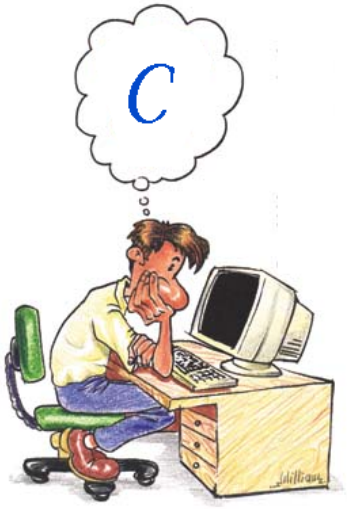
UNIVERSIDAD DE CONCEPCIÓN
FACULTAD DE INGENIERÍA
DEPARTAMENTO DE INGENIERÍA ELÉCTRICA

Lenguaje de Programación

Apuntes de C



Profesor: Ricardo Sánchez S.

Indice

Introducción

Estos apuntes no pretenden ser un tratado completo del Lenguaje de Programación C, que explique a cabalidad cada función del estándar y menos aún pretende explorar todas las posibilidades de este versátil lenguaje. Nos limitaremos más bien a entregar las herramientas básicas que permitan un desarrollo rápido, pero completo de programas de todo tipo, explicando no sólo las funciones estándar del ANSI C (C determinado por el Instituto Americano de Normas Nacionales, “American National Standars Institute”), sino también aquellas funciones necesarias específicas del compilador y así como los conceptos básicos para la comprensión completa de los programas.

Parte 1, Conceptos Básicos de Informática (o Conociendo el PC)

Un computador es una herramienta. Un herramienta terriblemente compleja, pero simplificada por los programadores hasta hacerlo MUY sencillo de usar. Un usuario es la persona que ocupa un programa (software) para resolver algún problema específico. Un programador (ustedes) es aquella persona que escribe el programa.

Funcionamiento Básico

Un computador está compuesto principalmente por 3 partes:
Unidad Central de Proceso (CPU)
Periféricos de Entrada
Periféricos de Salida

Esa es la definición clásica. En términos simples, la CPU es la que procesa, es decir, transforma los datos de entrada, en información útil de salida. Los datos de entrada son ingresados al computador a través de los periféricos de entrada y son obtenidos por el usuario a través de los periféricos de salida. Usando esta clasificación clásica,

enumeramos a continuación los componentes básicos distinguibles en un computador moderno:

- Procesador (CPU) (El que hace el trabajo)
- Memorias (CPU) (Guardan la información con la que se está trabajando)
- Tarjeta Madre (CPU/Periféricos de Entrada/Salida) (Aquí montamos todo, interconecta y controla la interacción de bajo nivel de los distintos componentes)
- Disco Duro (Periférico de Entrada/Salida) (Archivos, directorios, etc)
- Tarjeta de video (Periférico de Salida) (Permite conectar el monitor, incluye por lo general la aceleradora gráfica)
- Monitor (Periférico de Salida) (si, ese que parece televisor)
- Disquetera (Periférico de Entrada/Salida)
- Mouse (Periférico de Entrada)
- Teclado (Periférico de Entrada)

Hasta ahí tenemos a duras penas lo que actualmente consideramos un PC. Agreguémosle:

- Impresora (Periférico de Salida)
- Lector de CD (Periférico de Entrada)
- Tarjeta de Sonido (Periférico de Entrada/Salida)
- Parlantes (Periférico de Salida)

Y ya tenemos un PC funcional típico.

¿Y como funciona todo esto? Pues simplificando mucho: cuando uno ejecuta un programa, este es cargado desde el disco duro a la memoria principal (RAM) la cual procesa una a una las líneas del programa cargado, leyendo cuando es necesario información de los periféricos de entrada (disco duro, teclado, mouse) y enviando cuando es necesario información de salida a través de los periféricos de salida (disco duro, parlante, tarjeta de video, etc). Esto es en DOS (el sistema operativo que usaremos). En un sistema operativo más complejo, multitarea, pueden haber varios programas (procesos) corriendo al mismo tiempo, siendo el sistema operativo el encargado de gestionar el paso de uno a otro.

Simplificando y abusando un poco del lenguaje, llamaremos PC (personal computer) al computador personal que usamos normalmente; llamaremos “memoria” a la memoria principal (RAM) y llamaremos *variables* a la información guardada en memoria. Por otra parte, llamaremos “disco duro” o simplemente “disco” a las unidades de disco del sistema (disco duro, lector/grabador de cd, disquetera, disco Zip, etc) y “archivos” a la información guardado en estos. El procesador no trabaja directamente con el disco duro, por lo que cualquier información que deseemos usar que esté almacenada en un archivo deberemos primero cargarla a memoria. (Nota: la memoria es al menos 1000 veces más rápida que el disco duro!!!, pero tiene mucho menor tamaño y sus datos son volátiles (si apagas el PC, pierdes los datos)).

Toda la información usada por el PC es información digital, guardada en bits. Un bit es la entidad mínima de información y puede tener como valor un “0” ó un “1” (que equivalen a valores de voltaje. Por ejemplo: “0”=0[Volt] y “1”=5[Volts]). Ocho bits forman un byte, que es la unidad básica de información con la que comúnmente se trabaja.

Conceptos básicos de programación (en DOS)

CÓDIGO FUENTE: Es un archivo de texto plano que contiene una serie de sentencias (instrucciones, líneas de código), escrito por el programador en algún lenguaje de programación. En el caso de C, estos archivos tiene extensión .C (el código en sí) y .H (los archivos de cabecera). Estos archivos no son directamente comprendidos por el procesador, por lo que deben ser convertidos a un lenguaje comprensible por el computador.

CÓDIGO OBJETO: Mediante un programa llamado *Compilador* (Compiler), el Código Fuente es transformado a un lenguaje entendible por el procesador. Los archivos en código objeto, en C tienen la extensión .OBJ.

EJECUTABLE: A menudo, en nuestro programa usamos librerías de funciones pre-hechas o tenemos varios códigos fuentes interrelacionados entre sí. Los códigos objetos tienen una tabla de símbolos en los cuales guardan referencias a funciones y variables externas, no definidas en el mismo código fuente. Una vez obtenidos todos los códigos objeto necesarios (los códigos fuente fueron

compilados), un programa llamado *Enlazador* (Linker), enlaza las distintas referencias y crea un único archivo final, llamado Ejecutable, que será el que cargará en memoria el sistema operativo y que el procesador ejecutará paso a paso. Los archivos ejecutables generados en C tienen la extensión .EXE.

Actualmente, el compilador y enlazador vienen integrados en paquetes de software que incluyen, además del enlazador y compilador, un editor (que por lo general colorea la sintaxis propia del lenguaje de programación usado) y algún sistema de ayuda. En nuestro caso en particular, usaremos el paquete Borland C++ 3.1, para programación bajo DOS.

Orígenes de C

La base del C proviene del BCPL, escrito por Martin Richards, y del B escrito por Ken Thompson en 1970 para el primer sistema UNIX en un DEC PDP-7. Estos son lenguajes sin tipos, al contrario que el C que proporciona varios tipos de datos. Los tipos son caracteres, números enteros y en coma flotante, de varios tamaños. Además se pueden crear tipos derivados mediante la utilización de punteros, vectores, registros y uniones. El primer compilador de C fue escrito por Dennis Ritchie para un DEC PDP-11 y escribió el propio sistema operativo en C.

Características

C está caracterizado por ser de uso general, de sintaxis sumamente compacta y de alta portabilidad. Es un lenguaje de nivel medio porque combina elementos de lenguajes de alto nivel, manipulación de bits, bytes, direcciones y elementos básicos como números y caracteres.

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

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. Esto permite modularizar las distintas partes del programa y reutilizar código.

Estructura del lenguaje C

Todo programa en C consta de una o más funciones, una de las cuales se llama **main**. El programa comienza en la función main, desde la cual es posible llamar a otras funciones.

La manera más fácil de entender la estructura del lenguaje C es con un sencillo ejemplo. El siguiente programa es el primer programa típico, estándar, tradicional o como quiera llamársele, usado por la mayoría de los autores como ejemplo de un programa en C.

```
#include <stdio.h>

void main(void)
{
                                /*  Cuerpo del Programa */
      printf( "Hola mundo\n" );  /* Esto imprime “Hola mundo” en pantalla */
      return;
}
```

Ahora, explicaremos línea por línea el programa:

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

#include es lo que se llama una directiva de inclusión de archivo de cabecera (header). Sirve para indicar al compilador que incluya otro archivo. Cuando en compilador se encuentra con esta directiva la sustituye por el archivo indicado. En este caso es el archivo stdio.h que es donde está definida la función printf, que veremos luego.

```
void main(void)
```

Es la función principal del programa. Todos los programas de C deben tener una función llamada main, que será la primera que se ejecute al llamarse el programa… El primer `void` que tiene al principio significa que cuando la función main acabe no devolverá nada, el segundo `void` indica que la función main no requiere argumentos (esto lo explicaremos más claramente después).

```
{
```

Esta llave indica el comienzo de una función, en este caso la función main.

```
/* Cuerpo del programa */
```

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

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

es perfectamente válido. Un comentario comienza con los caracteres `/*` y termina con los caracteres `*/`. Cualquier cosa entre ambos caracteres será interpretado por el compilador como un comentario.

```
printf( "Hola mundo\n" );
```

Aquí es donde por fin el programa hace algo que podemos ver al ejecutarlo. La función printf muestra un mensaje por la pantalla. Al final del mensaje "Hola mundo" aparece el símbolo '\n'; este hace que después de imprimir el mensaje se pase a la línea siguiente.
El carácter ';' corresponder al final de una sentencia. Ésta es la forma que se usa en C para separar una instrucción de otra. Se pueden poner varias en la misma línea siempre que se separen por el punto y coma.

```
return;
```

Hace que el programa devuelva un valor o, en este caso, como elegimos no retornar nada, nos sirve únicamente para volver de una función.

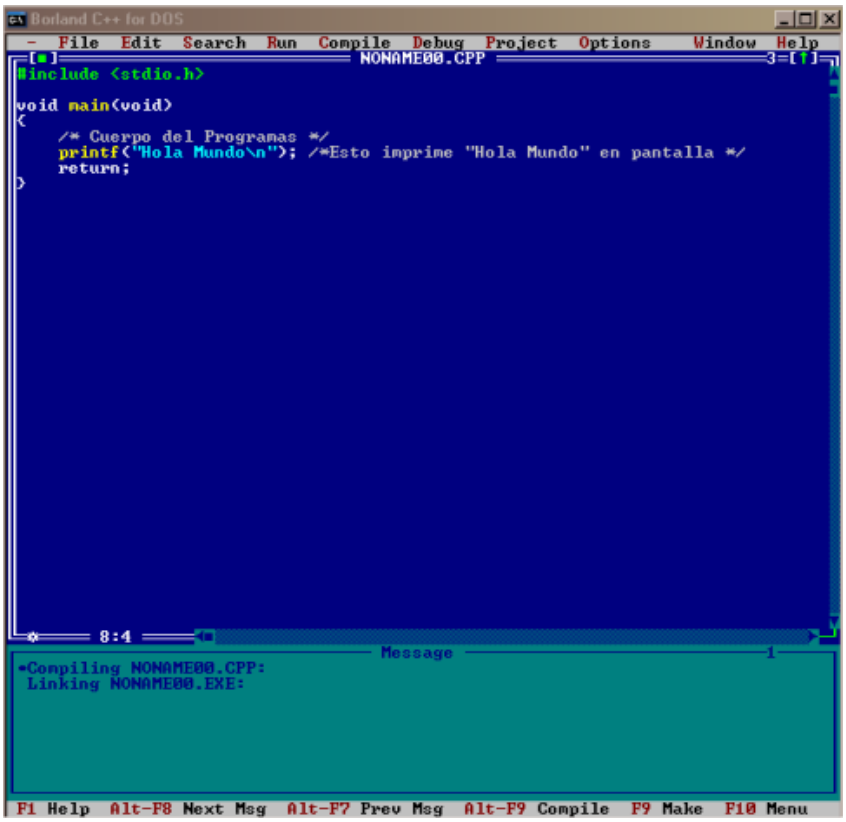
```
}
```

Esta llave indica el fin de la función. Si antes de llegar a esta llave no se ha puesto ningún `return`, la función terminaría aquí.

Compilando

Antes de explicar que son las funciones, compilemos el programa y veamos que tal se ve. Para esto abrimos el Borland C/C++ 3.1 (ejecutando el archivo BC.EXE dentro del directorio BIN, dentro del directorio donde está instalado el programa). Esto nos abre el editor.

En el menú *File* escogemos la opción *New*, con lo que nos encontraremos con la siguiente pantalla:



La ventana azul corresponde al espacio de edición, donde escribimos el código fuente (escribir ahí el código de ejemplo). La ventana calipso de más abajo muestra los mensajes del compilador. Una vez escrito el programa puede guardarse usando la opción *Save* del menú *File* (guardarlo como primero.c) . En el menú *Compile*, encontramos las opciones *compile*, *make* y *link*. *Compile* y *link* compilan el código fuente y enlazan el código objeto, respectivamente. *Make* simplemente hace ambos, primero compila y luego enlaza, generando el ejecutable. Elijamos pues la opción *make*. Esta creará el archivo ejecutable primero.exe (si es que guardamos el archivo con el nombre indicado).

Para ejecutarlo y ver que hace podemos volver a DOS y escribir “primero.exe” y presionar enter, en el lugar donde se encuentra el archivo o simplemente, eligiendo la opción *Run* del menú *Run*. El resultado es desplegado rápidamente y volvemos al editor. Para ver el resultado por más tiempo podemos apretar ALT+F5. Apretando Escape volvemos al editor.

Identificadores

Los identificadores son nombres que se les da a varios elementos de un programa, como variables, constantes, funciones.
Un identificador está formado por letras y dígitos, en cualquier orden, excepto el primer carácter, que debe ser una letra. Se pueden utilizar mayúsculas y minúsculas (no tildadas), aunque no se pueden intercambiar, esto es, una letra mayúscula no es equivalente a su correspondiente minúscula. El carácter de subrayado ‘_’ se puede incluir también y se suele utilizar en la mitad de los identificadores. No se limita la longitud de los identificadores aunque el compilador sólo reconoce los 31 primeros caracteres.
Hay ciertas palabras reservadas, denominadas palabras clave, que tienen en C un significado estándar y por tanto no pueden ser utilizadas como identificadores definidos por el programador.
Las palabras clave están en minúsculas.

La siguiente es una lista de las palabras claves del ANSI C:

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

CONSTANTES

Las variables pueden cambiar de valor a lo largo de la ejecución de un programa, o bien en ejecuciones distintas de un mismo programa. Además de variables, un programa utiliza también constantes, es decir, valores que siempre son los mismos. Un ejemplo típico es el número pi, que vale 3.141592654. Este valor, con más o menos cifras significativas, puede aparecer muchas veces en las sentencias de un programa. En C existen distintos tipos de constantes:

Constantes numéricas: Son valores numéricos, enteros o de punto flotante. Se permiten también constantes *octales* (números enteros en base 8) y *hexadecimales* (base 16).

Constantes carácter: Cualquier carácter individual encerrado entre apóstrofos (tal como 'a', 'Y', ')', '+', etc.) es considerado por C como una constante carácter, o en realidad como un *número entero pequeño* (entre 0 y 255, o entre -128 y 127, según los sistemas).

Existe un código, llamado código ASCII, que establece una equivalencia entre cada carácter y un valor numérico correspondiente.

Cadenas de caracteres: Un conjunto de caracteres alfanuméricos encerrados entre comillas es también un tipo de constante del lenguaje C, como por ejemplo: "espacio", "Esto es una cadena de caracteres", etc.

Constantes simbólicas: Las constantes simbólicas tienen un nombre (identificador) y en esto se parecen a las variables. Sin embargo, no pueden cambiar de valor a lo largo de la ejecución del programa. En C se pueden definir mediante el preprocesador o por medio de la palabra clave const.

Operadores

Los operadores son signos especiales que indican determinadas operaciones a realizar con las variables y/o constantes sobre las que actúan en el programa.

OPERADORES ARITMÉTICOS

Existen dos tipos de operadores aritméticos:

Los binarios:

+ Suma

- Resta
* Multiplicación
/ División
% Módulo (resto)

y los unarios:

++ Incremento (suma 1)
- - Decremento (resta 1)
- Cambio de signo

Su sintaxis es:

binarios:

<variable1><operador><variable2>

(por ejemplo a+b)

unarios:

<variable><operador> y al revés, <operador><variable>.

(por ejemplo -a o a++)

OPERADORES DE ASIGNACIÓN

La mayoría de los operadores aritméticos binarios explicados en el capítulo anterior tienen su correspondiente operador de asignación:

= Asignación simple

+= Suma

-= Resta

*= Multiplicación

/= División

%= Módulo (resto)

Con estos operadores se pueden escribir, de forma más breve, expresiones del tipo:

n=n+3 se puede escribir n+=3

k=k*(x-2) lo podemos sustituir por k*=x-2

OPERADORES RELACIONALES

Los operadores relacionales se utilizan para comparar el contenido de dos variables.

En C existen seis operadores relacionales básicos:

> Mayor que

< Menor que

>= Mayor o igual que

<= Menor o igual que

== Igual que

!= Distinto que

El resultado que devuelven estos operadores es 1 para Verdadero y 0 para Falso, sin embargo, C considera como Falso el valor 0 y Verdadero cualquier valor distinto de 0.

OPERADORES LÓGICOS

Los operadores lógicos básicos son tres:

&& AND
|| OR
! NOT (El valor contrario)

Estos operadores actúan sobre expresiones lógicas. Permiten unir expresiones lógicas simples formando otras más complejas.

OPERADORES A NIVEL DE BITS

Los operadores a nivel de bits permiten operar efectuar operaciones bit a bit entre dos variables o constantes.

& AND bit a bit
| OR bit a bit
! NOT bit a bit
<< x Desplazamiento a la izquierda en x bits
>> x Desplazamiento a la derecha en x bits

El desplazamiento a la izquierda en x bits corresponde a multiplicar el número por 2^x bits. El desplazamiento a la derecha corresponde a dividir el número por 2^x bits.

Jerarquía de los operadores

Será importante tener en cuenta la precedencia de los operadores a la hora de trabajar con ellos:

( ) Mayor precedencia
++, - -
*, /, %
+, -
==, != Menor precendencia

Las operaciones con mayor precedencia se realizan antes que las de menor precedencia.

Si en una operación encontramos signos del mismo nivel de precedencia, dicha operación se realiza de izquierda a derecha.

Tipos de datos

Los tipos de datos básicos definidos por C son caracteres, números enteros y números en coma flotante. Los caracteres son representados por char, los enteros por short, int, long y los números en coma flotante por float y double. Los tipos básicos disponibles y su tamaño son:

TIPOS	RANGO	TAMAÑO	DESCRIPCIÓN
char	-128 a 127	1	Para una letra o un dígito.
unsigned char	0 a 255	1	Letra o número positivo.
int	-32.768 a 32.767	2	Para números enteros.
unsigned int	0 a 65.535	2	Para números enteros.
long int	±2.147.483.647	4	Para números enteros
unsigned long int	0 a 4.294.967.295	4	Para números enteros
float	3.4E-38 decimales(6)	6	Para números con decimales
double	1.7E-308 decimales(10)	8	Para números con decimales
long double	3.4E-4932 decimales(10)	10	Para números con decimales

Ámbito de las Variables

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(). 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.

TIPOS DEFINIDOS POR EL USUARIO: C permite crear explícitamente nuevos nombres para tipos de datos. Realmente no se crea un nuevo tipo de dato, sino que se define un nuevo nombre para un tipo existente. Esto ayuda a hacer más transportables los programas con dependencias con las máquinas, solo habrá que cambiar las sentencias typedef cuando se compile en un nuevo entorno.

Sintaxis:

```
typedef tipo nuevo_nombre;
nuevo_nombre nombre_variable[=valor];
```

Por ejemplo:

```
#include<stdio.h>

void func1(void);
int numero;
extern int numero1;

void main(void)
{
       int numero3=10;
       static int numero4=20;
       register int numero5=30;

       clrscr();
       printf(“%d %d %d %d %d\n”,numero,numero1,
numero2,numero3,numero4,numero5);
       func1();
       getch();
}

void func1(void)
{
       printf(“Las variables globales, que aquí se conocen”);
       printf(“%d %d”,numero,numero1);
}
```

CONSTANTES: Se refieren a los valores fijos que no pueden ser modificados por el programa. Pueden ser de cualquier tipo de datos básicos. Las constantes de carácter van encerradas en comillas

simples. Las constantes enteras se especifican con números sin parte decimal y las de coma flotante con su parte entera separada por un punto de su parte decimal.

Sintaxis:

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

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.

Sintaxis:

```
#define IDENTIFICADOR valor_numerico
#define IDENTIFICADOR “cadena”
```

Ejemplo:

```
#include<stdio.h>

#define PI 3.141592654
#define TEXTO "Esto es una prueba"
const int peseta=1;

void main(void)
```

```
{
      clrscr();
      printf("El valor del pi es %f ",PI);
      printf("\n%s",TEXTO);
      printf(“Ejemplo de constantes y defines”);
      getch();
}
```

El preprocesador de C

DIRECTIVAS DEL PREPROCESADOR

El preprocesamiento es el primer paso en la étapa de compilación de un programa; esta propiedad es única del compilador de C.

El preprocesador tiene más o menos su propio lenguaje el cual puede ser una herrramienta muy poderosa para el programador. Todas las directivas de preprocesador o comandos inician con un #.

Las ventajas que tiene usar el preprocesador son:

- -los programas son más fáciles de desarrollar,
- -son más fáciles de leer,
- -son más fáciles de modificar
- -y el código de C es más transportable entre diferentes arquitecturas de máquinas.

#DEFINE

La directiva #define se usa para definir constantes o cualquier sustitución de macro. Su formato es el siguiente:

#define <nombre de macro> <nombre de reemplazo>

EJEMPLO:

```
#define FALSO 0
#define VERDADERO !FALSO
```

La directiva #define tiene otra poderosa característica: el nombre de macro puede tener argumentos. Cada vez que el compilador encuentra el nombre de macro, los argumentos reales encontrados en el programa reemplazan los argumentos asociados con el nombre de la macro.

EJEMPLO:

```
#define MIN(a,b) (a < b) ? a : b
main()
{
	int x=10, y=20;
	printf("EL minimo es %d\n", MIN(x,y) );
}
```

Cuando se compila este programa, el compilador sustiuirá la expresión definida por MIN(x,y), excepto que x e y serán usados como los operandos. Así después de que el compilador hace la sustitución, la sentencia printf será ésta:

printf("El minimo es %d\n", (x < y) ? x : y);

Como se puede observar donde se coloque MIN, el texto será reemplazado por la definición apropiada. Por lo tanto, si en el código se hubiera puesto algo como:

```
x = MIN(q+r,s+t);
```

después del preprocesamiento, el código podría verse de la siguiente forma:

```
x = ( q+r < s+t ) ? q+r : s+t;
```

El uso de la sustitución de macros en el lugar de las funciones reales tiene un beneficio importante: incrementa la velocidad del código porque no se penaliza con una llamada de función. Sin embargo, se paga este incremento de velocidad con un incremento en el tamaño del programa porque se duplica el código.

#UNDEF

Se usa #undef para quitar una definición de nombre de macro que se haya definido previamente. El formato general es:

```
#undef <nombre de macro>
```

El uso principal de #undef es permitir localizar los nombres de macros sólo en las secciones de código que los necesiten.

#INCLUDE

La directiva del preprocesador #include instruye al compilador para incluir otro archivo fuente que esta dado con esta directiva y de esta forma compilar otro archivo fuente. El archivo fuente que se leerá se debe encerrar entre comillas dobles o paréntesis de ángulo.

```
#include <archivo>
#include "archivo"
```

Cuando se indica <archivo> se le dice al compilador que busque donde estan los archivos incluidos o ``include'' del sistema.
Si se usa la forma "archivo" es buscado en el directorio actual, es decir, donde el programa esta siendo ejecutado.
Los archivos incluidos usualmente contienen los prototipos de las funciones y las declaraciones de los archivos cabecera (header files) y no tienen código de C (algoritmos).

#IF INCLUSIÓN CONDICIONAL

La directiva #if evalúa una expresión constante entera. Siempre se debe terminar con #endif para delimitar el fin de esta sentencia.
Se pueden así mismo evaluar otro código en caso se cumpla otra condición, o bien, cuando no se cumple ninguna usando #elif o #else respectivamente.

EJEMPLO:

```
#define MEX 0
#define EUA 1
#define FRAN 2
#define PAIS_ACTIVO MEX
#if PAIS_ACTIVO == MEX
char moneda[]="pesos";
#elif PAIS_ACTIVO == EUA
char moneda[]="dólar";
#else
char moneda[]="franco";
#endif
```

Otro método de compilación condicional usa las directivas #ifdef (si definido) y #ifndef (si no definido).

#IFDEF

El formato general de #ifdef es:

#ifdef <nombre de macro>
<secuencia de sentencias>
#endif

Si el nombre de macro ha sido definido en una sentencia #define, se compilará la secuencia de sentencias entre el #ifdef y #endif.

El formato general de #ifdef es:

#ifndef <nombre de macro>

```
<secuencia de sentecias>
#endif
```

Las directivas anteriores son útiles para revisar si las macros están definidas -- tal vez por módulos diferentes o archivos de cabecera.
Por ejemplo, para poner el tamaño de un entero para un programa portable entre Borland C de DOS y un sistema operativo con UNIX, sabiendo que Borland C usa enteros de 16 bits y UNIX enteros de 32 bits, entonces si se quiere compilar para Borland C se puede definir una macro Borland C, la cual será usada de la siguiente forma:

```
#ifdef  Borland C
#define INT_SIZE 16
#else
#define INT_SIZE 32
#endif
```

Otras directivas del preprocesador

#ERROR

Esta función fuerza al compilador a parar la compilación cuando la encuentra. Se usa principalmente para depuración.

EJEMPLO:

```
#ifdef OS_MSDOS
    #include <msdos.h>
#elifdef OS_UNIX
    #include "default.h"
#else
    #error Sistema Operativo incorrecto
#endif
```

La directiva #line número ``cadena'' informa al preprocesador cual es el número siguiente de la línea de entrada. Cadena es opcional y nombra la siguiente línea de entrada. Esto es usado frecuentemente cuando son traducidos otros lenguajes a C. Por ejemplo, los mensajes de error producidos por el compilador de C pueden referenciar el nombre del archivo y la línea de la fuente original en vez de los archivos intermedios de C.

EJEMPLO: El siguiente programa especifica que el contador de línea empezará con 100.

```
#line 100 "test.c"    /* inicializa el contador de linea y
nombre de archivo */
main()                /* linea 100 */
{
    printf("%d\n",__LINE__); /* macro predefinida, linea 102
*/
    printf("%s\n",__FILE__); /* macro predefinida para el
nombre */
}
```

Funciones

Las funciones son la base de la programación estructurada. Una función tiene la siguiente sintaxis:

```
tipo_r nombre_de_función(tipo_arg1 nombre_arg1, tipo_arg2
nombre_arg2, …etc)
{
      /* Declaración de variables locales*/
      /* Cuerpo de la función */
      return algun_valor;  /* Se devuelve algún valor del
tipo tipo_r */
}
```

donde tipo_r corresponde al tipo de valor que devolverá la función, tipo_arg1, tipo_arg2, etc el tipo de variables de los argumentos pasados a la función y nombre_arg1, nombre_arg2, etc., el nombre con los que conoceremos a estos argumento. En caso de no querer recibir argumentos o devolver algún valor, podemos usar reemplazar tipo_r o los argumentos por la palabra clave void.

A lo anteriormente visto le llamamos *definición* de la función. Sin embargo, es necesario también *declarar* la función, lo que permitirá al compilador chequear si los tipos introducidos a la y devueltos por la función corresponden con lo que deberían ser. La declaración de una función se hace usando la siguiente sintaxis:

```
tipo_r nombre_de_función(tipo_arg1 nombre_arg1, tipo_arg2
nombre_arg2, …etc);
```

Es decir, igual que la primera línea de la definición, pero poniendo al final un punto y coma. La declaración de la función se suele hacer bajo el encabezado del programa (sobre el main) o en un archivo aparte llamado archivo de cabecera, de extensión .H, al cual se le acostumbra poner el mismo nombre del .c que contiene la definición de la función. Esto permite que el código sea reutilizado después por otros programas con sólo incluir el archivo de cabecera donde está la función.

Un ejemplo:

```
#include <stdio.h>
int suma(int primer_sumando, int segundo sumando); /*
Declaración de la función*/

void main(void)
{
      int a=4;
      int resultado;
      resultado=suma(a,8);
      printf(“El resultado de la suma es %i”,resultado);
}

/* Definición del la función
int suma(int primer_sumando, int segundo sumando)
{
    int result;
```

```
    result= primer_sumando+segundo_sumando;
    return result;
}
```

En este caso, suma recibe como argumento la variable a y la constante 8, ambos de tipo entero. El resultado devuelto por suma (12) es guardado en la variable resultado, para posteriormente imprimirla en pantalla.

E/S por Consola

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.

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));

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

CONSTANTES DE CARÁCTER	
CONSTANTE	DESCRIPCION
\n	Salto de línea
\f	Salto de página
\r	Retorno de carro
\t	Tabulación
\b	Retroceso
\’	Comilla simple
\”	Comillas
\\	Barra invertida
\?	Interrogación

%h	Short
%%	signo %

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 varía. 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.

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);
```

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);
```

En muchas ocasiones se combinaran varias funciones para pedir datos y eso puede traer problemas para el buffer de teclado, es decir que asigne valores que no queramos a nuestras variables. Para evitar esto hay dos funciones que se utilizan para limpiar el buffer de teclado.

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

OTRAS FUNCIONES E/S: El archivo de cabecera de todas estas funciones es STDIO.H. Todas estas funciones van a ser utilizadas para leer caracteres o cadenas de caracteres. Todas ellas tienen asociadas una función de salida por consola.

Funciones	DESCRIPCIÓN
var_char=getchar();	Lee un carácter de teclado, espera un salto de carro.
var_char=getche();	Lee un carácter con eco, no espera salto de carro.
var_char=getch();	Lee un carácter sin eco, no espera salto de carro.
gets(var_cadena);	Lee una cadena del teclado.
putchar(var_char);	Muestra un carácter en pantalla.
puts(variables);	Muestra una cadena en pantalla.

Sentencias Condicionales

Este tipo de sentencias permiten variar el flujo del programa en base a unas determinadas condiciones.

Existen varias estructuras diferentes:

ESTRUCTURA IF...ELSE

Sintaxis:

```
if (condición) sentencia;
```

La sentencia solo se ejecuta si se cumple la condición. En caso contrario el programa sigue su curso sin ejecutar la sentencia.

Otro formato:

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

Si se cumple la condición ejecutará la sentencia1, si no ejecutará la sentencia2. En cualquier caso, el programa continuará a partir de la sentencia2.

Ejemplo:

```
#include <stdio.h>

main() /* Simula una clave de acceso */
{
   int usuario,clave=18276;
   printf("Introduce tu clave: ");
   scanf("%d",&usuario);
   if(usuario==clave)
      printf("Acceso permitido");
   else
      printf("Acceso denegado");
}
```

Otro formato:

```
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:

```
#include <stdio.h>

main() /* Escribe bebé, niño o adulto */
{
   int edad;
   printf("Introduce tu edad: ");
   scanf("%d",&edad);
   if (edad<1)
  {
      printf("Lo siento, te has equivocado o");
```

```
      printf(“ a lo mejor no has nacido todavía”);
  }
  else if (edad<3) printf("Eres un bebé");
    else if (edad<13) printf("Eres un niño");
       else printf("Eres adulto ;-) ");
}
```

ESTRUCTURA SWITCH

Esta estructura se suele utilizar en los menús, de manera que según la opción seleccionada se ejecuten una serie de sentencias.

Su sintaxis es:

```
switch (variable){
      case contenido_variable1:
            sentencias;
            break;
      case contenido_variable2:
            sentencias;
            break;
      default:
            sentencias;
}
```

Cada case puede incluir una o más sentencias sin necesidad de ir entre llaves, ya que se ejecutan todas hasta que se encuentra la

sentencia BREAK. La variable evaluada sólo puede ser de tipo entero o caracter. La sentencia `default` ejecutará las sentencias que incluya, en caso de que la opción escogida no exista.

Ejemplo:

```
#include <stdio.h>

main() /* Escribe el día de la semana */
{
   int dia;
   printf("Introduce el día: ");
   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;
   }
}
```

Sentencias de Bucle

Un bucle o loop es simplemente la repetición de bloque de instrucciones, de acuerdo a cierta condición. A continuación presentamos las instrucciones que nos permiten implementar los bucles en C.

WHILE: Ejecuta repetidamente el mismo bloque de código hasta que se cumpla una condición de terminación.

Sintaxis:

```
while(condición){
    /* Código a repetir */
}
```

DO-WHILE: Es lo mismo que en el caso anterior pero aquí como mínimo siempre se ejecutara el cuerpo una vez, en el caso anterior es posible que no se ejecute ni una sola vez.

Sintaxis:

```
do{
        /*  Código a repetir */
} while(condición);
```

Ejemplo:

Este programa va sumando números y el resultado lo suma a otro, así hasta 100.000.

```
#include <stdio.h>

void main(void)
```

```
{
   int n1=0,n2=1,n3;
   printf("Serie de Fibonacci: %d %d\n",n1,n2);
   do{
      n3=n1+n2;
      n1=n2+n3;
      n2=n1+n3;
      printf("%d %d %d\n",n3,n1,n2);
   }while(n1<1000000 && n2<1000000 && n3<1000000);
}
```

FOR: Realiza las mismas operaciones que en los casos anteriores pero la sintaxis es una forma compacta. Normalmente la condición para terminar es de tipo numérico. La iteración puede ser cualquier expresión matemática valida. Si de los 3 términos que necesita no se pone ninguno se convierte en un bucle infinito.

Sintaxis

```
for (inicio;condición;iteración)
      sentencia1;
```

Ejemplo: Este programa muestra números del 1 al 100. Utilizando un bucle de tipo FOR.

```
#include<stdio.h>

   void main(void)
   {
```

```
        int n1=0;
        for (n1=1;n1<=100;n1++)
             printf("%d\n",n1);
        getch();
}
```

CONTINUES, BREAK, EXIT();

Estas sentencias permite cambiar la normal ejecución de un bucle o programa.

Continue hace que el programa pase a la próxima iteración, saltándose las líneas que quedan hasta llegar al fin del programa.

Ejemplo:

Este programa imprimirá los valores de i de 0 al 9 y luego el texto “Fin del programa”.

```
#include<stdio.h>

   void main(void)
   {
      int i=0;
      while(i<10)
      {
            if(i==5)
                  continue;
            printf(“El valor de i es %i \n”,i);
      }
      printf(“\nFin del programa”);
   }
```

Break hace que el programa termine en ese punto el bucle más interior. Si este bucle está anidado dentro de otro bucle, el programa continuará en la ejecución del bucle más externo.

Ejemplo:

Este programa imprimirá los valores de i de 0 al 4 y luego el texto “Fin del programa”.

```
#include<stdio.h>

   void main(void)
   {
      int i=0;
      while(i<10)
      {
            if(i==5)
                  break;
            printf(“El valor de i es %i \n”,i);
      }
      printf(“\nFin del programa”);
   }
```

Exit() no es una sentencia de C propiamente tal, sino una función que se encuentra en la librería estándar de C (<stdlib.h>). Lo que hace esta función es terminar abruptamente el programa en el punto donde la llamemos. Por lo general se usa la sentencia exit(0) cuando salimos del programa normalmente y exit(1) si salimos debido a un error.

Ejemplo:

Este programa imprimirá los valores de i de 0 al 4 y luego saldrá del programa.

```
#include<stdio.h>

   void main(void)
   {
      int i=0;
      while(i<10)
      {
            if(i==5)
                  exit(0);
            printf("El valor de i es %i \n",i);
      }
      printf("\nFin del programa");
   }
```

ARRAYS Y CADENAS

Un array es una colección de variables del mismo tipo que se referencian por un nombre en común. A un elemento específico de un array se accede mediante un índice. En C todos los array constan de posiciones de memoria contiguas. La dirección mas baja corresponde al primer elemento y la dirección más alta al útimo elemento. Los arrays pueden tener de una a varias dimensiones. El array más común

en C es la cadena el cual es un array de caracteres terminado por un caracter nulo (\0).

UNIDIMENSIONALES

A éstos se les llaman comúnmente vectores. Todos los arrays tienen al 0 como índice de su primer elemento. Hay que tener muy presente de no rebasar el último índice. Los arrays deben declararse explícitamente àra que el compilador pueda reservar espacio en memoria para ellos. La cantidad de memoria requerida para guardar un array está directamente relacionada con su tipo y su tamaño.

SINTAXIS:

```
tipo nombre_array [nº elementos];
tipo nombre_array [nº elementos]={valor1,valor2,valorN};
tipo nombre_array[]={valor1,valor2,valorN};
```

De esta forma si deseo declarar: char p[10];
Estoy declarando 10 elementos desde p[0] a p[9]].
Para un array unidimensional el tamaño total en bytes es:

Total Bytes=sizeof(tipo) *tamaño de array

O sea si p[10] es del tipo int, el cual se almacena en 2 Bytes, el porte de p es 20 Bytes.

INICIALIZACIÓN DE UN ELEMENTO:

```
nombre_array[indice]=valor;
```

UTILIZACIÓN DE ELEMENTOS:

```
nombre_array[indice];
```

EJEMPLO: Reserva 100 elementos enteros, los inicializa todos y muestra el 5º elemento.

```
#include <stdio.h>

void main(void)
{
     int x[100];
     int cont;
     clrscr();
     for(cont=0;cont<100;cont++)
             x[cont]=cont;
     printf(“%d”,x[4]);
     getch();
}
```

Esto no lo controla el compilador, y tendremos que ser nosotros los que insertemos este caracter al final de la cadena.

Por tanto, en un vector de 10 elementos de tipo char podremos rellenar un máximo de 9, es decir, hasta vector[8]. Si sólo rellenamos los 5 primeros, hasta vector[4], debemos asignar el caracter nulo a vector[5]. Es muy sencillo: vector[5]='\0'.

Ahora veremos un ejemplo de como se rellena un vector de tipo char.

El uso más común de los arrays unidimensionales es con mucho las **cadenas**. Las cadenas son, como su nombre lo dicee, cadenas de caracteres y tienen la particularidad de que se debe indicar en que elemento se encuentra el fin de la cadena, lo cual se realiza colocando en este el caracter nulo (\0). Esto no lo controla el compilador, y tendremos que ser nosotros los que insertemos este caracter adicional al final de la cadena. Por tanto no se debe olvidar que al declarar un arrays de caracteres, agregar siempre un elemento más y si se da el caso de tener que rellenar uno a uno los elementos de la cadena, hay que asignar específicamente el caracter nulo al último elemento.

SINTAXIS:

```
char nombre[tamaño];
char nombre[tamaño]=”cadena“;
char nombre[]=”cadena”;
```

Aunque C no define un tipo de datos de cadenas, permite disponer de constantes de cadena: Una constante de cadena es una lista de caracteres encerrada entre dobles comillas.

Ejemplo:

```
Cadena=”Hola Mundo”;
```

En este caso no es necesario añadir explícitamente el carácter nulo al final de las constantes de la cadena, pues el compilador de C lo hace automáticamente.

BIDIMENSIONALES:

C admite arrays multidimensionales, la forma más sencilla son los arrays bidimensionales. Un array bidimensional es esencialmente un array de arrays unidimensionales. Los array bidimensionales se almacenan en matrices fila-columna, en las que el primer índice indica la fila y el segundo indica la columna. Esto significa que el índice más a la derecha cambia más rápido que el de más a la izquierda cuando accedemos a los elementos.

SINTAXIS:

```
tipo nombre_array[fila][columna];
tipo nomb_array[fil][col]={{v1,v2,vN},{v1,v2,vN},{vN}};
tipo nomb_array[][]={{v1,v2,vN},{v1,v2,vN},{vN}};
```

INICIALIZACIÓN DE UN ELEMENTO:

```
nombre_array[indice_fila][indice_columna]=valor;
```

UTILIZACIÓN DE UN ELEMENTO:

```
nombre_array[indice_fila][indice_columna];
```

Para conocer el tamaño que tiene un array bidimensional tenemos que multiplicar las filas por las columnas por el número de bytes que ocupa en memoria el tipo del array. Es exactamente igual que con los array unidimensionales lo único que se añade son las columnas.

filas * columnas * bytes_del_tipo

A diferencia de la gran mayoría de los lenguajes de computadoras, que utilizan comas para separar las dimensiones del arry, C coloca cada dimensión en suy propio conjunto de corchetes.

Ejemplo: Forma de acceder al elemento 1,2 del array p:

p[1][2]

Un uso muy común de los arrays bidimensionales es crear un array de cadenas. En este tipo de array el número de filas representa el número de cadenas y el de las columnas representa la longitud de cada una de esas cadenas.

EJEMPLO: Introduce 10 cadenas y luego las muestra.

```
#include <stdio.h>

void main(void)
{
     char texto[10][80];
     int indice;
     clrscr();

     for(indice=0;indice<10;indice++)
     {
         printf("%2.2d:",indice+1);
         gets(texto[indice]);
     }

     printf("Pulsa tecla");
```

```
    getch();
    clrscr();

    for(indice=0;indice<10;indice++)
        printf("%s\n",texto[indice]);

    getch();
}
```

LLAMADAS A FUNCIONES CON ARRAYS

Los arrays sólo pueden ser enviados a una función por referencia. Para ello deberemos enviar la dirección de memoria del primer elemento del array. Por tanto, el argumento de la función deberá ser un puntero[1].

Ejemplo: Llamada de un array a una función

```
#include <stdio.h>

void visualizar(int []); /* prototipo */

void main(void) /* rellenamos y visualizamos */
{
      int array[25],i;
      for (i=0;i<25;i++)
      {
            printf("Elemento nº %d",i+1);
            scanf("%d",&array[i]);
      }
      visualizar(&array[0]);
```

[1] Ver Capítulo de Punteros

```
}

void visualizar(int array[]) /* desarrollo */
{
      int i;
      for (i=0;i<25;i++) printf("%d",array[i]);
}
```

ESTRUCTURAS, UNIONES, ENUMERACIONES y TYPEDEF

C proporciona 5 formas diferentes de creación de tipos de datos propios. El primero de ellos es la agrupación de variables bajo un mismo nombre a la cual se le llama *estructura*. La segunda clase de tipos definidos es la *unión*, que permite que la misma parte de memoria sea definida como dos o más tipos diferentes de variables. Un tercer tipo de datos definibles es la *enumeración* que es una lista constantes entera con nombre. Otra muy utilizada son las creadas por el comando *typedef* la cual define un nuevo nombre para un tipo ya existente.

ESTRUCTURAS

Una estructura es una colección de variables que se referencia bajo un único nombre, proporcionando un medio eficaz de mantener junta una información relacionada. Las variables que componen la estructura se llaman miembros de la estructura y está relacionado lógicamente con los otros. Otra característica es el ahorro de memoria y evitar declarar

variables que técnicamente realizan las mismas funciones. Su utilización más habitual es para la programación de bases de datos, ya que están especialmente indicadas para el trabajo con registros o fichas.

SINTAXIS:

```
struct nombre{
     var1;
     var2;
     varN;
};
struct nombre etiqueta1,etiquetaN;
```

La palabra clave *struct* indica al compilador que se está declarando una estructura.
Los miembros individuales de la estructura se referencian utilizando la etiqueta de la estructura, el operador punto(**.**) y la variable a la que se hace referencia.
Los miembros de la estructura deben ser inicializados fuera de ella, si se hace en el interior da error de compilación.

etiqueta.variable;

EJEMPLO: Ejemplo utilización de Estructuras

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

```
void main (void)
{
     int opcion=0;
     struct ficha{
          char nombre[40];
          char apellido[50];
          unsigned edad;
     } emplead,usuario;

     do
     {
          clrscr();
          gotoxy(2,4);printf("1.empleados");
          gotoxy(2,5);printf("2.usuarios");
          gotoxy(2,6);printf("0.visualizar");
          gotoxy(2,7);scanf("%d",&opcion);

          if (opcion==0)
               break;
          if(opcion==1)
          {
               gotoxy(2,10);printf("Nombre: ");
               gets(emplead.nombre);
               gotoxy(2,11);printf("Apellido: ");
               gets(emplead.apellido);
               gotoxy(2,12);printf("Edad: ");
               scanf("%d",&emplead.edad);
          }
          else
          {
               gotoxy(2,10);printf("Nombre: ");
               gets(usuario.nombre);
```

```
                gotoxy(2,11);printf("Apellido: ");
                gets(usuario.apellido);
                gotoxy(2,12);printf("Edad: ");
                scanf("%d",&usuario.edad);
           }
     }while(opcion!=0);

     gotoxy(2,18);
     printf("%s %s\n",emplead.nombre,emplead.apellido);
     gotoxy(2,19);
     printf("%u años",emplead.edad);
     gotoxy(30,18);
     printf("%s %s\n",usuario.nombre,usuario.apellido);
     gotoxy(30,19);
     printf("%u años",usuario.edad);
     getch();
}
```

Una de las operaciones más comunes es asignar el contenido de una estructura a otra estructura del mismo tipo mediante una única sentencia de asignación como si fuera una variable; es decir, no es necesario asignar los valores de cada miembro por separado.

EJEMPLO:

```
#include <stdio.h>

void main(void)
{
      struct{
```

```
            int a;
            int b;
      }X,Y;
      X.a=10;
      Y=X;  /* Asigna una estructura a la otra */
      printf("%d",Y.a);
}
```

Otra operación muy común son los arrays de estructuras. Para declarar un array de estructuras, se debe definir primero la estructura y luego declarar la etiqueta indicando el número de elementos.

SINTAXIS:

```
struct nombre{
     var1;
     var2;
     varN;
}etiqueta[nº elementos];
```

Un miembro de una estructura puede ser del tipo simple o complejo. Un miembro simple es de cualquiera de los tipos ya mencionados: enteros, caracteres, doubles, etc. En cambio, un miembro complejo puede ser un array de caracteres, un array unidimensional o multidimensional, de datos simples o de estructuras. Si se da el último caso, a la estructura se le denominará estructura anidada.

Ejemplo: miembro de estructura array.

```
struct X{
```

```
  int a[10][10]; /* Array de 10x10 int */
  float b;
}Y;
```

Ejemplo: estructura anidada.

```
struct Ejem{
        struct x;
        float c;
}prueba;
```

Acá se ha definido una estructura Ejem con 2 miembros. el primero es la estructura de array, que contiene a y b; y el segundo es c.

Como ejemplo de array de estructuras deberemos crear una lista de correos utilizando estructuras para guardar la información. Las operaciones que deben hacerse son añadir datos, borrar datos y realizar listados. El número de registros que va a mantener es de 100. Se deberá controlar que no supere el límite máximo.

UNIONES

Una unión es una posición de memoria que es compartida por dos o más variables diferentes, generalmente de distinto tipo, en distintos momentos. La declaración de la unión es similar a la de la estructura. Cuando se crea una etiqueta de la unión, el compilador reserva memoria para el mayor miembro de la unión. El uso de una unión puede ayudar a la creación de código independiente de la máquina y

ahorra memoria en la declaración de variables. Para referencia a un elemento de la unión se hace igual que si fuera una estructura.

SINTAXIS:

```
union nombre{
     var1;
     var2;
     varN
};
.
.
.
union nombre etiqueta1,etiquetaN;
```

ENUMERACIONES

Es un conjunto de constantes enteras con nombre que especifica todos los valores válidos que una variable de ese tipo puede tener. La definición es muy parecida a las estructuras, la palabra clave enum señala el comienzo de un tipo enumerado. El valor del primer elemento de la enumeración es 0 aunque se puede especificar el valor de uno o más símbolos utilizando una asignación.

SINTAXIS:

```
enum nombre{lista_de_enumeraciones}etiqueta1,etiquetaN;
enum nombre{lista_de_enumeraciones};
.
.
.
enum nombre etiqueta1,etiquetaN;
```

EJEMPLO:

```
#include<stdio.h>

void main(void)
{
     enum moneda{dolar,penique=100,medio=50,cuarto=25};
     enum moneda dinero;

     clrscr();
     printf("Valor de la moneda: ");
     scanf("%d",&dinero);

     switch(dinero)
     {
         case dolar:
              printf("Con ese valor es un Dolar");
              break;
         case penique:
              printf("Con ese valor es un Penique");
              break;
         case medio:
              printf("Con ese valor es Medio-Dolar");
              break;
         case cuarto:
```

```
            printf("Con ese valor es un Cuarto");
            break;
        default:
            printf("Moneda Inexistente");
    }
    getch();
}
```

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.

EJEMPLO: Declaración de estructuras mediante typedef

```
#define MAX_NOM 30
#define MAX_ALUMNOS 400

struct s_alumno {
      char nombre[MAX_NOM];
      short edad;
};

typedef struct s_alumno ALUMNO;
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. */
```

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.

PUNTEROS

Este es uno de los temas que más le suele costar a la gente al aprender C. Los punteros son una de las más potentes características de C, pero a la vez uno de sus mayores peligros. Los punteros nos permiten acceder directamente a cualquier parte de la memoria. Esto da a los programas en C una gran potencia, pero a la vez son una fuente ilimitada de errores. Un error usando un puntero puede

bloquear el sistema (si usamos MS-DOS o Windows, en Linux no sucede esto) y a veces puede ser difícil detectarlo.
Otros lenguajes no utilizan punteros para evitar estos problemas, pero a la vez pierden en parte el control que estos otorgan.
A pesar de todo, no es difícil entenderlos, pero es necesario pues casi todos los programas C usan punteros.

Un puntero es una variable que contiene una dirección de memoria. Esa dirección es la posición de un objeto (normalmente una variable) en memoria. Si una variable va a contener un puntero, entonces tiene que declararse como tal. Cuando se declara un puntero que no apunte a ninguna posición valida ha de ser asignado a un valor nulo (un cero).

SINTAXIS:

```
tipo *nombre;
```

Al declarar una variable, se le dice al procesador que reserve una parte de la memoria para almacenarla. Cada vez que se ejecuta el programa, la variable se almacenará en un sitio diferente, pues depende de la memoria disponible y otros factores. Puede que se almacene en el mismo sitio, pero generalmente esto no ocurre. Dependiendo del tipo de variable que declaremos el procesador reservará más o menos memoria. Por ejemplo si se declara un char, el procesador reserva 1 byte (8 bits) de memoria. Cuando finaliza el programa todo el espacio reservado se libera.

Vamos a ver un ejemplo: Declaramos la variable 'a' y obtenemos su valor y dirección

OPERADORES:

& y *

Son dos operadores monarios (necesita sólo un operando). El primero, &, devuelve la dirección de memoria de su operando. El segundo, *, es el complemento de & y devuelve el valor de la variable que se encuentra en la posición de memoria almacenada en su operando.

Ejemplo:

```
m= &cuenta;
q=*m;
```

Esto simplemente pone en m la dirección de memoria de la variable cuenta, la dirección no tiene nada que ver con el valor de cuenta. En la segunda línea pone el valor de cuenta en q.

Para comprender mejor este ejemplo le pondremos valores: Sea cuenta igual 100 y se encuentra almacenada en la posición de memoria 2000. Así & devuelve la dirección de memoria de cuenta, 2000, y lo almacena en m.. El * devuelve el valor de la variable localizada en la dirección dada por el valor de m. Por lo tanto q es igual al valor almacenado en la posición de memoria 2000, por lo tanto q=100.

EJEMPLO:

```
#include <stdio.h>

void main(void)
{
     int *p;
     int valor;
     clrscr();
     printf("Introducir valor: ");
     scanf("%d",&valor);
     p=&valor;
     printf("Direccion de memoria de valor es: %p\n",p);
     printf("El valor de la variable es: %d",*p);
     getch();
}
```

ASIGNACIONES:

Como en el caso de cualquier variable, un puntero puede utilizarse a la derecha de una declaración de asignación para asignar su valor a otro puntero.

Ejemplo:

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

```
    int *p1, *p2;

    p1=&x;
    p2=p1;

    printf("%p",p2);
}
```

Tanto *p1* como *p2* apuntan ahora a *x.* Se imprime la dirección de *x* usando el modificador de formato de *printf() %p*, que hace que *printf()* muestre una dirección en el formato utilizado por el procesador.

ARITMÉTICA:

Existen sólo dos operaciones aritméticas que se pueden usar como punteros, la suma y la resta. Para entender que ocurre en la aritmética de punteros, considere p1 un puntero a float (4 bytes) con valor actual de 2000. Ahora al pasar por la expresión:

p++

P1 ahora contiene 2004, no 2001, pues cuando p1 se incrementa, apunta al siguiente al siguiente entero. Generalizando tenemos que:

- ✓ Cuando se incrementa un puntero, apunta a la posición de memoria del siguiente elemento de su tipo base (int, float, double, etc.).
- ✓ Cada vez que se decrementa, apunta a la posición del elemento anterior.

- ✓ Cuando se aplican punteros a caracteres, se aplica una aritmética normal, pues los caracteres ocupan 1 byte.

Pero las operaciones aritméticas no sólo están limitadas a los operadores de incremento y decremento, también se pueden sumar o restar *enteros* (nótese que sólo enteros) a punteros. La expresión:

p1=p1 + 3;

hace que p1 apunte al tercer elemento del tipo p1, que está más allá del elemento que apunta p1. Esto es si p1 es entero y p1 apunta a la dirección 2000, p1 apuntaría después de esta sentencia a la dirección 2006, en caso que p1 sea float, p1 apuntaría a la dirección 2012.

COMPARACIÓN:

Se pueden comparar dos punteros en una expresión relacional (mayor que, menor que, igual que, etc.). Generalmente, la comparación de punteros se utiliza cuando dos o más punteros apuntan a un objeto común.

if(p<q) printf (“p apunta a menor memoria que q”);

EJEMPLO:

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

void guarda(int i);
int recupera(void);
```

```
int *top, *p1, pila[50];

void main(void)
{
     int valor;
     clrscr();
     top=pila;
     p1=pila;

     do{
          printf("introducir un valor: ");
          scanf("%d",&valor);
          if(valor!=0)
          guarda(valor);
          else
          printf("En lo alto: %d \n",recupera());
     }while(valor!=-1);
}

void guarda(int i)
{
     p1++;
     if(p1==(top+50))
     {
          printf("pila desbordada");
          exit(1);
     }
     *p1=i;
}

int recupera(void)
{
```

```
     if(p1==top)
     {
          printf("pila vacia");
          exit(1);
     }
     p1--;
     return *(p1+1);
}
```

Se puede observar en este programa, que la memoria para la pila la suministra el array pila. El puntero p1 apunta al primer byte de la pila. La variable p1 es la que realmente accede a la pila. La variable top previene el acceso a la pila vacía y el desbordamiento. Una vez que ha sido inicializada la pila, se puede usar guarda() y recupera(). Tanto guarda() como recupera realizan una prueba relacional del puntero p1 para detectar errores de límites. en guarda(), p1 se compara con el final de la pila, sumando TAM (tamaño de la pila) a top. Esto previene el desbordamiento. En recupera() se compara p1 con top para asegurar que no se da un intento de acceder a la pila vacía.

ARRAY DE PUNTEROS:

Los punteros pueden estructurarse en arrays como cualquier otro tipo de datos. Hay que indicar el tipo y el número de elementos. Su utilización posterior es igual que la de los arrays que se vieron en los capítulos previos, con la diferencia de que se asignan direcciones de memoria.

DECLARACIÓN:

```
tipo *nombre[nº elementos];
```

ASIGNACIÓN:

```
nombre_array[indice]=&variable;
```

UTILIZAR ELEMENTOS:

```
*nombre_array[indice];
```

EJEMPLO:

```
#include<stdio.h>

void dias(int n);
void main(void)
{
     int num;
     clrscr();
     printf("Introducir nº de Dia: ");
     scanf("%d",&num);
     dias(num);
     getch();
}

void dias(int n)
{
     char *dia[]={"Nº de dia no Valido", "Lunes", "Martes",
     "Miercoles", "Jueves", "Viernes", "Sabado", "Domingo"};
     pritnf("%s",dia[n]);
}
```

PUNTEROS A CADENAS:

Existe una estrecha relación entre punteros y los arrays. Se asigna al nombre del array la dirección del primer elemento del array y así se conoce el comienzo del array, en el caso de las cadenas de textos (arrays de caracteres) el final lo sabemos mediante el caracter nulo que tienen todas las cadenas de texto al final. Con esta relación se puede declarar una cadena de caracteres sin tener en cuenta la longitud de la cadena. En el caso de las cadenas normales, se debe conocer de antemano el tamaño de ésta.

Otra característica importante es la posibilidad es poder pasar cadenas a las funciones.

SINTAXIS:

```
funcion(char *nombre);
funcion(char *nombre[]);
```

Considérese la siguiente sentencia:

```
char cad[80], *p1;
p1=cad;
```

Esto asigno p1 a la dirección del primer elemento del array cad. Para acceder al quinto elemento, se escribe:

cad[4] o bien *(p1+4)

En el primer caso simplemente estamos asignado el 4 como índice del puntero, ya que el primer elemento es 0, el quinto tiene índice 4.
En el segundo caso estamos sumando 4 al puntero p1, porque p1 ya está apuntando al primer elemento de cad (recordar que el nombre de un array sin índice devuelve la dirección del comienzo del primer elemento del array), por lo que sólo bastan 4 lugares para acceder al dato requerido.

EJEMPLO: En este ejemplo se pueden incluir primer apellido o el primero y el segundo, la longitud de la cadena no importa. Mostramos el contenido de apellidos y su dirección de memoria.

```
#include<stdio.h>

void main(void)
{
     char *apellidos;
     clrscr();

     printf(“Introducir apellidos: “);
     gets(apellidos);
     printf(“Tus apellidos son: %s”,apellidos);
     printf(“La dirección de memoria es: %p”,apellidos);
     getch();
}
```

PUNTEROS A ESTRUCTURAS:

C permite punteros a estructuras igual que permite punteros a cualquier otro tipo de variables. El uso principal de este tipo de punteros es pasar estructuras a funciones. Si tenemos que pasar toda la estructura el tiempo de ejecución puede hacerse eterno, utilizando punteros sólo se pasa la dirección de la estructura. Los punteros a estructuras se declaran poniendo * delante de la etiqueta de la estructura.

SINTAXIS:

```
struct nombre_estructura etiqueta;
struct nombre_estructura *nombre_puntero;
```

Para encontrar la dirección de una etiqueta de estructura, se coloca el operador & antes del nombre de la etiqueta. Para acceder a los miembros de una estructura usando un puntero usaremos el operador flecha (->).

EJEMPLO:

```
#include<stdio.h>
void main(void)
{
     struct ficha{
          int balance;
          char nombre[80];
     }*p;
```

```
     clrscr();
     printf(“Nombre: “);
     gets(p->nombre);
     printf(“\nBalance: “);
     scanf(“%d”,&p->balance);
     printf(“%s”,p->nombre);
     printf(“%d”,p->balance);
     getch();
}
```

ASIGNACIÓN DINAMICA DE MEMORIA

La asignación dinámica es la forma en que un programa puede obtener memoria mientras se está ejecutando. Cuando se ejecuta un programa normalmente se les asigna, a las variables globales, memoria en tiempo de compilación. Las variables locales usan la pila. Sin embargo, durante la ejecución del programa no se pueden añadir variables globales o locales. Sin embargo, hay ocasiones en que un programa necesita usar cantidades de memoria variable. Por ejemplo, un procesador de textos o una base de datos pueden sacar partido de toda la RAM de que disponga el sistema. Sin embargo, dado que las cantidades de memoria RAM disponible varían entre los distintos computadores, los procesadores de texto y las bases de datos no pueden hacerlo mediante variables normales. En su lugar, esos y otros programas disponen de la memoria a medida que la necesiten mediante las funciones de asignación dinámica de C.

La memoria, dispuesta mediante las funciones de asignación dinámica, se obtiene del montón (región de la memoria libre que queda entre el programa y la pila). Aunque el tamaño del montón es desconocido generalmente contiene una gran cantidad de memoria.

El centro del sistema de asignación dinámica esta compuesto por las funciones *malloc(,)* que asigna memoria a un puntero, y *free()*, que libera la memoria asignada. Aunque muchos de los compiladores de C proporcionan otras funciones de asignación dinámica, las anteriores son las más importantes. Estas funciones trabajan juntas usando la región de memoria libre para establecer y gestionar una lista de memoria disponible. Cada vez que se hace una petición de memoria con malloc(), se asigna una parte de la memoria libre restante, y cada vez que se hace una liberación de memoria con free(), se devuelve la memoria al sistema. Para poder utilizar estas funciones se necesita el archivo de cabecera stdlib.h.

SINTAXIS:

```
puntero=malloc(numero_de_bytes);
free(puntero);
```

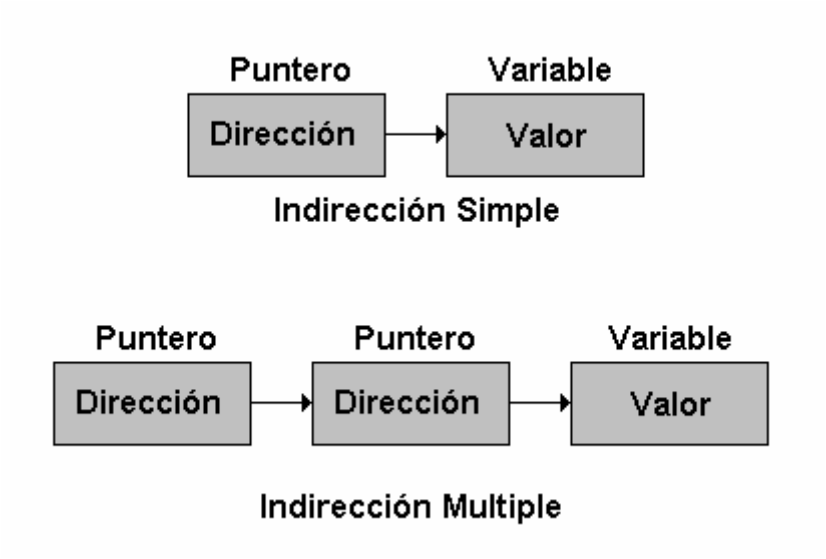
EJEMPLO:

```
#include <stdio.h>
#include <stdlib.h>
void main(void)
{
     char *c;
     clrscr();
```

```
     c=malloc(80);
     if(!c)
     {
          printf("Fallo al asignar memoria");
          exit(1);
     }
     printf("Introducir cadena: ");
     gets(c);
     for(t=strlen(c)-1;t>=0;t--)
     putchar(c[t]);
     free(c);
     getch();
}
```

INDIRECCIÓN MÚLTIPLE

Se puede hacer que un puntero apunte a otro puntero que apunte a un valor de destino. Esta situación se denomina indirección múltiple o punteros a punteros. Los punteros a punteros pueden resultar confusos.



En el caso de un puntero a puntero, el primer puntero contiene la dirección del segundo puntero, que apunta al objeto que contiene el valor deseado. La siguiente declaración indica al compilador que es un puntero a puntero de tipo float.

```
float **balance;
```

EJEMPLO DE DISTINTOS TIPOS DE DECLARACIÓN 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 unpuntero 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

FUNCIONES

Las funciones son los bloques constructores de C y el lugar donde se produce toda la actividad del programa. Además, son una de las características más importantes de C. Su función (valga la redundancia) es subdividir el programa en varias tareas concurrentes, así el programa principal sólo *observará* unas pocas secuencias de

sentencias con diminutas piezas de programa, de pocas líneas, cuya escritura y corrección es una tarea simple. Las funciones pueden o no devolver y recibir valores del programa.
Una forma general de una función es:

```
especificador_tipo_nombre_de_la_función(lista de parámetros)
{
      cuerpo de la función
}
```

El *especificador_de_tipo* especifica el tipo de valor que devuelve la función. una función puede devolver cualquier tipo de dato excepto un array. Si no se especifica ningún tipo, el compilador asume que la función devuelve como resultado un entero. La *lista de parámetros* es la lista de nombres de variables, separados por comas, con sus tipos asociados que reciben los valores de los argumentos cuando se llama a la función. Una función puede no tener parámetros, en cuyo caso la lista de parámetros está vacía. Sin embargo, incluso cuando no hay parámetros se requieren los paréntesis.
El mecanismo para trabajar con funciones es el siguiente, primero debemos declarar el prototipo de la función, a continuación debemos hacer la llamada y por último desarrollar la función. Los dos últimos pasos pueden cambiar, es decir, no es necesario que el desarrollo de la función este debajo de la llamada.

Antes de seguir debemos conocer las reglas de ámbito de las funciones, las cuales son las reglas que controlan si un fragmento de

código conoce o tiene acceso a otro fragmento de código de datos. En C el código de una función es privado a esa función y no se puede acceder a él mediante una expresión de otra función, a menos que se haga a través de una llamada a esa función. No es posible, por ejemplo, utilizar un goto para saltar en medio de otra función. , el código que comprende el cuerpo de una función está oculto al resto del programa y, a no ser que se usen datos o variables globales, no puede ser afectado por otras partes del programa ni afectarlas.

Las variables que están definidas dentro de una función se llaman variables locales. Una variable local empieza a existir cuando se entra en la función y se destruye cuando se sale de ella. Así, las variables locales no pueden conservar sus valores entre distintas llamadas a la función. La única excepción a esta regla son las denominadas variables estáticas, las cuales se declaran con el especificador de clase de almacenamiento *static.* Esto hace que el compilador trate a la variable como si fuese variable global en cuanto a almacenamiento se refiere, pero también que siga limitando su ámbito al interior de la función.

En C, todas las funciones están al mismo nivel de ámbito. Es decir, no se puede definir una función dentro de otra función. Esta es la razón por la cual C no es técnicamente un lenguaje estructurado en bloques.

SINTAXIS DEL PROTOTIPO:

```
tipo_devuelto nombre_funcion ([parametros]);
```

SINTAXIS DE LA LLAMADA:

```
nombre_funcion([parametros]);
```

SINTAXIS DEL DESARROLLO:

```
tipo_devuelto nombre_funcion ([parametros])
{
     cuerpo;
}
```

Cuando se declaran las funciones es necesario informar al compilador los tamaños de los valores que se le enviarán y el tamaño del valor que retorna. En el caso de no indicar nada para el valor devuelto toma por defecto el valor int.

Si una función va a usar argumentos, debe declarar variables que acepten los valores de los argumentos. Estas variables se llaman parámetros formales de la función. Se comportan como otras variables locales dentro de la función, creándose al entrar en la función y destruyéndose al salir.

LLAMADA POR VALOR Y LLAMADA POR REFERENCIA.

En general, se pueden pasar argumentos a las subrutinas de dos formas. Al primer método se denomina *llamada por valor*. Este método

copia el valor de un argumento en el parámetro no afectan al argumento.

La segunda forma de pasar argumentos es por la *llamada por referencia*. En este método se copia la dirección del argumento en el parámetro. Dentro de la subrutina se usa la dirección para acceder al argumento usado en llamadas. Esto significa que los cambios sufridos por el parámetro afectan al argumento.

En el caso de hacerla por valor se copia el contenido del argumento al parámetro de la función, es decir si se producen cambios en el parámetro no afecta a los argumentos. C utiliza esta llamada por defecto. Cuando se utiliza la llamada por referencia lo que se pasa a la función es la dirección de memoria del argumento, por tanto si se producen cambios estos afectan también al argumento. La llamada a una función se puede hacer tantas veces como se quiera.

PRIMER TIPO

Las funciones de este tipo no devuelven valor ni se les pasa parámetros. Ente caso hay que indicarle que el valor que devuelve es de tipo void y para indicar que no recibirá parámetros también utilizamos el tipo void. Cuando realizamos la llamada no hace falta indicarle nada, se abren y cierran los paréntesis.

Expresión 1: void nombre_funcion(void);

Expresión 2:nombre_funcion();

Ambas expresiones son equivalentes

EJEMPLO: El siguiente programa muestra el paso del programa, del cuerpo principal *main()* al secundario mostrar(), y su vuelta la programa principal. Como el proceso es casi instantáneo al ojo humano (del orden de nano segundos), se aplica un retardo (delay) para observar el paso de uno a otro.

```
#include <stdio.h>

void mostrar(void);

void main(void)
{
     clrscr(); /*Limpia la pantalla*/
     printf("Estoy en la principal\n");
     mostrar(); /*Llamada a función mostrar
     printf("De vuelta en la principal");
     getch();
}

void mostrar(void)
{
     printf("Ahora he pasado a la función\n");
     delay(2000); /*Retardo*/
}
```

SEGUNDO TIPO

Son funciones que devuelven un valor una vez han terminado de realizar sus operaciones pero que solo pueden devolver uno. La

devolución se realiza mediante la sentencia *return*, que además de devolver un valor hace que la ejecución del programa vuelva al código que llamó a esa función. Es necesario indicarle al compilador el tipo de valor que se va a devolver poniendo delante del nombre de la función el tipo a devolver. En este tipo de casos la función es como si fuera una variable, pues toda ella equivale al valor que devuelve.

```
tipo_devuelto nombre_funcion(void);
variable=nombre_funcion();
```

EJEMPLO: El siguiente programa imprime un nº al azar, tipo un sorteo.

```
#include <stdio.h>

int sorteo(void);

void main(void)
{
     int N_azar
     clrscr(); /*Limpia la pantalla*/
     N_azar=sorteo();
     printf("El Nº Sorteado es: %i ", N_azar,"\n");
     getch();
}

int sorteo(void)
{
     return rand();
}
```

TERCER TIPO

Este tipo de funciones pueden o no devolver valores pero lo importante es que estas funciones pueden recibir valores. Para eso es necesario indicarle al compilador cuantos valores recibe y de que tipo es cada uno de ellos. Estos tipos se le indican poniéndolos en los paréntesis que tienen las funciones. Deben ser los mismos que en el prototipo.

Expresión 1: void nombre_funcion(tipo1,tipo2…tipoN);
Expresión 2: nombre_funcion(var1,var2…varN);

Ambas expresiones son equivalentes.

EJEMPLO: En este programa se deben ingresar 2 enteros, luego una cadena (Ejemplo: "Ejemplo de funciones de C"). La función resta recibe por parámetros los datos, multiplica los enteros, imprime el resultado y la cadena.

```
#include<stdio.h>

void resta(int x, int y, char *cad);

void main(void)
{
     int a,b;
     char cadena[20];
     clrscr();
```

```
        printf(“Resta de Valores\n”);
        printf(“Introduzca 2 valores enteros: “);
        scanf(“%d %d”,&a,&b); /*Ingresa los datos*/
        printf(“Introduzca la cadena: “);
        gets(cadena); /*Lee la cadena*/
        resta(a,b,cadena);/*Paso de datos por parámetros*/
        getch();
}

void resta(int x, int y, char *cad)
{
        printf(“total= %d”,x-y);
        printf(“La cadena es: %s”,cad);
}
```

Notar que en la llamada a la función *resta* el primer parámetro ingresado (primer nº entre paréntesis) es el que queda como X, el segundo queda como Y, y el tercero como la cadena. Si se llama a resta con:

resta(b,a,cadena);

El resultado sería distinto, pues ahora *X*, sería *b* e *Y* sería *a*.

LLAMADA POR VALOR Y POR REFERENCIA

En general se pueden pasar argumentos a las subrutinas de 2 formas: por valor o por referencia. La llamada *por valor* copia el valor de un

argumento en el parámetro formal de la subrutina. En este caso, los cambios en el parámetro no afectan el argumento.

Las llamadas por referencia, en cambio, copian la dirección del argumento en el parámetro. Dentro de la subrutina se usa la dirección para acceder al argumento usado en la llamada. Esto significa que los cambios sufridos por el parámetro afectan al argumento.

PASO DE ESTRUCTURAS A FUNCIONES

Cuando se utiliza o se pasa una estructura como argumento a una función, se le pasa íntegramente toda la estructura o sea se pasan los argumentos por valor, pero hay que definir como parámetro de la función una estructura.

SINTAXIS:

```
tipo_devuelto nombre_funcion(struc nombre etiqueta);
nombre_funcion(etiqueta_estructura);
```

EJEMPLO: En este programa se crea una estructura ficha, que contiene espacio para nombre y edad. En el programa principal se crea una estructura ficha llamada *trabaja,* y se le pasa por argumento a *funcio*n. En esa función se posiciona el cursor y pide ingresar el nombre y edad, los cuales los ingresa como datos de emple. Luego imprime los datos ingresados en pantalla.

```
#include <stdio.h>

struct ficha{      /* Se define una estructura ficha*/
     char nombre[20];
     int edad;
};

void funcion(struct ficha emple);/*Notar que el argumento de
la función es una estructura del tipo ficha llamada emple */

void main(void)
{
     struct ficha trabaja;/*Se crea una estructura trabaja
del tipo ficha*/
     clrscr();

     funcion(trabaja);
}
void funcion(struct ficha emple)
{
     gotoxy(5,2);
     printf("Nombre: ");
     gotoxy(5,3);
     printf("edad: ");
     gotoxy(13,2);
     gets(emple.nombre);
     gotoxy(13,3);
     scanf("%d",&emple.edad);
     clrscr();
     printf("%s de %d a¤os",emple.nombre,emple.edad);
     getch();
}
```

PASO Y DEVOLUCION DE PUNTEROS:

Cuando se pasan punteros como parámetros se esta haciendo una llamada por referencia, los valores de los argumentos cambian si los parámetros de la función cambian. La manera de pasar punteros es igual que cuando se pasan variables, sólo que ahora hay que preceder al nombre del parámetro con un asterisco.

```
valor_devuelto nombre_funcion(*param1,*param2,…*paramN);
nombre_funcion(var1,var2,…varN);
```

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;
}
```

ARGUMENTOS DE MAIN(): ARGC Y ARGV

Algunas veces es útil pasar información al programa cuando se ejecuta: el método general es pasar información a la función main() mediante el uso de argumentos, el lugar desde donde se pasan esos valores es la línea de órdenes del sistema operativo. Los argumentos de órdenes de línea son la información que sigue al nombre del programa en la línea de órdenes del sistema operativo.

Hay dos argumentos especiales ya incorporados, **argc** y **argv** que se utilizan para recibir esa información. En **argc** contiene el número de argumentos de la línea de órdenes y es un entero. Siempre vale como mínimo 1, ya que el nombre del programa cuenta como primer argumento. El caso de **argv** es un puntero a un array de punteros de caracteres donde se irán guardando todos los argumentos. Todos ellos son cadenas.

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.

SINTAXIS:

```
void main(int argc, char *argv[])
{
     cuerpo;
}
```

EJEMPLO:

Al ejecutar el programa a continuación del nombre le indicamos 2 valores.

```
#include<stdio.h>

void main(int argc, char *argv[])
{
     clrscr();

     if(argc!=3)   //controlo el nº de argumentos
     {
          printf("Error en nº de argumentos");
          exit(0);
     }

     printf("Suma de valores\n");
```

```
     printf("Total= %d",atoi(argv[1])+atoi(argv[2]));
     getch();
}
```

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: Esta función calcula el factorial de un entero.

```
int fact(int numero)
{
     int resp;

     if(numero==1)
          return 1;

     resp=fact(numero-1)*numero;
     return(resp);
}
```

FUNCIONES DE CARACTERES Y CADENAS

La biblioteca estándar de C tiene un rico y variado conjunto de funciones de manejo de caracteres y caracteres. En una implementación estándar, las funciones de cadena requieren el archivo de cabecera STRING.H, que proporciona sus prototipos. Las funciones de caracteres utilizan CTYPE.H, como archivo de cabecera.
Como C no comrpueba los límites durante las operaciones sobre arrays, el programador es el responsable de prevenir los posibles desbordamientos de los arrays. De acuerdo con el estándar ANSI C, si se desborda un array, "su comportamiento quedará indefinido", lo que en otras palabras significa que el programa seguramente fallará.
En C, un caracter imprimible es cualquiera que pueda mostrarse en una pantalla. Normalmente son los caracteres que se encuentran entre el espacio (0x20) y el tilde (0x7E). Los caracteres de control tienen valores entre 0 y 0x1F, además de SUPR (0x7F).
Las funciones de caracteres tieen declarados sus parámetros como enteros. Sin embargo, sólo se utiliza el byte menos significativo; la función de carateres convierte automáticamente su argumento a unsigned char. De caulquier forma, se pueden utilizar argumentos de tipo caracter al llamarlas, ya que los caracteres se transforman automáticamente en enteros en el momento de la llamada.

A continuación una breve dscripción de las funciones más comunes:

ISALPHA:

Esta función devuelve un entero *distinto de cero* si la variable es una letra del alfabeto, en caso contrario devuelve un cero. El archivo de cabecera necesario es <ctype.h>.

int isalpha(*char);

ISDIGIT:

Esta función devuelve un entero *distinto de cero* si la variable es un número (entre 0 y 9), en caso contrario devuelve un cero. El archivo de cabecera necesario también es <ctype.h>.

int isdigit(*char);

ISGRAPH:

Esta función devuelve un entero *distinto de cero* si la variable es cualquier carácter imprimible distinto del espacio, pero si es un espacio devuelve *cero*. El archivo de cabecera necesario es <ctype.h>.

int isgraph(*char);

ISLOWER:

Esta función devuelve un entero *distinto de cero* si la variable esta en minúscula, en caso contrario devuelve un cero. El archivo de cabecera necesario es <ctype.h>.

int islower(*char);

ISPUNCT:

Esta función devuelve un entero *distinto de cero* si la variable es un carácter de puntuación, en caso contrario, devuelve un cero. El archivo de cabecera necesario es <ctype.h>

int ispunct(*char);

ISUPPER:

Esta función devuelve un entero *distinto de cero* si la variable esta en mayúsculas, en caso contrario, devuelve un cero. El archivo de cabecera necesario es <ctype.h>

int isupper(varible_char);

EJEMPLO: Este programa cuenta el número de letras y números que hay en una cadena. La longitud debe ser siempre de cinco por no conocer aún la función que me devuelve la longitud de una cadena.

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

void main(void)
{
     int ind,cont_num=0,cont_text=0;
     char temp;
```

```
     char cadena[6];
     clrscr();

     printf("Introducir 5 caracteres: ");
     gets(cadena);/*Se pide ingresar la cadena al usuario*/
     for(ind=0;ind<5;ind++)
     {
          temp=isalpha(cadena[ind]);
          if(temp)
               cont_text++;
          else
               cont_num++;
     }

     printf("El total de letras es  %d\n",cont_text);
     printf("El total de numeros es %d",cont_num);
     getch();
}
```

EJEMPLO: Utilizando el resto de las funciones, este programa nos entrega una información completa del valor que contiene la variable.

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

void main(void)
{
     char letra;
     clrscr();

     printf("Introducir valor: ");
```

```
     letra=getchar();

     if(isdigit(letra))
          printf("Es un numero");
     else
     {
          if(islower(letra))
               printf("Letra en minuscula");
          else
               printf("Letra en mayuscula");
          if(ispunct(letra))
               printf("Caracter de puntuacion");
          if(!isgraph(letra))
               printf("Es un espacio");
     }
     getch();
}
```

MEMSET:

Esta función Inicializa una región de memoria (buffer) con un valor determinado. Se utiliza principalmente para inicializar cadenas con un valor determinado. El archivo de cabecera necesario es <string.h>.

memset (var_cadena,'carácter',tamaño);

STRCAT

Esta función concatena cadenas, es decir, añade la segunda cadena a la primera, sin que la primera cadena pierda su valor original. Lo único

que se debe tener en cuenta es que la longitud de la primera cadena debe tener la longitud suficiente para guardar la suma de las dos cadenas. El archivo de cabecera necesario es <string.h>.

strcat(cadena1,cadena2);

STRCHR:

Esta función devuelve un puntero a la primera ocurrencia, del carácter especificado, en la cadena donde se busca. Si no lo encuentra, devuelve un puntero nulo. El archivo de cabecera necesario es <string.h>.

strchr(cadena,'carácter');
strchr("texto",'carácter');

STRCMP:

Esta función compara alfabéticamente dos cadenas y devuelve un entero basado en el resultado de la comparación. La comparación no se basa en la longitud de las cadenas. Esta función es muy utilizada para comprobar contraseñas. El archivo de cabecera necesario es <string.h>.

strcmp(cadena1,cadena2);
strcmp(cadena2,"texto");

Resultado Comparación STRCMP	
Valor	Descripción
Menor a Cero	Cadena 1 menor que Cadena 2
Cero	Cadena 1 igual a Cadena 2
Mayor a Cero	Cadena 1 mayor que Cadena 2

STRCPY:

Esta función copia el contenido de un array, especfícamente el de la segundo array en la primera. El contenido de la primera cadena se pierde, y el la segunda cadena debe estar aunpuntando a NULL. Lo único que se debe contemplar es que el tamaño de la segunda cadena sea menor o igual a la cadena donde la copiamos. El archivo de cabecera necesario es <string.h>.

```
strcpy(cadena1,cadena2);
strcpy(cadena1,”texto”);
```

STRLEN

Esta función devuelve la longitud de una cadena terminada en nulo. El carácter nulo no se contabiliza. Y la función devuelve un valor entero que indica la longitud de la cadena. El archivo de cabecera necesario es <string.h>.

```
variable=strlen(cadena);
```

STRNCAT

La función strncat() concatena el número de caracteres, definido en la misma, de la segunda cadena en la primera, y coloca un caracter nulo al final de la primera cadena. La primera cadena no pierde información pues se posicionan desde su último caracter (el caracter nulo). Esto no modifica la segunda cadena. Se debe controlar que la longitud de la primera cadena sea lo suficientemente larga para almacenar las dos cadenas, pues en caso de solaparse, strncat() queda indefinido y el programa falla. El archivo de cabecera necesario es<string.h>.

strncat(cadena1,cadena2,nº de caracteres);

STRNCMP

Esta función compara alfabéticamente un número de caracteres entre dos cadenas y devuelve un entero basado según el resultado de la comparación. Ambas cadenas deben estar finalizadas por un caracter nulo. Los valores devueltos son los mismos que en la función strcmp. El archivo de cabecera <string.h>.

strncmp(cadena1,cadena2,nº de caracteres);
strncmp(cadena1,"texto",nº de caracteres);

STRNCPY

Esta función copia el número de caracteres definidos de la segunda cadena a la primera. La cadena 2 debe estar finalizada por un caracter nulo. En la primera cadena se pierden aquellos caracteres que se copian de la segunda. En caso de que la cadena 2 tenga menos caracteres que la segunda, se añaden caracteres nulos al final de cad 1 hasta que se hayan copiado n caracteres. Alternativamente, si la cadena 2 tiene más de n caracteres, la cadena 1 no tendrá caracter nulo de terminación. El archivo de cabecera necesario es <string.h>.

strncpy(cadena1,cadena2,nº de caracteres);
strncpy(cadena1,"texto",nº de caracteres);

STRRCHR

La función strrchr() devuelve un puntero a la última ocurrencia del carácter buscado en la cadena. Si no lo encuentra devuelve un puntero nulo. El archivo de cabecera necesario es <string.h>.

strrchr(cadena,'carácter');
strrchr("texto",'carácter');

STRPBRK

Esta función devuelve un puntero al primer carácter de la cadena que coincida con algún carácter de la cadena a buscar. Si no hay correspondencia devuelve un puntero nulo. El archivo de cabecera necesario es <string.h>.

strpbrk(“texto”,”cadena_de_busqueda”);

strpbrk(cadena,cadena_de_busqueda);

TOLOWER

Esta función devuelve el carácter equivalente al de la variable en minúsculas, si la variable es una letra; en cas contrario devuelve el caracter sin modificar. El archivo de cabecera necesario es <ctype.h>.

*char=tolower(*char);

TOUPPER

La función toupper devuelve el carácter equivalente al de la variable en mayúsculas, si la variable es una letra; en caso contrario devuelve el caracter sin modificar. El archivo de cabecera necesario es <ctype.h>.

*char=toupper(*char);

EJEMPLO: Copia, concatena, mide e inicializa cadenas.

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

void main(void)
{
```

```
     char *origen,destino[20];
     clrscr();
     printf("Introducir Origen: ");
     gets(origen);
     strcpy(destino,origen);
     printf("%s\n%s\n\n",destino,origen);
     getch();
     memset(destino,'\0',20);
     memset(destino,'x',6);

     if(strlen(origen)<14)
     {
          printf("Se pueden concatenar\n");
          strcat(destino,origen);
     }
     else
          printf("No se pueden concatenar\n");
     printf("%s",destino);
     getch();
}
```

EJEMPLO: Pide una contraseña de entrada. Luego pide 10 códigos de 6 dígitos. Por último pide los 3 primeros dígitos de los códigos que deseas ver.

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

void main(void)
{
     int cont;
```

```
char ver_codigo[4];
char contra[6]="abcde";
char tu_contra[6];
char codigos[10][7];
clrscr();

printf("CONTRASEÑA ");
gets(tu_contra);
clrscr();

if(strcmp(tu_contra,contra))
{
     printf("ERROR");
     delay(2000);
     exit(0);
}

printf("Introducir Codigos\n");
for(cont=0;cont<10;cont++)
{
     printf("%2.2d: ",cont+1);
     gets(codigos[cont]);
}
clrscr();
printf("código a listar? ");
gets(ver_codigo);
for(cont=0;cont<10;cont++)
{
     if(!strncmp(ver_codigo,codigos[cont],3))
          printf("%s\n",codigos[cont]);
}
getch();
```

```
}
```

EJEMPLO: En este ejemplo se busca en una cadena a partir de un grupo de caracteres. Si no encuentra coincidencia con ninguna letra en la cadena muestra un mensaje de error.

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

void main(void)
{
     char letras[5];
     char *resp;
     char cad[30];
     clrscr();

     printf("Introducir cadena: ");gets(cad);
     printf("Posibles letras(4): ");gets(letras);
     resp=strpbrk(cad,letras);
     if(resp)
          printf("%s",resp);
     else
          printf("Error");
     getch();
}
```

FUNCIONES MATEMÁTICAS

El estándar C define 22 funciones matemáticas que entran en las siguientes categorías: trigonométricas, hiperbólicas, logarítmicas, exponenciales y otras. Todas las funciones requieren el archivo de cabecera **MATH.H**. Si un argumento de una función matemática no se encuentra en el rango para el que está definido, devolverá un valor definido **EDOM**.

ACOS

Esta función muestra el arcocoseno de la variable y devuelve este valor en un tipo double.. La variable debe ser de tipo double y debe estar en el rango –1 y 1, en otro caso se produce un error de dominio. El archivo de cabecera necesario es <math.h>.

double acos(variable_double);

ASIN:

Esta función muestra el arcoseno de la variable y devuelve este valor en un tipo double.. La variable debe ser de tipo double y debe estar en el rango –1 y 1, en otro caso se produce un error de dominio. El archivo de cabecera necesario es <math.h>.

double asin(variable_double);

ATAN:

Esta función muestra el arcotangente de la variable y devuelve este valor en un tipo double. La variable debe ser de tipo double y debe estar en el rango –1 y 1. En otro caso se produce un error de dominio. El archivo de cabecera necesario es <math.h>.

double atan(variable_double);

COS

Esta función muestra el coseno de la variable y devuelve este valor en un tipo double. La variable debe ser de tipo double y debe estar expresada en radianes. El archivo de cabecera necesario es <math.h>.

double cos(variable_double_radianes);

SIN

Esta función muestra el seno de la variable y devuelve este valor en un tipo double. La variable debe ser de tipo double y estar expresada en radianes. El archivo de cabecera necesario es <math.h>.

double sin(variable_double_radianes);

TAN:

Esta función muestra la tangente de la variable y devuelve este valor en un tipo double. La variable debe ser de tipo double y debe estar expresada en radianes. El archivo de cabecera necesario es <math.h>.

double tan(variable_double_radianes);

COSH

ESta función muestra el coseno hiperbólico de la variable y devuelve este valor en un tipo double.. La variable debe ser de tipo double y debe estar en el rango –1 y 1. En otro caso se produce un error de dominio. El archivo de cabecera necesario es <math.h>.

double cosh(variable_double);

SINH

Esta función muestra el seno hiperbólico de la variable y devuelve un tipo double. La variable debe ser de tipo double y debe estar en el rango de –1 y 1. En otro caso se produce un error de dominio. El archivo de cabecera necesario es <math.h>.

double sinh(variable_double);

TANH

Esta función muestra la tangente hiperbólica de la variable y devuelve este valor en un tipo double. La variable debe ser de tipo double y debe estar dentro del rango –1 y 1. En otro caso se produce un error de dominio. El archivo de cabecera necesario es <math.h>.

double tanh(variable_double);

EJEMPLO: Esta función entrega el coseno y el seno de los ángulo ingresados.

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

void main(void)
{
     double radianes;
     clrscr();

     printf("Introducir radianes: ");
     scanf("%f",&radianes);

     printf("Coseno=   %f\n",cos(radianes));
     printf("Seno=     %f\n",sin(radianes));
     printf("Tangente= %f",tan(radianes));
     getch();
}
```

CEIL

Esta función devuelve un double que representa el menor entero que no es menor que la variable redondeada. Por ejemplo, dado 1.02 devuelve 2.0. En cambio si asignamos –1.02 devuelve –1. En resumen redondea la variable al entero superior más cercano. El archivo de cabecera necesario es <math.h>.

double ceil(variable_double);

FLOOR

Esta función devuelve un double que representa el mayor entero que no es mayor que la variable redondeada. Por ejemplo dado 1.02 devuelve 1.0. En cambio si asignamos –1.2 devuelve –2. En resumen redondea la variable al entero inferior más cercano. El archivo de cabecera necesario es <math.h>.

double floor(variable_double);

FABS:

Esta función devuelve un valor float o double que representa el valor absoluto de una variable float. Se considera una función matemática, pero la cabecera necesaria para su uso es <stdlib.h>.

var_float fabs(variable_float);

LABS

Esta función devuelve un valor long que representa el valor absoluto de una variable long. Se considera una función matemática, pero su cabecera es <stdlib.h>.

var_long labs(variable_long);

ABS

Esta función devuelve un valor entero que representa el valor absoluto de una variable int. Se considera una función matemática, pero su cabecera es <stdlib.h>.

var_float abs(variable_float);

MODF:

Esta función devuelve un double. Descompone la variable en su parte entera y fraccionaria. La parte decimal es el valor que devuelve la función, su parte entera la guarda en el segundo termino de la función. Las variables tienen que ser obligatoriamente de tipo double. El archivo de cabecera necesario es <math.h>.

var_double_deimal= modf(variable,var_parte_entera);

POW

Esta función devuelve un double que representa la elevación de un número base a un exponente que nosotros le indicamos. Se produce un error si la base es cero o si el exponente es menor o igual a cero. El archivo de cabecera necesario es <math.h>.

var_double=pow(base_double,exponente_double);

SQRT

Esta función devuelve un double que representa la raíz cuadrada de la variable que se le entrega. La variable no puede ser negativa, y si lo es se produce un error de dominio. El archivo de cabecera necesario es <math.h>.

var_double=sqrt(variable_double);

EJEMPLO: En este ejemplo se ingresa una variable de tipo float. Luego se muestra en pantalla el redondeo al más alto (ceil), el redondeo al más chico (floor), se muestra su parte entera, su parte decimal, su val. absoluto y su raíz cuadrada.

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

void main(void)
{
```

```
     float num;
     double num_dec,num_ent;
     clrscr();

     printf("Introducir Numero: ");
     scanf("%f",&num);
     gotoxy(9,3);
     printf("ALTO: %.1f",ceil(num));
     gotoxy(1,4);
     printf("REDONDEO");
     gotoxy(9,5);
     printf("BAJO: %.1f",floor(num));
     num_dec=modf(num,&num_ent);
     gotoxy(12,8);
     printf("ENTERA:  %.2f",num_ent);
     gotoxy(1,9);
     printf("DESCONPONGO");
     gotoxy(12,10);
     printf("DECIMAL: %.2f",num_dec);
     gotoxy(1,13);
     printf("VALOR ABSOLUTO: %.2f",fabs(num));
     gotoxy(1,16);
     printf("R.CUADRADA:  %.2f",sqrt(fabs(num)));
     getch();
}
```

LOG

Esta función devuelve un double. que representa el logaritmo natural (o neperiano) de la variable. Se produce un error de dominio si la

variable es negativa, y un error de rango si el valor es cero. El archivo de cabecera necesario es <math.h>.

double log(variable_double);

LOG10:

Esta función devuelve un valor double. que representa el logaritmo decimal de la variable de tipo double. Se produce un error de dominio si la variable es negativo y un error de rango si el valor es cero. El archivo de cabecera necesario es <math.h>.

double log10(var_double);

RAMDOMIZE()

Esta función inicializa la semilla para generar números aleatorios. Utiliza las funciones de tiempo para crear esa semilla. Esta función esta relacionada con random. El archivo de cabecera necesario es <stdlib.h>.

void randomize();

RANDOM:

Esta función devuelve un entero. Genera aleatoriamente un número entre 0 y la variable menos uno. Utiliza el reloj del ordenador para ir generando estos valores. El programa debe llevar la función randomize para cambiar la semilla en cada ejecución. El archivo de cabecera necesario es <stdlib.h>.

int random(variable_int);

EJEMPLO: Genera seis números aleatorios entre 1 y 49. No se repite ninguno de ellos.

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

void main(void)
{
	int num=0,num1=0,repe,temp;
	int valores[6];
	clrscr();

	printf("Loteria primitiva: ");
	randomize();

	for(;;) //Ciclo de for infinito, sólo se sale si num=6
	{
		repe=1;
```

```
            if(num==6)
                  break;
            temp=random(49)+1;
            for(num1=0;num1<=num;num1++)
            {
                  if(valores[num1]==temp)
                  {
                        valores[num1]=temp;
                        num--;
                        repe=0;
                        break;
                  }
            }
            if (repe==1)
            valores[num]=temp;
            num++;
      }

      for(num=0;num<6;num++)
            printf("%d ",valores[num]);
      getch();
}
```

FUNCIONES DE CONVERSIÓN

En el estándar de C se definen funciones para realizar conversiones entre valores numéricos y cadenas de caracteres. La cabecera de todas estas funciones es STDLIB.H. Se pueden dividir en dos grupos,

conversión de valores numéricos a cadena y conversión de cadena a valores numéricos.

ATOI

Esta función convierte un string (cadena de caracteres) en un valor entero. El string debe contener un número entero válido, si no es así, el valor devuelto queda indefinido. El string puede estar finalizado con espacios en blanco, signos de puntuación u otros símbolos que no sean dígitos, pues la función los ignora. El archivo de cabecera necesario es <stdlib.h>.

int atoi(*char);

ATOL

Esta función convierte un string en un valor long. El string debe contener un número long válido, si no es así el valor devuelto queda indefinido. La cadena puede terminar con espacios en blanco, signos de puntuación y otros que no sean dígitos, pues la función los ignora. El archivo de cabecera necesario es <stdlib.h>.

long atol(*char);

ATOF

Esta función convierte un string en un valor double. La cadena debe contener un número double válido, si no es así el valor devuelto queda

indefinido. La cadena puede terminar con espacios en blanco, signos de puntuación y otros que no sean dígitos, pues la función los ignora. El archivo de cabecera necesario es <stdlib.h>.

double atof(*char);

SPRINTF

Esta función convierte cualquier tipo numérico a string (o cadena). Para convertir de número a string hay que indicar el tipo de variable numérica y tener presente que la longitud del string debe poder guardar la totalidad del número. Admite también los formatos de salida, es decir, que se puede escoger distintas partes del número. El archivo de cabecera necesario es <stdlib.h>

sprintf(var_cadena,"identificador",var_numerica);

ITOA

Esta función convierte un entero en un string (cadena) equivalente y sitúa el resultado en el string definido en segundo termino de la función. Hay que asegurarse que el string sea lo suficientemente grande para guardar el número. El archivo de cabecera necesario es <stdlib.h>.

itoa(var_entero,var_cadena,base);

Base	DESCRIPCIÓN
2	Convierte el valor en su equivalente binario.
8	Convierte el valor en su equivalente octal.
10	Convierte el valor en su equivalente decimal.
16	Convierte el valor en su equivalente hexadeicmal.

LTOA

Esta función convierte un long en su string equivalente y sitúa el resultado en el string definido en el segundo término de la función. Hay que asegurarse que el string sea lo suficientemente grande para guardar el número. El archivo de cabecera necesario es <stdlib.h>.

ltoa(var_long,var_cadena,base);

EJERCICIO: Mostrar las salidas de este programa para las entradas 193, 105360151,245801640.

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

void main(void)
{
     char texto[4];
     char ntext[10],ntext1[10];
     int num;
     float total;
```

```
     clrscr();

     printf("Numero de 3 digitos: ");
     scanf("%d",&num);
     fflush(stdin);
     printf("Cadena numerica: ");
     gets(ntext);
     fflush(stdin);
     printf("Cadena numerica: ");
     gets(ntext1);
     fflush(stdin);

     sprintf(texto,"%d",num);
     printf("%c  %c  %c\n",texto[0],texto[1],texto[2]);

     total=atof(ntext)+atof(ntext1);
     printf("%.3f",total);
     getch();
}
```

15. FUNCIONES DE FECHA Y HORA

Estas funciones utilizan la información de hora y fecha del sistema operativo. Se definen varias funciones para manejar la fecha y la hora del sistema, así como los tiempos transcurridos. Estas funciones requieren el archivo de cabecera **TIME.H**. En este archivo de cabecera se definen cuatro tipos de estructuras para manejar las funciones de fecha y hora (size_t , clock_t , time_t , time).

TIME

Devuelve la hora actual del calendario del sistema. Si se produce un error devuelve –1. Utiliza la estructura time_t a la cual debemos asignar una etiqueta que nos servirá para trabajar con la fecha y hora. Por si sola no hace nada, necesita otras funciones para mostrar los datos. El archivo de cabecera necesario es <time.h>.

```
time_t nombre_etiqueta;
.
.
nombre_etiqueta=time(NULL);
```

CTIME

Devuelve un puntero a una cadena con un formato d*ía semana mes hora:minutos:segundo año \n\0*. La hora del sistema se obtiene mediante la función time. El archivo de cabecera necesario es <time.h>.

```
puntero=ctime(&etiqueta_estructura_time_t);
```

EJEMPLO: Este programa muestra la hora local definida por el sistema.

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

```
void main(void)
{
     time_t fecha_hora;
     clrscr();

     fecha_hora=time(NULL);
     printf(ctime(&fecha_hora));
     getch();
}
```

GETTIME:

Esta función devuelve la hora del sistema, pero no está definida en el estándar ANSI C y sólo se encuentran en los compiladores de C para DOS. Utiliza la estructura dostime_t para guardar la información referente a la hora. Antes de hacer referencia a la función hay que crear una etiqueta de la estructura. El archivo de cabecera necesario es <dos.h>.

_dos_gettime(&etiqueta_estructura_dostime_t);

Microsoft define la estrutura de dostime_t de la siguiente forma:

```
struct dostime_t{
        unsigned hour;
        unsigned minute;
        unsigned second;
        unsigned hsecond;
}
```

EJEMPLO: Este programa muestra la hora utilizando llamadas a DOS:

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

void main(void)
{
     struct dostime_t ho;
     clrscr();

     _dos_gettime(&ho);
     printf(" %d:%d:%d",ho.hour,ho.minute,ho.second);
     getch();
}
```

GETDATE

Devuelve la fecha del sistema pero no está definida en el estándar ANSI C y sólo se encuentran en los compiladores de C para DOS. Utiliza la estructura dosdate_t para guardar la información referente a la fecha. Antes de hacer referencia a la función hay que crear una etiqueta de la estructura. El archivo de cabecera necesario es <dos.h>.

_dos_getdate(&etiqueta_estructura_dosdate_t);

Microsoft define la estructura _dosdate_t de la siguiente forma:

```
struct dosdate_t{
```

```
    unsigned day; /* Dia */
    unsigned month; /* Mes */
    unsigned year; /* Año */
    unsigned dayofweek; /* Dia de la semana*/
                              /*(0=domingo) */
}
```

EJEMPLO: Este programa muestra la fecha utilizando llamadas a DOS.

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

void main(void)
{
     struct dosdate_t fec;
     clrscr();

     _dos_getdate(&fec);
     printf("%d/%d/%d\n",fec.day,fec.month,fec.year);
     getch();
}
```

SETTIME

Esta función permite cambiar la hora del sistema, pero no está definida en el estándar ANSI C y sólo se encuentran en los compiladores de C para DOS. Utiliza la estructura dostime_t para guardar la información referente a la hora. Antes de hacer referencia a la función hay que crear una etiqueta de la estructura. El archivo de cabecera necesario es <dos.h>.

_dos_settime(&etiqueta_estructura_dostime_t);

SETDATE

Esta función permite cambiar la fecha del sistema, pero no está definida en el estándar ANSI C y sólo se encuentran en los compiladores de C para DOS. Utiliza la estructura dosdate_t para guardar la información referente a la fecha. Antes de hacer referencia a la función hay que crear una etiqueta de la estructura. El archivo de cabecera necesario es <dos.h>.

_dos_setdate(&etiqueta_estructura_dosdate_t);

EJEMPLO: Este programa permite setear la hora y la fecha del sistema.

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

void main(void)
{
     struct dosdate_t fec;
     struct dostime_t ho;
     clrscr();

     printf("Introducir fecha: ");
     gotoxy(19,1);
     scanf("%u%*c%u%*c%u",&fec.day,&fec.month,&fec.year);
```

```
     printf("Introducir hora: ");
     gotoxy(18,2);
     scanf("%u%*c%u%*c%u",&ho.hour,&ho.minute,&ho.second);
     _dos_settime(&ho);
     _dos_setdate(&fec);
}
```

DIFFTIME

Esta función devuelve un double que representa la diferencia, en segundos, entre una hora inicial y hora final. Esto se hace restando la hora final a la hora inicial. Para esto tenemos que obtener la hora al iniciar un proceso y al terminar el proceso. El archivo de cabecera necesario es <time.h>.

double difftime(hora_final,hora_inicial);

EJEMPLO: Este programa determina el número de segundos que tarda en ejecutarse el bucle *for* que va de 0 a 50000:

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

void main(void)
{
     time_t inicio, final;
     double tiempo,cont;
     clrscr();
```

```
     inicio=time(NULL);
     for(cont=0;cont<50000;cont++)
         clrscr();
     final=time(NULL);

     tiempo=difftime(final,inicio);
     printf("Tiempo: %.1f",tiempo);
     getch();
}
```

FUNCIONES DE SISTEMA

Este tema trata sobre funciones que, de una u otra forma, están más próximas al sistema operativo que las demás funciones de la biblioteca estándar de C. Estas funciones permiten interactuar directamente con el sistema operativo. Las funciones de este tema interactúan con el sistema operativo DOS. Este tipo de funciones atizan en unos casos el archivo de cabecera DOS.H y en otros DIR.H.

BIOS_EQUIPLIST

Esta función devuelve un valor que especifica el equipo existente en la computadora. Ese valor está codificado tal como se muestra a continuación.

Bit	Descripción	Equipo
0	Tiene una unidad de Disco	--
1	Con Microprocesador Matemático.	--
2, 3	Tamaño de la RAM en la placa base.	00: 16KB
		01: 32 KB
		10:48 KB
		11:64 KB
4, 5	Modo inicial de Video	00: No usado
		01:42x25 BN adap. color
		10:80x25 BN adap. color
		11:80x25 adap. monocromo
6,7	Unidades de Diskette	00: Una
		01: Dos
		10: Tres
		11: Cuatro
8	Chip DMA	0
9,10,11	Nº de Puertos Serie	000: Cero
		001: Uno
		010: Dos
		011: Tres
		100: Cuatro
		101: Cinco
		110: Seis
		111: Siete

12	Con adaptador de juegos	1
13	Con módem	1
14, 15	Nº de impresoras	00: Cero
		01: Una
		10: Dos
		11: Tres

La cabecera que utiliza esta función es <bios.h>. El modo de trabajar es igualar la función a una variable sin signo, después el resultado se le desplaza a los bits que nos interesan para mostrar la información.

```
var_sinsigno=_bios_equiplist();
var_sinsigno >> nº_bit;
```

EJEMPLO: Este programa entrega el número de disqueteras del equipo donde se ejecuta.

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

void main(void)
{
     unsigned num_bit;
     clrscr();

     num_bit=_bios_equiplist();
```

```
     num_bit>>=6;

     printf("Nº de disqueteras: %d",(num_bit & 3)+1);
     getch();
}
```

GETDISKFREE

Esta función devuelve la cantidad de espacio libre del disco especificado por la unidad (numeradas a partir de 1 que corresponde a la unidad A). Esta función utiliza la estructura diskfree_t. El espacio lo devuelve indicando el número de cilindros libres, para pasarlo a bytes hay que multiplicarlo por 512 y por 64. El archivo de cabecera necesario es <dos.h>.

_dos_getdiskfree(unidad,&etiq_struct_diskfree_t);

EJEMPLO: Este programa entrega la cantidad de memoria libre en disco.

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

void main(void)
{
     struct diskfree_t disco;
     float tam;
     clrscr();
```

```
     _dos_getdiskfree(3,&disco);
     tam= (float)(disco.avail_clusters)*
     (float)(disco.sectors_per_cluster)*
     (float)(disco.bytes_per_sector);
     printf("Tamaño Bytes:    %.0f\n",tam);
     printf("Tamaño Sectores: %d",disco.avail_clusters);
     getch();
}
```

GETDRIVE

Esta función devuelve el número de la unidad de disco actual del sistema y deja el valor en la variable int. Ese valor se puede discriminar por medio de un switch o como en el ejemplo utilizar directamente un printf. La cabecera <dos.h>.

_dos_getdrive(&var_intera);

Valor	Unidad
1	A
2	B
3	C

EJEMPLO: Este programa muestra la unidad de disco en uso.

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

```
void main(void)
{
     int unidad;
     clrscr();

     _dos_getdrive(&unidad);
     printf("La unidad actual es: %c",unidad+'A'-1);
     getch();
}
```

SETDRIVE

Esta función cambia la unidad de disco actual a la especificada por la variable de tipo entero. La función devuelve el número de unidades del sistema en el entero apuntado por la segunda variable de la función. La cabecera <dos.h>.

_dos_setdrive(var_int_unidad,&var_int_unidades);

EJEMPLO: Muestra el Nº de unidades existentes en el equipo.

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

void main(void)
{
     unsigned unit;
     clrscr();

     _dos_setdrive(3,&unit);
```

```
     printf("Nº unidades: %u",unit);
     getch();
}
```

GETCUDIR

Esta función obtiene el directorio actual de la unidad que se le especifica mediante un entero. Esta función devuelve CERO si todo se produce correctamente, en caso contrario devuelve uno. El archivo de cabecera necesario es <dir.h>.

int getcurdir(int_unidad,cadena);

EJEMPLO:

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

void main(void)
{
     char *director;
     clrscr();

     getcurdir(3,director);
     printf("Directorio: %s",director);
     getch();
}
```

FINDFIRST/FINDNEXT

La función findfirst busca el primer nombre de archivo que coincida con el patrón de búsqueda. El patrón puede contener la unidad como el camino donde buscar. Además el patrón puede incluir los caracteres comodines ***** y **¿?**. Si se encuentra alguno, rellena con información la estructura find_t.

int _dos_findfirst(patron,atrib,&etique_find_t);

La función findnext continúa la búsqueda que haya comenzado con findfirst. Devuelve CERO en caso de éxito y un valor DISTINTO DE CERO si no tiene éxito la búsqueda. El archivo de cabecera necesario es <dos.h>.

int _dos_findnext(&etiqueta_find_t);

EJEMPLO:

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

void main(void)
{
     struct find_t fiche;
     int fin;

     fin=_dos_findfirst("*.c",_A_NORMAL,&fiche);
```

```
     while(!fin)
     {
          printf("%s %ld\n",fiche.name,fiche.size);
          fin=_dos_findnext(&fiche);
     }
     getch();
}
```

REMOVE

La función elimina el archivo especificado en la variable. Devuelve CERO si consigue eliminar el archivo, y MENOS UNO si se produce algún error. El archivo de cabecera necesario es <stdio.h>.

int remove(variable_cadena);

RENAME

La función cambia el nombre del archivo especificado en primer término por el nombre de la segunda variable cadena. El nuevo nombre no debe coincidir con ninguno que exista en el directorio. Devuelve CERO si tiene éxito y DISTINTO DE CERO si se produce algún error. El archivo de cabecera necesario es <stdio.h>.

int rename(var_nombre_antiguo,var_nombre_nuevo);

EJEMPLO:

```
#include<stdio.h>

void main(int argc, char *argv[])
{
     clrscr();
     if (argc!=3)
     {
          printf("Error en los argumentos");
          exit(0);
     }
     if(remove(argv[1]))
          printf("El archivo no esta\n");
     if(rename(argv[2],"nuevo.txt"))
          printf("No puedo cambiar nombre\n");
}
```

MKDIR

Esta función permite crear directorios. El directorio dependerá de aquel donde estemos situados a la hora de crearlo. Devuelve CERO si todo ha ido correctamente y DISTINTO de cero si hay algún error. El archivo de cabecera necesario es <dir.h>.

int mkdir(variable_cadena);

CHDIR

Esta función permite cambiarse de directorio. Hay que indicarle la ruta completa para poder cambiarse. Devuelve CERO si todo ha ido correctamente y DISTINTO de cero si hay algún error. El archivo de cabecera necesario es <dir.h>.

int chdir(variable_cadena);

RMDIR

Esta función permite borrar el directorio que le indiquemos. Las condiciones para borrar el directorio es que este vacío y estar en el directorio que le precede. Devuelve CERO si todo ha ido correctamente y DISTINTO de cero si hay algún error. El archivo de cabecera necesario es <dir.h>.

int rmdir(variable_cadena);

SYSTEM

La función pasa la cadena como una orden para el procesador de órdenes del sistema operativo. El valor devuelto por system normalmente es CERO si se realiza todo correctamente y DISTINTO de cero en cualquier otro caso. No es muy utilizada, al llamarla perdemos todo el control del programa y lo coge el sistema operativo.

Para probar esta orden es necesario salir a MS-DOS. La cabecera utilizada es <stdlib.h>.

int system(variable_cadena);

SOUND/NOSOUND

La función sound hace que el altavoz del ordenador comience a pitar con una frecuencia determinada por la variable. El altavoz seguirá pitando hasta que el programa lea una línea con la función nosound(). El archivo de cabecera necesario es <dos.h>.

sound(int_frecuencia);
nosound();

EJEMPLO:

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

void main(void)
{
     char *directorio;
     clrscr();

     printf("Nombre del directorio: ");
     gets(directorio);
     if(!mkdir(directorio))
          printf("Directorio creado\n");
```

```
     else
     {
          printf("No se pudo crear directorio\n");
          delay(1000);
          exit(0);
     }
     getch();
     system(“dir/p”);
     getch();
     if(!rmdir(directorio))
          printf("\nDirectorio borrado\n");
     else
    {
          printf("\nNo se pudo borrar\n");
          delay(1000);
          exit(0);
     }
     getch();
}
```

FUNCIONES GRÁFICAS

Los prototipos de las funciones gráficas y de pantalla se encuentran en GRAPHICS.H. Lo primero que hay que tener en cuenta son los distintos modos de vídeo para el PC. Cuando utilizamos funciones gráficas hay que iniciar el modo gráfico y esto supone un cambio en las coordenadas en pantalla y el tamaño.

La parte más pequeña de la pantalla direccionable por el usuario en *modo texto* es un carácter, en cambio en *modo gráfico* es el pixel. Otro aspecto es que en modo texto la esquina superior izquierda en la coordenada 1,1 y como esquina inferior derecha 24,80. En el modo gráfico la esquina superior izquierda es 0,0 y la esquina inferior derecha depende del modo de vídeo. Se pude obtener mediante las funciones getmaxx() y getmaxy().

Para compilar un archivo fuente con funciones gráficas tenemos primero que indicar al compilador que se va a trabajar con ese tipo de funciones.

Esto se realiza en el Menú OPTIONS → LINKER → LIBRARIES

LISTA DE COLORES

Texto	
Valor	Color
0	Negro
1	Azul
2	Verde
3	Cyan
4	Rojo
5	Magenta
6	Marrón
7	Gris Claro

Fondo Gráfico	
Valor	Color
0	Negro
1	Azul
2	Verde
3	Cyan
4	Rojo
5	Magenta
6	Marrón
7	Gris Claro

8	Gris Oscuro
9	Azul Claro
10	Verde Claro
11	Cyan Claro
12	Rojo Claro
13	Magenta Claro
14	Amarillo
15	Blanco
n+128	Parpadean

8	Gris Oscuro
9	Azul Claro
10	Verde Claro
11	Cyan Claro
12	Rojo Claro
13	Magenta Claro
14	Amarillo
15	Blanco

Fondo del Color	
Valor	Color
0	Negro
1	Azul
2	Verde
3	Cyan
4	Rojo
5	Magenta
6	Marrón

TEXTCOLOR

Establece el color del texto. No es una función gráfica. La variable que tiene como parámetro especifica el color, se puede poner una variable o directamente el nº de color. Se mantiene ese color hasta el momento en que lee otro valor para textcolor. Cabecera <conio.h>.

textcolor(int color);

CPRINTF

Tiene la misma función y formato que printf pero mediante esta función el texto se muestra con el color especificado en texcolor. Cabecera <conio.h>.

cprintf(“mensaje”);
cprintf(“identificador_formato”,variable);

TEXTBACKGROUND

Establece el color de fondo donde se va mostrando el texto, solo aparece cuando hay texto escrito. Se puede utilizar una variable o el valor numérico. Se mantiene hasta que lee otro valor para textbackground. Cabecera <conio.h>.

texbackground(int color);

SETCURSORTYPE

Esta función establece el tipo de cursor que se va a ver en pantalla. Se mantiene hasta el momento en que lee otro valor para en la función setcursortype. Admite tres constantes definidas en el compilador de C. Cabecera <conio.h>.

_setcursortype(constante_tipo_cursor);

Constantes del Cursor	
Constante	Descripción
_NOCURSOR	No se ve el cursor.
_SOLIDCURSOR	Cursor Grande.
_NORMALCURSOR	Cursor Normal.

EJEMPLO:

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

void main(void)
{
     char cadena[20]="hola que tal";
     clrscr();

     _setcursortype(_NOCURSOR);
     textbackground(4);
     textcolor(1);
     cprintf("%s",cadena);
     getch();
}
```

INICIO GRAFICO

Antes de poder utilizar cualquiera de las funciones gráficas hay que iniciar el modo gráfico. Al iniciar el modo gráfico hay que indicarle cual

es el modo de vídeo y el directorio donde están las funciones gráficas en C. Para indicar la tarjeta y el modo se utilizan variables de tipo entero asignándoles una constante llamada DETECT. El archivo de cabecera necesario es <graphics.h>.

```
initgraph(&var_tarjeta,&var_modo,"dir_lib_graficas");
```

GRAPHRESULT

Una vez iniciado el modo gráfico debemos comprobar que se ha iniciado correctamente el modo gráfico para poder utilizar las funciones gráficas sin ningún problema. La función devuelve un entero que debemos comparar con la constante grOk, si son iguales el modo gráfico se ha iniciado correctamente.

```
variable_int=graphresult();
```

CERRAR GRAFICO

Una vez que se ha terminado de trabajar con las funciones gráficas se debe cerrar siempre el modo gráfico. Si un programa termina sin cerrar el modo gráfico el sistema operativo mantendrá el modo gráfico. El archivo de cabecera necesario es <graphics.h>.

```
closegraph();
```

CLEARDEVICE

Esta función borra la pantalla una vez que se ha inicializado el modo gráfico. Si se utiliza la orden clrscr solo se limpiara una parte de la pantalla. El archivo de cabecera necesario es <graphics.h>.

cleardevice();

SETBKCOLOR

Establece el color de fondo en la pantalla. Hay que indicarle mediante variable o valor numérico. Cuando se borra la pantalla este no desaparece. El archivo de cabecera necesario es <graphics.h>.

setbkcolor(int_color);

EJEMPLO:

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

void main(void)
{
     int tarjeta= DETECT,modo=DETECT;
     int color;
     clrscr();

     initgraph(&tarjeta, &modo, "c:\\tc\\bgi");
     if(graphresult()!=grOk)
```

```
     {
          printf("Error en el modo grafico");
          exit(0);
     }
     cleardevice();
     for(color=0;color<=15;color++)
     {
          setbkcolor(color);
          delay(1000);
     }
     closegraph();
}
```

SETCOLOR

Establece el color del borde con el que van a mostrarse los objetos gráficos que se van pintando. Por defecto los objetos aparece con el borde de color blanco. El valor que tiene la función es un entero. El archivo de cabecera necesario es <graphics.h>.

setcolor(int_color);

SETFILLSTYLE

Establece el estilo de relleno de los objetos. Hay que indicar el tipo y el color, teniendo en cuenta que el color debe ser igual al que establecemos en setcolor. Esto simplemente establece el estilo. El archivo de cabecera necesario es <graphics.h>.

setfillstyle(int_estilo,int_color);

ESTILO	DESCRIPCIÓN
1	Sólido 100%
2	Líneas horizontales gruesas.
3	Líneas diagonales finas ///
4	Líneas diagonales gruesas ///
5	Líneas diagonales gruesas \\\
6	Líneas diagonales finas \\\
7	Líneas horizontales y verticales cruzadas.
8	Líneas diagonales cruzadas.
9	Líneas diagonales cruzadas muy juntas.
10	Puntos separados.
11	Puntos cercanos.

FLOODFILL

Rellena el objeto que ha sido pintado previamente. Hay que indicarle las mismas coordenas que tiene el objeto y el color debe ser igual al borde y al estilo de relleno. El archivo de cabecera necesario es <graphics.h>.

floodfill(int_x,int_y,in_color);

CIRCULO

Dibuja un circulo en una posición indicada mediante las coordenadas x e y, también hay que indicarle un color. El archivo de cabecera necesario es <graphics.h>.

circle(int_x,int_y,int_color);

RECTANGULO

Dibuja un rectángulo o cuadrado. Se le indica la posición de la esquina superior izquierda mediante los valores x1,y1 y la esquina inferior derecha mediante los valores de x2,y2. El archivo de cabecera necesario es <graphics.h>.

rectangle(int_x1,int_y1,int_x2,int_y2);

EJEMPLO:

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

void main(void)
{
     int tarjeta= DETECT,modo=DETECT;
     int color;
```

```
     initgraph(&tarjeta, &modo, "c:\\tc\\bgi");
     if(graphresult()!=grOk)
     {
          printf("Error en el modo gráfico");
          exit(0);
     }
     cleardevice();
     setcolor(3);
     circle(200,200,90);
     setcolor(14);
     circle(500,200,90);
     setfillstyle(3,14);
     floodfill(500,200,14);
     rectangle(300,100,400,300);
     getch();
     closegraph();
}
```

ESTILO LINEA

Sirve para establecer el tipo de línea con el que se van a pintar los objetos. Los tipos de línea establecen cinco valores definidos. También hay que especificar el ancho y la separación. El archivo de cabecera necesario es <graphics.h>.

setlinestyle(int tipo,int_separacion,int_ancho);

Valor Línea	Descripción
0	Continua.
1	Guiones.
2	Guiones largos y cortos.
3	Guiones largos.
4	Puntos.

LINEAS

Dibuja una línea desde la posición donde nos encontramos. Primero debemos posicionarnos en la coordenada de inicio y después llamar a la función dando la coordenada final mediante 2 enteros. En la segunda función directamente se le da el comienzo y el final. El archivo de cabecera necesario es <graphics.h>.

moveto(x,y);
lineto(x,y);
line(x1,y1,x2,y2);

EJEMPLO:

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

void main(void)
{
     int tarjeta= DETECT,modo=DETECT;
     int color,coord_y,tipo_linea=0;
```

```
     clrscr();

     initgraph(&tarjeta, &modo, "c:\\tc\\bgi");
     if(graphresult()!=grOk)
     {
          printf("Error en el modo gr fico");
          exit(0);
     }
     cleardevice();
     setcolor(14);
     moveto(350,150);
     lineto(350,250);
     for(coord_y=150;coord_y<=250;coord_y+=25)
     {
          setlinestyle(tipo_linea,1,1);
          line(300,coord_y,400,coord_y);
          tipo_linea++;
     }
     getch();
     closegraph();
}
```

ELIPSES

Dibuja una elipse o arco de elipse. Se le indica la posición mediante los valores x, y. El ángulo inicial (que es cero si es una elipse) y el ángulo final (que es 360 si es una elipse). Hay que indicarle el radio para x y el radio para y. El archivo de cabecera necesario es <graphics.h>.

ellipse(x,y,ang_inicio,ang_fin,radio_x,radio_y);

ARCOS

Dibuja un arco. hay que indicarle la posición, el ángulo inicial y el final y el radio que tiene dicho arco. Todos sus valores son enteros. El archivo de cabecera necesario es <graphics.h>.

arc(x,y,ang_inicial,ang_final,radio);

PUNTOS

Pintan un punto en una coordenada determinada con un color establecido. Hay que indicarle las coordenadas y el color mediante variables o valores enteros. Se pude pintar puntos aleatoriamente. El archivo de cabecera necesario es <graphics.h>.

putpixel(int_x,int_y,int_color);

EJEMPLO: Se pinta un tarrina con pajilla.

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

void main(void)
{
     int tarjeta= DETECT,modo=DETECT;
     clrscr();

     initgraph(&tarjeta, &modo, "c:\\tc\\bgi");
```

```
     if(graphresult()!=grOk)
     {
          printf("Error en el modo gr fico");
          exit(0);
     }
     cleardevice();
     setcolor(2);
     ellipse(400,200,0,360,90,40);
     setcolor(15);
     arc(310,200,0,45,90);
     setcolor(11);
     line(490,200,470,350);
     line(310,200,330,350);
     setcolor(14);
     ellipse(400,350,180,360,70,25);
     getch();
     closegraph();
}
```

SETTEXTSTYLE

Establece el estilo de letra que se va a utilizar en modo gráfico. Hay que indicarle mediante valores enteros el tipo, la dirección y el tamaño que tiene la letra. El color se establece con la función setcolor. El archivo de cabecera necesario es <graphics.h>.

settextstyle(int_tipo,const_direccion,int_tamaño);

TIPO DE LETRAS	
Valor	Descripción
0	**A B C**
1	**A B C**
2	**A B C**
3	A B C
4	ABC
5	A B C
6	A B C
7	*A B C*
8	A B C
9	A B C (tamaño grande).
10	**A B C** (tamaño grande)
11	**ABC**

DIRECCIÓN DEL TEXTO	
CONST.	DESCRIPCIÓN
HORIZ_DIR	En Horizontal.
VERT_DIR	En Vertical.

ALINEAR TEXTO

La función settextjustify() alineara el texto con respecto a las coordenadas donde se muestra el texto. Lo hace en horizontal y en vertical. El archivo de cabecera necesario es <graphics.h>.

settextjustify(const_horizontal,const_vertical);

HORIZONTAL	
CONST.	Descripción
LEFT_TEXT	Alineación izquierda.
CENTER_TEXT	Alineación centrada.
RIGHT_TEXT	Alineación derecha.

VERTICAL	
CONST.	Descripción
TOP_TEXT	Alineación superior.
CENTER_TEXT	Alineación centrada.
BOTTOM_TEXT	Alineación inferior.

VER TEXTO

La función outtextxy(), muestra el texto de una cadena o el contenido de una variable utilizando el modo gráfico. Hay que indicarle la posición mediante dos valores enteros y a continuación la variable o cadena entre comillas. El archivo de cabecera necesario es <graphics.h>.

```
outtextxy(x,y,variable_cadena);
outtextxy(x,y,”textto);
```

EJEMPLO:

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

void main(void)
{
```

```
     int tarjeta= DETECT,modo=DETECT;
     int tipo=0;
     clrscr();

     initgraph(&tarjeta, &modo, "c:\\tc\\bgi");
     if(graphresult()!=grOk)
     {
          printf("Error en el modo gr fico");
          exit(0);
     }
     cleardevice();
     for(tipo=0;tipo<=11;tipo++)
     {
          settextstyle(tipo,HORIZ_DIR,5);
          outtextxy(200,200,"ABC");
          delay(1000);
          cleardevice();
     }
closegraph();
}
```

COPIAR IMAGEN

Existen tres funciones gráficas relacionadas entre si, que permiten copiar y mover objetos gráficos por la pantalla. Las tres tienen la cabecera <graphics.c>. Deben estar siempre las tres para funcionar. El modo de trabajo es: copiar una zona de la pantalla a la memoria (utilizando un puntero), asignar la memoria necesaria y, mostrar la imagen.

GETIMAGE:

Copia el contenido del rectángulo definido por las cuatro coordenadas (tipo int) y lo guarda en memoria mediante un puntero. El archivo de cabecera necesario es <graphics.h>.

```
getimage(x1,y1,x2,y2,puntero);
```

IMAGESIZE:

Devuelve el tamaño en bytes necesarios para contener la región especificada en el rectángulo de la orden getimage().El archivo de cabecera necesario es <graphics.h>.

```
variable=imagesize(x1,y1,x2,y2);
```

PUTIMAGE:

Muestra en pantalla una imagen obtenida por getimage(). El puntero es quien contiene la imagen y la sitúa en la posición dada por x,y. El modo en que se muestra esta definido por 5 constantes. El archivo de cabecera necesario es <graphics.h>.

```
putimage(x,y,puntero,modo);
```

CONST.	Valor	Descripción
COPY_CUT	0	Copia la imagen.
XOR_PUT	1	Borra la imagen anterior.
OR_PUT	2	Mezcla las imágenes.
AND_PUT	3	Muestra encima de la anterior.
NOT_PUT	4	Copia en color inverso.

EJEMPLO: Muestra un circulo dando salto por la pantalla.

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

void main(void)
{
     int tarjeta=DETECT,modo=DETECT,fil,col;
     long tam;
     char *imagen;

     initgraph(&tarjeta,&modo,"c:\\tc\\bgi");
     if(graphresult()!=grOk)
     {
          printf("Error en modo grafico");
          exit(0);
     }
     cleardevice();
     circle(100,100,40);
     tam=imagesize(50,50,150,150);
     imagen=malloc(tam);
     getimage(50,50,150,150,imagen);
```

```
     cleardevice();
     while(!kbhit())
     {
          delay(400);
          putimage(col,fil,imagen,XOR_PUT);
          delay(400);
          putimage(col,fil,imagen,XOR_PUT);
          col=random(getmaxx());
          fil=random(getmaxy());
     }
     closegraph();
}
```

ARCHIVOS

El sistema de archivos de C está diseñado para secuencias que son independientes del dispositivo. Existen dos tipos se secuencias: de texto que es una ristra de caracteres organizados en líneas terminadas por el carácter salto de línea. Secuencia binaria que es una ristra de bytes con una correspondencia uno a uno con los dispositivos. En esencia C trabaja con dos tipos de archivo secuenciales (texto) y binarios (registros).

Todas las operaciones de archivos se realizan a través de llamadas a funciones que están definidas en en archivo de cabecera CONIO.H o IO.H. Esto permite una gran flexibilidad y facilidad a la hora de trabajar con archivos.

Mediante una operación de apertura se asocia una secuencia a un archivo especificado, esta operación de apertura se realiza mediante un puntero de tipo FILE donde están definidas las características del archivo (nombre, estado, posición,…). Una vez abierto el archivo escribiremos y sacaremos información mediante las funciones implementadas y por último cerraremos los archivos abiertos.

ABRIR ARCHIVO

La función fopen abre una secuencia para que pueda ser utilizada y vinculada con un archivo. Después devuelve el puntero al archivo asociado, si es NULL es que se ha producido un error en la apertura. Se utiliza un puntero de tipo FILE para abrir ese archivo. Sirve para los dos tipos de archivos. El archivo de cabecera necesario es <stdio.h>.

```
FILE *nombre_puntero_archivo;
fopen(char_nombre_archivo,char_modo_apertura);
```

MODOS DE APERTURA	
Valor	Descripción
r	Abre un archivo de texto para lectura.
w	Crea un archivo de texto para escritura.
a	Abre un archivo de texto para añadir información.
rb	Abre un archivo binario para lectura.

wb	Crea un archivo binaro para escritura.
ab	Abre un archivo binario para añadir información.
r+	Abre un archivo de texto para lectura / escritura.
w+	Crea un archivo de texto para lectura / escritura.
a+	Abre o Crea un archivo de texto para añadir información.
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

CERRAR ARCHIVO

Una vez terminadas las operaciones de escritura y lectura hay que cerrar la secuencia (archivo). Se realiza mediante la llamada a la función fclose() que cierra un archivo determinado o fcloseall() que cierra todos los archivos abiertos. Estas funciones escriben la información que todavía se encuentre en el buffer y cierra el archivo a nivel de MS-DOS. Ambas funciones devuelve CERO si no hay problemas. El archivo de cabecera necesario es <stdio.h>.

```
int fclose(puntero_archivo);
int fcloseall();
```

EJEMPLO: Prototipo de programa para leer y/o escribir un archivo.

```
#include <stdio.h>

void main(void)
{
     FILE *punt_fich;
     clrscr();

     if((punt_fich=fopen("hla.txt","a"))==NULL)
     {
          printf("Error en la apertura");
          exit(0);
     }
     .
     .
     .
     operaciones lectura/escritura
     .
     .
     .
     fclose(punt_fich);
}
```

ESCRITURA DE CARACTERES Y CADENAS EN SECUENCIALES

FPUTC/PUTC:

Esta función escribe un carácter en el archivo 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. El archivo de cabecera necesario es <stdio.h>.

```
int fputc(*char,puntero_archivo);
int fputc('carácter',puntero_archivo);
int putc(*char,puntero_archivo);
int putc('carácter',puntero_archivo);
```

FPUTS

Esta función escribe el contenido de la cadena puesta como primer parámetro de la función. El carácter nulo no se escribe en el archivo. Si se produce algún error devuelve EOF y si todo va bien devuelve un valor no negativo. El archivo de cabecera necesario es <stdio.h>.

```
int fputs(variable_cadena,puntero_archivo);
int fputs("texto",puntero_archivo);
```

FPRINTF

Esta función escribe en el archivo cualquier tipo de valor, cadenas, números y caracteres. Esta función tiene el mismo formato que printf. Hay que indicarle el puntero, el identificador de formato y nombre de la variables o variables a escribir. El archivo de cabecera necesario es <stdio.h>.

```
fprintf(puntero_archivo,"texto");
fprintf(puntero_archivo"identificador",var);
```

fprintf(puntero_arch“ident(es)_formato”,variable(s));

EJEMPLO: Este programa escribe en un archivo, letra a letra o por frasee.

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

void letra(void);
void frase(void);

FILE *punt_fich;

void main(void)
{
     int opt;
     clrscr();

     if((punt_fich=fopen("hla.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);
     clrscr();
     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,"$"));
}
```

LECTURA DE CARACTERES Y CADENAS EN SECUENCIALES

FGETC/FGET:

Esta función devuelve el carácter leído del archivo e incrementa el indicador de posición del archivo. Si se llega al final del archivo la función devuelve EOF. Todos los valores que lee los transforma a carácter. El archivo de cabecera necesario es <stdio.h>.

```
var_char=fgetc(puntero_archivo);
var_char=getc(puntero_archivo);
```

FGETS:

Esta función lee un determinado número de caracteres de un archivo y los pasa a una variable de tipo cadena. Lee caracteres hasta que encuentra un salto de línea, un EOF o la longitud especificada en la función. Si se produce un error devuelve un puntero NULL. El archivo de cabecera necesario es <stdio.h>.

```
fgets(variable_cadena,tamaño,puntero_archivo);
```

EJEMPLO:

```
#include <stdio.h>

void letra(void);
void frase(void);

FILE *punt_arch;

void main(void)
{
     int opt;
     clrscr();

     if((punt_arch=fopen("hla.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);
     clrscr();
     switch(opt)
     {
          case 1:
               letra();
               break;
          case 2:
```

```
                frase();
                break;
        }

        getch();
        fclose(punt_arch);
}

void letra(void)
{
        char t=0;
        for(;t!=EOF;)
        {
                t=getc(punt_arch);
                printf("%c",t);
        }
}

void frase(void)
{
        char frase[31];

        fgets(frase,30,punt_arch);
        printf("%s",frase);
}
```

POSICION EN ARCHIVOS SECUENCIALES Y BINARIOS

REWIND:

Lleva el indicador de posición al principio del archivo. No devuelve ningún valor. El archivo de cabecera necesario es <stdio.h>.

```
rewind(puntero_archivo);
```

FGETPOS

Guarda el valor actual del indicador de posición del archivo. El segundo término 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. El archivo de cabecera necesario es <stdio.h>.

```
int fgetpos(puntero_archivo,&objeto_fpos_t);
```

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. El archivo de cabecera necesario es <stdio.h>.

```
int fsetpos(puntero_archivo,&objeto_fpos_t);
```

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. El archivo de cabecera necesario es <io.h>.

```
var_long =tell(puntero_archivo);
```

FEOF

Determina el final de un archivo binario. Se utiliza siempre que se realizan consultas, informes y listados, va asociado a un bucle que recorre todo el archivo. El archivo de cabecera necesario es <stdio.h>.

```
feof(puntero_archivo);
```

FSEEK

Sitúa el indicador del archivo en la posición indicada por la variable de tipo long (el segundo termino) desde el lugar que le indiquemos mediante el tercer termino de la función (mediante una constante). Devuelve –1 si hay error, si no hay error devuelve la nueva posición. El archivo de cabecera necesario es <io.h>.

```
var_long=fseek(puntero_archivo,long_despl,int_origen);
```

Constante	Descripción
SEEK_SET	Desde el principio del Archivo
SEEK_CUR	Desde la posición actual.
SEEK_END	Desde el final del Archivo

EJEMPLO:

```
#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);
}
```

ESCRITURA Y LECTURA EN BINARIOS

FWRITE

Escribe los datos de una estructura a un archivo binario e incrementa la posición del archivo en el número de caracteres escritos. Hay que tener en cuenta que el modo de apertura del archivo debe ser binario. El archivo de cabecera necesario es <stdio.h>.

fwrite(&eti_estru,tamaño_estru,nº_reg,punter_archivo);

FREAD

Lee registros de un archivo 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 archivo. Hay que tener en cuenta el modo de apertura del archivo. El archivo de cabecera necesario es <stdio.h>.

fread(&eti_estru,tamaño_estru,nº_reg,punter_archivo);

EJEMPLO: Este programa crea una base de datos de un hospital.

```
#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 archivo");
          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);
     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;
          }
     }
}
```

E/S por Archivos

En C, un archivo puede ser cualquier cosa, desde un archivo de disco hasta un impresora. Mediante una operación de apertura se asocia una secuencia a un archivo específico. Pero... ¿qué es una secuencia?

SECUENCIA

El sistema de archivos de C está diseñado para trabajar con una amplia variedad de dispositivos, incluyendo terminales y controladores

de disco. Aunque cada dispositivo es diferente, el sistema de archivos con buffer transforma cada uno de ellos en un dispositivo lógico llamado secuencia. Cada una de ellas se comporta en forma similar. Debido a que son independientes del dispositivo, la misma función que se usa para escribir en un archivo de disco se puede usar para escribir en otro dispositivo. Existen 2 tipos de secuencias:

- **de Texto**: Es una ristra de caracteres, en la cual se organiza una secuecia de texto terminadas por un caracter de salto de línea, exceptuando en la última línea, en la cual es opcional, pero la mayoría de los compiladores de C no la utilizan. Como a veces existen conversiones de caracteres, no siempre existe una relación uno a uno entre los que se escribe y lo que se almacena (Ejemplo conversión: salto de línea a par retorno de carro y salto de línea).

- **Binarias:** Es una ristra de Bytes con una correspondencia uno a uno con lo almacenado. Sin embargo se puede agregar un número definidos por la implentación, de bytes nulos al final de la secuencia. Estos bute nulos se pueden utilizar, por ejemplo, para ajustar la información

Una vez que hemos abierto el archivo se puede intercambir inforación con éste y el programa. Pero no todos los archivos tienen las mismas posiblidades. Por ejemplo: un archivo de disco permite un acceso directo, mientras que un puerto de modem no. Esto nos indica algo

muy inportante: todas las secuencias son iguales, pero todos los archivos no:
Si el archivo permite *solicitudes de posición*, la apertura del archivo también inicializa el *indicador de posición de archivo* al principio del mismo. A medida que se leen o se escriben caracteres del archivo o en el archivo, el indicador de posición se va incrementando, asegurando así la progresión sobre el archivo.

Todos los archivos se cierran automáticamente cuando el programa termina normalmente, porque el **main()** devuelve el control al sistema operativo, o por una llamada a **exit(),** pero no lo hacen cuando se interrumpe la ejecución del programa de forma anormal.
Cada secuencia asociada a un archivo tiene una estructura de control de tipo **FILE**. Esta estrcutura está definida en el archivo de cabecera STDIO.H.

Funciones más comunes del Sistema de Archivos de ANSI C	
Nombre	Función
fopen()	Abre un archivo.
fclose()	Cierra un archivo.
putc()	Escribe un carácter en un archivo.
fputc()	Idem a putc()
getc()	Lee un carácter de un archivo.
fgetc()	Idem a getc()
fseek()	Busca un byte específico de un archivo.
fprintf()	Hace lo mismo en archivos que printf() en la pantalla.

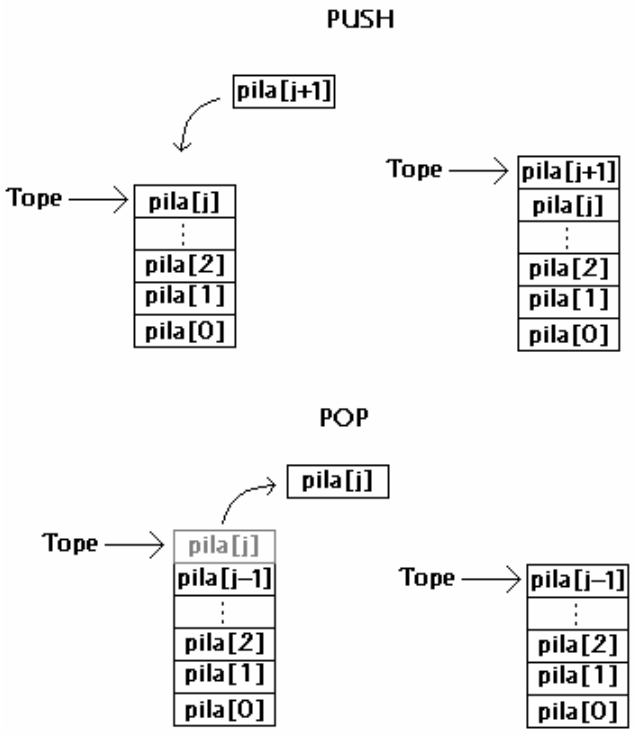
fscanf()	Hace lo mismo en archivos que scanf() en la pantalla.
feof()	Devuelve el valor **true** (Verdadero) si se llega la final del archivo.
ferror()	Devuelve el valor **true** (Verdadero) si se ha producido un error del archivo.
rewind()	Coloca el indicador de posición del archivo al principio del mismo.
remove()	Borra un archivo.
fflush()	Vacía un archivo.

ESTRUCTURAS DE DATOS

Existen ciertas estructuras de datos, comunes a casi todos los lenguajes de programación, que permiten almacenar la información de manera óptima, aplicar algoritmos pre-hechos para lograr ciertas tareas y, en general, manejar distintos tipos de información de manera clara, eficaz y estandarizada. A continuación, algunas de los principales y más usadas estructuras de datos y su implementación en C.

PILAS

Una pila es un estructura del tipo LIFO (“Last In First Out”), sigla inglesa que corresponde a “El primero que sale es el último que entro”. Como su nombre lo indica, una pila es una estructura de datos en la cual se van apilando datos, uno encima de otro, de manera que el primer dato que se saca es el último que se ingresó a la pila. Gráficamente, podemos imaginarnos, por ejemplo, una pila de platos: si vamos colocando, de uno en uno, un plato encima de otro, si queremos sacar un plato, sacaremos siempre el que está en el tope de la pila. Por lo general, a la acción de poner un dato en la pila se le llama comúnmente “push” y a sacar un dato “pop”. Los siguiente gráficos muestran esquemáticamente el procedimiento usado para agregar y obtener un elemento de la pila:



El tope estará apuntado o indexado mediante una variable llamada *Top* (“Tope o parte superior”). Podemos definir la pila estática o dinámicamente. En el caso de definirla estáticamente, se deberá definir con antelación el tamaño máximo (lo llamaremos *MAX*) que puede alcanzar la pila; en caso de no conocer el tamaño máximo que puede alcanzar la pila, deberá usarse asignación dinámica. El proceso *PUSH* puede resumirse en los siguientes pasos:

1.- Verificar que la pila no esté llena (si corresponde).
2.- Agregar el elemento al tope de la lista.
3.- Mover Top al nuevo tope de la lista

El proceso POP puede resumirse en los siguientes pasos:

1.- Verificar que la pila no esté vacía.
2.- Devolver el elemento del Tope de la lista.
3.- Mover Top al elemento anterior de la lista.

A continuación mostraremos varios ejemplos de implementación de una Pila.

EJEMPLO: Pila estática de enteros

```
#define MAX 100   /* La pila puede guardar 10 elementos como
máximo */
#include <stdio.h>
```

```
int pila[MAX];
int Top=-1;            /* Top == -1  --> pila vacía

void push(int);
int pop(void);

void main(void)
{
      int val ;
      push(4) ;
      push(5) ;
      push(6) ;
      val=pop( ) ;
      printf("%d",val);
      val=pop( ) ;
      printf(" %d",val);
}

/*  Guarda un elemento en la pila */

void push(int x)
{
if(Top<(MAX-1)  /* Si hay espacio en la pila */
{
      pila[Top]=x;
      Top++;
}
else                    /* Si no hay espacio */
{
      printf("El valor no pudo ser almacenado, la pila esta
llena");
}
```

```
}

/*  Recupera un elemento de la lista */
int pop(void)
{
      int val;
if(Top>=0)   /* Si la pila no está vacía  */
      {
            Top--;
            return pila[Top+1];
      }
}
```

/* Este programa imprime en pantalla "6 5". */

El ejemplo anterior estaba muy bien si los objetos a guardar eran enteros, pero ¿que ocurriría si en vez de guardar 100 enteros en la pila, hubiésemos querido guardar 100 estructuras como la siguiente?:

EJEMPLO: Pila estática de estructuras; uso de punteros

```
struct FichaPersona
{
      char Nombre[20];
      char Direccion[60];
      int edad;
};
```

*int pila[100] -> 100 enteros * 2 bytes cada uno = 200 bytes*

struct FichaPersona pila[100] → 100 struct FichaPersona →

100(20 char*1 byte +60 char*1byte+1 int*2 bytes)=82*100=8200 bytes*

Lo cual es un tamaño ya grande. Imagínense si quisieran guardar el registro de 1000 personas. Notemos que la pila ocupará todo ese espacio **independientemente de cuán llena esté**. La solución a esto es crear una pila, no como un vector estructuras, sino como un **vector de punteros** que puedan apuntar a ese tipo de estructura:

struct FichaPersona *pila[100];

Esto crea un vector de 100 punteros capaces de apuntar a una estructura del tipo FichaPersona. Recordemos que un puntero es una variable que almacena una **dirección de memoria** y que, por lo tanto, siempre tiene el mismo tamaño, independientemente del tipo de variable al que apunte (2 bytes). Por lo que este vector de punteros tendría un tamaño de sólo 100 * 2 bytes = 200 bytes.

EJEMPLO: Programa para utilizar la estructura recién mencionada.

```
#include <stdio.h>
#include <string.h>
#define MAX 100

struct FichaPersona
{
	char Nombre[20];
	char Direccion[60];
```

```
      int edad;
};

struct FichaPersona *pila[MAX];
int Top= -1;

void push(struct FichaPersona x);
struct FichaPersona pop(void);

void main(void)
{
      struct FichaPersona val1 ;
      struct FichaPersona val2;
      strcpy(val1.Nombre,"Jaime Bond");
      strcpy(val1.Direccion,"Nueva Q Nº007, Conception");
      val1.edad=69;
      push(val1) ;
      val2=pop( ) ;
      printf("%s %s %d", val2.Nombre, val2.Direccion,
val2.edad);
}

/*  Guarda un elemento en la pila */

void push(int x)
{
      if(Top<(MAX-1)  /* Si hay espacio en la pila */
      {
            pila[Top]=&x;   /* &x -> dirección de la
estructura x
            Top++;
      }
```

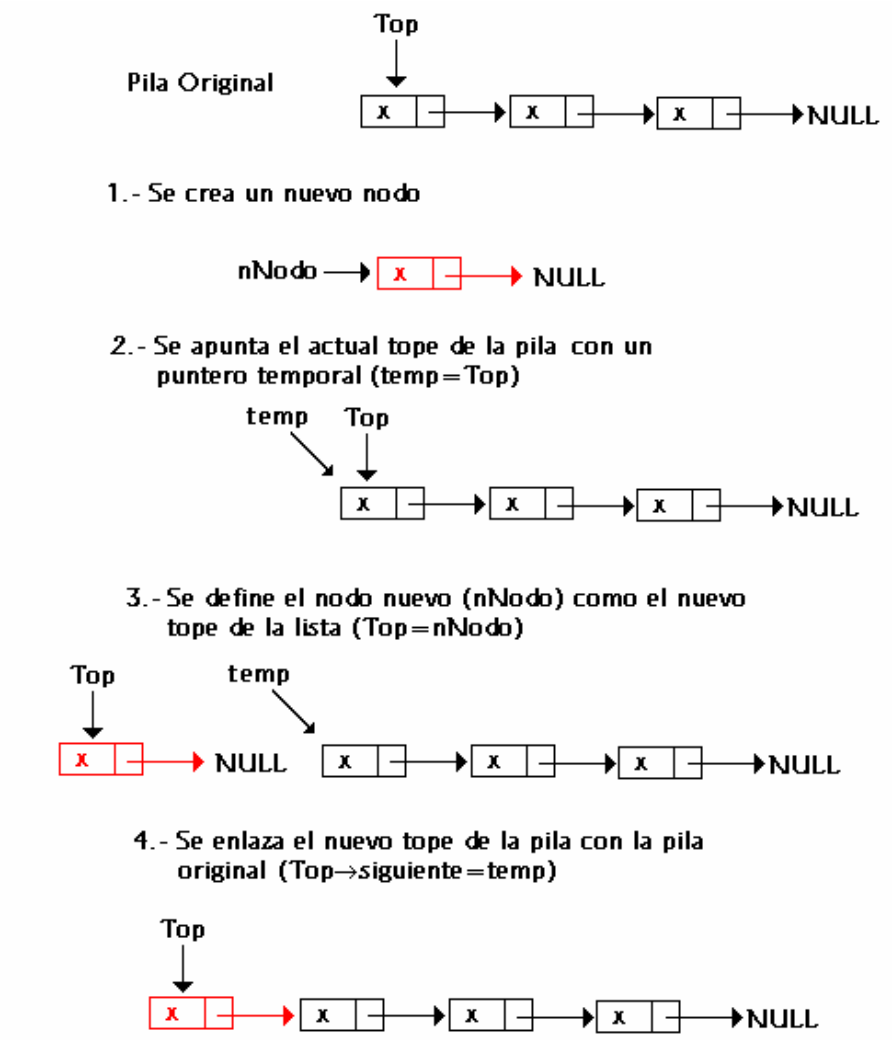
```
      else                    /* Si no hay espacio */
      {
            printf("El valor no pudo ser almacenado, la pila
esta llena");
      }
}

/*  Recupera un elemento de la lista */
int pop(void)
{
      int val;
      if(Top>=0)   /* Si la pila no está vacía */
      {
            Top--;
            return *pila[Top+1];  /*  *pila[Top+1] debido a
que pila[Top+1] es un puntero */
      }
}
```

Si desconocemos el tamaño que puede alcanzar la pila, es necesario usar asignación de memoria dinámica para crear, en tiempo de ejecución, una pila de largo variable. Para esto definiremos una estructura (la llamaremos *nodo*) que almacene, además del valor que deseamos, un campo de enlace (*link*) a otro nodo, de forma que podamos formar una pila a partir de un conjunto de nodos enlazados. El primer nodo estará apuntado por Top (que será un puntero a esta estructura) y el último apuntará a NULL, indicando que no quedan más nodos en la lista.

EJEMPLO:

```
struct Nodo {
	int x;	/* El valor que queremos guardar (pueden ser
varios) */
	struct Nodo *siguiente;
}
```

El procedimiento para agregar o recuperar elementos de la lista está esquematizado en el siguiente dibujo:

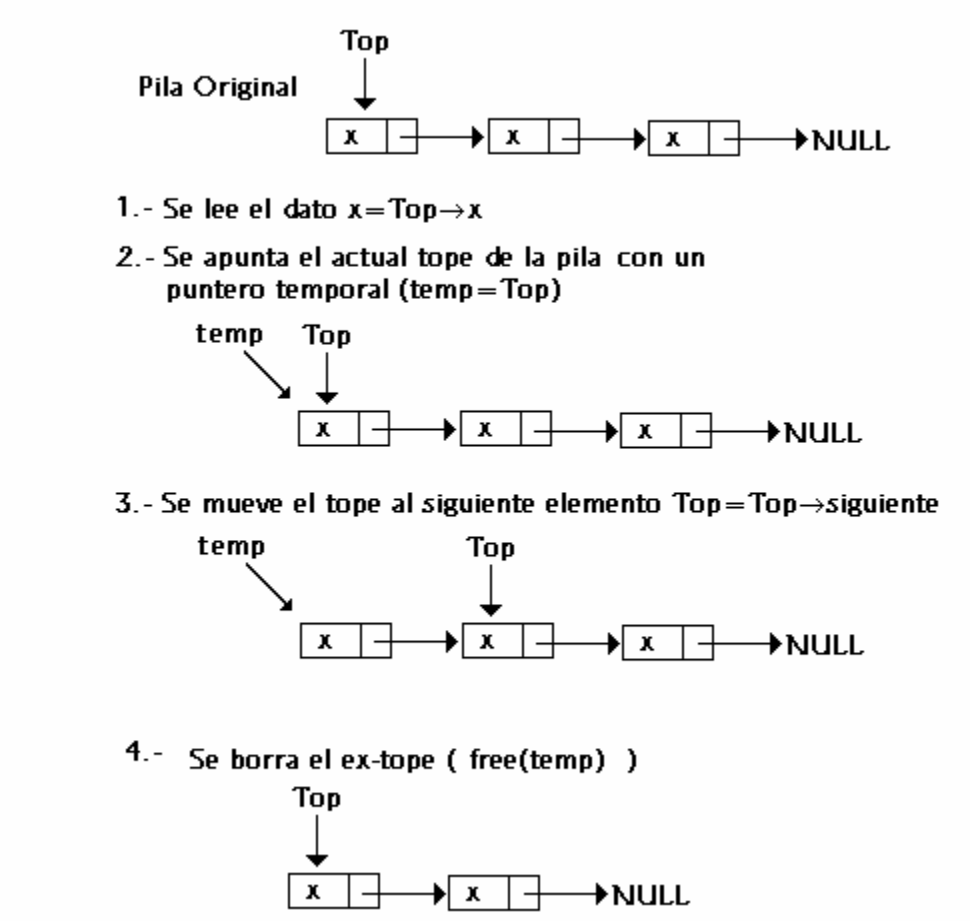


En este dibujo representamos el puntero siguiente como una flecha y la estructura Nodo como un cajón con 2 elementos: uno que deseamos guardar y puntero de enlace. Lo cual, traducido a código sería

```
void push(int x)
{
	struct Nodo *temp;
	struct Nodo *nNodo= (struct Nodo *)malloc(sizeof(struct
Nodo));
```

```
        nNodo.x=x;
        temp=Top;
        Top=nNodo;
        Top->siguiente=temp;
}
```

El proceso de obtención de un valor de la pila está ilustrado en el siguiente diagrama:



Lo cual, traducido a código C es

```
int pop(void)
{
	struct Nodo *temp;
	int val=Top->x;
	temp=Top;
	Top=Top->siguente;
	free(temp);
}
```

COLAS

Una cola, es una estructura parecida a una pila, salvo que su orden de entrada/salida de los datos es FIFO ("first in, first out"), esto es, el primero que entra es el primero que sale. Una cola es similar, por ejemplo, a una fila en un banco, en la que la primera persona en llegar a la fila es la primera en ser atendida y, por ende, la primera en salir de ella.

Ejemplo: Programa ejemplo de implementación de una cola en C. El programa funcionará como un planificador de citas. Las funciones almacena( ) y recupera( ) , guarda y recupera citas, respectivamente.

```
#define MAX 100

char *p[MAX];
int spos=0;
int rpos=0;
```

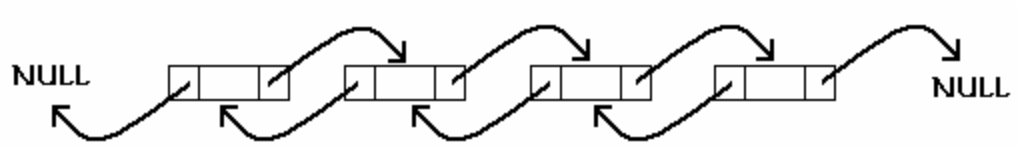
```
/* Guarda una cita */
void alamacena(char *cita)
{
	if ( (spos+1)%MAX==rpos )
	{
	printf("Lista llena\n");
	return;
}
p[spos]=cita;
spos++;
spos=spos%MAX;
}
/* Recupera una cita  */
char *recupera(void)
{
	if(rpos==spos)
	{
		printf("No hay más citas.\n");
		return NULL;
	}
	rpos++;
	rpos=rpos%MAX;
	return p[rpos-1];
}
```

LISTAS ENLAZADAS

Una lista enlazada es aquella en la que los nodos de información están conectados entre sí mediante enlaces, los cuales pueden ser simples o dobles.

Una lista enlazada puede ser usada para construir estructuras de datos como las pilas o las colas. Si, por ejemplo, añadimos datos al principio de la lista y los sacamos también del principio (LIFO) nos encontramos ante una pila (ver el ejemplo anterior). Si añadimos datos al final de la lista y sacamos por el inicio (FIFO), nos encontramos en presencia de una cola (por lo general se usa, aparte del puntero Top (tope de la lista), un puntero Bottom, End, Fin, etc. que apunte al final de la lista, para una rápida incorporación de nuevo elementos.

Sin embargo, la lista no está limitada únicamente a tener el comportamiento de pila o cola, una lista puede eliminar y agregar elementos en cualquier parte de ella. Además, para un recorrido en ambas direcciones, se puede implementar un doble enlace, en el cual cada nodo tiene dos enlaces, en vez de uno, uno que apunta hacia el nodo siguiente, y otro al nodo anterior. La siguiente figura muestra esquemáticamente una lista doblemente enlazada.



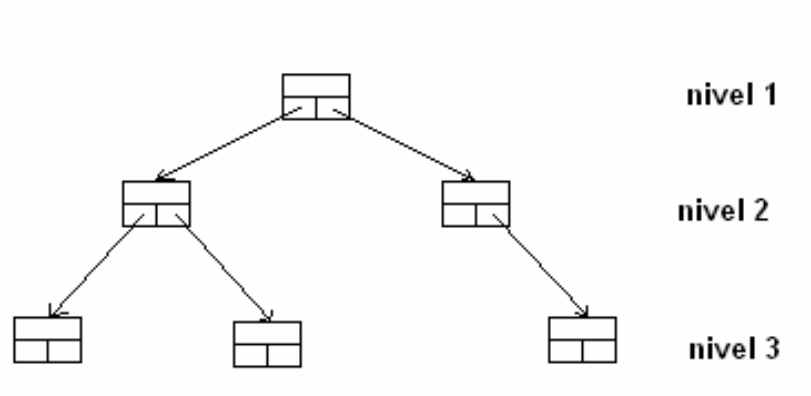
Lista doblemente enlazada

Para un ejemplo de lista enlazada, ver el último ejemplo de la pila.

ÁRBOLES

Cuando hablamos de lista simplemente enlazada y doblemente enlazada, nos limitamos a seguir las direcciones de enlaces ya

explicadas, donde cada nodo tiene un nodo anterior y uno siguiente. Sin embargo, existen otras formas de organizar los nodos, como por ejemplo los árboles. En estos, cada nodo da origen a más de un nodo. La forma más común de utilizar árboles son los llamados árboles binarios, en los cuales cada nodo da origen, a lo más, a dos nodos hijos. El diagrama siguiente muestra un árbol binario, de 3 niveles.



Los árboles binarios, no binarios, balanceados, desbalanceados y otras estructuras complejas de datos, escapan del alcance de este curso, por lo que no serán tratados en mayor profundidad.