III

EL LENGUAJE DE PROGRAMACIÓN C

El lenguaje C reúne características de programación intermedia entre los lenguajes ensambladores y los lenguajes de alto nivel; con gran poderío basado en sus operaciones a nivel de bits (propias de ensambladores) y la mayoría de los elementos de la programación estructurada de los lenguajes de alto nivel, por lo que resulta ser el lenguaje preferido para el desarrollo de software de sistemas y aplicaciones profesionales de la programación de computadoras.

HISTORIA

En 1970 Ken Thompson de los laboratorios Bell se había propuesto desarrollar un compilador para el lenguaje Fortran que corría en la primera versión del sistema operativo UNIX tomando como referencia el lenguaje BCPL; el resultado fue el lenguaje B (orientado a palabras) que resulto adecuado para la programación de software de sistemas. Este lenguaje tuvo la desventaja de producir programas relativamente lentos.

En 1971 Dennis Ritchie, con base en el lenguaje B desarrolló NB que luego cambio su nombre por C; en un principio sirvió para mejorar el sistema UNIX por lo que se le considera su lenguaje nativo. Su diseño incluye una sintaxis simplificada, la aritmética de direcciones de memoria (permite al programador manipular bits, bytes y direcciones de memoria) y el concepto de apuntador; además, al ser diseñado para mejorar el software de sistemas, se busco que generase códigos eficientes y uno portabilidad total, es decir el que pudiese correr en cualquier máquina. Logrados los objetivos anteriores, C se convirtió en el lenguaje preferido de los programadores profesionales.

En 1980 Bjarne Stroustrup de los laboratorios Bell de Murray Hill, New Jersey, inspirado en el lenguaje Simula67 adiciono las características de la programación orientada a objetos (incluyendo la ventaja de una biblioteca de funciones orientada a objetos) y lo denomino C con clases. Para 1983 dicha denominación cambio a la de C++. Con este nuevo enfoque surge la nueva metodología que aumenta las posibilidades de la programación bajo nuevos conceptos.

1

ELEMENTOS DEL LENGUAJE C

Un programa en C se conforma como una colección de procedimientos (a menudo llamadas funciones, aunque no tengan valores de retorno). Estos procedimientos contienen declaraciones, sentencias, expresiones y otros elementos que en conjunto indican a la computadora que realice cierta acción .

1.1. IDENTIFICADORES ESTANDAR

Los identificadores son nombres dados a constantes, variables, tipos, funciones y etiquetas de un programa.

Un identificador es una secuencia de letras (mayúsculas y/o minúsculas), dígitos (0,1,...,9) y el caracter especial de subrayado (_). El primer caracter de un identificador debe de ser un caracter letra o el carácter de subrayado.

Las letras pueden ser mayúsculas o minúsculas y se consideran como caracteres diferentes.

Por ejemplo:

Suma
Calculo_numeros_primos
ab123
_ordenar
i

1.2 PALABRAS RESERVADAS DEL LENGUAJE C (ANSI-C)

Las palabras reservadas son identificadores predefinidos que tienen un significado especial para el compilador C. Un identificador definido por el usuario, no puede tener el mismo nombre que una palabra reservada.

auto	**continue**	**else**	**for**	**long sizeof**
typedef	**wile**	**break**	**default**	**num**
goto	**register**	**static**	**union**	**main**
case	**do**	**extern**	**if**	**return**
struct	**unsigned**	**char**	**double**	**float**
int	**short**	**switch**	**void**	**signed**

Algunas versiones de compiladores pueden tener palabras adicionales, asm, ada, fortran, pascal, etc. Cabe hacer mención que el lenguaje que analizaremos es el ANSI C, y éste debe de compilarse en cualquier compilador y cualquier plataforma, que soporte el ANSI C (LINUX, UNIX, MS-DOS, etc.).

1.3 ESTRUCTURA DE UN PROGRAMA

Como en todos los lenguajes siempre es bueno comenzar con un programa, simple y sencillo.

```
/* Un primer programa en C*/
#include  <stdio.h>
void main(void)
{
  printf("Hola  Puebla");
  return;
}
```

Explicación:

La primera línea dice que se debe de incluir un archivo de cabecera, este archivo de cabecera contiene todas las funciones de entrada y salida (por ejemplo el printf, es una función que imprime datos y/o letreros en pantalla), la segunda línea de código es el inicio que todo programa en C debe de tener (función main), la tercera línea imprime en pantalla el letrero hola Puebla, y después finaliza el programa, en las dos líneas finales se da por concluido el programa en C, la palabra return se utiliza para regresar un valor, en este caso es el termino de la función main y no se regresa ningún valor.

Un programa en C está formado por una secuencia de caracteres que incluyen:

Letras Minúsculas	: a, b, ..., z
Letras Mayúsculas	: A, B, ..., Z
Digitos	: 0, 1, ..., 9
Caracteres Especiales	: “, !, #, ”, $, %, &, /, (, etc.

A partir de estos podemos generar cualquier programa en C, hay que tener cuidado por que algunos caracteres, tiene significados distintos, es decir, depende del contexto donde se utilicen.

Un programa fuente C es una colección de cualquier número de directrices (inclusión de archivos), declaraciones, definiciones, expresiones, sentencias y funciones.

Todo programa en C debe contener una función nombrada main(), donde el programa comienza a ejecutarse. Las llaves ({}) que incluyen el cuerpo de esta función principal, definen el principio y el final del programa.

Un programa C, además de la función principal main(), consta generalmente de otras funciones que definen rutinas con una función específica en el programa.

- Estructura general de un programa en C:
- Directrices para el preprocesador
- Definición de constantes
- Definición de variables globales
- Declaración de funciones (función prototipo o declaración forward)
- Función main

Los comentarios en C son cadenas arbitrarias de símbolos colocados entre los delimitadores /* y */ .

Ejemplos:

```
/* Comentarios */
/* Este es un comentario *
  muy largo ya que ocupa *    mas de un renglón   */
```

Ejemplo de la estructura general de un programa en C:

```
#include <stdio.h>          /* Directrices del preprocesador */
#include <stdlib.h>
#define SU 100              /* Definición de constantes */
int x,y;                    /* Variables globales */
main()                      /* Programa principal */
{                           /* Inicia el programa principal */
 float real;                /* Variables locales */
                            /* Acciones */
 printf (“\n Dame dos numero entero: ”);
 scanf(“%d%d”, &x,&y) ;
 real=x/y;
 printf(“\n Resultado: %f” ,real);
}                           /* Fin del programa principal */
```

1.4. TIPOS DE DATOS ESTÁNDAR DEL LENGUAJE C

Los tipos básicos del lenguaje son:

Carácter: Este tipo de dato se declara con la palabra reservada char y ocupa un byte en memoria, con un byte se pueden representar 256 símbolos posibles.

Real: Este tipo de datos se declara con la palabra reservada double o float, si se utiliza la primera, entonces la variable que se declare ocupa 8 bytes de memoria y si se utiliza la segunda entonces la variable que se declare utiliza 4 bytes de memoria.

Entero: Este tipo de datos se declara con la palabra reservada int y ocupa 2 bytes de memoria. En algunos compiladores ocupa 4 bytes de memoria.

En la Tabla 1 se muestran todos los tipos de datos estándar en el lenguaje C.

1.4.1. Acerca de los tipos de datos reales (flotantes)

C proporciona los tipos flotantes **float** y **double** para manejar números de la forma 1.7, 0.0001, 3.14159. También existe una forma exponencial para representar un número, por ejemplo, 1.092332e5. La correspondiente notación científica de este número es:

1.092332e5 = 1.092332*10*10*10*10*10
= 1.092332*100000
= 109233.2

De forma similar se tiene el número 1.092332e-3, esto significa que el punto decimal se desplaza 3 lugares a la izquierda y se tiene el siguiente valor 0.001092332.

El número 333.777e.22 se puede descomponer de la siguiente forma:

Parte entera = 333
Parte fraccionaría = 777
Parte exponencial = e-22

Combinaciones	
Char	8 caracteres ASCII -128 a 127
unsigned char	8 caracteres ascii 0 a 255
signed char	8 caracteres ascii -128 a 127
int	16 bits -32768 a 32767
unsigned int	16 bits 0 a 65535
signed int	16 bits -32768 a 32767
short int	16 bits -32768 a 32767
unsigned short int	8 bits 0 a 255 0 a 65535
signed short int	16 bits -32768 a 32767
long int	32 bits -2147483648 a 2147483647
signed long int	32 bits -2147483648 a 2147483647
unsigned long int	32 bits 0 a 4294967295
float	32 bits 6 dígitos de precisión 3.4E-38 a 3.4E+38
double	64 bits 12 dígitos de precisión 1.7E-308 a 1.7E+308
long double	64 bits 12 dígitos de precisión 1.7E-308 a 1.7E+308

Tabla 1. Tipos de datos estandar

1.4.2. Acerca del tipo de datos char

Las constantes y las variables de tipo char se usan para representar caracteres y cada carácter se almacena en un byte.

Un byte esta compuesto de 8 bits, el cual es capaz de almacenar 2 a la 8 o 256 valores diferentes, pero solo un grupo pequeño de ellos es realidad representa a un conjunto de caracteres imprimibles.

A continuación se muestran algunas constantes enteras y sus valores enteros correspondientes.

Constante de tipo char	Valor entero correspondiente
'a'	97
'b'	98
....	
'z'	117
'0'	48
'1'	49
'2'	50
...	...
'9'	57

1.4.3. Acerca de las cadenas

Una cadena es una secuencia de caracteres entre comillas " ".Obsérvese que " es un solo caracter y no dos. Si el caracter (") tiene que aparecer en una cadena, éste debe de ir precedido por el carácter \.

Ejemplos:

```
"Una cadena de texto"
"                    "
"z"
"x-x-0-.1-basura"
"Una cadena con \" comillas"
"a+b=suma; x=cos(y)"
""                                  /*cadena nula*/
```

1.5. DECLARACIÓN DE VARIABLES Y CONSTANTES

Una **variable** es un identificador que tiene asociado un valor que puede cambiar a lo largo de la ejecución del programa.

Las **constantes** en C se refieren a valores fijos que no pueden ser alterador por un programa y pueden ser de cualquier tipo.

Las variables y las constantes son los objetos que manipulan un programa. En general se deben declarar las variables antes de usarlas.

1.5.1. Declaración de variables

Una **variable** en C se declara de la siguiente manera:

tipo identificador [, identificador , ..., identificador] ;

donde ,

tipo :determina el tipo de la variable (char, int, ...).
identificador: indica el nombre de la variable. Los corchetes ([ ]) indica que se pueden definir en línea mas de una variable del mismo tipo separadas por coma (,) y terminando con punto y coma (;).

Por ejemplo:

```
int i,j,k;
float largo, ancho;
char c;
```

El inicio de un programa en C se ve de la siguiente manera:

```
main ( )
{

   declaración de variables;

   proposiciones;

   return;

}
```

Las llaves '{' y '}' encierran un **bloque** de proposiciones y se usan para enmarcar declaraciones y proposiciones. **Si hay declaraciones, entonces estas deben de ir antes de las proposiciones**. Las declaraciones tienen dos objetivos:

1. Piden al compilador que separe la cantidad de memoria necesaria para almacenar los valores asociados con las variables.

2. Debido a que se especifican los tipos de datos asociados con las variables, éstas permiten al compilador instruir a la máquina para que desempeñe correctamente ciertas operaciones.

1.5.2. Declaración de CONSTANTES

Las constantes en C pueden ser:

Números reales	**Números enteros**	**Cadenas**	**Carácter**
3.10	1234	"hola C.U."	'a'
0.987	-10	""	'#'

Una **constante** (cualquier tipo de constante) en C se define de la siguiente manera:

#define identificador valor

donde,

identificador es el nombre de la constante
valor es el valor asociado a la constante

Ejemplo:

```
#define  entero     10
#define  real       1.09982
#define  cad        "Estoy definiendo una constante que se llama cad "
#define  car        'a'
```

1.6 EXPRESIONES, PROPOSICIONES Y ASIGNACIONES

Las **expresiones** son combinaciones de constantes, variables, operadores y llamados a funciones.

Algunos ejemplos de expresiones son:

```
tan(1.8)
a+b*3.0*x-9.3242
3.77+sen(3.14*98.7)
```

1.6.1 Operadores

Un **operador** es un símbolo que indica al compilador que se lleven a cabo específicas manipulaciones matemáticas o lógicas. El lenguaje C tiene tres clases de operadores: *aritméticos, relacionales y lógicos y de bits*. Además de otros operadores especiales.

1.6.2.Asignación simple

El signo de igualdad (=) es el operador básico de asignación en C. Un ejemplo de una **"expresión"** de asignación es: **i = 7**. A la variable **i** se le asigna el valor de **7** y la expresión como un todo toma ese valor.

Las proposiciones de asignación simples tiene la siguiente sintaxis:

variable = expresión;

1.6.3 Proposiciones

Cuando la expresión va seguida de un punto y coma (;) se convierte en una **proposición**.

Ejemplo de proposiciones:

```
i=7;
x=3.1+sin(10.8);
printf("hola");
```

Las siguiente proposiciones son válidas pero no tienen ningún significado útil:

```
3.10;
a+b;
```

1.6.4. Operadores Aritméticos

Los operadores aritméticos binarios se muestran en la siguiente tabla 2.

Operador	Operación
+	Suma. Los operandos pueden ser enteros o reales
-	Resta. Los operandos pueden ser enteros o reales
*	Multiplicación. Los operandos pueden ser enteros o reales
/	División. Los operandos pueden ser enteros o reales. Si ambos operandos son enteros el resultado es entero. En el resto de los casos el resultado es real.
%	Módulo o resto de la división entera. Los operandos tienen que ser enteros.
-(unario)	Menos unario. Los operadores pueden ser enteros o reales.

Tabla 2. Principales operadores aritméticos

Ejemplos:

```
int a=10, b=3, c;
float x=2.0, y;

y = x +a; /* El resultado es 12.0 de tipo float */
c = a / b; /* El resultado es 3 de tipo int */
c = a % b; /* El resultado es 1 de tipo int */
a = -b;  /* El resultado es –3 */
```

1.6.4.1. Prioridad de los operadores aritméticos

Los operadores tiene reglas de prioridad y asociatividad, estas reglas determinan la forma de evaluar las expresiones. Al evaluarse en primer lugar las expresiones que se encuentran entre paréntesis (), éstas pueden emplearse para aclarar o cambiar el orden de ejecución de las operaciones que se desean realizar.

La tabla 3. muestra las reglas de prioridad y asociatividad para los operadores aritméticos vistos hasta este momento.

Operadores	Asociatividad
-(unuario)	derecha a izquierda
*,/,%	izquierda a derecha
+,-	izquierda a derecha
=	derecha a izquierda

Tabla3. Prioridad de los operadores aritméticos

Los operadores de una misma línea como *, /, %, tienen la misma prioridad y ésta es mayor que la prioridad de las líneas inferiores. La regla de asociatividad que rige a los operadores con la misma prioridad se muestra en la columna de la derecha

Ejemplo:

Encontrar la expresión equivalente y el valor de la expresión, utilizando las siguientes declaraciones y asignaciones. También se debe de utilizar la tabla de prioridad mencionada anteriormente.

```
int a, b, c, d;   /*Se declaran 3 variables de tipo entero*/

a=2; b=-3; c=7; d=-19;
```

Expresión	Expresión Equivalente	Valor
a / b	(a / b)	0
b / b / a	(b / b) / a	-1
c%a	(c%a)	1
a%b	(a%b)	?
d/b%a	((d/b)%a)	0
-a*d	(-a)*d	38
a%-b*c	A%-(b*c)	14
9/c+-20/d	(9/c)+ ((-20)/d)	2
-d%c-b/a*5+5	(((-d)%c)-((b/a)*5))+5	15
7-a%(3+b)	7-(a%(3+b)	Error
- - - a	- (- (- a))	-2
A= b= c= -33	A= (b= (c= -33))	a=-33 y b=-33

1.6.5. Operadores de Relación y Lógicos

Al igual que los operadores anteriores, los operadores de relación y lógicos tiene reglas de prioridad y asociatividad que determinan de forma exacta la evaluación de la expresiones que los incluye.

Un operador relacional se refiere a la relación entre unos valores con otros, y un operador lógico se refiere a las formas en que estas relaciones pueden conectarse entre sí.

Los **Operadores de Relación** son binarios (tabla 4). Cada uno de ellos toma dos expresiones como operando y dan como resultado el valor int 0 o el valor int 1 (tómese en cuenta que en el lenguaje C cualquier valor distinto de 0 es verdadero y el cero es falso).

Operador	Operación	Ejemplos
<	Primer operando menor que el segundo	a<3
>	Primer operando mayor que el segundo	b>w
<=	Primer operando menor o igual que el segundo	-7.7<=-99.335
>=	Primer operando mayor o igual que el segundo	-1.3>=(2.0*x+3.3)
= =	Primer operando igual que el segundo	c= ='w'
!=	Primer operando distinto del segundo	x!=-2.77

Tabla 4. Operadores de Relación Los operandos pueden ser de tipo entero, real o apuntador.

Los **Operadores Lógicos** (tabla 5) al igual que los operadores anteriores cuando se aplican a expresiones producen los valores int 0 o int 1. La negación lógica es aplicable a una expresión arbitraria. Los operadores lógicos binarios && y || también actúan sobre expresiones.

Operador	Operación	Ejemplo
&&	AND. Da como resultado el valor lógico 1 si ambos operandos son distintos de 0. Si uno de ellos es cero el resultado es el valor lógico 0. Si el primer operando es igual a cero, el segundo operando no es evaluado.	(z<x) && (y>w)
||	OR. El resultado es cero si ambos operandos son 0. Si uno de los operandos tiene un valor distinto de 0, el resultado es 1. Si el primer operando es distinto de 0, el segundo operando no es evaluado.	(x= =y) || (z!=p)
!	NOT. El resultado es 0 si el operando tiene un valor distinto de cero, y 1 en caso contrario. El resultado es de tipo int. El operando puede ser entero, real o un apuntador.	!a

Tabla 5. Operadores lógicos

Ejemplo:

Operador de Negación

Expresión	Valor
!5	0
!a	Depende del valor de a
!'z'	0
!(x+7.7)	Depende del valor de x
!!5	1

Ejemplo:

Operadores Lógicos. Supóngase que se tienen las siguientes declaraciones y asignaciones:

```
char c;
int i,j,k;
double x,y;

c='w'; i=j=k=3; x=0.0; y=2.3;
```

Expresión	Expresión Equivalente	Valor
i && j && k	(i && j) && k	1
X && i || j-3	(x && i) || j-3	0
X || i && j-3	(x|| i) && (j-3)	0
I<j && x<y	(i<j) && (x<y)	0
I<j || x<y	(i<j) || (x<y)	1

I= =j && x<=y	(i= =j) && (x<=y)	1
I= = 2 || j= = 4 ||k= =6	(i= = 2) || (j= = 4) ||(k= =6)	0

1.6. 6 Operadores de manejo de bits

Los operadores (tabla 6) para este tipo de operaciones tienen que ser de tipo entero de uno o dos bytes o char, no pueden ser reales.

Operador	Operación
~	Complemento a 1. El operando tiene que ser entero
&	AND a nivel de bits
|	OR a nivel de bits
^	XOR a nivel de bits
<<	Corrimiento (desplazamiento) a la izquierda
>>	Corrimiento (desplazamiento) a la derecha

Tabla 6. Operadores para el manejo de bits

Ejemplo:

```
int a=0777, m=2;
a=a & 0177;      /* Pone a cero todos los bits de a excepto los 7 bits de
                    menor peso */

a=a | m;         /* pone a uno todos los bits de a que están a 1 en m */
a = a & ~077;    /* pone los 6 bits de menor peso de a, a 0 */
```

Ejemplo:
Multiplicación y división mediante operaciones de desplazamientos

char X;	X después de cada ejecución	Valor de X
X=7;	0 0 0 0 0 1 1 1	7 (Asignación)
X << 1;	0 0 0 0 1 1 1 0	14 (X * 2)
X << 3;	0 1 1 1 0 0 0 0	112 (X * 8)
X << 2;	1 1 0 0 0 0 0 0	192 (X * 4, con pérdida del bit más significativo)
X >> 1;	0 1 1 0 0 0 0 0	96 (X / 2)
X >> 2;	0 0 0 1 1 0 0 0	24 (X / 4)

1.6.7. Operadores de asignación

Lista de operadores de asignación

Operador	Operación
++	Incremento
--	Decremento
=	Asignación simple
+=	Suma más asignación
-=	Resta más asignación
|=	Operación OR sobre bits más asignación
&=	Operación AND sobre bits más asignación
>>=	Corrimientos a la derecha más asignación
<<=	Corrimientos a la izquierda más asignación
*=	Multiplicación más asignación
/=	División más asignación
%=	Módulo más asignación

Tabla 7. Operadores de asignación

Los operadores de incremento ++ y decremento -- son unarios y tienen la misma prioridad que el operador unario - , éstos se asocian de derecha a izquierda. Tanto ++ como -- se pueden aplicar a variables, pero no a constantes o a expresiones. Además pueden estar en la posición de prefijos o sufijos, con diferentes significados posibles.
Incremento: ++ añade 1 a una variable
Decremento: - - resta 1 a una variable

Ejemplo:

A+=3 es equivalente a A=A+3
k*=3+x es equivalente a k=k*(3+x)

x = x+1; equivalente a ++x;
x = x-1; equivalente a - -x;

Estos operadores pueden ir antes o después de la variable.

x = x+1; ++x; x++; x = x-1; - -x; x - -;

- **Antes de la variable**: Si el operador ++ o - - aparece antes del operando, entonces el operando se incrementa o decrementa antes de que se evalúe la expresión.

  Ejemplo:

  x = 10;
  y = ++x; ==> y = 11;

- **Después del operando**: Si el operador ++ o -- aparece después del operando, entonces primero se evalúa la expresión con el valor actual del operando y posteriormente se incrementa o decrementa el operando.

  Ejemplo:

  x = 10;
  y = x++; ==> y =10;

en ambos casos x = 11;

1.6.8. Tabla final de prioridad y orden de evaluación

Una expresión entre paréntesis, siempre se evalúa primero. Los paréntesis tienen mayor prioridad y son evaluados de más internos a más externos.

El operador coma (,) hace que las expresiones se evalúen de izquierda a derecha.

Operadores	Asociatividad
( )	Izquierda a derecha
-(unario), ++, --, ~, !, *, &(los últimos dos como operadores apuntadores)	Derecha a izquierda
*,/,%	Izquierda a derecha
+,-	Izquierda a derecha
<,<=,>,>=	Izquierda a derecha
= =, !=	Izquierda a derecha
&	Izquierda a derecha
^	Izquierda a derecha
|	Izquierda a derecha
&&	Izquierda a derecha
||	Izquierda a derecha
=, +=, -=, *=, <<=, >>=, %=	Derecha a izquierda
, (operador coma)	Izquierda a derecha

Tabla 8. Tabla final de todos los operadores en C, así como su prioridad y su asociatividad

Los operadores que se encuentran en las líneas superiores tiene más prioridad que los operadores de las líneas inferiores y los operadores que se encuentran en la misma línea tiene la misma prioridad y si existen operadores con la misma prioridad en una expresión, éstos se asocian según la columna de la derecha de la tabla anterior.

1.6.8.1. Otros operadores

El operador de **indirección** (*) accede a un valor indirectamente a través de un apuntador. El resultado es el valor direccionado por el operando.

El operador de **dirección-de** (&) indica la dirección de su operando. Este operador no se puede aplicar a un campo de bits perteneciente a una estructura o a un identificador declarado con el especificador register.

1.7. INSTRUCCIONES DE ENTRADA Y SALIDA

Las operaciones de entrada y salida no forman parte del conjunto de sentencias del lenguaje C, sino que pertenecen al conjunto de funciones de la librería estándar de entrada y salida de C. Por ello, todo programa que deberá contener la línea (o líneas) iniciales:

#include < stdio.h >

Esta línea le dice al compilador que incluya la librería **stdio.h** en el programa permitiendo así la entrada y salida de datos.

Las siguientes funciones son algunas de las más utilizadas para entrada y salida de datos.

printf, scanf, getch, getchar, puts, gets,

Todas y cada una de ellas tiene una sintaxis que las identifica.

1.7.1 Salida de datos utilizando la función printf ( )

printf*(cadena de control, lista de argumentos*)

cadena de control

especifica como va a ser la salida. Es una cadena delimitada por comillas (“ “), formada por caracteres ordinarios, secuencias de escape (ver tabla 9) y especificaciones de formato bajo el cual se requiere la salida de la información (datos) hacia pantalla

lista de argumentos

representa el valor o valores a escribir en la pantalla. Una especificación de formato está compuesta por:

% [flags] [width] [.prec] [F|N|h|l|L] **tipo_de_dato**

Cada uno de los datos que se desee mandar a imprimir debe de ir antecedido por el caracter **%** y después debe de venir (en este orden) lo siguiente (no es necesario poner todo, lo que se encuentra entre corchetes es opcional) :

Componente	**Que se especifica**
[flags]	(Opcional) Justificación, etc.
[width]	(Opcional) Número de dígitos significativos parte entera
[.prec]	(Opcional) Número de dígitos significativos parte real
[F|N|h|l|L]	(Opcional) Modificadores de salida N = near apuntador h = entero corto F = far apuntador l = entero largo L = real largo
tipo_de_dato	(requerido), el tipo de dato puede ser: c = imprime un carácter d = imprime un entero e = notación cientifica s = imp rime una cadena f = decimal en punto flotante

Ejemplo:

```
printf("hola puebla son las %d \n", tiempo);
```

Explicación:

*en este caso se pone el letrero “**hola Puebla**” (cadena de control) y se tiene una salida de un dato de tipo entero (%d), después se imprime la variable (lista de argumentos, en este caso solo hay un argumento o variable la cual se llama tiempo), después de imprimir el valor de la variable tiempo hará un cambio de línea (secuencia de escape \n).*

Ejemplo:

```
n1=5;
n2=6.7;
printf(“El dato 1 es: %d y El dato 2 es: %f \n”,n1,n2);
```

Explicación:

En este ejemplo el primer formato %d corresponde a un numero entero (n1) y el %f corresponde a un numero real (n2), dejando un cambio de línea al final(\n).

Salida: El dato 1 es: 5 y El dato 2 es: 6.700000

Secuencia	Nombre
\ n	Nueva línea
\ t	Tab horizontal
\ v	Tab vertical (solo para impresora)
\ b	Backspace (retroceso)
\ r	Retorno de carro
\ f	Alineación de página (solo para impresora)
\ a	Bell (sonido)
\ ´	Comilla simple
\ ”	Comilla doble
\ 0	Nulo
\ \	Bacslash (barra hacia atrás)
\ ddd	Carácter ASCII. Representación Octal
\ xdd	Carácter ASCII. Representación hexadecimal

Tabla 9. Secuencias de escape en lenguaje C

1.7.2. Entrada de datos utilizando la función scanf( )

La función **scanf( )** es una rutina de entrada de propósito general, que permite leer datos formateados y convertir automáticamente la información numérica a enteros o a flotantes por ejemplo. El formato general de esta función es:

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

cadena de control

esta formada por códigos de formato de entrada, que están precedidas por un signo **%** y encerrados entre comillas “ “ dobles. Ver tabla 10.

Código	Significado
C	Lee un único carácter
D	Lee un entero decimal base 10
I	Lee un entero en base 10, 16 u 8
F	Lee un número en punto flotante
E	Lee un número en punto flotante
H	Lee un número entero corto
S	Lee una cadena de caracteres
O	Lee un entero en octal
X	Lee un número hexadecimal

Tabla 10. Códigos de formato de **scanf()**

lista de argumentos

representa el valor o valores a escribir en la pantalla.

Una especificación de formato esta compuesta por:

Símbolo	Significado
*	Un asterisco a continuación de % suprime la asignación del siguiente dato en la entrada.
Ancho	Máximo numero de caracteres a leer de la entrada. Los caracteres en exceso no son tenidos en cuenta.
F	Indica que se quiere leer un valor apuntado por una dirección far(dirección segmentada).
N	Indica que se quiere leer un valor apuntado por una dirección near (dirección dada por el valor offset). F y N no pertenecen al C estándar.
H	Se utiliza como prefijo con los tipos d, i, n, o y x, para especificar en el argumento es short int, o con u para especificar un short unsigned int.
L	Se utiliza como prefijo con los tipos d,i,n,o, y x, para especificar que el argumento es long int. También se utiliza con los tipos e,f y g para especificar un double.
Tipo	El tipo determina si el dato de entrada es interpretado como un carácter, como una cadena de caracteres o como un número.

Ejemplo:

```
printf(“Da un numero : ”);
scanf(“%d ”, &n);
```

Explicación:

La función scanf() lee un dato de tipo entero y es almacenado en la variable n. El & es necesario para leer datos de tipo entero y real.

Cuando se especifica mas de un argumento, los valores correspondientes en la entrada hay que separarlos por uno o mas espacios en blanco (' '),tabuladores(\t) y cambios de línea(\n).

La función scanf() devuelve un entero correspondiente al número da datos leidos de la entrada.

Ejemplo:

```
Main()
{
        int a,r;
        float b;
        char c;
        printf("Introducir un valor entero, un real y un carácter \n => ");
        r=scanf(" %d %f %c \n",a,b,c);
        printf("Numero de datos leidos: %d \n",r);
        printf(" Datos leídos: %d %f %c \n",a,b,c);
}
```

Salida:

```
        Introducir un valor entero, un real y un carácter
        => 12 3.5 x

        Numero de datos leidos: 3
        Datos leídos: 12 3.500000 x
```

Para leer datos de tipo cadena no lleva el operador de dirección &. Y tenemos que cambiar el formato %s por %[^\n].

Ejemplo:

```
char nombre[40];
        scanf("%[^\n]", nombre);
        printf("%s",nombre);
```

Entrada: Francisco Javier

Salida: Francisco Javier

Si en lugar de especificar el formato **%[^\n]** se hubiera especificado el formato **% s** , el resultado hubiera sido: **Francisco**.

1.7.3. Entrada de caracteres getchar( )

Lee un carácter de la entrada estándar y avanza la posición de lectura al siguiente carácter a leer

```
int getchar(void);
```

Ejemplo:

```
        car = getchar();
```

Resultado: Lee un caracter y lo almacena en la variable car.

1.7.4. Salida de caracteres putchar( )

Escribe un carácter en la salida estándar en la posición actual y avanza a la siguiente posición de escritura.

```
int putchar(int ch);
```

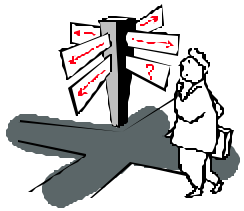
Ejemplo:

```
        putchar(ch);
```

Salida: escribe en pantalla el carácter contenido en la variable car

2

ESTRUCTURAS DE CONTROL DE PROGRAMAS



Los programas que se suelen redactar en la practica incluyen algún tipo de elementos de control lógico, es decir, aparecen comprobaciones de condiciones que se sean ciertas o falsas (selección o decisión). Además, los programas pueden requerir que un grupo de instrucciones se ejecute repetidamente un determinado número de veces o hasta que satisfaga alguna condición lógica (ciclo de repetición).

En el lenguaje C se incluyen las siguientes estructuras condicionales: if y switch Y las sentencias de repetición : for, while y do/while.

Utilizamos indistintamente Proposiciones, Acciones y Sentencias para referirnos al mismo concepto.

2.1. PROPOSICIÓN if

Una proposición if es una construcción de la forma:

```
if (expresión)
        {
          propocisiones; /* bloque */
        }
proposición_siguiente;
```

donde:

expresión

debe ser una expresión numérica, relacional o lógica.

proposiciones

si se tiene solo una proposición entonces no es necesario poner llaves {}, si tenemos varias acciones(proposiciones) a realizar cada una de estas será separada por punto y coma (;).

Si el resultado de la expresión es verdadera (cualquier valor distinto de cero), se ejecutara la proposición o proposiciones (bloque).

Si el resultado es falso entonces no se realiza ninguna acción y continua el flujo del programa a la siguiente línea de código después del if (proposición _siguiente).

Ejemplos:

```
if (grado>=90)
        printf("\n FELICIDADES"):

printf("\nSu grado es %d", grado);
```

Explicación:

Se evalúa la expresión (grado >=90) si es verdadera se muestra el letrero FELICIDADES y escribe "Su grado es : x". Si la expresión es falsa entonces solo se muestra el letrero "Su grado es : x".

2.2. PROPOSICIÓN if-else

Una proposición if-else es una construcción de la forma:

```
if (expresión)
        {
          proposición_1 /* bloque 1*/
        }
else    {
          proposición_2 /* bloque 2*/
        }
proposición_siguiente;
```

Se evalua la expresión si es verdadera entonces se ejecuta el bloque 1 que puede estar formado por una proposición (no usar llaves {})o un conjunto de proposiciones (usar llaves {}).

Si la expresión es falsa entonces se ejecuta la proposición 2 o el bloque 2.

Ejemplo:

```
if (x<=y)
        min=x
else
        min=y;
```

Expliación:

En este segmento del programa se asigna a la variable min el menor de x y y.

Ejemplos:

```
if (a>=0)
        printf("\el numero es positivo");
else
        printf("\el numero es negativo");
```

2.2.1. Anidamiento de sentencias if

Las sentencias if-else pueden estar anidadas. Esto quiere decir que como proposión1 o proposición2, deacuerdo al formato anterior, puede escribirse otra sentencia if.

```
if (expresion1)
        Sentencias1;
else if(expresión2)
        Sentencias2;
    else if(expresión3)
            sentencia3;
....
            else
                sentenciasN;
```

Si se cumple la expresion1, se ejecuta las sentencias1, si no se cumple se examinan secuenciales las expresiones siguientes hasta el else, ejecutándose las sentencias correspondientes al primer else if, cuya expresión sea cierta. Si todas las expresiones son falsas, se ejecutan las sentenciasN.

El if se puede anidar indistintamente después del if o después del else y puede ser en cada una de estas una sentencia o un conjunto de sentencias.

Ejemplo

```
if (a>b)
    printf("%d es mayor que %d",a,b);
else if (a<b)
        printf("%d es menor que %d",a,b);
    else
        printf("%d es igual a %d",a,b);
```

Cuando en una línea de programa aparecen anidadas sentencias if-else, la regla para diferenciar cada una de estas sentencias, es que cada else se corresponde con el if más próximo que no haya sido emparejado.

2.3. LA PROPOSICIÓN switch

Esta proposición permite ejecutar una de varias acciones, en función del valor de una expresión.

El formato de esta proposición es:

```
switch (expresión-test)
        {
            case constante1: sentencia;
            case constante2: sentencia;
            case constante3: sentencia;
            ...
            default : sentencia
        }
```

donde:

expr-test
es una constante entera, una constante de caracteres o una expresión constante. El valor es convertido a tipo int.

sentencia
es una sentencia simple o compuesta (bloque).

La sentencia **switch** evalúa la expresión entre paréntesis y compara su valor con las constantes de cada case, si se cumple la **expresión-test** con alguna constante entonces se ejecutaran todas las sentencias después de la igualdad. Usualmente se utiliza la sentencia **break** para romper la sentencia switch.

En el caso de que ninguna de las constantes coincida con la expresión-test entonces se ejecuta la opción default. La sentencia switch puede incluir cualquier número de cláusulas case y opcionalmente la cláusula default.

Ejemplo:

```
char ch;
ch=getchar();
switch (ch)
    {
        case '0':
        case '1':
        case '2':
        case '3':
        case '4':
        case '5':
        case '6':
        case '7':
        case '8':
        case '9':   printf("\n Es un digito ");
                    break;
        case ' ':
        case'\n':
        case'\t': printf("\n Es un separador");
                            break;
        default: printf("\n Otro");
                 break;
    }
```

Explicación:

Se almacena un caracter en ch y se evalua en el switch si este carácter esta entre '0' .. '9' entonces se dice que es un digito y termina el switch con el break. Si el carácter es un separador se indica con un letrero, pero si no es ninguno de los anteriores, entonces se dice que es "otro".

2.4. PROPOSICIÓN for

Cuando se desea ejecutar una acción simple o propuesta, repetidamente un número de veces conocido, es recomendable utilizar la proposición for.

```
for (inicialización ; condición ; incremento)
{
        sentencia1;
        sentencia2;
        ...
        sentencian;
}
```

donde:

inicialización:
es una proposición de asignación que se utiliza para establecer la variable de control. Se pueden utilizar varias variables de control en un ciclo de repetición for.

condición:
es una expresión relacional o lógica que determina cuando terminara el ciclo de repetición.

incremento:
define como cambiara la variable de control o las variables de control cada vez que cambia éste.

Estas tres partes tienen que estar separadas por **puntos y comas (;)**. Si la acción a repetir es solo una no es necesario incluir las llaves{}.

Ejemplo: Imprimir los números del 1 al 100.

```
for (i =1 ; i <= 100 ; i++)
        printf("%d",I);
```

Literalmente dice: desde I igual a 1, mientras I sea menor o igual que 100, con incrementos de 1, escribir el valor de i.

Ejemplo: Imprimir los múltiplos de 7 que hay entre 7 y 112

```
for (k = 7 ; k <= 112 ; k = k+7)
        Printf("%d",k)
```

Ejemplo: codificación de la suma de 10 números (del 1 al 10).

```
suma=0;
for(i=0; i<=10; i++)
        suma+=i;
```

Explicación:
Se inicializa la variable suma en 0 para ir acumulando la suma de los valores de i que van desde 0 hasta 10.

Ejemplo: Se imprimen los números del 9 al 1.

```
for (a=9 ; a >= 1 ; a--)
        Printf("%d",a);
```

2.5. PROPOSICIÓN while

Ejecuta una sentencia simple o compuesta ({}), cero o más veces, dependiendo del valor de una expresión.

```
while (expresión)
{
        sentencia1;
        sentencia2;
        sentencia3;
        ...
        sentencian;
}
```

expresión:
es cualquier expresión numérica, relacional o lógica.

sentencia_i:
es una sentencia (no es necesario las llaves({}) o un conjunto de sentencias.

Una proposición **while** se ejecuta mientras la expresión sea verdadera (cualquier valor distinto de cero), en el momento en que se convierte en falsa la expresión entonces se ejecuta la siguiente línea después del fin del while.

Ejemplo:

```
char ca;
ca=0;
while (ca!='A')
{
        ca=getchar();
        putchar(ca);
}
```

Explicación:

*Se puede ver que **ca** se inicializo con cero, de esta forma podemos iniciar el while, entonces se lee un carácter con la función getchar() y se manda a escribir a la pantalla, a continuación se compara este valor de **ca** con **'A'** si es distinta entonces vuelve a leer otro carácter y a escribirlo en pantalla. Se estará entonces pidiendo caracteres hasta que el carácter **ca** se igual a 'A' (haciéndose falsa la condición).*

Ejemplo: Realizar un programa que nos imprima los números z, comprendidos entre 1 y 50, que cumplan la expresión:

$$z^2 = x^2 + y^2$$

z,x e y son números enteros positivos.

```
/* Cuadrados que se pueden expresar como suma de otros dos cuadrados */
#include <stdio.h>
#include <math.h>
  main()
  {
        unsigned int x,y,z;
        printf("%10s %10s %10s \n","Z","X","Y");
        printf("----------------------------------------\n");
        x=1;
        y=1;
        while (x <= 50)
        {
           /* calcular la parte entera (z) de la raiz cuadrada */
           z = sqrt(x * x + y * y);
           while (y <= 50 && z <=50)
           {
                /*comprobamos si z es suma de dos cuadrados perfectos*/
                if(z*z == x*x + y*y)
                        printf("\n %10d %10d %10d",z,x,y);
                y++;
                z = sqrt(x*x + y*y);
           }
           x++;
           y = x;
  }
```

2.6. PROPOSICIÓN do-while

Ejecuta una sentencia o varias una o mas veces, dependiendo del valor de una expresión.

```
do
  {
    sentencia1;
    sentencia2;
    sentencia3;
    ...
    sentencian;
  }while (expresión);
```

donde:

expresión
: es cualquier expresión numérica, relacional o lógica.

Sentencia:
: es una sentencia simple (sin usar llaves {}) o un bloque de sentencias

La sentencia **do-while** se ejecuta de la siguiente manera. Se ejecuta primero la sentencia o bloque de sentencias dentro del do. Se evalúa la expresión si es falsa (igual cero) termina la proposición do-while y si al evaluar la expresión el resultado es cualquier valor distinto de cero entonces se repite la sentencia o sentencias dentro del do {}.

Ejemplo:

```
int n;
do
{
        printf("\n Da un número : ");
        scanf("%d",&n);
}while ( n>100);
```

Explicación:

se leeran números enteros de teclado hasta que se de un número menor que 100.

Ejemplo: Calcular la raíz cuadrada de un número n, por el método de Newton que dice:

ri+1 = (n/ri + ri)/2

La raíz calculada será válida, cuando se cumpla que:

abs(ri –ri+1) < = épsilon

```
/* Raiz cuadrada de n por el metodo de Newton */
#include <stdio.h>
main()
{
    double n; /* número */
    double aprox;  /* aproximación a la raíz cuadrada */
    double antaprox;  /* anterior aproximación a la raíz cuadrada */
    double epsilon;  /* coeficiente de error */
    printf("Numero:                          ");
    scanf("%lf",&n);
    printf("Raiz cuadrada aproximada:        ");
    scanf("lf",&aprox);
    printf("Coeficiente de error:            ");
    scanf(%lf",&epsilon);
    do
    {
        antaprox = aprox;
        aprox = (n / antaprox + antaprox) / 2;
    }while ( fabs (aprox – antaprox) >= epsilon);
    /* fabs calcula el valor absoluto de una expresión real */
    printf(" \n \n La raiz cuadrada de %.2lf es %.2lf",n, aprox);
    /* el .2 en el %.2lf nos toma solo dos cifras después del punto */
}
```

Entrada:

```
Número:                          10
Raiz cuadrada aproximada:        1
Coeficiente de error:            1e-6
```

Salida:

```
La raiz cuadrada de 10.00 es 3.16
```

2.7. CICLOS ANIDADOS

La proposición for al igual que el while y el do-while puede colocarse dentro de otro for (while o do-while) y entonces se dice que están anidados. En estos casos el ciclo de repetición más interno se ejecutara primero totalmente, por cada valor del ciclo que lo contiene.

Ejemplo: Escribir un programa que imprima un triangulo construido con caracters (#).

```
#
# #
# # #
# # # #
# # # # #
```

```
/* construcción de un triangulo de n filas con # */
#include <stdio.h>
main()
{
  unsigned int filas, columnas;
  unsigned int nfilas;
  printf(" Número de filas del triángulo : ");
  scanf("%d",&nfilas);
  for (filas =1 ; filas <= nfilas ; filas++)
  {
        for ( columnas =1 ; columnas <= filas ; columnas++)
              printf("#  ");
        printf ("\n");
  }
}
```

2.8. LA SENTENCIA break

La sentencia **break** finaliza la ejecución de una proposición switch, for, while y do-while en la cual aparece, saltándose la condición normal del ciclo de repetición.

Ejemplo: Un break causara una salida sólo del ciclo más interno.

```
for (t=0;t<100; t++)
{
        contador=1;
        do{
              printf("%d ",contador);
              contador++;
              if (contador == 10) break;
           }while(1) /* por siempre */
}
```

Explicación:

se imprimira 100 veces en la pantalla los números del 1 al 10. Cada vez que se encuentre break, el control se devuelve al ciclo for, terminando en do-while.

2.9. LA FUNCIÓN exit()

Otra forma de terminar un ciclo de repetición desde dentro es utilizar la función **exit(**). A diferencia con el break, la función exit terminara con la ejecución del programa y regresara el control al sistema operativo.

Ejemplo:

```
for (i =0 ; i <= 1000 ; i++)
{
        printf(" El valor de I es %d",I);
        if (I>10) exit();
}
```

Explicación: Aunque se ha elaborado una estructura para que la varable de control I tome los valores de 0a 100 cuando llega al valor de 11 termina el programa y solo escribira de 0 a 10 para el valor de i.

2.10. LA SENTENCIA continue

La sentencia continue actuá al revés que la sentencia break: obliga a que tenga lugar la siguiente iteración, saltándose cualquier código intermedio.

Ejemplo:

```
do {
        printf (" Da un número : ");
        scanf("%d",&x);
        if (x<0) continue;
        printf(" El número es %d",x);
   }while ( x != 100)
```

Explicación:

aquí solo se imprimen los números positivos; introducir un número negativo provocara que el control del programa se salte a evaluar la expresión x!=100.

3

TIPOS ESTRUCTURADOS DE DATOS

Muchas aplicaciones requieren el procesamiento de múltiples datos que tienen características comunes y es conveniente colocarlos en estructuras donde todos los elementos comparten el mismo nombre. Para el caso de datos del mismo tipo se tienen los *arreglos* y para datos de tipos diferentes tenemos las e*structuras.*

3.1. ARREGLOS

Un arreglo es una estructura homogénea, compuesta por varias componentes, todas del mismo tipo y almacenadas consecutivamente en memoria. Cada componente puede ser accedido directamente por el nombre de la variable del arreglo seguido de un subíndice encerrado entre corchetes ([ ]) .

3.1.1. arreglos unidimensionales

Un arreglo unidimensional es simplemente una lista con información del mismo tipo. La declaración de un arreglo unidimensional es de la siguiente manera:

tipo nombre [tamaño];

tipo

es uno de los tipos predefinidos por el lenguaje, es decir int, float, etc.

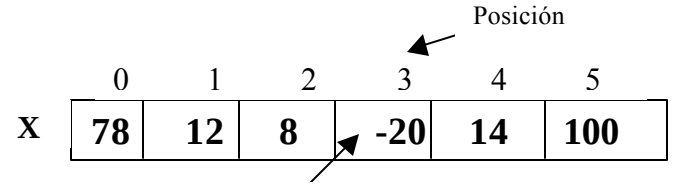
nombre

es un identificador que nombra el arreglo .

tamaño

es una constante que especifica el numero de elementos del arreglo.

int X[6];



Contenido del arreglo X en la posición 3

Explicación:

x es un arreglo de números enteros, para acceder a los datos en el arreglo se hace mediante el ***nombre*** *y la* ***posición****. Por ejemplo para almacenar el –20 en la posición 3 del arreglo X se hace de la forma: X[3]=-20; Para obtene ralgun valor del arreglo:C=X[2]; /* C=8 */*

Los arreglo en C siempre inician en la posición 0.

Ejemplos:

char x[20]; /* De esta forma se define una cadena de 20 caracteres, que se manejan desde x[0] hasta x[19] */

float z[40]; /* Es un arreglo donde cada uno de los datos es un numero real, los datos se manejan desde z[0] hasta z[39]*/

int vector[100]; /* Es una arreglo de enteros, los datos que se manejan son desde vector[0] hasta vector[99] */

Ejemplo: Sumar dos vectores de tamaño n.

```
#include <stdio.h>
main( )
{
  int n;   /* donde n<=100 */
  int a[100], b[100], c[100];  /* Arreglos */
  int i; /* indice */

  printf (“Numero de elementos a sumar: “);
  scanf(“%d”,&n);
  printf (“Elementos del vector a \n”);
  for (i=0; i<n; i++)
    scanf(“%d”,&a[i]);
  printf (“Elementos del vector b \n”);
  for (i=0; i<n; i++)
    scanf(“%d”,&b[i]);
  printf (“Suma de vectores \n”);
  for (i=0; i<n; i++)
    c[i]=a[i] + b[i];
  printf (“Resultados \n”);
  for (i=0; i<n; i++)
    printf(“%d”,c[i]);
}
```

3.1.1.1 Cadenas

Las cadenas en C son arreglos unidimensionales de tipo char. Por convención una cadena en C se termina con el centinela de fin de cadena o carácter nulo **‘\0’**.

Ejemplo: La siguiente es una función que dada una cadena de 4 elementos, calcula el número de caracteres que contiene dicha cadena.

```
main ( )
{
   char  * cadena ;
   int tam,i;
   printf( “ dame la cadena ” );
   gets ( cadena );
   for ( i = 0; cad [ i ]! = “ \ 0 ”; i + +);
   tam = i;        / * tam= largo de una cadena * /
   printf(“La cadena %s tiene %d elementos”,cadena, tam);
};
```

El lenguaje C nos proporciona una variedad de funciones para la manipulación de las cadenas en la biblioteca ***string.h***.

3.1.2. Arreglos bidimensionales

A los arreglos bidimensionales generalmente se les llaman **tablas** o **matrices**. Para acceder a los elementos de una matriz se utilizan dos **índices,** uno para indicar el renglón y el segundo para indicar la columna.

Declaración:

Tipo nombre[renglones][columnas];

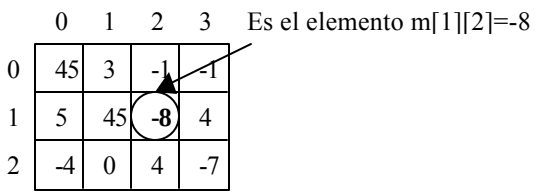
Tipo

es uno de los tipos predefinidos por el lenguaje, es decir int, float, etc.

nombre
es un identificador que nombra el arreglo .
renglones
indica el numero de renglones de la matriz
columnas
indica el numero de columnas de la matriz

Ejemplo:

int m[3][4];



Explicación:
m es una matriz de numeros enteros formada por 3 renglones (etiquetados con las posiciones 0,1 y 2) y 4 columnas(etiquetadas con las posiciones 0,1,2 y 3). m[1][2]=-8 coloca el dato en el renglón 1 y la columna 2. Si hacemos c=m[2][3], obtenemos el valor de –7 que se almacena en la variable c.

Ejemplo: Elaborar un programa que sume dos matrices de n-rengloes y m-columnas.

```
#include <stdio.h>
main()
{
   int n,m,r,c;
   int ma[10][10],mb[10][10], mc[10][10];
   printf("Da el numero de renglones: ");
   scanf("%d",&n);
   printf("Da el numero de columnas: ");
   scanf("%d",&m);
   printf("Lectura de la Matriz 1: \n")
   for (r=0 ; r<n;r++)
          for(c=0; c<m;c++)
             scanf("%d",&ma[r][c]);

   printf("Lectura de la Matriz 2: \n")
   for (r=0 ; r<n;r++)
          for(c=0; c<m;c++)
             scanf("%d",&mb[r][c]);

   printf("Sumando Matrices \n")
   for (r=0 ; r<n;r++)
          for(c=0; c<m;c++)
             mc[r][c]=ma[r][c]+mb[r][c];

   printf("Resultados : \n")
   for (r=0 ; r<n;r++)
   {
          for(c=0; c<m;c++)
             printf("%d",mc[r][c]);
          printf("\n");
    }
}
```

3.1.3. Arreglos multidimensionales

Los arreglos pueden ser de cualquier dimensión, aunque los más usuales son de una, dos o hasta tres dimensiones y al igual que los arreglos y las matrices los índices empiezan en 0.

La sintaxis general de un arreglo multidimensional es:

tipo nombre[tamaño1][tamaño2].....[tamaño-n];

tipo
es uno de los tipos predefinidos por el lenguaje, es decir int, float, etc.
nombre
es un identificador que nombra el arreglo .

tamaño1
indica el número de elementos de la primera dimensión
tamaño2
indica el número de elementos de la segunda dimensión
.
tamaño-n
indica el número de elementos de la n-ésima dimensión

Ejemplo:

```
int a[2][3][4][5][3];
char m[10][5][6];
```

Explicación:

*este ejemplo declara un arreglo denominado a de cinco dimensiones. El numero de elementos es 2*3*4*5*3 = 360 datos de tipo entero. El primer elemento es a[0][0][0][0][0] y el ultimo es a[1][2][3][4][2]. También se declara un arreglo m de tres dimensiones con 10*5*6=300 datos de tipo char, el primer elemento es m[0][0][0], y el ultimo es m[9][4][5].*

Nota: El lenguaje C no revisa los limites de un arreglo. Es responsabilidad del programador el realizar este tipo de operaciones.

3.2. ESTRUCTURAS

Una estructura es una colección de datos elementales (básicos) de diferentes tipos, lógicamente relacionados. A una estructura se le da también el nombre de **registro**. Crear una estructura es definir un nuevo tipo de datos. En la definición de la estructura se especifican los elementos que la componen así como sus tipos. Cada elemento de una estructura se denomina **miembro** (o **campo**). Declaración de una estrucura:

```
struct nombre_estructura
        {
          tipo nombre_variable;
          tipo nombre_variable;
          tipo nombre_variable;
           ...
        } variables_estructura;
```

nombre_estructura
es un identificador que nombra el nuevo tipo definido, la estructura
variables_estrutura
son identificadores para acceder a los campos de la estructura

Después de definir un tipo estructura, podemos declarar una variable de este tipo de la forma:

struct nombre_structura variable_estructura1, variable_estructura2„...;

Ejemplo:

```
struct ficha
        {
          char nombre[20];
          char apellidos[40];
          long int ndi;
          int edad;
        }
struct ficha provedor, cliente;
```

Este ejemplo define las variables **proveedor** y **cliente** de tipo ficha, por lo que cada una de las variables consta de los campos: **nombre**, **apellidos**, **ndi** y **edad**.

Para asignar datos en la estructura se hace mediante la **variable_estructura** y el campo donde se almacenara el valor separados por un punto (**.**). En el ejemplo anterior para almacenar para poner edad en el **cliente**:

cliente.edad = 22;

3.2.1 Arreglos de estrucuras

Una de las ventajas de las estructuras es utilizarlas como arreglos de estructuras, donde lo único que tendríamos que aumentar seria definir una **variable_estructura** como un arreglo de tamaño cualquiera.

Ejemplo: De la declaración de la estructura **ficha**.

struct ficha cliente[20] ;

Ahora Para almacenar la **edad** en alguno de los **clientes**.

cliente[3].edad = 25;

donde se esta asignando la edad 25 al cliente que se encuentra en la posición 3 (recuerde que los arreglos inician desde 0, por lo que en realidad es la posición 4 del mismo).

Ejemplo: El siguiente programa lee una lista de 10 alumnos y su correspondiente nota final de curso, dando como resultado el tanto por ciento de alumnos aprobados y reprobados.

```
#include <stdio.h>
#define NA 10                   /* número máximo de alumnos*/
main()
{
  struct datos
        {
          char nombre[60];
          float nota;
        }
  struct datos alumnos[NA];   /* arreglo se estructuras */
  int n=0,i;
  char fin;          / apuntador al  nombre  leído */
  float aprobados=0, reprobados=0;

  /*  Entrada de datos */
  printf("Nombre:     ");
  fin = gets(alumnos[n].nombre);
  while ( n < NA)
  {
        printf(" Calificación :  ");
        /* %*c se utiliza para eliminar el retorno de carro */
        scanf("%f%*c",&alumnus[n++].nota);
        printf("Nombre:     ");
        fin = gets(alumnos[n].nombre);
   }
   /* Calcular resultados */
   for (I=1ñi< n; i++)
   if (alumnus[i].nota >= 6)
        aprobados++;
   else
        reprobados++;
   /* escribir resultados */
   printf ("Aprobados %.4f  \n", aprobados/n*100);
   printf ("Reprobados %.4f  \n", reprobados/n*100);
}
```

3.3. UNIONES

La unión es una variable que puede contener, datos de diferentes tipos, denominados **miembros** o **elementos**, en una misma zona de memoria.

```
union nombre_union
        {
          tipo nombre_variable;
          tipo nombre_variable;
          tipo nombre_variable;
           .....
        }variables_union;
```

nombre_union
es un identificador que designa el nuevo tipo union

variables_union
son identificadores para acceder a los campos de la union

La declaración de una unión tiene la misma forma que la de una estructura, excepto que en lugar de la palabra reservada **struct** se pone la palabra **union**. Todo lo expuesto para las estructuras es aplicable a las uniones, excepto la forma de almacenamiento de sus elementos. Cuando se define una estructura cada uno de sus componentes ocupa una localidad de memoria diferente, mientras en una union se utiliza el espacio de memoria igual a la que ocupa el miembro mas largo de la union.

Después de definir un tipo union, podemos declarar una o mas variables de ese tipo de la forma:

union nombre_union variable1, variable2,..., variablen;

Para referirse a un determinado miembro o campo de la union,. Se utiliza la notación:

variable_union.campo

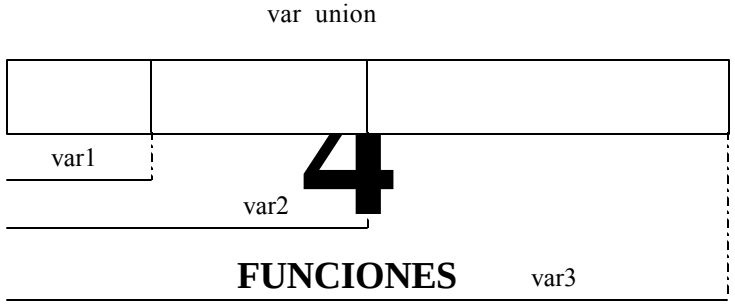
Ejemplo:

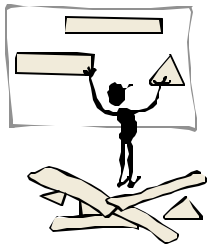
```
union info
        {
          char var1;
          int var2;
          int var3;
        }
union info var_union;
```

Explicación:

en este ejemplo, se declara una variable var_union que representa a una union llamada info, con tres miembros o campos. La variable var_union debe ser lo suficientemente grande como para contener el mayor de los tres campos. Cualquiera de los tres campos puede asignarse a var_union y utilizar en expresiones. Es responsabilidad del programador recordar cual es el campo actual que esta en la union.

El esquema siguiente muestra una union, donde var_union ocupa un tamaño de memoria igual al tamaño de memoria ocupado por el mayor de sus elementos.





El Lenguaje C utiliza funciones de biblioteca con el fin de realizar un cierto número de operaciones. Sin embargo, C permite también al programador definir sus propias funciones que realicen determinadas tareas. El uso de funciones (procedimientos) definidas por el programador permite dividir un programa grande en cierto número de componentes más pequeñas (modularización), cada una de las cuales con un propósito único e identificable.

4. FUNCIONES

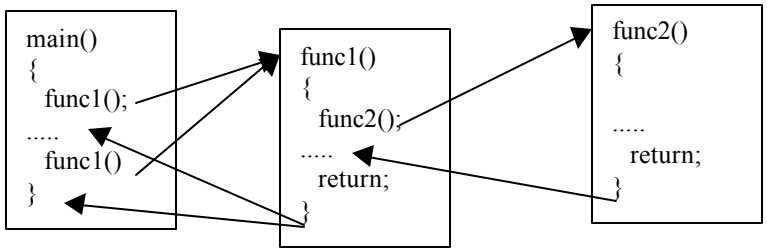
4.1. FUNCIONES

Las funciones son bloques con los que se construyen programas C, y en ellos se lleva a cabo toda actividad del programa. Una vez que una función ha sido escrita y depurada, puede utilizarse una y otra vez. Este es uno de los aspectos más importantes del C como lenguaje de programación.

Todo programa en C consta al menos de una función, la función main() que es donde inicia la ejecución de un programa.

Las funciones nos van a permitir dividir un problema, en subproblemas más fáciles de realizar, lo que facilitará la realización y el mantenimiento del programa.

Cuando una función se llama, el control se pasa a la misma para su ejecución y cuando ésta finaliza, el control es devuelto de nuevo al módulo que la llamo, para continuar con la ejecución del mismo, a partir de la secuencia que efectuó la llamada (ver figura siguiente)



Flujo de control del llamdo de funciones

La definición de una función consiste en un **encabezado** y **un cuerpo**. De manera explicita, se puede decir que es un bloque o una proposición compuesta. La estructura básica de la definición de una función es:

```
tipo nombre([,argumentos])
{
  [declaraciones]
  proposiciones
  [return(expresión);]
}
```

- **Encabezado de una función.**

tipo:

indica el tipo del valor devuelto por la función. Puede ser cualquier tipo básico, estructura o unión. Por defecto es int. Cuando no queremos que devuelva ningún valor usaremos el tipo void.

nombre:

es un identificador que indica el nombre de la función. Si el nombre va precedido por un *, el resultado devuelto por la función en el return será un apuntador. Una función no puede retornar arrays o funciones pero si puede retornar un apuntador a un array o a una función.

[argumentos]:

es una secuencia de declaraciones de parámetros separados por comas, cada argumento(o parámetro) deberá ir con su tipo correspondiente. Si no se pasan argumentos a la función podemos utilizar la palabra reservada *void*.

- **Cuerpo de una función.**

El cuerpo de la función está formado por una sentencia compuesta que contiene sentencias que definen lo que hace la función. También puede contener declaraciones de variables utilizadas en dichas sentencias. Estas variables, por defecto son locales a la función.

[return(expresion)]:

se utiliza para devolver el valor de la función el cual debe ser del mismo tipo declarado en el encabezado de la función. Si la sentencia return no se especifica o se especifica sin contener una expresión, la función no devuelve un valor.

Ejemplos:

```
void letrero(void)
{
        printf("\n Esta es una función muy simple");
        return;
}
```

El siguiente ejemplo es más complicado:

```
void letrero(int i)
{
        if (i>0)
                printf("\n i es positivo");
        else
```

```
                printf("\n i es negativo");
        return;
}
```

Ejemplo: la siguiente función usa dos argumentos de tipo entero y regresa un valor (la suma de ambos argumentos) de tipo entero.

```
int suma(int a, int b)
{
   int valor;
   valor = a+b;
   return(valor);
}
```

- **LLamado a una función**

Para llamar a una función se hace mediante su nombre con los argumentos entre paréntesis. Generalmente se asigna el valor de la función a una variable del mismo tipo de esta.

Ejemplo:

```
#include <stdio.h>
main()
{
        letrero();  /* llamado a la función letrero definida antes */
        return;
}
```

Ejemplo:

```
main()
{
        letrero(10); /* llamdo de la segunda funcion con
        un argumento*/
        getch();
        return;
}
```

Ejemplo: llamado de la función **suma**, en el primer llamado se dan los argumentos constantes y en el segundo llamado se pasan los valores en dos variables, en ambos casos el resultado de evaluar la función esta en la variable **res** que es del mismo tipo que la función **suma**.

```
main()
{
        int res;
        int n1=23;
        int n2=55;
        res = suma (5,10);
        printf("La suma es : %d\n",res);
        res = suma(n1,n2);
        printf("La suma de %d + %d = %d\n",n1.n2,res);
}
```

4.2. Declaración de la función prototipo

Una función prototipo (declaración forward) permite conocer el nombre, el tipo de resultado, los tipos y nombres de los argumentos, pero no se define el cuerpo de la función. Esta información permite al compilador char los tipos de los parámetros actuales por cada llamada a la función. Una función prototipo tiene la misma sintaxis que la definición de una función, excepto que esta, termina con un punto y coma (;). Es necesario hacer una declaración forward cuando se hace el llamado de la función antes de haberla definido.

Ejemplo:

```
main()
{
        /* declaración de una función prototito */
        double escribir(duble x, int y);
```

```
        douible r,a=3.14;
        int b= 5;
        r = escribe(a,b) ; /* llamada a la función */
        printf("%.2lf \n",r);
}

/* definición de la función */
duoble escribir(double x, int y)
{
        return( x + y * 13 );
}
```

Explicación:

En este ejemplo. La declaración forward o funcion prototipo es necesaria, ya que se esta llamando en el main() antes de haber sido declarada.

Si la función se define antes del llamado de ésta en algún lugar del programa, entonces no es necesario hacer una declaración **forward**.

4.3. Paso de parámetros

En C existen dos tipos de paso de parámetros en el llamado a una función:

Paso de parámetros por valor :

Pasar parámetros por valor, significa copiar los parametros actuales en sus correspondientes *lista de argumentos*, operación que se hace automáticamente, con lo cual no se modifican los argumentos actuales.

Paso de parámetros por referencia :

Pasar parámetros por referencia, significa que lo transferido no son los valores, sino las direcciones de las variables que contienen esos valores, con lo cual los argumentos actuales de la función pueden verse modificados.

Cuando se llama a una función, los argumentos especificados en la llamada son pasados por valor, excepto los arrays (arreglos) que se pasan por referencia.

Ejemplo:

```
        #include <stdio.h>
        /* función sumar*/
        int sumar(int a, int b, int c, int *s)
        {
                b += 2;
                *s = a + b + c;
        }

        main()
        {
                int v = 5, res:
                sumar(4,v,v*2-1,&res) /* llamado a la función */
                printf("%d",suma);
        }
```

Explicación:

*la llamada a la función **sumar**, pasa a esta función los parámetros **4**, **v** y **v*2-1** por valor y el parámetro **res** por referencia. Cualquier cambio que sufra el argumento **s**, sucede también en su correspondiente parámetro actual **res**. En cambio, la variable **v**, no se ve modificada, a pesar de haber variado **b**, ya que ha sido pasada por valor.*

Nótese el uso del operador de referencia '**&**' u operador de dirección para pasar un valor por referencia.

4.4. Reglas básicas de alcance (scope)

La regla básica de alcance consiste en que los identificadores son accesibles sólo dentro del bloque en el que se declararon, fuera de éste son desconocidos, por ejemplo:

```
{
```

```
        int a=5;
        printf("\n%d", a);
        {
                int a=7:

                printf("\n%d", a);
        }
        printf("\n%d", ++a);
}
```

Obsérvese que se ha declarado la misma variable dos veces, pero aunque tenga el mismo nombre son variables distintas y por lo tanto sus valores son distintos, en la segunda declaración de la variable a, ésta se destruye cuando alcanza el fin de bloque de proposiciones, primera llave cerrada (“}”).

4.4.1 Variables Locales y Variables Globales

La regla de alcance es utilizada comúnmente para utilizar variables globales y locales. Las **Variables Globales** se declaran al inicio del programa fuera del main() y fuera de cualquier función, en cambio las **Variables Locales** se declaran dentro e algún bloque. La diferencia sustancial entre estas dos tipos de variables es el alcance: Las variables globales pueden modificar su valor en cualquier parte del programa, mientras que las variables locales solo pueden ser usadas en el bloque donde fueron definidas.

Ejemplo:

```
#include <stdio.h>
int x=20;

void escribe_x()
{
  printf(“ El valor de x (Global) es =  %d\n”,x);        /* x=15 */
}

main()
{
int x=12;
escribe_x();                              /* Escribe el Valor de x = 15 */
printf(“ El valor de x (Local) es =  %d\n”,x);          /* x=12 */
}
```

5

APUNTADORES

Dentro de la memoria de la computadora, cada dato almacenado ocupa una o más celdas contiguas de memoria dependiendo del tipo de dato almacenado es el tamaño que ocupa. Los apuntadores son usados frecuentemente en C para apuntar a la dirección de memoria lo que incrementa el desempeño de los programas y permite la devolución de varios datos de una función, entre otras cosas.

5.1. APUNTADORES

Una variable sencilla de un programa se almacena en un número de bytes (p/e, un entero utiliza dos bytes). El apuntador se usa en programas que acceden a la memoria y manipulan direcciones. Ya se han utilizado apuntadores (p/e, en la lista de argumentos del scanf...)), aunque de una manera sencilla y sin saber exactamente como funcionan.

Sea v una variable, entonces &v es la ubicación o bien la **dirección** en la memoria. El operador de dirección **&** es unario y tiene la misma prioridad y asociatividad (de derecha a izquierda) que los demás operadores unarios.

Para poder usar apuntadores y direcciones de datos se introducen dos nuevos operadores, el primero es el operador apuntador, que se representa con un *, el cual permite definir las variables como apuntadores y también acceder a los datos. El otro nuevo operador, el de dirección (**&**), que permite obtener la dirección en la que se halla ubicada una variable en la memoria y es complementario al operador apuntador, ya antes mencionado.

Para definir un apuntador lo primero que se debe tener en cuenta es que todo apuntador tiene asociado un tipo de dato. Un apuntador se define igual que una variable normal, salvo que delante del identificador se coloca el operador apuntador. Por ejemplo:

```
char *pc; /*apuntador a carácter */

char *pi; /* apuntador a entero */
```

Normalmente al definir un apuntador se inicializa para que apunte a algún dato. Se dispone de tres formas de inicializar un apuntador:

- Inicializarlo con la dirección de una variable que ya existe en memoria.

Para obtener la dirección en la que está ubicada una variable se coloca el operador dirección. Por ejemplo:

```
char *p = &p1;
```

- Asignarle el contenido de otro apuntador que ya está, inicializado:

  ```
  char *p = &p1;

  char *p2 = p; /* p ya está  inicializado */
  ```

- Inicializarlo con cualquier expresión constante que devuelva un *lvalue*.

  Los *lvalue* más frecuentes son una cadena de caracteres, el identificador de un arreglo, el identificador de una función, entre otros. Este es un detalle importante: los identificadores de funciones y de arreglos son en sí mismos valores por la izquierda (*lvalue*), por lo que se pueden usar directamente para inicializar apuntadores.

Una forma adicional de inicializar apuntadores es darle directamente una posición de memoria. Este método no es portable, ya que depende del sistema, pero suele ser muy útil en programación de sistemas, que es uno de los usos fundamentales del C.

El operador **&** sólo se puede aplicar a variables y a elementos de arreglos, de tal forma que si se tiene la siguiente construcciones: &(x+1) y &3 no son permitidas dentro de C.

Por otra parte, un error muy frecuente consiste en no inicializar el apuntador antes de usarlo. Este error frecuentemente lo localiza el compilador y avisa de ello.

Las direcciones que se manejan con los apuntadores son un conjunto de valores que pueden manipularse. Las variables de apuntadores pueden declararse en los programas y después utilizarse para tomar las direcciones como valores.

La declaración:

```
int *p;
```

dice que *p* es de tipo apuntador a un entero. Ejemplos de apuntadores utilizando asignaciones (bajo la declaración anterior):

```
p=0;              /*P apunta a NULL*/
p=NULL;           /* es igual a p=0*/
p=&i;             /*la dirección de i se le asigna a p */
p=(int *) 1501;   /* p apunta a la dirección 1501 de memoria*/
```

Otros ejemplos.

Supongamos la siguiente declaración:

```
double   x, y, *p;  /* aquí se declaran dos variables simples de tipo double y
         una variable que es de tipo apuntador a un double (real) */

p=&x;    /* a p se le pasa la dirección de x  */
y=*p;    /* a u se le pasa el contenido de p */
y=x;     /* a y se le pasa el contenido de x */
              y=*&x;  /* son operadores unarios *y &y se asocian de derecha a izquierda y dice el contenido de la dirección de x
              asignalo a y */
```

Construcciones a las que no se deben de apuntar son las siguientes:

```
&4              /* no se debe apuntar a constantes */

int a[10];      /*se declara un arreglo */
&a              /* como a es un arreglo no se debe de hacer referencia  a la
                dirección de arreglo ya que por si sola la variable a significa una dirección */
```

Para ver el uso de los apuntadores dentro de las funciones y como paso de parámetros, a continuación se muestran dos funciones que hacen lo mismo, es decir intercambian sus valores, si es que el primer parámetro es mayor que el segundo.

```
Ordena (int p, int q)
{
   int tmp;
```

```
   if (p>q){
                    tmp=p;
                    p=q;
                    q=tmp;
            }
   return;
}

Ordena1(int *p, int *q)
{
   int tmp;
   if (*p>*q){
                    tmp=*p;
                    *p=*q;
                    *q=tmp;
                }
   return;
}
```

La diferencia entre la función **Ordena** y la función **Ordena1** es que en la primer al regresar de la función, los valores que se enviaron en las variables *p* y *q*, no son modificadas puesto que se crearon copias de éstas variables, y los cambios realizados dentro de la función no se ve reflejada en quien la llamó. Por otra parte, la segunda función al tener como paso de parámetros las direcciones de las variables que se envían, en el momento que se regresa de la función, el cambio se refleja, puesto que se esta trabajando sobre el mismo espacio en la memoria.

La forma en que pasa sus valores la función **Ordena** se le conoce como paso de parámetros por valor; puesto que se envía a la función los valores de las variables y a su vez ésta recibe los valores en "nuevas" variables, y a **Ordena1** se le conoce como paso de parámetros por referencia; debido a que los que se envía son las direcciones de las variables y se trabaja sobre el mismo espacio, aunque las variables se "llamen" de manera distinta.

La forma de llamar a **Ordena** desde una función es: **Ordena(i,j)** y la forma de llamar a **Odena1** desde un función es: **Ordena(&i, &,j)**

Por último, es importante hacer notar que los apuntadores no son enteros, puesto que generalmente al hacer una asignación de un apuntador a una variable de tipo entero no hay pérdida de información, ni escalado o conversión, pero esto puede ser una práctica peligrosa.

Una aplicación también muy extendida de los apuntadores corresponde a definir una parte de la memoria para trabajar en ella y manejarla mediante un solo objeto, el apuntador a ella. Ésta metodología se llama manejo de Memoria Dinámica.

Ésta técnica permite hacer manipulaciones muy eficientes y flexibles de la memoria, las instrucciones de C que permiten lograr esto son *malloc* y *free*, la primera reserva una cantidad de memoria definida por el usuario y la segunda la libera. Éstas funciones pertenecen a la librería estándar de C.

La sintaxis para asignar la memoria es muy simple, en la zona de declaración de variables se define un apuntador a objetos de cierto tipo (p.e. char *p, define un apuntador a caracteres), luego se reserva la memoria requerida mediante *malloc(N)*, donde *N* indica el número de localidades (definida en bytes) deseadas. Para liberar la memoria luego de haber sido utilizada se utiliza la función *free(p)*.
En el caso que la memoria no se pueda asignar *malloc* regresa típicamente cero. La función *sizeof* regresa el tamaño en bytes de un tipo de variable.

Ejemplo. El siguiente código reserva 75 localidades de memoria de tipo entero, observe el uso de la función *sizeof*, luego se almacenan los primeros 75 números naturales en ella y se enlistan finalmente.

```
main(){
        int *p, t;
        p = malloc(75*sizeof(int));
        if (p==0){
          printf("Memoria insuficiente \n");exit(0);
        }
        for(t=0;t<75;t++) *(p+t) = t+1;
        for(t=0;t<75;t++) printf("%d  ", *(p+t));
        free(p);
}
```

6

ARCHIVOS

Muchas aplicaciones requieren escribir o leer información de un dispositivo de almacenamiento auxiliar, generalmente en un disco magnético, a este tipo de almacenamiento se le llama ***Archivo de datos***, permitiéndonos almacenar información de modo permanente y acceder y alterar la misma cuando sea necesario.

6.1. ARCHIVOS

Un archivo o fichero es una colección de información que almacenamos en un soporte magnético para poder manipularla en cualquier momento. Esta información se puede almacenar y leer de las siguientes maneras:

Función	Descripción
fputc, fgetc	Los datos pueden ser escritos y leídos carácter a carácter
Putw, getw	Los datos pueden ser escritos y leídos palabra a palabra
fputs, fgets	Los datos pueden ser escritos y leídos como cadenas de caracteres
fprintf, fscanf	Los datos pueden ser escritos y leídos con formato
fwrite, fread	Los datos pueden ser escritos y leídos como registros o bloques, es decir como un conjunto de longitud fija, tales como estructuras o elementos de un arreglo

Tabla 11. Formas de acceder a los datos de un archivo

Un archivo visto como un conjunto de caracteres contiene un carácter especial de fin de línea (\n) y un carácter de fin de archivo (EOF).

6.1.1 Funciones para abrir y cerrar archivos

Para poder leer o escribir un archivo primero es necesario abrirlo.

fp = fopen (nombre_archivo, acceso)

Abre un archivo especificado por **nombre_archivo**. El argumento **acceso** especifica el tipo de acceso al archivo ver tabla 12. Esta función básicamente hace dos cosas: abre un archivo en disco para utilizarlo y devuelve un apuntador al fichero **fp**.

acceso	Descripción
“r”	Abrir un archivo para leer. Si el archivo no existe o no se encuentra, se obtiene un error.
“w”	Abrir un fichero para escribir. Si el archivo no existe se crea y si existe su contenido se destruye para ser creado de nuevo.
“a”	Abrir un archivo para añadir información al final del mismo. Si el archivo no existe se crea.
“r+”	Abrir un archivo para leer y escribir. El archivo debe existir.
“w+”	Abriri un archivo para leer y escribir. Si el fichero no existe se crrea y si existe su contenido se destruye para ser creado de nuevo.
“a+”	Abrir un archivo para leer y añadir. Si el archivo no existe se crea.

Tabla 12.Tipo de acceso a un archivo.

Ejemplo: abrir un archivo para lectura con nombre Pre.tex

```
FILE *fp ; /*Declara una variable de apuntador a un archivo, fp */
fp = fopen( “Pre.tex” ,”r”);
```

Sin embargo es más conveniente verificar si existe el archivo, en caso de que no exista, termina el programa.

```
if((fp=fopen("Pre.tex","r")) == NULL)
{   printf("No se puede abrir el archivo \n");
    exit(o);
}
```

Es necesario cerrar el archivo al dejar de utilizarlo, esto se realiza de la siguiente forma:

fclose(fp);

esta función cierra el archivo apuntado por **fp**.

6.1.2 Entrada carácter a carácter: fputc y fgetc

fputc(car,fp);

```
int car;
FILE *fp ;
```

Esta función escribe un carácter **car**, en el archivo apuntado por **fp**, en la posición indicada por el apuntador de lectura/escritura.

Ejemplo:

```
#incluide <stdio.h>
#include <string.h>   /* bilioteca de cadenzas */

FILE fp;          /* declaración de un apuntador a archivo */
char buffer[81];  /* arreglo de caracteres  */
int i=0, longitud;

main()
{
   fp = fopen (“texto.txt”, “w”);      /* abrir archivo para escritura */
          /* copia el letrereo  en el buffer */
   strcpy(buffer,”Este mensaje es un texto para fputc! \n”);
   longitud = strlen(buffer);  /* tamaño de la cadena */
   /*  escribir el arreglo en el archivo, la función ferror checa si se ocurre un
error al escribir */
   while (¡ferror(fp) && i < longitud)
          fputc(buffer[i++], fp);
   if (ferror( fp))
          printf(“ \n Error durante la escritura \n”);
   fclose(fp);       /* cerrar el archivo */
}
```

fgetc(fp);

FILE * fp;

La función fgetc devuelve el carácter leido o un EOF si se ocurre un error o se detecta el final del archivo. Se utiliza la función feof para distinguir si se ha detectado el final del archivo.

Ejemplo:

```
#include <stdio.h>
FILE *fp1;          /* declaración de un apuntador a un archivo */

main()
{
   char car;
   /* se verifica si se puede abrir el archivo para lectura */
    if((fp=fopen("texto.txt","r")) == NULL)
            {    printf("No se puede abrir el archivo \n");
                 exit(o);
            }
    car = getc(fp1);    /* lectura del primer caracter del archivo */
    while (car != eof(fp1))
   {
            putc(car);     /* escribe el carcater en pantalla */
            car = getc(fp1);     /* lectura del siguiente carácter del archivo */
   }
   fclose (fp1);
}
```

6.1.3 Entrada/Salida con formato: fprintf y fscanf

Las funciones fprintf y fscanf sirven para hacer escritura y lectura con formato de un archivo. Las descripción del formato es el mismo que se especifican para printf() y scanf();

fprintf(fp, cadena_de_control, lista_de_arcgumentos);

Esta función escribe la **lista_de_argumentos**, con el formato especificado en la **cadena_de_control**, en el archivo apuntado por **fp**.

Ejemplo: suma de dos números escribiendo los resultados en un archivo de tipo añadir.

```
#include <stdio.h>
main()
{
   FILE *fp; / declaración de apuntador a archivo */
   int a,b,s;
   fopen("salida.dat","a");       /* abrir archivo para añadir */
   printf(" Da el valor de a : ");
   scanf("%d", &a);
   printf(" Da el valor de b : ");
   scanf("%d", &b);
   s = a+b ;
   printf("\n La suma de %d + %d = %d\n",a,b,s);
   /* escritura de los datos al archivo */
  fprintf(fp,"El valor de a = %d\n",a);
  fprintf(fp,"El valor de b = %d\n",b);
  fprintf(fp,"La suma de %d + %d = %d\n",a,b,s);
  fprintf(fp,"_____\n");
  fclose(fp);
}
```

fscant(fp, cadena_de_control, lista_de_arcgumentos);

Esta función lee la **lista_de_argumentos**, con el formato especificado en la **cadena_de_control**, desde el archivo apuntado por **fp.**

Ejemplo: Lectura de un archivo de números reales (tabla)

```
#include <stdio.h>
main()
{
   FILE *fp; / declaración de apuntador a archivo */
```

```
   float a,b,c,d,s;
     /* se verifica si se puede abrir el archivo para lectura */
   if((fp=fopen("texto.txt","r")) == NULL)
          {   printf("No se puede abrir el archivo \n");
              exit(o);
          }

          /* Lectura del archivo */
   while(!feof(fp))
   {
          /* lectura de un renglon  del archivo de la tabla de 4 columnas*/
          fscanf(fp,"%f  %f  %f  %f",&a,&b,&c,&d);
          s = (a+b+c+d)/4.0;
          printf(" El promedio es %f\n",s);
   }
fclose(fp);
}
```

6.1.4. Argumentos argc y argv de líneas de comando

Los parámetros **argc** y **argv** sirven para trasmitir argumentos al programa desde la línea de comandos cuando se inicia la ejecución de un programa.

argc:

Es de tipo int, y guarda el número de argumentos que se dieron desde la línea de comandos, y se declara de la siguiente manera:

```
int argc;
```

argv:

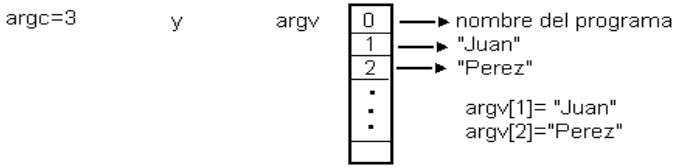
Es un arreglo de apuntadores a una cadena de caracteres, donde la posición 0 apunta a la cadena que tiene el nombre del programa en ejecución, y las siguientes posiciones apuntan a cada uno de los argumentos, y se declara de la siguiente manera:

```
char * argv[];                /* No tiene dimensión */
```

Ejemplo: Ejecución de un programa desde la línea de comandos en MS-DOS

```
c:>prog Juan Perez        /* El programa fue salvado con el nombre "prog"
                             y los argumentos son "Juan" y " Perez" */
```

La siguiente figura describe exactamente a las variables argc y a argv.




```
void main(int argc,char *argv[])
{
  if (argc!=3) {
                printf("Número de argumentos no valido \n");
                exit(0);
              }
   else   {
            printf("El nombre es %s\n",argv[1]);
            printf("El apellido es %s\n",argv[2]);
          }
    exit(0);
  }
```