

Lenguaje C

PROF. ADRIANA M. GIL
PROF. JORGE R. ROMERO

Reglas de escritura y Estructura de un programa C

1. ANATOMÍA DE UN PROGRAMA C

Siguiendo la tradición, la mejor forma de aprender a programar en cualquier lenguaje es editar, compilar, corregir y ejecutar pequeños programas descriptivos.

```
#include <stdio.h>
main()
{
    printf ("Bienvenido a la Programacion en lenguaje C \n");
    return 0;
}
```

La función *main()* indica donde empieza el programa, cuyo cuerpo principal es un conjunto de sentencias delimitadas por dos llaves, una inmediatamente después de la declaración *main()* : "{", y otra que finaliza el listado "}". Todos los programas C arrancan del mismo punto: la primer sentencia dentro de dicha función, - en este caso *printf (".....")*.

En el *ejemplo1* el programa principal está compuesto por sólo dos sentencias: la primera es un llamado a una función denominada *printf ()*, y la segunda, *return*, que finaliza el programa retornando al Sistema Operativo.

El lenguaje C no tiene operadores de entrada-salida por lo que para escribir en pantalla es necesario llamar a una función externa de la librería (*stdio.h*). En este caso se invoca a la función *printf (argumento)* y a la cual se le envía como argumento aquellos caracteres que se desean escribir en la pantalla. Los mismos deben estar delimitados por comillas dobles. La secuencia "\n" que aparece al final del mensaje es la notación para el caracter "nueva línea" que hace avanzar al cursor a la posición extrema izquierda de la línea siguiente.

La segunda sentencia (*return 0*) termina el programa y devuelve un valor al Sistema operativo, por lo general cero si la ejecución fue correcta y valores distintos de cero para indicar diversos errores que pudieron ocurrir. Si bien no es obligatorio terminar el programa con un return, es conveniente indicarle a quien lo haya invocado, sea el Sistema Operativo o algún otro programa, si la finalización ha sido exitosa, o no.

Cada sentencia de programa queda finalizada por el terminador ";", el que indica al compilador el fin de la misma. Esto es necesario ya que, sentencias complejas pueden llegar a tener más de un renglón, y habrá que avisarle al compilador donde terminan.

2. ENCABEZAMIENTO

Las líneas anteriores a la función *main ()* se denominan *encabezamiento (header)* y son informaciones que se le suministran al Compilador.

La primera línea del programa está compuesta por una directiva: *#include*, que implica la orden de leer un archivo de texto (*<stdio.h>*) y reemplazar esta línea por el contenido de dicho archivo. En este archivo están incluidas declaraciones de las funciones luego llamadas por el programa (como *printf ()*). La directiva "*#include*" no es una sentencia de programa sino una orden de que se copie literalmente un archivo de texto en el lugar en que ella está ubicada, por lo que no es necesario terminarla con ";".

Hay dos formas distintas de invocar al archivo, a saber, si el archivo invocado está delimitado por comillas (por ejemplo "*stdio.h*") el compilador lo buscará en el directorio activo en el momento de compilar y si en cambio se lo delimita con los signos *<.....>* lo buscará en algún otro directorio, cuyo nombre habitualmente se le suministra en el momento de la instalación del compilador en el

disco (por ejemplo `c:\tc\include`). Por lo general estos archivos son guardados en un directorio llamado *include* y el nombre de los mismos está terminado con la extensión *.h*. La razón de la existencia de estos archivos es la de evitar la repetición de la escritura de largas definiciones en cada programa.

3. COMENTARIOS

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

En el lenguaje C se toma como comentario todo caracter interno a los símbolos: `/* */`, o lo que se encuentra luego de `//`. Los comentarios pueden ocupar uno o más renglones, por ejemplo:

// este es un comentario corto

/ este otro
es mucho más largo que el anterior */*

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

Variables y constantes

1. DEFINICION DE VARIABLES

Si se deseara imprimir los resultados de multiplicar un número fijo por otro que adopta valores entre 0 y 9, la forma normal de programar esto sería crear una *constante* para el primer número y un par de *variables* para el segundo y para el resultado del producto. Una variable no es más que un nombre para identificar una (o varias) posiciones de memoria donde el programa guarda los distintos valores de una misma entidad. Un programa debe *definir* a todas las variables que utilizará, antes de comenzar a usarlas, a fin de indicarle al compilador de que tipo serán, y por lo tanto cuanta memoria debe destinar para albergar a cada una de ellas.

```
/* programa que multiplica dos numeros*/
#include <stdio.h>
main()
{ int multiplicador;           // se define multiplicador como un entero
  int multiplicando;          // se define multiplicando como un entero
  int resultado;               // se define resultado como un entero
  multiplicador = 1000 ;       // se le asigna valores
  multiplicando = 2 ;
  resultado = multiplicando * multiplicador ;
  printf ("Resultado = %d\n", resultado); // muestra el resultado
  return 0; }
```

En las primeras líneas de texto dentro de *main()* se definen las variables como números enteros, es decir del tipo *"int"* seguido de un identificador (*nombre*) de la misma. Este puede tener la cantidad de caracteres que se desee, sin embargo de acuerdo al Compilador que se use, este tomará como significantes sólo los primeros n de ellos; siendo por lo general n igual a 32. Es conveniente darle a los identificadores de las variables, nombres que tengan un significado que luego permita una fácil lectura del programa. Los identificadores deben comenzar con una letra ó con el símbolo de subrayado `"_"`. El único símbolo no alfanumérico aceptado en un nombre es el `"_"`. El lenguaje C es sensible al tipo de letra usado; así tomará como variables distintas a una llamada *"variable"*, de otra escrita como *"VARIABLE"*. Es una convención entre los programadores

de C escribir los nombres de las variables y las funciones con minúsculas, reservando las mayúsculas para las constantes.

Los compiladores reservan determinados términos ó palabras claves (*Keywords*) para el uso sintáctico del lenguaje, tales como: *asm, auto, break, case, char, do, for*, etc.

En las líneas siguientes a la definición de las variables, ya se les puede asignar valores (como 1000 y 2) y luego efectuar el cálculo de la variable "**resultado**". La función *printf()* mostrará la forma de visualizar el valor de una variable. Insertada en el texto a mostrar, aparece una secuencia de control de impresión "%d" que indica, que en el lugar que ella ocupa, deberá ponerse el contenido de la variable (que aparece luego de cerradas las comillas que marcan la finalización del texto, y separada del mismo por una coma) expresado como un número entero decimal. Si se compila y corre el programa, se obtendrá la siguiente salida: **Resultado = 2000**

2. INICIALIZACION DE VARIABLES

Las variables del mismo tipo pueden definirse mediante una definición múltiple separándolas mediante ",", a saber : *int multiplicador, multiplicando, resultado;*. Esta sentencia es equivalente a las tres definiciones separadas. Las variables pueden también ser inicializadas en el momento de definirse: *int multiplicador = 1000, multiplicando = 2, resultado;*

De esta manera el ejemplo anterior podría escribirse:

```
#include <stdio.h>
main()
{
    int multiplicador=1000 , multiplicando=2 ;
    printf("Resultado = %d\n", multiplicando * multiplicador);
    return 0;
}
```

En la primera sentencia se definen e inician simultáneamente ambas variables. La variable "**resultado**" ha desaparecido debido a que es innecesaria. En la función *printf()* se ha reemplazado "**resultado**" por la operación entre las otras dos variables. Esta es una de las particularidades del lenguaje, en los parámetros pasados a las funciones pueden ponerse operaciones (incluso llamadas a otras funciones), las que se *ejecutan antes* de ejecutarse la función, pasando finalmente a esta el valor resultante de las mismas. El *ejemplo2bis* funciona exactamente igual que el *ejemplo2* pero su código ahora es mucho más compacto y claro.

3. TIPOS DE VARIABLES

Variables del tipo entero

En el ejemplo anterior definimos a las variables como enteros (*int*). De acuerdo a la cantidad de bytes que reserve el compilador para este tipo de variable, queda determinado el "alcance" ó máximo valor que puede adoptar la misma. Debido a que el tipo *int* ocupa dos bytes su alcance queda restringido al rango entre **-32.768** y **+32.767**, incluyendo el cero (0).

En caso de necesitar un rango más amplio, puede definirse la variable como "*long int nombre_de_variable*" ó en forma más abreviada "*long nombre_de_variable*". Declarada de esta manera, la variable puede alcanzar valores entre **- 2.347.483.648** y **+2.347.483.647**.

Para variables de muy pequeño valor puede usarse el tipo "*char*" cuyo alcance está restringido a **-128**, **+127** y por lo general ocupa un único byte.

Todos los tipos citados hasta ahora pueden alojar valores positivos ó negativos y, aunque es

redundante, esto puede explicitarse agregando el calificador "*signed*" delante; por ejemplo:

*signed int**signed long**signed long int**signed short**signed short int**signed char*

Si en cambio, se tiene una variable que sólo puede adoptar valores positivos (como la edad de una persona) se puede aumentar el alcance de cualquiera de los tipos, restringiéndolos a que sólo representen valores sin signo por medio del calificador "*unsigned*". Si se omite el calificador delante del tipo de la variable entera, éste se adopta por omisión (default) como "*signed*".

Tipo	Bytes	Valor mínimo	Valor máximo
signed char	1	-128	127
unsigned char	1	0	255
unsigned short	2	-32.768	+32.767
unsigned short	2	0	+65.535
signed int	2	-32.768	+32.767
unsigned int	2	0	+65.535
signed long	4	-2.147.483.648	+2.147.483.647
unsigned long	4	0	+4.294.967.295

Tabla 1: variables del tipo numero entero

Variables de numero real o punto flotante

Un número real ó de punto flotante es aquel que además de una parte entera, posee fracciones de la unidad. Por convención numérica éstos se suelen escribir de la siguiente manera: 2,3456, lamentablemente los compiladores usan la convención del *punto decimal* (en vez de la coma). Así el número Pi se escribirá: **3.14159**. Otro formato de escritura, es la notación científica. Por ejemplo podrá escribirse **2.345E+02**, equivalente a $2.345 * 100$ ó 234.5

Las variables de punto flotante son *siempre* con signo, y en el caso que el exponente sea positivo puede obviarse el signo del mismo. De acuerdo a su alcance hay tres tipos de punto flotante:

Tipo	Bytes	Valor mínimo	Valor máximo
float	4	3.4E-38	3.4E+38
double	8	1.7E-308	1.7E+308
long double	10	3.4E-4932	3.4E+4932

tabla 2: tipos de variables de punto flotante

4. VARIABLES DE TIPO CARACTER

El lenguaje C guarda los caracteres como números de 8 bits de acuerdo a la norma ASCII extendida, que asigna a cada caracter un número comprendido entre 0 y 255. Es común entonces que las variables que vayan a alojar caracteres sean definidas como: ***char c;***

Sin embargo, también funciona de manera correcta definirla como: ***int c;***

Esta última opción desperdicia un poco más de memoria que la anterior, pero en algunos casos particulares presenta ciertas ventajas. Por ejemplo: si se tiene una función que lee un archivo de texto ubicado en un disco. Dicho archivo puede tener cualquier caracter ASCII de valor comprendido entre 0 y 255. Para que la función pueda avisar que el archivo ha finalizado deberá enviar un número *no* comprendido entre 0 y 255 (por lo general se usa el -1, denominado ***EOF***, fin de archivo ó End Of File), en este caso dicho número no puede ser mantenido en una variable del tipo char, ya que esta sólo puede guardar entre 0 y 255 si se la define unsigned ó no podría mantener los caracteres comprendidos entre 128 y 255 si se la define signed (ver *tabla 1*). El problema se obvia fácilmente definiéndola como ***int***.

Las variables del tipo caracter también pueden ser inicializadas en su definición, por ejemplo es válido escribir: `char c = 97 ;`. Para que `c` contenga el valor ASCII de la letra "a", sin embargo esto resulta algo engorroso, ya que obliga a recordar dichos códigos. Existe una manera más directa de asignar un caracter a una variable la inicialización `char c = 'a' ;` es idéntica a la anterior.

Es decir que si se delimita un caracter con comilla simple, el compilador entenderá que debe suplantarlos por su correspondiente código numérico.

Existen una serie de caracteres que no son imprimibles. Un caso típico sería el de "nueva línea" ó ENTER. Con el fin de tener acceso a los mismos es que aparecen ciertas secuencias de escape convencionales. Las mismas están listadas en la *tabla 3* y su uso es idéntico al de los caracteres normales, así para resolver el caso de una asignación de "nueva línea" se escribirá:

`char c = '\n' ;` // secuencia de escape

Código	Significado	Valor ascii (decimal)	Valor ascii (hexadecimal)
'\n'	nueva línea	10	0x0A
'\r'	retorno de carro	13	0x0D
'\f'	nueva página	2	x0C
'\t'	tabulador horizontal	9	0x09
'\b'	retroceso (backspace)	8	0x08
'\"'	comilla simple	39	0x27
'\"'	comillas	4	0x22
'\\'	barra	92	0x5C
'\?'	interrogación	63	0x3F
'\nnn'	cualquier caracter (donde nnn es el código ASCII expresado en octal)		
'\xnn'	cualquier caracter (nn es el código ASCII expresado en hexadecimal)		

tabla 3: secuencias de escape

5. DEFINICION DE NUEVOS TIPOS (typedef)

A veces resulta conveniente crear otros tipos de variables, o bien redefinir con otro nombre las existentes, esto se puede realizar mediante la palabra reservada "*typedef*", por ejemplo:

`typedef unsigned long double enorme ;`

A partir de este momento, las definiciones siguientes tienen idéntico significado:

`unsigned long double nombre_de_variable ;`

`enorme nombre_de_variable ;`

6. CONSTANTES SIMBOLICAS

Por lo general es una mala práctica de programación colocar en un programa constantes en forma literal (sobre todo si se usan varias veces en el mismo) ya que el texto se hace difícil de comprender y aún más de corregir, si se debe cambiar el valor de dichas constantes.

Se puede en cambio asignar un símbolo a cada constante, y reemplazarla a lo largo del programa por el mismo, de forma que este sea más legible y además, en caso de querer modificar el valor, bastará con cambiarlo en la asignación.

El compilador, en el momento de crear el ejecutable, reemplazará el símbolo por el valor asignado.

Para dar un símbolo a una constante bastará, en cualquier lugar del programa (previo a su uso) poner la directiva: *#define*, por ejemplo:

```
#define VALOR_CONSTANTE 342
#define PI 3.1416
```

Tipos de operadores y expresiones

1. INTRODUCCION

var1 = var2 + var3;

Con la sentencia anterior, se le esta diciendo al programa, por medio del operador $+$, que compute la suma del valor de dos variables, y una vez realizado esto asigne el resultado a otra variable *var1*. Esta última operación (*asignación*) se indica mediante otro operador, el signo $=$.

C tiene una amplia variedad de operadores, y todos ellos caen dentro de 6 categorías, a saber: aritméticos, relacionales, lógicos, incremento y decremento, manejo de bits y asignación.

2. OPERADORES ARITMETICOS

Los operadores aritméticos, mostrados en la *tabla4*, comprenden las cuatro operaciones básicas, con un agregado, el operador *módulo*.

Símbolo	Descripción	Ejemplo
+	SUMA	$a + b$
-	RESTA	$a - b$
*	MULTIPLICACION	$a * b$
/	DIVISION	a / b
%	MODULO	$a \% b$
-	SIGNO	$-a$

Tabla 4: Operadores Aritmeticos

El operador módulo (%) se utiliza para calcular el resto del cociente entre dos ENTEROS, y NO puede ser aplicado a variables del tipo float ó double .

Si en una operación existen varios operadores, primero se evaluarán los de multiplicación, división y módulo y luego los de suma y resta. La precedencia de los tres primeros es la misma, por lo que si hay varios de ellos, se comenzará a evaluar a aquel que quede más a la izquierda. Lo mismo ocurre con la suma y la resta. Para evitar errores en los cálculos se pueden usar paréntesis, sin limitación de anidamiento, los que fuerzan a realizar primero las operaciones incluidas en ellos. Los paréntesis no disminuyen la velocidad a la que se ejecuta el programa sino que tan sólo obligan al compilador a realizar las operaciones en un orden dado, por lo que es una buena costumbre utilizarlos ampliamente. Los paréntesis tienen un orden de precedencia 0, es decir que antes que nada se evalúa lo que ellos encierran.

No existen operadores de potenciación, radicación, logaritmación, etc., ya que en el lenguaje C todas estas operaciones (y muchas otras) se realizan por medio de llamadas a Funciones.

El último de los operadores aritméticos es el de *signo*. No debe confundírsele con el de resta, ya que este es un operador unitario que opera sobre una única variable cambiando el signo de su contenido numérico. Obviamente no existe el operador $+$ unitario, ya que su operación sería DEJAR el signo de la variable, lo que se consigue simplemente por omisión del signo.

```
... int a=10,b=3,c;
...
c= a%b;
```

Como resultado de la operación, queda **c** con el valor 1

3. OPERADORES RELACIONALES

Todas las operaciones relacionales dan sólo dos posibles resultados: VERDADERO ó FALSO. En C, Falso es un valor entero nulo (cero) y Verdadero cualquier número distinto de cero

Símbolo	Descripción	Ejemplo
<	menor que	(a < b)
>	mayor que	(a > b)
<=	menor o igual que	(a <= b)
>=	mayor o igual que	(a >= b)
==	igual que	(a == b)
!=	distinto que	(a != b)

Tabla 5: operadores relacionales

Uno de los errores más comunes es confundir el operador relacional **IGUAL QUE** (==) con el de asignación **IGUAL A** (=). La expresión $a=b$ copia el valor de b en a , mientras que $a == b$ retorna **cero**, si a es distinto de b ó un número distinto de **cero** si son iguales.

Los operadores relacionales tiene menor precedencia que los aritméticos, de forma que $a < b + c$ se interpreta como $a < (b + c)$, pero aunque sea superfluo es recomendable el uso de paréntesis a fin de aumentar la legibilidad del texto.

Cuando se comparan dos variables tipo **char** el resultado de la operación dependerá de la comparación de los valores **ASCII** de los caracteres contenidos en ellas. Así el caracter 'a' (**ASCII 97**) será mayor que el 'A' (**ASCII 65**) ó que el 9 (**ASCII 57**).

4. OPERADORES LOGICOS

Hay tres operadores que realizan las conectividades lógicas **Y** (AND) , **O** (OR) y **NEGACION** (NOT) y están descritos en la TABLA 6 .

Símbolo	Descripción	Ejemplo
&&	Y (AND)	(a>b) && (c < d)
	O (OR)	(a>b) (c < d)
!	NEGACION (NOT)	! (a>b)

Tabla 6: operadores lógicos

Los resultados de la operaciones lógicas siempre adoptan los valores *cierto* ó *falso*. La evaluación de las operaciones lógicas se realiza de izquierda a derecha y se *interrumpe* cuando se ha asegurado el resultado.

El operador **NEGACION** invierte el sentido lógico de las operaciones, así será

$!(a > b)$ equivale a $(a < b)$
 $!(a == b)$ equivale a $(a != b)$

5. OPERADORES DE INCREMENTO Y DECREMENTO

Los operadores de incremento y decremento son sólo dos

Símbolo	Descripción	Ejemplo	Orden de evaluación
++	Incremento	$++i$ ó $i++$	1
--	Decremento	$--i$ ó $i--$	1

Tabla 7: operadores de incremento y decremento

Para visualizar rápidamente la función de los operadores antedichos, basta con decir que las sentencias: $a = a + 1;$ $a++;$ tienen una acción idéntica, de la misma forma que $a = a - 1;$ $a--;$ es decir incrementa y decrementa a la variable en una unidad.

Si bien estos operadores se suelen emplear con variables *int*, pueden ser usados sin problemas con cualquier otro tipo de variable. Así si *a* es un *float* de valor 1.05, luego de hacer *a++* adoptará el valor de 2.05 y de la misma manera si *b* es una variable del tipo *char* que contiene el caracter 'C', luego de hacer *b--* su valor será 'B'.

Si bien las sentencias *i++*; *++i*; son absolutamente equivalentes, la ubicación de los operadores incremento ó decremento indica *cuando* se realiza éste . Por ejemplo:

```
int i = 1, j, k;
j = i++;
k = ++i;
```

j es igualado al valor de *i* y *posteriormente* a la asignación *i* es incrementado por lo que *j* será igual a 1 e *i* igual a 2, luego de ejecutada la sentencia. En la siguiente instrucción *i* se incrementa *antes* de efectuarse la asignación tomando el valor de 3, el que luego es copiado en *k*.

6. OPERADORES DE ASIGNACION

En principio puede resultar algo fútil gastar papel en describir al operador *igual a* (*=*), sin embargo es necesario remarcar ciertas características del mismo .

Anteriormente se definió a una asignación como la copia del resultado de una expresión (*rvalue*) sobre otra (*lvalue*), esto implica que dicho lvalue debe tener *lugar* (es decir poseer una posición de memoria) para alojar dicho valor.

Es por lo tanto válido escribir *a = 17*; pero no es aceptado , en cambio *17 = a*; /* incorrecto */ ya que la constante numérica 17 no posee una ubicación de memoria donde alojar al valor de *a*.

Aunque parezca un poco extraño al principio las asignaciones, al igual que las otras operaciones, dan un resultado que puede asignarse a su vez a otra expresión. De la misma forma que *(a + b)* es evaluada y su resultado se puede copiar en otra variable: *c = (a + b)*; una asignación *(a = b)* da como resultado el valor de *b*, por lo que es lícito escribir *c = (a = b)*;

Debido a que las asignaciones se evalúan de derecha a izquierda, los paréntesis son superfluos, y podrá escribirse entonces: *c = a = b = 17*; con lo que las tres variables resultarán iguales al valor de la contante .

El hecho de que estas operaciones se realicen de derecha a izquierda también permite realizar instrucciones del tipo: *a = a + 17* ; significando esto que al valor que *tenia* anteriormente *a*, se le suma la constante y *luego* se copia el resultado en la variable .

Como este último tipo de operaciones es por demás común, existe en C un pseudocódigo, con el fin de abreviarlas. Así una operación aritmética o de bit cualquiera (simbolizada por OP) *a = (a) OP (b)* ; puede escribirse en forma abreviada como : *a OP = b* ;

Por ejemplo

```
a += b; /* equivale : a = a + b */
a -= b; /* equivale : a = a - b */
a *= b; /* equivale : a = a * b */
a /= b; /* equivale : a = a / b */
a %= b; /* equivale : a = a % b */
```

El pseudooperador debe escribirse con los dos símbolos seguidos, por ejemplo *+=*, y no será aceptado *+(espacio) =* .

Los operadores de asignación están resumidos en la siguiente tabla .

Símbolo	Descripción	Ejemplo	Orden de evaluación
=	igual a	a = b	13
op=	Pseudocódigo	a += b	13
=?:	asig.condicional	a = (c>b)?d:e	12

Tabla 8: operadores de asignación

En la tabla anterior aparece otro operador denominado ASIGNACION CONDICIONAL. El significado del mismo es el siguiente: *Ivalue = (operación relacional ó lógica) ? (rvalue 1) : (rvalue 2)*; de acuerdo al resultado de la operación condicional se asignará a Ivalue el valor de rvalue 1 ó 2. Si aquella es CIERTA será *Ivalue = rvalue 1* y si diera FALSO, *Ivalue = rvalue 2*.

Por ejemplo, si se quiere asignar a c el menor de los valores a ó b, bastará con escribir:

$$c = (a < b) ? a : b ;$$

Funciones de E/S

1. FUNCIONES PRINTF () Y SCANF ()

La función *printf* () escribe datos formateados en la salida estándar. La función *scanf* () lee datos formateados de la entrada estándar.

Sintaxis:

printf ("cadena de control", lista de argumentos);

scanf ("cadena de control", lista de argumentos);

La cadena de control está formada por caracteres imprimibles y códigos de formato.

Código	Formato
%c	Carácter
%d	Entero decimal con signo
%i	Entero decimal con signo
%e	Punto flotante en notación no científica: [-]d.ddd e [±] ddd
%f	Punto flotante en notación no científica: [-]dddd.ddd
%g	Usa %e o %f, el que sea más corto de longitud
%o	Entero octal sin signo
%s	Cadena de caracteres
%u	Entero decimal sin signo
%x	Entero hexadecimal sin signo
%%	Signo de tanto por ciento: %
%p	Puntero
%n	El argumento asociado debe ser un puntero a entero en el que se pone el número de caracteres impresos hasta el momento.

Tabla 9: especificadores de formato

Las órdenes de formato pueden tener modificadores. Estos modificadores van entre el % y la letra correspondiente.

2. DIFERENCIAS entre PRINTF y SCANF

En *printf* los argumentos son expresiones, pero en *scanf* los argumentos han de ser direcciones de memoria (punteros). Para obtener la dirección de una variable se ha de utilizar el operador *&*: *&variable*. En el caso de los arrays, el puntero apunta al primer elemento del vector.

En la cadena de control de *scanf* se pueden distinguir tres elementos:

- Especificadores de formato.
- Caracteres con espacios en blanco.
- Caracteres sin espacios en blanco.

Un espacio en blanco en la cadena de control da lugar a que *scanf()* salte uno o más espacios en blanco en el flujo de entrada. Un carácter blanco es un espacio, un tabulador o un carácter de nueva línea.

Un carácter que no sea espacio en blanco lleva a *scanf()* a leer y eliminar el carácter asociado. Por ejemplo, "%d, %d" da lugar a que *scanf()* lea primero un entero, entonces lea y descarte la coma, y finalmente lea otro entero. Si el carácter especificado no se encuentra, *scanf()* termina.

3. FUNCIONES PUTCHAR() Y GETCHAR()

La función *putchar* escribe un carácter en la salida estándar y necesita un argumento que es el carácter a escribir. La función *getchar* lee un carácter de la entrada estándar pero no recibe ningún argumento.

Ambas funciones devuelven, en caso de éxito, el carácter procesado, y en caso de error o fin de fichero, *EOF*.

Siempre que se manejen caracteres con las funciones de E/S es preferible declararlos como enteros, porque EOF tiene asignado el valor -1.

```
#include <stdio.h>
int main(void)
{
    char ch;
    ch = getchar();                //lee un caracter desde el teclado
    while (ch != '\n')            // mientras no se toque la tecla ENTER
    { if (ch == ' ')
        printf("\n");
      else
        putchar(ch);              //muestra el caracter
      ch = getchar();
    }
}
```

4. FUNCIONES PUTS() Y GETS()

La función *puts()* escribe una cadena de caracteres y un carácter de nueva línea al final de la cadena en la salida estándar. Acepta como argumento una cadena (sin formato). Si tiene éxito devuelve el último carácter escrito (siempre es '\n'). En otro caso, devuelve *EOF*.

La función *gets()* lee una cadena de caracteres de la entrada estándar hasta que se encuentra el carácter '\n', aunque este carácter no es añadido a la cadena. Acepta como argumento un puntero al principio de la cadena, es decir, el nombre de la variable cadena de caracteres; y devuelve dicho puntero si tiene éxito o la constante *NULL* si falla.

NULL es una constante definida en el fichero *stdio.h* que tiene valor 0. Esta constante se suele utilizar para denotar que un puntero no apunta a ningún sitio.

Con las funciones de lectura de cadenas es necesario tener una precaución muy importante: si, por ejemplo, en la declaración de cadena se ha reservado memoria para 100 caracteres y la función *gets* o *scanf* leen más de 100 caracteres, los caracteres a partir del 100 se están escribiendo en memoria en posiciones no reservadas.

Este problema se puede solucionar con la función *scanf()* y el modificador de formato *%100s*, por ejemplo, donde los caracteres introducidos a partir del número 100 son ignorados y no se

escriben en la variable cadena. Con la función *gets* no se puede hacer esto.

```
#include <stdio.h>
#include <string.h>
int main(void)
{
    char nombre[21], saludo[27];
    do
    {
        puts("Ingrese un nombre: ");           // muestra el mensaje
        gets(nombre);                          // lee desde el teclado
        strcpy(saludo, "Hola ");               // copia
        strcat(saludo, nombre);                // agrega al final
        strcat(saludo, "!");
        puts(saludo);                          // muestra por pantalla
    } while(nombre[0] != '\0');
}
```

6. FUNCIONES PUTC(), GETCH(), GETCHE() Y UNGETCH()

La función *putch* escribe un carácter en consola. Es similar a *putchar*. Las funciones *getch* y *getche* leen un carácter de consola, con eco a pantalla (*getche*) o sin eco (*getch*). Son similares a *getchar*.

Las teclas especiales, tales como las teclas de función, están representadas por una secuencia de dos caracteres: un carácter cero seguido del código de exploración de la tecla presionada. Así, para leer un carácter especial, es necesario ejecutar dos veces la función *getch* (o *getche*).

La función *ungetch*() devuelve un carácter al teclado. Acepta como parámetro el carácter a devolver y devuelve el propio carácter si hace la operación con éxito o *EOF* si ha ocurrido un error. Si hace *ungetch*(), la próxima llamada de *getch*() o cualquier otra función de entrada de teclado, leerá el carácter.

```
#include <stdio.h>
#include <conio.h>
int main(void)
{
    char ch;
    printf("Contraseña: ");
    do
    {
        ch = getch();           //lee un caracter desde el teclado
        putch('*');             //muestra un asterisco
    } while(ch != '\r');
}
```

7. FUNCIÓN KBHIT()

La función *kbhit*() devuelve un valor cierto (distinto de cero) si hay una tecla disponible y falso (0) si no la hay.

```
#include <stdio.h>
#include <conio.h>
int main(void)
{
    long cont = 0;
    printf("Presione una tecla para continuar...\n");
    while (! kbhit() )           //mientras no se toque una tecla
        cont++;
    printf("Conte hasta %ld\n", cont);
    return 0;
}
```

Proposiciones para el control de flujo de programa

1. INTRODUCCION

Se denomina BLOQUE DE SENTENCIAS al conjunto de sentencias individuales incluidas dentro un par de llaves. Por ejemplo $\{ \textit{sentencia 1} ;$

.....
sentencia n ; }

Este conjunto se comportará sintácticamente como una sentencia simple y la llave de cierre del bloque NO debe ir seguida de punto y coma. Un ejemplo de bloque ya visto , es el cuerpo del programa principal de la función *main()* .

```
main()
{ bloque de sentencias
}
```

En las proposiciones de control de flujo de programa, se trabaja alternativamente con sentencias simples y bloques de ellas.

2. PROPOSICION IF - ELSE

Esta proposición sirve para ejecutar ciertas sentencias de programa, si una expresión resulta CIERTA ú otro grupo de sentencias, si aquella resulta FALSA. Su interpretación literal sería: SI es CIERTA tal cosa, haga tal otra, si no lo es salteéla .

El caso más sencillo sería :

if(expresión) ó *if*(expresión) sentencia ;
sentencia ;

Quando la sentencia que sigue al **if** es única, las dos formas de escritura expresadas arriba son equivalentes. La sentencia sólo se ejecutará si el resultado de "expresión" es distinto de cero (CIERTO), en caso contrario el programa saltará dicha sentencia, realizando la siguiente en su flujo.

En casos más complejos, la proposición *if* puede estar seguida por un bloque de sentencias :

```
if (expresión)
{ sentencia 1;
  .....
  sentencia n ;}
```

Las dos maneras son equivalentes, por lo que la posición de la llave de apertura del bloque queda librada al gusto del programador. El indentado de las sentencias (sangría) es también optativo, pero sumamente recomendable, sobre todo para permitir la lectura de proposiciones muy complejas ó anidadas. El bloque se ejecutará en su conjunto si la expresión resulta CIERTA. El uso del *else* es optativo, y su aplicación resulta en la ejecución de una, ó una serie de sentencias en el caso de que la expresión del *if* resulta FALSA. Su aplicación puede verse en el siguiente ejemplo:

```

if (expresion)
{  sentencia 1;
   sentencia 2;
}
sentencia 3;
sentencia 4;
sentencia 5;

```

```

if (expresion)
{ sentencia 1;
  sentencia 2;
}
else
{ sentencia 3;
  sentencia 4;
}
sentencia 5;

```

En el ejemplo de la izquierda no se usa el `else` y por lo tanto las sentencias 3, 4 y 5 se ejecutan siempre. En el segundo caso, las sentencias 1 y 2 se ejecutan solo si la expresión es CIERTA, en ese caso las 3 y 4 NO se ejecutarán para saltarse directamente a la 5, en el caso de que la expresión resulte FALSA se realizarán las 3 y 4 en lugar de las dos primeras y finalmente la 5. La proposición `else` queda siempre asociada al `if` más cercano, arriba de él.

Es común también, en caso de decisiones múltiples, el uso de anidamientos *else-if*.

3. PROPOSICION SWITCH

El `switch()` es una forma sencilla de evitar largos y confusos anidamientos de *else-if*.

Suponiendo que se esta implementando un Menú, con varias elecciones posibles. El esqueleto de una posible solución al problema usando *if-else* podría ser el 1º del ejemplo, Como es fácil de ver, cuando las opciones son muchas, el texto comienza a hacerse difícil de entender y engorroso de escribir.

El mismo programa, utilizando un `switch()`, quedaría mucho más claro de leer, y sencillo de escribir, como se aprecia a la derecha del anterior.

```
#include <stdio.h>
main()
{ int c;
printf("\nMENU :");
printf("\n  A = ADICIONAR A LA LISTA ");
printf("\n  B = BORRAR DE LA LISTA  ");
printf("\n  O = ORDENAR LA LISTA  ");
printf("\n  I = IMPRIMIR LA LISTA  ");
printf("\n\nESCRIBA SU SELECCION \n");

if( (c = getchar()) != '\n' )
{ if( c == 'A') printf ("UD. QUIERE AGREGAR");
else
if( c == 'B') printf("UD. QUIERE BORRAR");
else
if( c == 'O') printf ("UD. QUIERE ORDENAR");
else
if(c== 'I') printf ( "UD. QUIERE IMPRIMIR");
else
printf("\a\a  CARACTER ILEGAL !!!");
}
else
printf("\n UD. NO SELECCIONÓ NADA !"); }
```

```
#include <stdio.h>
main()
{ int c;
printf("\nMENU :");
printf("\n  A = ADICIONAR A LA LISTA ");
printf("\n  B = BORRAR DE LA LISTA  ");
printf("\n  O = ORDENAR LA LISTA  ");
printf("\n  I = IMPRIMIR LA LISTA  ");
printf("\n\nESCRIBA SU SELECCION \n ");
c = getchar();
switch (c)
{ case 'A':
printf("UD. QUIERE AGREGAR"); break;
case 'B':
printf("UD. QUIERE BORRAR"); break;
case 'O':
printf("UD. QUIERE ORDENAR"); break;
case 'I':
printf("UD. QUIERE IMPRIMIR"); break;
case '\n':
printf("\n UD. NO SELECCIONÓ NADA !");
break;
default :
printf("\a\a  CARACTER ILEGAL !!!");
}
}
```

El `switch()` empieza con la sentencia: `switch (expresión)`. La expresión contenida por los paréntesis debe ser ENTERA, en nuestro caso un caracter; luego mediante una llave abre el bloque de las sentencias de comparación. Cada una de ellas se representa por la palabra clave "`case`" seguida por el valor de comparación y terminada por dos puntos. Seguidamente se ubican las sentencias que se quieren ejecutar, en el caso que la comparación resulte CIERTA. En el caso

de resultar FALSA, se realizará la siguiente comparación, y así sucesivamente.

La sentencia con la que se termina cada *case* es *break*. Una característica poco obvia del *switch*, es que si se eliminan los *break* del programa anterior, al resultar CIERTA una sentencia de comparación, se ejecutarán las sentencias de ese *case* particular pero TAMBIEN la de todos los *case* por debajo del que ha resultado verdadero. Quizás se aclare esto diciendo que, las sentencias propias de un *case* se ejecutarán si su comparación ú otra comparación ANTERIOR resulta CIERTA. La razón para este poco "juicioso" comportamiento del *switch* es que así se permite que varias comparaciones compartan las mismas sentencias de programa, por ejemplo:

```
...
case 'X':
case 'Y':
case 'Z':
    printf(" UD. ESCRIBIO X, Y, ó Z ");
    break ;
...
```

La forma de interrumpir la ejecución luego de haber encontrado un *case* cierto es por medio del *break*, el que da por terminado el *switch*().

Al final del bloque de sentencias del *switch*(), aparece una optativa llamada *default*, que implica: si no se ha cumplido ningún *case*, ejecute lo que sigue. Es algo superfluo poner el *break* en este caso, ya que no hay más sentencias después del *default*, sin embargo, como el orden en que aparecen las comparaciones no tiene importancia para la ejecución de la instrucción, puede suceder que en futuras correcciones del programa se agregue algún nuevo *case* luego del *default*, por lo que es conveniente preverlo, agregando el *break*, para evitar errores de laboriosa ubicación.

4. LA ITERACION WHILE

El *while* es una de las tres iteraciones posibles en C. Su sintaxis podría expresarse de la siguiente forma:

<pre>while (expresion) proposición 1 ;</pre>	ó <pre>while(expresión) { proposición 1 ; proposición 2 ; proposición n ; }</pre>
--------------------------------------------------	---------------------------------------------------------------------------------------------------------

Esta sintaxis expresada en palabras significaría: mientras (*expresión*) dé un resultado CIERTO ejecútase la proposición 1, en el caso de la izquierda ó ejecútase el bloque de sentencias, en el caso de la derecha.

Por lo general, dentro de la proposición ó del bloque de ellas, se modifican términos de la expresión condicional, para controlar la duración de la iteración.

Por ejemplo: Ingresar una lista de 20 números y obtener su promedio

```
#include<stdio.h>
#include<conio.h>
main ()
{
    int a , suma=0 , i=0 ;
    float promedio;
    scanf ( "%d" , &a );
    while(i<20)
        // se lee el número, antes de entrar al ciclo
        //las siguientes sentencias se ejecutarán mientras
```

```

        {      suma = suma + a ;           // i sea menor a 20
              i++;
              scanf ( "%d" , &a ) ;         //se lee otro número, dentro del ciclo
        }

    promedio = 1.0 * suma / 20 ;           // se multiplica por un float, para obtener otro float
    printf ( "\nel promedio es %.2f\n" , promedio ) ; // %.2f muestra la variable float con dos decimales
    getch ( ) ;
    return 0;
}

```

5. LA ITERACION DO - WHILE

Su sintaxis será:

```

do
{
    proposición 1 ;
    .....
    proposición n ;
}
while (expresión) ;

```

Expresado en palabras, esto significa: ejecute las proposiciones, luego repita la ejecución mientras la expresión dé un resultado CIERTO. La diferencia fundamental entre esta iteración y la anterior es que el *do-while* se ejecuta siempre AL MENOS una vez, sea cual sea el resultado de expresión. Siguiendo con el mismo ejemplo anterior

```

#include<stdio.h>
#include<conio.h>
main()
{
    int a , suma=0 , i=0 ;
    float promedio ;
    scanf ( "%d" , &a ) ;
    do                                     // comienza el ciclo
    {
        scanf ( "%d" , &a ) ;
        suma += a ;                       // reemplaza a suma = suma + a
        i++ ;
    } while ( i < 20 ) ;                   //el ciclo se ejecuta, mientras i sea menor a 20
    promedio = 1.0 * suma / 20 ;
    printf ( "\nel promedio es %f\n" , promedio ) ;
    getch ( ) ;
    return 0;
}

```

6. ITERACION FOR

El *for* es simplemente una manera abreviada de expresar un *while* , veamos su sintaxis :

```

for ( expresión1 ; expresión2 ; expresion3 )
{
    proposición 1 ;
    .....
    proposición n ;
}

```

Esto es equivalente a :

```

expresión1 ;
while ( expresión2 )
{
    proposición1 ;
}

```



```

.....
    proposición2 ;
    expresion3 ;
}

```

La *expresión1* es una asignación de una ó más variables, (equivale a una inicialización de las mismas), la *expresión2* es una relación de algún tipo que, mientras dé un valor CIERTO, permite la iteración de la ejecución y *expresión3* es otra asignación, que comúnmente varía alguna de las variables contenida en *expresión2*.

Todas estas expresiones, contenidas en el paréntesis del FOR deben estar separadas por PUNTO Y COMA y NO por comas simples.

No es imprescindible que existan TODAS las expresiones dentro del paréntesis del FOR, pudiéndose dejar en blanco algunas de ellas, por ejemplo:

```

for ( ; exp2 ; exp3)      ó
for (exp1 ; ; )           ó
for ( ; ; )

```

Estas dos últimas expresiones son interesantes desde el punto de vista de su falta de término relacional, lo que implica que el programador deberá haber previsto alguna manera alternativa de salir del lazo (probablemente mediante *break* ó *return*) ya que sino, la ejecución del mismo es infinita.

```

#include<stdio.h>
#include<conio.h>
main()
{   int a , suma=0 , i=0 ;
    float promedio ;
    for ( i=0 ; i<20 ; i++ )      //el ciclo se ejecutará con i desde 0 hasta 19
    {   scanf ( "%d" , &a ) ;
        suma+ = a ;
    }
    promedio = 1.0 * suma/20 ;
    printf ( "\nel promedio es %.2f\n" , promedio);
    getch();
    return 0;
}

```

Del mismo modo las expresiones pueden ser compuestas

```

for ( exp1a, exp1b ; exp2a, exp2b ; exp3a, exp3b)
.....

```

7. LA SENTENCIA BREAK

El *break* sirve también para terminar loops producidos por *while*, *do-while* y *for* antes que se cumpla la condición normal de terminación. Se puede utilizar para terminar un ciclo indeterminado.

```

#include <stdio.h>
#include <ctype.h>
main()
{   char c ;
    printf ( " ESTE ES UN LOOP INDEFINIDO " ) ;

```

```

while ( 1 )                                // 1 se interpreta como verdadero
{
    printf( " DENTRO DEL LOOP INDEFINIDO (apriete una tecla): " );
    c = toupper ( getch() );                //convierte la lectura a mayúscula
    if( ( c == 'Q' ) break ;
    printf( "\nNO FUE LA TECLA CORRECTA PARA ABANDONAR EL LOOP " );
}
printf("\nTECLA CORRECTA : FIN DEL WHILE ");
}

```

La expresión *while (1)* SIEMPRE es cierta, por lo que el programa correrá imparable hasta que el operador oprima la tecla "secreta" Q . Esto se consigue en el *if*, ya que cuando c es igual al *ascii* Q se ejecuta la instrucción *break*, dando por finalizado el *while*.

El mismo criterio podría aplicarse con el *do-while* ó con *for*, por ejemplo haciendo

```

for (;;) {          .....          /* loop indefinido */
    if( expresión )
        break ;      /* ruptura del loop cuando expresión sea verdadera */
}

```

8. LA SENTENCIA CONTINUE

La sentencia *continue* es similar al *break* con la diferencia que en vez de terminar violentamente un loop, termina con la realización de una iteración particular y vuelve a verificar la expresión.

9. LA FUNCION EXIT ()

La función *exit (argumento)* tiene una operatoria mucho más drástica que las anteriores, en vez de saltar una iteración ó abandonar un lazo de programa, esta abandona directamente al programa mismo dándolo por terminado. Realiza también una serie de operaciones útiles como ser, el cerrado de cualquier archivo que el programa hubiera abierto, el vaciado de los buffers de salida, etc.

Normalmente se la utiliza para abortar los programas en caso de que se esté por cometer un error fatal é inevitable. Mediante el valor que se le ponga en su **argumento** se le puede informar a quien haya llamado al programa (Sistema Operativo, archivo .bat, u otro programa) el tipo de error que se cometió.

Funciones

1. INTRODUCCION

La forma más razonable de encarar el desarrollo de un programa complicado es aplicar lo que se ha dado en llamar "**Programación Top - Down**". Esto implica que, luego de conocer cual es la meta a alcanzar, se subdivide esta en varias tareas concurrentes o más simples, por ejemplo :

Leer desde el teclado, procesar datos, mostrar los resultados.

Luego a estas se las vuelve a dividir en otras menores. Y así se continúa hasta llegar a tener un gran conjunto de pequeñas y simples tareas, del tipo de "*leer una tecla*" ó "*imprimir un caracter*". Luego sólo resta abocarse a resolver cada una de ellas por separado.

Tal es el criterio con que está estructurado el lenguaje C, donde una de sus herramientas fundamentales son las funciones. Trae una gran cantidad de Librerías de toda índole, matemáticas, de Entrada - Salida, de manejo de textos, de manejo de gráficos, etc., que solucionan la mayor parte de los problemas básicos de programación. Sin embargo será

inevitable que en algún momento se tengan que crear funciones propias.

Para hacer que las instrucciones contenidas en una función, se ejecuten en determinado momento, no es necesario más que escribir su nombre como una línea de sentencia en el programa.

Convencionalmente en C los nombres de las funciones se escriben en minúscula y siguen las reglas dadas anteriormente para los de las variables, pero deben ser seguidos, para diferenciarlas de aquellas por un par de paréntesis.

Dentro de estos paréntesis estarán ubicados los datos que se les pasan a las funciones. Está permitido pasarles uno, ninguno ó una lista de ellos separados por comas, por ejemplo: *pow10(a)*, *getch()*, *strcmp(s1, s2)*.

Un concepto sumamente importante es que los argumentos que se les envían a las funciones son los VALORES de las variables y NO las variables mismas. En otras palabras, cuando se invoca una función de la forma *pow10(a)* en realidad se está copiando en el "stack" (pila del programa) de la memoria el valor que tiene en ese momento la variable *a*, la función podrá usar este valor para sus cálculos, pero está garantizado que los mismos no afectan en absoluto a la variable en sí misma. Es posible que una función modifique a una variable, pero para ello, será necesario comunicarle la DIRECCION EN MEMORIA de dicha variable.

Las funciones pueden ó no devolver valores al programa invocante. Hay funciones que tan sólo realizan acciones, como por ejemplo *clrscr()*, que borra la pantalla de video, y por lo tanto no retornan ningún dato de interés; en cambio otras efectúan cálculos, devolviendo los resultados de los mismos.

La invocación a estos dos tipos de funciones difiere algo, por ejemplo:

```
clrscr();  
c = getch();
```

donde en el segundo caso el valor retornado por la función se asigna a la variable *c*. Obviamente ésta deberá tener el tipo correcto para alojarla.

2. DECLARACION DE FUNCIONES

Antes de escribir una función es necesario informarle al Compilador los tamaños de los valores que se le enviarán en el *stack* y el tamaño de los valores que ella retornará al programa invocante. Estas informaciones están contenidas en la DECLARACION del PROTOTIPO DE LA FUNCION.

Formalmente dicha declaración queda dada por :

tipo del valor de retorno nombre_de_la_función(lista de tipos de parámetros)

Por ejemplo:

```
float mi_funcion ( int i , double j ) ;  
double otra_funcion ( void ) ;  
otra_mas ( long p ) ;  
void la_ultima ( long double z , char y , int x , unsigned long w ) ;
```

El primer término del prototipo da el tipo del dato retornado por la función; en caso de obviarse el mismo se toma, por omisión, el tipo *int*. Sin embargo, aunque la función devuelva este tipo de dato, para evitar malas interpretaciones es conveniente explicitarlo.

Ya que el tipo de retorno por defecto es el *int*, se debe indicar cuando la función NO retorna nada, esto se realiza por medio de la palabra *void* (sin valor). De la misma manera se actúa, cuando no se le enviarán argumentos.

La declaración debe anteceder en el programa a la definición de la función. Es normal, por

razones de legibilidad de la documentación, encontrar todas las declaraciones de las funciones usadas en el programa, en el **HEADER** del mismo, junto con los *include* de los archivos **.h* que tienen los prototipos de las funciones de Librería.

3. DEFINICION DE LAS FUNCIONES

La definición de una función puede ubicarse en cualquier lugar del programa, con sólo dos restricciones: debe hallarse luego de dar su prototipo, y no puede estar dentro de la definición de otra función (incluida *main()*). Es decir que en C las definiciones no pueden anidarse.

No confundir la definición de una función con la llamada a la misma; una función puede llamar a tantas otras como desee.

La definición debe comenzar con un encabezamiento, que debe coincidir totalmente con el prototipo declarado para la misma, y a continuación del mismo, encerradas por llaves se escribirán las sentencias que la componen; *por ejemplo*:

```
#include <stdio.h>

float mi_funcion(int i, double j);    // DECLARACION del PROTOTIPO, termina en ";"

main()
{
    float k ;                        //variables locales a main()
    int p ;
    double z ;
    .....
    k = mi_funcion( p, z );          //LLAMADA a la función
    .....
}                                     // fin de la función main()

float mi_funcion(int i, double j)    // DEFINICION de la función del usuario, NO lleva ";"
{
    float n;                        //variables locales a la funcion del usuario
    .....
    printf("%d", i);                // LLAMADA a una función estándar
    .....
    return ( 2 * n );               // RETORNO devolviendo un valor float
}
```

Ejemplo de funciones con parámetros.

```
#include < conio.h >
#include < stdio.h >
int suma ( int a , int b );          //Prototipo de funciones
int muestra_suma ( int st );
main ( )
{
    int s , z , w ;                  //Declaración de variables locales
    clrscr ( ) ;
    printf ( " Ingrese un valor --> " );
    scanf ( "%d" , &z );
    printf ( " Ingrese un valor --> " );
    scanf ( "%d" , &w );
```

```

    s = suma ( z , w );           //asigna en la variable s , el retorno de la función suma
    muestra_suma ( s );           //llama a la funcion muestra_suma ( )
    getch();
    return 0;
}

int suma ( int a , int b )        //función que suma dos números
{
    int st ;
    st = a + b ;
    return ( st );
}

int muestra_suma(int st)           //función que muestra la suma
{
    printf("El valor de la suma es ...%d",st);
}

```

4. FUNCIONES QUE NO RETORNAN VALOR NI RECIBEN PARAMETROS

```

#include <stdio.h>
void pausa ( void ) ;
main ()
{
    int cont = 1;
    printf("VALOR DEL CONTADOR DENTRO DEL while \n");
    while (cont <= 10)
    {
        if (contador == 5 ) pausa ( ) ;
        printf ( "%d\n" , cont++);    //suma 1 al contador luego de mostrarlo
    }
    pausa ( ) ;
    printf ( "VALOR DEL CONTADOR LUEGO DE SALIR DEL while: %d", cont) ;
    return 0;
}

void pausa ( void )
{
    char c ;
    printf("\nAPRIETE ENTER PARA CONTINUAR ") ;
    while ( (c = getchar ( )) != '\n') ;    //mientras no toque enter, no sale del while
}

```

En la segunda línea se declaró la función *pausa ()*, sin valor de retorno ni parámetros.

Luego esta es llamada dos veces por el programa principal, una cuando contador adquiere el valor de 5 (antes de imprimirlo) y otra luego de finalizar el *loop*.

Posteriormente la función es definida. El bloque de sentencias de la misma está compuesto, en este caso particular, por la definición de una variable *c*, la impresión de un mensaje de aviso y finalmente un *while* que no hace nada, solo espera recibir *<ENTER>*.

En cada llamada, el programa principal transfiere el comando a la función, ejecutándose, hasta que ésta finalice, su propia secuencia de instrucciones. Al finalizar la función esta retorna el comando al programa principal, continuándose la ejecución por la instrucción que sucede al llamado.

Si bien las funciones aceptan cualquier nombre, es una buena técnica de programación nombrarlas con términos que representen, aunque sea vagamente, su operatoria.

Se puede salir prematuramente de una función *void* mediante el uso de *return*, sin que este sea seguido de ningún parámetro ó valor.

5. FUNCIONES QUE RETORNAN VALOR

```
#include <stdio.h>
#define FALSO 0
#define CIERTO 1
int finalizar(void);
int lea_char(void);
main ()
{ int i = 0;
  int fin = FALSO;
  printf ("Ejemplo de Funciones que retornan valor\n");
  while (fin == FALSO)
  {    i++;
      printf ("i == %d \n", i);
      fin = finalizar ();
  }
  printf ("\n\nFIN DEL PROGRAMA.....");
  return 0;
}

int finalizar ( void )
{    int c;
    printf ("Otro número ? (s/n) ");
    do
    {    c = lea_char ();
    } while (( c != 'n') && (c != 's'));
    return (c == 'n');           // si es verdadero, devuelve 1 y si es falso devuelve 0
}

int lea_char(void)
{    int j;
    if ( (j = getch()) >= 'A' && j <= 'Z')
        return (j + ( 'a' - 'A' ));
    else
        return j;
}
```

Las dos primeras líneas del programa incluyen los prototipos de las funciones de librería usadas, (en este caso *printf()* y *getch()*). En las dos siguientes se les da nombres simbólicos a dos constantes que se usaran en las condiciones lógicas y posteriormente se dan los prototipos de dos funciones creadas.

Podrían haberse obviado, en este caso particular, estas dos últimas declaraciones, ya que ambas retornan un *int* (*default*), sin embargo el hecho de incluirlas hará que el programa sea más fácilmente comprensible en el futuro.

Comienza luego la función *main()*, inicializando dos variables, *i* y *fin*, donde la primera servirá de contador y la segunda de indicador lógico. Luego de imprimir el rótulo del programa, se entra

en un loop en el que se permanece todo el tiempo en que fin sea *FALSO*. Dentro de este loop, se incrementa el contador, se imprime, y se asigna a fin un valor que es el retorno de la función *finalizar()*.

Esta asignación realiza la llamada a la función, la que toma el control del flujo del programa, ejecutando sus propias instrucciones.

La función *finalizar()* define su variable propia, *c*, y luego entra en un *do-while*, que efectúa una llamada a otra función, *lea_char()*, y asigna su retorno a *c* iterando esta operativa si *c* no es 'n' ó 's', note que: *c != 'n' && c != 's'* es equivalente a: *!(c == 'n' / / c == 's')*.

La función *lea_char()* tiene como misión leer un caracter enviado por el teclado, (lo realiza dentro de la expresión relacional del *IF*) y salvar la ambigüedad del uso de mayúsculas ó minúsculas en las respuestas, convirtiendo las primeras en las segundas. Es fácil de ver que, si un caracter esta comprendido entre A y Z, se le suma la diferencia entre los ASCII de las minúsculas y las mayúsculas (97 - 65 = 32) para convertirlo, y luego retornarlo al invocante.

Esta conversión fue incluida a modo de ejemplo solamente, ya que existe una de Librería, *tolower()* declarada en *ctype.h*, que realiza la misma tarea.

Cuando *lea_char()* devuelva un caracter *n* ó *s*, se saldrá del *do-while* en la función *finalizar()* y se retornará al programa principal, el valor de la comparación lógica entre el contenido de *c* y el ASCII del caracter *n*. Si ambos son iguales, el valor retornado será *1* (CIERTO) y en caso contrario *0* (FALSO). Mientras el valor retornado al programa principal sea *falso*, este permanecerá dentro de su *while* imprimiendo valores sucesivos del contador, y llamadas a las funciones, hasta que finalmente un retorno de *cierto* (el operador presionó la tecla *n*) hace terminar el loop e imprimir el mensaje de despedida.

Existen funciones que retornan datos de tipo distinto al *int*. Pero para ello es conveniente presentar antes, otra función muy común de entrada de datos: *scanf()*, que nos permite leer datos completos (no solo caracteres) enviados desde el teclado, su expresión formal es algo similar a la del *printf()*:

scanf("secuencia de control", dirección de la variable);

Donde en la secuencia de control se indicará que tipo de variable se espera leer, por ejemplo :

%d	si se desea leer un entero decimal	(int)
%o	" " " " " " octal	"
%x	" " " " " " hexadecimal	"
%c	" " " " " carácter	
%f	leerá un float	
%ld	leerá un long int	
%lf	leerá un double	
%Lf	leerá un long double	

Tabla 6: especificadores de formato

Por "*dirección de la variable*" deberá entenderse que se debe indicar, en vez del nombre de la variable en la que se cargará el valor leído, la dirección de su ubicación en la memoria de la máquina. Esto suena sumamente apabullante, pero por ahora solo se dirá que para ello es necesario simplemente anteponer el signo *&* al nombre de la misma.

Así, si se desea leer un entero y guardarlo en la variable "*valor_leido*" se escribirá: *scanf("%d",&valor_leido);* en cambio si se deseara leer un entero y un valor de punto flotante será: *scanf("%d %f", &valor_entero, &valor_punto_flotante);*

El tipo de las variables deberá coincidir EXACTAMENTE con los expresados en la secuencia de control, ya que de no ser así, los resultados son impredecibles.

El prototipo de *scanf()* está declarado en *stdio.h*.

EJEMPLO: Desarrollar un programa que presente primero, un menú para seleccionar la conversión de °C a Fahrenheit ó de centímetros a pulgadas, hecha la elección, pregunte el valor a convertir y posteriormente dé el resultado.

Si se supone que las funciones que se usaran en el programa serán frecuentemente usadas, se puede poner las declaraciones de las mismas, así como todas las constantes que usen, en un archivo texto, por ejemplo convers.h. Este podrá ser guardado en el subdirectorío donde están todos los demás (include) ó dejado en el directorio activo, en el cual se compila el programa fuente del ejemplo. Para variar, esto último es nuestro caso.

```
CONVERS.H
#include <conio.h>
#define FALSO 0
#define CIERTO 1
#define CENT_POR_INCH 25.4
void pausa(void);
void mostrar_menu(void);
int seleccion(void);
void cm_a_pulgadas(void);
void grados_a_fahrenheit(void);
double leer_valor(void);
```

Vemos que un *Header* puede incluir llamadas a otros (en este caso *conio.h*). Hemos puesto también la definición de todas las constantes que usaran las funciones abajo declaradas. De dichas declaraciones vemos que se usaran funciones que no retornan nada, otra que retorna un entero y otra que devuelve un *double*.

En el programa en sí, la invocación a *conversión.h* se hace con comillas, por haber decidido dejarlo en el directorio activo.

```
#include <stdio.h>
#include "convers.h"
main()
{
    int fin = FALSO;
    while (! fin )
    {
        mostrar_menu();
        switch(seleccion())
        {
            case 1: cm_a_pulgadas ();
                    break;
            case 2: grados_a_fahrenheit ();
                    break;
            case 3: fin = CIERTO;
                    break;
            default: printf("\n¡Error en la Seleccion!\a\a\n");
                    pausa ();
        }
    }
    return 0;
}
```



```

/* Funciones */
void pausa ( void )
{
    char c = 0;
    printf ( "\n\n\nAPRIETE ENTER PARA CONTINUAR " );
    while ( ( c = getch ( ) ) != '\r' );
}

void mostrar_menu(void)
{
    clrscr ( ) ;
    printf ( "\n      Menu\n" );
    printf ( "-----\n" );
    printf ( "1: Centimetros a pulgadas\n" );
    printf ( "2: Celsius a Fahrenheit\n" );
    printf ( "3: Terminar\n" );
}

int selección (void)
{
    printf ( "\nEscriba el número de su Selección: " );
    return ( getch ( ) - '0' );    //devuelve el n° entero que representa el ASCII
}

void cm_a_pulgadas (void)
{
    double centimetros;           /* Guardará el valor pasado por leer_valor() */
    double pulgadas;             /* Guardará el valor calculado */
    printf ( "\nEscriba los Centimetros a convertir: ");
    centimetros = leer_valor();

    pulgadas = centimetros * CENT_POR_INCH;
    printf ( " %.3f Centimetros = %.3f Pulgadas\n " , centimetros, pulgadas );
    pausa();
}

void grados_a_fahrenheit(void)
{
    double grados;               /* Guardará el valor pasado por leer_valor() */
    double fahrenheit;          /* Guardará el valor calculado */
    printf ( "\nEscriba los Grados a convertir: ");
    grados = leer_valor();
    fahrenheit = ( ( grados * 9.0)/5.0 ) + 32.0 ;
    printf ( "%.3f Grados = %.3f Fahrenheit", grados, fahrenheit);
    pausa ( ) ;
}

double leer_valor(void)
{
    double valor;                /* Variable para guardar lo leído del teclado */
    scanf("%lf", &valor);
    return valor;
}

```

Descripción del ejemplo: primero se incluyeron todas las definiciones presentes en el archivo *convers.h* que se había creado previamente. Luego *main()* entra en un loop, que finalizará cuando la variable *fin* tome un valor CIERTO, y dentro del cual lo primero que se hace es llamar a *mostrar_menú()*, que pone los rótulos de opciones .

Luego se entra en un *switch* que tiene como variable, el retorno de la función *selección()* (que tiene que ser un entero), según sea éste se saldrá por alguno de los tres *case*. La función *seleccion()* lee el teclado mediante un *getche()*, (similar a *getch()* antes descripta, pero con la diferencia que aquella hace eco del carácter en la pantalla) y finalmente devuelve la diferencia entre el ASCII del número escrito menos el ASCII del número **cero**, es decir, un entero igual numéricamente al valor que el operador quiso introducir. Si este fue 1, el *switch* invoca a la función *cm_a_pulgadas()* y en caso de ser 2 a *grados_a_fahrenheit()*.

Estas dos últimas proceden de igual manera: indican que se escriba el dato y pasan el control a *leer_valor()*, la que mediante *scanf()* lo hace, retornando en la variable valor, un double, que luego es procesado por aquellas convenientemente. Como hasta ahora la variable fin del programa principal no ha sido tocada, y por lo tanto continua con FALSO, la iteración del *while* sigue realizándose, luego que se ejecuta el *break* de finalización del *case* en cuestión. En cambio, si la *selección()* hubiera dado un resultado de tres, el tercer case, la convierte en CIERTO, con lo que se finaliza el *while* y el programa termina.

6. AMBITO DE LAS VARIABLES

Variables Globales

Las variables se pueden diferenciar según su "tipo" (el cual se refería a la cantidad de bytes que la conformaban como lo visto hasta ahora), pero también existe otra forma de diferenciación de las mismas, de acuerdo a la clase de memoria en la que residen.

Si se define una variable AFUERA de cualquier función (incluyendo esto a *main()*), se denomina a la misma VARIABLE GLOBAL. Este tipo de variable será ubicada en el segmento de datos de la memoria utilizada por el programa, y existirá todo el tiempo que esté ejecutándose este. Este tipo de variables son automáticamente inicializadas a CERO cuando el programa comienza a ejecutarse.

Son accesibles a todas las funciones que estén declaradas en el mismo, por lo que cualquiera de ellas podrá actuar sobre el valor de las mismas. Por ejemplo:

```
#include <stdio.h>
double una_funcion(void);
double variable_global ;
main()
{
    double i ;
    printf("%f", variable_global );          /* se imprimirá 0 */
    i = una_funcion() ;
    printf("%f", i);                          /* se imprimirá 1 */
    printf("%f", variable_global );          /* se imprimirá 1 */
    variable_global += 1 ;
    printf("%f", variable_global );          /* se imprimirá 2 */
    return 0 ;
}

double una_funcion(void)
{
    return( variable_global += 1 );
}
```

La *variable_global* está definida afuera de las funciones del programa, incluyendo al *main()*, por lo que le pertenece a TODAS ellas. En el primer *printf()* del programa principal se la imprime,

demostrándose que está automáticamente inicializada a cero .

Luego es incrementada por una *_funcion* () que devuelve además una copia de su valor, el cual es asignado a *i*, la que, si es impresa mostrará un valor de uno, pero también la *variable_global* ha quedado modificada, como lo demuestra la ejecución de la sentencia siguiente. Luego *main* () también modifica su valor, lo cual es demostrado por el *printf* () siguiente.

Esto permite deducir que dicha variable es de uso público, sin que haga falta que ninguna función la declare, para actuar sobre ella.

Puede resultar muy difícil evaluar el estado de las variables globales en programas algo complejos, con múltiples llamados condicionales a funciones que las afectan, dando comúnmente origen a errores muy engorrosos de corregir.

Variables Locales

A diferencia de las anteriores, las variables definidas DENTRO de una función, son denominadas **VARIABLES LOCALES** a la misma, a veces se las denomina también como AUTOMATICAS, ya que son creadas y destruidas automáticamente por la llamada y el retorno de una función, respectivamente.

Estas variables se ubican en la pila dinámica (*stack*) de memoria, destinándosele un espacio en la misma cuando se las define dentro de una función, y borrándose cuando la misma devuelve el control del programa, a quien la haya invocado.

Este método permite que, aunque se haya definido un gran número de variables en un programa, estas no ocupen memoria simultáneamente en el tiempo, y sólo vayan incrementando el *stack* cuando se las necesita, para luego, una vez usadas desaparecer, dejando al *stack* en su estado original.

El identificador ó nombre que se la haya dado a una variable es sólo relevante entonces, para la función que la haya definido, pudiendo existir entonces variables que tengan el mismo nombre, pero definidas en funciones distintas, sin que haya peligro alguno de confusión .

La ubicación de estas variables locales, se crea en el momento de correr el programa, por lo que no poseen una dirección prefijada, esto impide que el compilador las pueda inicializar previamente. Por ello, siempre es necesario recordar entonces que, si no se las inicializa expresamente en el momento de su definición, su valor será indeterminado (basura).

CREACIÓN DE BIBLIOTECAS DEL USUARIO

El siguiente *ejemplo* muestra como el usuario puede crear su propia biblioteca de funciones

```
/* biblioteca de funciones, guardar como biblio.h en el directorio INCLUDE */
#include<....>           /*se incluyen las bibliotecas estándar necesarias para las funciones dentro
                           de la biblioteca del usuario*/

int suma(int a,int b);           //prototipos de las funciones
int resta (int x, int y);
void muestra( int);

int suma(int a,int b)           //función que suma dos números
{
    return(a+b);
}
```

```
int resta (int x, int y)           //función que resta dos números
{
    return (x-y);
}

void muestra( int x)               //función que muestra un número
{
    printf( "\n%d", x);
}
```

```
//programa que usa las funciones de la biblioteca creada
#include<stdio.h>
#include<conio.h>
#include<biblio.h>                //biblioteca del usuario
main()
{
    int m=5,n=8;
    clrscr();
    muestra(suma(n,m));
    muestra(resta(n,m));
    getch();
    return 0;
}
```

Estructuras de agrupamiento de variables

1. CONJUNTO ORDENADO DE VARIABLES (ARRAYS)

Los arreglos ó conjuntos de datos ordenados (*arrays*) recolectan variables del MISMO tipo, guardándolas en forma secuencial en la memoria. La cantidad máxima de variables que pueden albergar está sólo limitada por la cantidad de memoria disponible. El tipo de las variables involucradas puede ser cualquiera de los ya vistos, con la única restricción de que todos los componentes de un *array* deben ser del mismo tipo.

La declaración de un *array* se realiza según la siguiente sintaxis:

tipo de las variables nombre[cantidad de elementos] ;

Por ejemplo :

```
int var1[10];
char nombre[50];
float numeros[200];
long double cantidades[25];
```

En el primer caso, se declara un *array* de 10 variables enteras, cada una de ellas quedará individualizada por el subíndice que sigue al nombre del mismo es decir:

var1[0], var1[1], etc. , hasta var1[9].

La CANTIDAD de elementos es 10, pero su numeración va de 0 a 9, y no de 1 a 10 . En resumen un array de N elementos tiene subíndices válidos entre 0 y N - 1. Cualquier otro número usado como subíndice, traerá datos de otras zonas de memoria , cuyo contenido es impredecible .

Se puede referenciar a cada elemento, en forma individual, tal como se ha hecho con las variables anteriormente , por *ejemplo* :

```
var1[5] = 40;
contador = var1[3] + 7;
if(var1[0] >= 37)
```

También es posible utilizar como subíndice expresiones aritméticas , valores enteros retornados por funciones , etc. . Por *ejemplo*:

```
printf(" %d ", var1[ ++i ] );
var1[8] = var1[ i + j ];
int una_funcion(void);
var1[0] = var1[ una_funcion() ] * 15;
```

Los subíndices resultantes de las operaciones tienen que estar acotados a aquellos para los que el **array** fue declarado y ser enteros.

La inicialización de los **arrays** sigue las mismas reglas que vimos para los otros tipos de variables, es decir: Si se declaran como **globales** (afuera del cuerpo de todas las funciones) cada uno de sus elementos será automáticamente inicializado a cero. Si en cambio, su declaración es local a una función, no se realiza ninguna inicialización, quedando a cargo del programa cargar los valores de inicio.

La inicialización de un **array** local, puede realizarse en su declaración, dando una lista de valores iniciales: `int numero[8] = { 4, 7, 0, 0, 0, 9, 8, 7 };` La lista debe estar delimitada por llaves.

Otra posibilidad, sólo válida cuando se inicializan todos los elementos del **array**, es escribir:

```
int numero[] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
```

donde se obvia la declaración de la cantidad de elementos, ya que está implícita en la lista de valores constantes .

También se puede inicializar parcialmente un **array** , por ejemplo : `int numero[10] = { 1, 1, 1 };` en éste caso los tres primeros elementos del mismo valdrán **1**, y los restantes cero en el caso que la declaración sea global , ó cualquier valor impredecible en el caso de que sea local .

2. CONJUNTO ORDENADO DE CARACTERES (STRINGS)

Los **strings** son simplemente arrays de caracteres con el agregado de un último elemento constante: el caracter **NULL** (**ASCII** = 0, simbolizado por la secuencia de escape `\0`). Este agregado permite a las funciones que procesan a los mismos, determinar fácilmente la finalización de los datos.

Es posible generar un **string**, declarando:

```
char car_str[] = { 'A', 'B', 'C', 'D', 0 };
char car_str[] = { 'A', 'B', 'C', 'D', '\0' };
```

Ambas maneras son equivalentes. Sin embargo hay, en el lenguaje C, una forma más compacta de declararlos:

```
char car_str[] = "ABCD";
char car_str[5] = "ABCD";
int texto[] = "renglon 1 \n renglon 2 \n "; /* ERROR */
unsigned char texto[] = "renglon 1 \n renglon 2 \n ";
```

Simplemente en la declaración del mismo se encierran los caracteres que lo componen entre

comillas dobles. En la segunda declaración, se ha explicitado (no es necesario), la cantidad de elementos que tiene el string, y es uno más que la cantidad de caracteres con que se lo inicializa, para dejar lugar al *NULL*. Todas éstas declaraciones agregan automáticamente el *NULL* como último elemento del *array*.

Un caso interesante es el de la tercera línea (comentada como *ERROR*), con el fin de poder albergar al carácter "\n" se intentó asignar el *string* a un *array de enteros*. Esto no es permitido por el compilador, que lo rechaza como una asignación inválida. La razón de ello se verá más adelante. La solución más común para este caso es, declarar el *array* como *unsigned char*, con lo que se lleva el alcance de sus elementos a 255. Si se diera el caso de tener que albergar en un *string* el carácter *EOF* y al mismo tiempo caracteres con *ASCII* mayor que 127, se podría definir el *array* como *int*, pero su inicialización se tendrá que hacer obligatoriamente usando llaves.

Se deduce entonces que un *string* sigue siendo un *array de caracteres*, con la salvedad del agregado de un terminador, por lo que las propiedades que se verán a continuación, se aplicaran indistintamente a ambos.

3. ARRAYS Y STRINGS COMO ARGUMENTOS DE FUNCIONES

Los *arrays*, como todos los otros tipos de variables, pueden ser pasados como argumentos a las funciones. La sintaxis sería la siguiente:

```
double funcion_1( float numeros[10] , char palabra[] );           /*línea 1*/

main()
{
float numeros[10] = { 1.1 , 2.2 , 3.0 };                         /*línea 2*/
char palabra [] = " Lenguaje C ";                                /*línea 3*/
double c;                                                         /*línea 4*/
.....
c = funcion_1( numeros , palabra);                               /*línea 5*/
.....
}
double funcion_1( float numeros [10] , char palabra [] )        /*línea 6*/
{
.....
}
```

En la primer línea se declara el prototipo de *funcion_1 ()* que recibe como argumentos dos *arrays*, uno de 10 elementos del tipo *float*, y otro de caracteres de longitud indeterminada. En el primer caso la función necesitará saber de alguna manera cual es la longitud del *array* numérico recibido, mientras que en el segundo, no hace falta, ya que la función puede ser construida para que, por sí misma, detecte la finalización del *string* por la presencia del carácter *NULL*. Se podría generalizar más el programa declarando: *double funcion_1(double numeros [] , int longitud_array , char palabra [])*; en donde, en la variable *longitud_array* se enviaría la cantidad de elementos de *numero []*.

En la tercer línea se declara el *array* numérico, inicializándose sólo los tres primeros elementos, y en la cuarta línea se declara el *string*. En la séptima línea se da la definición de la función, de acuerdo al prototipo escrito anteriormente. En la sexta línea se llama a la función y los argumentos pasados sólo tienen el *NOMBRE* de ambos *arrays*.

Esta es la diferencia más importante entre este tipo de estructura de datos y las variables simples vistas anteriormente, ya que los arrays son pasados a las funciones por DIRECCION y no por valor.

En el lenguaje C se prefiere, para evitar el uso abusivo del *stack*, cuando hay que enviar a una función una larga estructura de datos, en lugar de copiar a todos ellos, cargar el *stack* sólo con la dirección de la posición de memoria donde está ubicado el primero de los mismos.

El nombre de un array equivale sintácticamente a la dirección del elemento cero así será:

numero == dirección de numero[0]

palabra == dirección de palabra[0]

Esto habilita a las funciones a que puedan acceder a los *arrays* directamente, allí donde el programa los ha ubicado en la memoria, por lo que pueden MODIFICARLOS EN FORMA PERMANENTE aunque no hayan sido declarados como locales a la función misma ni globales al programa.

Es muy importante recordar este último concepto a fin de evitar errores muy comunes en los primeros intentos de programación en C.

Otra característica importante de los arrays es que, su nombre (ó dirección del primer elemento) es una CONSTANTE y no una variable. El nombre de los arrays implican para el compilador el lugar de memoria donde empieza la estructura de datos por lo que, intentar cambiar su valor es tomado como un error,

Así si se escribiera por ejemplo:

char titulo [] = "Primer titulo";

.....

titulo = "subtitulo";

La primer sentencia es correcta, ya que se esta inicializando al *string*, pero la segunda produciría un error del tipo " *LVALUE REQUERIDO* ", es decir que el compilador espera ver, del lado izquierdo de una expresión, a una variable y en cambio se ha encontrado con una constante titulo (o sea la dirección de memoria donde está almacenada la P de "Primer título"). Esto para el compilador es similar a una expresión de la clase: *124 = j* y se niega rotundamente a compilarla.

Ejemplo de funciones utilizando vectores.

```
#include<conio.h>
#include<stdio.h>
int aux,vec[10],i,j;                                //Variables globales

void ordenacion(int vec[10]);                        //prototipo de funciones
void muestra_ordenacion(int vec[10]);

main()
{
    clrscr ();
    for ( i = 0 ; i < 10 ; i++)
    {
        printf ("Ingrese un valor a la posición.. %d --->", i );
        scanf ( "%d" , &vec[i] ) ;
    }
    ordenacion (vec) ;                               //Llamo a la función que ordena
```

```

        muestra_ordenacion (vec) ;
        getch() ;
        return 0;
    }

    void ordenacion(int vec[10])                //función que ordena vectores
    {   for(i=0 ; i<10-1 ; i++)
        for(j=0 ; j<10-i ; j++)
        {   if (vec[j]>vec[j+1])
            {   aux=vec[j];
                vec[j]=vec[j+1];
                vec[j+1]=aux;
            }
        }
    }

    void muestra_ordenacion(int vec[10]) //función que muestra el vector ordenado
    {   for ( i=0 ; i<10 ; i++)
        printf("%d \t",vec[i]);
    }

```

4. ARRAYS MULTIDIMENSIONALES

Las estructuras de datos del tipo *array* pueden tener más de una dimensión, es bastante común el uso de *arrays* "planos" ó *matriciales* de dos dimensiones, por ejemplo:

int matriz[número total de filas] [número total de columnas] ;

Si se declara: *int matriz[3][4] ;* la disposición "espacial" de los elementos seria:

<i>columnas:</i>	<i>0</i>	<i>1</i>	<i>2</i>	<i>3</i>		
<i>filas</i>	<i>0</i>	<i>[0][0]</i>	<i>[0][1]</i>	<i>[0][2]</i>	<i>[0][3]</i>	<i>matriz [0] []</i>
	<i>1</i>	<i>[1][0]</i>	<i>[1][1]</i>	<i>[1][2]</i>	<i>[1][3]</i>	<i>matriz [1] []</i>
	<i>2</i>	<i>[2][0]</i>	<i>[2][1]</i>	<i>[2][2]</i>	<i>[2][3]</i>	<i>matriz [2] []</i>

También se pueden generar *arrays* de cualquier número de dimensiones.

Para inicializar arrays multidimensionales, se aplica una técnica muy similar a la ya vista, por ejemplo para dar valores iniciales a un array de caracteres de dos dimensiones, se escribirá:

```

char dia_de_la_semana[7][8] = {
    "lunes", "martes", "miercoles",
    "jueves", "viernes", "sábado",
    "domingo" };

```

El elemento *[0][0]* será la "l" de lunes , el *[2][3]* la "r" de miercoles , el *[5][2]* la "b" de sabado, etc. Los elementos *[0][5]* , *[1][6]* ,etc. están inicializados con el caracter *NULL* y demás *[0][6]* y *[0][7]*, etc. no han sido inicializados. Este último ejemplo también podría verse como un *array unidimensional de strings*.

5. ESTRUCTURAS

Declaracion De Estructuras

Así como los *arrays* son organizaciones secuenciales de variables simples de un mismo tipo cualquiera dado, resulta necesario en múltiples aplicaciones agrupar variables de distintos tipos en una sola entidad . Este sería el caso, si se quisiera generar la variable "legajo personal", en

ella se tendrían que incluir variables del tipo: *strings*: para el nombre, apellido, nombre de la calle en donde vive, etc., *enteros*: para la edad, número de código postal, *float* (o *double*, si tiene la suerte de ganar mucho) para el sueldo, y así siguiendo. Existe en C un tipo de variable compuesta, para manejar ésta situación típica de las Bases de Datos, llamada **ESTRUCTURA**. No hay limitaciones en el tipo ni cantidad de variables que pueda contener una *estructura*, mientras que la máquina posea memoria suficiente como para alojarla, con una sola salvedad: una *estructura* no puede contenerse a sí misma como miembro.

Para usarlas, se deben seguir dos pasos. Hay que: primero declarar la estructura en sí, esto es, darle un nombre y describir a sus miembros, para finalmente declarar a una ó más variables, del tipo de la estructura antedicha. Por ejemplo:

```
struct legajo {  
    int edad ;  
    char nombre[50] ;  
    float sueldo ;  
};  
struct legajo legajos_vendedores, legajos_profesionales ;
```

En la primera sentencia se crea un tipo de estructura, mediante el declarador "*struct*", luego se le da un nombre "*legajo*" y finalmente, entre llaves se declaran cada uno de sus miembros, pudiendo estos ser de cualquier tipo de variable, incluyendo a los *arrays* o alguna otra estructura. La única restricción es que no haya dos miembros con el mismo nombre, aunque sí pueden coincidir con el nombre de otra variable simple, (o de un miembro de otra estructura), declaradas en otro lugar del programa. Esta sentencia es sólo una declaración, es decir que no asigna lugar en la memoria para la estructura, sólo le avisa al compilador como tendrá que manejar a dicha memoria para alojar variables del tipo *struct legajo*.

En la segunda sentencia, se definen dos variables del tipo de la estructura anterior, (ésta definición debe colocarse luego de la declaración), y se reserva memoria para ambas. Las dos sentencias pueden combinarse en una sola, dando la definición a continuación de la declaración:

```
struct legajo {  
    int edad ;  
    char nombre[50] ;  
    float sueldo ;  
} legajo_vendedor , legajo_programador ;
```

Y si no fueran a realizarse más declaraciones de variables de éste tipo, podría obviarse el nombre de la estructura (*legajo*).

Las variables del tipo de una estructura, pueden ser inicializadas en su definición, así por ejemplo se podría escribir:

```
struct legajo {  
    int edad ;  
    char nombre[50] ;  
    float sueldo ;  
    char observaciones[500] ;  
} legajo_vendedor = { 40 , "Juan Eneene" , 1200.50 , "Asignado a zona A" } ;  
struct legajo legajo_programador = { 23 , "Jose Peres" , 2000.0 , "Asignado a zona B" } ;
```

Aquí se utilizaron las dos modalidades de definición de variables, inicializandolas a ambas .

6. ARRAYS DE ESTRUCTURAS

Es posible agrupar a las *estructuras* como elementos de un *array*. Por ejemplo :

```
typedef struct {
    char   material[50] ;
    int     existencia ;
    double costo_unitario ;
} Item ;

Item stock[100] ;
```

Anteriormente se definió un *array* de *100 elementos*, donde cada uno de ellos es una estructura del tipo *Item* compuesta por tres variables, un *int*, un *double* y un *string* ó array de 50 caracteres.

Los *arrays de estructuras* pueden inicializarse de la manera habitual, así en una definición de *stock*, se podría haber escrito:

```
Item stock1[100] = { "tornillos" , 120 , .15 ,
                    "tuercas" , 200 , .09 ,
                    "arandelas" , 90 , .01
                    } ;

Item stock2[] = { { 'i','t','e','m','1','\0' } , 10 , 1.5 ,
                  { 'i','t','e','m','2','\0' } , 20 , 1.0 ,
                  { 'i','t','e','m','3','\0' } , 60 , 2.5 ,
                  { 'i','t','e','m','4','\0' } , 40 , 4.6 ,
                  { 'i','t','e','m','5','\0' } , 10 , 1.2 ,
                  } ;
```

Las diferencias que existen entre la dos inicializaciones dadas son: en la primera, el *array material []* es inicializado como un *string*, por medio de las comillas y luego en forma ordenada, se van inicializando cada uno de los miembros de los elementos del *array stock1 []*, en la segunda se ha preferido dar valores individuales a cada uno de los elementos del *array material*, por lo que es necesario encerrarlos con llaves

Sin embargo hay una diferencia mucho mayor entre las dos sentencias, en la primera es explícito el tamaño del array , *[100]* , y sólo se inicializan los tres primeros elementos, los restantes quedarán cargados de basura si la definición es *local* a alguna función, ó de cero si es *global*, pero de cualquier manera están alojados en la memoria, en cambio en la segunda esta implícito el número de elementos, por lo que será el compilador el que calcule la cantidad de ellos, basándose en cuantos se han inicializado, por lo tanto este array sólo tendrá ubicados en memoria cuatro elementos, *sin posibilidad de agregar nuevos datos posteriormente*.

En muchos casos es usual realizar un alojamiento dinámico de las estructuras en la memoria, en razón de ello, y para evitar además la saturación de *stack* por el pasaje ó retorno desde funciones, es necesario conocer el tamaño, ó espacio en bytes ocupados por ella.

Es posible aplicar el operador *sizeof ()* , de la siguiente manera :

```
longitud_base_de_datos = sizeof ( stock1 ) ;
longitud_de_dato       = sizeof ( Item ) ;
cantidad_de_datos      = sizeof ( stock1 ) / sizeof ( Item ) ;
```

Con la primera se calcula el tamaño necesario de memoria para albergar a todos datos, en la segunda la longitud de un sólo elemento (*record*) y por supuesto dividiendo ambas, se obtiene la cantidad de records.

Apuntadores y arreglos

1. INTRODUCCION A LOS PUNTEROS

Los *apuntadores* o *punteros* en el Lenguaje C, son variables que "apuntan", es decir que poseen la dirección de las ubicaciones en memoria de otras variables, y por medio de ellos se tiene un poderoso método de acceso a todas ellas.

Como se declara un puntero:

```
tipo de variable apuntada *nombre_del_puntero ;
int *pint ;
double *pfloat ;
char *letra , *codigo , *caracter ;
```

En estas declaraciones sólo se le dice al compilador que reserve una posición de memoria para albergar la dirección de una variable, del tipo indicado, la cual será referenciada con el nombre que se le haya dado al puntero.

Obviamente, un puntero debe ser inicializado antes de usarse, y una de las eventuales formas de hacerlo es la siguiente:

```
int var1 ;           /* se declara ( y crea en memoria ) una variable entera ) */
int *pint ;          /* " " " un puntero que contendrá la dirección de una variable entera */
pint = &var1 ;       /* escribe en la dirección de memoria donde está el puntero la dirección de la
                      variable entera */
```

Como se mencionó anteriormente "*&nombre_de_una_variable*" implica la dirección de la misma. En realidad en la declaración del puntero, está implícita otra información: cual es el tamaño (en bytes) de la variable apuntada.

El símbolo *&*, ó *dirección*, puede aplicarse a variables, funciones, etc., pero no a constantes ó expresiones, ya que éstas no tienen una posición de memoria asignada.

La operación inversa a la asignación de un puntero, de referenciación del mismo, se puede utilizar para hallar el valor contenido por la variable apuntada. Así por ejemplo serán expresiones equivalentes:

```
y = var1 ;
y = *pint ;
printf("%d", var1 ) ;
printf("%d", *pint) ;
```

En estos casos, la expresión "**nombre_del_puntero*", implica "contenido de la variable apuntada por el mismo". *Por ejemplo*:

```
#include <stdio.h>
main()
{
    char var1;                               /*una variable del tipo caracter */
    char *pchar;                             /* un puntero a una variable del tipo caracter */
    pc = &var1 ;                             /*se asigna al puntero la direccion de la variable */
    for (var1 = 'a'; var1 <= 'z'; var1++)
        printf("%c", *pchar);               /* se imprime el valor de la variable apuntada */
    return 0 ;
}
```

En el *for* se incrementa el valor de la variable, y luego para imprimirla se usa la

dereferenciación de su puntero. El programa imprimirá las letras del abecedario de la misma manera que lo habría hecho si la sentencia del `printf()` hubiera sido, `printf("%c", var1)`.

Hay un error, que se comete con bastante frecuencia, y es cargar en la dirección apuntada por un puntero a un tipo dado de variable, el contenido de otro tipo de las mismas, *por ejemplo*:

```
double d = 10.0 ;
int i = 7 , *pint ;
pint = &i ;
*pint = 10 ;      /* correcto, equivale a asignar a i el valor 10 */ ;
*pint = d ;      /* ERROR se pretende cargar en una variable entera un valor double */
pint = &d ;      /* INCORRECTO se pretende apuntar a una variable double con un puntero
                    declarado como apuntador a int */
pint = 4358 ;     /* ?????? */
```

El primer error, la asignación de un *double*, produce la pérdida de información dada por la conversión automática de tipo de variable, ya vista anteriormente, el segundo produce un llamado de atención rotulado como "*asignación sospechosa de un pointer*". Resumiendo, las variables ó constantes cargadas por dereferenciación de un puntero, deben coincidir en tipo con la declaración de aquel.

La asignación de una constante a un *pointer*, y no a la variable apuntada por él, es un serio error, ya que debe ser el compilador, el encargado de poner en él el valor de la dirección, aquel así lo declara dando un mensaje de "*conversión de puntero no transportable*". Si bien lo compila, ejecutar un programa que ha tenido esta advertencia es similar a jugar a la ruleta rusa, puede "colgarse" la máquina ó lo que es peor destruirse involuntariamente información contenida en un disco, etc.

Hay un sólo caso en el que esta asignación de una constante a un puntero es permitida , muchas funciones para indicar que no pueden realizar una acción ó que se ha producido un error de algún tipo, devuelven un puntero llamado "*Null Pointer*", lo que significa que no apunta a ningún lado válido , dicho puntero ha sido cargado con la dirección *NULL* (por lo general en valor 0), así la asignación: `pint = NULL`; es válida y permite luego operaciones relacionales del tipo `if (pint)` ó `if (pint != NULL)` para convalidar la validez del resultado devuelto por una función .

Se debe tener en cuenta que los punteros no son enteros, como parecería a primera vista, ya que el número que representa a una posición de memoria, sí lo es. Debido al corto alcance de este tipo de variable, algunos compiladores pueden, para apuntar a una variable muy lejana, usar cualquier otro tipo, con mayor alcance que el antedicho.

2. PUNTEROS Y ARRAYS

Hay una relación muy cercana entre los *punteros* y los *arrays*. El *designador* (ó nombre de un array) es equivalente a la dirección del *elemento [0]* del mismo. La explicación de esto es ahora sencilla: el nombre de un *array*, para el compilador C, es un *PUNTERO* inicializado con la dirección del primer elemento del array. Sin embargo hay una importante diferencia entre ambos.

Algunas operaciones permitidas entre punteros:

ASIGNACION

```
float var1 , conjunto[] = { 9.0 , 8.0 , 7.0 , 6.0 , 5.0 };
float *punt ;
punt = conjunto ;      /* equivalente a hacer : punt = &conjunto [0] */
var1 = *punt ;
*punt = 25.1 ;
```

Es perfectamente válido asignar a un puntero el valor de otro, el resultado de ésta operación es cargar en el puntero *punt* la dirección del *elemento [0]* del array *conjunto*, y posteriormente en la variable *var1* el valor del mismo (9.0) y luego cambiar el valor de dicho primer elemento a 25.1.

La diferencia entre un puntero y el denominador de un array es: el primero es una VARIABLE, es decir que se puede asignar, incrementar, etc., en cambio el segundo es una CONSTANTE, que apunta siempre al primer elemento del array con que fue declarado, por lo que su contenido NO PUEDE SER VARIADO.

El siguiente *ejemplo* muestra un tipo de error bastante frecuente:

ASIGNACION ERRONEA

```
int conjunto [ 5 ], lista [ ] = { 5, 6, 7, 8, 0 };
int *apuntador ;
apuntador = lista ;           // correcto
conjunto = apuntador;        // ERROR ( se requiere en Lvalue no constante )
lista = conjunto ;           // ERROR ( idem )
apuntador = &conjunto        //NO se puede aplicar el operador & (dirección) a una constante
```

3. PUNTEROS A STRINGS

No hay gran diferencia entre el trato de punteros a *arrays* y a *strings*, ya que estos dos últimos son entidades de la misma clase. Sin embargo existen algunas particularidades. Así como se inicializa un *string* con un grupo de caracteres terminados en '\0', se puede asignar al mismo un puntero:

```
p = "Esto es un string constante " ;
```

Esta operación no implica haber copiado el texto, sino sólo que a *p* se le ha asignado la dirección de memoria donde reside la *"E"* del texto. A partir de ello se puede manejar a *p* como se lo ha hecho hasta ahora. Por *ejemplo*

```
#include <stdio.h>
#define TEXTO1 "¿ Hola , como "
#define TEXTO2 "le va a Ud. ? "
main()
{
    char palabra [20] , *p ;
    int i ;
    p = TEXTO1 ;
    for ( i = 0 ; ( palabra[i] = *p++ ) != '\0' ; i++ ) ;
    p = TEXTO2 ;
    printf ("%s" , palabra ) ;
    printf ("%s" , p ) ;
    return 0 ;
}
```

Se definen primero dos *strings* constantes *TEXTO1* y *TEXTO2*, luego se asigna al puntero *p* la dirección del primero, y seguidamente en el *for* se copia el contenido de éste en el *array palabra*, dicha operación termina cuando el contenido de lo apuntado por *p* es el terminador del *string*, luego se asigna a *p* la dirección de *TEXTO2* y finalmente se imprimen ambos *strings*, obteniendo una salida del tipo: "¿Hola, como le va a UD.? " (espero que bien) .

Esto se podría haber escrito más compacto, si se hubiera recordado que *palabra* también es un puntero y

NULL es cero, así se puede poner en vez del *for*, la sentencia *while(*palabra++ = *p++);*

Aquí se ha agregado muy poco a lo ya sabido, sin embargo hay un tipo de error muy frecuente, que se puede analizar en el *ejemplo* siguiente:

(*CON ERRORES*)

```
#include <stdio.h>
main()
{
    char *p , palabra[20] ;
    printf ( "Escriba su nombre : " );
    scanf ( "%s" , p );
    palabra = "¿ Como le va " ;
    printf ( "%s%s" , palabra , p );
}
```

Pues hay dos errores, a falta de uno, el primero ya fue analizado antes, la expresión *scanf("%s",p)* es correcta pero, el error implícito es no haber inicializado al puntero *p*, el cual sólo fue definido, pero aun no apunta a ningún lado válido. El segundo error está dado por la expresión: *palabra="¿ Como le va "*, (también visto anteriormente) ya que el nombre del *array* es una constante y no puede ser asignado a otro valor .

(*CORRECTO*)

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
main()
{
    char *p , palabra[20] ;
    p = (char *) malloc (sizeof(char)128) ;
    printf ( "Escriba su nombre : " );
    scanf ( "%s" , p );
    strcpy ( palabra , "¿ Como le va " );
    printf ( "%s%s" , palabra , p );
}
```

Antes de *scanf()* se ha inicializado a *p*, mediante el retorno de *malloc()* y a al *array palabra* se le copiado el *string* mediante la función vista anteriormente *strcpy()*.

Cabe aclarar también que, la secuencia de control *%s* en *el printf()* impone enviar a la pantalla un *string*, estando éste apuntado por el argumento siguiente al control, éste puede ser tanto el nombre de un *array*, como un *puntero*, ya que ambos explicitan direcciones.

Una forma alternativa de resolverlo, sería:

(*CORRECTO*)

```
#include <stdio.h>
main()
{
    char p[20] , *palabra ;
    printf ( "Escriba su nombre : " );
    scanf ( "%s" , p );
    palabra = "¿ Como le va " ;
    printf("%s%s" , palabra , p );
}
```

Esto es idéntico al primero, con la salvedad que se han invertido las declaraciones de las variables, ahora el *puntero* es *palabra* y el *array* es *p*. Ambas soluciones son equivalentes y dependerá del resto del programa cual es la mejor elección.

4. ARRAYS DE PUNTEROS

Es una práctica muy habitual, sobre todo cuando se tiene que tratar con *strings* de distinta longitud, generar *array* cuyos elementos son punteros, que albergarán las direcciones de dichos *strings*. Así como: *char *flecha;* definía a un puntero a carácter, la definición *char *carcaj [5];* implica un array de 5 punteros a caracteres .

Inicializacion De Arrays De Punteros

Los *arrays de punteros* pueden ser inicializados de la misma forma que un array común, es decir dando los valores de sus elementos, durante su definición, por ejemplo si se quisiera tener un *array* donde el subíndice de los elementos coincidiera con el nombre de los días de la semana, se podría escribir:

```
char *dias [] = {  
    "número de día no válido",  
    "lunes",  
    "martes",  
    "miercoles",  
    "jueves",  
    "viernes",  
    "sabado",  
    "por fin es domingo"  
}
```

Igual que antes, no es necesario en este caso indicar la cantidad de elementos, ya que el compilador los calcula por la cantidad de términos dados en la inicialización. Así el elemento *dias[0]* será un puntero con la dirección del primer string, *dias[1]*, la del segundo, etc.

Funciones de manejo de strings

1. INTRODUCCION

La mayoría de las funciones que se verán a continuación, responden a la norma *ANSI C*, por lo que serán independientes del compilador que usemos. Estas tienen sus prototipos definidos en los archivos de encabezamiento *stdio.h*, *stdlib.h*, *string.h* y *ctype.h*.

Algunas de las características básicas de los strings. Estos pueden aparecer en un programa como una constante simbólica, de la forma siguiente: *#define TITULO "Capítulo 9"*, en este caso, en cada lugar donde aparece *TITULO* se reemplazará esta constante simbólica por la DIRECCION de la C del texto con que fue definida . Así, será correcto escribir: *char *p = TITULO ;*

2. FUNCIONES DE CONVERSION ENTRE STRING Y VARIABLES NUMERICAS

Puede darse el caso que la información a ingresarse a un programa ejecutable, por el operador pueda ser en algunos casos un valor numérico y en otros un *string de caracteres*. Un problema típico se presento en el ejemplo en que se ingresaba a la base de datos un articulo, ya sea por su nombre ó por su número de código.

Más cómodo que escribir dos instancias del programa, una para cada una de las opciones, es

dejar que el operador escriba lo que quiera, leyéndolo como un *string*, luego verificar si éste está compuesto exclusivamente por números ó posee algún caracter no numérico.

Para evaluar si un *string* cae dentro de una categoría dada, es decir si está compuesto exclusivamente por números, letras, mayúsculas, minúsculas, caracteres alfanuméricos, etc. existen una serie de funciones, algunas de las cuales ya hemos usado, que describimos a continuación. Estas deben ser usadas con los strings, analizando caracter a caracter de los mismos, dentro de un *ciclo*:

```
for (i=0 ; palabra [ i ] != NULL ; i++)
{ if (isalnum(palabra[i])
..... }
```

IS.....()

- Header: <ctype.h>
- Prototipo:

```
int isalnum( int c )   int isalpha( int c )   int isascii( int c )   int iscntrl( int c )
int isdigit( int c )   int islower( int c )   int isupper( int c )   int ispunct( int c )
int isspace( int c )   int isxdigit( int c )
```

- Descripción: Retornarán un valor CIERTO (distinto de cero) si el caracter enviado como argumento cae dentro de la categoría fijada para la comparación y FALSO ó cero en caso contrario. Las categorías para cada función son las siguientes :

La Función	Retorna CIERTO si c es:
isalnum(c)	Alfanumérico (letras ó números)
isalpha(c)	Alfabético, mayúscula ó minúscula
isascii(c)	Si su valor está entre 0 y 126
iscntrl(c)	Si es un caracter de control cuyo ASCII está comprendido entre 0 y 31 ó si es el código de "delete", 127 .
islower(c)	Si es un caracter alfabético minúscula.
isupper(c)	Si es un caracter alfabético mayúscula
isdigit(c)	Si es un número comprendido entre 0 y 9
ispunct(c)	Si es un caracter de puntuación
isspace(c)	Si es el caracter espacio, tabulador, avance de línea, retorno de carro, etc.
isxdigit(c)	Si es código correspondiente a un número hexadecimal, es decir entre 0-9 ó A-F ó a-f

Una vez que se sabe que un string está compuesto por números, se puede convertir en una variable numérica, de cualquier tipo, para poder realizar con ella los cálculos que correspondan

ATOI(), ATOL(), ATOF()

- Header: <stdlib.h>
- Prototipo:

```
int  atoi( const char *s )
long atol( const char *s )
double atof( const char *s )
```

- Descripción: Convierten el *string* apuntado por *s* a un número. *atoi()* retorna un entero, *atol()* un long y *atof()* un double. (Algunos compiladores traen una función adicional, *_atold()* que retorna un *long double*). El *string* puede tener la siguiente configuración:

[espacios , blancos , tabulador , etc.] [signo] xxx donde xxx son caracteres entre 0 y 9, para *atoi()* y *atol()*. Para *atof()* en cambio, se aceptan :
[espacios , etc.] [signo] xxx[.][xxx] ó [espacios , etc.] [signo] xxx[.] [xxx] [e ó E [signo] xxx]

según se desee usar la convención de punto flotante ó científica.

Es posible también, aunque menos frecuente, realizar la operación inversa, es decir, convertir un número en un string.

ITOA(), ULTOA()

- Header: <stdlib.h>
- Prototipo:

```
char *itoa( int numero , char *s , int base )
char *ultoa( unsigned long numero , char *s , int base )
```

- Descripción: Retornan un puntero a un *string* formado por caracteres que representan los dígitos del número enviado como argumento. Por base se entiende la de la numeración en la que se quiere expresar el *string*, *10 para decimal*, *8 para octal*, *16 para hexadecimal*, etc. *itoa()* convertirá un entero, mientras *ultoa()* lo hará con un *unsigned long*.

3. DETERMINACION DE LA LONGITUD DE UN STRING

STRLEN(), FSTRLEN

- Header: <string.h>
- Prototipo:

```
size_t strlen( const char *s )
size_t far _fstrlen( const char far *s )
```

- Descripción: Retornan un *entero* con la cantidad de caracteres del *string*. No toma en cuenta al terminador *NULL*. Por lo que la memoria real necesaria para albergar al string es *1+strlen(s)*. *_fstrlen()* da idéntico resultado, pero acepta como argumento un puntero *"far"*.
- Ejemplo:

```
char s[128];
gets(s);
p = (char *)malloc( sizeof( strlen(s) + 1 ) );
```

4. COPIA Y DUPLICACION DE STRINGS

El operador de asignación no está definido para *strings*, es decir que hacer *p = q*, donde *p* y *q* son dos arrays, no produce la copia de *q en p* y directamente la expresión no es compilada. Si en cambio *p* y *q* son *dos punteros a caracteres*, la expresión es compilada, pero no produce el efecto de copia, simplemente, cambia el valor de *p*, haciendo que apunte al MISMO string que *q*. Es decir que no se genera uno nuevo, por lo que todo lo operado sobre *p* afectará al original, apuntado por *q*.

Para generar entonces, una copia de un *string* en otro lugar de la memoria, se deben utilizar alguna de las funciones abajo descriptas.

Hay que diferenciar la copia de la duplicación: la primera copia un string sobre un lugar PREVIAMENTE reservado de memoria (mediante *malloc()*, *calloc()* ó alguna otra función de alocaión), en cambio la duplicación GENERA el espacio para guardar al nuevo *string* así creado.

STRCPY()

- Header: <string.h>
- Prototipo: *char *strcpy(char *destino , const char *origen)*
- Descripción: Copia los caracteres del string "origen", incluyendo el NULL, a partir de la dirección apuntada por "destino". No verifica que haya suficiente memoria reservada para tal efecto, por lo que el programador debe ubicar previamente suficiente memoria como para albergar a una copia de "origen". Aunque es superfluo, retorna el puntero "destino".

Existe también una función para realizar la copia PARCIAL. Por lo general las funciones que realizan acciones sobre una parte solamente, de los strings, llevan el mismo nombre de las que los afectan totalmente, pero con la adición de la letra "n".

STRNCPY()

- Header: <string.h>
- Prototipo: *char *strncpy(char *destino , const char *origen , size_t n_char)*
- Descripción: Copia *n_char* caracteres del string "*origen*", NO incluyendo el *NULL*, si la cantidad de caracteres copiada es menor que *strlen(origen) + 1*, en la dirección apuntada por "*destino*". *n_char* es un número entero y deberá ser menor que la memoria reservada apuntada por *destino*.
- Ejemplo:

```
#include <string.h>

main()
{
    char strvacio[11];
    char storigen[] = "0123456789";
    char strdestino[] = "ABCDEFGHILJ";
    .....
    strncpy( strdestino , storigen , 5 );
    strncpy( strvacio , storigen , 5 );
    strvacio[5] = '\0';
    .....
}
```

Los strings quedarían , luego de la copia :

Strdestino [] = 0 , 1 , 2 , 3 , 4 , F , G , H , I , J , \0

Strvacio [] = 0 , 1 , 2 , 3 , 4 , \0 , indefinidos

En el caso de *strdestino* no hizo falta agregar el *NULL*, ya que éste se generó en la inicialización del mismo, en cambio *strvacio* no fue inicializado, por lo que para terminar el *string*, luego de la copia, se deberá forzosamente agregar al final del mismo.

La función siguiente permite la duplicación de *strings* :

STRDUP()

- Header: <string.h>
- Prototipo: *char *strdup(const char *origen)*
- Descripción: Duplica el contenido de "*origen*" en una zona de memoria por ella reservada y retorna un puntero a dicha zona. El retorno de la función debe ser siempre asignado a un dado puntero.
- Ejemplo :

```
#include <string.h>

main()
{
    char *p ;
    char q[] = "Duplicación de strings";
    p = strdup( q );
    .....
}
```

5. CONCATENACION DE STRINGS

Se puede, mediante la concatenación de dos ó más strings, crear otro, cuyo contenido es el agregado del de todos los anteriores.

La concatenación de varios strings puede anidarse de la siguiente manera: *strcat(strcat(x, w), z)*; en la cual al *x* se le agrega a continuación el *w*, y luego el *z*. Por supuesto *x* tiene que tener suficiente longitud como para albergarlos.

STRCAT()

- Header: <string.h>
- Prototipo: *char *strcat(char *destino , const char *origen)*
- Descripción: Agrega el contenido de "origen" al final del string inicializado "destino", retornando un puntero a este.
- Ejemplo :

```
#include <string.h>
char z[20];
main()
{
    char p[20];
    char q[] = "123456789";
    char w[] = "AB";
    char y[20] = "AB";
    strcat(y, q);          /* OK, el contenido de y[] será: y[] == A,B,1,2,3,4,5,6,7,8,9,\0 */
    strcat(z, q);          /* OK, por ser global z[] quedó inicializado con 20 NULLS por lo que
                           luego de la operación quedará: z[] == 1,2,3,4,5,6,7,8,9,\0 */

    strcat(p, q);          /* Error! p no ha sido inicializado por lo que la función no encuentra
                           el NULL para empezar a agregar, por lo que barre la memoria
                           hasta encontrar alguno, y ahí escribe con resultados, generalmente
                           catastróficos. */

    strcat(w, q);          /* Error ! w solo tiene 3 caracteres, por lo el resultado final será: w[]
                           == A,B,1 sin la terminación del NULL por lo que cualquier próxima
                           operación que se haga utilizando este array, como string, fallará
                           rotundamente. */
}
```

STRNCAT()

- Header: <string.h>
- Prototipo: *char *strncat(char *destino , const char *origen , size_t cant)*
- Descripción: Similar en un todo a la anterior, pero solo concatena cant caracteres del string "origen" en "destino".

6. COMPARACION DE STRINGS

No se debe confundir la comparación de strings, con la de punteros, es decir *if (p == q) { ... }* sólo dará CIERTO cuando ambos apunten al MISMO string, siempre y cuando dichos punteros sean "near" ó "huge". El caso que se evaluara es más general, ya que involucra a dos ó más strings ubicados en distintos puntos de la memoria (abarca el caso anterior , como situación particular).

STRCMP()

- Header: <string.h>
- Prototipo: *int strcmp(const char *s1 , const char *s2)*
- Descripción: Retorna un entero, cuyo valor depende del resultado de la comparación
 - < 0 si s1 es menor que s2
 - == 0 si s1 es igual a s2
 - > 0 si s1 es mayor que s2

La comparación se realiza caracter a caracter, y devuelve el resultado de la realizada entre los primeros dos que sean distintos. La misma se efectúa tomando en cuenta el código *ASCII* de los caracteres así será por ejemplo: '9' < 'A', 'A' < 'Z' y 'Z' < 'a'.

STRCMP()

- Header: <string.h>
- Prototipo: *int strcmp(const char *s1 , const char *s2)*
- Descripción: Retorna un entero, de una manera similar a la anterior pero no es sensible a la diferencia entre mayúsculas y minúsculas, es decir que en este caso 'a' == 'A' y 'Z' > 'a'.

STRNCMP() , STRNCMPI()

- Header: <string.h>
- Prototipo: *int strncmp(const char *s1 , const char *s2 , size_t cant)*
- Descripción: Retornan un entero, con características similares a las de las funciones hermanas, descritas arriba, pero solo comparan los primeros "cant" caracteres .

7. BUSQUEDA DENTRO DE UN STRING

Muchas veces dentro de un programa, se necesita ubicar dentro de un string, a un determinado caracter ó conjunto ordenado de ellos . Para simplificar la tarea existen una serie de funciones de Librería , que toman por su cuenta la resolución de este problema :

STRSTR()

- Header: <string.h>
- Prototipo: *char *strstr(const char *s1 , const char *s2)*
- Descripción: Busca dentro de s1 un substring igual a s2, devolviendo un puntero al primer caracter del substring. Cualquier caracter contenido en s2, dentro de s1 .

STRCHR() Y STRRCHR()

- Header: <string.h>
- Prototipo:
*char *strchr(const char *s1 , int c)*
*char *strrchr(const char *s1 , int c)*
- Descripción: Retornan un puntero, a la posición del caracter dentro del string, si es que lo encuentran, ó en su defecto *NULL*. *strchr()* barre el string desde el comienzo, por lo que marcará la primer aparición del caracter en él, en cambio *strrchr()* lo barre desde el final, es decir que buscará la última aparición del mismo. El terminador *NULL* es considerado como parte del *string* .

8. FUNCIONES DE MODIFICACION DE STRING

Resulta conveniente a veces uniformizar los string leídos de teclado, antes de usarlos, un caso típico es tratar de independizarse de la posibilidad de que el operador escriba, algunas veces con mayúscula y otras con minúscula.

STRLOWER() Y STRUPR()

- Header: <string.h>
- Prototipo:
*char *strlwr(char *s1)*
*char *strupr(char *s1)*
- Descripción: Convierten respectivamente los caracteres comprendidos entre *a* y *z* a minúsculas ó mayúsculas, los restantes quedan como estaban.

FICHEROS EN C.

En C hay 2 tipos de ficheros:

- *Los ficheros de alto nivel:* en los que la escritura se hace carácter a carácter y que sirven tanto para almacenar textos como para almacenar campos.
- *Los ficheros de bajo nivel o ficheros binarios:* son los que se basan en el uso del buffer, contadores y tamaños a la hora de leer y escribir datos

Nosotros vamos a centrarnos en los ficheros de alto nivel

Los ficheros de alto nivel: A continuación vamos a ver las acciones más comunes que se realizan cuando trabajamos con un fichero.

1. Abrir ficheros:

Lo primero que hacemos al trabajar con ficheros es abrirlo. Esta es una instrucción obligatoria. El formato de apertura es: *fopen("nombre.ext", modo);* los modos son:

- *r* (read) de solo lectura.
- *w* (write) es de escritura. Sin embargo hay que tener en cuenta que utilizamos *l* para abrir un documento nuevo. De hecho si abrimos un documento existente con formato *n*, machacaremos su contenido.
- *a* (append). Es de escritura al final del fichero. Si no existe un fichero con ese nombre lo crea y empieza a escribir. Si existe, empieza a escribir al final del fichero, por tanto no pierde datos.
- *rw* de lectura y de escritura. Tampoco pierde datos.

Cada modo cumple su función. Por ejemplo, con *a* introducimos nuevos datos; con *r* podemos buscar; y con *rw* podemos modificar. Hay que tener en cuenta una cosa. Para modificar el modo, esto es, si hemos abierto un fichero en tipo *r* y queremos acceder a él de modo *rw*, primero hemos de cerrar y luego abrir en otro modo.

Además, si queremos abrir desde otra unidad o desde un directorio diferente, tendremos que consignar la ruta absoluta; *fopen("a:\...\...\nombre.ext", modo)*

Para evitar errores, no se suele utilizar directamente el *fopen* de una manera tan directa, sino, más bien, del siguiente modo...

```
If ((archivo1=fopen("alumnos.dat", "r"))== NULL)
{
    printf("el archivo no se puede abrir");
    abort();
}
```

Si se ha podido cargar el archivo, a partir de ese momento, a nivel interno, el fichero se llama *archivo1*, que tiene que estar previamente definido como una variable de tipo *FILE*, en mayúsculas y tal y como sigue: *FILE *archivo1;*

2. Cerrar ficheros:

fclose() es una instrucción obligatoria. Su sintaxis cubre 2 supuestos:

fclose(archivo1); cierra el *archivo1*. siempre se ha indicar el nombre interno del archivo.

Fclose(); cierra todos los archivos.

3. Escribir en un fichero:

podemos utilizar diferentes variantes de put, en funcion de lo que queramos escribir:

- *Putc* para un solo carácter.
- *Puts* para una cadena de caracteres.
- *Putw* para un valor numérico.

La sintaxis de put es: *put*(variable, archivo1);*

Pero si queremos también podemos escribir mediante *fprintf*, siendo su sintaxis:

fprintf(archivo, "formato escritura", variables);

por ejemplo...

fprintf(archivo1, "%20s_%2s_%30s_%9s", nombre, curso, direccion, DNI);

así este registro ocupará= *20+(1)+2+(1)+30+(1)+9= 64 espacios*. Este dato es muy importante para luego colocar el cursor etc.

además, por comodidad, se puede acabar por escribir con *\n*. De este modo, escribiremos de registro en registro, con lo que cada línea será de un registro. De lo contrario C meterá todos los caracteres quen puede por línea antes de saltar a la siguiente, con loo que se destroza la alineacióbñ de los caracteres.

A efectos de examen es importatnte tener en cuenta que a la hora de leer o de escribir hay que respetar todos los espacios y todos los datos. Así hay que recordar que se leen y se escriben todos los datos.

4. Leer en un fichero:

podemos utilizar diferentes variantes de get, en funcion de lo que queramos escribir:

- *getc* para un solo carácter.
- *gets* para una cadena de caracteres.
- *getw* para un valor numérico de cualquier tipo.

La sintaxis de put es: *variable=get*(archivo);*

Pero si queremos también podemos realizar la lectura gráfica, –de golpe–, de un fichero mediante *fscanf*, siendo su sintaxis: *fscanf(archivo, "formato escritura", variables);*

por ejemplo... *fscanf(archivo1, "%20s_%2s_%30s_%9s", nombre, curso, direccion, DNI);*

No hay que olvidar que: las *fscanf*, al igual que *scanf* lleva siempre *&* en las variables, excepto en las de tipo *cadena*s. Estas tienen que ser declaradas con un tamaño constante determinado, por ejemplo: *char palabra[20]* al leer, toma del fichero en cuestión y lo guarda en las variables. A nivel externo no hace nada. Para ver algo, hemos de *printf* las variables anteriores en una orden posterior.

Hay que tener mucho cuidado a la hora de intentar leer del fichero más allá de su final. Para evitar errores de lectura, la expresión correcta deberá de ser.

```

Variable= get(archivo1)
While(variable!=eof)
{
    variable=gets(archivo1);
}

```

y con *fscanf*

```

while fscanf (archivo1, "%20s_%2s_%30s_%9s", nombre, curso, direccion, DNI) != eof)
{
    código resultante
}

```

5. fseek:

coloca el puntero en un sitio determinado del fichero. Sirve para buscar dentro de una archivo Su sintaxis es: *fseek(archivo1, desplazamiento, origen)*

El desplazamiento se consigna en número de bytes. Por su parte, los valores posibles de origen son:

- *0* inicio del archivo.
- *1* posición final del archivo.
- *2* posición inicial del archivo.

Por ejemplo, *fseek (archivo1, 64,1);*

De este modo se colocará al final del registro actual, en caso de que el puntero estuviese al inicio de ese registro.

6. fgetpos:

Esta instrucción devuelve la posición actual del puntero en el fichero. Devuelve el número de byte en el que estoy. El valor lo devuelve en la variable posición, que habrá que declarar de un modo un tanto especial. *fpost_t posición;*

Su sintaxis es: *fgetpos(archivo, &posicion);*

Más adelante, en el mismo programa podremos trabajar con la variable posición.

7. Unlink:

sirve para borrar, ficheros. Su sintaxis es *unlink("nombre.ext");* el fichero sirve a borrar ha de estar cerrado, sino dará error y se bloqueará.

Entonces la cuestión es; ¿cómo borrar registros y datos dentro un fichero? Creando un fichero nuevo, copiando los registros que no voy a borrar, cerrando el fichero de origen, destruyendo el fichero de origen, y renombrando a la copia con el nombre del fichero de origen.

8. Rename

sirve para cambiar de nombre. Su sintaxis es *rename("viejo.ext", "nuevo.ext");*

9. Uso de estructuras de datos y guardar en archivos

Nota: el ejemplo es válido para variables tipo cadenas que no contengan espacios en blanco

```
#include<conio.h>
#include<stdio.h>
#include<io.h>
main()
{
    FILE *archi;                //Declaro un puntero al archivo
    struct{ char *nombre;       //Defino los campos de la estructura
            int punt;
        } record;
    clrscr ();
    if ((archi=fopen("a:\\record.txt","a"))!=NULL) //pregunto si el archivo esta creado
        archi=fopen("a:\\record.txt","a");        // Abro el archivo para añadir datos.
    else archi=fopen("a:\\record.txt","w");         // Creo el archivo
    scanf(" %s %d",record.nombre,&record.punt);     //lee los datos
    fprintf(archi," %s %d",record.nombre,record.punt) ; //graba en el archivo
    fclose(archi);                                  //cierro el archivo
    archi=fopen("a:\\record.txt","r+");              //abro el archivo como solo lectura
    rewind(archi);                                  //Se coloca el puntero al principio del
    archivo
    do
    {
        fscanf(archi," %s %d",record.nombre,&record.punt); //leo del archivo
    }
```

```

        printf("los datos son %s %d \n",record.nombre,record.punt); //muestro los datos
    }
    while (!feof(archi));                                     //Este ciclo se repite hasta final de archivo
    fclose (archi);                                           //cierro el archivo
    getch();
    return(0);
}

```

IMPRESORAS EN C.

Para utilizar una impresora en C, hay que seguir una serie de pasos:

Hay que definir impresora. Esto lo expresamos del siguiente modo.

```

FILE *impre;
main()

```

Antes de usarla, hay que abrir el canal de comunicación:

```

Impre= fopen("prn","w");

```

Imprimimos...

```

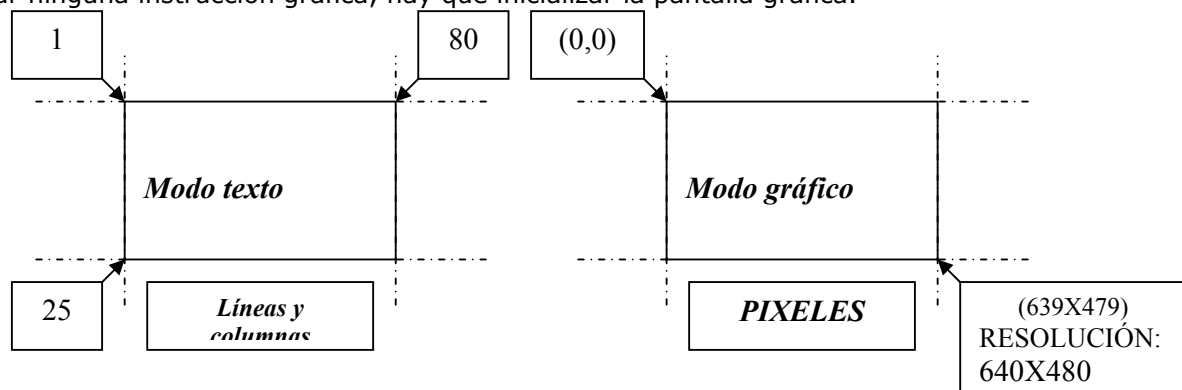
Fprintf(canal, "formato", variables);
Fprintf(impre, "hola");
Fprintf(canal, "el valor de la suma vale %d", suma);

```

Además, dentro del formato, podemos especificar saltos de línea, -con \n-, tabulaciones con \t -, y saltos de página, -\f-. Este último no está disponible en pantalla, con lo que tendremos que contar líneas para la impresora

GRÁFICOS EN C

Lo 1º que hay que hacer es incluir la librería de gráficos. # include <graphics.h>. Antes de usar ninguna instrucción gráfica, hay que inicializar la pantalla gráfica.



Para inicializar la pantalla gráfica, hay que definir dos variables enteras. Por supuesto, las declaramos antes del *main()*:

Por tanto el código, hasta ahora vendría a ser:


```
#include <graphics.h>
main()
{
    int g_driver;
    int g_mode;
    g_driver=DETECT;
    g_mode=DETECT;
    initgraph (&g_drive, &g_mode, "...\\bgi");
    ...
}
```

Esto inicializa
la pantalla
gráfica.

1. INSTRUCCIONES GRÁFICAS

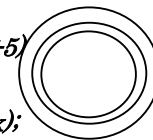
A continuación vamos a ver las instrucciones para dibujar diferentes elementos gráficos:

CÍRCULOS: *circle (x,y, radio)*

Por ejemplo, para hacer una serie de círculos concéntricos

Centro del
círculo

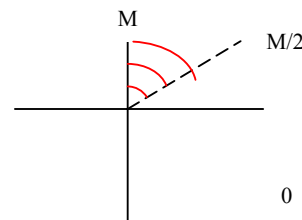
```
For (x=5; x<=200; x=x+5)
{
    circle(320,240,x);
}
```



ARCOS: *arc (x, y, principio, fin, radio)*

Centro del
círculo

Ángulos,
Indicados en
grados

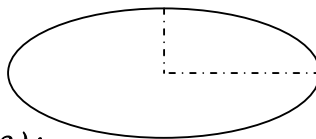


ARCOS CON LÍNEAS EN EL CENTRO: *pieslice (x, y, principio, fin, radio);*

ELIPSES:

ellipse (x, y, principio, fin, radiox, radioy);

Si no se indica, se asume que es una elipse completa. Si indicamos principio y fin nos hará arcos de elipses

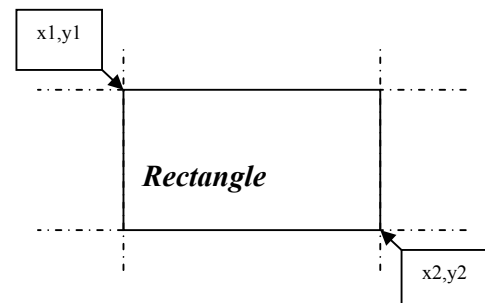


LÍNEAS: *line (x1, y1, x2, y2);*

Principio de
línea

Final de
línea

Elipse (320,240, , ,100,40).
Si indicamos principio y fin nos
hará arcos de elipse.



RECTÁNGULOS: *rectangle(x1,y1, x2,y2);*

Principio de
línea

Final de
línea

Si ponemos *rectangle(x,y,x+10,y+10);* conseguimos un cuadrado.

Para conseguir introducir color en las aplicaciones, podemos utilizar las siguientes instrucciones

COLOR DE FONDO: *setbkcolor(valor);*

el valor rondará entre 0 y 15. Hemos de marcarlo al principio, de lo contrario nos borrará el dibujo hecho con anterioridad. Si a un número de color le sumamos 128, conseguimos que el color parpadee. $7 + 128 = 135 \rightarrow$ verde parpadeante.

COLOR DE BORDE DE FIGURAS: *setcolor(valor);*

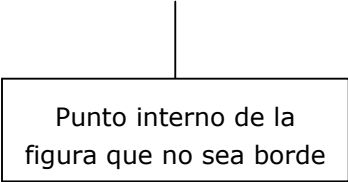
el valor rondará entre 0 y 15. El color del borde ha de ser igual al color del relleno, sino se escapa y pinta toda la pantalla. Por defecto sale color de fondo negro y color de borde blanco.

RELLENO DE FIGURAS CON COLOR:

la figura ha rellenar ha de ser cerrada, además, hemos de elegir un punto interior de la figura; de lo contrario se nos escapará todo el color, con lo que pintaremos toda la pantalla. El relleno se compone de 2 funciones:

```
setfillstyle(SOLID_FILL, color);
floodfill(x,y, color);
```

```
/*establece el tipo de relleno.*/
/* rellena en función del color. El color ha de ser el mismo, que
además ha de coincidir con el del setcolor.
```



Punto interno de la
figura que no sea borde

BORRAR PANTALLA GRÁFICA: *cleardevice();*

Hay una instrucción que no es específica de los gráficos, pero que se utiliza mucho en aplicaciones de este tipo.

kbhit(); Detecta si se ha pulsado una tecla (cualquier tecla) o no. Tiene 2 valores, True= sí pulsado; False= no pulsado. Por ejemplo:

```
while (!kbhit()) /*mientras no se pulse una tecla...*/
```

Pero si por ejemplo, queremos que un proceso se repita mientras no se pulse una tecla en concreto lo que haremos será...

```
Char c; /*definimos c como variable tipo texto*/
```

```
C='';
```

```
While(c!=13) c = getch(); /* mientras que no se pulse el carácter ascii 13, esto es, mientras no se pulse el
return*/
```

Las Bibliotecas del C

Funciones y macros

Fichero de cabecera	Propósito	Notas
Conio.h	E/S por consola.	Cada una de las funciones representa una rutina de servicio de la ROM BIOS. Todas estas funciones se pueden implementar con las funciones <code>int86()</code> y <code>int86x()</code> del fichero de cabecera <code><dos.h></code> .
Ctype.h	Funciones de caracteres.	
dos.h	Sistema operativo DOS.	
graphics.h	Gráficos.	
math.h	Definiciones utilizadas por la biblioteca matemática.	Las funciones matemáticas toman argumentos de tipo <code>double</code> y devuelven valores de tipo <code>double</code> .
stdio.h	Soporta la E/S de ficheros.	
Stdlib.h	Otras declaraciones.	
String.h	Soporta funciones de cadena.	
time.h	Soporta las funciones de tiempo del sistema.	

CONIO.H			
cgets	<code>char *cgets (char *cad);</code>	Lee una cadena de la consola.	<code>cad [0]</code> debe contener la longitud máxima de la cadena a ser leída. A la vuelta, <code>cad[1]</code> contiene el número de caracteres leídos realmente. La cadena empieza en <code>cad[2]</code> . La función devuelve <code>&cad[2]</code> .
clreol	<code>void clreol (void);</code>	Borra hasta el final de la línea en ventana de texto.	
clrscr	<code>void clrscr (void);</code>	Borra ventana de texto.	
cprintf	<code>int cprintf (const char *formato [, argumento, ...]);</code>	Escribe salida formateada en la ventana de texto de la pantalla.	Devuelve el número de bytes escritos.
cputs	<code>int cputs (const char *cad);</code>	Escribe una cadena en la ventana de texto de la pantalla.	Devuelve el último carácter escrito.
cscanf	<code>int cscanf (char *formato [, dirección, ...]);</code>	Lee entrada formateada de consola.	Devuelve el número de campos procesados con éxito. Si una función intenta leer en final de fichero, el valor devuelto es EOF.
delline	<code>void delline (void);</code>	Borra línea en ventana de texto.	
getch	<code>int getch (void);</code>	Lee carácter de consola sin eco en pantalla.	Devuelve el carácter leído. Los caracteres están disponibles inmediatamente (no hay buffer de líneas completas).
getche	<code>int getche (void);</code>	Lee carácter de consola con eco	Como <code>getch()</code> .
getpass	<code>char *getpass (const char *prompt);</code>	Lee un password.	El valor devuelto es un puntero a una cadena estática que es sobrescrita en cada llamada.

gettext	int gettext (int izq, int ar, int der, int ab, void *destino);	Copia text de pantalla en modo texto a memoria.	Las coordenadas son absolutas. La esquina superior izquierda es (1,1). Devuelve un valor distinto de cero si tiene éxito.
gettextinfo	void gettextinfo (struct text_info *r);	Obtiene información de vídeo en modo texto.	El resultado es devuelto en r.
gotoxy	void gotoxy (int x, int y);	Posiciona el cursor en ventana de texto.	
insline	void insline (void);	Inserta línea en blanco en ventana de texto en la posición actual del cursor.	Las líneas por debajo de la posición del cursor son subidas una línea hacia arriba y la última líneas se pierden.
kbhit	int kbhit (void);	Chequea para ver si se ha pulsado alguna tecla.	Si una tecla está disponible, kbhit() devuelve un entero distinto de cero; si no es así, devuelve 0.
movetext	int movetext (int izq, int ar, int der, int ab, int izqdest, int ardest);	Copia texto en pantalla de un rectángulo a otro (en modo texto).	Las coordenadas son relativas a la esquina superior izquierda de la pantalla (1,1). Devuelve un valor distinto de cero si la operación tuvo éxito.
putch	int putch (int ch);	Escribe un carácter en la ventana de texto sobre en la pantalla.	Devuelve ch, el carácter visualizado. En caso de error, devuelve EOF.
puttext	int puttext (int izq, int ar, int der, int ab, void *fuente);	Copia texto de memoria a la pantalla.	Como movetext().
textbackground	void textbackground (int nuevocolor);	Selecciona nuevo color de fondo de los caracteres en modo texto.	
textcolor	void textcolor (int nuevocolor);	Selecciona nuevo color de texto de los caracteres en modo texto.	
textmode	void textmode (int nuevomodo);	Cambia modo de pantalla (en modo texto).	No sirve para cambiar de modo gráfico a modo texto.
wherex	int wherex (void);	Devuelve posición horizontal del cursor dentro de la ventana de texto corriente.	Devuelve un entero en el rango de 1 a 80.
wherey	int wherey (void);	Devuelve posición vertical del cursor dentro de la ventana de texto corriente.	Devuelve un entero en el rango de 1 a 25.
window	void window (int izq, int ar, int der, int ab);	Define ventana activa en modo texto.	La esquina superior izquierda de la pantalla es (1,1).

GRAPHICS.H

arc	void far arc (int x, int y, int ang_comienzo, int ang_final, int radio);	Dibuja un arco.	(x,y) es el punto central; los angulos en grados.
bar	void far bar (int izq, int ar, int der, int ab);	Dibuja una barra.	
cleardevice	void far cleardevice (void);	Borra la pantalla gráfica.	

bar3d	void far bar3d (int izq, int ar, int der, int ab, int profundidad, int flag_de_encima);	Dibuja un barra en 3-D.	Si flag_de_encima es 0 no se dibuja la cara superior de la barra.
circle	void far circle (int x, int y, int radio);	Dibuja un círculo en (x,y) con el radio dado.	
clearviewport	void far clearviewport (void);	Borra el viewport actual.	
closegraph	void far closegraph (void);	Cierra el sistema gráfico.	
detectgraph	void far detectgraph (int far *graphdriver, int far *graphmode);	Determina el controlador y el modo gráfico a usar chequeando el hardware.	
drawpoly	void far drawpoly (int número de puntos, int far *puntos_por_poligono);	Dibuja un polígono.	*puntos_por_poligono apunta a un numero_de_puntos pares de valores. Cada par de los valores de x e y par un punto del polígono.
ellipse	void far ellipse (int x, int y, int ang_comienzo, int ang_final, int radiox, int radioy);	Dibuja un arco elíptico.	Como arc().
fillellipse	void far fillellipse (int x, int y, int radiox, int radioy);	Dibuja y rellena una ellipse	Usa (x,y) como el punto central y rellena el arco usando el patrón de relleno actual.
fillpoly	void far fillpoly (int numpoints, int far *polypoints[]);	Dibuja y rellena un polígono.	Como drawpoly().
floodfill	void far floodfill (int x, int y, int color_borde);	Rellena una región definida.	(x,y) es un punto que reside dentro de la región a rellenar.
getbkcolor	int far getbkcolor (void);	Devuelve el color de fondo actual.	
getcolor	int far getcolor (void);	Devuelve el color de dibujar actual.	
getmaxx	int far getmaxx (void);	Devuelve la coordenada x máxima de pantalla.	
getmaxy	int far getmaxy (void);	Devuelve la coordenada y máxima de pantalla.	
getviewsettings	void far getviewsettings (struct viewporttype far *viewport);	Obtiene información acerca del viewport actual.	
getx	int far getx (void);	Devuelve la coordenada x de la posición actual.	El valor es relativo al viewport.
gety	int far gety (void);	Devuelve la coordenada y de la posición actual.	El valor es relativo al viewport.
line	void far line (int x1, int y1, int x2, int y2);	Dibuja un línea entre dos puntos	Dibuja una línea desde (x1, y1) hasta (x2, y2)
moveto	void far moveto (int x, int y);	Cambia la posición actual a (x,y).	
outtext	void far outtext (char far *cadena_de_texto);	Visualiza una cadena en el viewport (modo gráfico).	
outtextxy	void far outtextxy (int x, int y, char far *cadena_de_text);	Visualiza una cadena en el lugar especificado (modo gráfico).	

initgraph	void far initgraph (int far *controlador_gráfico, int far *modo_gráfico, char far *path_para_controlador);	Inicializa el sistema gráfico.	El parámetro path_para_controlador usa la sintaxis: “..\\bgi\\drivers” donde: -bgi\\drivers es el nombre de directorio donde buscar los controladores -el parámetro está encerrado entre comillas -el path para los ficheros de controladores gráficos incluyen dos barras invertidas.
pieslice	void far pieslice (int x, int y, int ang_comienzo, int ang_final, int radio);	Dibuja y rellena un sector de círculo	
putpixel	void far putpixel (int x, int y, int color);	Escribe un pixel en el punto especificado.	
rectangle	void far rectangle (int izq, int ar, int der, int ab);	Dibuja un rectángulo (modo gráfico).	Usa el estilo, color y grosor de línea actual.
sector	void far sector (int x, int y, int ang_comienzo, int ang_final, int radiox, int radioy);	Dibuja y rellena un sector elíptico.	x e y definen el punto central. El sector es dibujado con el color activo y es rellenado con el color y patrón de relleno actual.
setbkcolor	void far setbkcolor (int color);	Pone el color de fondo actual usando la paleta.	
setcolor	void far setcolor (int color);	Pone el color actual para dibujar.	
setfillpattern	void far setfillpattern (char far *patron_usuario, int color);	Selecciona un patrón de relleno definido por el usuario.	El parámetro patron_usuario apunta a un área de 8 bytes donde se encuentra el patrón de bits
setfillstyle	void far setfillstyle (int patrón, int color);	Pone el patrón y color de relleno.	El parámetro patrón identifica un patrón predefinido. Para poner un patrón de relleno definido por el usuario, llamar a la función setfillpattern().
setlinestyle	void far setlinestyle (int estilo_de_linea, unsigned patron_usuario, int grosor);	Pone el estilo de línea, anchura y patrón actual	Pone el estilo y grosor para el dibujo de líneas en funciones gráficas.
setpalette	void far setpalette (int num_de_color, int color);	Cambia un color de la paleta.	
settextstyle	void far settextstyle (int estilo, int dirección, int tamaño de caracter);	Pone las características actuales del texto.	
setviewport	void far setviewport (int izq, int ar, int der, int ab, int clip);	Pone el viewport actual para salida gráfica.	
settextjustify	void far settextjustify (int horiz, int vert);	Pone justificación de texto para modo gráfico.	Los valores para horiz y vert son los siguientes: horiz 0 LEFT_TEXT vert 0 BOTTOM_TEXT 1 CENTER_TEXT 1 CENTER_TEXT 2 RIGHT_TEXT 2 TOP_TEXT
CTYPE.H			
toupper	int toupper (int ch);	Devuelve ch en mayúsculas.	
Tolower	int tolower (int ch);	Devuelve ch en minúsculas.	

Is*	-	Macros variadas.	<p>Macro Verdad (<code>__ltr</code>) si c es...</p> <p><code>isalnum</code>carácter Una letra o dígito.</p> <p><code>isalpha</code>carácter Una letra.</p> <p><code>isdigit</code>carácter Un dígito.</p> <p><code>isctrl</code>carácter Un carácter de borrado o de control ordinario.</p> <p><code>isascii</code>carácter Un carácter ascii válido.</p> <p><code>isprint</code>carácter Un carácter imprimible.</p> <p><code>isgraph</code>carácter Un carácter imprimible, excepto el carácter espacio.</p> <p><code>islower</code>carácter Una letra minúscula.</p> <p><code>isupper</code>carácter Una letra mayúscula.</p> <p><code>ispunct</code>carácter Un carácter de puntuación.</p> <p><code>isspace</code>carácter Un espacio, tab, retorno de carro, nueva línea, tab vertical, o alimentación de línea.</p> <p><code>isxdigit</code>carácter Un dígito hexadecimal.</p> <p><code>_toupper</code>carácter Convierte c en el rango [a-z] a caracteres [A-Z].</p> <p><code>_tolower</code>carácter Convierte c en el rango [A-Z] a caracteres [a-z].</p> <p><code>ascii</code>carácter Convierte c mayor de 127 al rango 0-127 poniendo los bits menos significat. A 0.</p>
------------	---	------------------	-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

MATH.H

abs	<code>int abs (int x);</code>	Macro que devuelve el valor absoluto de un entero.	
Atof	<code>double atof (const char *s);</code>	Convierte cadena a punto flotante.	Devuelve el valor contenido en s convertido a tipo double, o 0 si s no puede ser convertido.
Cabs	<code>double cabs (struct complex z);</code>	Valor absoluto de un número complejo.	Devuelve el valor absoluto de z como un double.
Ceil	<code>double ceil (double x);</code>	Redondea por arriba.	Devuelve el menor entero mayor o igual que x.
Exp	<code>double exp (double x);</code>	Calcula e elevando a la x-ésima potencia.	
Fabs	<code>double fabs (double x);</code>	Valor absoluto de valor en punto flotante.	
Floor	<code>double floor (double x);</code>	Redondea por abajo.	Devuelve el mayor entero que no es mayor que x.
Fmod	<code>double fmod (double x, double y);</code>	Calcula x módulo y.	
Labs	<code>long int labs (long int x);</code>	Calcula el valor absoluto de un long.	
Ldexp	<code>double ldexp (double x, int exp);</code>	Calcula el producto entre x y 2 elevado a exp.	
Modf	<code>double modf (double x, double *parte_entera);</code>	Descompone en parte entera y parte fraccionaria.	La función <code>modf()</code> descompone x en sus partes entera y fraccionaria. Devuelve la parte fraccionaria y sitúa la parte entera en la variable apuntada por <code>parte_entera</code> .

Pow	double pow (double base, double exponente);	Función potencia, x elevado a y);	Devuelve base elevado a exponente. Se produce un error de dominio si base es 0 y exponente es menor o igual que 0. También puede ocurrir si base es negativo y exponente no es entero. Un desbordamiento produce un error de rango.
Pow10	double pow10 (int p);	Función potencia, 10 a la p.	Devuelve 10 elevado a p.
Sqrt	double sqrt (double x);	Calcula la raíz cuadrada.	Devuelve la raíz cuadrada de x. Si se llama con un número negativo se produce un error de dominio.
STDIO.H			
fclose	int fclose (FILE *flujo);	Cierra un flujo.	Devuelve 0 si tiene éxito; devuelve EOF si se detecta algún error. Un error puede ocurrir cuando se intenta cerrar un fichero que ya ha sido cerrado.
Fcloseall	int fcloseall (void);	Cierra todos los flujos abiertos.	Devuelve el número total de flujos cerrados, o EOF si fue detectado algún error.
Feof	int feof (FILE *flujo);	Macro que devuelve un valor distinto de cero si se ha detectado el fin de fichero en un flujo.	Una vez alcanzado el final del fichero, las operaciones posteriores de lectura devuelve EOF hasta que se cambie la posición del puntero del fichero con funciones como rewind() y fseek(). La función feof() es particularmente útil cuando se trabaja con ficheros binarios porque la marca de fin de fichero es también un entero binario válido.
Ferror	int ferror (FILE *flujo);	Macro que devuelve un valor distinto de cero si ha ocurrido algún error	Los indicadores de error asociados al flujo permanecen activos hasta que se cierra el fichero, se llama a rewind() o a perror().
Fdopen	FILE *fdopen (int descriptor, char *tipo);	Asocia un flujo con un descriptor de fichero.	Devuelve un puntero al nuevo flujo abierto o NULL en el caso de error. Los valores posibles del tipo son los mismos que para los de la función fopen(). Consiste en un string con los siguientes caracteres: r Abre para lectura solamente. w Crea para escritura; <input type="checkbox"/> ltra <input type="checkbox"/> tro <input type="checkbox"/> e fichero existente. A Añade, abre para escritura al final del fichero, o crea fichero para escritura. + Símbolo de suma para permitir operaciones de lectura/escritura. b Abre en modo binario. t Abre en modo texto.
Fflush	int fflush (FILE *flujo);	Vuelca un flujo.	Si el flujo está asociado a un fichero para escritura, una llamada a flush() da lugar a que el contenido del buffer de salida se escriba en el fichero. Si el flujo apunta a un fichero de entrada, entonces el contenido del buffer de entrada se vacía. El fichero permanece abierto. Devuelve EOF si hay error
Fgetc	int fgetc (FILE *flujo);	Obtiene un carácter de un flujo.	La función fgetc() devuelve el siguiente carácter desde el flujo de entrada e incrementa el indicador de posición del fichero. El carácter se lee como unsigned char que se convierte en entero. Si se alcanza el final del fichero, fgetc() devuelve EOF. Recuerda que EOF es un valor entero. Por tanto cuando trabajes con ficheros binarios debes utilizar feof() para comprobar el final del fichero. Si fgetc() encuentra un error, devuelve EOF también. si trabajas con ficheros binarios debes utilizar ferror() para comprobar los errores

fgetchar	int fgetchar (void);	Obtiene un carácter de stdin.	Si tiene éxito, getchar() devuelve el carácter leído, después de convertirlo a int sin extensión de signo. En caso de fin de fichero o error, devuelve EOF.
Fgets	char *fgets (char *s, int n, FILE *flujo);	Obtiene una cadena de caracteres de un flujo.	La función fgets() lee hasta n-1 caracteres desde el flujo y los sitúa en el array apuntado por s. Los caracteres se leen hasta que se recibe un carácter de nueva línea o un EOF o hasta que se llega al límite especificado. <code>□ltra□t</code> de leídos los caracteres, se sitúa en el array un carácter nulo inmediatamente después del último carácter leído. Se guarda un carácter de nueva línea y forma parte de s. Si fgets() tiene éxito devuelve la dirección de s; se devuelve un puntero nulo cuando se produce un error. Ya que se devuelve un puntero nulo cuando se produce un error o cuando se alcanza el final del fichero, utiliza feof() o ferror(9 para identificar lo que ocurre realmente.
Fgetpos	int fgetpos (FILE *flujo, fpos_t *pos);	Obtiene la posición actual del puntero de fichero.	La posición almacenada en *pos puede pasar a la función fsetpos() para poner la posición del puntero de fichero. Devuelve 0 en caso de éxito y un valor distinto de cero en otro caso. Fpos_t es un tipo declarado con typedef en el fichero <code>□ltra.h</code> que indica la posición
flushall	int flushall (void);	Vuelca todos los flujos abiertos.	Vacía los buffers para los flujos de entradas y escribe los buffers en los ficheros para los flujos de salida.
Fopen	FILE *fopen (const char *nombre_fichero, const char *modo_apertura);	Abre un flujo.	Devuelve un puntero al flujo abierto si tiene éxito; en otro caso devuelve NULL.
Fprintf	int fprintf (FILE *flujo, const char *formato[, argumento, ...]);	Envía salida formateada a un flujo.	Esta función es idéntica a la función printf() con la excepción que printf() escribe en la salida estándar (flujo stdout) y la función fprintf() escribe en la salida especificada (indicado en su primer argumento).
Fputc	int fputc (int c, FILE *flujo);	Escribe un carácter en un flujo.	La función fputc() escribe un carácter c en el flujo especificado a partir de la posición actual del fichero y entonces incrementa el indicador de posición del fichero. Aunque ch tradicionalmente se declare de tipo in, es convertido por fputc() a unsigned char. Puesto que todos los argumentos de tipo carácter son pasados a enteros en el momento de la llamada, se seguirán viendo variables de tipo carácter como argumentos. Si se utilizara un entero, simplemente se eliminaría el byte más significativo.
Fputchar	int fputchar (int c);	Escribe un carácter en stdout.	Una llamada a fputchar() es funcionalmente equivalente a fputc
fputs	int fputs (const char *s, FILE *flujo);	Escribe una cadena de caracteres en un flujo.	La función fputs() escribe el contenido de la cadena de caracteres apuntada por s en el flujo especificado. El carácter nulo de terminación no se escribe. La función devuelve 0 cuando tiene éxito, y un valor no nulo bajo condición de error. Si se abre el flujo en modo texto, tienen lugar ciertas transformaciones de caracteres. Esto supone que puede ser que no haya una correspondencia uno a uno de la cadena frente al fichero. Sin embargo, si se abre en modo binario, no se producen transformaciones de caracteres y se establece una correspondencia uno a uno entre la cadena y el fichero.

Fread	int fread (void *buf, int tam, int n, FILE *flujo);	Lee datos de un flujo.	Lee n elementos de tam bytes cada uno. Devuelve el número de elementos (no bytes) realmente leídos. Si se han leído menos caracteres de los pedidos en la llamada, es que se ha producido un error o es que se ha alcanzado el final del fichero. Utiliza feof() o ferrord() para determinar lo que ha tenido lugar. Si el flujo se abre para transformaciones de texto, el flujo de retorno de carro y salto de línea se transforma automáticamente en un carácter de nueva línea.
Freopen	FILE *freopen (const char *nombre_fichero, const char *modo, FILE *flujo);	Asocia un nuevo fichero con un flujo abierto.	El flujo es cerrado. El fichero nombre_fichero es abierto y asociado con el flujo. Devuelve flujo si tiene éxito o NULL si falló.
Fscanf	int fscanf (FILE *flujo, const char *formato[, dirección, ...]);	Ejecuta entrada formateada de un flujo.	Esta función es idéntica a la función scanf() con la excepción que scanf() lee de la entrada estandar (flujo stdin) y la función fscanf() lee de la entrada especificada (flujo indicado en su primer parámetro).
Fseek	int fseek (FILE *flujo, long desplazamiento, int origen);	Posiciona el puntero de fichero de un flujo.	La función fseek() sitúa el indicador de posición del fichero asociado a flujo de acuerdo con los valores de desplazamiento y origen. Su objetivo principal es soportar operaciones de E/S aleatorias; desplazamiento es el número de bytes desde el origen elegido a la posición seleccionada. El origen es 0, 1 o 2 (0 es el principio del fichero, 1 es la posición actual y 2 es el final del fichero). El estandar ANSI fija los siguientes nombres para los orígenes: comienzo = SEEK_SET, actual = SEEK_CUR, final = SEEK_END. Si se devuelve 0, se supone que fseek() se ha ejecutado correctamente. Un valor distinto de 0 indica fallo.
Fsetpos	int fsetpos (FILE *flujo, const fpos_t *pos);	Posiciona el puntero de fichero de un flujo.	La nueva posición apuntada por pos es el valor obtenido por una llamada previa a la función fgetpos(). En caso de éxito, devuelve 0. En caso de fallo, devuelve un valor distinto de 0.
Ftell	long ftell (FILE *flujo);	Devuelve la posición actual del puntero de fichero.	Devuelve el valor actual del indicador de posición del fichero para el flujo especificado si tiene éxito o -1L en caso de error.
Fwrite	int fwrite (const void *buff, int tam, int n, FILE *flujo);	Escribe en un flujo.	Escribe n elementos de tam bytes cada uno. Devuelve el número de elementos (no bytes) escritos realmente.
Gets	char *getl (char *string);	Obtiene una cadena de caracteres de stdin.	Lee caracteres de stdin hasta que un carácter de nueva línea (\n) es encontrado. El carácter \n no es colocado en el string. Devuelve un puntero al argumento string.
Getw	int getw (FILE *flujo);	Obtiene un entero de un flujo.	Devuelve el próximo entero en el flujo de entrada, o EOF si ocurre un error o se detecta el fin de fichero. Usa las funciones feof() o ferrord() para verificar.
Printf	int printf (const char *formato [, argumento, ...]);	Escribe con formateo a stdout.	
Puts	int puts (const char *s);	Escribe un string en stdout (y añade un carácter de nueva línea).	Si la escritura tiene éxito, puts() devuelve el último carácter escrito. En otro caso, devuelve EOF.
Putw	int putw (int d, FILE *flujo);	Escribe un entero en un flujo.	Devuelve el entero d. En caso de error, devuelve EOF.
Remove	int remove (const char *nombre_fichero);	Función que borra un fichero.	Borra el fichero especificado por nombre_fichero. Devuelve 0 si el fichero ha sido correctamente borrado y -1 si se ha producido un error.

Rename	int rename (const char *viejo_nombre, const char *nuevo_nombre);	Renombra un fichero.	La función rename() cambia el nombre del fichero especificado por viejo_nombre a nuevo_nombre. El nuevo_nombre no debe estar asociado a ningún otro en el directorio de entrada. La función rename() devuelve 0 si tiene éxito y un valor no nulo si hay algún error
rewind	void rewind (FILE *flujo);	Reposiciona el puntero a fichero al comienzo del flujo.	La función rewind() mueve el indicador de posición del fichero al principio del flujo especificado. También inicializa los indicadores de error y fin de fichero asociados con flujo. No devuelve valor.
Scanf	int scanf (const char *formato [, ...]);	Ejecuta entrada formateada de stdin.	
Sprintf	int sprintf (char *buffer, const char *formato [, argumento, ...]);	Envía salida formateada a un string.	Esta función es igual que la función printf() con la diferencia de que la salida de la función printf() va al flujo stdout y la salida de la función sprintf() va al string buffer. Devuelve el número de bytes escritos. En caso de error, sprintf() devuelve EOF.
Sscanf	int sscanf (const char *buffer, const char *formato [, dirección, ...]);	Ejecuta entrada formateada de string.	Esta función es igual que la función scanf() con la diferencia de que la entrada de la función scanf() se toma del flujo stdin y la entrada de la función sscanf() del string buffer. Devuelve el número de bytes escritos. En caso de error, sscanf() devuelve EOF. Devuelve el número de campos leídos, explorados, convertidos y almacenados. Si sscanf quiere leer más allá del final de buffer, devuelve EOF
ungetc	int ungetc (int c, FILE *flujo);	Devuelve un carácter al flujo de entrada.	La próxima llamada a getc (u otras funciones de entrada de flujos) para flujo devolverá c. La función ungetc() devuelve el carácter c si tiene éxito. Devuelve EOF si la operación falla.

STDLIB.H

abort	void abort (void);	Termina anormalmente un programa.	
Abs	int abs (int x);	Devuelve el valor absoluto de un entero.	
Atof	double atof (const char *s);	Convierte una cadena a un punto flotante.	La función atof() convierte la cadena apuntada por s a un tipo double. La cadena debe contener un número válido. Si no es así, se devuelve el valor 0.
Atoi	int atoi (const char *s);	Convierte una cadena en un entero.	La función atoi() convierte la cadena apuntada por s a un valor int. La cadena debe contener un número válido. Si no es este el caso, se devuelve el valor 0.
Atol	long atol (const char *s);	Convierte una cadena a un long.	Convierte la cadena apuntada por s a un valor long int. La cadena debe contener un número entero de tipo long válido. Si no es así, se devuelve el valor 0.
Div	div_t div (int numer, int denom);	Divide dos enteros.	Devuelve el cociente y el resto como un tipo div_t. <pre> typedef struct { long int quot; /*cociente*/ long int rem; /* resto */ } div_t;</pre>

ecvt y fcvt	char *.cvt (double valor, int ndig, int *dec, int *sign);	Convierte número en coma flotante a cadena.	Para ecvt(), ndig es el número de dígitos a almacenar, mientras que fcvt() es el número de dígitos a almacenar. El valor devuelto apunta a un área estática
Exit	void exit (int estado);	Termina el programa.	Antes de terminar, la salida buffereada es volcada, los ficheros son cerrados y las funciones exit() son llamadas.
_exit	void _exit (int estado);	Termina programa.	
ltoa	char *ltoa (int valor, char *cad, int radix);	Convierte un entero a una cadena.	La función ltoa() convierte el entero valor a su cadena correspondiente y sitúa el resultado en la cadena apuntada por cad. La base de la cadena de salida se determina por radix, que se encuentra normalmente en el rango de 2 a 16. La función ltoa() devuelve un puntero a cad. Lo mismo se puede hacer con sprintf().
Labs	long int labs (long int x);	Calcula el valor absoluto de un long.	
Ldiv	ldiv_t ldiv (long int numer, long int denom);	Divide dos longs, devuelve el cociente y el resto.	La función ldiv() divide dos longs y devuelve el cociente y el resto como tipo ldiv_t. <pre> Typedef struct { long int quot; /* cociente */ long int rem; /* resto */ } ldiv_t; </pre>
_lrotl _lrotr	unsigned long _lrotl.. (unsigned long val, int cont);	Rota un valor long a la izquierda (_lrotl) o a la derecha (_lrotr).	Las dos funciones devuelve el valor de val rotado cont bits.
Ltoa	char *ltoa (long valor, char *cadena, int radix);	Convierte un long a una cadena.	Para una <code>%ommando</code> <code>%n</code> <code>%n</code> <code>%ommand</code> , usa radix=10. Para hexadecimal, usa radix=16. Devuelve un puntero al argumento cadena.
Max min	max (a,b) min (a,b)	Macros que generan código en línea para encontrar el valor máximo y mínimo de dos enteros.	
Rand	int rand (void);	Generador de números aleatorios.	Devuelve números aleatorios entre 0 y RAND_MAX.
Random	int random (int num);	Macro que devuelve un entero.	Devuelve un entero entre 0 y (num-1).
Randomize	void <code>%ommando</code> (void);	Macro que inicializa el generador de números aleatorios.	Inicializa el generador de número aleatorios con un valor aleatorio. Esta función usa la función time(), así que debemos incluir time.h cuando usemos esta rutina.
Qsort	void qsort (void base, size_t num, size_t tam, int (compara) (const void *, const void *));	Ordena usando el algoritmo quicksort.	La función qsort() ordena el array apuntado por base. El número de elementos en el array se especifica mediante num, y el tamaño en bytes de cada elemento está descrito por tam. La función compara se utiliza para comparar un elemento del array con la clave. El array es ordenado en orden ascendente

strtod	double strtod (const char *inic, char **fin);	Convierte cadena a double.	La función strtod() convierte la cadena de un número almacenado en la cadena apuntada por inic a un valor double y devuelve el resultado.. Primero, se elimina cualquier blanco de la cadena apuntada por inic. Cada carácter que no pueda ser parte de un número en coma flotante dará lugar a que el proceso se detenga. Esto incluye el espacio en blanco, signos de puntuación distintos del punto, y caracteres que no sean E o e. Finalmente se deja apuntando al resto, si lo hay, de la cadena original. Si se produce un error devuelve HUGH_VAL para overflow o HUGN_VAL para underflow. Si no se produce devuelve 0.
Strtol	long strtol (const char *inic, char **final, int radix);	Convierte cadena a long usando la base fijada.	La función strtol() convierte la cadena de caracteres de un número (almacenada en la cadena apuntada por inic) en un número de tipo long int y devuelve el resultado. La base del número está determinada por radix. Si radix es 0. Si radix es distinto de 0, debe estar en el rango de 2 a 36. La función strtol() trabaja de la siguiente forma: Primero, elimina cualquier espacio en blanco de la cadena apuntada por inic. A continuación, se lee cada uno de los caracteres que constituyen el número. Cualquier carácter que no pueda formar parte de un número de tipo long int finaliza el proceso. Finalmente, se deja apuntando al resto, si lo hay, de la cadena original. Si se produce un error, strtol() devuelve LONG_MAX en caso de desbordamiento por arriba o LONG_MIN en caso de desbordamiento por abajo. Si no se produce error, se devuelve 0.
Strtoul	unsigned long strtoul (const char *inic, char **final, int radix);	Convierte una cadena a un unsigned long con la base fijada.	Como strtol().
System	int system (const char *comando);	Ejecuta un comando DOS.	Ver dos.h.
ultoa	char *ultoa (unsigned long valor, char *cadena, int radix);	Convierte un unsigned long a una cadena.	Devuelve un puntero a la cadena. No devuelve error.

STRING.H

strcpy	char *strcpy (char *destino, const char *fuente);	Copia un string en otro.	Es igual que strcpy(), excepto que strcpy devuelve destino + strlen(fuente).
Strcat _fstrcat	char *strcat (char *destino, const char *fuente);	Añade fuente a destino.	Devuelve destino.
Strchr _fstrchr	char *strchr (const char *s, int c); (_fstrchr: todos los punteros far)	Encuentra c en s.	Devuelve un puntero a la primera ocurrencia del carácter c en s; si c no aparece en s, strchr() devuelve NULL.
Strcmp _fstrcmp	int strcmp (const char *s1, const char *s2); (_fstrcmp: todos los punteros far)	Compara un string con otro.	Devuelve: <0 si s1<s2; =0 si s1=s2; >0 si s1>s2. Ejecuta una comparación con signo.
Strcmpi	int strcmpi (const char *s1, const char *s2);	Macro que compara strings.	Esta rutina está implementada como una macro para compatibilidad con otros compiladores. Es igual que strcmp().

Strcpy	char *strcpy (char *destino, const char *fuente);	Copia el string fuente al string destino.	Devuelve destino.
Strcspn _fstrcspn	size_t strcspn (const char *s1, const char *s2);	Explora un string.	Devuelve la longitud del substring inicial apuntado por s1 que está constituido sólo por aquellos caracteres que no están contenidos en s2., strcspn() devuelve el índice del primer carácter en el string apuntado por s1 que está como carácter del string apuntado por s2.
Strdup _fstrdup	char *strdup (const char *s);	Obtiene una copia duplicada de s, o copia s a una nueva localización.	Devuelve un puntero a la copia duplicada de s, o devuelve NULL si no hubo espacio suficiente para la asignación de la copia. El programador es responsable de liberar el espacio asignado por strdup() cuando ya no sea necesario.
Strcmp _fstrcmp	int strcmp (const char *s1, const char *s2); (_fstrcmp: todos los punteros far)	Compara un string con otro ignorando el caso.	Devuelve: s1-s2. Ejecuta comparación con signo.
Strlen _fstrlen	size_t strlen (const char *s); (_fstrlen: todos los punteros far)	Calcula la longitud de un string.	Devuelve el número de caracteres que hay en s, sin contar el terminador nulo.
Strlwr _fstrlwr	char *strlwr (char *s); (_fstrlwr: todos los punteros far)	Convierte s a caracteres en minúsculas.	Devuelve un puntero a s.
Strncat _fstrncat	char *strncat (char *destino, const char *fuente, size_t longmax); (_fstrncat: todos los punteros far)	Añade como máximo longmax caracteres de fuente a destino.	Devuelve destino.
Strncmp _fstrncmp	int strncmp (const char *s1, const char *s2, size_t longmax); (_fstrncmp: todos los punteros far)	Compara como mucho longmax caracteres de un string con otro.	Devuelve: s1-s2. Ejecuta comparación con signo.
Strncmpi	int strncmpi (const char *s1, const char *s2, size_t n);	Compara un trozo de un string con un trozo de otro, sin sensibilidad al caso.	La función strncmpi() ejecuta una comparación con signo entre s1 y s2, para una longitud máxima de n bytes, empezando con el primer carácter de cada string y continuando con los caracteres siguientes hasta encontrar caracteres correspondientes diferentes o hasta que se han examinado n caracteres.
Strncpy _fstrncpy	char *strncpy (char *destino, const char *fuente, size_t longmax); (_fstrncpy: todos los punteros far)	Copia como máximo longmax carácter de fuente a destino.	Si son copiados longmax caracteres, no es añadido el carácter nulo; por la tanto el contenido de destino no es un string terminado en nulo. Devuelve destino.
Strnicmp _fstrnicmp	int strnicmp (const char *s1, const char *s2, size_t longmax); (_fstrnicmp: todos los punteros far)	Compara como máximo n caracteres de un string con otro, ignorando el caso.	Devuelve s1-s2. Ejecuta comparación con signo.
Strpbrk _fstrpbrk	char *strpbrk (const char *s1, const char *s2); (_fstrpbrk: todos los punteros far)	Explora un string.	Devuelve un puntero al primer carácter del string apuntado por s1 que se corresponde con algún carácter en el string apuntado por s2. El carácter nulo de terminación no se incluye. Si no hay correspondencia, se devuelve un puntero nulo.
Strrchr _fstrchr	char *strrchr (const char *s, int c); (_fstrchr: todos los punteros far)	Encuentra la última ocurrencia de c en s.	Devuelve un puntero a la última ocurrencia del carácter c, o NULL si c no aparece en s.

strrev _fstrrev	char *strrev (char *s); (_fstrrev: todos los punteros far)	Invierte todos los caracteres de s(excepto el carácter terminador nulo).	Devuelve un puntero al string invertido.
Strstr _fstrstr	char *strstr (const char *s1, const char *s2); (_fstrstr: todos los punteros far)	Encuentra la primera ocurrencia de un substring en un string.	Devuelve un puntero a la primera ocurrencia en la cadena apuntada por s1 de la cadena apuntada por s2 (excepto el carácter nulo de terminación de s2). Devuelve un puntero nulo si no se encuentra.
Strtok _fstrtok	char *strtok (cahr *s1, const char *s2); (_fstrtok: todos los punteros far)	Explora s1 para encontrar el primer token no contenido en s2.	S2 define caracteres separadores; strtok() interpreta el string s1 como una sere de tokens separados por los caracteres separadores que hay en s2. Si no se encuentra ningún token en s1, strtok() devuelve NULL. Si se encuentra un token, se pone un carácter nulo en s1 siguiendo al token, y strtok() devuelve un puntero al token. En las llamadas siguientes a strtok() con NULL como primer argumento, usa el string previo s1, empezando después del último token encontrado. Notese que la cadena inicial es, por tanto, destruida.
Strupr _fstrupr	char *strupr (char *s); (_fstrupr: todos los punteros far)	Convierte todos los caracteres de s a mayúsculas.	Devuelve un puntero a s.
TIME.H			
clock	clock_t clock (void);	Devuelve el número de pulsos de reloj desde el comienzo del programa.	Devuelve el tiempo de procesador usado desde el comienzo de la ejecución del programa medido en pulsos de reloj. Para transformar este valor en segundos, se divide entre CLK_TCK. Se devuelve el valor -1 si el tiempo no está disponible.
ctime	char *ctime (const time_t *time);	Convierte fecha y hora cadena.	Esta función es equivalente a: asctime (localtime (hora));
difftime	double difftime (time_t hora2, time_t hora1);	Calcula la diferencia entre dos horas.	Devuelve la diferencia, en segundos, entre hora1 y hora2. Es decir hora2-hora1.
localtime	struct tm *localtime (const time *hora);	Convierte fecha y hora a una estructura.	La función localtime() devuelve un puntero a la forma esperada de hora en la estructura tm. La hora se trepresenta en la hora local. El valor hora se obtiene normalmente a través de una llamada a time(). La estructura utilizada por localtime() para mantener la hora separada está situada de forma estática y se reescribe cada vez que se llama a la función Si se desea guardar el contenido de la estructura, es necesario copiarla en otro lugar.
mktime	time_t mktime (struct tm *t);	Convierte hora a formato de calendario.	La función mktime() devuelve la hora de calendario equivalente a la hora separada que se encuentra en la estructura apuntada por hora. Esta función se utiliza principalmente para incializar el sistema. Los elementos tm_wday y tm_yday son activados por la función, de modo que no necesitan ser definidos en el momento de la llamada. Si mktime() no puede representar la información como una hora válida, se devuelve -1.
stime	int stime (time_t *pt);	Pone fecha y hora del sistema.	Devuelve 0.

strftime	<code>size_t strftime (char *cad, size_t maxtam, const char *fmt, const struct tm *tm);</code>	Formatea hora para salida.	La función <code>strftime()</code> sitúa la hora y la fecha (junto con otra información) en la cadena apuntada por <code>cad</code> . La información se sitúa de acuerdo a las órdenes de formato que se encuentran en la cadena apuntada por <code>fmt</code> y en la forma de hora separada de <code>t</code> . Se sitúan un máximo de <code>maxtam</code> caracteres en <code>cad</code> . La función <code>strftime()</code> trabaja de forma algo parecida a <code>sprintf()</code> en el que se reconoce un conjunto de órdenes de formato que comienzan con el signo de porcentaje (%) y sitúa su salida con formato en una cadena. Las órdenes de formato se utilizan para especificar de forma exacta en que se representan diferentes informaciones de hora y fecha en <code>cad</code> . Cualquier otro carácter que se encuentre en la cadena de formato se pone en <code>cad</code> sin modificar. La hora y fecha presentadas están en hora local.
time	<code>time_t time (time_t *hora);</code>	Obtiene la hora actual.	La función <code>time()</code> devuelve la hora actual de calendario del sistema. Si el sistema no tiene hora, devuelve -1. La función <code>time()</code> puede llamarse con un puntero nulo o con un puntero a una variable de tipo <code>time_t</code> .

Constantes, tipos de datos y variables globales**1. CONIO.H****Colores :** Tipo enumerado, Colores/atributos del video CGA estándar

0. BLACK	1. DARKGRAY	2. BLUE	3. LIGHTBLUE
4. GREEN	5. LIGHTGREEN	6. CYAN	7. LIGHTCYAN
8. RED	9. LIGHTRED	10. MAGENTA	11. LIGHTMAGENTA
12. BROWN	13. YELLOW	14. LIGHTGRAY	15. WHITE

BLINK: Constante

Esta constante se le suma a color de fondo para visualizar caracteres parpadeantes en modo texto.

Directvideo: Variable global, Controla la salida de vídeo.: **int directvideo;**

La variable global directvideo controla si la salida en consola de nuestro programa va directamente a la RAM de vídeo (por defecto, directvideo = 1) o va vía llamadas a la ROM BIOS (directvideo = 0).

text_modes: Tipo enumerado. Modos de vídeo estándar.

LASTMODE	BW80	BW40	C80
C40	C4350	MONO	

2. GRAPHICS.H**fill_patterns:**Tipo enumerado, Patrones de relleno para las funciones *getfillsettings()* y *setfillsettings()*.

EMPTY_FILL	Usa color de fondo.
SOLID_FILL	Usa color de relleno sólido.
LINE_FILL	Relleno con ---
LTSLASH_FILL	Relleno con ///
SLASH_FILL	Relleno con líneas gruesas ///.
BKSLASH_FILL	Relleno con líneas gruesas \\\.
LTBKSLASH_FILL	Relleno con \\\.
HATCH_FILL	Sombreado claro.
XHATCH_FILL	Sombreado espeso.
INTERLEAVE_FILL	Líneas entrelazadas.
WIDE_DOT_FILL	Puntos bastante espaciados.
CLOSE_DOT_FILL	Puntos pocos espaciados.
USER_FILL	Definido por el usuario.

font_names: Tipo enumerado, Nombres de tipos de caracteres gráficos.

0. DEFAULT_FONT	1. TRIPLEX_FONT	2. SMALL_FONT	3.
SANS_SERIF_FONT	4. GOTHIC_FONT		

graphics_drivers: Tipo enumerado

CGA	MCGA	EGA	EGA64
EGAMONO	IBM8514	HERCMONO	ATT400
VGA	PC3270	DETECT	(Requiere autodetección)

line_styles: Tipo enumerado, Estilos de línea para las funciones *getlinesettings()* y *setlinestyle()*.

SOLID_LINE	DOTTED_LINE	CENTER_LINE
DASHED_LINE	USERBIT_LINE (User-defined)	

line_widths: Tipo enumerado, Anchuras de línea para las funciones *getlinesettings()* y *setlinesettings()*.

NORM_WIDTH

THICK_WIDTH

text_just: Tipo enumerado, Justificación horizontal y vertical para la función *settextjustify()*.

LEFT_TEXT

CENTER_TEXT

RIGHT_TEXT

BOTTOM_TEXT

TOP_TEXT

***_DIR (Dirección):** Dirección de salida gráfica.

HORIZ_DIR

De izquierda a derecha.

VERT_DIR

De abajo hacia arriba.

3. TIME.H

Estructura: de la hora usada por las funciones *dostounix()*, *gettime()*, *settime()* y *unixtodos()*.

```
struct time
{
    unsigned char ti_min;
    unsigned char ti_hour;
    unsigned char ti_hund;
    unsigned char ti_sec;
};
```

clock_t: Tipo, Este tipo de datos devuelto por la función *clock()* almacena un tiempo transcurrido medido en pulsos de reloj.

La constante CLK_TCK define el número de pulsos de reloj por segundo.

time_t: Tipo, Este tipo de variable define el valor usado por las funciones de tiempo.

En TC, también está declarado en *sys/types.h*. El fichero *sys/types.h* únicamente contiene la declaración de este tipo.

TM: Estructura, Describe una estructura que contiene una fecha y hora separada en sus componentes.

```
struct tm
{
    int tm_sec;           /* segundos, 0-59 */
    int tm_min;           /* minutos, 0-59 */
    int tm_hour;          /* horas, 0-23 */
    int tm_mday;          /* día del mes, 1-31 */
    int tm_mon;           /* mes, 0-11, Enero=0 */
    int tm_year;          /* año, año actual-1900 */
    int tm_wday;          /* día de la semana, 0-6, Domingo=0 */
    int tm_yday;          /* día del año, 0-365, 1 de Enero=0 */
    int tm_isdst;         /* indicador de horario de verano */
};
```

INDICE**REGLAS DE ESCRITURA Y ESTRUCTURA DE UN PROGRAMA C**

1. ANATOMÍA DE UN PROGRAMA C	1
2. ENCABEZAMIENTO	1
3. COMENTARIOS	2

VARIABLES Y CONSTANTES

1. DEFINICION DE VARIABLES	2
2. INICIALIZACION DE VARIABLES	3
3. TIPOS DE VARIABLES	3
4. VARIABLES DE TIPO CARÁCTER	4
5. DEFINICION DE NUEVOS TIPOS (typedef)	5
6. CONSTANTES SIMBOLICAS	5

TIPOS DE OPERADORES Y EXPRESIONES

1. INTRODUCCIÓN	6
2. OPERADORES ARITMÉTICOS	6
3. OPERADORES RELACIONALES	6
4. OPERADORES LÓGICOS	7
5. OPERADORES DE INCREMENTO Y DECREMENTO	7
6. OPERADORES DE ASIGNACIÓN	8

FUNCIONES DE E/S

1. FUNCIONES PRINTF() Y SCANF()	9
2. DIFERENCIAS PRINTF/SCANF	9
3. FUNCIONES PUTCHAR() Y GETCHAR()	10
4. FUNCIONES PUTS() Y GETS()	10
6. FUNCIONES PUTC(), GETCH(), GETCHE() Y UNGETCH()	11
7. FUNCIÓN KBHIT()	11

PROPOSICIONES PARA EL CONTROL DE FLUJO DE PROGRAMA

1. INTRODUCCIÓN	12
2. PROPOSICION IF - ELSE	12
3. PROPOSICION SWITCH	13
4. LA ITERACION WHILE	14
5. LA ITERACION DO - WHILE	15
6. ITERACION FOR	15
7. LA SENTENCIA BREAK	16
8. LA SENTENCIA CONTINUE	17
9. LA FUNCION EXIT()	18

FUNCIONES

1. INTRODUCCIÓN	17
2. DECLARACION DE FUNCIONES	18
3. DEFINICION DE LAS FUNCIONES	19
4. FUNCIONES QUE NO RETORNAN VALOR NI RECIBEN PARÁMETROS	20
5. FUNCIONES QUE RETORNAN VALOR	21
6. AMBITO DE LAS VARIABLES	25

CREACIÓN DE BIBLIOTECAS DEL USUARIO 26**ESTRUCTURAS DE AGRUPAMIENTO DE VARIABLES**

1. CONJUNTO ORDENADO DE VARIABLES (ARRAYS)	27
2. CONJUNTO ORDENADO DE CARACTERES (STRINGS)	28
3. ARRAYS Y STRINGS COMO ARGUMENTOS DE FUNCIONES	29
4. ARRAYS MULTIDIMENSIONALES.	31
5. ESTRUCTURAS	31

6. ARRAYS DE ESTRUCTURAS	33
APUNTADORES Y ARREGLOS	
1. INTRODUCCION A LOS PUNTEROS	34
2. PUNTEROS Y ARRAYS	35
3. PUNTEROS A STRINGS	36
4. ARRAYS DE PUNTEROS	38
FUNCIONES DE MANEJO DE STRINGS	
1. INTRODUCCIÓN	38
2. CONVERSION ENTRE STRING Y VARIABLES NUMÉRICAS	38
3. DETERMINACION DE LA LONGITUD DE UN STRING	40
4. COPIA Y DUPLICACION DE STRINGS	40
5. CONCATENACION DE STRINGS	42
6. COMPARACION DE STRINGS	42
7. BUSQUEDA DENTRO DE UN STRING	43
8. FUNCIONES DE MODIFICACION DE STRING	43
FICHEROS EN C.	
1. ABRIR FICHEROS:	44
2. CERRAR FICHEROS:	44
3. ESCRIBIR EN UN FICHERO:	45
4. LEER EN UN FICHERO:	45
5. FSEEK:	45
6. FGETPOS:	46
7. UNLINK:	46
8. RENAME	46
9. USO DE ESTRUCTURAS DE DATOS Y GUARDAR EN ARCHIVOS	46
IMPRESORAS EN C.	47
GRÁFICOS EN C	
1. INSTRUCCIONES GRÁFICAS	48
LAS BIBLIOTECAS DEL C	
FUNCIONES Y MACROS	50
CONIO.H	50
GRAPHICS.H	51
CTYPE.H	53
MATH.H	54
STDIO.H	54
STDLIB.H	58
STRING.H	60
TIME.H	63
CONSTANTES, TIPOS DE DATOS Y VARIABLES GLOBALES	
1. CONIO.H	64
2. GRAPHICS.H	64
3. TIME.H	66