# Manual de Visual C+

**http://www.wikilearning.com/manual_de_visual_c+-wkc-3391.htm**

http://www.lawebdelprogramador.com/

## 1. Introducción

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 un determinada tarea. Un lenguaje estructurado permite colocar las sentencias en cualquier parte de una línea. Todos los lenguaje modernos tiende a ser estructurados.

El componente estructural principal de C es la *función*, subrutina independiente que permiten definir tareas de un programa y codificarlas por separado haciendo que los programas sean modulares. Otra forma de estructuración es el *bloque de código* que es un grupo de sentencias conectadas de forma lógica que es tratado como una unidad.

El compilador de C lee el programa entero y lo convierte a código objeto o código máquina. Una vez compilado, las líneas de código fuente dejan de tener sentido durante la ejecución. Este código maquina puede ser directamente ejecutado por la computadora. El compilador de C incorpora una biblioteca estándar que proporciona las funciones necesarias para llevar a cabo las tareas más usuales. En C se puede partir un programa en muchos archivos y que cada uno sea compilado por separado.

En la compilación de un programa consiste en tres pasos. Creación del código fuente, Compilación del programa y Enlace del programa con las funciones necesarias de la biblioteca. La forma en que se lleve a cabo el enlace variará entre distintos compiladores y entornos, pero la forma general es:

## 2. 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.

|| || || 0 1: 32K ||

|| 1 0: 48K ||
|| 1 1: 64K ||
|| 4,5 || Modo inicial de vídeo. || 0 0: no utilizado ||
|| 0 1: 40x25BN adap.color ||
|| 1 0: 80x25BN adap.color ||
|| || || 1 1: 80x25adap.monocromo ||
|| || || 0 0: una ||
|| 6,7 || Número de unidades de disquete. || 0 1: dos ||
|| 1 0: tres ||
|| || || 1 1: cuatro ||
|| 8 || Chip DMA. || 0 ||
|| || || 0 0 0: cero ||
|| || || 0 0 1: uno ||
|| || || 0 1 0: dos ||
|| 9,10,11 || Número de puertos serie. || 0 1 1: tres ||
|| 1 0 0: cuatro ||
|| || || 1 0 1: cinco ||
|| || || 1 1 0: seis ||
|| || || 1 1 1: siete ||
|| 12 || Con adaptador de juegos. || 1 ||
|| 13 || Con módem || 1 ||
|| || || 0 0: cero ||
|| 14,15 || Número de impresoras || 0 1: uno ||
|| 1 0: dos ||
|| 1 1: tres ||

La cabecera que utiliza esta función es . 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:

```
#include

#include

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: 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. Cabecera .

_dos_getdiskfree(unidad,&etiq_struct_diskfree_t);

EJEMPLO:

#include #include

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: 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_getdrive(&var_intera);

EJEMPLO:

#include #include

void main(void)

{ int unidad; clrscr();

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

}

SETDRIVE: La 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_setdrive(var_int_unidad,&var_int_unidades);

EJEMPLO:

#include #include

```
void main(void)

{ unsigned unit; clrscr();

_dos_setdrive(3,&unit); printf("Nº unidades: %u",unit); getch();

}
```

GETCUDIR: Devuelve un entero. La 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. La cabecera es .

```
int getcurdir(int_unidad,cadena);
```

EJEMPLO:

```
#include #include

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 continua 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. Cabecera .

```
int _dos_findnext(&etiqueta_find_t);
```

EJEMPLO:

```
#include #include

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 fichero, y MENOS UNO si se produce algún error. La cabecera .

int remove(variable_cadena);

RENAME: La función cambia el nombre del archivo especificado en primer termino 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. Cabecera .

int rename(var_nombre_antiguo,var_nombre_nuevo);

EJEMPLO:

#include

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

{ clrscr(); if (argc!=3) {

printf("Error en los argumentos");

exit(0);

}

if(remove(argv[1]))

printf("El fichero 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. Cabecera .

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. Cabecera .

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. Cabecera .

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 .

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(). Cabecera .

sound(int_frecuencia);

nosound(); EJEMPLO:

#include #include

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(); }

**3. Primer programa**

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

Forma general de un programa en C:

#Archivos de cabecera #Archivos de cabecera

Declaraciones globales

tipo_devuelto **main**(lista de parámetros) {

secuencia de sentencias; }

tipo_devuelto funcion1(lista de parámetros) {

secuencia de sentencias; }

tipo_devuelto funcion2(lista de parámetros) {

secuencia de sentencias; } . . . tipo_devuelto funcionN(lista de parámetros) {

secuencia de sentencias; }

*EJEMPLO*:

#include

void main(void) {

**clrscr();**

printf("HOLA MUNDO"); }

Este primer programa lo único que hace es mostrar un mensaje en pantalla. Vamos a ir comentando cada una de las líneas (*sentencias*) que aparecen en este programa. La primera línea que aparece se denomina *ENCABEZAMIENTO* y son ficheros que proporcionan información al compilador. En este archivo están incluidas las funciones llamadas por el programa. La ubicación de estos ficheros es el directorio C:\TC\INCLUDE\.

La segunda línea del programa indica donde comienza la función main indicando los valores que recibe y los que devuelve. La primera línea de la función main (clrscr();) limpia la pantalla. y la segunda muestra un mensaje en la pantalla.

## 4. Variables y constantes

Unidad básica de almacenamiento, la creación es la combinación de un identificador, un tipo y un ámbito. Todas las variables en C tienen que ser declaradas antes de ser usadas. El lugar donde se declaran las variables puede ser dentro o en la definición de una función, fuera de todas las funciones.

| TIPOS | RANGO | TAMANO | DESCRIPCION |
|---|---|---|---|
| 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 |

El nombre de las variables conocido como *identificadores* debe cumplir las siguientes normas. La longitud puede ir de un carácter a 31. El primero de ellos debe ser siempre una letra. No puede contener espacios en blanco, ni acentos y caracteres gramaticales. Hay que tener en cuenta que el compilador distingue entre mayúsculas y minúsculas.

*SINTAXIS*: tipo nombre=valor_numerico; tipo nombre='letra'; tipo nombre[tamaño]="cadena de letras", tipo nombre=valor_entero.valor_decimal;

**CONVERSION (*casting*)**: Las conversiones automáticas pueden ser controladas por el programador. Bastará con anteponer y encerrar entre parentesis, el tipo al que se desea convertir. Este tipo de conversiones solo es temporal y la variable a convertir mantiene su valor.

*SINTAXIS*: variable_destino=(tipo)variable_a_convertir; variable_destino=(tipo) (variable1+variable2+variableN);

*EJEMPLO*: convertimos 2 variables float para guardar la suma en un entero.

```
#include

void main(void)

{ float num1=25.75, num2=10.15; int total=0;

clrscr(); total=(int)num1+(int)num2; // total=(int)(num1+num2); printf("Total: %d",total);

getch(); }
```

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

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

*GLOBALES*: Son conocidas a lo largo de todo el programa, y se pueden usar desde cualquier parte del código. Mantienen sus valores durante toda la ejecución. Deban ser declaradas fuera de todas las funciones incluida main(). 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 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*:

tydef tipo nuevo_nombre;

nuevo_nombre nombre_variable[=valor];

*EJEMPLO*:

```
#include
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 globales, que aquí se conocen"); printf("%d %d",numero,numero1);
}
```

**LAS 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

**#define EURO 166.386 #define TEXTO "Esto es una prueba" const int peseta=1;**

void main(void)

{ clrscr(); printf("El valor del Euro es %.3f ptas",**EURO**); printf("\nEl valor de la Peseta es %d pta",**peseta**); printf("\n%s",**TEXTO**); printf(“Ejemplo de constantes y defines”); getch();

}

**5. Operadores**

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

**ASIGNACION *(=)***: En C se puede utilizar el operador de asignación en cualquier expresión valida. Solo con utilizar un signo de igualdad se realiza la asignación. El operador destino (parte izquierda) debe ser siempre una variable, mientras que en la parte derecha puede ser cualquier expresión valida. Es posible realizar asignaciones múltiples, igualar variables entre si y a un valor.

*SINTAXIS*:

variable**=**valor;

variable**=**variable1;

variable**=**variable1**=**variable2**=**variableN**=**valor;

**ARITMÉTICOS**: Pueden aplicarse a todo tipo de expresiones. Son utilizados para realizar operaciones matemáticas sencillas, aunque uniéndolos se puede realizar cualquier tipo de operaciones. En la siguiente tabla se muestran todos los operadores aritméticos.

LÓGICOS Y RELACIONALES: Los operadores relacionales hacen referencia a la relación entre unos valores y otros Los lógicos se refiere a la forma en que esas relaciones pueden conectarse entre si. Los veremos a la par por la estrecha relación en la que trabajan.

A NIVEL DE BITS: Estos operadores son la herramienta más potente, pueden manipular internamente las variables, es decir bit a bit. Este tipo de operadores solo se pueden aplicar a las variables de tipo char, short, int y long. Para manejar los bits debemos conocer perfectamente el tamaño de las variables.

## 6. 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.

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

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

delante antes del especificador de longitud.

printf(“%f ”,numero); //salida normal.

printf(“%10f ”,numero); //salida con 10 espacios.

printf(“%010f “,numero); //salida con los espacios poniendo 0.

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

printf(“%10.4f “,numero); //salida con 10 espacios con 4 decimales.

printf(“%10.15s”,cadena); //salida con 10 caracteres dejando 15 espacios.

printf(“%4.4d”,numero); //salida de 4 dígitos mínimo.

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

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

**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(“%**10**s”,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.

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

## 7. Sentencias de control

Es la manera que tiene un lenguaje de programación de provocar que el flujo de la ejecución avance y se ramifique en función de los cambios de estado de los datos.

**IF-ELSE**: La ejecución atraviese un conjunto de estados bolean que determinan que se ejecuten distintos fragmentos de código.

if (expresion-booleana)if (expresion-booleana) Sentencia1; { else Sentencia1;

Sentencia2; sentencia2;

**} else Sentencia3;**

La cláusula else es opcional, la expresión puede ser de c unan mediante operadores lógicos).Otra opción posible es la utilización de if anidados, es decir unos dentro de otros compartiendo la cláusula else.

Un operador muy relacionado con la sentencia if es **?**. Es un operador ternario que puede sustituir al if. El modo de trabajo es el siguiente, evalúa la primera expresión y si es cierta toma el valor de la segunda expresión y se le pasa a la variable. Si es falsa, toma el valor de la tercera y la pasa a la variable

varible=condicion ? expresion2:expresion3; *EJEMPLO*:

```
#include
void main(void)
{ int peso; clrscr();
gotoxy(5,3);printf(“Introducir edad: “); gotoxy(22,3);scanf(“%d”,&peso);
if(peso<500)
{ gotoxy(5,5); printf(“No es ni media Tn”);
} else {
gotoxy(5,5);
pritnf(“Es más de media Tn”); } getch();
}
```

Para probar la utilización de un if anidado realizar un programa que pida una edad. Si la edad es igual o menor a 10 poner el mensaje NIÑO, si la edad es mayor a 65 poner el mensaje JUBILADO, y si la edad es mayor a 10 y menor o igual 65 poner el mensaje ADULTO.

**SWITCH**: Realiza distintas operaciones en base al valor de una única variable o expresión. Es una sentencia muy similar a if-else, pero esta es mucho más cómoda y fácil de comprender. Si los valores con los que se compara son números se pone directamente el pero si es un carácter se debe encerrar entre comillas simples.

```
switch (expresión){
case valor1: sentencia; break;
case valor2: sentencia; break;
case valor3: sentencia; break;
case valorN: sentencia; break;
default: }
```

El valor de la expresión se compara con cada uno de los literales de la sentencia case si coincide alguno se ejecuta el código que le sigue, si ninguno coincide se realiza la sentencia default (opcional), si no hay sentencia default no se ejecuta nada.

La sentencia break realiza la salida de un bloque de código. En el caso de sentencia switch realiza el código y cuando ejecuta break sale de este bloque y sigue con la ejecución del programa. En el caso que varias sentencias case realicen la misma ejecución se pueden agrupar, utilizando una sola sentencia break.

```
switch (expresión){ case valor1: case valor2: case valor5:
sentencia;
break; case valor3: case valor4:
sentencia; break; default: }
```

*EJEMPLO*:

```
#include void main(void) {
int opcion; clrscr(); pritnf(“1.ALTAS\n”); pritnf(“2.BAJAS\n”); pritnf(“3.MODIFICA\n”); pritnf(“4.SALIR\n”); pritnf(“Elegir opción: “); scanf(“%d”,&opcion);
switch(opcion) {
```

case 1: printf(“Opción Uno”); break;

case 2: printf(“Opción Dos”); break;

case 3: printf(“Opción Tres”); break;

case 4: printf(“Opción Salir”); break;

default:

printf(“Opción incorrecta”); } getch();

} **WHILE**: Ejecuta repetidamente el mismo bloque de código hasta que se cumpla una condición de terminación. Hay cuatro partes en cualquier bucle. *Inicialización, cuerpo, iteración* y *terminación*.

[inicialización;]

while(terminación){ cuerpo; [iteración;]

}

**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.

[inicialización;]

do{ cuerpo; [iteración;]

}while(terminación);

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

#include

void main(void)

{ int n1=0,n2=1,n3; printf("Numero 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.

for (inicio;fin;iteración)for (inicio;fin;iteración) sentencia1; {

**sentencia1; sentencia2; }**

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

#include

void main(void)

{ int n1=0; for (n1=1;n1<=100;n1++)

printf("%d\n",n1); getch(); }

La sentencia **continue** lo que hace es ignorar las sentencias que tiene el bucle y saltar directamente a la condición para ver si sigue siendo verdadera, si es así sigue dentro del

bucle, en caso contrario saldría directamente de el.

La sentencia **exit()** tiene una operatoria más drástica, que la sentencia break, en vez de saltear una iteración, esta abandona directamente el programa dándolo por terminado. Aparte realiza operaciones útiles como cerrar cualquier archivo que estuviera abierto, vaciado de buffers. Se suele utilizar cuando se desea abandonar programas cuando se produce algún error fatal e inevitable.

Otra función relacionada con los bucles es **kbhit()**, lo que hace esta función es la detectar el momento en que se pulsa una tecla (cualquiera). Devuelve 0 cuando NO SE HA PULSADO ninguna tecla, cuando SE PULSA una tecla devuelve cualquier valor DISTINTO DE 0.

*EJEMPLO*: Muestra números hasta que se pulsa una tecla.

```
#include
void main(void)
{
long num=0;
clrscr();
for( ; ; )
{
num++;
printf("%ld\n",num);
if(kbhit())
exit();
}
getch(); }
```

### 8. Arrays y cadenas

Un array es una colección de variables del mismo tipo que se referencian por un nombre en común. A un elemento especifico de un array se accede mediante un índice. Todos los array constan de posiciones de memoria contiguas. La dirección mas baja corresponde al primer elemento. Los arrays pueden tener de una a varias dimensiones.

**UNIDIMENSIONALES**

El array más común en C es la cadena (array de caracteres terminado por un nulo). Todos los array tienen el 0 como primer elemento. Hay que tener muy presente el no rebasar el último índice. 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};
```

*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 void main(void)

{ int x[100]; int cont;

clrscr(); for(cont=0;cont<100;cont++) x[cont]=cont;

printf("%d",x[4]); getch(); }

El uso más común de los arrays unidimensionales es con mucho las **cadenas**, conjunto de caracteres terminados por el carácter nulo ('\0'). Por tanto cuando se quiera declarar una arrays de caracteres se pondrá siempre una posición más. No es necesario añadir explícitamente el carácter nulo el compilador lo hace automáticamente.

*SINTAXIS*: char nombre[tamaño]; char nombre[tamaño]="cadena"; char nombre[]="cadena";

**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]=*Unknown action; the action name must not contain special characters.*;
tipo nomb_array[][]=*Unknown action; the action name must not contain special characters.*;

*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

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 las columnas representa la longitud de cada una de esas cadenas.

*EJEMPLO*: Introduce 10 cadenas y luego las muestra.

```
#include 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(); }
```

**9. Estructuras, uniones y enumeraciones**

C proporciona formas diferentes de creación de tipos de datos propios. Uno de ellos es la agrupación de variables bajo un mismo nombre, Otra es permitir que la misma parte de memoria sea definida como dos o más tipos diferentes de variables y también crear una lista de constantes entera con nombre.

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.

SINTAXIS:

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

. . . . . struct nombre etiqueta1,etiquetaN;

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:

#include 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);printf("Elegir Opcion: "); 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();

}

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 ir elemento por elemento. 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];

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 es similar a 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{union nombre{ var1; var1; var2; var2;

varN

}; varN }etiqueta1,etiquetaN;

. . . 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

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();

}
```

**10. Punteros**

Un puntero es una variable que contiene una dirección de memoria. Esa dirección es la posición de n 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;
```

OPERADORES: El & devuelve la dirección de memoria de su operando. El * devuelve el valor de la variable localizada en la dirección que sigue. 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.

```
m= &cuenta;

q=*m;
```

EJEMPLO:

```
#include

void main(void)

{ int *p; int valor;

clrscr(); printf("Intruducir 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.

```
#include void main(void) {

int x; int *p1, *p2;

p1=&x; p2=p1;

printf(“%p”,p2);
```

}ARITMÉTICA: Dos operaciones aritméticas que se pueden usar como punteros, la suma y la resta. Cuando se incrementa un puntero, apunta a la posición de memoria del siguiente elemento de su tipo base. Cada vez que se decrementa, apunta a la posición del elemento anterior.

```
p1++; p1--; p1+12;
```

COMPARACIÓN: Se pueden comparar dos punteros en una expresión relacional. Generalmente, la comparación de punteros se utiliza cuando dos o más punteros apuntan a un objeto común.

```
if(p
```

EJEMPLO:

```
#include > #include

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); }

(sigue) 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);
```

}

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 los arrays que hemos visto anteriormente, 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

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 la dirección del primer elemento del array y así se conoce el comienzo del array, el final lo sabemos mediante el nulo que tienen todas las cadenas al final. Con esta relación se puede declarar una cadena sin tener en cuenta la longitud de la cadena. Otra característica importe es la posibilidad es poder pasar cadenas a las funciones.

SINTAXIS:

char *nombre;

char *nombre[];

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

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 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. Hay ocasiones en que un programa necesita usar cantidades de memoria variable.

La memoria se obtiene del montón (región de la memoria libre que queda entre el programa y la pila). El tamaño del montón es desconocido pero contiene gran cantidad de memoria. El sistema de asignación dinámica esta compuesto por las funciones malloc()que asigna memoria a un puntero y free()que libera la memoria asignada. Ambas funciones necesitan el archivo de cabecera stdlib.h.

SINTAXIS:

puntero=malloc(numero_de_bytes);

free(puntero);

EJEMPLO:

#include #include

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: Que un puntero apunte a otro puntero que apunte a un valor de destino. 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;

**11. Funciones**

Las funciones son los bloques constructores de C, es el lugar donde se produce toda la actividad del programa. Es la característica más importante de C. Subdivide en varias tareas concurrentes el programa, así sólo se las tendrá que ver 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.

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. El código de una función es privado a esa función, el código que comprende el cuerpo de esa función está oculto al resto del programa a menos que se haga a través de una llamada. Todas las variables que se definen dentro de una función son locales con la excepción de las variables estáticas.

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.

Al llamar a una función se puede hacer la llamada por valor o por referencia. 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 ni 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 parentesis.

void nombre_funcion(void);

nombre_funcion();

EJEMPLO:

#include

void mostrar(void);

void main(void)

{

```
clrscr();

printf("Estoy en la principal\n");

mostrar();

printf("De vuelta en la principal"); getch(); }

void mostrar(void)

{

printf("Ahora he pasado a la función\n"); delay(2000);

}
```

SEGUNDO TIPO: Son funciones que devuelven un valor una vez han terminado de realizar sus operaciones, solo se puede 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 llamo a esa función. Al compilador hay que indicarle 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:

uno de ellos. Se le indica poniéndolo en los parentesis que tienen las funciones. Deben ser los mismos que en el prototipo. void nombre_funcion(tipo1,tipo2…tipoN);

```
nombre_funcion(var1,var2…varN);
```

EJEMPLO:

```
#include

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

void main(void)

{ int a,b; char cadena[20]; clrscr(); printf("Resta de Valores\n"); printf("Introducir valores y cadena: "); scanf("%d%*c%d",&a,&b); gets(cadena);

resta(a,b,cadena);

getch(); }

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

{ printf(“total= %d”,x+y); printf(“La cadena es: %s”,cad);

} PASO DE ESTRUCTURAS A FUNCIONES: Cuando se utiliza o se pasa una estructura como argumento a una función, se le pasa toda la estructura integra, se le pasa por valor. Esto significa que todos los cambios realizados en el interior de la función no afectan a la estructura utilizada como argumento. La única diferencia con los casos anteriores es que, ahora hay que definir como parámetro de la función una estructura.

SINTAXIS:

tipo_devuelto nombre_funcion(struc nombre etiqueta);

nombre_funcion(etiqueta_estructura);

EJEMPLO:

#include

struct ficha{ char nombre[20]; int edad;

};

void funcion(struct ficha emple);

void main(void)

{ struct ficha trabaja; 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 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;

}LA FUNCIÓN MAIN: Método para pasar información a la función main mediante el uso de argumentos, el lugar desde donde se pasan esos valores es la línea de ordenes del sistema operativo. Se colocan detrás del nombre del programa.

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 ordenes 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

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:

```
int fact(int numero) {

int resp;

if(numero
1) return 1;
```

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

 return(resp); }
```

## 12. Proyectos

Procedimiento con el que se compilan de forma independiente distintas partes de un programa y luego se enlazan para formar el programa ejecutable final. Se utiliza con aplicaciones muy grandes. Los proyectos tienen la extensión PRJ. Solo se genera un ejecutable, los ficheros fuente generan ficheros OBJ.
Lo primero es crear el proyecto (que es el que va a contener los ficheros fuente). El proyecto va a tener la extensión PRJ. El modo de crearlo es:
MENU PROJECT OPCION OPEN PROJECT Asignar un nombre al proyecto.

Lo siguiente es crear cada uno de los ficheros fuente que van a componer el proyecto. La creación de los ficheros es como siempre, no hay nada de particular. Cuando tengamos los ficheros hay que añadirlos al proyecto. El modo de hacerlo.
Seleccionar la ventana del proyecto (ventana inferior de color gris).
MENU PROJECT OPCION ADD ITEM Ir seleccionando cada uno de los ficheros que forman el proyecto

Cuando se tengan todos seleccionados se pulsa el botón DONE.
El siguiente paso es COMPILAR CADA UNO DE LOS FICHEROS FUENTE. Para hacerlo hay que seleccionar primero el fichero y luego pulsar ALT+F9. Así con todos los ficheros. Posteriormente se tiene que linkar el proyecto. MENU COMPILE

OPCION LINK. Cuando se termine con el proyecto hay que cerrarlo. MENU PROJECT OPCION CLOSE.

EJEMPLO: En este proyecto hay dos ficheros fuente. En el primero se muestra 2 valores y se hace la llamada a funciones del segundo programa.
Fichero: UNO.C Fichero:DOS.C

```
#include #include> #include #include
void mostrar(); void sumar() void sumar(); {
void main(void) int x,y; { printf("%d",x+y); int x,y; getch(); clrscr(); }
x=10; y=20; printf("%d %d",x,y); void mostrar()
{
getch(); mostrar(); int z,p; sumar(); z=15;
p=25; printf("%d %d",z,p); getch();
}
getch();
}
```

## 13. 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.

ISALPHA: Devuelve un entero. DISTINTO DE CERO si la variable es una letra del alfabeto, en caso contrario devuelve cero. Cabecera: .

```
int isalpha(variable_char);
```

ISDIGIT: Devuelve un entero. DISTINTO DE CERO si la variable es un número (0 a 9), en caso contrario devuelve cero. Cabecera .

```
int isdigit(variable_char);
```

ISGRAPH: Devuelve un entero. DISTINTO DE CERO si la variable es cualquier carácter imprimible distinto de un espacio, si es un espacio CERO. Cabecera .

```
int isgraph(variable_char);
```

ISLOWER: Devuelve un entero. DISTINTO DE CERO si la variable esta en minúscula, en caso contrario devuelve cero. Cabecera .

```
int islower(variable_char);
```

ISPUNCT: Devuelve un entero. DISTINTO DE CERO si la variable es un carácter de puntuación, en caso contrario, devuelve cero. Cabecera

```
int ispunct(variable_char);
```

ISUPPER: Devuelve un entero. DISTINTO DE CERO si la variable esta en mayúsculas, en caso contrario, devuelve cero. Cabecera

```
int isupper(varible_char);
```

EJEMPLO: Cuenta el numero 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

#include

void main(void)

{

int ind,cont_num=0,cont_text=0;

char temp;

char cadena[6]; clrscr();

printf("Introducir 5 caracteres: "); gets(cadena);

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 funciones nos va a dar información completa del valor que contiene la variable.

```
#include #include

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: Inicializa una región de memoria (buffer) con un valor determinado. Se utiliza principalmente para inicializar cadenas con un valor determinado. Cabecera .

```
memset (var_cadena,'carácter',tamaño);
```

STRCAT: Concatena cadenas, es decir, añade la segunda cadena a la primera, la primera cadena no pierde su valor origina. Lo único que hay que tener en cuenta es que la longitud de la primera cadena debe tener la longitud suficiente para guardar la suma de las dos cadenas. Cabecera .

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

STRCHR: Devuelve un puntero a la primera ocurrencia del carácter especificado en la

cadena donde se busca. Si no lo encuentra, devuelve un puntero nulo. Cabecera . strchr(cadena,'carácter'); strchr("texto",'carácter');

STRCMP: 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. Muy utilizado para comprobar contraseñas. Cabecera .

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

|| RESULTADO ||
|| VALOR DESCRIPCIÓN ||
|| Menor a Cero Cadena1 menor a Cadena2. ||
|| Cero Las cadenas son iguales. ||
|| Mayor a Cero Cadena1 mayor a Cadena2. ||

STRCPY: Copia el contenido de la segunda cadena en la primera. El contenido de la primera se pierde. Lo único que debemos contemplar es que el tamaño de la segunda cadena sea menor o igual a la cadena donde la copiamos. Cabecera .

strcpy(cadena1,cadena2);

strcpy(cadena1,"texto");

STRLEN: Devuelve la longitud de la cadena terminada en nulo. El carácter nulo no se contabiliza. Devuelve un valor entero que indica la longitud de la cadena. Cabecera .

variable=strlen(cadena);

STRNCAT: Concatena el número de caracteres de la segunda cadena en la primera cadena. La primera no pierde la información. Hay que controlar que la longitud de la primera cadena debe tener longitud suficiente para guardar las dos cadenas. Cabecera .

strncat(cadena1,cadena2,nº de caracteres);

STRNCMP: Compara alfabéticamente un número de caracteres entre dos cadenas. Devuelve un entero según el resultado de la comparación. Los valores devueltos son los mismos que en la función strcmp. Cabecera .

strncmp(cadena1,cadena2,nº de caracteres);

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

STRNCPY: Copia un número de caracteres de la segunda cadena a la primera. En la primera cadena se pierden aquellos caracteres que se copian de la segunda. Cabecera .

strncpy(cadena1,cadena2,nº de caracteres);

strncpy(cadena1,"texto",nº de caracteres); STRRCHR: Devuelve un puntero a la última ocurrencia del carácter buscado en la cadena. Si no lo encuentra devuelve un puntero nulo. Cabecera .

strrchr(cadena,'carácter');

strrchr(“texto”,’carácter’);

STRPBRK: 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. Cabecera .

strpbrk(“texto”,”cadena_de_busqueda”);

strpbrk(cadena,cadena_de_busqueda);

TOLOWER: Devuelve el carácter equivalente al de la variable en minúsculas, si la variable es una letra y la deja como esta si la letra esta en minúsculas. Cabecera .

variable_char=tolower(variable_char);

TOUPPER: Devuelve el carácter equivalente al de la variable en mayúsculas, si la variable es una letra y la deja como esta si la letra esta en minúsculas. Cabecera .

variable_char=toupper(variable_char);

EJEMPLO: Copia, concatena, mide e inicializa cadenas.

#include #include

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 #include

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 el cuadro de la izquierda busca la primera coincidencia y muestra la cadena a partir de esa coincidencia. En el cuadro de la derecha busca la última coincidencia y muestra la cadena a partir de ese punto.

```
#include #include

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(); }
```

### 14. Funciones matemáticas

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

ACOS: Devuelve un tipo double. Muestra el arcocoseno de la variable. 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. Cabecera .

```
double acos(variable_double);
```

ASIN: Devuelve un tipo double. Muestra el arcoseno de la variable. 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. Cabecera .

```
double asin(variable_double);
```

ATAN: Devuelve un tipo double. Muestra el arcotangente de la variable. 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. Cabecera .

double atan(variable_double);

COS: Devuelve un tipo double. Muestra el coseno de la variable. La variable debe ser de tipo double y debe estar expresada en radianes. Cabecera .

double cos(variable_double_radianes);

SIN: Devuelve un tipo double. Muestra el seno de la variable. La variable debe ser de tipo double y debe estar expresada en radianes. Cabecera .

double sin(variable_double_radianes);

TAN: Devuelve un tipo double. Muestra la tangente de la variable. La variable debe ser de tipo double y debe estar expresada en radianes. Cabecera .

double tan(variable_double_radianes);

COSH: Devuelve un tipo double. Muestra el coseno hiperbólico de la variable. 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. Cabecera debe ser .

double cosh(variable_double);

SINH: Devuelve un tipo double. Muestra el seno hiperbólico de la variable. 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. Cabecera debe ser .

double sinh(variable_double);

TANH: Devuelve un tipo double. Muestra la tangente hiperbólico de la variable. 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. Cabecera debe ser .

double tanh(variable_double);

EJEMPLO:

#include #include

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: Devuelve un double que representa el menor entero sin ser menor de la variable redondeada. Por ejemplo, dado 1.02 devuelve 2.0. Si asignamos –1.02 devuelve –1. En resumen redondea la variable a la alta. Cabecera .

double ceil(variable_double);

FLOOR: Devuelve un double que representa el entero mayor que no sea mayor a la variable redondeada. Por ejemplo dado 1.02 devuelve 1.0. Si asignamos –1.2 devuelve –2. En resumen redondea la variable a la baja. Cabecera .

double floor(variable_double);

FABS: Devuelve un valor float o double. Devuelve el valor absoluto de una variable float. Se considera una función matemática, pero su cabecera es .

var_float fabs(variable_float);

LABS: Devuelve un valor long. Devuelve el valor absoluto de una variable long. Se considera una función matemática, pero su cabecera es .

var_long labs(variable_long);

ABS: Devuelve un valor entero. Devuelve el valor absoluto de una variable int. Se considera una función matemática, pero su cabecera es .

var_float abs(variable_float);

MODF: 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. La variables tienen que ser obligatoriamente de tipo double. Cabecera .

var_double_deimal= modf(variable,var_parte_entera);

POW: Devuelve un double. Realiza la potencia de un número base elevado a un exponente que nosotros le indicamos. Se produce un error si la base es cero o el exponente es menor o igual a cero. La cabecera es .

var_double=pow(base_double,exponente_double);

SQRT: Devuelve un double. Realiza la raíz cuadrada de la variable. La variable no puede ser negativa, si lo es se produce un error de dominio. La cabecera .

var_double=sqrt(variable_double);

EJEMPLO:

#include #include

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: Devuelve un double. Realiza el logaritmo natural (neperiano) de la variable. Se produce un error de dominio si la variable es negativa y un error de rango si es cero. Cabecera .

```
double log(variable_double);
```

LOG10: Devuelve un valor double. Realiza 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. La cabecera que utiliza es .

```
double log10(var_double);
```

RAMDOMIZE(): Inicializa la semilla para generar números aleatorios. Utiliza las funciones de tiempo para crear esa semilla. Esta función esta relacionada con random. Cabecera .

```
void randomize();
```

RANDOM: Devuelve un entero. Genera un número entre 0 y la variable menos uno. Utiliza el reloj del ordenador para ir generando esos valores. El programa debe llevar la función randomize para cambiar la semilla en cada ejecución. Cabecera .

```
int random(variable_int); EJEMPLO: Genera seis números aleatorios entre 1 y 49. No se repite ninguno de ellos.

#include #include

void main(void) {

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

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

for(;;)
```

```
 { 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(); }
```

**15. 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: Devuelve un entero. Esta función convierte la cadena en un valor entero. La cadena debe contener un número entero 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, la función los ignora. La cabecera es .

```
int atoi(variable_char);
```

ATOL: Devuelve un long. Esta función convierte la cadena en un valor long. La cadena 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, la función los ignora. La cabecera es .

```
long atol(variable_char);
```

ATOF: Devuelve un double. Esta función convierte la cadena 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, la función los ignora. La cabecera es .

```
double atof(variable_char);
```

SPRINTF: Devuelve una cadena. Esta función convierte cualquier tipo numérico a cadena. Para convertir de número a cadena hay que indicar el tipo de variable numérica y tener presente que la longitud de la cadena debe poder guardar la totalidad del número. Admite también los formatos de salida, es decir, que se puede coger distintas partes del número. La cabecera es .

sprintf(var_cadena,”identificador”,var_numerica);

ITOA: Devuelve una cadena. La función convierte un entero en su cadena equivalente y sitúa el resultado en la cadena definida en segundo termino de la función. Hay que asegurarse que la cadena se lo suficientemente grande para guardar el número. Cabecera .

itoa(var_entero,var_cadena,base);

|| BASE 2 8 10 16 || DESCRIPCIÓN Convierte el valor en binario. Convierte el valor a Octal. Convierte el valor a decimal. Convierte el valor a hexadecimal. ||

LTOA: Devuelve una cadena. La función convierte un long en su cadena equivalente y sitúa el resultado en la cadena definida en segundo termino de la función. Hay que asegurarse que la cadena se lo suficientemente grande para guardar el número. Cabecera .

ltoa(var_long,var_cadena,base);

EJEMPLO:

#include #include

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(); }

**16. 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. Cabecera .

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

CTIME: Devuelve un puntero a una cadena con un formato día semana mes hora:minutos:segundo año \n\0. La hora del sistema se obtiene mediante la función time. Cabecera .

puntero=ctime(&etiqueta_estructura_time_t);

EJEMPLO:

#include #include

void main(void)

{ time_t fecha_hora; clrscr();

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

}

GETTIME: Devuelve la hora del sistema. 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. Cabecera .

_dos_gettime(&etiqueta_estructura_dostime_t);

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

}

EJEMPLO:

#include #include

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. 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. Cabecera .

_dos_getdate(&etiqueta_estructura_dosdate_t);

struct dosdate_t{ unsigned day; unsigned month; unsigned year; unsigned dayofweek;

}

EJEMPLO:

#include #include

void main(void)

{ struct dosdate_t fec; clrscr();

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

}

SETTIME: Permite cambiar la hora del sistema. 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. Cabecera .

_dos_settime(&etiqueta_estructura_dostime_t);

SETDATE: Permite cambiar la fecha del sistema. 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. Cabecera .

_dos_setdate(&etiqueta_estructura_dosdate_t);

EJEMPLO:

#include #include

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: Devuelve un double. La función devuelve la diferencia, en segundos, entre una hora inicial y hora final. Se restara la hora final menos la hora inicial. Tenemos que obtener la hora al iniciar un proceso y al terminar el proceso volveremos a tomar la hora. Cabecera .

double difftime(hora_final,hora_inicial);

EJEMPLO:

#include #include

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();

}

**17. 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 modo gráfico es el pixel. Otro aspecto es que en modo gráfico 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 fichero fuente con funciones gráficas tenemos primero que indicar al compilador que se va a trabajar con ese tipo de funciones. OPTIONS

LINKER

LIBRARIES

Activar la opción Graphics Library

LISTA DE COLORES:

|| VALOR 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 nº+128 || TEXTO COLOR Negro. Azul. Verde. Cían. Rojo. Magenta. Marrón. Gris claro. Gris oscuro. Azul Claro. Verde claro. Cían claro Rojo claro. Magenta claro Amarillo. Blanco. Parpadean. ||
|| FONDO DEL TEXTO VALOR COLOR 0 Negro. 1 Azul. 2 Verde. 3 Cían. 4 Rojo. 5 Magenta. 6 Marrón. ||
|| FONDO GRÁFICO ||
|| 0 VALOR || Negro. COLOR ||
|| 1 || Azul. ||
|| 2 || Verde. ||
|| 3 || Cían. ||
|| 4 || Rojo. ||
|| 5 || Magenta. ||
|| 6 || Marrón. ||
|| 7 || Gris claro. ||

|| 8 || Gris oscuro. ||
|| 9 || Azul Claro. ||
|| 10 || Verde claro. ||
|| 11 || Cían claro ||
|| 12 || Rojo claro. ||
|| 13 || Magenta claro ||
|| 14 || Amarillo. ||
|| 15 || Blanco. ||

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 .

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 .

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 .

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 .

_setcursortype(constante_tipo_cursor);

EJEMPLO:

#include #include

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. La cabecera .

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. Cabecera .

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. Cabecera .

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. Cabecera .

setbkcolor(int_color);

EJEMPLO:

#include #include #include

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. Cabecera .

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. Cabecera .

setfillstyle(int_estilo,int_color);

|| 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. La cabecera .

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. Cabecera .

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. Cabecera .

rectangle(int_x1,int_y1,int_x2,int_y2);

EJEMPLO:

#include #include #include

void main(void)

{ int tarjeta= DETECT,modo=DETECT; int color;

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

printf("Error en el modo gr fico"); 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. Cabecera .

setlinestyle(int tipo,int_separacion,int_ancho);

|| VALOR LINEA 0 1 2 3 4 || DESCRIPCIÓN Continua. Guiones. Guiones largos y cortos. Guiones largos. 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. Cabecera .

moveto(x,y); lineto(x,y);

line(x1,y1,x2,y2);

EJEMPLO:

#include #include #include

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. Cabecera .

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. Cabecera .

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. Cabecera .

putpixel(int_x,int_y,int_color);

EJEMPLO: Se pinta un tarrina con pajilla.

#include #include #include

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. Cabecera .

settextstyle(int_tipo,const_direccion,int_tamaño);

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. Cabecera .

settextjustify(const_horizontal,const_vertical);

HORIZONTAL

VERTICAL

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. Cabecera .

outtextxy(x,y,variable_cadena);

outtextxy(x,y,”textto); EJEMPLO:

#include #include #include

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 . 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. Cabecera .

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(). Cabecera .

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. Cabecera .

putimage(x,y,puntero,modo);

|| CONST. COPY_CUTXOR_PUTOR_PUTAND_PUTNOT_PUT || VALOR 0 1 2 3 4 || DESCRIPCIÓN Copia la imagen. Borra la imagen anterior. Mezcla las imágenes. Muestra encima de la anterior. Copia en color inverso. ||

EJEMPLO: Muestra un circulo dando salto por la pantalla.

#include #include #include

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(); }

## 18. Ficheros

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

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 fichero (nombre, estado, posición,…). Una vez abierto el fichero escribiremos y sacaremos información mediante las funciones implementadas y por último cerraremos los ficheros abiertos.

ABRIR FICHERO: 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 fichero. Sirve para los dos tipos de ficheros. Cabecera .

FILE *nombre_puntero_fichero;

fopen(char_nombre_archivo,char_modo_apertura);

|| VALOR r w a rb wb ab r+ w+ a+ r+b w+b a+b || MODOS DE APERTURA DESCRIPCIÓN Abre un archivo de texto para lectura. Crea un archivo de texto para escritura. Abre un archivo de texto para añadir información. Abre un archivo binario para lectura. Crea un archivo binaro para escritura. Abre un archivo binario para añadir información. Abre un archivo de texto para lectura / escritura. Crea un archivo de texto para lectura / escritura. Abre o Crea un archivo de texto para añadir información. Abre un archivo binario para lectura / escritura. Crea un archivo binaro para lectura / escritura. Abre o Crea un archivo binario para añadir información ||

CIERRE FICHERO: 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 fichero determinado o fcloseall() que cierra todos los archivos abiertos. Estas funciones escribe la información que todavía se encuentre en el buffer y cierra el archivo a nivel de MS-DOS. Ambas funciones devuelve CERO si no hay problemas. La cabecera que utilizan es .

int fclose(puntero_fichero);

int fcloseall();

EJEMPLO:

```
#include

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: Escribe un carácter en el fichero abierto por el puntero que se pone como parámetro. Si todo se produce correctamente la función devuelve el propio carácter, si hay algún error devuelve EOF. La cabecera que utiliza es .

int fputc(variable_char,puntero_fichero);

int fputc('carácter',puntero_fichero);

int putc(variable_char,puntero_fichero);

int putc('carácter',puntero_fichero);

FPUTS: Escribe el contenido de la cadena puesta como primer parámetro de la función. El carácter nulo no se escribe en el fichero. Si se produce algún error devuelve EOF y si todo va bien devuelve un valor no negativo. La cabecera .

int fputs(variable_cadena,puntero_fichero);

int fputs("texto",puntero_fichero);

FPRINTF: Escribe en el fichero 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. Cabecera .

fprintf(puntero_fichero,"texto");

fprintf(puntero_fichero"identificador",var);

fprintf(puntero_fich"ident(es)_formato",variable(s));

EJEMPLO:

#include #include

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();

 (SIGUE) 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: Devuelve el carácter leído del fichero e incrementa el indicador de posición del archivo. Si se llega al final del fichero la función devuelve EOF. Todos los valores que lee los transforma a carácter. Cabecera es .

```
var_char=fgetc(puntero_fichero);

var_char=getc(puntero_fichero);
```

FGETS: Lee un determinado número de caracteres de un fichero 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. La Cabecera es .

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

EJEMPLO:

```
#include

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

FILE *punt_fich;

void main(void)

{ int opt; clrscr();

if((punt_fich=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_fich); }

(sigue) void letra(void) {

char t=0;

for(;t!=EOF;)

{ t=getc(punt_fich); printf("%c",t);

} }

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

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

POSICION EN FICHEROS SECUENCIALES Y BINARIOS

REWIND: Lleva el indicador de posición al principio del archivo. No devuelve ningún valor. La cabecera que utiliza es .

```
rewind(puntero_fichero);
```

FGETPOS: Guarda el valor actual del indicador de posición del archivo. El segundo termino es un objeto del tipo fpos_t que guarda la posición. El valor almacenado sólo es valido para posteriores llamadas a fsetpos. Devuelve DISTINTO DE CERO si se produce algún error y CERO si todo va bien. Cabecera .

int fgetpos(puntero_fichero,&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. Cabecera .

int fsetpos(puntero_fichero,&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. Cabecera .

var_long =tell(puntero_fichero);

FEOF: Determina el final de un fichero binario. Se utiliza siempre que se realizan consultas, informes y listados, va asociado a un bucle que recorre todo el fichero. Cabecera .

feof(puntero_fichero);

FSEEK: Sitúa el indicador del archivo en la posición indicada por la variable de tipo long (en 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. La Cabecera es .

var_long=fseek(puntero_fichero,long_despl,int_origen);

EJEMPLO:

#include #include

void main(void)

{ FILE *punte; clrscr();

if((punte=fopen("hla.txt","r") {

printf("Error de lectura"); exit(0); }

fseek(punte,7,SEEK_SET); printf("%c",fgetc(punte));

getch(); fclose(punte); }

)
NULL)

ESCRITURA Y LECTURA EN BINARIOS

FWRITE: Escribe los datos de una estructura a un fichero binario e incrementa la posición del archivo en el número de caracteres escritos. Hay que tener en cuenta que el modo de apertura del fichero debe ser binario Cabecera .

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

FREAD: Lee registros de un fichero binario, cada uno del tamaño especificado en la función y los sitúa en la estructura indicada en primer termino de la función. Además de leer los registros incrementa la posición del fichero. Hay que tener en cuenta el modo de apertura del fichero. Cabecera .

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

EJEMPLO:

#include #include

void altas(void); void muestra(void); FILE *fich; struct ficha{

int código; char nombre[25]; char direcion[40]; int edad;

}cliente;

void main(void) { char opcion;

if((fich=fopen("gestion.dat","a+b"))
NULL)
{ printf("Error al crear fichero"); exit(0);

}

do

{ clrscr(); printf("Altas\n"); printf("Consulta\n"); printf("Salir\n\n"); printf("Elegir opcion: ");
scanf("%c",&opcion); fflush(stdin); switch(toupper(opcion)) {

case 'A': altas(); break;

case 'C': muestra(); break;

} }while(toupper(opcion)!='S'); fclose(fich);

}

void altas(void)

{ clrscr(); printf("Código: "); scanf("%d",&cliente.codigo); fflush(stdin);

printf("Nombre: "); gets(cliente.nombre); fflush(stdin);

```
printf("Direccion: "); gets(cliente.direcion); fflush(stdin);

printf("Edad: "); scanf("%d",&cliente.edad); fflush(stdin);

fwrite(&cliente,sizeof(cliente),1,fich);

}

void muestra(void)

{ int cod_temp; clrscr();

rewind(fich); 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;

} }

}
```