CAPITULO I

INTRODUCCION AL LENGUAJE C

Antecedentes Históricos del C.

El lenguaje C fue inventado e implementado por primera vez en 1970 en AT&T por Dennis Ritchie en un DEC PDP-11 usando Unix como Sistema Operativo. C es el resultado de un proceso de desarrollo comenzando con un lenguaje anteriormente denominado BCPL, que aún hay quien lo usa principalmente en Europa.

¿Por qué el nombre de C? Por la influencia que tuvo del lenguaje B, BCPL fue desarrollado por Martin Richards e influenciado por B inventado en Bell Labs. por Ken Thompson. En los años 70´s el lenguaje B llevó al desarrollo de C.

La versión original de C fue popularizada por el libro clásico "El Lenguaje de Programación C" por Brian Kernigan y Dennis Ritchie. Durante muchos años el estándar de C fue realmente la versión proporcionada con la versión 5 del Sistema Operativo Unix. Con la popularidad de las microcomputadoras se crearon muchas implementaciones de C, pero en realidad, como no existía ningún estándar aparecieron muchas discrepancias. Para remediar la situación, el Instituto de Estándares Americano (ANSI) estableció un comité a principios del verano de 1983 para crear el estándar que definiera de una vez por todas al lenguaje C. La definición resultante, el estándar ANSI o " ANSI C " que se esperaba fuera aprobada a fines de 1988.

La mayoría de las características del estándar ya se encuentran soportadas por compiladores modernos.

Uno de los propósitos del estándar fue asegurar que la mayoría de los programas existentes pudiesen permanecer válidos o, al menos, que las computadoras pudieran producir mensajes de advertencia acerca de nuevos comportamientos.

Una segunda contribución significativa del estándar es la definición de una biblioteca que acompaña a C, la cual especifica funciones para:

- Tener acceso al Sistema Operativo (por ejemplo: leer archivos, escribir en ellos, etc.).
- E/S con formato
- Asignación de Memoria
- Manipulación de Cadenas y otras actividades semejantes,

Y una colección de headers estándar que proporcionan un acceso uniforme a las declaraciones de funciones y tipos de datos.

La mayor parte de las bibliotecas está estrechamente modelada con base en la biblioteca estándar del sistema Unix.

BCPL y B son lenguajes "carentes de tipos". En contraste, C proporciona una variedad de tipos de datos. Los tipos de datos fundamentales son caracteres, enteros y números de punto flotante de varios tamaños. Además existe una jerarquía de tipo de datos derivados, creados con apuntadores, arreglos, estructuras y uniones. Las expresiones se forman a partir de operadores y operandos, cualquier expresión, incluyendo una asignación o una llamada a función, puede ser una proposición. Los apuntadores proporcionan una aritmética de direcciones independiente de la máquina.

C proporciona las construcciones fundamentales de control de flujo que se requieren en programas bien estructurados: Agrupación de proposiciones, toma de decisiones (if-else), selección de un caso entre un conjunto de ellos (switch), iteración con la condición de parar en la parte superior (while, for) o en la parte inferior (do) y terminación prematura de ciclos (break).

Dentro de los cambios de C, hay uno de relevancia, en New Jersey en 1980, Bjarne Stroustrop desarrolló en Bell Labs. C++ o llamado también "C con clases", por lo que en 1983 se cambió el nombre por el de C++. Desde entonces ha experimentado 2 revisiones de importancia, una en 1985 y otra en 1989.

El propósito de C++ es extender a C para proporcionar ocultamiento de información y un estilo de programación que haga énfasis en las Clases de Objetos.

La compatibilidad con C, la eficiencia y la verificación estricta en tiempo de compilación fueron las metas principales para el diseño de C++:

- La compatibilidad con C permite que el código fuente existente en C continúe utilizándose. La mayoría de las implementaciones de C++ son incluso compatibles con C.

- Ya se destacó la eficiencia, por la cual no debe de haber ningún problema al usar C++ en lugar de C.

La similitud entre C y C++ es sin embargo engañosa, debido a que C++ requiere un método de programación nuevo, un nuevo paradigma llamado "Orientado a Objetos" que da nuevos beneficios con el uso de las clases y los objetos.

Como podrá comprobar, una de las razones que motivaron al desarrollo de C++ fue la de permitirle al programador manejar programas de una complejidad cada vez mas creciente.

Es un hecho que C++ es un superconjunto de C; por lo que la mayoría de los programas de C son también implícitamente programas de C++, sin embargo, hay una cuantas diferencias mínimas entre ANSI C y C++ que impiden que pocos programas de C se puedan compilar mediante un compilador C++ o viceversa:

- En C una constante de carácter se eleva de forma automática a un entero. En C++ esto no sucede.

- En C no es un error declarar una variable global varias veces (aunque esto sea una mala práctica). En C++ esto es un error.

- En C un identificador puede tener hasta 31 caracteres de longitud. En C++ no hay tal límite. No obstante que desde un punto de vista práctico, identificadores extremadamente largos son difíciles de manejar.

- El uso en C++ de la cabecera IOSTREAM.H y de todas sus funciones que en C no se manejan.

- El manejo en C++ de todas sus características de Clases y Objetos tal como los constructores, destructores, funciones amigas, sobrecarga de funciones, referencias, herencia, poliformismo, encapsulación, abstracciones etc. que en C no se manejan.

- El uso en C++ de la cabecera COMPLEX.H para el manejo de números complejos.

- El uso en C++ de la clase BCD. Otra manera de representar un numero real es la de usar la técnica de Decimal Codificado en Binario o BCD. Su principal ventaja es que no se producen errores de redondeo, esto es mediante la cabecera BCD.H

- Las extensiones de los archivos son para ANSI C ***.c** y para C++ ***.cpp.**

Conceptos Generales de C

De C a C++.

C++ es un súper conjunto de C. Todas las construcciones de C están presentes en C++. Los programas que se compilan bajo C deberían poder hacerlo bajo C++. Los programas que siguen el estándar de Kernighan y Ritchie pueden dar origen a errores cuando se someten a un compilador de C++, y siempre dan origen a avisos. Los programas que no generan avisos bajo Turbo C 2.0 tampoco deberían generar errores al ser compilados bajo turbo C++.

Para confirmar esto, Turbo C++ distingue entre un programa C y un programa C++ examinando su extensión. Si el nombre del archivo fuente termina en **.c** se considera un programa de C; si termina en **.cpp**, se entiende que es un programa C++, a no ser que la opción C++ haya sido activada en el menú Options/Compiler.

C++ y ANSI C.

Comparten algo más que una herencia en común. Muchas de las extensiones de C ANSI, tales como el formato de la declaración de sus funciones y el uso de los tipos de fuentes de datos, se trasladan a C++. Además de esto, C++ se ha ampliado con objeto de incluir las últimas novedades ANSI y guardar la máxima compatibilidad. C++ comparte las siguientes características con ANSI, aunque exista alguna diferencia menor:

- Prototipo de Funciones.
- Variables register
- #pragmas

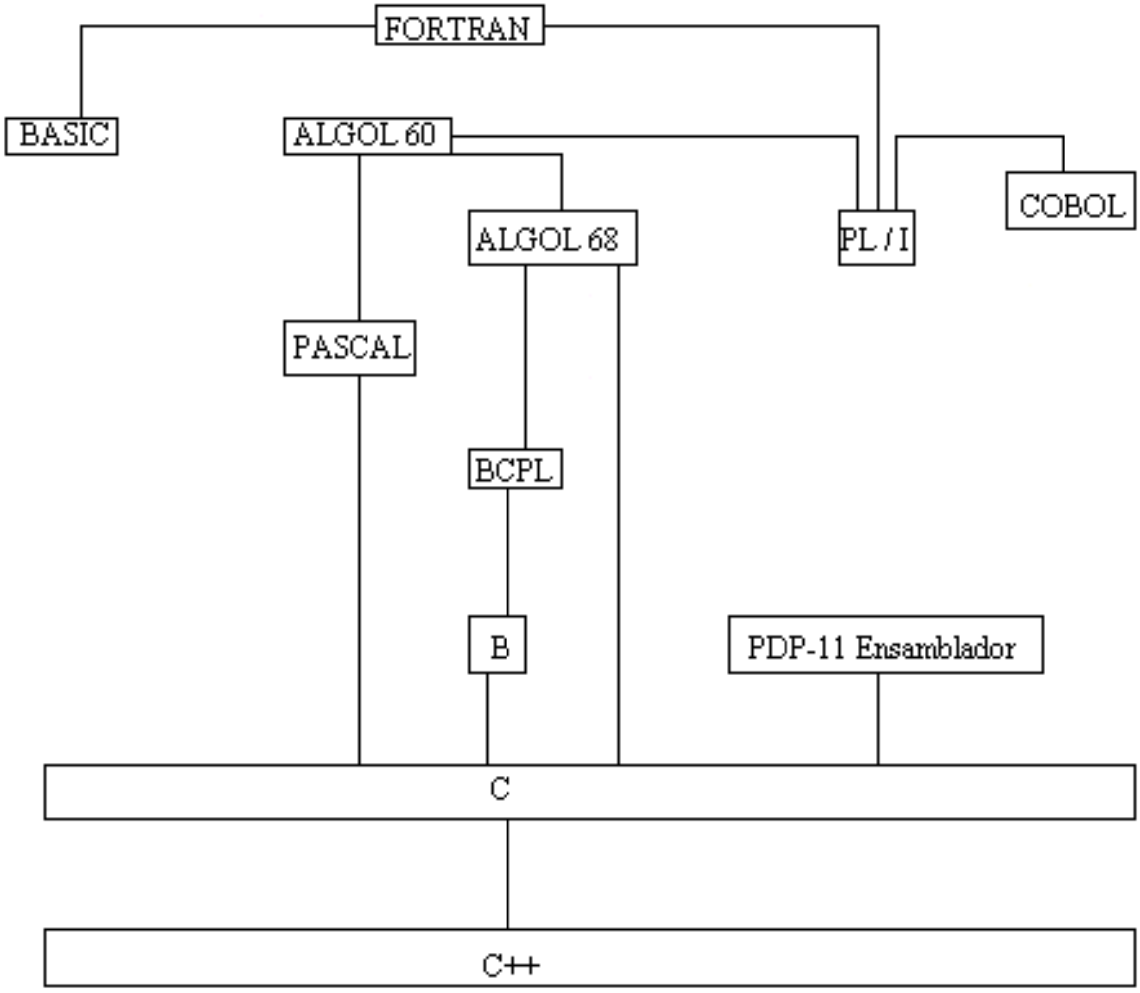
De Turbo C a Turbo C++.

Aunque Turbo C++ se desvía de Turbo C, la interrelación entre ambos es inconfundible. Turbo C++ conserva tanto la versión del compilador Orientado a la línea de órdenes, como el Entorno Interactivo de Desarrollo (IDE), pero en un entorno de ventanas. Además, Turbo C++ admite ahora la entrada de información desde un ratón o desde el teclado.

Las 3 mejoras significativas de Turbo C++ son:

- El Entorno Interactivo de Desarrollo que permite al programador editar, compilar, enlazar y depurar en el mismo entorno sin necesidad de conmutar del editor al compilador y al depurador. El programador puede ajustar la compilación y el enlazado a través de opciones del menú, en vez de memorizar las opciones de compilación de las líneas de órdenes.
- El Gestor de Memoria VROOMM (Virtual Runtime Object Oriented Memory Manager), que es un sofisticado gestor de superposición de memoria que permite a los programadores crear programas de tamaño superior a los 640 Kb.
- Soporte del lenguaje Ensamblador, al admitir pseudo registros y tipo interrupciones.

Antecesores de C y C++



¿Por qué usar C o C++?

- Es un lenguaje de alto nivel.
- Tienen una escritura elegante, es decir, programas legibles y entendibles.
- Tiene programas portables a otros sistemas.
- Excelentes compiladores disponibles.
- La compilación es rápida y pequeña.
- Sus características permiten acceder al control del hardware.

- Durante la programación se pueden considerar detalles de bajo nivel.

Específicamente el compilador de Borland C++

- Una magnifica interfase con el usuario.
- Se pueden configurar la mayoría de sus características para fomentar hábitos.
- Los errores de sintaxis son inmediatamente detectados con el editor.
- El editor emula a los populares editores de texto.
- Desde el editor tú tienes el control del depurador.
- Borland C++ incluye características estándar para facilitar el desarrollo de grandes aplicaciones de Software.
- Borland C++ corre bajo el ambiente Windows de cualquier versión (inferiores a la 3.0), y contiene un Kit de herramientas para el desarrollo de aplicaciones de Windows (cabecera windows.h)

¿CUALES SON LAS CARACTERISTICAS DE TURBO C++ 3.0 PARA DOS?

De manera interna Turbo C++ contiene una pequeña guía de aprendizaje para guiar al programador en su avance cotidiano de trabajo, con una ayuda sensible al contexto, y mantiene un gran poder en su editor, compilador, depurador y ensamblador, todo para un desarrollo de aplicaciones de C y C++ fácilmente. Además cuenta con las siguientes ventajas:

- contiene capacidades como ensamblador
- programación Orientada a Objetos
- soporte para Templates, Librerías de Clases Standares incluyendo iostreams, y números complejos
- cabeceras precompiladas para una compilación rápida
- un administrador inteligente de Proyectos con una utilería MAKE
- más de 450 librerías de funciones incluyendo las gráficas
- incluye editor multi-archivos con soporte para macros y archivos muy grandes
- ilimitado uso de un-do y re-do
- una sintaxis sobresaltada por colores.

C Pertenece a una familia bien establecida de lenguajes cuya tradición enfatiza virtudes como fiabilidad, regularidad, simplicidad de uso. Los miembros de esta familia se llaman a menudo "Lenguajes Estructurados", cuya disciplina es hacer programas fáciles de leer y escribir.

C++ ya no entra dentro de esta familia de lenguajes Estructurados, si no en el último de los paradigmas establecidos en los 80´s y 90´s: El Orientado a Objetos.

¿QUE REQUERIMIENTOS DE SISTEMA NECESITO?

Una PC 286 o superior, MS-DOS 3.31 o posterior, 1 MB en RAM, un disco duro con un mínimo de espacio libre de 5 Mb.

Forma general de un programa en C

Un programa en C está formado por una ó más funciones que a su vez, contienen una serie de elementos propios. Una de las funciones que debe estar **obligatoriamente** presente en el programa es la ***función main***.

OBSERVACION

Una función es un conjunto de instrucciones que realizan una tarea específica en el programa.

Así mismo el programa tendrá que contener una serie de directivas de preprocesador ó archivos de cabecera escritos como **#include<*.h>** que son archivos de librerías con extensión *.h que contienen la declaración de variables y funciones que permiten el funcionamiento y compilación de los programas.

La forma general de un programa en C es:

```
/* Descripción de lo que hace el programa
   incluir comentarios sobre lo que hace el programa*/

/* Directivas del preprocesador */

# include <nombre1.h>
# include <nombre2.h>
# define NOMB_CONST   valor
```

```
/* Declaración de variables GLOBALES */

tipo_var nombre_variable1, nombre_variable2 ;
tipo_var nombre_variable3 ;

/* Declaración de funciones del usuario */

int Nombre_Func1 (tipo_var nombre_variable4, tipo_var nombre_variable5) ;
void Nombre_Func2 (tipo_var nombre_variable 6) ;
float Nombre_Func3 (tipo_var nombre_variable7, tipo_var nombre_variable8);
```

```
int Nombre_Func1 (tipo_var nombre_variable4, tipo_var nombre_variable5)
{
  /* declaración de variables LOCALES de la función */
  int nom_var_dev ;
  sentencias ;
  return (nom_var_dev) ;
}
```

```
void  Nombre_Func2 (tipo_var nombre_variable6)
{
  /* declaración de variables LOCALES de la función */
  sentencias ;
}
```

```
float Nombre_Func3 (tipo_var nombre_variable7, tipo_var nombre_variable8)
{
  /* declaración de variables LOCALES de la función */
  float nom_var_dev ;
  sentencias ;
  return (nom_var_dev) ;
}
```

```
int main(void) /* FUNCION PRINCIPAL*/
{
    /* declaración de variables LOCALES de la función */

     int nombre_variable9 ;
      float nombre_variable10 ;

     sentencias ;

    /* llamadas a las funciones */

     nombre_variable9 = Nombre_Func1( nomb_var4, nomb_var5 ) ;
     Nombre_Func2( nomb_var6 ) ;
     nombre_variable10 = Nombre_Func3( nomb_var7, nomb_var8 ) ;

     sentencias ;

      return 0 ;
}
```

Ejemplo

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

int main(void)
{
      clrscr();
      printf("Bienvenido a la programacion en C");
      getch();

      return 0;
}
```

Obsérvese que cada sentencia de programa queda finalizada por el terminador " ; ", 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.

Es perfectamente lícito escribir cualquier sentencia abarcando los renglones que la misma necesite, por ejemplo podría ser:

```
printf(" Esto es un ejemplo de sentencia que abarca más de"
"un renglón");
```

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 especificado en el nombre que sigue a la misma (<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 (por ejemplo printf()) necesarias para que el compilador las procese.

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 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. Nótese que 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 "**;**".

Inclusión de ficheros

En la programación en C es posible utilizar funciones que no estén incluidas en el propio programa. Para ello utilizamos la directiva **#include**, que nos permite añadir librerías o funciones que se encuentran en otros ficheros a nuestro programa.

Para indicar al compilador que vamos a incluir ficheros externos podemos hacerlo de dos maneras (siempre antes de las declaraciones).

1. Indicándole al compilador la ruta donde se encuentra el fichero.

```
#include "misfunc.h"
#include "c:\includes\misfunc.h"
```

2. Indicando que se encuentran en el directorio por defecto del compilador.

```
#include <misfunc.h>
```

Ejemplos

1. #include <stdio.h>
2. #include <math.h>

3. #include "a:\lineas.h"

Comentarios

A la hora de programar es conveniente añadir comentarios (cuantos más mejor) para poder saber que función tiene cada parte del código, en caso de que no lo utilicemos durante algún tiempo. Además facilitaremos el trabajo a otros programadores que deseen utilizar nuestro archivo fuente.

Para poner comentarios en un programa escrito en **C** usamos los símbolos ***/ *** y **** /***

Ejemplos

1. /* Este es un ejemplo de comentario */

2. /* Un comentario también puede
   estar escrito en varias líneas */

El símbolo /* se coloca al principio del comentario y el símbolo */ al final. El comentario, contenido entre estos dos símbolos, no será tenido en cuenta por el compilador.

Palabras clave

Existen una serie de indicadores reservados, con una finalidad determinada, que no podemos utilizar como identificadores.

A continuación vemos algunas de estas palabras clave:

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

Identificadores

Un identificador es el nombre que damos a las variables y funciones. Está formado por una secuencia de letras y dígitos, aunque también acepta el caracter de subrayado _. Por contra no acepta los acentos ni la **ñ**/**Ñ**.

El primer caracter de un identificador **NO** puede ser un número, es decir que debe ser una letra o el símbolo _.

Se diferencian las mayúsculas de las minúsculas, así **num**, **Num** y **nuM** son distintos identificadores.

La práctica tradicional de C es usar letras minúsculas para nombre de variables y mayúsculas para las constantes simbólicas. Al menos los primeros 31 caracteres de un nombre son significativos, para nombre de funciones el número puede ser menor de 31.

Las palabras reservadas del lenguaje no se pueden utilizar como variables.

A continuación vemos algunos ejemplos de identificadores válidos y no válidos:

Válidos	No válidos	Observaciones
_num	**1**num	Los identificadores siempre deben empezar con una letra o el caracter _
var1	n**ú**mero2	Un identificador no puede tener acentos
Fecha_Nac	a**ñ**o_nac	Un identificador no puede tener ñ o Ñ

Tipos de Datos en C

En 'C' existen básicamente cuatro tipos de datos, aunque como se verá después, podremos definir nuestros propios tipos de datos a partir de estos cuatro. A continuación se detalla su nombre, el tamaño que ocupa en memoria y el rango de sus posibles valores.

TIPO	TAMAÑO	RANGO DE VALORES
char	1 byte	-128 a 127
int	2 bytes	-32768 a 32767
float	4 bytes	3.4 E-38 a 3.4 E+38
double	8 bytes	1.7 E-308 a 1.7 E+308

Calificadores de tipo

Los calificadores de tipo tienen la misión de modificar el rango de valores de un determinado tipo de variable. Estos calificadores son cuatro:

- **signed**

  Le indica a la variable que va a llevar signo. Es el utilizado por defecto.

- **unsigned**

  Le indica a la variable que no va a llevar signo (valor absoluto).

- **short**

  Rango de valores en formato corto (limitado). Es el utilizado por defecto.

- **long**

  Rango de valores en formato largo (ampliado).

TIPO	TAMAÑO	RANGO DE VALORES
char	**1 byte**	**- 128 a 127**
signed char	1 byte	- 128 a 127
unsigned char	1 byte	0 a 255
int	**2 bytes**	**- 32768 a 32767**
unsigned int	2 bytes	0 a 65535
long int	4 bytes	- 2147483648 a 2147483647
unsigned long int	4 bytes	0 a 4.294.967.295 (El mayor entero permitido en 'C')
float	**4 bytes**	**3.4 E- 38 a 3.4 E+38**
double	**8 bytes**	**1.7 E- 308 a 1.7 E+308**
long double	10 bytes	3.4 E -4932 a 1.1 E +4932
void	**0 bytes**	**sin valor**

Ejemplos

a) **short int** n;
b) **long int** contar;

El calificador signed o unsigned puede aplicarse **sólo** a char o a enteros. Los números unsigned son mayores o iguales que cero, por ejemplo la declaración:

- **unsigned int** numero;

  es igual a

- **unsigned** numero;

El tipo **long double** especifica punto flotante de precisión extendida, igual que los enteros, los tamaños de objetos de punto flotante se definen en la implementación, float, double y long double representan 3 tamaños distintos.

Los archivos de cabecera estándar **<limits.h>** y **<float.h>** contienen constantes similares para todos esos tamaños.

Variables en C

Una variable es un tipo de dato, referenciado mediante un identificador (que es el nombre de la variable). Su contenido podrá ser modificado a lo largo del programa. Una variable **sólo** puede pertenecer a un tipo de dato.

Para poder utilizar una variable, primero tiene que ser declarada de la siguiente forma:

```
[calificador] <tipo> <nombre>
```

donde:

calificador : puede ser unsigned, signed, long, short o posibles combinaciones vistas anteriormente

tipo : puede ser char, int, float o double

nombre : es el nombre que se le da a la variable

Ejemplos

a) **int** num, suma **;**
b) **float** prom, division, peso **;**
c) **long int** fono;
d) **short int** dim;

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

```
[calificador] <tipo> <nombre1>, <nombre2> = <valor>,
        <nombre3> = <valor>, <nombre4>
```

La declaración de una variable especifica que debe reservarse memoria para el objeto del tipo especificado y que podemos referirnos a ese objeto por medio del identificador

de la variable, además de que especifica cuanta memoria debe apartarse para esos objetos, también implica el como han de interpretarse los datos representados por cadenas de bits en una localidad de memoria.

Ejemplo

```
/* Uso de las variables */

#include <stdio.h>

void main(void) /* Suma dos valores */
{
      int num1=4, num2, num3=6;

      printf("El valor de num1 es %d",num1);
      printf("\nEl valor de num3 es %d",num3);
      num2=num1+num3;
      printf("\nnum1 + num3 = %d",num2);

      getch();

}
```

Constantes en C

Al contrario que las variables, las constantes mantienen su valor a lo largo de todo el programa.

Para indicar al compilador que se trata de una constante, usaremos la directiva **#define** ó la palabra reservada **const**.

1ra. Forma: Uso de la directiva del preprocesador #define:

#define <identificador> <valor>

Observe que no se indica el punto y coma de final de sentencia ni tampoco el tipo de dato. La directiva **#define** no sólo nos permite sustituir un nombre por un valor numérico, sino también por una cadena de caracteres.

2da. Forma: Uso de la palabra reservada const:

const tipo_de_dato identificador = <valor>;

En C hay diferentes tipos de constantes, las cuales pueden ser declaradas de la siguiente forma:

- **Constantes Enteras.**

  #define IVA 15 ó **const int** IVA=15.

  El valor de un entero también se puede hacer constante con un valor octal o hexadecimal, un 0 al principio de una constante entera refiere a un octal y un 0x refiere a un hexadecimal.

  #define IVA 017 ó **#define** IVA 0xF

- **Constante Reales.**

  #define PI 3.14159265 ó **const float** PI=3.14159265

- **Constante Carácter.**

  Es un entero escrito como un carácter dentro de apostrofes. El valor de una constante de carácter es el valor numérico del carácter en el código ASCII.

  #define SALIR 's' ó **const char** SALIR='s'.

- **Constante Cadena ó Cadena Literal.**

  Secuencia de cero o más caracteres encerrados entre comillas como en: "Hola mundo". Las comillas no son parte de la cadena, sólo sirven para delimitarla.

  #define MENSAJE "Presione cualquier tecla para continuar…."

  const char *MENSAJE = "Presione cualquier tecla para continuar…."

Ejemplo

```
/* Uso de las constantes */

#include <stdio.h>
#include <conio.h>
#define pi 3.1416
#define escribe printf /*reemplazo del nombre de la funcion printf por escribe*/

void main(void)
/* Calcula el perímetro */
{
      int r;

      clrscr();
      escribe("Introduce el radio: ");
      scanf("%d",&r);
      escribe("El perimetro es: %f",2*pi*r);
      getch();
}
```

Operadores en C

Los operadores son símbolos que permiten al programa llevar acabo funciones de aritmética, asignación, lógicas y relacionales.

Operadores aritméticos

Existen dos tipos de operadores aritméticos:

- operadores binarios
- operadores unarios

Binarios		Unarios	
+	Suma	+ +	Incremento (suma 1)
-	Resta	- -	Decremento (resta 1)
*	Multiplicación	-	Cambio de signo
/	División		
%	Módulo (resto)		

Sintaxis	Sintaxis
<variable1> <operador> <variable2>	<variable> <operador> <operador> <variable>

Ejemplo

```
/* Uso de los operadores aritméticos */

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

void main(void)
/* Realiza varias operaciones utilizando operadores binarios y unarios  */
{
      int a=1, b=2, c=3, r;

      clrscr();

      r = a+b; /* operador binario */
      printf("%d + %d = %d\n",a,b,r);
      r = c-a; /* operador binario */
      printf("%d - %d = %d\n",c,a,r);
      b++; /* operador unario */
      printf("b + 1 = %d",b);

      getch();
}
```

Operadores de asignación

La mayoría de los operadores aritméticos binarios explicados anteriormente tienen su correspondiente operador de asignación:

=	Asignación simple
+ =	Suma
- =	Resta
*** =**	Multiplicación
/ =	División
% =	Módulo (resto)

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

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

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

Ejemplo

```
/* Uso de los operadores de asignación */

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

void main(void) /* Realiza varias operaciones */
{
     int a = 1, b = 2, c = 3, r;

     clrscr();

     a+= 5;
     printf("a + 5 = %d\n",a);
     c-=1;
     printf("c - 1 = %d\n",c);
     b*=3;
     printf("b * 3 = %d",b);

     getch();
}
```

Prioridad de los operadores

En C también existe una prioridad dentro de estos operadores, téngalos en cuenta para el momento de efectuar operaciones en los programas:

Operador	Prioridad
++	1
--	1
*	2
/	2

%	2
+	3
-	3
=	4

Operadores Relacionales.

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

En C existen seis operadores relacionales básicos:

Operador	Acción
<	Menor que
>	Mayor que
<=	Menor o igual que
>=	Mayor o igual que
==	Idéntico ó igual a
!=	Diferente de

El resultado que devuelven estos operadores es **1** para Verdadero y **0** para Falso.

Si hay más de un operador se evalúan de izquierda a derecha. Además los operadores **==** y **!=** están por debajo del resto en cuanto al orden de precedencia

Ejemplos

```
/* Uso de los operadores relacionales.  */

#include <stdio.h>

void main(void) /* Compara dos números entre ellos */
{
  int a, b;

  printf("Introduce el valor de A: ");
```

```
  scanf("%d",&a);
  printf("Introduce el valor de B: ");
  scanf("%d",&b);

  if(a>b)
      printf("A es mayor que B");
  else
      if(a<b)
          printf("B es mayor que A");
      else
          printf("A y B son iguales");
}
```

Operadores Lógicos.

Los operadores lógicos básicos son tres:

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

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

p	q	!p	p&&q	p||q
V	V	F	V	V
V	F	F	F	V
F	V	V	F	V
F	F	V	F	F

Ejemplo

```
/* Uso de los operadores lógicos AND,OR,NOT. */

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

void main(void)

/* Verifica si un número introducido está entre un cierto intervalo de números */
{
  int numero;
```

```
  clrscr();
  printf("Introduce un número: ");
  scanf("%d",&numero);

 if( !(numero >= 0))
    printf("El número es negativo");
 else
      if((numero<=100) && (numero>=25))
         printf("El número está entre 25 y 100");
      else
         if((numero<25) || (numero>100))
            printf("El número no está entre 25 y 100");
}
```

Operador ?

Puede usarse para sustituir ciertas sentencias de la forma if-then-else.

El operador ternario **?** toma la forma:

```
Exp1 ? Exp2 : Exp3
```

Donde Exp1, Exp2 y Exp3 son expresiones y se interpreta así:

Si Exp1 es verdad
Entonces
    Exp2
Caso contrario
    Exp3

Ejemplos

a) x = 10;
   y = x > 9 ? 100 : 200;

ó lo que es lo mismo

```
x=10;
if (x > 9)
  y = 100;
else
  y = 200;
```

Operadores de puntero & y *

Un puntero es la dirección en memoria de una variable. Una variable puntero es una variable declarada específicamente para contener un puntero hacia un valor de su tipo específico. El conocer la dirección de una variable puede ser de gran ayuda en ciertos casos. En C los punteros tienen dos funciones principales:

1. Proporcionan una forma rápida de referenciar los elementos de un array.
2. Permiten a las funciones de C modificar los parámetros de llamada.

El operador **&** es un operador monario que devuelve la dirección de memoria del operando. Por ejemplo:

```
m = &cont;
```

coloca en m la dirección de memoria de la variable cont.

El segundo operador ***** es el complementario de **&**. Es un operador monario que devuelve que devuelve el valor de la variable ubicada en la dirección que se especifica. Por ejemplo si m contiene la dirección de memoria de la variable cont entonces:

```
q = *m;
```

colocará el valor de cont en q.

Ejemplos

a)
```
char *pc;   /* declara a pc como puntero a un carácter */
```

b)
```
int *pi;      /* declara a pi  como puntero a un entero */
```

c)
```
char *p = &p1; /* coloca en p la dirección de la variable p1 */
```

d)
```
char *p = &p1;
char *p2 = p; /* ahora p y p2 contienen la dirección de la variable p1 */
```

e) si el valor de la variable cont es 100 y su ubicación en memoria es 2000, entonces
m = **&**cont; /* coloca 2000 en m */
q = *****cont; /* coloca 100 en q */

f) int x, *****y, cont; /* declara a x como entero, a y como puntero a entero, a cont como entero */

Conversión automática de tipos

Cuando dos ó mas tipos de variables distintas se encuentran DENTRO de una misma operación ó expresión matemática, ocurre una conversión automática del tipo de las variables. En todo momento al realizarse una operación se aplica la siguiente secuencia de reglas de conversión (previamente a la realización de dicha operación):

1. Las variables del tipo char ó short se convierten en int.
2. Las variables del tipo float se convierten en double.
3. Si alguno de los operandos es de mayor precisión que los demás, éstos se convierten al tipo de aquel y el resultado es del mismo tipo.
4. Si no se aplica la regla anterior y un operando es del tipo unsigned el otro se convierte en unsigned y el resultado es de este tipo.

Las reglas 1 a 3 no presentan problemas, sólo nos dicen que previamente a realizar alguna operación las variables son promovidas a su instancia superior. Esto no implica que se haya cambiado la cantidad de memoria que las aloja en forma permanente.

Otro tipo de regla se aplica para la conversión en las asignaciones. Por ejemplo, el pasaje de float a int provoca el truncamiento de la parte fraccionaria, en cambio de double a float se hace por redondeo.

Funciones de Entrada/Salida

Función printf( )

La rutina printf permite la aparición de valores numéricos, caracteres y cadenas de texto por pantalla. Esta función está definida por el estándar ANSI y requiere del archivo de cabecera **stdio.h**, para poder ejecutarse. La sintaxis de esta función es la siguiente:

printf ***(*** *formato, argumentos* **);**

La cadena apuntada por formato se compone de dos tipos de elementos. En el primer tipo se encuentran los caracteres que serán desplegados en pantalla. El segundo tipo contiene las órdenes de formato que indican la forma en que serán desplegados los argumentos. Una orden de formato comienza con un signo de % y le sigue el código

del formato. Debe existir el mismo número de argumentos que de órdenes de formato. De este modo, los argumentos y las órdenes se asocian en el orden expuesto. Si no hay suficientes argumentos que asociar a las órdenes de formato, la salida queda indeterminada. Si hay más argumentos que órdenes de formato, se descarta el resto de los argumentos.

En la siguiente tabla se muestran las órdenes de formato más usadas.

TABLA DE FORMATO

Código	Formato
%c	Carácter único.
%d	Enteros decimales con signo.
%i	Enteros decimales con signo.
%e	Notación científica e minúscula.
%f	Coma flotante.
%o	Octal sin signo.
%s	Cadena de caracteres.
%u	Enteros decimales sin signo.
%x	Hexadecimal sin signo letras minúsculas.
%X	Hexadecimal sin signo letras MAYUSCULAS
%p	Apuntador.
%g	Usa el más corto entre %e y %f
%%	Signo de tanto por ciento: %
%n	Se debe indicar la dirección de una variable entera (como en scanf), y en ella quedará guardado el número de caracteres impresos hasta ese momento
%ld	Entero largo

Además, las órdenes de formato pueden tener **modificadores**, que se sitúan entre el % y la letra identificativa del código.

- Si el modificador es un número, especifica la anchura mínima en la que se escribe ese argumento.
- Si ese número empieza por 0, los espacios sobrantes (si los hay) de la anchura mínima se rellenan con 0.

- Si ese número tiene decimales, indica el número de dígitos enteros y decimales si los que se va a escribir es un número, o la anchura mínima y máxima si se trata de una cadena de caracteres.
- Si el número es negativo, la salida se justificará a la izquierda (en caso contrario, es a la derecha -por defecto-).
- Hay otros dos posibles modificadores: la letra l, que indica que se va a escribir un long, y la letra h, que indica que se trata de un short.

"Todo esto es para printf, pero coincide prácticamente en el caso de scanf. "

Secuencias de Escape

Existen caracteres especiales que suelen representar caracteres tales como la comilla, doble comilla, signo ? y son llamados **Secuencias de Escape**:

TIPO	DESCRIPCION
'\n'	Salto de línea
'\r'	Retorno de carro
'\t'	Tabulación horizontal
'\v'	Tabulación Vertical
'\f'	Avance de pagina
'\b'	Retroceso de espacio
'\a'	Alerta(pitido sonoro)
'\\'	Barra inclinada inversa
'\'	Comilla simple
'\"'	Doble comilla
'\?'	Signo interrogación
'\000'	Numero Octal
'\xhhh'	Numero hexadecimal

Ejemplos

1. **printf**("Bienvenidos al mundo de C.");

   Despliega el mensaje **Bienvenidos al mundo de C**.

2. **printf**("El numero es: %d", 10);

   Despliega el mensaje **El número es: 10**.

3. **printf**("%f es un número de coma flotante.", 23.98);

   Despliega el mensaje **23.98 es un número de coma flotante**.

4. **printf**("%05d", c);

   Despliega el valor de c rellenado con ceros si tiene menos de 5 dígitos

5. **printf**("%5.7s",s);

   Muestra s como una cadena de al menos 5 caracteres de longitud y no más de 7.

6. **printf**("%10.4f",x);

   Imprime un número real x de al menos 10 dígitos con 4 posiciones decimales.

Función scanf()

De igual modo, para la lectura de datos de todos los tipos desde el teclado tenemos la función **scanf**(). Esta función está definida por el estándar ANSI y requiere del archivo header stdio.h, para poder ejecutarse. La sintaxis de esta función es:

scanf(*formato, argumentos* **);**

La cadena de control apuntada por formato se compone de tres tipos de elementos: especificadores de formato, caracteres de espacio en blanco y caracteres que no sean espacios en blanco.

Los especificadores de formato de entrada están precedidos por el signo % y le indican a scanf que tipo de dato será leído. Los códigos de formato se mostraron en la tabla correspondiente a la función **printf**() (ver Tabla anterior).

La cadena de formato se lee de izquierda a derecha y los códigos de formato se asocian en orden, con los argumentos que se encuentran en la lista.

Todas las variables usadas para recibir valores a través de **scanf**() se deben pasar por sus direcciones. Esto significa que todos los argumentos deben ser apuntadores a las variables donde se almacenarán los datos leídos. Esto se especifica con el uso del carácter **&** precediendo al nombre de la variable.

Esta función delimita las cadenas con espacios y saltos de línea. Por tal razón, solo la primera palabra tecleada por el usuario es leída por **scanf**(). Esta función coloca en buffers todas las palabras introducidas desde el teclado. Esto significa que si más de una palabra es tecleada, **scanf**() tomará automáticamente la segunda palabra y la introducirá en la siguiente llamada a **scanf**().

Ejemplos

1. **scanf**("%d", &numero);

   Lee un entero que se almacenará en la variable numero.

2. **scanf**("%s%f", cadena, &numero);

   Lee una cadena de caracteres y un número de coma flotante.

3. **scanf**("%d:%d", &hora, &minuto);

   Lee dos enteros y descarta los dos puntos.

4. char cad[80];

   scanf("%s", cad);

   Lee la cadena cad

5. char s1[80], s2[80];

   scanf("%[0-9]s%[a-zA-Z]s", s1,s2);

   Lee las cadenas s1 y s2 formadas solamente por números y letras.

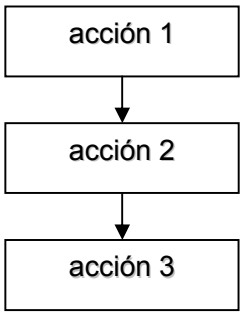
6. char cad[20];

   scanf("%20s", cad);

   Limita el número de caracteres leidos para la cadena cad a no más de 20.

SENTENCIAS DE CONTROL DE PROGRAMA

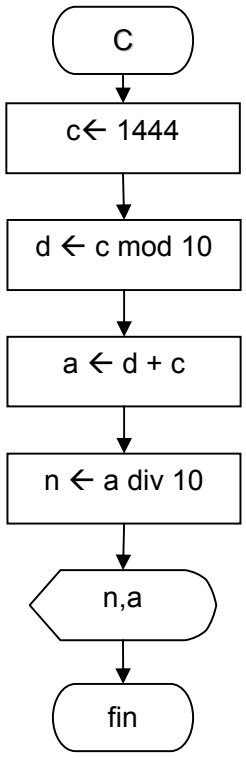
Estructuras Secuenciales




```
acción 1;
acciön 2;
acción 3;
```

Este conjunto se comportará sintácticamente como una sentencia simple y la llave de cierre del bloque NO debe ir seguida de punto y coma (**;**).

Ejemplo

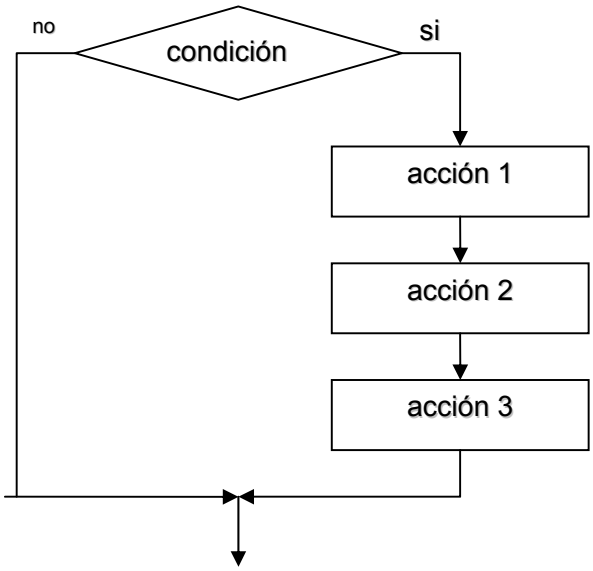



```
#include<stdio.h>
#include<conio.h>

void main (void)
{
    int c, d, a, n ;
    c = 1444;
    d = c % 10;
    a = d + c;
    n = a/10;
    printf("%d %d",n,a);
}
```

c	d	a	n	Mostrar
~~oo~~ 1444	~~oo~~ 4	~~oo~~ 1448	~~oo~~ 144	144 1448

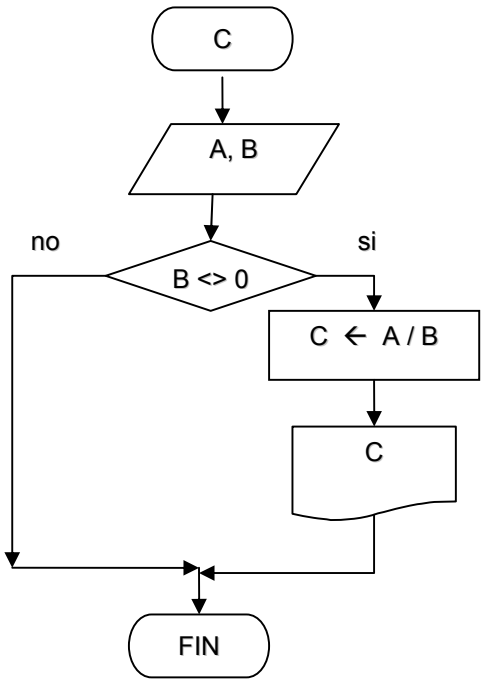
Estructuras Selectivas

Selectiva Simple



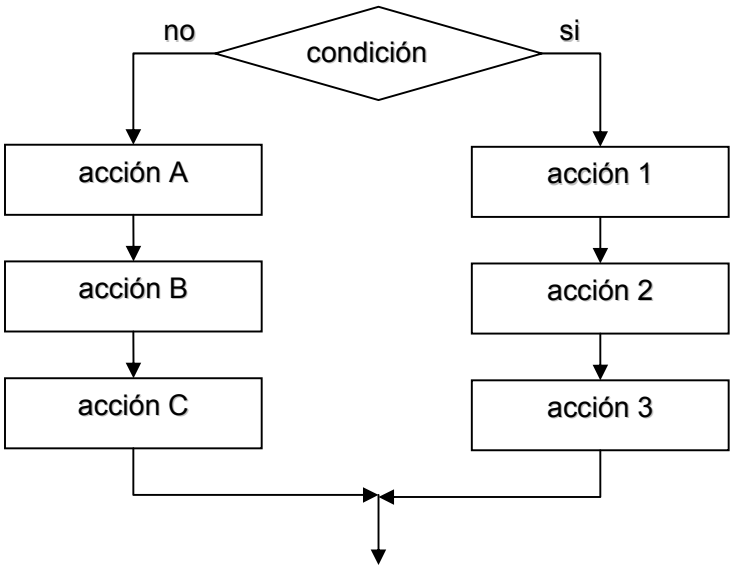

```
if (condición)
{
    acción 1;
    acción 2;
    acción 3;
}
```

Ejemplo



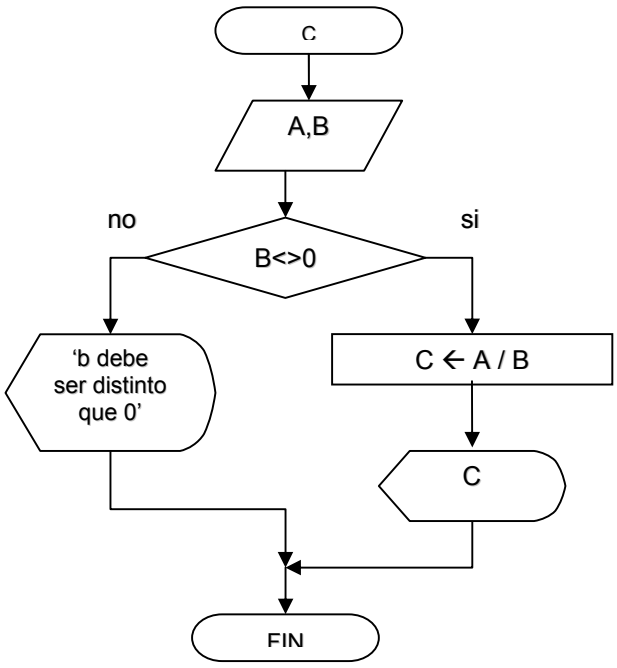

```
#include<stdio.h>
#include<conio.h>

void main (void)
{
    float a, b,  c;

    scanf("%f%f", &a, &b);
    if( b != 0)
    {
        c = a/b;
        printf("%.3f" ,c);
    }
}
```

Selectiva Doble



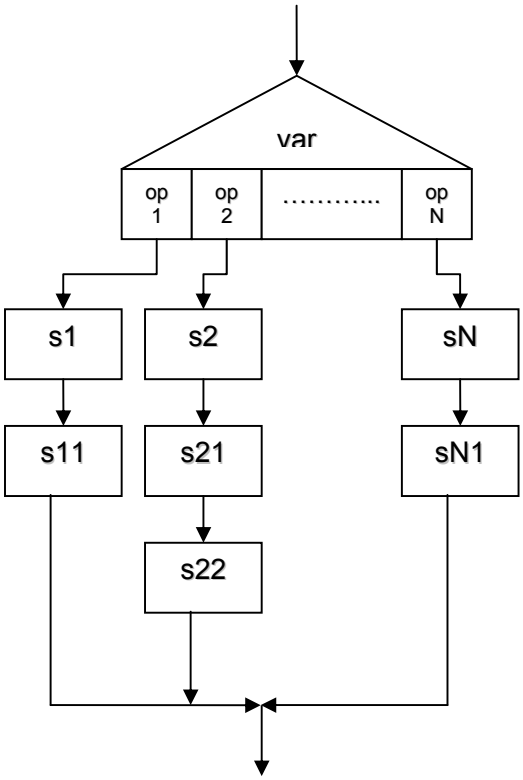

```
if (condición)
{
    acción 1;
    acción 2;
    acción 3;
}

else

{
    acción A;
    acción B;
    acción C;
}
```

Ejemplo

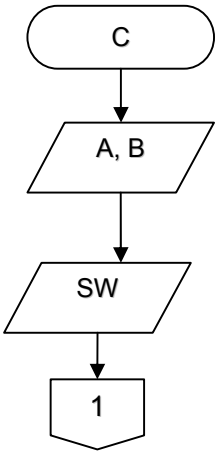


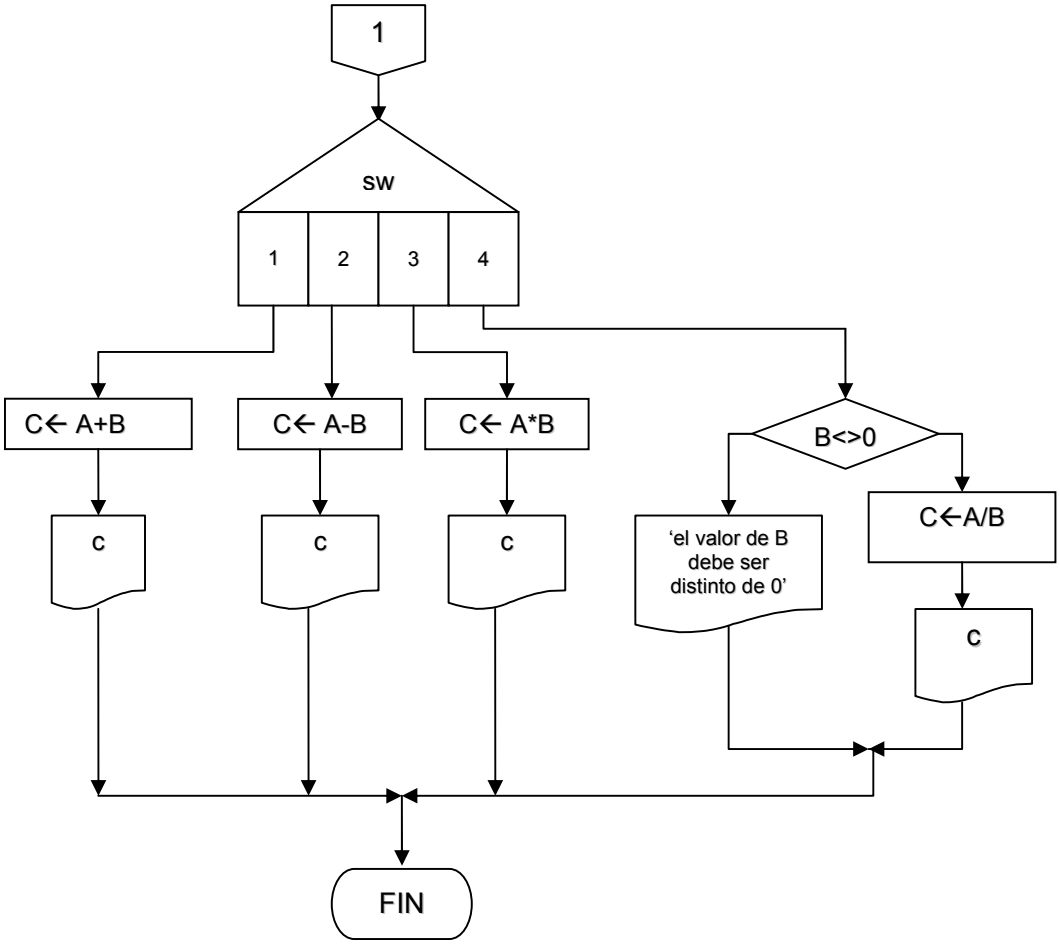

```
#include<stdio.h>
#include<conio.h>

void main (void)
{
   float a, b,  c;

   scanf("%f%f", &a, &b);
   if( b != 0)
   {
      c = a/b;
      printf("%.3f" ,c);
   }
   else
   {
      printf("b debe ser distinto
                que 0");
   }
}
```

Selectiva Múltiple




```
switch (var)
{
   case op1 :
      s1;
      s11;
      break;

   case op2 :
      s2;
      s21;
      break;
   .
   .
   .
   case opN :
      sN;
      sN1;
      break;

   default :
      sd1;
      sd2;
}
```





Codificación en C

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

void main (void)
{
   float a, b,  c ;
   int sw ;

   printf("CALCULADORA ELEMENTAR");
   printf("\n\n\nIngrese los valores con los que desea operar ===>");
   scanf("%f%f", &a, &b);
   printf("\n\n\n\n1. Sumar  2. Restar.   3. Multiplicar   4. Dividir");
   scan("%d", &sw);
```

```
   switch (sw)
   {
      case 1 :
         c = a + b ;
         printf("%.3f + %.3f = %.3f", a, b, c);
         break;

      case 2 :
         c = a - b ;
         printf("%.3f - %.3f = %.3f", a, b, c);
         break;

      case 3 :
         c = a * b ;
         printf("%.3f *%.3f = %f", a, b, c);
         break;

      case 4 :
          if( b != 0)
          {
             c = a/b;
             printf("%.3f / %.3f = %.3f" ,c);
          }
          else
             printf("b debe ser distinto que  0");
          break;
   }

   getch();
} /*function main */
```

Ejemplos de Programas

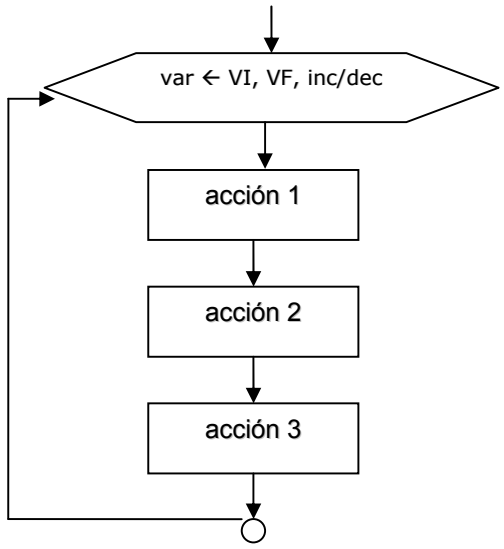
1)

```
/*****************************************************

                Programa que adivina el número mágico

*****************************************************/

#include <stdio.h>

#include <stdlib.h>

#include <conio.h>
```

```
/*Programa del numero magico */
int main()
{
	int magico = 123; /*numero magico*/
	int intento;
	clrscr();
	printf("adivine el numero magico:   ");
	scanf("%d", &intento);
	if (intento == magico)
	{
		printf("***Correcto***");
		printf("%d es el numero magico",magico);
	}
	else
	if (intento > magico)
		printf(".. Incorrecto.. Demasiado alto ");
	else
		printf(".. Incorrecto.. Demasiado bajo ");
	getch();
	return 0;
}
```

Estructuras Repetitivas

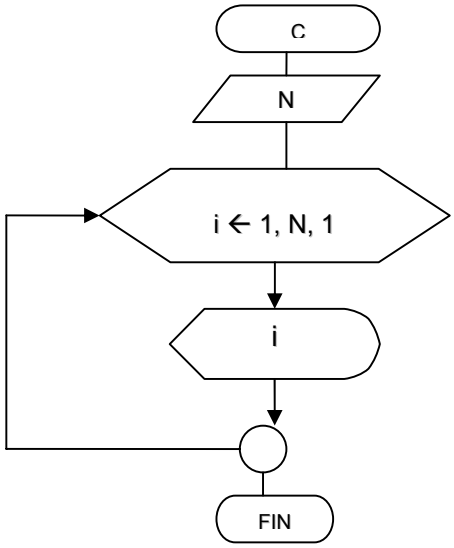
Para-Desde (for)




```
/* si VI es menor que VF */
for ( var = VI;  var <= VF; inc)

{
   sentencia 1;
   sentencia 2;
   sentencia 3;
}
```

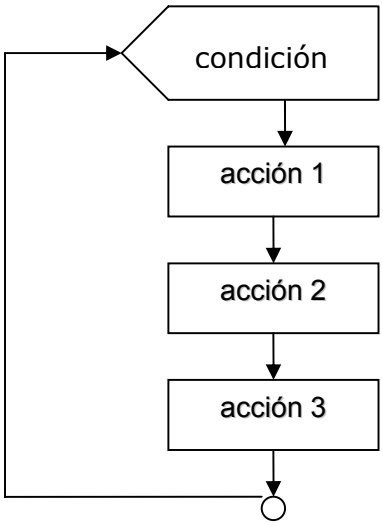
```
/* si VI es menor que VF */
for ( var = VI;  var >= VF; dec)

{
   sentencia 1;
   sentencia 2;
   sentencia 3;
}
```

Ejemplo: Programa que muestra los primeros N números naturales




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

void main (void)
{
   int n, i ;

   clrscr();
   scanf("%d", &n);
   for ( i = 1 ; i <= n ; i++)
   {
      printf("%4d" ,i);
   }
   getch();
}
```

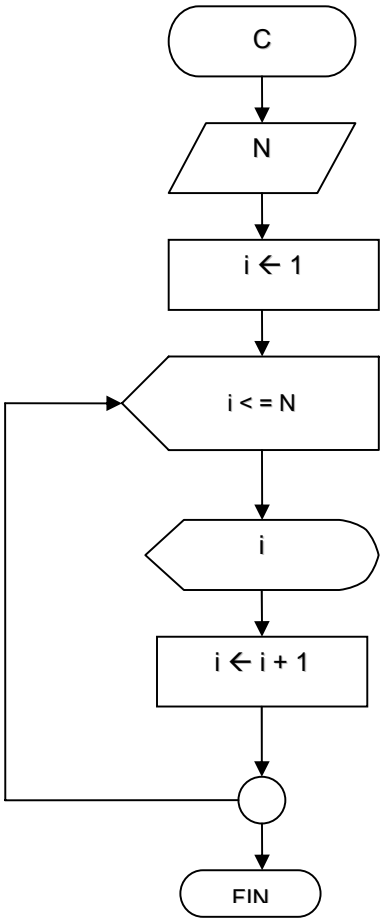
Mientras Hacer (while)




```
while ( condición)
{
    sentencia 1;
    sentencia 2;
    sentencia 3;
}
```

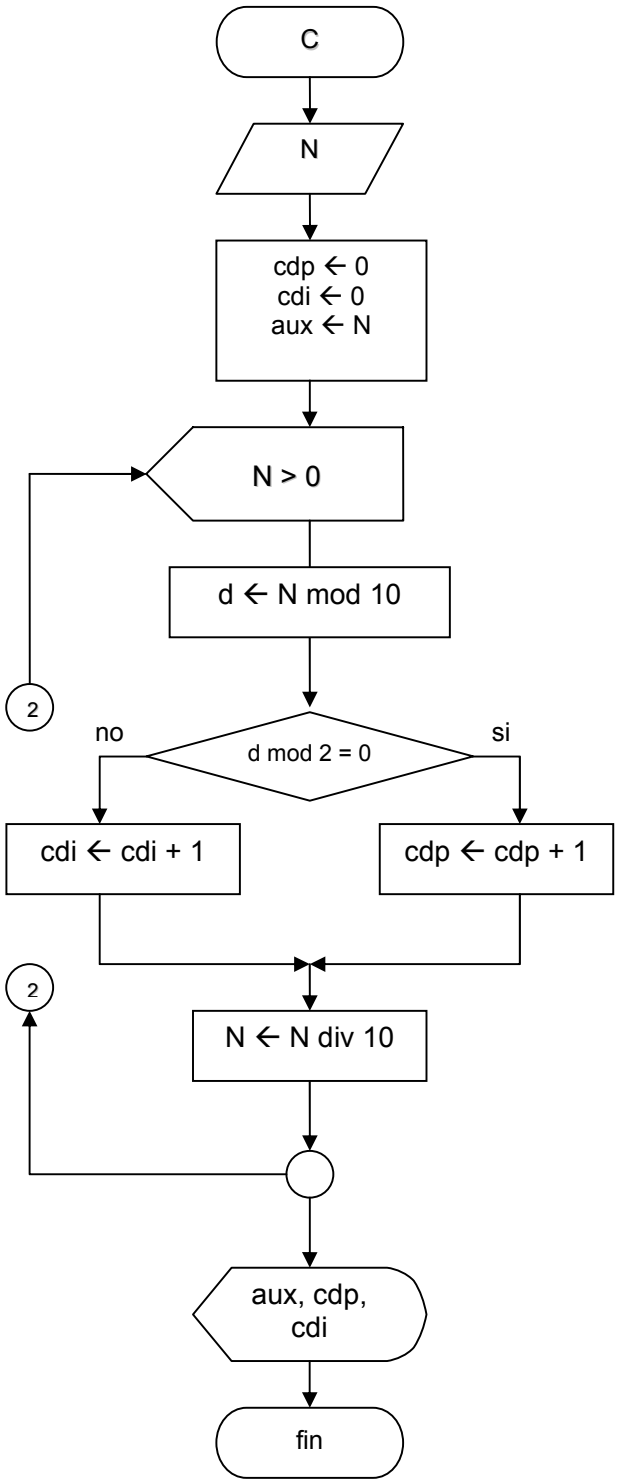
Ejemplos

1) Programa que muestra los primeros N números naturales



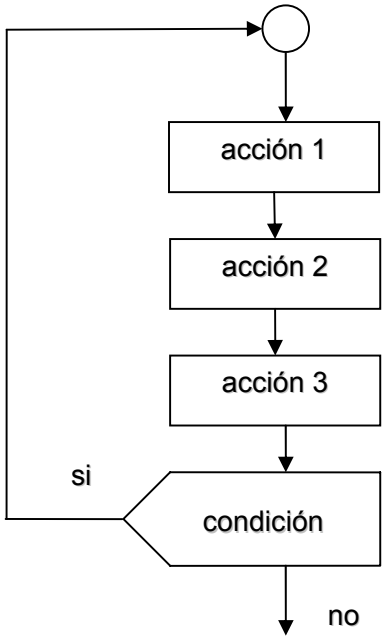

```
#include<stdio.h>
#include<conio.h>

void main (void)
{
   int n, i ;

   clrscr();
   scanf("%d", &n);
   i = 1;
   while ( i <= n)
   {
      printf("%4d" ,i);
      i = i +1 ;
   }
   getch();
}
```

2) Programa que lee un número N y determina la cantidad de dígitos pares e impares que tiene.




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

void main (void)
{
   int n, d, cdi, cdp, aux ;
   clrscr();
   scanf("%d", &n);
   cdp = 0;
   cdi = 0;
   aux = n;
   while(n > 0)
   {
        d = n % 10;
        if(d % 2 == 0)
        {
             cdp = cdp  + 1;
        }
        else
        {
             cdi = cdi  + 1;
        }
        n = n / 10;
   }
   printf("%d\n", aux);
   printf("%d\n", cdp);
   printf("%d", cdi);
   getch();
}
```

Hacer Mientras (do-while)




```
do
{
    sentencia 1;
    sentencia 2;
    sentencia 3;

} while ( condición );
```

Ejemplos

1) Para validar que un número N sea positivo

```
do
{
   scanf("%d", &num);
} while (num <= 0);
```

2)

```
/* ************************************************************
Programa que muestra  un menú en la pantalla hasta que se presione una de las
opciones.
************************************************************ */
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
```

```
/* ************     M  A  I  N  ************* */

int main()
{
	float a, b, s, r, m, d;
	char op;
	clrscr();

	printf("Ingrese los valores con los que desea operar...\n");
	printf("\nA = ");
	scanf("%f", &a);
	printf("\nB = ");
	scanf("%f", &b);
	printf("\n");
	do
	{
		clrscr();
		printf("1. Sumar\n");
		printf("2. Restar\n");
		printf("3. Multiplicar\n");
		printf("4. Dividir\n");
		printf("5. Salir");
		printf("\nIntroduzca opcion ==> ");
		op = getche();
		printf("\n");
		switch(op)
		{
			case '1' : s = a + b;
				  printf("\nLa suma de %.2f + %.2f es %.2f ", a, b, s);
				  break;
			case '2' : r = a - b;
				  printf("\nLa resta de %.2f - %.2f es %.2f ", a, b, r);
				  break;
			case '3' : m = a * b;
				  printf("\nEl producto de %.2f * %.2f es %.2f ", a, b,m);
				  break;
			case '4' : d = a/b;
				  printf("\nEl cociente de %.2f / %.2f es %.2f ", a, b, d);
				  break;
		}
		getch();
	} while (op != '5');

	return 0;
}
```

La sentencia break

La sentencia **break** sirve para terminar loops (bucles) producidos por WHILE, DO-WHILE y FOR antes que se cumpla la condición normal de terminación. En el EJEMPLO siguiente vemos su uso para terminar un WHILE indeterminado.

```
#include <stdio.h>
#include <conio.h>
main()
{
  char c ;

  printf("ESTE ES UN LOOP INDEFINIDO ") ;
  while(1)
  {
      printf( "DENTRO DEL LOOP INDEFINIDO (presione una tecla para salir):" ) ;
      if( (c = getch()) == 'Q' )
          break ;
     printf( "\nNO FUE LA TECLA CORRECTA PARA ABANDONAR EL LOOP ") ;
  }
   printf("\nTECLA CORRECTA : FIN DEL WHILE ") ;
}
```

Obsérvese que la expresión while(1) SIEMPRE es cierta , por lo que el programa correrá indefinidamente hasta que el operador oprima la tecla "secreta" Q . Esto se consigue con 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 */
 }
```

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 permitiendo al programa continuar con la siguiente.

En este ejemplo se mostrarán sólo aquellos números que sean positivos mientras sean distintos de 100.

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

   do
   {
      scanf("%d", &num);
      if (x<0)
         continue
      printf("%d", x);
   } while (x != 100);
}
```

La funcion exit()

La función **exit()** tiene una operatoria mucho más drástica que las anteriores, en vez de saltear una iteración ó abandonar un lazo de programa, ésta abandona directamente el programa mismo dándolo por terminado y forzando su regreso al sistema operativo. 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 e 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.

CAPÍTULO II

PROGRAMACIÓN MODULAR

Introducción

Una estrategia muy utilizada para la resolución de problemas complejos con la computadora, es la división del problema en otros problemas más pequeños o subproblemas. Estos subproblemas se implementarán mediante módulos o subprogramas.

Los subprogramas son una herramienta importante para el desarrollo de algoritmos y programas de modo que normalmente un proyecto de programación está compuesto generalmente de un programa principal y un conjunto de subprogramas con las llamadas a los mismos dentro del programa principal.

Los subprogramas se clasifican en:

- Procedimientos
- Funciones

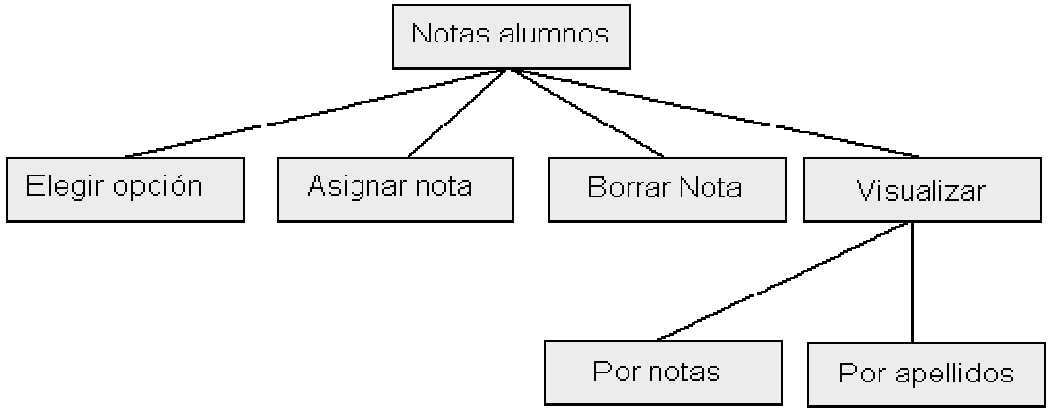
El diseño modular: subprogramas

Uno de los métodos fundamentales para resolver un problema es dividirlo en problemas más pequeños, llamados *subproblemas*.

Estos problemas a su vez pueden ser divididos repetidamente en problemas más pequeños hasta que los problemas más pequeños puedan ser solucionados.

Esta técnica de dividir el problema principal en subproblemas se denomina frecuentemente ***divide y vencerás***. El método de diseño se denomina **diseño descendente**, debido a que se comienza en la parte superior con un problema general y se diseñan soluciones específicas a sus subproblemas.

Veamos un ejemplo de cómo emplear el diseño descendente para resolver un problema. Supongamos que un profesor quiere crear un programa para gestionar las **notas** de sus alumnos. Quiere que dicho programa le permita realizar tareas tales como asignar notas, cambiar notas, ver las notas según distintas calificaciones, etc. A continuación se tiene un esquema que representa una de las posibles divisiones del problema en módulos.



El problema principal se resuelve con el programa principal (también llamado conductor del programa), y los subproblemas (módulos) mediante subprogramas: *procedimientos y funciones*. Cada subprograma realiza una tarea concreta que se describe con una serie de instrucciones.

Ejemplo

Leer el radio de un círculo y calcular e imprimir su superficie y longitud.

- **Análisis**

  Especificaciones de Entrada
  Radio: Real

  Especificaciones de Salida
  Superficie: Real
  Longitud: Real

- **Algoritmo**

  1. Leer el valor del radio
  2. Calcular la Superficie
  3. Calcular la Longitud
  4. Visualizar los valores de la superficie y la longitud

- **Refinamiento del Algoritmo**

  1. Leer el valor del radio
  2. Calcular la superficie
     2.1. pi = 3.141592 (constante pi)
     2.2. S ← pi * Radio * Radio

3. Calcular la longitud
   3.1. pi ← 3.141592
   3.2. L ← 2 * pi * Radio
4. Visualizar los valores de la superficie y la longitud

El proceso de descomposición de un problema en módulos se conoce como **modularización** y a la programación relativa a ellos **programación modular**.

Procedimientos y funciones

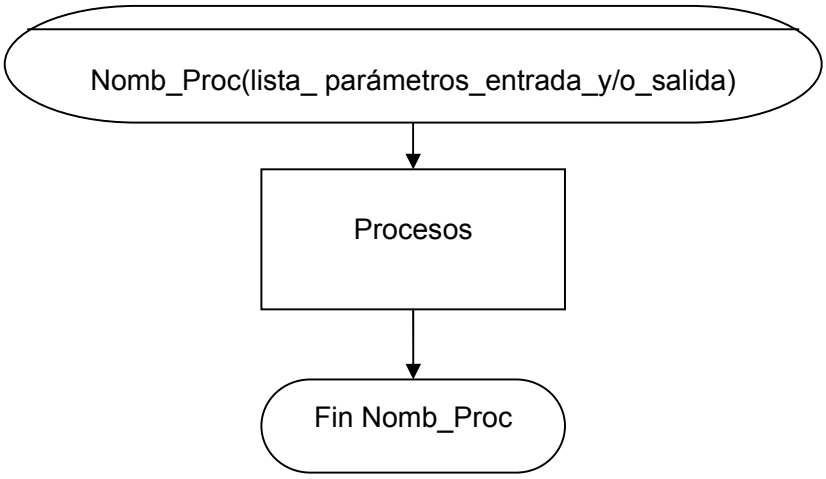
Procedimientos

Un procedimiento es un subprograma que realiza una tarea específica. Puede recibir cero o más valores del programa que llama (conocidos con el nombre de parámetros de entrada) y devolver cero o más valores a dicho programa (conocidos como parámetros de salida).

Un procedimiento está compuesto por un grupo de sentencias al que se asigna un nombre de procedimiento (identificador) y constituye una unidad de programa. La tarea determinada al procedimiento se ejecuta siempre que se encuentra el nombre del procedimiento.

Declaración de procedimientos

La forma de declarar un procedimiento es la siguiente:



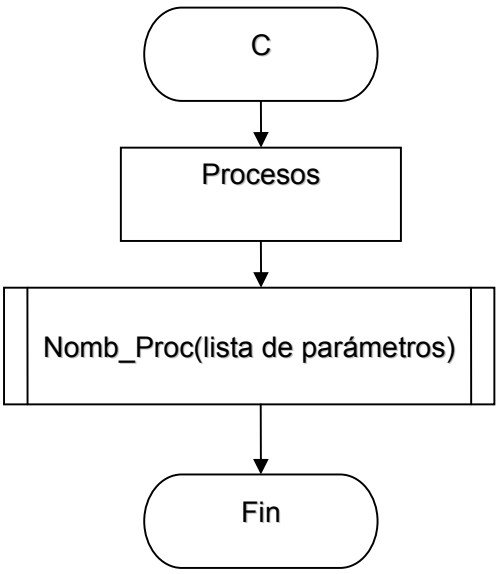
donde:

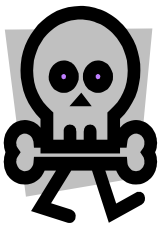
Nomb_Proc : Nombre que se le da al procedimiento

lista parámetros entrada y/o salida : Son los parámetros de entrada y/o salida necesarios para el procedimiento

Llamada a un procedimiento

La forma de llamar a un procedimiento es:





Precaución

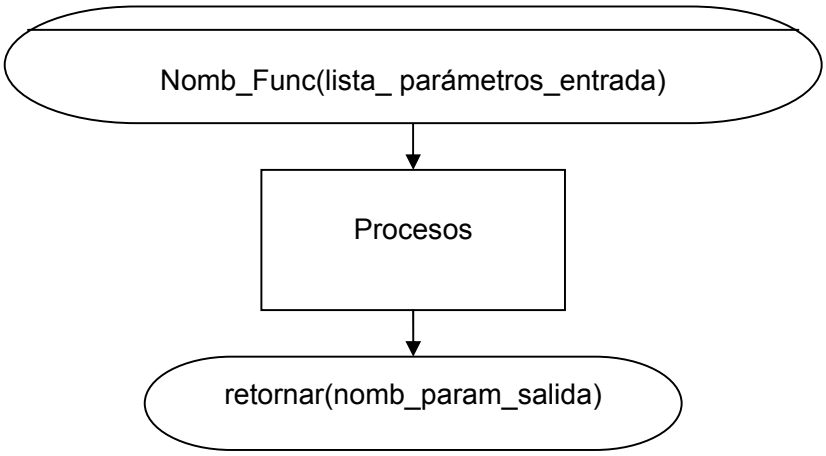
Los parámetros al momento de hacer la llamada al procedimiento **deben coincidir** en número, orden y tipo con los parámetros de la declaración del mismo.

Funciones

Una función es un subprograma que recibe como argumentos o parámetros, datos de tipos numérico o no numérico, y devuelve un único resultado. Esta característica le diferencia esencialmente de un procedimiento.

Declaración de funciones

La forma de declarar un procedimiento es la siguiente:

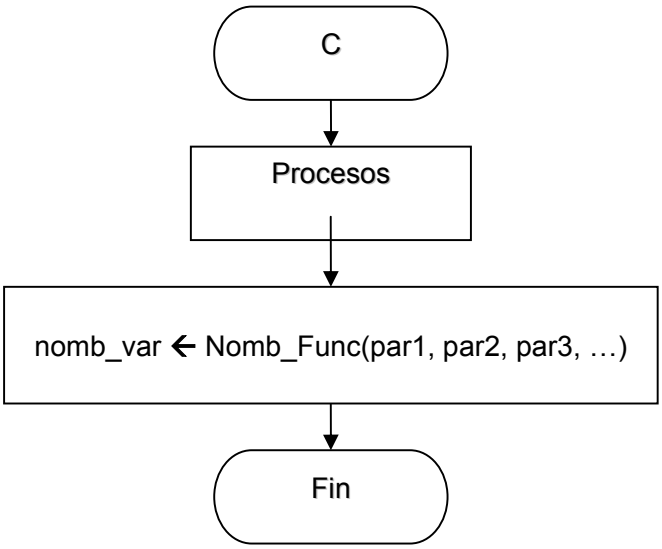


donde:

Nomb_Func	:	Nombre que se le da a la función
Lista de parámetros entrada	:	Son los parámetros de entrada necesarios para la función
nomb_param_salida	:	Es el nombre de la variable que almacena el parámetro de salida o resultado de la función.

Llamada a funciones

Una función es llamada por medio de su nombre, en una sentencia de asignación:



Diferencias entre procedimientos y funciones

Los procedimientos y funciones son similares, aunque presentan notables diferencias entre ellos:

1. Las **funciones** normalmente, devuelven **un solo valor** a la unidad de programa (programa principal u otro subprograma) que las referencia o llama. Los **procedimientos** pueden devolver **cero, uno o varios valores**. En el caso de no devolver ningún valor, realizan alguna tarea tal como operaciones de entrada/salida.

2. A un nombre de procedimiento no se puede asignar un valor, y por consiguiente ningún tipo esta asociado con un nombre de procedimiento.

3. Una función se referencia utilizando su nombre en una instrucción (de asignación o expresión matemática), mientras que un procedimiento se referencia por una llamada o invocación al mismo.

Transferencia de información a/desde módulos

Los parámetros

Una de las características más importantes de los subprogramas es la posibilidad de **comunicación entre el programa principal y los subprogramas** (o entre los subprogramas). Esta comunicación se realiza a través de una lista de **parámetros o argumentos**.

Así pues, los módulos se clasifican en:

- Módulos sin parámetros (no existe comunicación entre el programa principal y los módulos o entre módulos).

- Módulos con parámetros (existe comunicación entre el programa principal y los módulos, y entre ellos).

Un parámetro es prácticamente, una variable cuyo valor debe ser o bien proporcionado por el programa principal al módulo o ser devuelto desde el módulo hasta el programa principal. Por consiguiente hay dos tipos de parámetros: parámetros de *entrada y parámetros de salida*.

Los parámetros de entrada son aquellos cuyos valores deben ser proporcionados por el programa principal, y los de salida son aquellos cuyos valores se calcularán en el subprograma o módulo y se deben devolver al programa principal para su proceso posterior.

Parámetros por valor y parámetros por referencia

Existen dos tipos de parámetros, que nos ayudan a transferir/recibir información de otros subprogramas, o del programa principal, a saber: *parámetros de entrada (por valor) y parámetros de salida o de entrada/salida (por referencia).*

Parámetros por Valor

Son parámetros unidireccionales que se utilizan para proporcionar información a un subprograma, pero no pueden devolver valores, al programa llamador.

Se les llama parámetros de entrada, ya que en la llamada al subprograma el valor del parámetro actual se pasa a la variable que representa a la variable actual. Este valor puede ser modificado dentro del subprograma pero el valor modificado no es devuelto al programa o subprograma llamador. En otras palabras, cuando se define un parámetro por valor el subprograma sólo recibe una **copia del valor** del dato que el programa invocador le pasa. Por tanto si en el procedimiento modificamos este valor, el dato original permanecerá **inalterado**.

Parámetros por Referencia

Se utilizan tanto para recibir como para transmitir valores entre el subprograma y el programa llamador. Este puede actuar como parámetro de salida o de entrada/salida. En otras palabras, cuando definimos parámetros por referencia lo que se pasa al procedimiento son los datos en sí, y si éste los modifica, los cambios permanecerán una vez que la ejecución vuelva al módulo que invocó al procedimiento.

Variables locales y variables globales

Las variables utilizadas en un programa con subprogramas pueden ser de dos tipos: *locales y globales*.

Variables Locales

Una variable local es una variable que esta declarada dentro de un subprograma y se dice que es local al subprograma. Una variable local sólo está disponible durante el funcionamiento del subprograma, al terminar su función el subprograma y regresar al programa llamador, se pierde el valor que se encontraba guardado en la variable local.

Variables Globales

Las variables declaradas en el programa principal se denominan variables globales. Al contrario que las variables locales cuyos valores se pueden utilizar sólo dentro del subprograma en que fueron declaradas, las variables globales se pueden utilizar en todo el programa principal y en todos los subprogramas, donde se haga referencia al identificador de esta variable.

Ambito de un identificador

La mayoría de los programas tienen una estructura tipo árbol, el programa principal es la raíz y de este penden muchas ramas (procedimientos y funciones).

Los subprogramas en los que un identificador puede ser utilizado se conocen como ámbito o alcance del identificador, dicho de otro modo, es en esta sección donde el identificador es válido.

Reglas de Ámbito

1. El ámbito de un identificador es el dominio en que esta declarado. Por consiguiente un identificador declarado en un bloque P puede ser usado en el subprograma P y en todos los subprogramas llamados en el subprograma P.

Si un identificador j declarado en el procedimiento P se redeclara en algún subprograma interno Q invocado en P, entonces el subprograma Q y todas sus invocaciones a otros subprogramas se excluyen del ámbito de j declarado en P.

FUNCIONES EN C

Las funciones son bloques de código utilizados para dividir un programa en partes más pequeñas, cada una de las cuáles tendrá una tarea determinada.

Forma General

```
tipo_función nomb_función( lista de parámetros )
{
     cuerpo de la función;
}
```

Donde:

tipo_función	:	Especifica el tipo de valor que devuelve la función, mediante la sentencia **return**. Si no se especifica ningún tipo, se asume que la función devuelve un entero. Si no queremos que retorne ningún valor deberemos indicar el tipo vacío (**void**).
nomb_función	:	Es el nombre que se utiliza para invocar a la función.
lista de parámetros	:	Lista de variables separadas por comas con sus tipos. Una función puede no tener parámetros en cuyo caso se coloca sólo la palabra **void** en vez de lista de parámetros.
cuerpo de la función	:	Es el conjunto de sentencias que serán ejecutadas cuando se realice la llamada a la función.

Las funciones pueden ser llamadas desde la función **main** o desde otras funciones. Nunca se debe llamar a la función **main** desde otro lugar del programa. Por último recalcar que los argumentos de la función y sus variables locales se destruirán al finalizar la ejecución de la misma.

Declaración de funciones

Al igual que las variables, las funciones también han de ser declaradas. Esto es lo que se conoce como prototipo de una función. Para que un programa en C sea compatible entre distintos compiladores es imprescindible escribir los prototipos de las funciones.
Los prototipos de las funciones pueden escribirse antes de la función **main** o bien en otro archivo. En este último caso se lo indicaremos al compilador mediante la directiva **#include**.

Ejemplos

1. int **cuadrado**(int x);

2. void **multiplica**(int a, int b);

3. float **suma**(float a, float b);

4. int **factorial**(int x);

Definición 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 a diferencia de Pascal, en C las definiciones no pueden anidarse.

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.

Ejemplo

1)

```
#include <stdio.h>

/*********** DECLARACION observe que termina en  ";"  **********/

float Mi_Funcion(int i, double j );

/************* DEFINICION  observe que NO lleva  ";"  *************/

float Mi_Funcion(int i, double j )
{

        float n

        .....................

        printf("%d", i );  /* LLAMADA  a otra función */


        return ( 2 * n );  /* RETORNO devolviendo un valor float */
}
/********************** M   A   I   N ************************/
```

```
void main()

{

        float k ;

        int p ;

        double z ;

        ...........
        k = Mi_Funcion( p, z );   /* LLAMADA a la función */

        ...........

}  /* fin de la función main() */
```

Llamada a una función

El llamado a una función, en el lenguaje C consiste en transferir el control a esa función. Cuando se llama a esa función, se proporciona un nombre de función y una lista de parámetros, si es que los hay. Cuando se llama a una función se realizan los siguientes pasos:

1. El compilador toma nota de la localización desde donde se llamó la función y hace una copia de la lista de parámetros en caso de haberlos.
2. Se crea temporalmente cualquier espacio de almacenamiento que se requiere para ejecutar la función.
3. Comienza la ejecución de la función que se llama, usando copias de los datos que se proporcionaron en la lista de parámetros.
4. Después de que la función termina la ejecución, se regresa el control a la función que la llamo y se libera memoria que se usó para la función.

Formato

```
nomb_variable = nomb_función( parámetros actuales );
```

Donde:

nomb_variable : Especifica el nombre de la variable en donde se guardará el valor devuelto por la función

nomb_función : Especifica el nombre de la función a la que se está llamando.

parámetros actuales : Son los valores que son pasados a la función y asignados a sus correspondientes parámetros. El número de valores en la lista debe ser igual al número de parámetros utilizados en la función y del mismo tipo.

La sentencia return

Tiene dos usos:

1. fuerza una salida inmediata de la función en que se encuentra, para retornar a la función que la llamó.
2. se puede utilizar para devolver un valor

Finalización de una función

Hay dos formas en las que una función puede terminar su ejecución y volver al sitio en que se llamó.

1. cuando se ha ejecutado la última sentencia de la función y se encuentra la llave **}** del final de la función.

2. usando la sentencia **return** para devolver un valor. Se debe tomar en cuenta que una función puede tener varias sentencias return. La forma de devolver un valor es la siguiente:

return(valor o expresión **);**

Ejemplos

1)

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

/********** P  R  O  T  O  T  I  P  O  S **********/

int Mul(int a, int b);
void Pausa(void);
```

```
/******** D E F I N I C I O N  D E  F U N C I O N E S ********/
void Pausa(void)
{
  printf("Presione cualquier tecla para continuar...");
  getch( );
}
/*****************************************************/
int Mul( int a, int b )
{
      return (a*b);
}
/*****************  M  A  I  N  **********************/
int main(void)
{
      int x, y;

      clrscr();
      x = 10;
      y = 20;
      printf("El resultado de %d * %d es %d", x, y, Mul(x, y));
      printf("\n\n");
      Pausa();

      return(0);
}
```

2)

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

/************* P R O T O T I P O S **************/

void Menu(float a, float b);
void Pausa(void);

/********* D E F I N I C I O N  D E  S U B P R O G R A M A S ***********/

void Menu(float a, float b)
{

      char op;
      float s, r, m, d;

      do
      {
            clrscr();
```

```
            printf("1. Sumar\n");
            printf("2. Restar\n");
            printf("3. Multiplicar\n");
            printf("4. Dividir\n");
            printf("5. Salir");
            printf("\nIntroduzca opcion ==> ");
            op = getche();
            printf("\n");
            switch(op)
            {
                  case '1' : s = a + b;
                             printf("\nLa suma de %.2f + %.2f es %.2f", a, b, s);
                             printf("\n\n");
                             Pausa();
                             break;
                  case '2' : r = a - b;
                             printf("\nLa resta de %.2f - %.2f es %.2f ", a, b, r);
                             printf("\n\n");
                             Pausa();
                             break;
                  case '3' : m = a * b;
                             printf("\nEl producto de %.2f * %.2f es %.2f ", a, b,m);
                             printf("\n\n");
                             Pausa();
                             break;
                  case '4' : d = a/b;
                            printf("\nEl cociente de %.2f / %.2f es %.2f        ", a, b, d);
                            printf("\n\n");
                            Pausa();
                            break;
          }
       } while (op != '5');
}
/*************************************************************/

void Pausa(void)
{
  printf("Presione cualquier tecla para continuar...");
  getch();
}

/*************     M  A  I  N  **************/

int main()
{
 float a, b;

 clrscr();
```

```
  printf("Ingrese los valores con los que desea operar...\n");
  printf("\nA = ");
  scanf("%f", &a);
  printf("\nB = ");
  scanf("%f", &b);

  printf("\n");
  Menu(a, b);

  return 0;
}
```

Valores devueltos

Todas las funciones, excepto aquellas de tipo **void**, devuelven un valor. Este valor se especifica explícitamente en la sentencia **return**. Si una función no es especificada como void y si no se especifica un valor de vuelta, entonces el valor devuelto por la función queda técnicamente indefinido. Si no se ha declarado una función como void puede ser usada como operando en cualquier expresión válida.

El valor devuelto por la función debe asignarse a una variable, de lo contrario, el valor se perderá.

En el caso de la funcion **main** la sentencia return devuelve un código de terminación al proceso de llamada (que generalmente es el sistema operativo). El valor devuelto debe ser un número entero, 0 indica que el programa ha terminado normalmente. Todos los demás valores indican que se ha producido algún tipo de error.

Paso de parámetros a una función

Por valor

En este caso es el **valor** del parámetro el que se pasa a la función, de tal forma que los cambios que se hagan en dicho valor no afectarán su valor original (**variables de entrada).**

Por referencia

En este método es la **dirección** del parámetro el que se pasa a la función. Esto significa que los cambios hechos al valor del parámetro permanecen incluso cuando la función ha terminado, modificando de esta forma su valor original **(variables de salida o de entrada/salida)**.

Ejemplos

1. Llamada por valor:

```
#include <stdio.h>
#include <conio.h>
/***************** P R O T O T I P O S *****************/
int cuad(int x);
void Pausa(void);

/***************** D E F I N I C I O N *********************/
void Pausa(void)
{
  printf("Presione cualquier tecla para continuar...");
  getch();
}
/*****************************************************/
int cuad(int x)
{
      x = x*x;
  return(x);
}
/*****************  M  A  I  N  ********************/
int main(void)
{
      int t = 10;

      printf("El cuadrado de %d es %d \n\n", t, cuad(t));
      Pausa();

      return(0);
}
```

2. Llamada por referencia:

```
#include <stdio.h>
#include <conio.h>
/******************* P R O T O T I P O S ******************/
void inter(int *x, int *y);
void Pausa(void);

/******************* D E F I N I C I O N ******************/
void inter(int *x, int *y)
{
      int aux;
      aux = *x; /* guarda el valor de la variable x */
      *x = *y; /* asigna el valor de y en x */
      *y = aux; /* asigna en x el valor de y */
```

```
}
/*********************************************************/
void Pausa(void)
{
  printf("Presione cualquier tecla para continuar...");
  getch();
}
/******************* M A I N ************************/
int main(void)
{
	int x, y;

	clrscr();
	x = 10;
	y = 20;
	system("cls");
	printf("Antes x = %d  e  y = %d\n\n", x, y);
	inter(&x,&y);
	printf("Ahora x = %d  e  y = %d \n\n", x, y);
	Pausa();

	return(0);
}
```

3. Llamada por referencia:

```
/*Programa que muestra los 10 primeros números*/

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


/************** P R O T O T I P O S ****************/

void Mostrar(int num[10]); /*vector num de enteros*/
void Pausa(void);

/***************  D E F I N I C I O N  ****************/

void Pausa(void)
{
  printf("Presione cualquier tecla para continuar...");
  getch();
}
/*********************************************************/

void Mostrar(int num[10])
{
	int i;
```

```
        printf("Los primeros 10 numeros naturales son:\n\n");
        for(i = 0; i < 10; i = i+1)
        {
                printf(" %d ", num[i]);
        }
}

/******************** M A I N ************************/
int main(void)
{
        int t[10], i;

        clrscr();
        for(i=0; i<10;i=i+1)
        {
          t[i] = i;
        }
        Mostrar(t);

        printf("\n\n");
        Pausa();
        return(0);
}
```

4. Llamada por referencia:

```
/*  Programa que muestra los 10 primeros números  */
/* Alternativa 2  */

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

/************** P R O T O T I P O S ****************/

void Mostrar(int *num); /*vector num de enteros*/
void Pausa(void);

/***************  D E F I N I C I O N  ****************/

void Mostrar(int *num)
{
        int i;

        printf("Los primeros 10 numeros naturales son:\n\n");
        for(i = 0; i < 10; i = i+1)
        {
                printf(" %d ", num[i]);
        }
}
/*********************************************************/

void Pausa(void)
{
```

```
  printf("Presione cualquier tecla para continuar...");
  getch();
}
/******************** M A I N ************************/
int main(void)
{
        int t[10], i;

        clrscr();
        for(i=0; i<10;i=i+1)
        {
          t[i] = i;
        }
        Mostrar(t);

        printf("\n\n");
        Pausa();
        return(0);
}
```

5. Llamada por referencia

```
/*Programa que muestra los 10 primeros números*/
/* Alternativa 3  */

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

/************** P R O T O T I P O S ****************/

void Pausa(void);
void Mostrar(int num); /*vector num de enteros*/

/***************  D E F I N I C I O N  ****************/
void Pausa(void)
{
  printf("Presione cualquier tecla para continuar...");
  getch();
}
/*****************************************************/
void Mostrar(int num)
{
        int i;

        printf(" %d ", num);

}
```

```
/******************** M A I N ************************/
int main(void)
{
        int t[10], i;

        clrscr();
        for(i=0; i<10;i=i+1)
        {
          t[i] = i;
        }
        printf("Los primeros 10 numeros naturales son:\n\n");
        for (i = 0; i < 10; i++)
        {
                Mostrar(t[i]);
        }

        printf("\n\n");
        Pausa();
        return(0);
}
```

6. Llamada por referencia:

```
/*Programa con funcion que devuelve un valor no entero*/

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

/************** P R O T O T I P O S ****************/

double Raiz(double x); /*declaracion de la funcion*/
void Pausa(void);

/***************  D E F I N I C I O N  ****************/
double Raiz(double x)
{

  return(sqrt(x)*2.0); /*devuelve la raiz cuadrada de un numero
                         multiplicado por 2*/
}
/*****************************************************************/
void Pausa(void)
{
  printf("Presione cualquier tecla para continuar...");
  getch();
}
/******************** M A I N ************************/
int main(void)
{
        float x;
        x = 10.0;
        clrscr();
```

```
        printf("La raiz cuadrada de %.3f * 2 es %.3lf", x, Raiz(x));

        printf("\n\n");
        Pausa();
        return(0);
}
```

Ambito de las Variables

Según el lugar donde son declaradas puede haber dos tipos de variables: **globales** o **locales**.

Variables globales

Si definimos una variable FUERA de cualquier función (incluyendo a la función main()), estaremos frente a lo que denominaremos VARIABLE GLOBAL. Este tipo de variable existirá todo el tiempo que se esté ejecutando el programa. Se crean al iniciarse éste y se destruyen de la memoria al finalizar. 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 utilizarlas.

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 AUTOMÁTICAS, ya que son creadas y destruidas automáticamente por la llamada y el retorno de una función, respectivamente.

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. Reacuérdese entonces que, si no se las inicializa expresamente en el momento de su definición, su valor será indeterminado (basura).

El identificador (nombre de la variable) **NO** puede ser una **palabra clave** y los caracteres que podemos utilizar son las letras: **a-z** y **A-Z** (ojo! la **ñ** o **Ñ** no está permitida), los números: **0-9** y el símbolo de subrayado _. Además hay que tener en cuenta que el primer caracter no puede ser un número.

Ejemplo

```
/* Declaracion de variables */

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

int a; /* declaración de variable global */
void main(void) /* Muestra dos valores */
{
      int b=4;  /* declaración de variable local */

      clrscr();
      printf("b es local y vale %d",b);
      a=5;
      printf("\na es global y vale %d",a);
      getch();
}
```

Algunas funciones estándar del C

El C del estándar ANSI define 22 funciones matemáticas que entran en las siguientes categorías: Funciones trigonométricas, hiperbólicas, logarítmicas y exponenciales, otras.

Todas las funciones matemáticas necesitan que se incluya el archivo de cabecera **math.h** en cualquier programa que las utilice.

1. **double log(double x) :** Perrmite calcular el logaritmo neperiano (base e) del argumento x. Produce error si x es negativo o si x =0.
   Ejemplo: Y= log (x) + 6;

2. **double log10(double x) :** Permite calcular el logaritmo en base 10 del argumento x. Produce error si x es negativo o si x =0.
   Ejemplo: Y= log10(x) + 6;

3. **double exp(double x):** Permite calcular el exponencial del argumento x, es decir permite calcular ex

   Ejemplo: Y= exp(x);

4. **double sqrt(double x):** Permite calcular la raíz cuadrada del argumento. El argumento debe ser mayor o igual que cero y real, el resultado es real. Se usa de igual manera que las funciones anteriores.

5. **double pow(double base, double exp):** Nos devuelve el argumento base elevado a exp(baseexp)

   Ejemplo: El siguiente programa escribe dos elevado al cuadrado

```
#include <math.h>
#include <stdio.h>
void main(void)
{
   printf("%lf"',pow(2,2))
}
```

6. **double sin(double x), double cos(double x) :** Estas dos funciones nos permiten calcular Seno y Coseno respectivamente de sus argumento dado en radianes. Se usa de igual manera que las funciones anteriores.

7. **double atan(double x) :** Esta función devuelve el arco tangente de x. El valor de x debe estar en el rango de -1 a l; en cualquier otro caso se produce un error de dominio. El valor se especifica en radianes

8. **double atan2(double y, double x) :** Esta función devuelve el arco tangente de y/x. Utiliza el signo de su argumento para obtener el cuadrante del valor devuelto. El valor de x se especifica en radianes.

9. **double abs(double x) :** Calcula el valor absoluto de un número dado.

10. **double fabs(double x) :** Devuelve el valor absoluto de x.

11. **double floor(double x):** Toma el argumento y retorna el mayor entero que no es mayor que x.

    Ejemplo

    floor de 1.02 devuelve 1.0, el floor de -1.02 devuelve -2.

12. **double fmod(double x, double y) :** La función fmod calcula el residuo de la división entera de x/y.

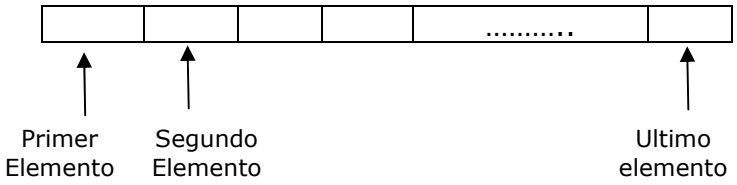
CAPITULO III
ESTRUCTURAS DE DATOS ESTATICAS

Un array es un identificador que referencia un conjunto de datos del mismo tipo. Imagina un tipo de dato **int**; podremos crear un conjunto de datos de ese tipo y utilizar uno u otro con sólo cambiar el índice que lo referencia. El índice será un valor entero y positivo. En **C** los arrays comienzan por la posición **0**.

Arreglos Unidimensionales (Vectores)

Los arreglos unidimensionales son arreglos de una sola dimensión, y para operar con cualquiera de sus elementos se debe escribir el nombre del arreglo seguido de **UN** subíndice entre corchetes [ ].

Representación gráfica:



Los arreglos están compuestos elementos, donde cada elemento, a su vez, está formado por los índices y datos, los índices hacen referencia a la cantidad de elementos del arreglo y a la posición en la cual se localizan, los datos se refieren a la información que se almacena en cada una de las casillas, por lo tanto para hacer referencia a un dato de un arreglo se utiliza el nombre del arreglo y el índice del elemento.

Ejemplo

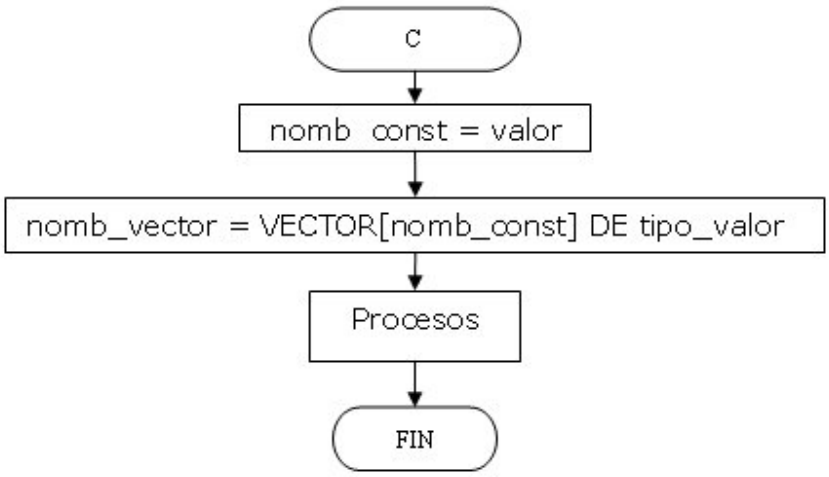
Por ejemplo podría ser un vector denominado NOTA:

NOTA[0]	NOTA[1]	NOTA[2]	NOTA[3]	.....	NOTA[k]	.....	NOTA[30]
75	68	51	49	.....	68	.....	100
0	1	2	3	.....	k	.....	n

Un vector es un array unidimensional, es decir, sólo utiliza un índice para referenciar a cada uno de los elementos.

Declaración

En Diagrama de Flujo



En Lenguaje C

tipo_datos **nombre_vector**[*tamaño*]

Donde:

tipo_datos	:	declara el tipo de los elementos del arreglo (array)
nombre_vector	:	indica el nombre de la variable que representa al vector
tamaño	:	indica cuantos elementos podrá almacenar el vector

Ejemplos

1. ***int*** **edad**[*20*]; /* vector edad almacenará 20 valores enteros */
2. ***float*** **precios**[*50*]; /* vector precios almacenará 50 valores reales */
3. ***long*** **fono**[*10*]; /* vector fono almacenará 10 valores entero largos */
4. ***char*** **nomb**[*41*]; /* vector nomb almacenará 40 caracteres */

También, podemos inicializar (asignarle valores) un vector en el momento de declararlo. Si lo hacemos así no es necesario indicar el tamaño. Su sintaxis es:

tipo **nombre [] = {** valor 1, valor 2, ……**};**

Ejemplos:

1. ***int*** **vector**[]={1, 2, 3, 4, 5, 6, 7, 8};
2. ***char*** **vector**[]="programador";
3. ***char*** **vector**[]={'p', 'r', 'o', 'g', 'r', 'a', 'm', 'a', 'd', 'o', 'r'};

Una particularidad con los vectores de tipo **char** (cadena de caracteres), es que deberemos indicar en que elemento se encuentra el fin de la cadena mediante el caracter nulo (**\0**). Esto no lo controla el compilador, y tendremos que ser nosotros los que insertemos este caracter al final de la cadena.

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

Operaciones con Vectores

Asignación de un Valor

En Diagrama de Flujo	En Lenguaje C
A[4] ← 17	A[4] = 17
A = 81 (0) | 57 (1) | 22 (2) | -3 (3) | 17 (4) | (5)	

Modificación de un Valor

En Diagrama de Flujo	En Lenguaje C
A[0] ← A[3]	A[0] = A[3]

A =

3	57	22	3	17	
0	1	2	3	4	5

Lectura de un Valor

En Diagrama de Flujo	En Lenguaje C
A[5]	scanf("%d", &A[5]); cin << A[5];

A =

-3	57	22	-3	17	1
0	1	2	3	4	5

Escritura de un Valor

En Diagrama de Flujo	En Lenguaje C
A[2]	printf("%d", A[2]); cout >> A[2];

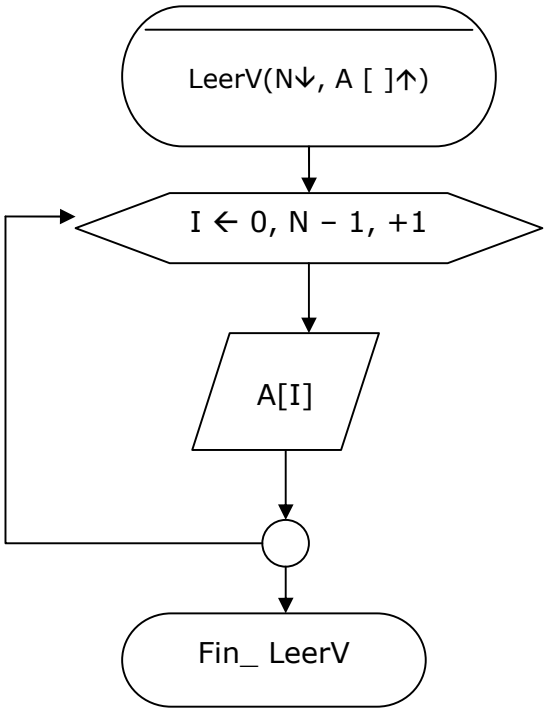
A =

-3	57	22	-3	17	1
0	1	2	3	4	5

Mostrar
22

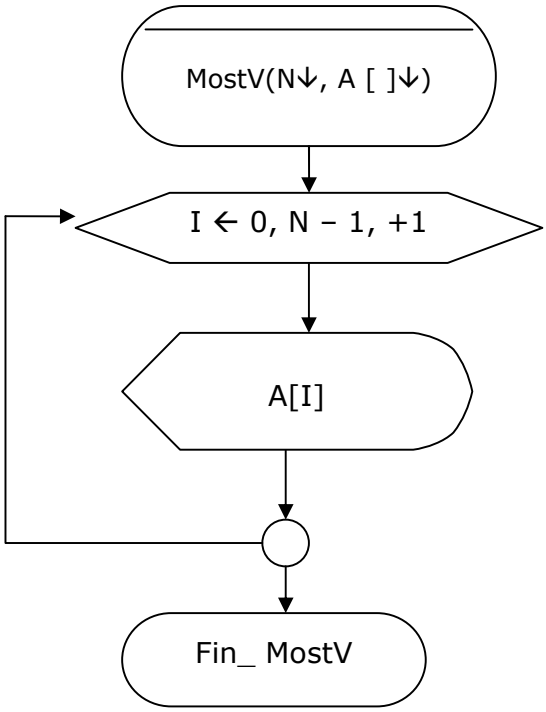
Lectura de elementos del Vector

En Diagrama de Flujo	En Lenguaje C

	/* Procedimiento de lectura de datos, en este caso, el vector ejemplo agrupa al conjunto de valores enteros */
	void LeerV(int n, int a[ ])
	{
	int i;
	
	for(i = 0; i < n; i ++)
	{
	//Mensaje para el usuario
	cout<<"A["<<i<<"] = ";
	//Use scanf ("%d", &a[i]);
	cin >>a[i];
	}
	}

Listado de elementos del Vector

En Diagrama de Flujo	En Lenguaje C

	/* Procedimiento de impresión de valores almacenados en un vector, en este caso, el vector ejemplo agrupa un conjunto de valores enteros */
	void MostV(int n, int a[ ])
	{
	int i;
	//Mensaje para el usuario
	cout<<"Los elementos de V son:\n"
	for (i = 0; i < n; i ++)
	{
	//Mensaje para el usuario
	cout<<"A["<<i<<"] = ";
	//Use printf ("%d, ", a[i]); ó
	cout<<a[i]<<", ";
	}
	}

Ejemplo

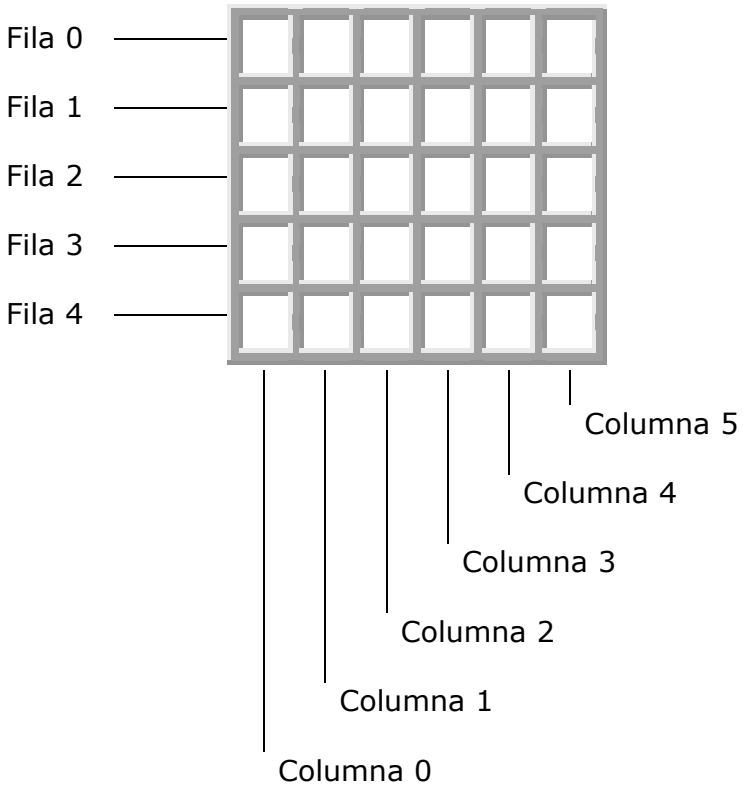
```
/* Programa que almacena los N primeros numeros naturales en un vector */

#include <stdio.h>
#include <conio.h>
#define MAX 100 /*definicion de la constante MAX que indica el tamaño
                  máximo del vector*/

/****************** P R O TO T I P O S ********************/

int LeerDim(void); /* lee la dimension del vector */
void MostrarV(int num[MAX], int n); /* muestra el vector */
```

```
/******* D E F I N I C I O N   D E  S U B P R O G R A M A S *******/

int LeerDim(void)
{
	int n;

	do
	{
	  scanf("%d",&n);
	  if (n<=0)
	  {
			printf("\n\n E  R  R  O  R  ! ! !");
			printf("\n\n Debe introducir un número entero mayor que
				 cero");
			printf("\n\n Intente de nuevo...");
	  }
	  else
	  {
		if (n > MAX)
		{
			printf("\n\n E  R  R  O  R  ! ! !");
			printf("\n\n Debe introducir un número entero menor que
				%d", MAX);
			printf("\n\n Intente de nuevo...");
		}
	  }

	}while (n<=0 || n > MAX);

	return(n);

}
/*******************************************************/
void MostrarV(int num[MAX], int n)
{
	int i;

	for(i = 0; i < n; i = i+1)
	{
		printf(" %d ", num[i]);
	}
}

/********************** M A I N ************************/
int main(void)
{
	int t[MAX]; /* declaracion del vector t de enteros de tamaño MAX */
	int a, i;
```

```
        clrscr();
        printf("Cuantos elementos desea generar?? ");
        a=LeerDim();
        for(i=0; i<a;i=i+1)
        {
          t[i] = i;
        }
        printf("\n\nLos primeros %d numeros naturales son:\n\n", a);
        MostrarV(t, a);
        printf("\n\n");
        getch();
        return(0);
}
```

Arreglos Bidimensionales (Matrices)

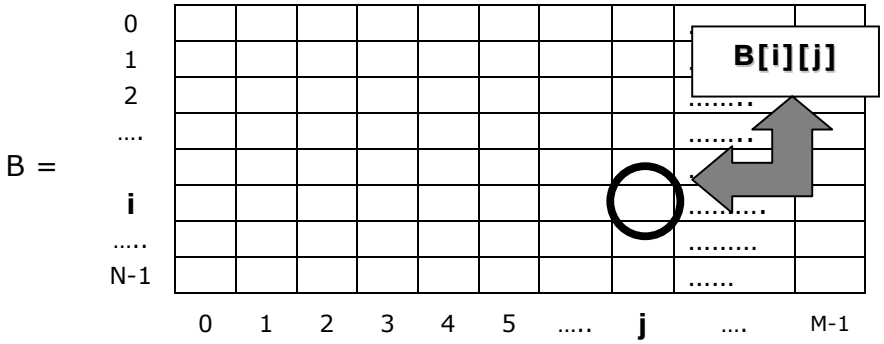
Hasta ahora, todos los arreglos que se han considerado han sido unidimensionales de modo que bastaba un sólo índice para especificar un elemento deseado. Cuando la información se acomoda de manera natural en una disposición rectangular o cuadrada, es decir en una tabla de valores, puede ser ventajoso almacenarla en un arreglo bidimensional, también llamado "matriz".



Una matriz puede considerarse como un vector de vectores. Es, por consiguiente, un conjunto de elementos, todos del mismo tipo, en el cual el orden de los componentes es significativo y se necesitan dos índices para especificar uno de sus elementos, un subíndice de renglón o fila y un subíndice de columna.

Un array bidimensional, también denominado *matriz* o *tabla*, tiene dos dimensiones (una dimensión por cada subíndice) y necesita un valor para cada subíndice y poder identificar a cada uno de sus elementos. El primer subíndice se refiere a la fila de la matriz, mientras que el segundo subíndice se refiere a la columna.

Sea B una matriz de **N** filas y **M** columnas:



el elemento B[i, j] es aquel que ocupa la i-ésima fila y la j-ésima columna.

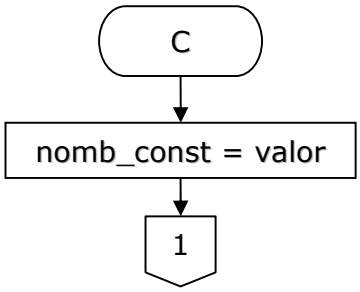
Donde i = 0..N o bien 0 <= i <= N - 1

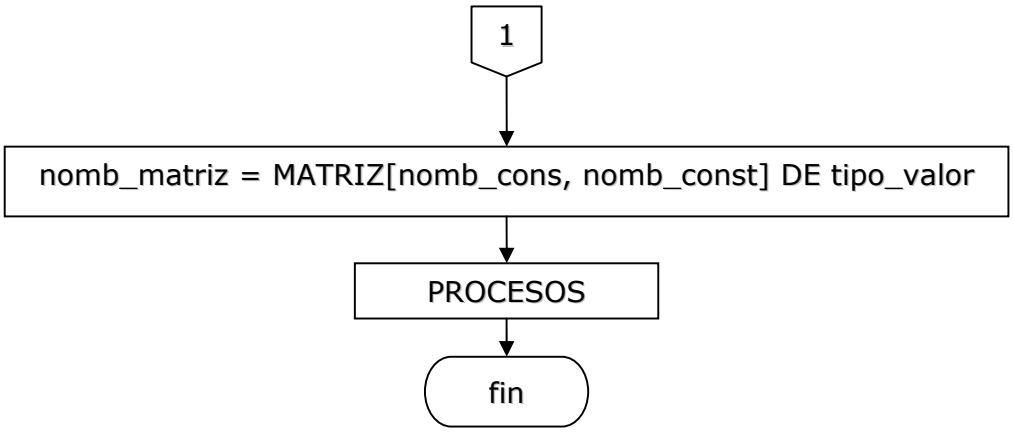
j = 0..M o bien 0 <= j <= M - 1

El arreglo B se dice que tiene N por M elementos ya que existen M elementos en cada fila y N elementos en cada columna.

Declaración

En Diagrama de Flujo





En Lenguaje C

```
tipo_datos nombre_matriz[tamaño_filas][tamaño_columnas]
```

Donde:

tipo_datos	:	declara el tipo de los elementos del arreglo (array)
nombre_matriz	:	indica el nombre de la variable que representa a la matriz
tamaño_filas	:	indica la cantidad máxima de filas de la matriz
tamaño_columas	:	indica la cantidad máxima de columnas de la matriz

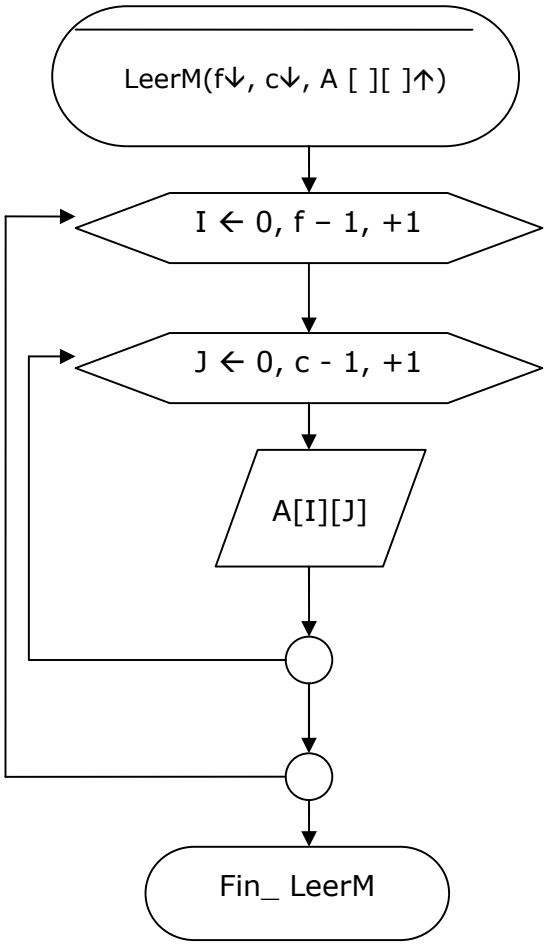
Ejemplos

1. ***int*** **notas**[*6*][50];
2. ***float*** **ventas**[*10*][12];
3. ***int*** **miss**[*10*][10];

Operaciones con Matrices

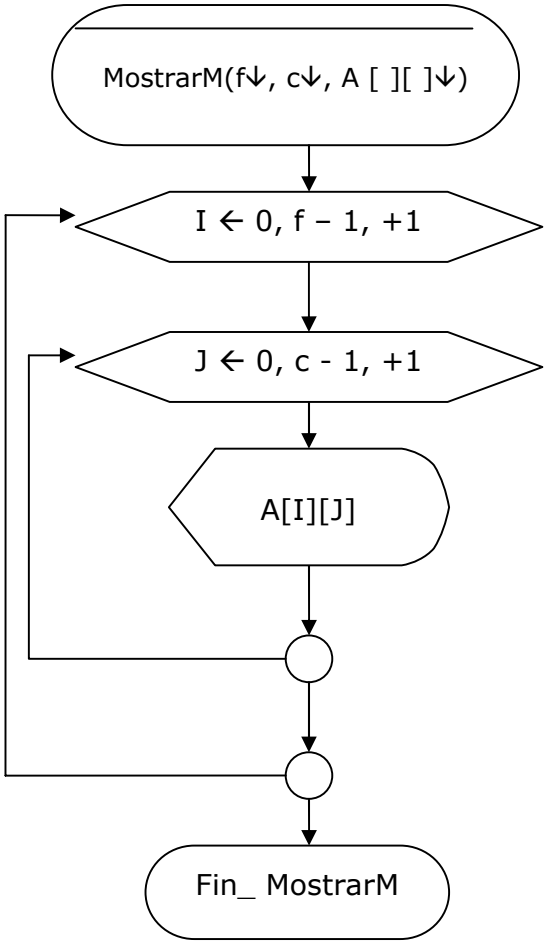
Lectura de elementos de la Matriz

En Diagrama de Flujo	En Lenguaje C

	/* Procedimiento de lectura de datos en una matriz, en el ejemplo que seguimos sus datos son enteros */
	
	void LeerM (int f, int c, int a[ ][ ])
	
	{
	
	int i, j;
	
	cout<<"\nLos elementos de la matriz son:\n";
	
	for (i = 0; i <= f - 1; i ++)
	
	for (j = 0; j <= c - 1; j ++)
	
	{
	
	cout<<"M["<< i <<"] ["<< j <<"]= ";
	
	// Use scanf("%d", &a[i][j]);
	// ó
	cin>> mat[i] [j];
	
	}
	
	}

Listado de elementos de la Matriz

En Diagrama de Flujo	En Lenguaje C

	/* Procedimiento que imprime los elementos que tiene una matriz, en nuestro ejemplo, la matriz de números enteros imprime sus valores */
	
	void MostrarM (int f, int c, int a[ ][ ])
	{
	int i, j;
	
	for(i = 0; i <= f - 1; i ++)
	{
	for(j = 0; j <= c – 1; j ++)
	{
	cout<<mat [i] [j] << " ";
	}
	cout <<"\n"; //Dando un enter
	}
	}

Ejemplo

```
/* Programa que lee y muestra los elementos de una matriz */
#include <stdio.h>
#include <conio.h>
#define MAX 10 /* definicion de la constante MAX que indica el tamanio
                  máximo del vector */

/***************** P R O TO T I P O S ******************/

int LeerDim(void); /* lee una de las dimensiones de la matriz */
void LeerM(int n, int m, int mat[MAX][MAX]); /* carga la matriz */
void MostrarM(int n, int m, int num[MAX][MAX]); /* muestra la matriz */
void Pausa(void);

/******* D E F I N I C I O N   D E  S U B P R O G R A M A S *******/

int LeerDim(void)
{
      int n;

      do
      {
        scanf("%d",&n);
        if (n<=0)
        {
                printf("\n\n E  R  R  O  R  ! ! !");
                printf("\n\n Debe introducir un numero entero mayor que cero");
                printf("\n\n Intente de nuevo...");
        }
        else
        {
              if (n > MAX)
              {
                    printf("\n\n E  R  R  O  R  ! ! !");
                    printf("\n\n Debe introducir un numero entero menor que %d",
                              MAX);
                    printf("\n\n Intente de nuevo...");
              }
        }

      }while (n<=0 || n > MAX);

      return(n);
}
```

```
/*******************************************************/
void MostrarM(int n, int m, int num[MAX][MAX])
{
	int i, j;

	for(i = 0; i < n; i = i+1)
	{
		for(j = 0; j < m; j++)
		{
			printf("%5d", num[i][j]);
		}
		printf("\n");
	}
}
/*******************************************************/
void LeerM(int n, int m, int mat[MAX][MAX])
{
	int i,j;

	for(i = 0; i < n; i = i+1)
	{
		for(j = 0; j < m; j++)
		{
			printf("\n\nT[%d][%d] = ", i, j);
			scanf("%d", &mat[i][j]);
		}
	}

}
/*******************************************************/
void Pausa(void)
{
  printf("Presione cualquier tecla para continuar...");
  getch();
}
/************************ M A I N ************************/
int main(void)
{
	int t[MAX][MAX]; /* declaracion de la matriz t de tamaño MAX*MAX */
	int a, b;

	clrscr();
	printf("Ingrese la cantidad de filas de la matriz...");
	a=LeerDim();
	printf("Ingrese la cantidad de columnas de la matriz...");
	b=LeerDim();
	printf("\n\n Introduzca los elementos de la matriz...");
	LeerM(a, b, t);
```

```
        printf("\n\nLos elementos de la matriz son:\n");
        MostrarM(a, b, t);
        printf("\n\n");
        Pausa();

        return(0);
}
```

Si al declarar una matriz también queremos inicializarla, habrá que tener en cuenta el orden en el que los valores son asignados a los elementos de la matriz. Veamos algunos ejemplos:

```
int numeros[3][4]={1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12};
```

Quedarían asignados de la siguiente manera:

numeros[0][0]=1 numeros[0][1]=2 numeros[0][2]=3 numeros[0][3]=4
numeros[1][0]=5 numeros[1][1]=6 numeros[1][2]=7 numeros[1][3]=8
numeros[2][0]=9 numeros[2][1]=10 numeros[2][2]=11 numeros[2][3]=12

numeros =

	0	1	2	3
0	1	2	3	4
1	5	6	7	8
2	9	10	11	12

También se pueden inicializar cadenas de texto:

```
char dias[7][10]={"lunes","martes","miércoles","jueves","viernes",
                  "sábado","domingo"};
```

Para referirnos a cada palabra bastaría con el primer índice:

```
printf("%s",dias[i]);
```

METODOS DE ORDENACIÓN

Introducción

Uno de los procedimientos más comunes y útiles en el procesamiento de datos, es la clasificación u ordenación de los mismos. Se considera ordenar al proceso de reorganizar un conjunto dado de objetos en una secuencia determinada. Cuando se analiza un método de ordenación, hay que determinar cuántas comparaciones e intercambios se realizan para el caso más favorable, para el caso medio y para el caso más desfavorable.

La colocación en orden de una lista de valores se llama **Ordenación**. Por ejemplo, se podría disponer una lista de valores numéricos en orden ascendente o descendente, o bien una lista de nombres en orden alfabético. La localización de un elemento de una lista se llama búsqueda.

Tal operación se puede hacer de manera más eficiente después de que la lista ha sido ordenada. Existen varios métodos para ordenamiento, clasificados en tres formas:

- Intercambio
- Selección
- Inserción.

En cada familia se distinguen dos versiones: un método simple y directo, fácil de comprender pero de escasa eficiencia respecto al tiempo de ejecución, y un método rápido, más sofisticado en su ejecución por la complejidad de las operaciones a realizar, pero mucho más eficiente en cuanto a tiempo de ejecución. En general, para arreglos con pocos elementos, los métodos directos son más eficientes (menor tiempo de ejecución) mientras que para grandes cantidades de datos se deben emplear los llamados métodos rápidos.

Método de Intercambio o de la Burbuja

El método de intercambio se basa en comparar los elementos del arreglo e intercambiarlos si su posición actual o inicial es contraria inversa a la deseada. Aunque no es muy eficiente para ordenar listas grandes, es fácil de entender y muy adecuado para ordenar una pequeña lista de unos 100 elementos o menos.

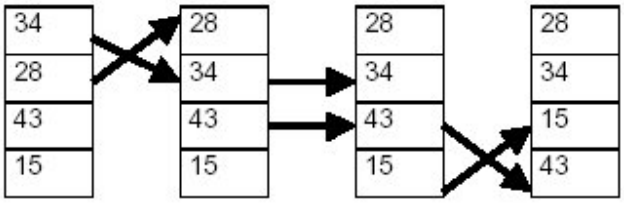
Una pasada por la ordenación de burbujeo consiste en un recorrido completo a través del arreglo, en el que se comparan los contenidos de las casillas adyacentes, y se cambian si no están en orden. La ordenación por burbujeo completa consiste en una serie de pasadas ("burbujeo") que termina con una en la que ya no se hacen cambios porque todo está en orden.

Ejemplo:

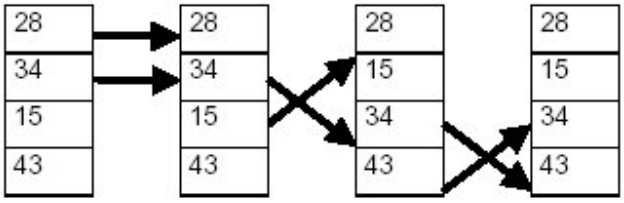
Supóngase que están almacenados cuatro números en un arreglo con casillas de memoria de x[0] a x[3]. Se desea disponer esos números en orden creciente. La primera pasada de la ordenación por burbujeo haría lo siguiente:

- Comparar el contenido de x[0] con el de x[1]; si x[0] contiene el mayor de los números, se intercambian sus contenidos.
- Comparar el contenido de x[1] con el de x[2]; e intercambiarlos si fuera necesario.
- Comparar el contenido de x[2] con el de x[3]; e intercambiarlos si fuera necesario.

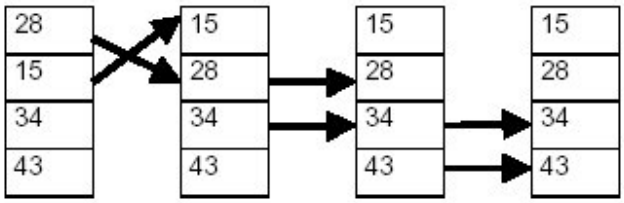
Al final de la primera pasada, el mayor de los números estará en x[3].



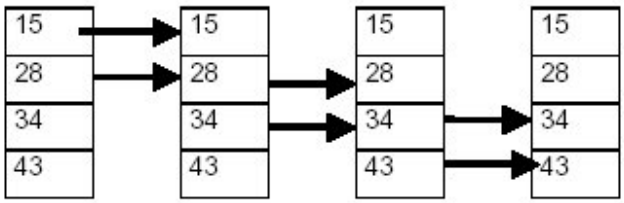
Al final de la segunda pasada, los últimos dos elementos estarán ordenados como se muestra:



Al final de la tercera pasada, los últimos tres elementos estarán ordenados como se muestra:



Nótese que el arreglo ya quedó ordenado, sin embargo, se hace una pasada más porque la computadora no advierte que el arreglo está en orden hasta que ocurre una pasada sin intercambios.



La ordenación por burbujeo se llama así porque los números más pequeños ascienden como burbujas hasta la parte superior, mientras que los mayores se hunden y caen hasta el fondo. Está garantizado que cada pasada pone el siguiente número más grande en su lugar, aunque pueden colocarse más de ellos en su lugar por casualidad.

El método expresado en pseudocódigo es:

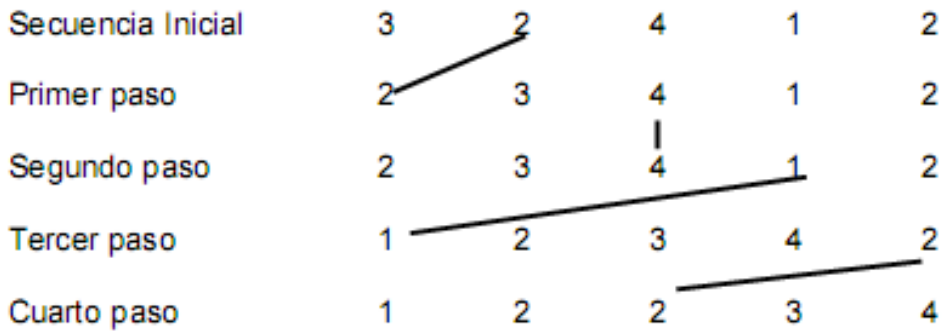
```
Desde I ← 1 hasta n - 1 hacer
        Desde J ← 0 hasta n - 2 hacer
                Si elemento [J] > elemento [J + 1]
                Entonces
                        aux ← elemento [J]
                        elemento [J] ← elemento [J + 1]
                        elemento [J + 1] ← aux
                Fin_si
        Fin_desde
Fin_desde
```

Método de Ordenación por Inserción

El fundamento de este método consiste en insertar los elementos no ordenados del arreglo en subarreglos del mismo que ya estén ordenados. Por ser utilizado generalmente por los jugadores de cartas se lo conoce también por el nombre de *método de la baraja.*

Dependiendo del método elegido para encontrar la posición de inserción tendremos distintas versiones del método de inserción.

Ejemplo



El método expresado en pseudocódigo es:

```
Desde I ← 1 hasta n - 1 hacer
        aux ← elemento[I]
        k ← I - 1
        sw ← 0
        Mientras (sw = 0) Λ (k >= 0) hacer
                Si aux < elemento [k]
                Entonces
                        elemento [k + 1] ← elemento [k]
                        k ← k – 1
                Si_no
```

```
                    sw ← 1

                Fin_si

            Fin_mientras

            elemento [k+1] ← aux

Fin_desde
```

Método de Ordenación por Selección

Los métodos de ordenación por selección se basan en dos principios básicos:

- Seleccionar el elemento más pequeño (o más grande) del arreglo.
- Colocarlo en la posición más baja (o más alta) del arreglo.

A diferencia del método de la burbuja, en este método el elemento más pequeño (o más grande) es el que se coloca en la posición final que le corresponde.

Ejemplo

Secuencia Inicial	3	2	4	1	2
Primer paso	1	2	4	3	2
Segundo paso	1	2	4	3	2
Tercer paso	1	2	2	3	4
Cuarto paso	1	2	2	3	4

El método expresado en pseudocódigo es:

```
Desde I  ← 0 hasta n - 2 hacer

        aux ← elemento [I]

        k ← I

        Desde J ← I + 1 hasta n - 1 hacer

                Si elemento [J] < aux

                Entonces
```

```
                aux ← elemento [J]

                k ← j

            Fin_si

        Fin_desde

        elemento [k] ← elemento [I]

        elemento [I] ← aux

Fin_desde
```

Método de Ordenación Rápida (Quicksort)

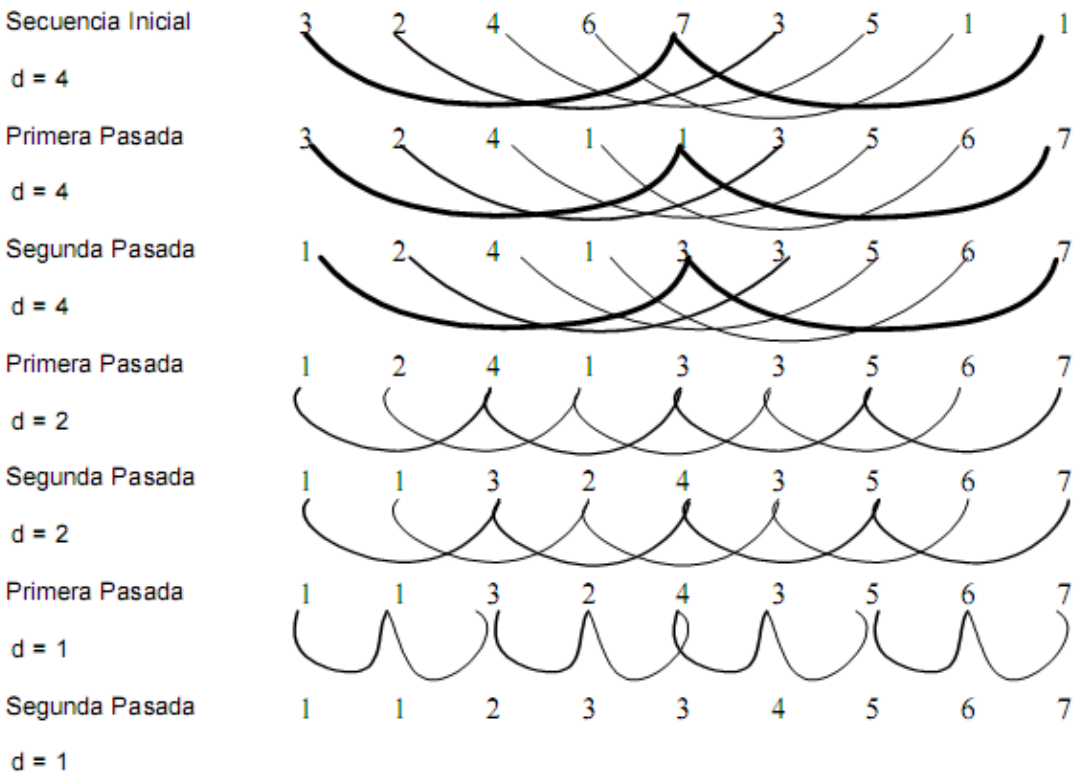
Si bien el método de la burbuja era considerado como el peor método de ordenación simple o menos eficiente, el método Quicksort basa su estrategia en la idea intuitiva de que es más fácil ordenar una gran estructura de datos subdividiéndolas en otras más pequeñas introduciendo un orden relativo entre ellas. En otras palabras, si dividimos el array a ordenar en dos subarrays de forma que los elementos del subarray inferior sean más pequeños que los del subarray superior, y aplicamos el método reiteradamente, al final tendremos el array inicial totalmente ordenado.

Método de Ordenación Shell

El método burbuja acerca cada elemento a la posición que le corresponde paso a paso, como se acaba de explicar. Sin embargo, su tiempo de ejecución para vectores pequeños puede ser aceptable, no así para los arreglos con muchos elementos; para eso surge este método que agiliza la ordenación en grandes vectores, y se denomina Shell en honor a su inventor.

Este método consiste en comparar, no elementos consecutivos como lo hacía el método de la burbuja, sino los que están separados por un intervalo grande, de modo que cada uno se acerca al lugar que le corresponde de forma más rápida. Inicialmente, ese intervalo o salto corresponde con la mitad de la longitud del vector. Posteriormente, cuando ya todos los elementos separados por esa distancia están ordenados, se divide ese intervalo por dos y se opera de igual forma hasta que, finalmente, el salto se reduzca a 1. Entonces, el proceso de ordenación funciona exactamente como el de la burbuja y realiza la comparación de los elementos adyacentes.

Ejemplo



Los elementos a utilizar en el algoritmo son los siguientes:

d: es una variable que indica la distancia de comparación.
sw: comprueba la ordenación a distancia d.
1 = desordenado a distancia d.
0 = ordenado a distancia d.

Y el pseudocódigo correspondiente es el siguiente:

```
d = n
Mientras d<>1
        d = |d/2|
        sw = 1
        Mientras sw = 1
                sw = 0
                Para (i = 1, i <= n – d, i++)
                        Si (v[j] > v[i + d])
                                aux = v[i]
                                v[i] = v[i + d]
                                v[i + d] = aux
                                sw = 1
                        FinSi
```

```
                FinPara
            FinMientras
FinMientras
```

MÉTODOS DE BÚSQUEDA

Introducción

La búsqueda es una operación que tiene por objeto la localización de un elemento dentro de la estructura de datos. A menudo un programador estará trabajando con grandes cantidades de datos almacenados en arreglos y pudiera resultar necesario determinar si un arreglo contiene un valor que coincide con algún valor clave o buscado.

Siendo el array de una dimensión o lista una estructura de acceso directo y a su vez de acceso secuencial, encontramos dos técnicas que utilizan estos dos métodos de acceso, para encontrar elementos dentro de un array: búsqueda lineal y búsqueda binaria.

Método de Búsqueda Secuencial

La búsqueda secuencial es la técnica más simple para buscar un elemento en un arreglo. Consiste en recorrer el arreglo elemento a elemento e ir comparando con el valor buscado (clave). Se empieza con la primera casilla del arreglo y se observa una casilla tras otra hasta que se encuentra el elemento buscado o se han visto todas las casillas. El resultado de la búsqueda es un solo valor, y será la posición del elemento buscado o cero. Dado que el arreglo no está en ningún orden en particular, existe la misma probabilidad de que el valor se encuentra ya sea en el primer elemento, como en el último. Por lo tanto, en promedio, el programa tendrá que comparar el valor buscado con la mitad de los elementos del arreglo.

El método de búsqueda lineal funciona bien con arreglos pequeños o para arreglos no ordenados. Si el arreglo está ordenado, se puede utilizar la técnica de alta velocidad de búsqueda binaria, donde se reduce sucesivamente la operación eliminando repetidas veces la mitad de la lista restante.

El método expresado en pseudocódigo sería:

```
ind ← 0
sw ← 0
mientras (sw = 0) Λ (ind < N) hacer
        si arreglo[ind] = valor_buscado
        entonces
                sw ← 1
        si_no
                ind ← ind + 1
```

```
        fin_si
fin_mientras
si sw = 1
entonces
        Mostrar "Encontrado"
si_no
        Mostrar "No encontrado"
fin_si
```

Método de Búsqueda Binaria

La búsqueda binaria es el método más eficiente para encontrar elementos en un arreglo ordenado. El proceso comienza comparando el elemento central del arreglo con el valor buscado. Si ambos coinciden finaliza la búsqueda. Si no ocurre así, el elemento buscado será mayor o menor en sentido estricto que el central del arreglo. Si el elemento buscado es mayor se procede a hacer búsqueda binaria en el subarray superior, si el elemento buscado es menor que el contenido de la casilla central, se debe cambiar el segmento a considerar al segmento que está a la izquierda de tal sitio central.

El método expresado en pseudocódigo sería:

```
sw  ← 0
primero ← 0
ultimo ← N - 1
mientras (primero <= ultimo) Λ  (sw = 0) hacer
        mitad ← (primero + ultimo)/2
        si arreglo[mitad] = valor_buscado
        entonces
                sw ← 1
        si_no
                si arreglo[mitad] > valor_buscado
                entonces
                        ultimo ← mitad - 1
                si_no
                        primero ← mitad + 1
                fin_si
        fin_si
fin_mientras

si sw = 1
entonces
        Mostrar "Encontrado"
si_no
        Mostrar "No encontrado"
fin_si
```

Método de Búsqueda Mediante Transformaciones de Claves (Hashing)

La idea principal de este método consiste en aplicar una función que traduce el valor del elemento buscado en un rango de direcciones relativas. Una desventaja importante de este método es que puede ocasionar colisiones.

El pseudocódigo correspondiente al método sería:

```
funcion hash (valor_buscado)
        inicio
                hash ← valor_buscado mod numero_primo
        fin

inicio ← hash (valor)
il ← inicio
sw ← 0
repetir
        si arreglo[il] = valor
        entonces
                sw ← 1
        si_no
                il ← (il +1) mod N
        fin_si
hasta (sw = 1) V (il = inicio)
si sw = 1
entonces
      Mostrar "Encontrado"
si_no
      Mostrar "No encontrado"
fin_si
```

CAPITULO IV

ARCHIVOS

Introducción

El almacenamiento de datos en las estructuras de datos, vistas anteriormente, sólo es temporal, es decir, cuando termina el programa los datos se pierden. Para la conservación permanente de grandes cantidades de datos se utilizan los archivos. Las computadoras almacenan los archivos en dispositivos de almacenamiento secundario, especialmente en dispositivos de almacenamiento como discos.

Cada archivo termina con un marcador de fin de archivo o en un número de bytes específicos registrado en una estructura de datos, mantenida por el sistema.

En C y C++ podemos clasificar los archivos según varias categorías:

1. Dependiendo de la dirección del flujo de datos:
   - De entrada: los datos se leen por el programa desde el archivo.
   - De salida: los datos se escriben por el programa hacia el archivo.
   - De entrada/salida: los datos pueden se escritos o leídos.
2. Dependiendo del tipo de valores permitidos a cada byte:
   - De texto: sólo están permitidos ciertos rangos de valores para cada byte. Algunos bytes tienen un significado especial, por ejemplo, el valor hexadecimal 0x1A marca el fin de fichero. Si abrimos un archivo en modo texto, no será posible leer más allá de ese byte, aunque el fichero sea más largo.
   - Binarios: están permitidos todos lo valores para cada byte. En estos archivos el final del fichero se detecta de otro modo, dependiendo del soporte y del sistema operativo. La mayoría de las veces se hace guardando la longitud del fichero. Cuando queramos almacenar valores enteros, o en coma flotante, o imágenes, etc, deberemos usar este tipo de archivos.
3. Según el tipo de acceso:
   - Archivos secuenciales: imitan el modo de acceso de los antiguos ficheros secuenciales almacenados en cintas magnéticas y
   - Archivos de acceso aleatorio: permiten acceder a cualquier punto de ellos para realizar lecturas y/o escrituras.
4. Según la longitud de registro:
   - Longitud variable: en realidad, en este tipo de archivos no tiene sentido hablar de longitud de registro, podemos considerar cada byte como un registro. También puede suceder que nuestra aplicación conozca el tipo y longitud de cada dato almacenado en el archivo, y lea o escriba los bytes necesarios en cada ocasión. Otro caso es cuando se usa una marca para el final de registro, por ejemplo, en ficheros de texto se usa el carácter de

retorno de línea para eso. En estos casos cada registro es de longitud diferente.

- Longitud constante: en estos archivos los datos se almacenan en forma de registro de tamaño contante. En C usaremos estructuras para definir los registros. C dispone de funciones de librería adecuadas para manejar este tipo de ficheros.
- Mixtos: en ocasiones pueden crearse archivos que combinen los dos tipos de registros, por ejemplo, dBASE usa registros de longitud constante, pero añade un registro especial de cabecera al principio para definir, entre otras cosas, el tamañ y el tipo de los registros.

Es posible crear archivos combinando cada una de estas categorías, por ejemplo: archivos secuenciales de texto de longitud de registro variable, que son los típicos archivos de texto. Archivos de acceso aleatorio binarios de longitud de registro constante, normalmente usados en bases de datos. Y también cualquier combinación menos corriente, como archivos secuenciales binarios de longitud de registro constante, etc.

ESTRUCTURAS

Concepto de estructura

Una estructura es un conjunto de una o más variables, de distinto tipo, agrupadas bajo un mismo nombre para que su manejo sea más sencillo.

Declaración

```
struct tipo_estructura
{
        tipo_campo1  nombre_campo1;
        tipo_campo2  nombre_campo2;
        tipo_campo3  nombre_campo3;
};
```

Donde:

tipo_estructura es el nombre del nuevo tipo de dato que hemos creado

tipo_campo y **nombre_campo** son las variables que forman parte de la estructura.

Ejemplos

1. **struct** reglib
   {

```
    char titlib[40];
    char autlib[30];
    int npag;
    int aned;
};
```

2.
```
struct regalu
{
    char nomb[30];
    char ape[30];
    int edad;
    char fecnac[8];
    int ci;
};
```

Definición de variables de tipo registro

Para definir variables del tipo que acabamos de crear lo podemos hacer de varias maneras, aunque las dos más utilizadas son estas:

- Una forma de definir la estructura:

```
/******** DEFINICION DEL TIPO TRABAJADOR *****/

struct trabajador
{
        char nombre[20];
        char apellidos[40];
        int edad;
        char puesto[10];
};

/*** DECLARACION DE LAS VARIABLES DE TIPO TRABAJADOR ***/

struct trabajador fijo, temporal;
```

- Otra forma:

```
/****DEFINICION DEL TIPO TRABAJADOR Y DECLARACION DE LAS VARIABLES
DE TIPO TRABAJADOR  ********/

        struct trabajador
        {
                char nombre[20];
                char apellidos[40];
                int edad;
                char puesto[10];
```

```
} fijo, temporal;
```

En el primer caso declaramos la estructura, y luego declaramos las variables. En el segundo las declaramos al mismo tiempo que la estructura. El problema del segundo método es que no podremos declarar más variables de este tipo a lo largo del programa. Para poder declarar una variable de tipo estructura, la estructura tiene que estar declarada previamente. Se debe declarar antes de la función **main**.

Ejemplos

1.
```
struct reglib
{
    char titlib[40];
    char autlib[30];
    int npag;
    int aned;
} libro;
```

2.
```
struct regalu
{
    char nomb[30];
    char ap[30];
    int edad;
    char fecnac[8];
    int ci;
};

struct alumno;
```

Manejo de las estructuras

Para acceder a los elementos individuales de la estructura se utiliza el operador **.** (operador punto) de la siguiente forma:

```
nomb_registro.nomb_campo
```

Ejemplos

1. Sea la variable libro de tipo reglib del ejemplo anterior, podemos asignarle valores a algunos de sus campos de la siguiente forma:

```
libro.npag = 200;
libro.aned = 1998;
fgets(libro.titlib, 40, stdin);
fgets(libro.autlib, 30, stdin);
```

2. Sea la variable alumno de tipo regal del ejemplo anterior, podemos asignarle valores a algunos de sus campos de la siguiente forma:

```
alumno.edad = 18;
alumno.ci = 5887456;
fgets(alumno.nomb, 30, stdin);
fgets(alumno.ap, 30, stdin);
```

3. Si queremos mostrar en la pantalla el valor de uno de los campos:
   printf("%d", libro.npag);

4. Ejemplo completo de manejo de estructuras :

```
/*************************************************************
Programa que pemite la introduccion de un registro y luego muestra
su contenido por pantalla
*************************************************************/

#include <stdio.h>
#include <conio.h>
#include "lineas.c"

/*************** DEFINICION DEL REGISTRO *****************/

struct reglib
{
        char titlib[50];
        char autlib[40];
        char edlib[30];
        int aned;
        int npag;
}libro;

/****************** PROTOTIPOS ********************/

char Menu(void);
void LeerR(void);
void MostrarR(void);

/*************** DEFINICION DE SUBPROGRAMAS **************/

char Menu(void)
{
         char resp;
         clrscr();
         Margen(29, 5, 50, 7, 2);
         Wxy(33,6, "MENU PRINCIPAL");
```

```
        Margen(20, 8, 60, 15, 2);
        Wxy(28, 9, "[L]eer el registro");
        Wxy(28, 11, "[M]ostrar el registro");
        Wxy(28,13, "[S]alir");
        Margen(25, 16, 55, 18, 1);
        Wxy(30, 17, "Digite su opcion... ");
        gotoxy(51,17);
        resp=(char)toupper(getch());

        return resp;
}
/*********************************************************/

void LeerR(void)
{
        int i;

        clrscr();
        Margen(28, 1, 52, 3, 2);
        Wxy(30, 2, "LECTURA DEL REGISTRO");

        /* TITULO */
        Wxy(1, 5,"Titulo del Libro : ");
        fgets(libro.titlib, 50, stdin);
        fflush(stdin);

        for (i = strlen(libro.titlib)-1; i && libro.titlib[1] < ''; i--)
                libro.titlib[i] = 0;

        /* AUTOR */
        Wxy(1, 7, "Autor del Libro : ");
        fgets(libro.autlib, 40, stdin);
        fflush(stdin);

        for (i = strlen(libro.autlib)-1; i && libro.autlib[1] < ''; i--)
                libro.autlib[i] = 0;

        /* EDITORIAL */
        Wxy(1, 9,"Editorial del Libro : ");
        fgets(libro.edlib, 30, stdin);
        fflush(stdin);

        for (i = strlen(libro.edlib)-1; i && libro.edlib[1] < ''; i--)
                libro.edlib[i] = 0;

        /* ANIO DE EDICION */
        Wxy(1, 11,"Anio de Edicion : ");
        scanf("%d", &libro.aned);
```

```
	/* CANTIDAD DE PAGINAS */
	Wxy(1, 13,"Cantidad de Paginas : ");
	scanf("%d", &libro.npag);

}
/*********************************************************/

void MostrarR(void)
{

	clrscr();
	Margen(1, 3, 79, 20, 2);

	Wxy(2, 9,"TITULO");
	gotoxy(24, 9);
	printf(": %s", libro.titlib);

	Wxy(2, 11, "AUTOR");
	gotoxy(24, 11);
	printf(": %s", libro.autlib);

	Wxy(2, 13, "EDITORIAL");
	gotoxy(24,13);
	printf(": %s", libro.edlib);

	Wxy(2, 15, "ANIO DE EDICION");
	gotoxy(24,15);
	printf(": %d", libro.aned);

	Wxy(2, 17,"CANTIDAD DE PAGINAS");
	gotoxy(24, 17);
	printf(": %d", libro.npag);
	Mensaje();

}

/******************* MAIN *********************/

int main(void)
{
	 char opcion = '0';

	 do
	 {
		 opcion = Menu();
		 switch(opcion)
		{
```

```
                    case 'L' : /* Lectura del registro */
                                        LeerR();
                                        break;
                    case 'M'  : /* Mostrar el registro */
                                        MostrarR();
                                        break;
            } /*fin switch*/
        }while(opcion != 'S');

   return 0;
}
```

Algunas funciones relacionadas con el manejo de archivos

Para poder trabajar con los archivos tenemos que tomar en cuenta algunas funciones que se encuentran en el archivo de cabecera **stdio.h:**

Función	Descripción
fopen()	Abre un archivo
fclose()	Cierra un archivo
putc	Escribe un carácter en un archivo
getc()	Lee un carácter desde un archivo
fputs()	Escribe una cadena en un archivo
fgets()	Obtiene una cadena de un archivo
fseek()	Salta al byte especificado en un archivo
fprintf()	Imprime datos con formato en un archivo
fscanf()	Lee datos con formato en un archivo
eof()	Devuelve verdadero o falso si se halla el fin del archivo.

Declaración

La declaración de archivos se hace de la siguiente forma:

```
FILE *nomb_archivo;
```

Donde:

FILE	**:**	estructura de datos definida en el archivo de cabecera stdio.h
nomb_archivo	**:**	puntero al archivo con el que se trabajará en el programa

Apertura

Antes de abrir un fichero se necesita declarar un puntero de tipo **FILE**, con el que se trabajará durante todo el proceso. Para abrir el fichero se utilizará la función **fopen( )**.

Su **sintaxis** es:

```
nomb_archivo = fopen ( "nomb_archivo_en_disco", "modo de apertura" );
```

Donde

nomb_archivo	:	Es la variable de tipo FILE
nomb_archivo_en_disco	:	Es el nombre que se le dará al archivo que se desea crear o abrir. Este nombre debe ir encerrado entre comillas. También se puede especificar la ruta donde se encuentra.
modo de apertura	:	Le indica al C el modo en que se abrirá o creará el archivo.

Ejemplos:

1. **FILE** *archlib;

   archlib = fopen("LIBROS.DAT","r");

2. **FILE** *archal;

   archal = fopen("C:\\datos\\ALUMNOS.DAT","w");

3. **FILE** *saltxt

```
saltxt = fopen("c:\\txt\\saludos.txt", "w");
```

MODOS DE APERTURA

Un archivo puede ser abierto en dos modos diferentes, en modo texto o en modo binario. A continuación se verá con más detalle.

Modo texto

Acceso	Descripción
"r"	Abre un archivo para lectura. El archivo debe de existir
"w"	Abre un archivo para escritura. Si el archivo no existe se crea, pero si existe se borra toda su información para crearlo de nuevo.
"a"	Abre un archivo para escribir al final. Si el archivo no existe se crea.
"r+"	Abre para Lectura/ Escritura. El archivo debe existir.
"w+"	Abre para Lectura/ Escritura. Si el archivo no existe se crea, pero si existe su información se destruye para crearlo de nuevo.
"a+"	Abre para leer y escribir al final del archivo. Si el archivo no existe se crea.

Modo binario

Acceso	Descripción
"rb"	Abre un archivo binario para lectura. El archivo debe existir.
"wb"	Abre un archivo binario para escritura. Si el archivo no existe se crea, pero si existe su información se destruye para crearlo de nuevo.
"ab"	Se abre un archivo binario para escribir al final de él. Si el archivo no existe se crea.
"rb+"	Abre un archivo binario para lectura/escritura. El archivo debe existir
"wb+"	Abre un archivo binario para lectura/escritura. Si el archivo no existe se crea, pero si existe su información se destruye para crearlo de nuevo.
"ab+"	Abre un archivo binario para lectura y escribir al final. Si el archivo no existe se crea.

La función **fopen** devuelve, como ya se ha visto, un puntero de tipo **FILE**. Si al intentar abrir el fichero se produce un error (por ejemplo si no existe y lo estamos abriendo en modo lectura), la función **fopen** devolvería **NULL**. Por esta razón es mejor controlar las posibles causas de error a la hora de programar. Un ejemplo:

```
FILE *pf;
pf = fopen("datos.txt","r");
if (pf == NULL)
        printf("Error al abrir el fichero");
```

Cierre

Una vez que se ha acabado de trabajar con el archivo es recomendable cerrarlo. Los archivos se cierran al finalizar el programa pero el número de estos que pueden estar abiertos es limitado. Para cerrarlos utilizaremos la función **fclose( )**.

Su sintaxis es:

```
fclose(nomb_archivo);
```

Donde:

nomb_archivo	:	es la variable de tipo FILE

Esta función cierra el fichero, cuyo puntero le indicamos como parámetro. Si el fichero se cierra con éxito devuelve 0.

Ejemplo

```
FILE *pf;
pf = fopen("AGENDA.DAT","rb");
if ( pf == NULL )
    printf ("Error al abrir el fichero");
else
    fclose(pf);
```

Escritura y lectura

A continuación se verán las funciones que se podrán utilizar dependiendo del dato que se quiera escribir y/o leer en el archivo.

ESCRIBIR UN CARACTER

fputc(variable_caracter , nomb_archivo);

Escribimos un carácter en un archivo (abierto en modo escritura).

Ejemplo

```
FILE *pf;
char letra = 'a';

if (!(pf = fopen("datos.txt","w"))) /* otra forma de controlar si se produce un error */
{
    printf("Error al abrir el archivo");
    exit(0); /* abandonamos el programa */
}
else
    fputc(letra,pf);
fclose(pf);
```

LEER UN CARACTER

fgetc(nomb_archivo);

Lee un caracter de un archivo (abierto en modo lectura). Se deberá guardar en una variable.

Ejemplo

```
FILE *pf;
char letra;

if (!(pf = fopen("datos.txt","r"))) /* controlamos si se produce un error */
{
    printf("Error al abrir el archivo");
    exit(0); /* abandonamos el programa */
}
else
{
    letra = fgetc(pf);
    printf("%c",letra);
    fclose(pf);
}
```

ESCRIBIR UN NÚMERO ENTERO

```
putw( variable_entera, nomb_archivo );
```

Escribe un número entero en formato binario en el archivo.

Ejemplo:

```
FILE *pf;
int num = 3;

if (!(pf = fopen("datos.txt","wb"))) /* controlamos si se produce un error */
{
    printf("Error al abrir el archivo");
    exit(0); /* abandonamos el programa */
}
else
```

```
{
    fputw(num,pf); /* también podíamos haber hecho directamente:
        fputw(3,pf); */
    fclose(pf);
}
```

LEER UN NÚMERO ENTERO

getw(nomb_archivo);

Lee un número entero de un archivo, avanzando dos bytes después de cada lectura.

Ejemplo

```
FILE *pf;
int num;

if (!(pf = fopen("datos.txt","rb"))) /* controlamos si se produce un error */
{
    printf("Error al abrir el archivo");
    exit(0); /* abandonamos el programa */
}

else
{
    num = getw(pf);
    printf("%d",num);
    fclose(pf);
}
```

ESCRIBIR UNA CADENA DE CARACTERES

fputs(variable_cadena, nomb_archivo);

Escribe una cadena de caracteres en el archivo.

Ejemplo

```
FILE *pf;
char cad = "Me llamo Vicente";

if (!(pf = fopen("datos.txt","w"))) /* controlamos si se produce un error */
{
    printf("Error al abrir el archivo");
    exit(0); /* abandonamos el programa */
}
else
{
    fputs(cad,pf); /* o también así: fputs("Me llamo Vicente",pf); */
    fclose(pf);
}
```

- **LEER UNA CADENA DE CARACTERES**

```
fgets( variable_cadena, variable_entera, nomb_archivo );
```

Lee una cadena de caracteres del archivo y la almacena en variable_cadena. La variable_entera indica la longitud máxima de caracteres que puede leer.

Ejemplo

```
FILE *pf;
char cad[80];

if (!(pf = fopen("datos.txt","rb"))) /* controlamos si se produce un error */
{
    printf("Error al abrir el archivo");
    exit(0); /* abandonamos el programa */
}
```

```
else
{
    fgets(cad,80,pf);
    printf("%s",cad);
    fclose(pf);
}
```

- **ESCRIBIR CON FORMATO**

fprintf(nomb_archivo, formato, argumentos);

Funciona igual que un **printf** pero guarda la salida en un archivo.

Ejemplo

```
FILE *pf;
char nombre[20] = "Santiago";
int edad = 34;
if (!(pf = fopen("datos.txt","w"))) /* controlamos si se produce un error */
{
    printf("Error al abrir el archivo");
    exit(0); /* abandonamos el programa */
}
else
{
    fprintf(pf,"%20s%2d\n",nombre,edad);
    fclose(pf);
}
```

- **LEER CON FORMATO**

fscanf(nomb_archivo, formato, argumentos);

Lee los argumentos del archivo. Al igual que con un **scanf**, deberemos indicar la dirección de memoria de los argumentos con el símbolo **&** (ampersand).

Ejemplo

```
FILE *pf;
char nombre[20];
int edad;
if (!(pf = fopen("datos.txt","rb"))) /* controlamos si se produce un error */
{
    printf("Error al abrir el archivo");
    exit(0); /* abandonamos el programa */
}
else
{
    fscanf(pf,"%20s%2d\",nombre,&edad);
    printf("Nombre: %s  Edad: %d",nombre,edad);
    fclose(pf);
}
```

- **ESCRIBIR ESTRUCTURAS**

fwrite(*buffer, tamaño, nº de veces, nomb_archivo);

Se utiliza para escribir bloques de texto o de datos, estructuras, en un archivo.

Donde:

*buffer	:	dirección de memoria de la cual se recogerán los datos
tamaño	:	tamaño en bytes que ocupan esos datos
nº de veces	:	número de elementos del tamaño indicado que se escribirán

LEER ESTRUCTURAS

```
fread( *buffer, tamaño, nº de veces, nomb_archivo );
```

Se utiliza para leer bloques de texto o de datos de un archivo.

Donde:

*buffer	:	dirección de memoria de la cual se recogerán los datos
tamaño	:	tamaño en bytes que ocupan esos datos
nº de veces	:	número de elementos del tamaño indicado que se escribirán

Otras funciones para archivos

Sitúa el puntero al principio del archivo.

```
rewind( nomb_archivo );
```

Sitúa el puntero en la posición que le indiquemos.

Como **origen** podremos poner:

0 o **SEEK_SET**, el principio del archivo
1 o **SEEK_CUR**, la posición actual
2 o **SEEK_END**, el final del archivo

```
fseek( nomb_archivo, long posicion, int origen );
```

- **Cambia el nombre del archivo nombre1 por nombre2**.

```
rename( nombre1, nombre2 );
```

- **Eliminar el archivo indicado en nombre**.

```
remove( nombre );
```

Detección de final de archivo

```
feof( nomb_archivo );
```

Siempre se debe controlar si se ha llegado al final de archivo cuando se está leyendo, de lo contrario podrían producirse errores de lectura no deseados. Para este fin se dispone de la función **feof( )**. Esta función retorna **0** si no ha llegado al final, y un valor diferente de **0** si lo ha alcanzado.

Ejemplos:

1. Muestra un archivo 2 veces

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

int main()
{
  FILE *fichero;

  fichero = fopen("ejemplo1.cpp", "r");
  while(!feof(fichero)) fputc(fgetc(fichero), stdout);
  rewind(fichero);
  while(!feof(fichero)) fputc(fgetc(fichero), stdout);
  fclose(fichero);
  system("pause");
  return 0;
}
```

```
/* Programa que permite la manipulación de archivos. Realizado el 01/01/04 por
BMEO. Versión 1.0 */
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
#include <dos.h>
/*define el nombre del archivo en disco duro */
#define NOM_ARCH "poblador.dat"

struct Registro /*definimos los campos del registro*/
{
	char Nom[40];
	char Poblacion[30];
	int Edad;
	/*campo que verifica si un registro ha sido eliminado logicamente*/
	char valido;
}  regpob;

/* *********  PROTOTIPOS  ************** */
void Mayusculas(int n, char cad[]);
void Xyc(int x, int y, int c, char s[]);
void Margen(int x1, int y1, int x2, int y2, int c, int nl);
void Mensaje(void);
char Menu(void);
void Creacion(void);
char MenuActualizacion(void);
char MenuListado(void);
void Actualizacion(void);
void Listar_Poblacion(FILE *fp);
void Listar_Edad(FILE *fp);
void ListadoPantalla(FILE *fp);
void ListadoGeneral(FILE *fp);
void Listados(void);
void LeerReg(void);
void MarcaReg(FILE *fp);
void EliminaReg(FILE *fp);
void Modificar(void);
void Ordenar(void);

/* ************ DEFINICION DE SUBPROGRAMAS **************/
void Mayusculas(int n, char cad[])
{
	int i;

	for (i = 0; i <=n-1; i++)
	{
```

```
            cad[i] = toupper(cad[i]);
      }
}
/* ************************************************** */
void Xyc(int x, int y, int c, char s[])
/* Muestra un mensaje en una posicion de la pantalla en un color definido */
{
      gotoxy(x, y);
      textcolor(c);
      cprintf("%s", s);
}
/* ************************************************** */
void Margen(int x1, int y1, int x2, int y2, int c, int no)
/* Dibuja recuadros en la pantalla en un color definido */
{
      unsigned int i;
      switch (no)
      {
            case 1:
                  Xyc(x1,y1, c, "┌");   /* ALT + 218 */
                  Xyc(x2,y1, c, "┐");   /* ALT + 191 */
                  Xyc(x1,y2, c, "└");   /* ALT + 192 */
                  Xyc(x2,y2, c, "┘");   /* ALT + 217 */
                  for (i=x1+1; i<=x2-1; i++)
                  {
                        Xyc(i,y1, c, "─");  /* ALT + 196 */
                        Xyc(i,y2, c, "─");  /* ALT + 196 */
                  }
                  for (i=y1+1;i <= y2-1; i++)
                  {
                        Xyc(x1,i, c, "│");  /* ALT + 179 */
                        Xyc(x2,i, c, "│");  /* ALT + 179 */
                  }
                   break;
            case 2:
                  Xyc(x1,y1, c, "╔"); /* ALT + 201 */
                  Xyc(x2,y1, c, "╗"); /* ALT + 187 */
                  Xyc(x1,y2, c, "╚"); /* ALT + 200 */
                  Xyc(x2,y2, c, "╝"); /* ALT + 188 */
                  for (i=x1+1; i<=x2-1; i++)
                  {
                        Xyc(i,y1, c, "═"); /* ALT + 205 */
                        Xyc(i,y2, c, "═"); /* ALT + 205 */
                  }
                  for (i=y1+1;i <= y2-1; i++)
                  {
                        Xyc(x1,i, c, "║"); /* ALT + 186 */
```

```
                        Xyc(x2,i, c, "|"); /* ALT + 186 */
                }
                break;
        }
}
/* ********************************************************** */
void Mensaje(void)
/* Muestra el mensaje en la izquina inferior derecha de la pantalla */
{
        Margen(33, 22, 79, 24, 15, 1);
        Xyc(34, 23, 14, "Presione cualquier tecla para continuar... ");
        getch();
}
/* ********************************************************** */
char Menu(void)
/* Muestra las opciones del menu y permite elegir una de ellas */
{
        char opcion;

        clrscr();
        Margen(29, 5, 50, 7, 15, 2);
        Xyc(33, 6, 14, "MENU PRINCIPAL");
        Margen(20, 8, 60, 15, 15, 2);
        Xyc(28, 9,  14, "[C]reacion");
        Xyc(28, 10, 14, "[A]ctualizacion");
        Xyc(28, 11, 14, "[O]rdenacion");
        Xyc(28, 12, 14, "[L]istados");
        Xyc(28, 13, 14, "[F]inalizar Programa");
        Margen(25, 16, 55, 18, 15, 1);
        Xyc(30, 17, 15, "Digite su opcion... ");
        gotoxy(51, 17);
        opcion = (char)toupper(getch());

        return opcion;

}
/* ********************************************************** */
void Creacion(void)
/* Permite crear el archivo, si ya existe muestra un mensaje pidiendo confirmacion */
{
        char opcion = 'N', continuar = 'N';
        int x, y;
        FILE *fp;

        clrscr();
        /* abrimos el archivo para leer*/
        fp = fopen(NOM_ARCH, "r+b");
        /* 2da FORMA ===> fp = fopen("poblador.dat", "r+b"); */
```

```
if(!fp) /* verifica si el archivo existe */
{       /* si el archivo no existe se abre para escritura */
        fp  = fopen(NOM_ARCH, "w+b");
        /* verfica si el archivo se ha podido abrir correctamente */
        if (fp != NULL)
        {       /* el archivo se ha abierto correctamente */
                do
                {       /* llama al procedimiento que lee un registro */
                        clrscr();
                        LeerReg();
                        /*mueve el puntero al final del archivo */
                        fseek(fp, 0, SEEK_END);
                        /* escribe el registro en el archivo */
                        fwrite(&regpob, sizeof(regpob), 1, fp);
                        putch('\n');
                        x = wherex();
                        y = wherey();
                        /* pregunta si se desea agregar mas registros */
                        Xyc(x, y, 7, "Agregar mas registros? <S/N> ");
                        continuar = (char)toupper(getch());
                } while (continuar !='N');
                fclose(fp);
        }
        else
        {       /* el archivo no se ha abierto correctamente */
                Xyc(40, 10, 4, "E R R O R!!! ");
                Xyc(20, 11, 4, "El archivo no se ha podido abrir o crear");
                Mensaje();
        }
}
else
{       /* si el archivo ya existe muestra un mensaje */
        Xyc(20, 11, 15, "CUIDADO SE BORRARAN SUS DATOS!!!");
        Xyc(25, 13, 15, "DESEA CONTINUAR ? <S/N>");
        gotoxy(45, 13);
        opcion = (char)toupper(getch());
        if (opcion == 'S')
        {       /* crea nuevamente el archivo */
                fp = fopen(NOM_ARCH, "w+b");
                /* verfica si el archivo se ha abierto correctamente */
                if (fp != NULL)
                {       /* el archivo se ha abierto correctamente */
                        do
                        {       /* llama al procedimiento que lee un registro */
                                clrscr();
                                LeerReg();
                                /*mueve el puntero al final del archivo */
                                fseek(fp, 0, SEEK_END);
```

```
                                /* escribe el registro en el archivo */
                                fwrite(&regpob, sizeof(regpob), 1, fp);
                                putch('\n');
                                x = wherex();
                                y = wherey();
                                /* pregunta si se desea agregar mas registros */
                                Xyc(x, y, 7, "Agregar mas registros? <S/N> ");
                                continuar = (char)toupper(getch());
                        } while (continuar !='N');
                        fclose(fp);
                }
                else
                {       /* el archivo no se ha abierto correctamente */
                        Xyc(40, 10, 4, "E R R O R!!! ");
                        Xyc(20, 11, 4, "El archivo no se ha podido abrir o
                             crear");
                        Mensaje();
                }
        }
    }
}
/* ********************************************************** */
char MenuActualizacion(void)
{
    char resact = '5';

    clrscr();
    Margen(29, 5, 50, 7, 15, 2);
    Xyc(31, 6, 14, "MENU ACTUALIZACION");
    Margen(20, 8, 60, 14, 15, 2);
    Xyc(28, 9, 7, "1.  Insertar registro");
    Xyc(28, 10, 7, "2.  Marcar registro");
    Xyc(28, 11, 7, "3.  Eliminar registros marcados");
    Xyc(28, 12, 7, "4.  Modificar registro");
    Xyc(28, 13, 7, "5.  Salir");
    Margen(25, 16, 55, 18, 15, 1);
    Xyc(30, 17, 14, "Digite su opcion...");
    resact = (char)getch();

    return resact;
}
/* ********************************************************** */
void Actualizacion(void)
{
    char opcact = '5';
    /* long int numero;*/

    FILE *fp;
```

```
	fp = fopen(NOM_ARCH, "r+b");
	if(!fp)
	{
		Xyc(30, 10, 4, "E R R O R!!!");
		Xyc(10, 11, 4, "EL ARCHIVO NO EXISTE!!!");
		Mensaje();
	}
	else
	{
		do
		{
			opcact = MenuActualizacion();
			switch (opcact)
			{
				case '1' :
					clrscr();
					LeerReg();
					fseek(fp, 0, SEEK_END);
					fwrite(&regpob, sizeof(regpob), 1, fp);
					break;
				case '2':
					clrscr();
					MarcaReg(fp);
					break;
				case '3':
					clrscr();
					EliminaReg(fp);
					Margen(30, 9, 60, 12, 7, 2);
					Xyc(35, 10, 14, "BORRANDO REGISTROS....");
					Xyc(37, 11, 14, "POR FAVOR ESPERE...");
					delay(1700);
					break;
				case '4':
					clrscr();
					Modificar();
					break;
			}
		} while (opcact != '5');
	}
	fclose(fp);
}
/* ******************************************************** */
char MenuListado(void)
{
	char reslis = '5';

	clrscr();
```

```
	Margen(29, 5, 50, 7, 15, 2);
	Xyc(34, 6, 14, "MENU LISTADOS");
	Margen(20, 8, 60, 14, 15, 2);
	Xyc(28, 9, 7, "1.  Listar por edad");
	Xyc(28, 10, 7, "2.  Listar por poblacion");
	Xyc(28, 11, 7, "3.  Listado General");
	Xyc(28, 12, 7, "4.  Listado por Pantallas");
	Xyc(28, 13, 7, "5.  Salir");
	Margen(25, 16, 55, 18, 15, 1);
	Xyc(30, 17, 14, "Digite su opcion...");
	reslis = (char)getch();

	return reslis;
}
/* ************************************************** */
void Listados(void)
{
	char oplis = '5';
	FILE *fp;

	fp = fopen(NOM_ARCH, "rb");
	if(!fp)
	{
		Xyc(30, 10, 4, "E R R O R!!!");
		Xyc(10, 11, 4, "EL ARCHIVO NO EXISTE!!!");
		Mensaje();
	}
	else
	{
		do
		{
			oplis = MenuListado();
			switch (oplis)
			{
				case '1' :
					clrscr();
					Listar_Edad(fp);
					break;
				case '2':
					clrscr();
					Listar_Poblacion(fp);
					break;
				case '3':
					clrscr();
					ListadoGeneral(fp);
					break;
				case '4':
					clrscr();
```

```
                                ListadoPantalla(fp);
                                break;
                        }
                } while (oplis != '5');
        }
        fclose(fp);
}
/* ******************************************************** */
void Listar_Poblacion(FILE *fp)
{
        char pobus[30];
        int i, sw = 1;

        Margen(20, 1, 61, 3, 15, 2);
        Xyc(31, 2, 7, "LISTADO POR POBLACION");
        Xyc(10, 5, 7, "Entre la poblacion a listar.... ");
        fflush(stdin);
        fgets(pobus, 30, stdin);
        /* la funcion fgets captura el retorno de linea, hay que eliminarlo: */
        for(i = strlen(pobus)-1; i && pobus[i] < ' '; i--)
        {
                pobus[i] = 0;
        }
        Mayusculas(strlen(pobus), pobus);
        printf("\n");
        printf("\n");
        /* va al principio del archivo */
        rewind(fp);
        /* lee el primer registro del archivo fp */
        fread(&regpob, sizeof(regpob), 1, fp);
        /* mientras no sea el final del archivo fp busca registros
        que cumplan con la condicion */
        while (!feof(fp))
        {
                if (strcmp(pobus, regpob.Poblacion) == 0  && regpob.valido != 'N')
                {
                        printf("%-40s %-5d", regpob.Nom, regpob.Edad);
                        printf("\n");
                        sw = 0;
                }
                fread(&regpob, sizeof(regpob), 1, fp);
        }
        if (sw == 1)
        {
                Xyc(35, 13, 14, "LO SIENTO...");
                Xyc(15, 15, 14, "NO EXISTEN REGISTROS QUE CUMPLAN LA
                    CONDICION!!!");
                Mensaje();
```

```
	}
	Mensaje();
}
/* ********************************************************** */
void Listar_Edad(FILE *fp)
{
	int edbus = 0, sw = 1;

	Margen(20, 1, 60, 3, 15, 2);
	Xyc(32, 2, 7, "LISTADO POR EDAD");
	Xyc(10, 6, 7, "Entre edad a partir de la cual desea listar ====> ");
	fflush(stdin);
	scanf("%d", &edbus);
	/* va al principio del archivo */
	rewind(fp);
	/* lee el primer registro del archivo fp */
	fread(&regpob, sizeof(regpob), 1, fp);
	/* mientras no sea el final del archivo fp busca registros
	que cumplan con la condicion */
	textcolor(15);
	printf("\n");
	printf("\n");
	while (!feof(fp))
	{
		if (regpob.Edad >= edbus && regpob.valido != 'N')
		{
			cprintf("%-40s %-30s %4d ", regpob.Nom,
				regpob.Poblacion, regpob.Edad);
			printf("\n");
			sw = 0; /*existe por lo menos un registro */
		}
		fread(&regpob, sizeof(regpob), 1, fp);
	}
	if (sw == 1)
	{
		Xyc(35, 13, 14, "LO SIENTO...");
		Xyc(15, 15, 14, "NO EXISTEN REGISTROS QUE CUMPLAN LA
		    CONDICION!!!");
		Mensaje();
	}
	Mensaje();
}
/* ********************************************************** */
void ListadoPantalla(FILE *fp)
{
	long int n;

	rewind(fp);
```

```
        fread(&regpob, sizeof(regpob), 1, fp);
        n = 0;
        while(!feof(fp))
        {
                clrscr();
                if (regpob.valido =='S')
                {
                        Margen(10, 3, 70, 17, 15, 2);
                        gotoxy(32, 5);
                        textcolor(14);
                        cprintf("Registro No. %ld", n);
                        Margen(30, 4, 50, 6, 7, 1);
                        Xyc(13, 9, 8, " NOMBRE POBLADOR  ");
                        gotoxy(35, 9);
                        textcolor(8);
                        cprintf(": %s", regpob.Nom);
                        Xyc(13, 11, 8, " LUGAR DE NACIMIENTO ");
                        gotoxy(35, 11);
                        textcolor(8);
                        cprintf(": %s", regpob.Poblacion);
                        Xyc(13, 13, 8, " EDAD ");
                        gotoxy(35, 13);
                        textcolor(8);
                        cprintf(": %d", regpob.Edad);
                        Mensaje();
                        n = n + 1;
                }
                fread(&regpob, sizeof(regpob), 1, fp);
        }
}
/* ************************************************** */
void ListadoGeneral(FILE *fp)
{

        int sw = 1;

        Margen(20, 1, 60, 3, 15, 2);
        Xyc(32, 2, 7, "LISTADO GENERAL");
        printf("\n");
        printf("\n");
        printf("\n");
        printf("\n");
        rewind(fp);
        /* lee el primer registro del archivo fp */
        fread(&regpob, sizeof(regpob), 1, fp);
        /* mientras no sea el final del archivo fp muestra todos
        los registros */
        textcolor(15);
```

```
        while (!feof(fp))
        {
                if (regpob.valido != 'N')
                {
                        cprintf("%-40s %-30s %-5d", regpob.Nom,
                             regpob.Poblacion, regpob.Edad);
                        printf("\n");
                        sw = 0;
                }
                fread(&regpob, sizeof(regpob), 1, fp);
        }
        if (sw == 1)
        {
                Xyc(35, 13, 14, "LO SIENTO...");
                Xyc(15, 15, 14, "NO EXISTEN REGISTROS QUE CUMPLAN LA
                     CONDICION!!!");
                Mensaje();
        }
        Mensaje();
}
/* ******************************************************** */
void LeerReg(void)
{
        int edad, i;
        char redad ='S';

        Margen(28, 1, 52, 3, 7, 2);
        Xyc(30, 2, 14, "LECTURA DE REGISTROS");
        regpob.valido = 'S';
        /* lectura de los compos del registro */

        Xyc(1, 5, 7, "Nombre Poblador : ");
        fflush(stdin);
        fgets(regpob.Nom, 40, stdin);
        /* la funcion fgets captura el retorno de linea, hay que eliminarlo: */
        for(i = strlen(regpob.Nom)-1; i && regpob.Nom[i] < ' '; i--)
        {
                regpob.Nom[i] = 0;
        }
        Mayusculas(strlen(regpob.Nom), regpob.Nom);

        Xyc(1, 7, 7, "Lugar de Nacimiento  : ");
        fflush(stdin);
        fgets(regpob.Poblacion, 30, stdin);
        /* la funcion fgets captura el retorno de linea, hay que eliminarlo: */
        for(i = strlen(regpob.Poblacion)-1; i && regpob.Poblacion[i] < ' '; i--)
        {
                regpob.Poblacion[i] = 0;
```

```
        }
        Mayusculas(strlen(regpob.Poblacion), regpob.Poblacion);
        Xyc(1, 9, 7, "Edad Poblador : ");
        do
        {
                redad = 'S';
                fflush(stdin);
                scanf("%d", &edad);
                if (edad <= 0)
                {
                        Xyc(37, 12, 15, "E R R O R !!!");
                        Xyc(35, 13, 15, "EDAD NO VALIDA!!!");
                        Xyc(25, 14, 15, "LA EDAD DEBE SER MAYOR QUE
                             CERO!!!");
                        Xyc(35, 16, 14, "Intente de nuevo... ");
                        Mensaje();
                        clrscr();
                        Xyc(1, 5, 7, "Edad Poblador : ");
                }
                else
                {
                        if (edad > 110)
                        {
                                Xyc(37, 12, 15, "E R R O R !!!");
                                Xyc(35, 13, 15, "EDAD NO VALIDA!!!");
                                Xyc(36, 14, 15, "ESTA SEGURO? <S/N>" );
                                redad = (char)toupper(getch());
                                if (redad == 'N')
                                {
                                        Xyc(35, 16, 14, "Intente de nuevo... ");
                                        Mensaje();
                                        clrscr();
                                        Xyc(1, 5, 7, "Edad Poblador : ");
                                }
                        }
                }
        } while (edad <=0 || redad != 'S' );
        regpob.Edad = edad;
}
/* ********************************************************** */
void MarcaReg(FILE *fp)
{
        long int numero, dist;

        Xyc(10, 1, 15, "Numero de registro que desea marcar ===> ");
        scanf("%ld", &numero);
        dist = numero*sizeof(regpob);
        fseek(fp, dist, SEEK_SET);
```

```
	fread(&regpob, sizeof(regpob), 1, fp);
	regpob.valido = 'N';

	fseek(fp, numero*sizeof(regpob), SEEK_SET);
	fwrite(&regpob, sizeof(regpob), 1, fp);
}
/* ************************************************** */
void EliminaReg(FILE *fp)
{
	FILE *artemp;

	artemp = fopen("poblador.tmp", "w+b");
	rewind(artemp);
	rewind(fp);
	while(fread(&regpob, sizeof(regpob), 1, fp))
	{
		if (regpob.valido == 'S')
		{
			fwrite(&regpob, sizeof(regpob), 1, artemp);
		}
	}
	fclose(artemp);
	fclose(fp);
	remove("poblador.bak");
	rename("poblador.dat", "poblador.bak");
	rename("poblador.tmp", "poblador.dat");
}
/* ************************************************** */
void Modificar(void)
{
	Xyc(20, 9, 15, "En construccion...");
	Mensaje();
}
/* ************************************************** */
void Ordenar(void)
{
      clrscr();
	Xyc(20, 9, 15, "En construccion...");
	Mensaje();
}
/* ******************** PRINCIPAL ***********************/
int main(void)
{
	char opcion;


	do
	{
```

```
            opcion = Menu();
            switch (opcion)
            {
                  case 'C':
                        Creacion();
                        break;
                  case 'A':
                        Actualizacion();
                        break;
                  case 'O':
                        Ordenar();
                        break;
                  case 'L':
                        Listados();
                        break;
            }
      } while (opcion != 'F');

      return 0;
}
```