Obtiene informacion de modelo y color actual de relleno
void setfillpattern(char *upattern,int color);	Selecciona un modelo relleno definido por usuario

Funciones de Pantalla

Funcion	Descripcion
int getx(void);	Devuelve el actual posicion en x horizontal
int gety(void);	Devuelve el actual posicion en y vertical
int getmaxx(void);	Devuelve maximo x horizontal
int getmaxy(void);	Devuelve maximo y vertical
void getarccoords(struct arccoordstype *arccoords);	Obtiene coordenadas de la ultima llamada a arc
void moverel(int dx,int dy);	Mueve la actual posicion a una relativa distancia
void moveto(int x,int y);	Mueve la posicion de el cursor a x, y
void getviewsettings (struct viewporttype *viewport);	Obtiene informacion acerca la actual ventana grafica
int getpixel(int x, int y);	Obtiene el color del pixel especificado
void getlinesettings(struct linesettingstype *lineinfo);	Obtiene el actual line estilo, modelo, y espesor
int getmaxmodo(void);	Devuelve numero maximo modos graficos para actual disco
void getaspectratio(int *xasp,int *yasp);	Devuelve maximo graficos modo numero para actual disco
void setaspectratio(int xasp,int yasp);	Pone la relacion de aspecto de graficos
void getimage(int A,int B,int C,int D,void *bitmap);	Salva una imagen de la region especificada,en la memoria
void putimage(int A,int B,void *bitmap,int op);	Pone una imagen en la pantalla
int imagesize(int A,int B,int C,int D);	Devuelve el numero de bytes requeridos para un bit imagen

Funciones de Color

Funcion	Descripcion
int getmaxcolor(void);	Devuelve maximo valor para color
void getpalette(struct palettetype *palette);	Obtiene informacion acerca la actual paleta
int getpalettesize(void);	Devuelve tamaño de paleta color lookup tabla
void getdefaultpalette(void);	Devuelve la paleta definicion estructura
void setcolor(int color);	Devuelve la paleta definicion estructura
int getcolor(void);	Devuelve el color actual
int setgraphbufsize(int bufsize);	Cambia el tamaño de el buffer grafico interno
void setpalette(int colornum,int color);	Cambia un color de la paleta
int getbkcolor(void);	Devuelve el color background actual
void setbkcolor(int color);	Pone el color background actual usando la paleta
void setallpalette(struct palettetype *palette);	Cambia todos los colores de la paleta especificada

Formato de Relleno

Modelo	Valor	Descripccion
Empty_Fill	0	Rellena con color de fondo
Solid_ Fill	1	Rellena con color solido
Line_ Fill	2	Rellena con lineas
Ltslash_ Fill	3	Rellena con barras claras
Slash_ Fill	4	Rellena con barras
Bkslash_ Fill	5	Rellena con barras inclinadas
Ltkslash_ Fill	6	Rellena con barras inclinadas claras
Hatch_ Fill	7	Rellena con entramado claro
Xhatch_ Fill	8	Rellena con entramado
Interleave_ Fill	9	Rellena con interleaving
Widedot_ Fill	10	Rellena con puntos espaciados
Closedot_ Fill	11	Rellena con puntos juntos

User_ Fill	12	Rellena con modelo definido

Declaracion de punteros

int *p;	p es un puntero a un entero
int *p[10];	p es un array de 10 punteros a enteros
int (*p)[10];	p es un puntero a un array de 10 enteros
int *p(void);	p es una función que devuelve un puntero a entero
int p(char *a);	p es una función que acepta un argumento que es un puntero a carácter, devuelve un entero
int *p(char *a);	p es una función que acepta un argumento que es un puntero a carácter, devuelve un puntero a entero
int (*p)(char *a);	p es un puntero a función que acepta un argumento que es un puntero a carácter, devuelve un puntero a entero
int (*p(char *a))[10];	p es una función que acepta un argumento que es un puntero a carácter, devuelve un puntero a un array de 10 enteros
int p(char (*a)[]);	p es un puntero a función que acepta un argumento que es un puntero a un array de caracteres, devuelve un puntero a entero
int p(char *a[]);	p es un puntero a función que acepta un argumento que es un array de punteros a caracteres, devuelve un puntero a entero
int *p(char a[]);	p es una función que acepta un argumento que es un array de caracteres, devuelve un puntero a entero
int *p(char (*a)[]);	p es una función que acepta un argumento que es un puntero a un array de caracteres, devuelve un puntero a entero
int *p(char *a[]);	p es una función que acepta un argumento que es un puntero a un array de punteros a caracteres, devuelve un puntero a entero
int (*p)(char (*a)[]);	p es una función que acepta un argumento que es un puntero a un array de caracteres, devuelve un puntero a entero.
int *(*p)(char (*a)[]);	p es un puntero a una función que acepta un argumento que es un puntero a un array de punteros a caracteres, devuelve un puntero a entero
int *(*p)(char *a[]);	p es un puntero a una función que acepta un argumento que es un array de punteros a caracteres, devuelve un puntero a entero

CAPITULO XI:

Bibliografía

PROGRAMACIÓN PROFESIONAL EN C - Editorial Microsoft Press. - Autor: Steve Schustock.

PROGRAMACIÓN EN C: Introducción y conceptos avanzados. - Editorial : Anaya Multimedia. - Autor: Mitchell Woite, Stephen Prota, Donald Martin

LENGUAJE C - Editorial : Anaya Multimedia. - Autor: F. Javier Moldes Teo.

CURSO DE PROGRAMACIÓN C/*C*++ - Editorial: Ra - ma. - Autor: F. Javier Ceballos.

Paginas WEB

http://Programandoenc.webcindario.com

http://usuarios.lycos.es/ismaelcamero

http://www.ciens.ula.ve/~amoret/c_plus.html

http://www.cvc.uab.es/shared/teach/a21292/docs/cpp_arg.ppt

http://www.lcc.uma.es/~pastrana/LP/tema2.pdf

http://www.geocities.com/studioxl/hc.htm

http://www.mundovb.net/mundoc/capitulo1-1.htm

www.monografias.com/trabajos/introc/introc.shtml

http://w3.mor.itesm.mx/~jfrausto/Algoritmos/estructuras/notas/C++_intro.html

http://www.zator.com/Cpp/E5.htm

http://www.hispan.com/eltaller/Taller4.htm

http://lacarcel.iespana.es/lacarcel/curso_c_2.htm#Bibliografia

http://elqui.dcsc.utfsm.cl/util/C/EjerciciosC-2/lengc.htm#contenido

http://www.geocities.com/ismaelcamarero