FACULTAD DE INGENIERIA U.N.A.M.
DIVISION DE EDUCACION CONTINUA

FACULTAD DE INGENIERIA U.N.A.M.
DIVISION DE EDUCACION CONTINUA

CENTRO DE INFORMACION Y DOCUMENTACION
"ING. BRUNO MASCANZONI"

El Centro de Información y Documentación Ing. Bruno Mascanzoni tiene por objetivo satisfacer las necesidades de actualización y proporcionar una adecuada información que permita a los ingenieros, profesores y alumnos estar al tanto del estado actual del conocimiento sobre temas específicos, enfatizando las investigaciones de vanguardia de los campos de la ingeniería, tanto nacionales como extranjeras.

Es por ello que se pone a disposición de los asistentes a los cursos de la DECFI, así como del público en general los siguientes servicios:

- **Préstamo interno.**
- **Préstamo externo.**
- **Préstamo interbibliotecario.**
- **Servicio de fotocopiado.**
- **Consulta a los bancos de datos: librunam, seriunam en cd-rom.**

Los materiales a disposición son:

- **Libros.**
- **Tesis de posgrado.**
- **Noticias técnicas.**
- **Publicaciones periódicas.**
- **Publicaciones de la Academia Mexicana de Ingeniería.**
- **Notas de los cursos que se han impartido de 1980 a la fecha.**

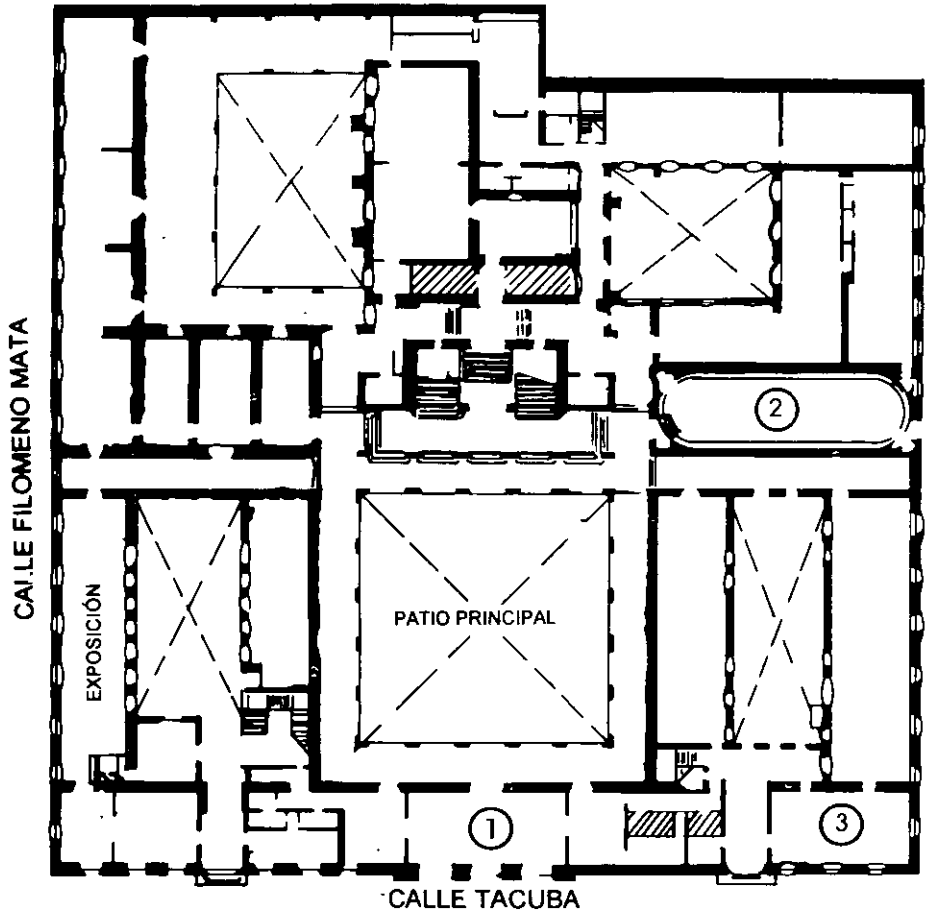
En las áreas de ingeniería industrial, civil, electrónica, ciencias de la tierra, computación y, mecánica y eléctrica.

El CID se encuentra ubicado en el mezzanine del Palacio de Minería, lado oriente.

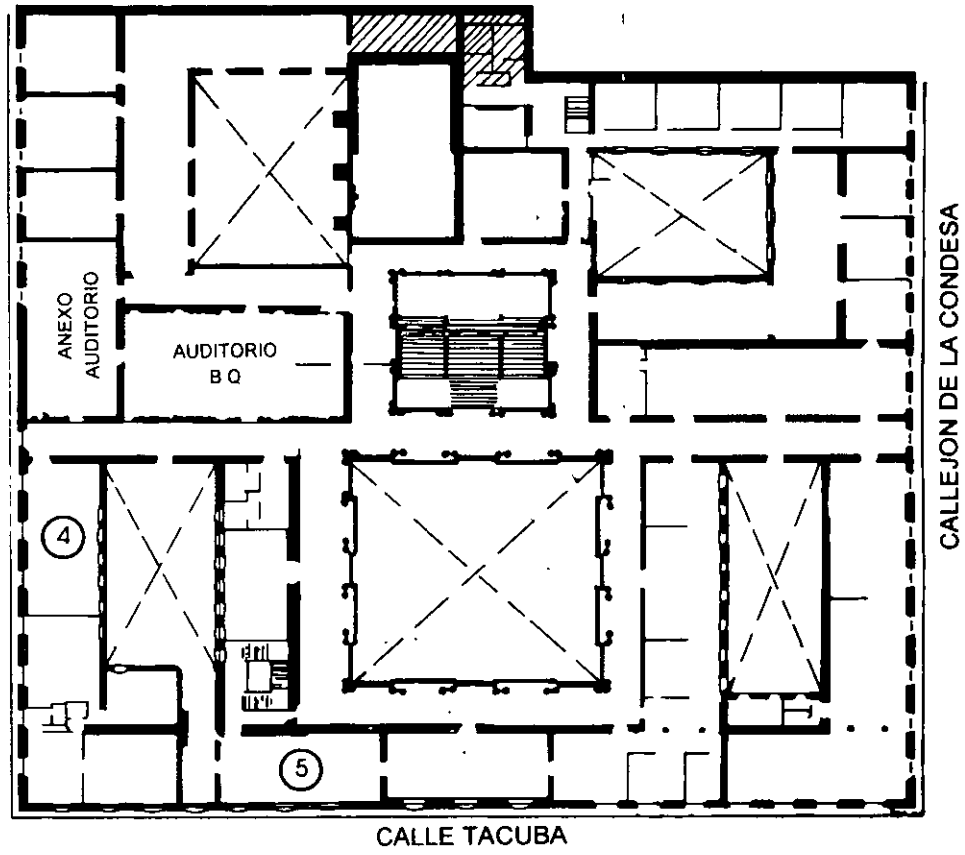
El horario de servicio es de 10:00 a 19:30 horas de lunes a viernes.

Palacio de Mineria Calle de Tacuba 5 Primer piso Deleg. Cuauhtémoc 06000 México, D.F. APDO. Postal M-2285
Teléfonos: 512-8955 512-5121 521-7335 521-1987 Fax 510-0573 521-4020 AL 26

PALACIO DE MINERIA

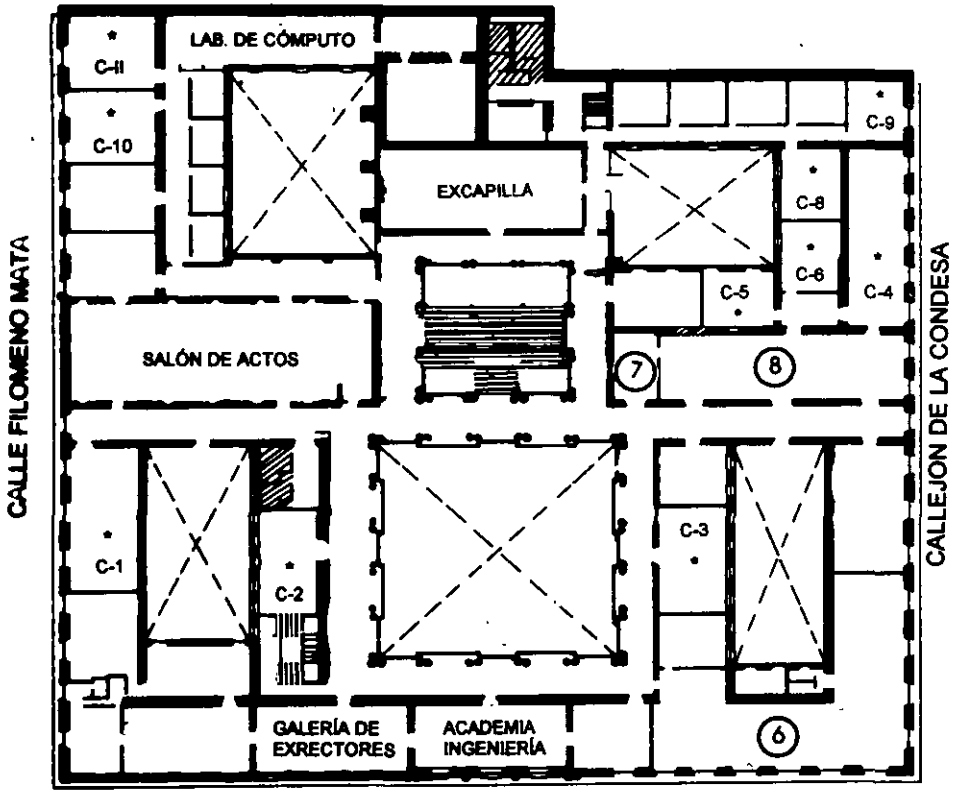


PLANTA BAJA



MEZZANINNE

PALACIO DE MINERIA



Ier. PISO

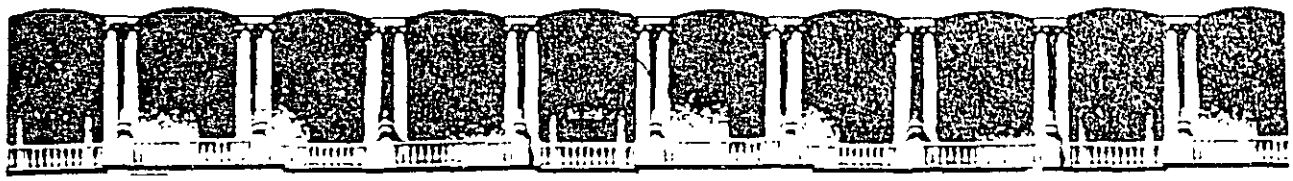
GUÍA DE LOCALIZACIÓN

1. ACCESO
2. BIBLIOTECA HISTÓRICA
3. LIBRERÍA UNAM
4. CENTRO DE INFORMACIÓN Y DOCUMENTACIÓN "ING. BRUNO MASCANZONI"
5. PROGRAMA DE APOYO A LA TITULACIÓN
6. OFICINAS GENERALES
7. ENTREGA DE MATERIAL Y CONTROL DE ASISTENCIA
8. SALA DE DESCANSO

SANITARIOS

* AULAS

DIVISIÓN DE EDUCACIÓN CONTINUA
FACULTAD DE INGENIERÍA U.N.A.M.
CURSOS ABIERTOS



FACULTAD DE INGENIERIA U.N.A.M.
DIVISION DE EDUCACION CONTINUA

MATERIAL DIDACTICO

LENGUAJE DE PROGRAMACION "C"

CURSO DIRIGIDO AL PERSONAL DEL BANCO DE MEXICO

MARZO, 1998

Palacio de Mineria Calle de Tacuba 5 Primer piso Deleg Cuauhtemoc 06000 Mexico D.F. APDO Postal M-2285
Telefonos 512-8955 512-5121 521-7335 521-1987 Fax 510-0573 521-4020 AL 26

TIPOS, OPERADORES Y EXPRESIONES

Identificadores

Un identificador no puede ser una palabra reservada (while, break, if, char, return, main, etc.).

El identificador puede estar formado por letras, dígitos y "_":

- El primer carácter debe ser una letra.
- El carácter "_" es utilizado como carácter de inicio de identificadores dentro de las rutinas de la biblioteca estándar.

Las letras minúsculas y mayúsculas son distintas.

Solamente los primeros 31 caracteres son significativos.

9

Tipos y tamaños de datos

Un tipo de dato es un conjunto de valores y un conjunto de operaciones que se pueden realizar con ellos.

Existen tres grupos básicos de tipos en C:

- Enteros
- De punto flotante
- Carácter

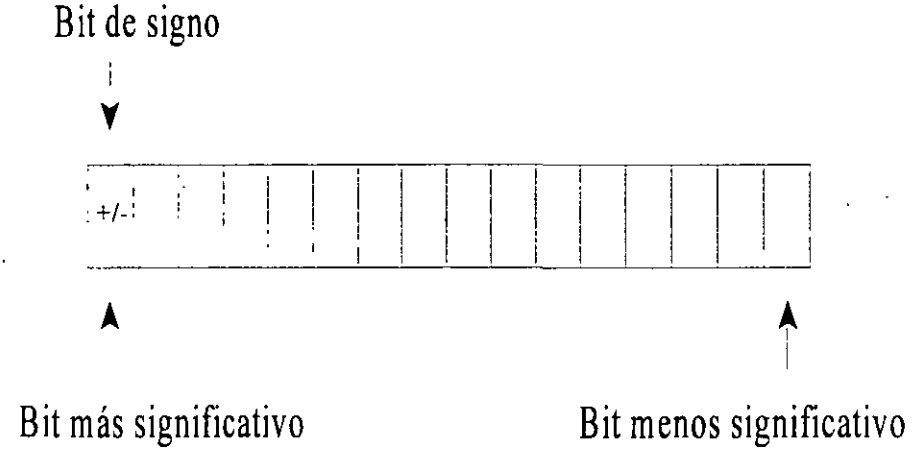
Enteros

El tamaño de los tipos enteros depende de la máquina.

Los tipos enteros signados son:

- short [int]
- int
- long [int]

Representación:

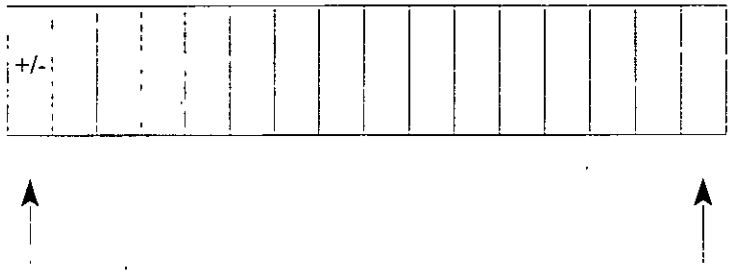


4

Los enteros sin signo son:

- unsigned short [int]
- unsigned [int]
- unsigned long [int]

Representación:



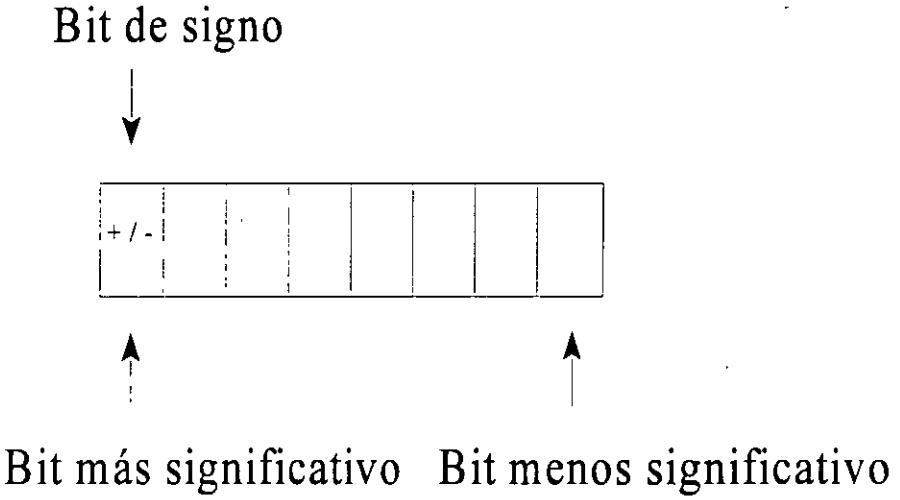
Carácter

Los tipos carácter son:

- [unsigned] char

Sus valores son enteros

Representación:



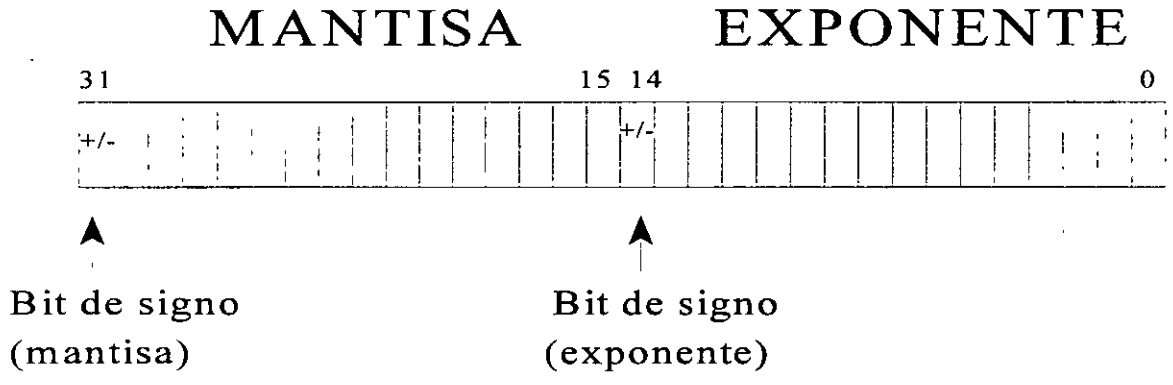
6

Punto flotante

Los tipos de punto flotante son:

- float
- double
- long double

Representación:



Constantes

Enteras:

- Decimal: 12, 125
- Octal: 007, 057
- Hexadecimal: 0xa95, 0xff23

De punto flotante:

- Pueden ser escritas como:

  .0034
  12.5
  3e1
  1.0E-3

Carácter:

- Se almacena el valor numérico del carácter.
- Pueden ser utilizadas en expresiones numéricas.
- Se escriben como: 'a', '+', '1'.
- Algunos caracteres especiales se representan por más de un carácter:

  '\n' (nueva línea) '\t' (tabulador) '\f' (form feed)
  '\a' (campana) '\b' (back space) '\r' (retorno de carro)

- También se pueden representar:

8

'\033' (ascii en octal) '\0xff' (ascii en hexadecimal)

9

Enumerados

Una enumeración es una lista de valores enteros constantes:

```
enum boolean {NO, YES};
```

El primer nombre en la lista de enumerados toma un valor de cero, el siguiente uno, y así sucesivamente.

Se pueden cambiar los valores que toman los elementos de la lista:

```
enum letras { alpha, beta, gamma = 30, epsilon, zeta = 65 };
```

Los valores que toman son:

```
alpha = 0
beta = 1
gamma = 30
epsilon = 31
zeta = 65
```

Se pueden declarar variables de tipo enumerado, que serán manejadas como int y a las cuales se les puede asignar alguno de los valores de la lista:

```
enum boolean x;

x = YES;
```

Los enumerados solamente son utilizados para propósitos de documentación.

Tamaño de tipos de datos

El tamaño de los tipos de datos depende de la máquina, el siguiente programa determina el número de bytes que ocupan los tipos básicos:

```
/*
**      Programa 2_1
**
**      Este programa determina el espacio en bytes que ocupa cada uno de
**      los tipos básicos. El operador sizeof calcula el espacio ocupado por
**      un tipo (puede ser un tipo definido por el usuario).
**
*/

#include <stdio.h>

main()
{
        printf("El tipo char ocupa %d bytes\n\n",sizeof(char));
        printf("El tipo int ocupa %d bytes\n\n",sizeof(int));
        printf("El tipo long ocupa %d bytes\n\n",sizeof(long));
        printf("El tipo short ocupa %d bytes\n\n",sizeof(short));
        printf("El tipo float ocupa %d bytes\n\n",sizeof(float));
        printf("El tipo double ocupa %d bytes\n\n",sizeof(double));
}
```

Declaraciones y definiciones

En una **declaración**, un identificador es asociado a un tipo; pero no se reserva memoria.

Una **definición** es una declaración en la que se reserva memoria.

Las variables y las funciones deben ser declaradas antes de que sean usadas.

Las variables pueden ser inicializadas al momento de definirse:

```
main()
{
        int             r = 2,i,j;
        float           pi=3.1415;
        char            opcion = 's';

        ...

}
```

No es válido:

```
        int     i = j = 0;
```

Al definir una variable se puede agregar el calificativo **const** para indicar que su valor no será cambiado:

```
const  double  pi = 3.1415;
```

Conversiones de tipos

Una expresión puede involucrar variables y constantes de diferentes tipos:

```
...
char        c;
int         i;
float       f;
double      d;

d = f * (i + c);

...
```

Reglas de conversión de tipos

En una expresión binaria si los operandos son de diferentes tipos el de menor grado es convertido al de mayor grado y el resultado de la expresión es del tipo de mayor grado.

La jerárquica de tipos de menor a mayor:

short,char
unsigned int
int
long int
unsigned long int
float
double
long double

En el ejemplo anterior ¿de que tipo es el resultado de la expresión asignado a la variable d?

Conversión de tipos en asignaciones

El tipo a la derecha del operador de asignación es convertido al tipo de la variable del lado izquierdo, de acuerdo a las reglas que se indican en la siguiente tabla:

Tipo =	Tipo	Resultado
float	int	Conversión exacta
int	float	Parte decimal es descartada; si el valor no esta en el rango, el resultado es desconocido.
int	char	Conversión exacta
char	int	Si el char es unsigned, los bits menos significativos son copiados; los restantes son descartados. Si el char es signed, depende de la implementación.
double	float	Conversión exacta
long double	float	Conversión exacta
float	double	Si el double esta fuera del rango de un float, el resultado es desconocido.

Conversión explícita de tipos (cast)

El valor del operando es convertido al tipo encerrado en paréntesis:

```
x = (int)c;
```

Operadores aritméticos

Binarios: **+, - , *, /, %**

Unuarios: **+, -**

Precedencia:	**+, -** (unuarios)
	***, /, %**
	+, -
Asociatividad:	izquierda a derecha

Operadores de relación, igualdad y lógicos

C no proporciona un tipo boolean:

La evaluación de una expresión puede resultar en un valor 0 (falso) o diferente de cero (verdadero).

Los operadores de relación son los siguientes:

<	**<=**
>	**>=**

Los operadores **<=** y **>=** no permiten espacios; así como los de igualdad y los lógicos.

Los operadores de igualdad son los siguientes:

!=	**==**

Los operadores lógicos son:

&& (and)	**||** (or)	**!** (not)

La asociatividad de los operadores de relación, igualdad y lógicos es de izquierda a derecha.

La precedencia de los operadores anteriores, de mayor a menor es la siguiente:

!
<, <=, >, >=
==, !=
&&
||

El operador ! (not) es unuario y cuando se evalúa con una expresión falsa 0 da como resultado 1; por otra parte cuando se evalúa con una expresión diferente de cero da por resultado 0, de esta forma, si x tiene un valor de 5 por ejemplo:

```
x != !(!x)
```

En una expresión que involucra operadores lógicos, cuando el resultado de la expresión se conoce, la evaluación termina:

```
/*
**      Programa 2_2
**
**      Este programa muestra el comportamiento de los operadores lógicos:
**      Cuando el resultado de la expresión se puede determinar, la evalucación
**      termina.
**
*/

#include <stdio.h>

main()
{
        int             i=0, j=0, x, y;

        x = 0 && (i = j = 999);         /* Se asigna a x el resultado de la expresión lógica (0) */
        printf("%d  %d  %d\n",i,j,x);                                   /* se imprime 0 0 0 */
        y = 1 || (i = j++);             /* Se asigna a y el resultado de la expresión lógica (1)*/
        printf("%d  %d  %d\n",i,j,y);                                   /* se imprime 0 0 1 */
}
```

Operadores de incremento y decremento

Los operadores de incremento y decremento son:

++ --

Pueden ser utilizados como prefijo o posfijo:

++x	x++
--x	x--

- ++x incrementa x antes de utilizar su valor
- x++ incrementa x después de utilizar su valor

Se aplican únicamente a variables.

```
/*
**      Programa 2_3
**
**      Este programa muestra el comportamiento de los operadores de incremento
**      y decremento.
**
*/

#include <stdio.h>

main()
{
        int     a=0, b=0, c=0;

        a = ++b + ++c;
        printf("\n%d  %d  %d", a,b,c);                          /* ¿que se imprime? */
        a = b++ + c++;
        printf("\n%d  %d  %d", a,b,c);                          /* ¿que se imprime? */
        a = ++b + c++;
        printf("\n%d  %d  %d", a,b,c);                          /* ¿que se imprime? */
        a = b-- + --c;
        printf("\n%d  %d  %d", a,b,c);                          /* ¿que se imprime? */
        a = ++c + c;
        printf("\n%d  %d  %d", a,b,c);                  /* depende de la maquina */
}
```

Operadores de asignación

Existen dos tipos: simples y compuestos.

El operador de asignación simple es:

=

El operador = asigna el valor de la derecha a la variable de la izquierda.

Se asocia de derecha a izquierda.

Cuando se lleva a cabo una asignación con =, el tipo del operando de la izquierda se convierte al de la derecha de acuerdo a las reglas de conversión vistas.

La asignación es una expresión, que da como resultado el valor y tipo del operando izquierdo, por lo tanto la siguiente operación es válida:

```
i = j = k = 0;
```

es equivalente a:

```
i = (j = (k = 0));
```

Para expresiones con el formato:

var = var operador expresión

donde: var = nombre de una variable

operador = alguno de los siguientes operadores:

+, -, *, /, %,
<<, >>, &, ^, |

expresión = cualquier expresión

se pueden utilizar los operadores de asignación compuestos, para lo cuál la expresión anterior se puede transformar a:

var operador= expresión

No debe existir espacios en blanco entre **operador** y **=**

```
/*
**      Programa 2_4
**
**      Este programa muestra el comportamiento de los operadores  de asignación.
**
*/

#include <stdio.h>

main()
{
        int     a=12,b=5;

        a += b;                                 /* equivalente:  a = a + b  */
        a -= b;                                 /* equivalente:  a = a - b  */
        a *= b+5;                               /* equivalente:  a = a * (b+5) */
        printf("El valor de a es %d",a);
}
```

Operadores para manejo de bits

Operadores binarios lógicos de bits

Los operadores binarios lógicos de bits son:

& (and)
| (or)
^ (xor)

Estos operadores operan de bit en bit. La siguiente tabla muestra su comportamiento:

x	y	x & y	x ^ y	x | y
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	0	1

Ejemplo, asumiendo que se tiene una representación de enteros de 2 bytes:

```
n = 34;      /* 0000 0000 0010 0010  */
x = 16;      /* 0000 0000 0001 0000  */
c = x & n;   /* 0000 0000 0000 0000  */
```

Operador de complemento a uno

El operador de complemento a uno ~, es un operador unuario.

Su comportamiento se muestra en la siguiente tabla:

x	~x
0	1
1	0

Ejemplo:

```
n = 499;        /* 0000 0001 1111 0011 */
x = 16;         /* 0000 0000 0001 0000 */
y = ~n;         /* 1111 1110 0000 1100 (-500) */
z = ~x;         /* 1111 1111 1110 1111 (-17)  */
```

Operadores de corrimiento de bits

Los operadores de corrimiento de bits son binarios y son:

>>
<<

En el caso de <<, se desplazan a la izquierda n bits indicados por el operador de la derecha en el operador de la izquierda:

- Los bits de exceso son descartados.

24

- Se colocan bits cero (0) en la derecha.

Ejemplo:

```
n = 16;          /* 0000 0000 0001 0000 */
c = n << 3;      /* 0000 0000 1000 0000  (128) */
```

En el caso de >>, se desplazan a la derecha n bits indicados por el operador de la derecha en el operador de la izquierda:

- Los bits de exceso son descartados.
- Los bits que entran por la izquierda son:

  para unsigned bit cero (0)
  para signed, depende de la implementación

Ejemplo:

```
n = 16;        /* 0000 0000 0001 0000 */
c = n >> 3;    /* 0000 0000 0000 0010  (2) */
```

en forma general:

x << n es equivalente a $x * 2^n$

x >> n es equivalente a $x / 2^n$

Tabla de precedencia y asociatividad

Operador	**Asociatividad**
() [] ->	izquierda a derecha
! ~ ++ -- + - * & sizeof	derecha a izquierda
* / %	izquierda a derecha
+ -	izquierda a derecha
>> <<	izquierda a derecha
> >= < <=	izquierda a derecha
== !=	izquierda a derecha
&	izquierda a derecha
^	izquierda a derecha
|	izquierda a derecha
&&	izquierda a derecha
||	izquierda a derecha
?:	derecha a izquierda
= += -= &= *= /= %= ^=	derecha a izquierda
|= <<= >>=	
, (operador coma)	izquierda a derecha

LABORATORIO

1. Escriba un programa que despliegue la representación binaria de un número entero.

2. Escriba un programa de empaquetamiento y desempaquetamiento de bits. En un entero de 16 bits se almacenarán en los 8 bits más significativos una clave de trabajador, en los 7 bits siguientes la edad y en el bit menos significativos el sexo.

3. Qué imprime el siguiente programa?

```
/* aritmética de enteros */

main()
{
        int x,y;
        printf("\n\n Aritmética de enteros");

        while(1)
        {
                printf("\n\n Teclea dos numeros enteros: ");
                scanf("%d %d",&x,&y);
                printf("\nx= %d\ty= %d\n",x,y);
                printf("x + y = %d\tx - y = %d\n",x+y,x-y);
                printf("x * y = %d\tx / y = %d\n",x*y,x/y);
                printf("x / y * y = %d\n",x/y*y);
                printf("x mod y = %d\n",x%y);
                printf("x / y * y + x mod y= %d\n",x/y*y+x%y);
        }
}
```

Página intencionalmente blanca.

FUNCIONES Y EL PREPROCESADOR

FUNCIONES

Las funciones son elementos que permiten el desarrollo de programas modulares. En el lenguaje C no existe el concepto de procedimiento o subrutina, todos los modulos de un programa se implementan en base a funciones.

Un programa es un conjunto de definiciones de variables y funciones. La comunicación entre funciones es por parámetros y valores regresados por las funciones, y a través de variables externas.

La función que controla la ejecución del programa se llama **main.**

Las funciones pueden presentarse en cualquier orden dentro del archivo fuente (la función main no se tiene necesariamente que definir primero), o en diferentes archivos, simpre y cuando las funciones no se dividan.

Todas las funciones se definen al mismo nivel, no se puede definir una función en otra (la definición de funciones no se puede anidar).

Las funciones pueden ser recursivas, salvo main.

La sintaxis para la definición de una función es la siguiente:

```
tipo_retorno        nombre (lista_parametros )
{

        definiciones de variables y sentencias

}
```

donde:	tipo_retorno	es el tipo asociado a el valor regresado por la función.
	nombre	identificador de la función.
	lista_parametros	lista de parametros en la siguiente forma: tipo parametro

Para cada parámetro en la lista de parámetros se debe especificar su tipo.

Por default, las funciones tienen un tipo de retorno *int*.

Para funciones que no regresan valores (por ejemplo aquellas que actuan como procedimientos), se puede especificar como tipo de retorno *void*.

Ejemplo:

```
double maximo(double x, double y)
{

    /* Funcion que recibe dos parametros de tipo double y su valor de retorno es de tipo double */

}
```

Declaración y definición de una función

Se define una función cuando se indica su nombre, el tipo del valor de retorno, el número de parámetros que recibe y su tipo, así como las sentencias que la forman.

Se declara una función cuando únicamente se indica su nombre, el tipo del valor de retorno, el número de parámetros que recibe y su tipo. A la declaración de una función se le conoce comunmente como **prototipo**.

Una función se debe definir una vez y se puede declarar más de una vez en un programa.

Si una función no se declara o define antes de que aparezca una llamada a ella, el compilador asume que regresa un valor de tipo *int* y que el valor, tipo y número de sus parámetros corresponden a los que aparecen en esa llamada.

Ejemplo:

```
double  maximo(double, double);          /* Prototipo o declaracion de la funcion*/

main()
{
        double x=5, y=8, z;

        z = maximo(x,y);

            ...

}

double  maximo(double x, double y)       /* Definicion de la función */
{
        double      max;
        ...

        return max;
}
```

En el ejemplo anterior, sino se especifica la declaración de la función, cuando se hace la llamada a la misma el compilador asume que el tipo del valor de retorno es int. En el momento en el que el compilador encuentra la defincion de la función genera un error indicando que la función ya habia sido definida, aunque realmente el compilador asumio una definición.

Valores de regreso

Una función puede regresar un valor, el cual puede ser utilizado en la expresión en donde se hizo la llamada como cualquier constante del tipo indicado para dicho valor.

El valor de retorno puede ser cualquier expresión indicada en una cláusula *return*.

return [(expresión)];

La cláusula return termina la ejecución de una función y pasa el control a la función que hizo la invocación.

Si se indica una expresión en la cláusula return, esta es evaluada y se regresa el resultado de esta a la función que hizo la llamada.

En la definición de la función, el valor de retorno no necesariamente debe de ser del tipo indicado para este, ya que al momento de ejecución se hace una conversión implicita (si existe) del valor de regreso al tipo definido para este. De esta forma, se puede regresar el valor de una variable entera en una función con tipo de valor de retorno double:

```
double      maximo(int a, int b)
{
      int   max;
      . . .

      return max;
```

}

Por lo tanto, la expresión del return debe de ser del mismo tipo que el especificado en el tipo de retorno, o bien, debe de poderse llevar a cabo una conversión implicita de dicha expresión al tipo del valor de retorno.

En una función pueden existir más de una cláusula return, en el caso de que no exista ninguna, la función termina al alcanzar su última sentencia y el valor de retorno es indefinido.

Ejemplo:

```
double  maximo(double x, double y)     /* Funcion que obtiene el mayor de dos numeros */
{
        if (x > y)
                return x;
        return y;
}
```

Paso de parámetros

Los **parámetros actuales** de una función son la lista de valores asociados a cada uno de los parámetros, con los que se hace una llamada a una función.

Los **parámetros formales** de una función son la lista de variables que aparecen como parámetros en la definición de la función y no guardan ninguna relación en cuanto a nombre o tipo con las variables que pudieran ser tomadas como parámetros actuales en alguna llamada a la función.

El paso de parámetros en las funciones es por valor, es decir, solamente se utiliza el valor de las expresiones indicadas como parámetros.

Cuando se hace una llamada a una función:

1. Cada expresión en la lista de parámetros actuales es evaluada (no existe un orden de evaluación).

2. Se crean variables que corresponden a los parámetros formales y los valores de los parámetros actuales se copian a estas variables.

3. Las sentencias de la función se ejecutan.

4. Si existe una cláusula return, el control del programa pasa a la función que hizo la llamada.

5. Si la cláusula return incluye una expresión, el valor de esta es convertido (si es válido) a el tipo especificado como tipo de retorno y el valor es regresado a la función que hizo la llamada.

6. Si no existe cláusula return o esta no contiene una expresión, la función regresa un valor desconocido.

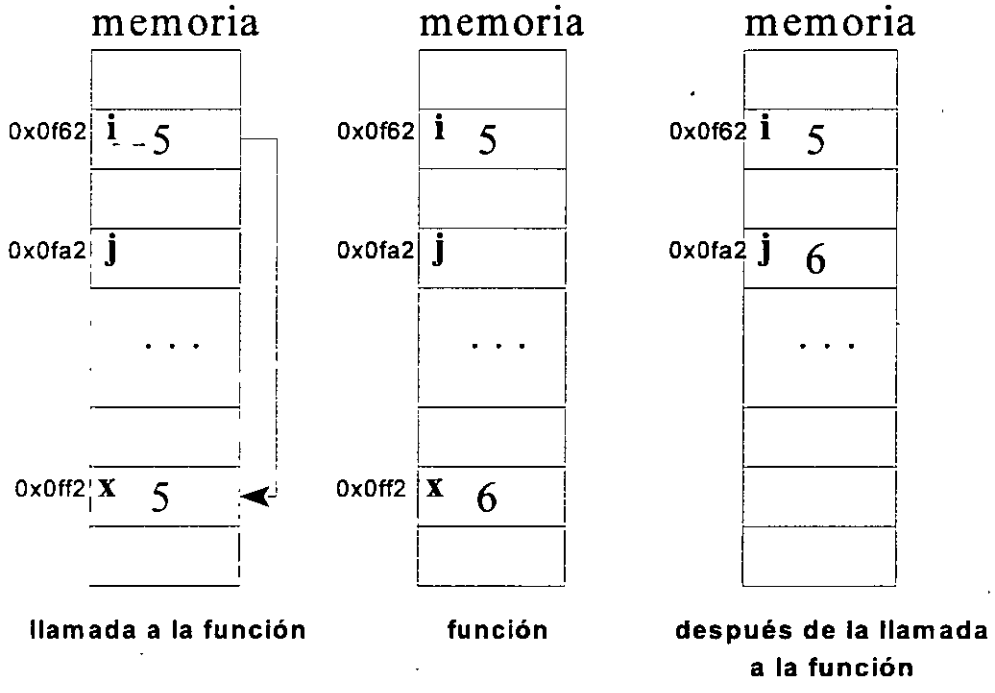
7. Las variables que representan a los parámetros formales se destruyen.

Ejemplo:

```
/*
**      Programa 4_1
**
**      Este programa muestra el comportamiento del paso de parametros por valor de
**      las funciones.
**
*/
#include      <stdio.h>

int incrementa(int);

main()
{
        int     i, j;

        i = 5;
        printf("\nValores antes de la llamada a la funcion: i=>%d, j=>%d\n",i,j);
        j = incrementa(i);
        printf("\nValores despues de la llamada a la funcion:i=>%d, j=>%d\n",i,j);
}
/*
**      incrementa
**
**      Funcion que incrementa en forma unitaria el valor pasado como parametro.
**
**      PARAMETROS
**              n       valor a incrementar
**
**      RETURN
**              valor incrementado
*/
int incrementa(int x)
{
        x++;
        printf("\nValor en la llamada a la funcion: %d\n",x);
        return x;
}
```

¿Qué salida genera el ejemplo anterior? ¿Cambia el valor de i depués de la llamada?

Cuando se hace la llamada a la función se crea una variable(x) en la cual se copia el valor del parámetro actual(i). En la función se utiliza el parámetro formal:



llamada a la función **función** **después de la llamada a la función**

Variables automáticas

Las variables automáticas son aquellas que se definen en un bloque o bien aquellas que se definen en una función.

Para estas variables se reserva espacio de memoria cada que se ejecuta el bloque o función.

Cuando termina la ejecución del bloque o función, estas variables se destruyen y se libera el espacio de memoria que ocupan.

Solo pueden ser accesadas desde el bloque o función que las define.

Los parámetros formales de una función son variables automáticas.

Se indican mediante la palabra *auto*, pero es opcional:

```
main()
{
        int         i,j,k;          /* Variables automaticas en una funcion */
        auto int    x,y;
        ...

        for (i=0; i< N ; i++)
        {
                int     a,b;        /* Variables automaticas en un bloque */
                ...
        }
```


38

}
# Variables externas

Las variables externas son aquellas que se definen fuera de cualquier función.

Para estas variables, se reserva espacio de memoria cuando se definen y permanecen hasta el termino del programa.

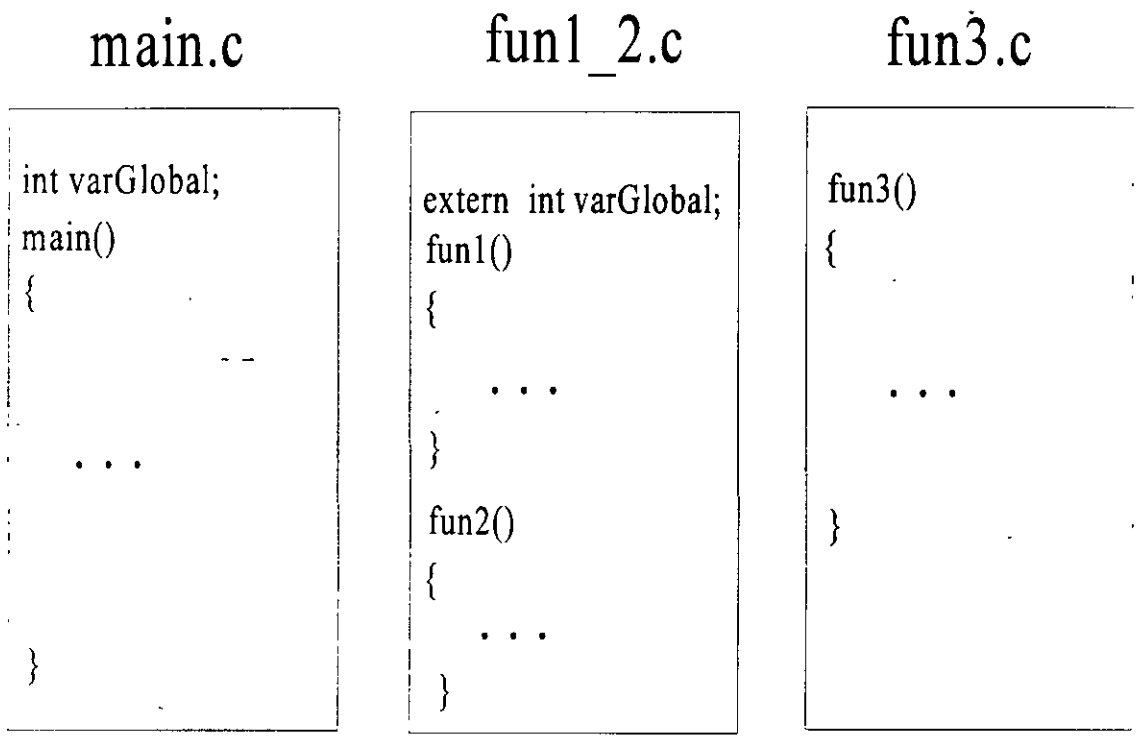
La declaracíon de una variable externa indica el tipo de ella, mientras que una definición, además reserva espacio de memoria para ella.

Para la declaración de una variable externa es necesario el calificativo *extern*.

Una variable externa se puede declarar muchas veces; pero solamente debe existir una definición de ella. Esto es especialmente útil cuando se tiene un programa con varios archivos fuentes, en todo el programa solamente se deberá definir una sola vez una variable; sin embargo, en todos los archivos en donde se desee utilizar, deberá ser declarada, de lo contrario los archivos fuente no podran ser compilados.

Todas las funciones que aparecen después de la definición(en el mismo archivo) de una variable externa pueden accesar a esta.

Ejemplo:



En este ejemplo la variable *varGlobal* se define en el archivo *main.c,* por lo que podra ser utilizada en la función *main.* Por otra parte, la variable es declarada en el archivo *fun1_2.c,* por lo que podra ser utilizada en las funciones *fun1* y *fun2.* En el archivo *fun3.c* no existe declaración alguna para la variable, por lo tanto, no podra ser utilizada en la función *fun3.*

Variables estáticas

Las variables estáticas son automáticas a un bloque o función; pero retienen su valor entre cada llamada a la función o ejecución del bloque en donde fueron definidas.

Las variables estáticas, solamente se inicializan una vez y conservan su valor entre cada llamada a la función.

Ejemplo:

```
/*
**      Programa 4_2
**
**      Este programa calcula el mayor de N numeros proporcionados por el usuario.
**      Muestra el comportamiento de las variables estaticas.
**
*/
#include      <stdio.h>
#define       N       10

int     maximo(short, int);

main()
{
        int num,i;

        printf("Proporciona %d numeros:\n",N);
        printf("1=> ");
        scanf("%d",&num);
        maximo(0, num);
        for ( i=2; i<= N; i++)
        {
                printf("%d=> ",i);
                scanf("%d",&num);
                maximo(1, num);
        }
        printf("\nEl numero mayor fue: %d\n",maximo(2,0));
```

```
}

/*
**      maximo
**
**      Funcion que controla las operaciones sobre el numero mayor del programa.
**
**      PARAMETROS:
**              opcion          operaciones a aplicar sobre el numero mayor:
**                              0       inicializar con el valor del parametro n.
**                              1       comparar el numero mayor con el parametro n para
**                                      obtener el nuevo valor mayor.
**                              2       unicamente regresa el valor mayor.
**              n               valor inicial (opcion=0) o valor a comparar con el mayor
**                              (opcion =1).
**
**      RETURN:
**              El valor mayor
**              -1 en caso de que el parametro opcion no sea valido.
*/
int     maximo(short opcion, int n)
{
        static int      max=0;

        switch (opcion)
        {
        case 0:         max = n;
                        return max;
        case 1:         max = (max > n ? max : n);
        case 2:         return max;
        default:        return -1;
        }
}
```

Inicialización

En ausencia de una inicialización explícita, las variables externas y estáticas se inicializan en cero.

En ausencia de una inicialización explícita, las variables automáticas se inicializan con valores indefinidos.

Las variables escalares se pueden inicializar cuando se definen:

```
int     x=1,
        j=5;
char    c='S';
```

Para variables externas y estáticas, el inicializador debe ser una expresión constante:

Para variables automáticas, el inicializador puede ser una constante o cualquier expresión que contenga valores previamente definidos, incluso llamadas a función:

Reglas de alcance

El alcance de una variable determina la parte del programa donde se esta se puede utilizar:

1. Para variables automáticas, el alcance de estas es el bloque o función en donde fueron definidas.

2. Las variables automáticas con el mismo nombre que estén en funciones diferentes no tienen relación. Lo mismo es válido para los parámetros formales de una función.

3. El alcance de una variable o función externa (todas las funciones son externas por default) abarca desde el lugar en donde se declaran hasta el fin del archivo fuente.

4. Si se hace referencia a una variable externa antes de su definición, o si esta definida en un archivo fuente diferente al que se esta utilizando, es obligatoria una declaración extern.

5. Para hacer llamadas a una función que se encuentra definida en un archivo fuente diferente, es conveniente hacer una declaración previa de la misma en el archivo en donde se desea hacer la llamada.

6. Cuando existe una definición de una variable externa y una automática con el mismo identificador, las referencias a través del identificador serán hacia la variable automática.

7. La declaración *static* aplicada a una variable o función externa, limita el alcance de ese objeto solamente al resto del archivo fuente.

Ejemplo:

```
/*
**      Programa 4_3
**
**      Este programa muestra el comportamiento de las reglas de alcance.
**      ¿Cual es la salida del programa? intentalo sin ejecutarlo.
**
*/
int  fun1(int);
int  fun2(int);

int     x=5,   y;

main()
{
        int x=10;

        printf("Valores al inicio del programa x=> %d, y=>%d\n",x,y);
        x = fun1(x);
        y = fun1(y);
        printf("Valores en main x=> %d, y=>%d\n",x,y);
        x = fun2(y);
        printf("Valores en main x=> %d, y=>%d\n",x,y);
}

int fun1(int  y)
{
        x += y++;
        printf("Valores en fun1 x=> %d, y=>%d\n",x,y);
        return x++;
}

int fun2(int  x)
{
        int  y=x;

        y += y;
        printf("Valores en fun2 x=> %d, y=>%d\n",x,y);
        return y;
```

}

Recursividad

Una función recursiva es aquella que se llama así misma directa o indirectamente. Es decir esta definida en terminos de la misma.

recursividad directa:

```
int     fun1()
{
        . . .
        fun1();
        . . .
}
```

recursividad indirecta:

```
int     fun1()
{
        . . .
        fun2();
        . . .
}

int     fun2()
{
        . . .
        fun1();
        . . .
}
```

Las funciones de C (excepto main) pueden ser recursivas.

Cada llamada recursiva reserva espacio para las variables automáticas que se definen en ella, los valores de estas variables para cada llamada se mantienen en un stack.

Las funciones recursivas deben incluir además de la llamada o llamadas recursivas sentencias para asegurar que la recursión terminará en algún momento.

Ejemplo:

```
/*
**      Programa 4_4
**
**      Este programa obtiene el factorial de un numero. El factorial de un numero es el
**      resultado de la multiplicacion del numero por el factorial del numero anterior.
**      Se toma como base que el factorial de o es 1.
*/
#include      <stdio.h>

int     factorial(int);

main()
{
        int     num;

        printf("Proporciona un numero=> ");
        scanf("%d",&num);
        printf("El factorial de %d es=> %d\n",factorial(num));
}
int     factorial(int n)
{
        return  (n == 0) ? 1: n*factorial(n-1);
}
```

El stack que se guardando los valores de las variables entre llamadas recursivas, cuando num es igual a 5 se muestra a continuación (cada renglon representa una llamada a la función):

n	return
0	1
1	1*factorial(0)
2	2*factorial(1)
3	3*factorial(2)
4	4*factorial(3)

5	5*factorial(4)

LABORATORIO

1. Haga un programa que obtenga el numero mayor y el menor de una serie de N números. Se deben utilizar dos funciones, una para obtener el mayor y otra para el menor. No utilice variables externas, tampoco estaticas. Las dos funciones deberan recibir dos parametros (los numeros a comparar) y regresar el valor mayor o menor, segun sea el caso.

2. Reconstruya su programa de empaquetamiento y desempaquetamiento del laboratorio anterior, de tal forma que se tengan 3 funciones: una que empaqueta la información proporcionada, otra que desempaqueta y otra que presenta la representación binaria del número en donde se guarda la información empaquetada.

3. Escriba una función que obtenga el número de Fibonacci n por medio de una función recursiva. Ejemplo: fib(8), debe dar como resultado 13.

NOTA: La definición de la serie de Fibonnaci se presenta en los laboratorios del capítulo 3.

EL PREPROCESADOR

C proporciona ciertas facilidades por medio de un preprocesador. El preprocesador actúa antes que el compilador ejecutando las instrucciones que comienzan con el carácter #, de tal forma que el archivo que se compila es realmente un archivo "preprocesado" que ya no incluye las instrucciones del preprocesador.

El alcance de las instrucciones que ejecuta el preprocesador es a nivel de archivo (definición de símbolos y macros son válidas solamente en el archivo en donde se hace la definición).

include

El preprocesador sustituye esta instrucción por el contenido del archivo especificado.

Sintaxis:

#include <archivo>
#include "archivo"

Cuando el nombre de archivo esta limitado por **<>**, el preprocesador busca el archivo en un directorio asignado por default (en UNIX generalmente es /usr/include y en compiladores para MS-DOS y windows este directorio es configurable), o en los directorios indicados al momento de la compilación.

Cuando el nombre de archivo esta limitado por **" "**, el preprocesador busca el archivo en el directorio de trabajo actual.

No existe restricción en cuanto al contenido del archivo.

El archivo que se sustituye puede contener instrucciones para el preprocesador, las cuales son ejecutadas una vez que se hace la sustitución.

El lenguaje C cuenta con un conjunto de librerias estandar, las cuales tienen asociadas "archivos de encabezados" (generalmente con extensión h) los cuales contienen los símbolos, tipos y prototipos utilizados con la librería en cuestion:

Archivo de encabezado	Contenido (prototipos, tipos y ctes. para)
alloc.h	alojación y liberación de memoria.
conio.h	control del display.
ctype.h	funciones de propósito general para caracteres.
dir.h	manejo de las estructuras asociadas a los directorios del sistema operativo.
errno.h	manejo de errores.
math.h	funciones matematicas.
signal.h	manejo de señales en el sistema operativo.
stdio.h	entrada/salida.
stdlib.h	funciones de propósito general.
string.h	manejo de strings.
sys/times.h	funciones para manejo de tiempos.
sys/types.h	estructuras de propósito general.

define

Esta instrucción del procesador permite la definición de símbolos y macros que son sustituidos a partir de que se da la definición.

Sintaxis:

```
#define     simbolo     token_string
#define     macro(simb1, ...,simbN)  token_string
```

#define simbolo token_string

El preprocesador sustituye cualquier ocurrencia del *simbolo* por el *token_string.* La sustitución se hace a partir de la definición solamente en el archivo en donde esta se encuentra. La sustitución no se lleva a cabo en comentarios y cadenas de caracteres encerradas por "".

Ejemplos:

```
#define     TRUE        1
#define     PI          3.1415
#define     SEG_X_DIA   (60 * 60 * 24)
```

El preprocesador solamente lleva a cabo la sustitución, si existen errores en la expresión reemplazada, estos son reportados por el compilador, en el lugar en la línea en donde se hizo la sustitución.

Las constantes simbólicas ayudan en la documentación al reemplazar lo que de otra forma sería una constante enigmática con un identificador nemónico, haciendo más portátil el programa permitiendo que se alteren en un solo lugar las constantes que pueden ser dependientes del sistema.

Una constante definida con **#define** puede revocarse con **#undef** (a partir del lugar en donde aparece, la definición ya no es válida):

#undef simbolo

#define macro(simb1, ...,simbN) token_string

Esta alternativa de la proposición **#define** sirve para la definición de macros, las cuales pueden entenderse como funciones pero realmente no lo son ya que no generan código.

No deben de existir blancos entre el identificador de macro y el primer paréntesis.

Ejemplo:

#define CUADRADO(x) ((x) * (x))

El parámetro "x" se sustituye cuando se encuentra una ocurrencia de la macro CUADRADO, en este caso si en algún lugar del archivo aparece CUADRADO(3), la sustitución se hara por ((3) * (3)).

Los paréntesis parecen excesivos, sin embargo son necesarios, suponiendo que la definición fuera la siguiente:

#define CUADRADO(x) (x * x)

si se utiliza la macro en la siguiente forma:

CUADRADO(7 + i)

la sustitución será:

(7 + i * 7 + i) lo que probablemente no generara los resultados deseados

Las macros son utilizadas para la sustitución de funciones que se pueden hacer en una línea de código:

#define MIN(x, y) ((x) < (y) ? (x) : (y))

De esta forma la macro puede servir para cualquier tipo de parámetros numéricos.

Cuando se hace la sustitución no se verifica sintaxis ni aspectos léxicos.

Ejemplo:

```
/*
**      Programa 4_5
**
**      Este programa calcula el mayor de N numeros proporcionados por el usuario.
**      Esta version proporciona una mejor documentacion gracias a la definicion de
**      simbolos.
**
*/
#define     N           10
#define     SET         0
#define     MATCH       1
#define     GET         2
#define     ERROR       -1

int     maximo(short, int);

main()
{
        int num,i;

        printf("Proporciona %d numeros:\n",N);
        printf("1=> ");
        scanf("%d",&num);
```

```
        maximo(SET, num);
        for ( i=2; i<= N; i++)
        {
                printf("%d=> ",i);
                scanf("%d",&num);
                maximo(MATCH, num);
        }
        printf("\nEl numero mayor fue: %d\n",maximo(GET,0));
}

/*
**      maximo
**
**      Funcion que controla las operaciones sobre el numero mayor del programa.
**
**      PARAMETROS:
**              opcion          operaciones a aplicar sobre el numero mayor:
**                      SET             inicializar con el valor del parametro n.
**                      MATCH           comparar el numero mayor con el parametro n para
**                                      obtener el nuevo valor mayor.
**                      GET             unicamente regresa el valor mayor.
**
**              n               valor inicial (opcion=0) o valor a comparar con el mayor
**                              (opcion =1).
**
**      RETURN:
**              El valor mayor
**              ERROR en caso de que el parametro opcion no sea valido.
*/
int     maximo(short opcion, int n)
{
        static int      max=0;

        switch (opcion)
        {
        case SET:       max = n;
                        return max;
        case MATCH:  max = (max > n ? max : n);
        case GET:       return max;
        default:        return ERROR;
```

```
    }
}
```

Compilación condicional

El preprocesador incluye algunas instrucciones que permiten llevar a cabo tareas de rastreo en los programas. También sirven para que un archivo de encabezado pueda incluirse sin ningún problema en cualquier programa, evitando conflictos por dobles definiciones.

Las líneas de control

```
#ifdef      simbolo
sentencias
#endif
```

permiten incluir las **sentencias** en el archivo a compilar, siempre y cuando se haya definido el **simbolo** anteriormente, la proposición **#ifndef** lo hace cuando el **simbolo** no ha sido definido.

Para la implementación de rastreo condicional de un programa se podría utilizar el siguiente esquema:

```
#define     DEBUG                    /* Solo se valida la definicion de DEBUG */
                                     /* por lo que no es necesario indicar el valor asociado al simbolo */
...
#ifdef      DEBUG
```

```
        printf("Valores obtenidos del proceso en el paso 1: %d, %d, %d\n",x,y,z);
#endif
```

Suponga que se crea un programa para cálculo de una nomina. Este programa utiliza una serie de funciones generales para el proceso. Las funciones estan definidas en un archivo: *funciones.c,* el cual tiene asociado su archivo de encabezados *(funciones.h)* en donde se definen los prototipos, simbolos y tipos utilizados en tales funciones.

Existe por otra parte un archivo llamado *tipos.h* en donde se definen los tipos de datos utilizados en todo el proceso de cálculo de nomina, dichas definiciones son incluidas en el archivos de encabezados funciones.h. En el archivo fuente del programa principal se incluyen los archivos *tipos.h* y *funciones.h:*

main.c

```
#include "tipos.h"
#include "funciones.h"
```

tipos.h

```
#define N 100
/* otras definiciones */
```

funciones.h

```
#include "tipos.h"

/* definiciones para funciones.c */
```

El código generado por el preprocesador antes de la sustitución de los simbolos finales sería:

```
#define N 100                          /* Codigo incluido por tipos.h*/
/* otras definiciones */

#define N 100                          /* Codigo incluido por funciones.h */
/* otras definiciones */

/* definiciones para funciones.c */
```

Para evitar dobles definiciones, principalmente de tipos, se pueden utilizar las sentencias **#ifdef** y **#ifndef** en *tipos.h:*

```
#ifndef     _TIPOS_H  /* Las instrucciones solo se incluyen cuando _TIPOS_H no ha sido */
#define     _TIPOS_H        /* definido, lo cual solo se da una sola vez, ya que en el mismo */
                                                    /* bloque de ifndef-endif se define */
#define     N     100
/* otras definiciones */

#endif
```

División de un programa en varios archivos

Un programa en C puede constar de varios archivos fuente; pero en ellos solamente debe definirse una sola función main.

Se pueden hacer llamadas a una función desde cualquier función no importando el archivo fuente en donde se defina; pero debe de existir una declaración en el archivo en donde se encuentra la función que hace la llamada.

Las variables globales pueden ser accesadas desde cualquier función; pero, debe haber una declaración de la variable en el archivo fuente en donde se utilice.

Es recomendable que se definan archivos de encabezado que contengan los símbolos, tipos y prototipos de las funciones definidas en un archivo fuente, para que de esta forma se utilice la proposición **#include** con el archivo de encabezados en todos los archivos fuente donde se pretenda utilizar dichas funciones.

Ejemplo:

```
/*
**      lab_4_6.c
**
**      Archivo que contiene la función main del Programa lab_4_6
*/
/*
**      Programa 4_6
**
**      Este programa calcula el mayor y menor de N numeros proporcionados por el
**       usuario.
**      Se utilizan varios archivos fuente para la implementación del programa
**      simbolos.
**
*/
#include      <stdio.h>
#include      "maxMin.h"

#define       N             10

main()
{
        int num,i;

        printf("Proporciona %d numeros:\n",N);
        printf("1=> ");
        scanf("%d",&num);
        maximo(SET, num);
        minimo(SET,num);
        for ( i=2; i<= N; i++)
        {
                printf("%d=> ",i);
                scanf("%d",&num);
                maximo(MATCH, num);
                minimo(MATCH, num);
        }
        printf("\nEl numero mayor fue: %d\n",maximo(GET,0));
        printf("\nY el numero menor fue: %d\n",minimo(GET,0));
}
```

```
/*
**      maxMin.c
**
**      Archivo para la definicion de las funciones maximo y minimo.
*/
/*
**      maximo
**
**      Funcion que controla las operaciones sobre el numero mayor del programa.
**
**      PARAMETROS:
**              opcion          operaciones a aplicar sobre el numero mayor:
**                      SET             inicializar con el valor del parametro n.
**                      MATCH           comparar el numero mayor con el parametro n para
**                                      obtener el nuevo valor mayor.
**                      GET             unicamente regresa el valor mayor.
**
**              n               valor inicial (opcion=0) o valor a comparar con el mayor
**                              (opcion =1).
**
**      RETURN:
**              El valor mayor
**              ERROR en caso de que el parametro opcion no sea valido.
*/

#include      "maxMin.h"

int     maximo(short opcion, int n)
{
        static int      max=0;

        switch (opcion)
        {
        case SET:    max = n;
                     return max;
        case MATCH:  max = (max > n ? max : n);
        case GET:    return max;
        default:     return ERROR;
        }
```

```
}

/*
**      minimo
**
**      Funcion que controla las operaciones sobre el numero menor del programa.
**
**      PARAMETROS:
**              opcion          operaciones a aplicar sobre el numero menor:
**                      SET             inicializar con el valor del parametro n.
**                      MATCH           comparar el numero menor con el parametro n para
**                                      obtener el nuevo valor menor.
**                      GET             unicamente regresa el valor menor.
**
**              n               valor inicial (opcion=0) o valor a comparar con el menor
**                              (opcion =1).
**
**      RETURN:
**              El valor menor
**              ERROR en caso de que el parametro opcion no sea valido.
*/

int     minimo(short opcion, int n)
{
        static int      min=0;

        switch (opcion)
        {
        case SET:       min = n;
                        return min;
        case MATCH:  min = (min > n ? min : n);
        case GET:       return min;
        default:        return ERROR;
        }
}
```

```
/*
**      maxMin.h
**
**      Archivo para definicion de prototipos y simbolos utilizados en las funciones del
**      archivo fuente maxMin.c
**
*/

#ifndef     _MAX_MIN_H
#define     _MAX_MIN_H

#define     SET         0
#define     MATCH       1
#define     GET         2
#define     ERROR       -1

int     maximo(short,int);
int     minimo(short,int);

#endif
```

LABORATORIO

1. Implemente el programa de empaquetamiento y desempaquetamiento en tres archivos, uno para el main, otro para las funciones de empaquetamiento y desempaquetamiento y otro para la función que imprime la representación binaria de un número.

Página intencionalmente blanca.

ARREGLOS
Y
APUNTADORES

ARREGLOS

Un arreglo es una colección de elementos del mismo tipo.

Un arreglo se define de la siguiente forma:

tipo nombre[tamaño];

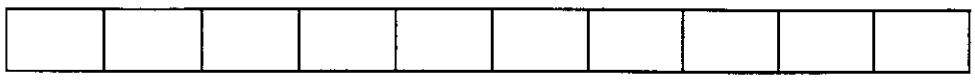
donde:	tipo	tipo de los elementos que constituyen el arreglo.
	nombre	identificador del arreglo.
	tamaño	número de elementos del arreglo.

Para accesar a los elementos de un arreglo hay que hacer referencia a ellos mediante un índice. El primer elemento del arreglo es tiene como índice 0, por lo tanto, para el último el índice es *tamaño-1.*

Cuando se define un arreglo se reservan localidades contiguas de memoria para almacenar cada uno de sus alementos, aún cuando el arreglo sea multidimensional.

Ejemplo:

int v[10];



v[0] v[1] v[2] v[3] v[4] v[5] v[6] v[7] v[8] v[9]

Inicialización de arreglos

Los arreglos externos y estáticos numéricos inicializan sus elementos con cero.

Los arreglos externos y estáticos de caracteres, inicializan sus elementos con el caracter cuyo código ASCII es '\0'.

Los arreglos pueden inicializarse de forma explícita de la siguiente forma:

```
int    x[5] = { 2, 6, 8, 12, 28};
```

En una inicialización explicita:

- El número de inicializadores puede ser menor que el número de elementos en el arreglo, en este caso los elementos restantes se inicializan con cero:

```
int    x[10] = { 4, 5, 7};
```

- Es un error el que el número de inicializadores sea mayor que el tamaño del arreglo.

- No es necesario especificar su dimensión, la dimensión sera el número de inicializadores:

```
int x[] = { 1, 5, 5, 7};      /* Se define un arreglo de 4 elementos: int x[4] */
```

Arreglos de caracteres

En el lenguaje C no existe el tipo *string* o *cadena,* sin embargo, una cadena de caracteres puede ser representada con un arreglo de caracteres.

Para poder llevar a cabo operaciones con la cadena de caracteres, por ejemplo, concatenaciones o busqueda de subcadenas en la misma, es necesario determinar en que posición dentro del arreglo termina la cadena.

Las funciones estandar para manejo de arreglos de caracteres asumen que el final de una cadena lo determina el caracter '\0', esto no impide que el programador pueda construir funciones para manipulación de cadenas utilizando sus convenciones.

Es importante remarcar, que si se desea trabajar con arreglos de cadenas utilizando las funciones de la libreria estandar, se debe tomar en cuenta esta convención. Todas las librerias disponibles para C toman esta convención como un estandar.

Por lo tanto, si se desea manipular una cadena de longitud N, se debera definir un arreglo de tamaño N+1, ya que se tiene que considerar un elemento adicional para el terminador.

Es responsabilidad del programador asegurarse de que no se excedan los límites de la cadena.

Las constantes cadena se escriben entre comillas:

```
"HOLA MUNDO"    /* La cadena tiene 11 elementos (incluyendo el terminador) */
```

Es un error común utilizar constantes como **'a'** y **"a"** en forma indistinta; hay que recordar, que la primera representa un solo caracter y la segunda una cadena de dos caracteres.

La inicialización de arreglos de caracteres se puede hacer de una forma semejante a la inicialización de arreglos numéricos

```
char   mensaje[] = { 'H', 'O', 'L', 'A', ' ', 'M', 'U', 'N', 'D', 'O', '\0'};
```

o bien:

```
char    cadena[] = "HOLA MUNDO";
```

Ejemplo:

```
/*
**      Programa 5_1
**
**      Este programa simplemente lee el nombre de una persona y lo almacena
**      como una cadena de caracteres; sin embargo, no se utilizan las funciones
**      de la libreria estandar para manejo de strings.
**
*/

#include    <stdio.h>
#define     MAX_NOM_LONG  100

main()
{
        int     i=0;
        char    nombre[N],
                c;

        printf("\tDame tu nombre: ");
        while ((c = getchar()) != '\n')        /*Cada caracter leido es almacenado como elemento del arreglo*/
                nombre[i++] = c;
        nombre[i] = '\0';                      /* Se agrega el terminador */
        printf("\nMuchas gracias %s por haber dado tu nombre\n",  nombre);
}
```

Arreglos multidimensionales

En un arreglo multidimensional, las localidades de memoria que se reservan, al igual que en un arreglo unidimensional, son contiguas; sin embargo, se utiliza una función de mapeo de direcciones en base a las dimensiones del arreglo para determinar la posición en donde se encuentra cada elemento.

El programador no debe preocuparse por el mapeo que se hace al "arreglo unidimensional".

Los arreglos multidimensionales más utilizados son aquellos de dos dimensiones, por ejemplo, en la representación de matrices.

La forma de definir un arreglo multidimensional, por ejemplo de dos dimensiones, es la siguiente:

```
int    matriz[10][10];  /* Se reserva localidades de memoria continua para 100 enteros */
```

En este caso el primer índice representa los renglones y el segundo las columnas; sin embargo, esto no implica que el compilador maneje un arreglo de dos dimensiones como un conjunto de renglones y columnas. Este manejo depende totalmente del programador, de tal forma que se podría manipular el arreglo de tal forma que el primer índice representara a las columnas y el segundo a los renglones.

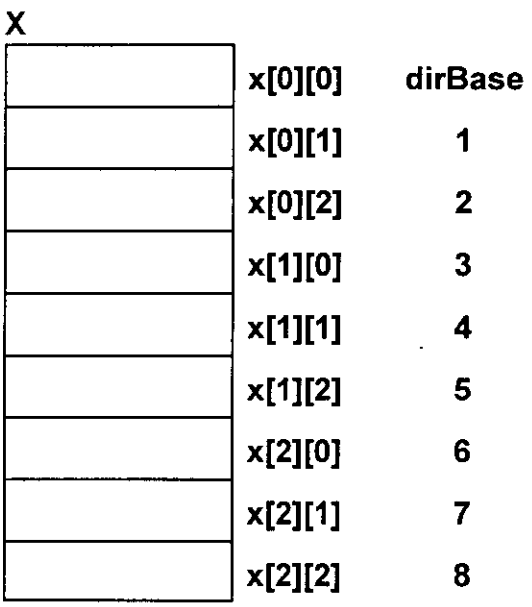
La función de mapeo o transformación de direcciones se utiliza para calcular la posición de un elemento en el espacio de memoria ocupado por el arreglo en base a sus índices; por ejemplo, para un arreglo bidimensional, la función sería la siguiente:

dirBase + n*i + j

donde:	dirBase	dirección de inicio del arreglo
	n	tamaño de la segunda dimensión
	i	primer índice del elemento
	j	segundo índice del elemento

Ejemplo:

```
int   x[3][3];
```



Para el ejemplo anterior, el elemento x[1][2] se encuentra 3*1 + 2 = 5 posiciones después de la dirección de inicio del arreglo.

Como se puede observar, para la función de mapeo de un arreglo bidimensional, no se utiliza la primera dimensión con la que fue definido el arreglo.

Cuando se inicializa un arreglo multidimensional (N dimensiones) es necesario especificar por lo menos N-1 dimensiones para que se pueda llevar a cabo el mapeo de direcciones.

La inicialización de arreglos multidimensionales es muy parecida a la de los arreglos unidimensionales:

Ejemplo:

Un arreglo de dos dimensiones, es un arreglo de arreglos:

int x[3][3] = { { 3, 6, 9}, { 8, 5, 1}, { 1, 1, 5}};

Sin ambargo, se almacena como un arreglo unidimensional:

int x[3][3] = { 3, 6, 9, 8, 5, 1, 1, 1, 5};

Si se inicializan todos sus elementos, no se necesita indicar la primera dimensión, en este caso se crea un arreglo con tantos "renglones" como sean necesarios para alojar a todos los valores de la inicialización:

int x[][3] = { 3, 6, 9, 8, 5, 1, 1, 1, 5};

LABORATORIO

1. Escriba una función que haga una búsqueda secuencial de un elemento sobre un arreglo. La función debe regresar como valor la posición en donde se encuentra el elemento ó -1 si no se encuentra.

APUNTADORES

Todas las variables se almacenan en memoria ocupando cierto número de bytes dependiendo del tipo de la misma. El lugar en donde se almacena la variable determina su dirección de almacenamiento.

Un apuntador es una variable que almacena una dirección de memoria.

Los apuntadores son utilizados para manejo de memoria dinámica, para simular paso de parámetros por variable y para la creación de estructuras de datos complejas. El manejo de apuntadores es la base para el desarrollo de aplicaciones complejas que tienen por objetivo la eficiencia del código que utilizan.

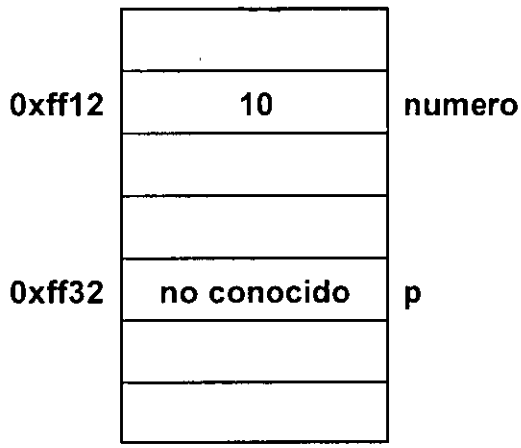
Un apuntador puede contener la dirección de una variable entera, caracter, double, de un apuntador, etc.; el tipo de esta variable, determina el tipo de apuntador: apuntador a entero, apuntador a carácter, apuntador a double, apuntador a apuntador, etc. Por ejemplo, para definir un apuntador a entero:

```
int          *p;
```

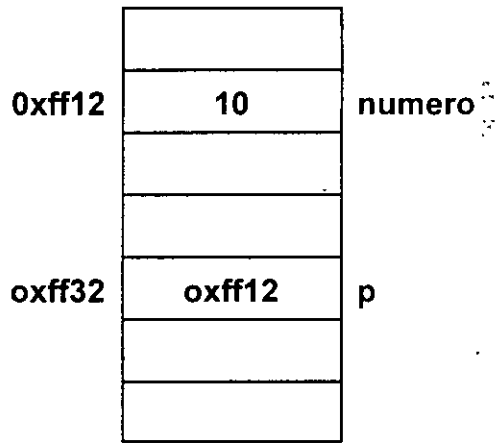
En este caso se esta definiendo una variable apuntador a entero. La dirección que contiene el apuntador, al momento de ser definido, es una dirección desconocida (en general, no es una dirección de memoria que se pueda accesar sin peligro de dañar otros procesos), puesto que este no ha sido inicializado.

Para asignar a un apuntador una dirección válida se puede utilizar el operador de dirección **&**, el cuál obtiene la dirección de una variable. Por ejemplo, supongase el siguiente fragmento de código:

```
int     numero;
int     *p;

numero = 10;
p = &numero;
```



numero = 10



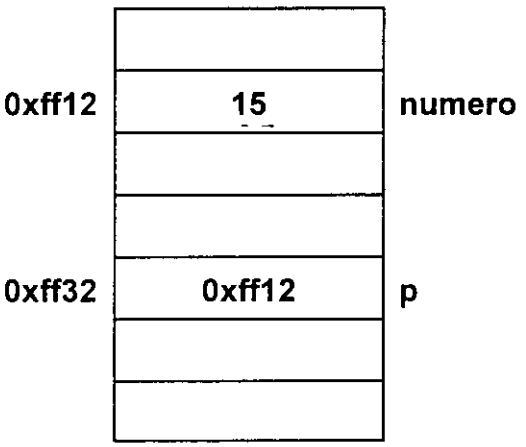
p = &numero

El ejemplo anterior asigna a **p** la dirección de **numero** y se puede accesar el valor de la variable **numero** directamente o indirectamente por medio del apuntador p.

El apuntador de dirección & es unuario y solamente se aplica a variables.

Para poder hacer referencia a la memoria direccionada por el apuntador, se utiliza el operador de desreferencia o indirección *. Por ejemplo, para cambiar el valor de la variable entera **numero** en forma indirecta (en este caso, el ejemplo no muestra un caso común del manejo de apuntadores):

***p = 15;**



***p = 15**

en este caso, la expresión ***p** es una variable entera y se puede utilizar en cualquier contexto que acepte expresiones enteras:

Ejemplo:

Suponga las siguientes definiciones:

```
int       x = 2, z, *p, *q;
double    d;
```

Analicemos la siguiente secuencia de operaciones:

```
p = &x;                          /* p contiene la direccion de x */
z = *p + 1;          /* A z se le asigna el valor de x + 1 (puesto que p apunta a x) => y = 3 */
d = sqrt(*p);          /* A d se le asigna la raiz cuadrada de el valor de x => d = sqrt(2) */

*p = 0;                                  /* El valor de x ahora es cero */
*p += 1;                              /* El valor de x se incrementa en uno=> x=1 */
(*p)++;          /* El valor de x se vuelve a incrementar (son importantes los parentesis,
                 debido a la precedencia de los operadores */

q = p;                          /* El apuntador q tambien se le asigna la direccion de x */
```

Los apuntadores se pueden inicializar al momento de ser definidos:

```
int     x = 4,
        *p = &x;
```

No se puede asignar a un apuntador una dirección de una variable que no es del tipo del apuntador:

```
int             *p;
double          f;

p = &f;     /* no es valido */
```

Generalmente muchos programas no conocen la cantidad de memoria que van a requerir, por ejemplo, si queremos crear un arreglo para guardar los nombres de los clientes de una empresa, de que tamaño debera ser el arreglo?, ¿qué sucede si se reserva más o menos memoria de la que se necesita?. Un programa eficiente solo debería ocupar la memoria que realmente requiere. Para solucionar este problema, se maneja el concepto de **memoria dinámica**, es decir, se reserva memoria a tiempo de ejecución.

Para reservar memoria a tiempo de ejecución, se utiliza la función **malloc,** la cuál reserva ń bytes de memoria y regresa como valor un apuntador generico (void *) el cuál se puede asignar a una variable apuntador utilizando una conversión explicita:

```
int     *p;
p  =  (int *)malloc(sizeof(int));
```

el operador sizeof obtiene el número de bytes que ocupa un tipo.

Apuntadores como parámetros de funciones

La forma de pasar parámetros a las funciones es por valor, esto implica, que la función no puede cambiar los valores de los parámetros actuales.

Para que en una función pueda cambiar el valor de una variable, no definida en la misma función, en primera instancia, es necesario que este definida como externa:

Ejemplo:

```
/*
**      Programa 5_2
**
**      Este programa muestra el manejo de variables externas como una alternativa al
**      paso de parámetros por referencia
**
*/

#include    <stdio.h>
#define     N     10

int     max;
void    maximo(int,int);

main()
{
        int i, num;

        printf("Proporciona 10 numeros:\n");
        scanf("%d", &num);
        max = num;
        for (i=1; i < N; i++)
        {
                scanf("%d", &num);
                maximo(max,num);
        }
        printf("\nEl numero mayor es: %d\n", max);
}

void    maximo(int x,int y)
{
        max = x > y ? x : y;

}
```

Otra forma de regresar un valor a la función que hace la llamada es con el uso de la sentencia return en una función.

Ejemplo:

```
/*
**      Programa 5_3
**
**      Este programa muestra el regreso de valores en la cláusula return, como una
**      alternativa para obtener valores de regreso de la llamada a una funcion.
**
*/

#include    <stdio.h>
#define     N       10

int maximo(int,int);

main()
{
        int     i, num, max;

        printf("Proporciona 10 numeros:\n");
        scanf("%d", &num);
        max = num;
        for (i=1; i < N; i++)
        {
                scanf("%d", &num);
                max = maximo(max, num);
        }
        printf("\nEl numero mayor es: %d\n", max);
}

int     maximo(int x, int y)
{
        return ( x > y ? x : y);
}
```

Sin embargo:

- El uso de variables externas no es muy recomendable por la programación estructurada.

- El uso de return solamente permite el regreso de un valor.

- Muchas funciones necesitan regresar más de un valor.

Cuando se utilizan apuntadores como parámetros, el valor de las variables, direccionadas por el apuntador, puede cambiar en la llamada a una función.

Consideremos el diseño de la función swap, esta función deberá intercambiar el valor de sus dos parámetros:

```
void    swap(int a, int b)
{
        int aux;

        aux = a;
        a = b;
        b = aux;
        printf("\nLos valores son a = %d y b = %d", a, b);
}
```

¿Cuál será la salida generada por el programa 5_4 que hace utiliza la función swap previamente definida?

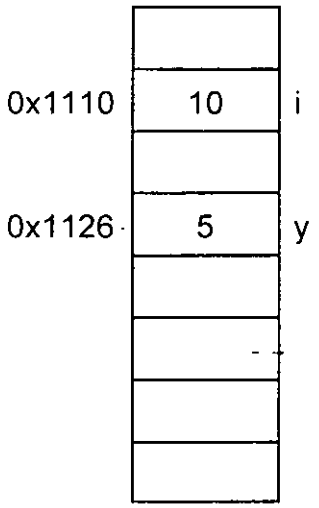
```
/*
**      Programa 5_4
**
**      Programa que implementa la función swap sin manejo de apuntadores
**
*/

#include <stdio.h>

void swap(int,int);

main()
{
        int     i = 10, y = 5;

        swap(i, y);
        printf("\nLos valores son i = %d y y = %d", i, y);
}

void    swap(int a, int b)
{
        int aux;

        aux = a;
        a = b;
        b = aux;
        printf("\nLos valores son a = %d y b = %d", a, b);
}
```

Consideremos el definir apuntadores a entero como parametros, como lo muestra el programa 5_5:

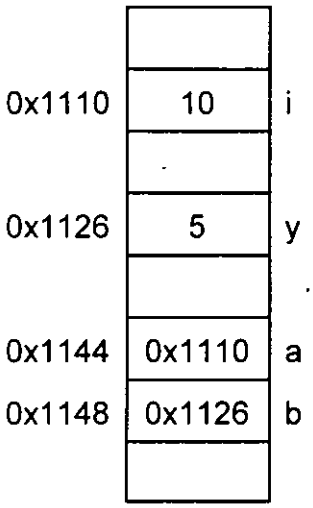
```
/*
**      Programa 5_5
**
**      Programa que implementa la función swap conmanejo de apuntadores
**
*/

#include <stdio.h>

void swap(int *, int *);

main()
{
        int     i = 10, y = 5;

        swap(&i, &y);
        printf("\nLos valores son i = %d y y = %d", i, y);
}

void    swap(int *a, int *b)
{
        int     aux;

        aux = *a;
        *a = *b;
        *b = aux;
        printf("\nLos valores son a = %d y b = %d", *a, *b);
}
```

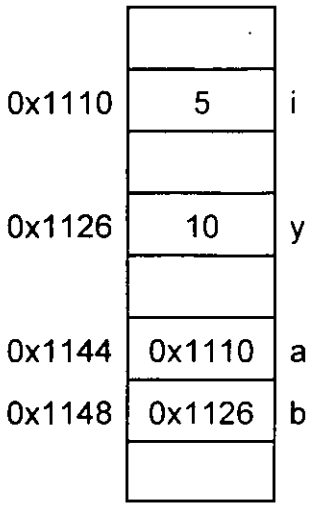
¿ Cuál es la salida que produce el programa?

El comportamiento de la función swap se muestra con la siguiente figura:

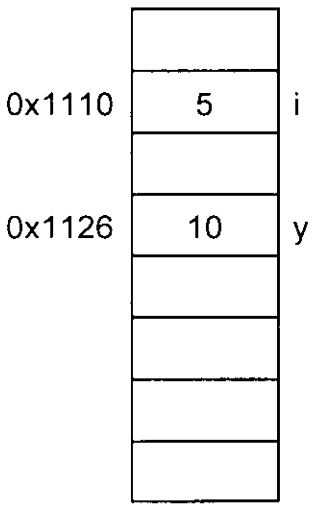


antes de la llamada a swap





en la llamada a swap



después de la llamada a swap

Para simular el paso de parámetros por variable, se debe considerar:

- Definir los parámetros formales como apuntadores.
- En la definición de la función, hay que utilizar el operador de indirección al momento de accesar la memoria direccionada por los apuntadores.
- El parámetro actual en la llamada a la función debe ser una dirección.

Apuntadores y arreglos

Existe una gran relación entre apuntadores y arreglos.

El nombre de un arreglo es una variable donde se guarda la dirección de inicio del arreglo, su valor no puede ser modificado.

Muchas veces los apuntadores y los arreglos son utilizados con el mismo propósito; sin embargo, cabe recordar que un apuntador es una variable y un arreglo es un apuntador constante.

Cuando se define un arreglo de tamaño N, se reservan N localidades contiguasde memoria. Por otra parte, cuando se define un apuntador solamente se reserva el espacio para la variable que representa.

Suponga las siguientes definiciones:

```
int     x[10] = {1,2,3,4,5,6,7,8,9,10},
        *p;
```

las siguientes expresiones son validas:

```
p = x;                          /* es equivalente a p = &x[0] */

p =  x + 1;                     /* utilizando notación de apuntadores, p apunta a x[1] */

p++;                            /* p apunta ahora a x[2] */

*(x + 5) = 10;                  /* x[5] = 10  */

*(p + 10) = 15;                 /* es válido pero se accesa una  localidad de memoria no válida */
```

la siguientes expresiones no son válidas:

```
x++;                            /* x es un apuntador constante   */

*(x + 10) = 20;                 /* x es la dirección de un arreglo  y la localidad máxima que se puede
                                   accesar es x + 9 */
```

Ejemplo:

```
/*
**      Programa 5_6
**
**      Programa que ayuda a comprender el manejo de apuntadores y arreglos.
**      Intente determinar la salida del programa sin ejecutarlo.
**
*/

#include      <stdio.h>

main()
{
      char          x[] = "ESTA ES UNA CADENA EJEMPLO";
      char          *p = x,
                    *q = x + 2;

      printf("%d %c %c %d %d %c", *p, p[0], *p + 1, *(p +5),  q[2] + 3, *x);
      p+=2;

      printf("%d %c %c", p[0], **&p + 5, *(p + 4));

      printf("%c %c %c", *(p++), *p, *(p + 1));
}
```

¿ Cuál es la salida del programa ?

Arreglos como parámetros de funciones

Cuando un arreglo es pasado como parámetro a una función, en realidad se esta pasando una dirección, ya que el nombre de un arreglo es la dirección del primer elemento del mismo.

El parámetro formal puede ser definido como arreglo a como apuntador, el parametro actual puede ser un apuntador o un arreglo.

En el caso de que el parámetro formal sea definido como un arreglo unidimensional no es necesario especificar el tamaño, en este caso es responsabilidad del programador no rebasar los límites del arreglo en la función.

En el caso de arreglos multidimensionales es necesario especificar todas las dimensiones a excepción de la primera, esta no es necesaria ya que no se utiliza en la fórmula de transformación de direcciones.

En el caso de arreglos multidimensionales, las dimensiones especificadas en el parámetro formal pueden no ser las mismas que las del parámetro actual. La función lo único que recibe es una dirección de inicio e información para accesar los elementos por medio de la fórmula de mapeo.

Ejemplo:

```
/*
**      Programa 5_7
**
**      Programa que muestra el paso de arreglos como parametros en funciones.
**
*/
#include <stdio.h>

void  f(int [][3]);

main()
{
        int     matriz[4][4] = { { 1, 2, 3, 4 },
                                 { 5, 6, 7, 8 },
                                 { 9, 10, 11, 12 },
                                 { 13, 14, 15, 16}};

        f(matriz);

}

/*
**      f
**
**      Esta funcion recibe como parametro un arreglo bidimensional. En la funcion,
**      el arreglo se manipula con 3 "columnas".
**
**      PARAMETROS:
**              a       direccion de inicio del arreglo.
**
*/
void    f(int a[][3])
{
        int     i;

        for(i=0; i < 5; i++)
                printf("%d ", a[i][2]);
```

Aritmética de apuntadores

La aritmética de apuntadores es una de las características más eficaces del lenguaje C.

Si p es un apuntador a un tipo de dato e inicialmente tiene una dirección x, por ejemplo la dirección 0x1876; la expresión p + 1 no es necesariamente la dirección 0x1877. El incremento no es unitario necesariamente, sino que depende del número de bytes que se necesiten para almacenar un elemento del tipo al cual direcciona el apuntador.

En el caso de que p fuera una apuntador a int y que el tipo int ocupará 2 bytes, p++ ocasionaría que p apuntara dos bytes adelante de su dirección original.

Se pueden manejar sumas y restas de direcciones exclusivamente.

En forma general se puede decir que la aritmética de apuntadores no es igual a la de enteros.

Funciones para manejo de caracteres y cadenas

Las funciones que se mencionan a continuación, permiten determinar la naturaleza de un carácter, todas ellas reciben como parámetro un valor numérico o char.

Para poder utilizar estas funciones es necesario especificar el header **ctype.h**

Todas estas funciones regresan un valor verdadero, si el carácter es del tipo que se esta validando y un valor de cero en cualquier otro caso.

NOMBRE DE FUNCION	PROPOSITO
isalnum(int c)	Determina si c es alfanumérico
isalpha(int c)	Determina si c es letra
iscntrl(int c)	Determina si c es carácter de control
isdigit(int c)	Determina si c es dígito
isprint(int c)	Determina si c es imprimible
islower(int c)	Determina si c es minúscula
isspace(int c)	Determina si c es blanco
isupper(int c)	Determina si c es mayúscula

El lenguaje C cuenta con un conjunto de funciones para manejo de cadenas, recuerde que en este caso se considera una cadena a un arreglo de caracteres terminado con '\0'. Los prototipos de estas funciones se encuentran en el archivo **string.h**

strcpy

char *strcpy(char *s1, char *s2)

Copia s2 a s1, incluye en s1 el carácter '\0'.

Regresa como valor s1.

strncpy

char *strncpy(char *s1, char *s2, int n)

Copia n caracteres de s2 a s1, incluye en s1 el carácter '\0'.

Regresa como valor s1.

strcat

char *strcat(char *s1, char *s2)

Concatena s2 a s1 incluye en s1 el carácter '\0'.

Regresa como valor s1.

strncat

char *strncat(char *s1, char *s2, int n)

Concatena n caracteres de s2 a s1 incluye en s1 el carácter '\0'.

Regresa como valor s1.

strcmp

int strcmp(char *s1, char *s2)

Compara las cadenas s1 y s2, no compara la longitud de ellas, sino el orden lexicográfico de cada uno de sus caracteres.

Regresa:

menor a cero si s1 < s2,
mayor a cero si s1 > s2,
cero si s1 = s2

strncmp

int strncmp(char *s1, char *s2, int n)

Compara a lo más n caracteres de las cadenas s1 y s2.

Regresa:

menor a cero si s1 < s2,
mayor a cero si s1 > s2,
cero si s1 = s2

strchr

char *strchr(char *s1, int c)

Busca la primera ocurrencia del carácter c en s1.

Regresa la dirección de la primera ocurrencia de c en s1o NULL si c no esta en s1.

strlen

int strlen(char *s1)

Calcula la longitud de s1 no contando el terminador.

Regresa la longitud de la cadena.

A continuación se muestra la implementación de algunas de ellas:

```
int strcmp(char *s, char *t)
{
        for( ; *s == *t ; s++, t++)
                if (*s ==  '\0')
                        return 0;
        return *s - *t;
}


char *strcat(char *s1, char *s2)
{
        char            *aux = s1;

        while(*aux++)
                ;
        --aux;
        while (*aux++ = *s2++)
                ;
        return s1;
}


int strlen(char *s)
{
        int     n=0;

        while(*s++)
                n++;
        return n;
}
```

LABORATORIO

1. Escriba una función que determine si una cadena de caracteres es un palíndromo. Un palíndromo es una cadena de caracteres que se lee igual hacia adelante que hacia atrás.

2. Escriba una función llamada substr que busque la ocurrencia de una cadena en otra y que regrese como valor la posición a partir de la que se encuentra; en caso de que no se encuentre, la función deberá regresar -1.

Arreglos de apuntadores

El lenguaje C permite la definición de tipos complejos a partir de los tipos básicos, de esta forma se pueden crear arreglos de apuntadores, arreglos de arreglos de apuntadores, etc.

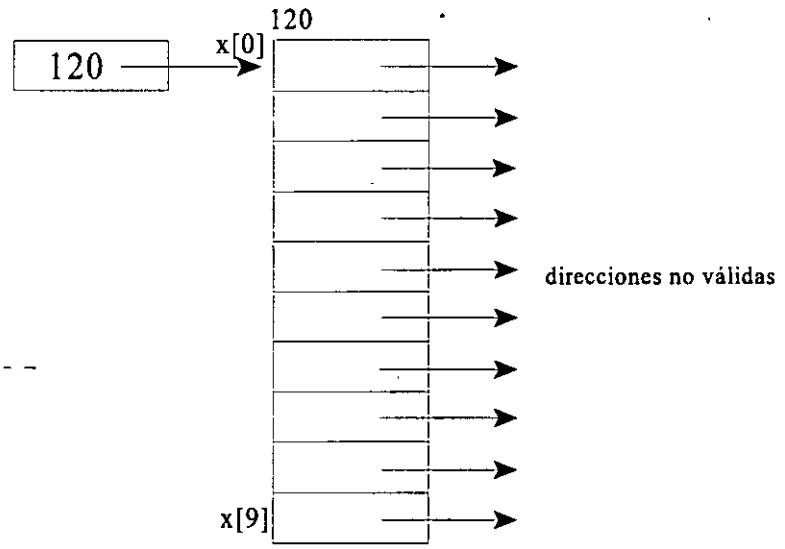
Uno de los tipos complejos más utilizados son los arreglos de apuntadores, ya que son mucho más flexibles que los arreglos multidimensionales, además de que generalmente, requieren de menos memoria.

La forma de definir un arreglo de apuntadores a carácter sería la siguiente:

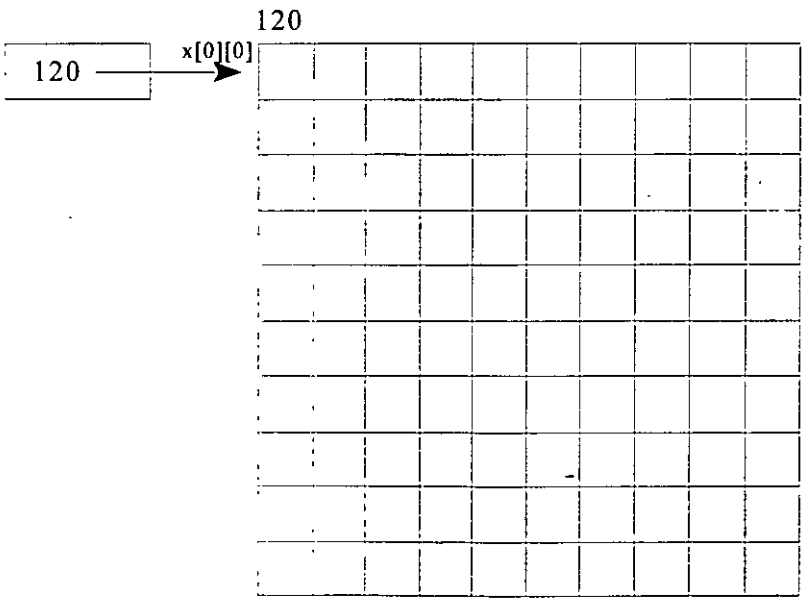
```
char        *x[10];
```

La representación interna de este arreglo es muy diferente a la de un arreglo bidimensional de caracteres, aunque la forma de hacer referencia a cada uno de sus elementos sea muy parecida.

```
char    *x[10];
```




```
char    x[10][10];
```



los elementos se almacenan en localidades contiguas

Una vez creado el arreglo de apuntadores, se deberá asignar a cada uno de los elementos direcciones de memoria previamente reservadas, esto se puede hacer dinamicamente a tiempo de ejecución de la forma siguiente:

```
x[0] = (char *)malloc(..);
```

De esta forma cada uno de los elementos se puede considerar como un arreglo de N caracteres, donde N no esta limitado más que por la cantidad de memoria que pueda tener disponible la máquina.

Ejemplo:

```
/*
**      Programa 5_7
**
**      Este programa permite la implementación de una agenda en donde se
**      guardan los nombres de los amigos de una persona. La agenda es
**      ordenada posteriormente.
**      Se utiliza un arreglo de apuntadores a caracter para la implementación
**      de la agenda, de esta forma no se desperdicia memoria por los
**      nombres cortos.
**
*/
#include <stdio.h>

#define     MAX         20    /* Máximo número de elementos en el arreglo */
#define     MAX_NOM     35    /* Longitud maxima de un nombre de persona */

void    ordena(char *[], int);
void    imprime(char *[], int);

main()
{
        char    *agenda[MAX],
                nombre[MAX_NOM];
         int    i=0,j;

        printf("\nProporciona el nombre de tus amigos:\n");
        while (gets(nombre) != NULL)
        {
                /*
                ** Se utiliza la variable nombre como un buffer para leer los nombres
                ** de las personas. Una vez leido el nombre, se reserva la memoria
                ** necesaria para almacenarlo, la dirección de inicio de esta se asigna
                ** a un elemento del arreglo y se copia el nombre en dicha direccion.
                */
                agenda[i] = (char *)malloc((strlen(nombre)+1) * sizeof (char));
                strcpy(agenda[i], nombre);
                i++;
```

```
        }
        ordena(agenda, i);
        printf("\n\nLista Ordenada:\n");
        imprime(agenda, i);
}

/*
**      ordena
**
**      Esta funcion ordena las cadenas apuntadas por los elementos de un arreglo de
**      apuntadores a caracter. Se utiliza el metodo conocido como burbuja.
**
**      PARAMETROS:
**              x               arreglo de apuntadores a caracter.
**              n               numero de elementos del arreglo.
**
**      RETURN:
**              ninguno
*/
void    ordena(char *x[], int n)
{
        int      i,j;
        char    *aux;

        for(i=0; i < n - 1; i++)
                for (j = n - 1; i<j; j--)
                        /*
                        ** Las cadenas se tienen que comparar caracter por caracter,
                        ** como lo hace la funcion strcmp.

                        */
                        if ((strcmp(x[j-1], x[j])) > 0)
                        {
                                /*
                                ** Solo se intercambian direcciones.
                                */
                                aux = x[j-1];
                                x[j-1] = x[j];
                                x[j] = aux;
```

```
                }
}

/*
**      imprime
**
**      Esta funcion imprime las cadenas de un arreglo de apuntadores a caracter.
**
**      PARAMETROS:
**              x               arreglo de apuntadores a caracter.
**              n               numero de elementos del arreglo.
**
**      RETURN:
**              ninguno
*/
void    imprime(char *x[], int n)
{
        int     i;

        for(i=0; i < n; i++)
                printf("%s\n", x[i]);
}
```

Parámetros para main

En la función main se pueden utilizar dos parámetros para establecer comunicación con el sistema operativo al momento que se lleva a cabo la ejecución del programa.

Los argumentos se llaman **argc** y **argv.** El primero de ellos almacena el número de argumentos en la línea de comandos que recibe el programa ejecutable y el segundo es una arreglo de apuntadores a char en donde se almacenan dichos argumentos.

El primer elemento argv[0] contiene el nombre del programa ejecutable.

Ejemplo:

```
/*
**      Programa 5_9
**
**      Este programa despliega los argumentos con los que se ejecuta el programa. Para
**      su ejecucion debera teclear a nivel de sistema operativo:
**
**      lab_5_8.exe  argumento1 argumento2 ... argumentoN
**
**      donde:          argumento1, ..., argumentoN  son cualquier palabra.
*/
#include <stdio.h>

main(int argc, char **argv)
{
        int i;

        for(i=0 ; i < argc; i++)
                printf("%s ", argv[i]);
        putchar('\n');
```

}

LABORATORIO

1. Modifique el programa 5_7, para que una vez que se han proporcionado y ordenado los elementos de la agenda, se hagan búsquedas sobre esta; es decir, se proporcionara un nombre y el programa determinara si se encuentra dado de alta o no.

ESTRUCTURAS
Y
UNIONES

typedef

C es un lenguaje que puede ampliarse con facilidad, por ejemplo, con la definición de símbolos mediante los #define y creando funciones de propósito general para uso de todos los usuarios.

También puede ampliarse el lenguaje al definir tipos de datos que se construyen con los tipos ya existentes.

También se pueden definir tipos mucho muy complejos con el uso de estructuras y apuntadores.

C proporciona diversos tipos fundamentales, como char, int, double, etc. y varios tipos derivados como arreglos y apuntadores; también proporciona la declaración **typedef,** que permite la definición de nuevos tipos a partir de los ya existentes o previamente definidos.

La forma de definir un nuevo tipo es como se muestra:

```
typedef     tipo_base   nuevo_tipo;
```

Ejemplos:

```
typedef     int         ENTERO;
typedef     char        CARACTER;
typedef     ENTERO      VECTOR[10];
typedef     CARACTER    *STRING;
typedef     VECTOR      MATRIZ[10];
```

Estos tipos definidos se pueden utilizar para la definición de variables, parametros de funciones, tipo del valor de retorno de las funciones, etc., de la misma forma en como se utilizan los tipos estándar:

```
STRING    lista[N];        /* Se define un arreglo de apuntadores a caracter */
MATRIZ    a,b,c;           /* Se definen tres arreglos bidimensionales de enteros */
```

Las definiciones de tipo permiten la documentación de programas. Normalmente las definiciones de tipo junto con los prototipos y las definiciones de símbolos con #define se colocan en archivos de encabezados (archivos con extensión .h).

Cuando los tipos de datos no tienen las mismas características en diferentes ambientes, el empleo de typedef permite que los programas sean transportables. Por ejemplo: el tipo int en los sistemas UNIX es de cuatro bytes y en otros es de dos, si la aplicación requiere del manejo de enteros en cuatro bytes se podría definir un tipo ENTERO (este se utilizaría para la definición de variables) y dependiendo del sistema, este tipo estaría asociado a un int o a un long. No se tendrían que modificar todas las definiciones de variables, solamente definir el tipo de sistema:

```
#ifdef      UNIX

typedef     int     ENTERO;

#endif

#ifdef      MS_DOS

typedef     long    ENTERO;

#endif
```

ESTRUCTURAS

Las estructuras permiten la representación de elementos cuyos componentes son de diferentes tipos.

Por ejemplo, suponga que se quiere representar mediante una estructura de datos la información de un empleado:

- Número de cuenta (ej. 1287BDG)
- Nombre
- Edad
- Dirección
- Teléfono
- Sexo (M, F)
- Tipo (V = vendedor, A = administrativo,
  T = técnico, D = directivo)
- Salario

La información del empleado contiene atributos de diferentes tipos, el nombre es un arreglo de caracteres, el sexo es un solo caracter, el salario es float, la edad es de tipo entero, etc.

La definición de la estructura para manejar la información de un empleado se muestra a continuación:

```
struct emp                                  /* Identificador para el nuevo tipo */
{
        char        noCta[8],               /* Se definen cada uno de los atributos */
                    nombre[35],
                    dirección[35],
                    teléfono[10],
                    sexo,
                    tipo;
        int         edad;
        float       salario;
};
```

Una vez definida la estructura, esta puede ser usada como tipo para la definición de variables:

```
struct emp  lista[N];
struct emp  empleado1;
```

Se puede hacer uso de typedef para hacer una definición de tipo más manejable:

```
typedef struct          /* El identificador de estructura (emp) en este caso es opcional */
{               /* ya que se utilizara el identificador de tipo EMPLEADO y no struct emp*/
        char        noCta[8],
                    nombre[35],
                    dirección[35],
                    teléfono[10],
                    sexo,
                    tipo;
        int         edad;
        float       salario;
} EMPLEADO;
```

Con la definición anterior, se podrían definir variables de la siguiente forma:

```
EMPLEADO  lista[N];
EMPLEADO  empleado1;
```

En la definición de el tipo EMPLEADO, la etiqueta emp después de struct es opcional, cuando esta se coloca, permite que se pueda utilizar también el identificador de tipo *struct emp* para la definición de variables. Es necesaria la etiqueta cuando la estructura tiene elementos del tipo que se esta definiendo.

Los nombres de los miembros de la estructura son únicos dentro de ella.

La forma de accesar los elementos de una estructura es por medio del operador "." (punto):

```
empleado1.edad = 27;
empleado1.sexo = 'F';
strcpy(empleado1.nombre,"Jessica Briseño");
```

Una expresión de la forma:

variable_estructura.atributo

se utiliza como cualquier expresión del tipo asociado al atributo.

Son válidas las expresiones de asignación de estructuras. En este caso, se copia el contenido de cada uno de los atributos, por ejemplo, si *empleado* y *empleado1* son estructuras del mismo tipo, la siguiente sentencia es válida:

```
empleado1 = empleado;
```

sin embargo, si el campo *nombre* en la estructura *struct emp* se hubiera definido como:

```
...
char   *nombre;
...
```

la asignación de un valor para el atributo *nombre* se tendría que realizar utilizando memoria dinámica:

```
empleado.nombre = (char *)malloc(strlen("Jessica Briseño")+1);
strcpy(empleado.nombre,"Jessica Briseño");
```

en este caso la asignación

```
empleado1 = empleado
```

provocaría que los atributos *nombre* de ambas estructuras apuntaran a la misma dirección de memoria, de tal forma que si se ejecuta la siguiente instrucción:

```
free(empleado.nombre)
```

el atributo *nombre* de la estructura *empleado1* direcciona memoria marcada como disponible, la cuál puede ser adquirida en cualquier momento dando como resultado comportamientos indeseables de las variables.

Ejemplo:

```
/*
**      Programa 6_1
**
**      Programa que implementa el manejo de números complejos con el uso de
**      estructuras.
**
*/

#include <stdio.h>

typedef struct
{
        float   real,
                imaginaria;
} COMPLEJO;

COMPLEJO  suma(COMPLEJO, COMPLEJO);
COMPLEJO  resta(COMPLEJO, COMPLEJO);

main()
{
        COMPLEJO  a,b,c;

        printf("\n\nProporciona dos numeros complejos (parte_real  parte_imaginaria):\n");
        printf("1=> ");
        scanf("%f",&a.real);
        scanf("%f",&a.imaginaria);
        printf("2=> ");
        scanf("%f",&b.real);
        scanf("%f",&b.imaginaria);
        c = suma(a,b);
        printf("\nLa suma de los numeros:\n%5.2f + %5.2fi\n%5.2f + %5.2fi\nes:\n\n"
                "%5.2f + %5.2fi\n", a.real, a.imaginaria, b.real, b.imaginaria,
                c.real, c.imaginaria);
        c = resta(a,b);
        printf("\n\ny la resta:\n\n%5.2f + %5.2fi\n", c.real, c.imaginaria);
}
```

```
/*
**      suma
**
**      Funcion que implementa la suma de numeros complejos.
**
**      PARAMETROS:
**              x               primer numero complejo
**              y               segundo numero complejo
**
**      RETURN:
**              suma de los dos numeros complejos
*/
COMPLEJO  suma(COMPLEJO  x, COMPLEJO y)
{
        COMPLEJO  z;

        z.real = x.real + y.real;
        z.imaginaria = x.imaginaria + y.imaginaria;
        return z;
}
/*
**      resta
**
**      Funcion que implementa la resta de numeros complejos.
**
**      PARAMETROS:
**              x               primer numero complejo
**              y               segundo numero complejo
**
**      RETURN:
**              resta de los dos numeros complejos
*/
COMPLEJO  resta(COMPLEJO  x, COMPLEJO y)
{
        COMPLEJO  z;

        z.real = x.real - y.real;
        z.imaginaria = x.imaginaria - y.imaginaria;
        return z;
}
```

Inicialización de estructuras

Las estructuras pueden ser inicializadas de una forma muy parecida a como se inicializan los arreglos:

```
EMPLEADO empleado = { "811CAFA",
                      "C. Jéssica Briseño C.",
                      "Norte 86B 4729",
                      "379-00-00",
                      'F',
                      'D',
                      24,
                      6500 };
```

Arreglos de estructuras

El lenguaje C permite la creación de arreglos de elementos cuyos tipos pueden ser cualquiera previamente definido, por ejemplo, se pueden definir arreglos de estructuras:

```
EMPLEADO        listaEmpleados[N];
```

Así, para poder accesar el salario del primer empleado en la lista se tendría que hacer referencia a listaEmpleados[0].salario

LABORATORIO

1. Implemente la agenda del capítulo anterior (programa 5_7 del capítulo "ARREGLOS Y APUNTADORES") con un arreglo de estructuras, de tal forma que se almacene el nombre , dirección y telefono de sus amigos.

2. Modifique el programa de agenda para que se puedan hacer consultas de telefonos de los amigos.

UNIONES

Una unión define a un conjunto de elementos de diferentes tipos que comparten el espacio de memoria asignado a la union.

Una unión al igual que una estructura, es un tipo derivado. La sintaxis de las uniones es semejante a la de las estructuras, solamente que en lugar de especificar el tipo *struct* se utiliza *union*.

Como los atributos de la union comparten la memoria asignada, el compilador asigna una porción de almacenamiento que pueda acomodar al más grande de los atributos especificados.

La notación para accesar a un atributo de una unión es idéntica a la que emplean las estructuras.

Como la union tiene asignado solo un espacio de memoria, el valor que contiene es interpretado dependiendo del atributo por el que se accese, elegir el correcto es responsabilidad del programador.

Ejemplo:

```
union numero {
        int             entero;
        double          real;
}

struct          numero          num;

num.real = 13.56;
printf("%f",num.real);          /* Al accesar la union con real obtendremos como valor 13.56 */
printf("%d",num.entero); /* Al accesar la union con enetro el valor obtenido es desconocido */
```

en este ejemplo, se crea una variable del tipo de la union definida. A la union se le asigna el espacio de memoria necesario para almacenar el atributo que ocupa mas espacio, en este caso *real*. El espacio de memoria de la union se puede manejar como de tipo *int* o como de tipo *double*, en el ejemplo, se accesa como *double*. Si hacemos referencia a ese espacio mediante el atributo *entero*, despues de la asignacion, el espacio de memoria se manejara como de tipo *int* y el valor obtenido es, en general, impredecible.

Ejemplo:

Suponga que se quiere representar la información de los empleados de una empresa mediante una estructura de datos. En la empresa existen diferentes tipos de empleados (vendedor, administrativo, directivo y técnico) y para el calculo de la nómina es importante considerar todos los puntos que influyen en el cálculo de las percepciones mensuales.

Los técnicos y directivos reciben un sueldo mensual, un bono adicional y un pago mensual para gastos de automovil; los vendedores perciben además de su sueldo base comisiones y premios, a los administrativos se les paga las horas extras trabajadas.

Para todos los empleados se tiene que almacenar su numero de cuenta, nombre, sexo, direccion, telefono, edad y año de ingreso.

Los tipos utilizados serían los siguientes:

```
/* Estructura para definir la informacion del sueldo de un empleado de confianza */

typedef  struct
{
        float   sueldoBase;
        float   bono;
        float   gastosAuto;
} CONFIANZA;


/* Estructura para definir la informacion del sueldo de un empleado administrativo */

typedef struct
{
        float   sueldoBase;
        int     horasExt;
} ADMON;


/* Estructura para definir la informacion del sueldo de un vendedor */

typedef struct
{
        float   sueldoBase;
        float   comision;
        float   premio;
} VENDEDOR ;


/* Union  para definir la informacion del sueldo de un empleado de la empresa */

typedef union
{
        CONFIANZA       confianza;
        ADMON           admon;
        VENDEDOR        vendedor;
} SALARIO;
```

La estructura EMPLEADO se definiría entonces, de la siguiente forma:

```
typedef struct
{
        char        noCta[8],
                    nombre[35],
                    dirección[35],
                    teléfono[10],
                    sexo,
                    tipo;
        int         edad,
                    anioIngreso;
        SALARIO     salario;
} EMPLEADO;
```

El atributo tipo es de gran importancia, ya que sirve para determinar como accesar la union:

```
EMPLEADO        emp;

        ...

if (emp.tipo = 'V')                                    /* Si se trata de un vendedor */
        sueldoMensual = emp.salario.vendedor.sueldoBase +
                        emp.salario.vendedor.comision + emp.salario.vendedor.premio;
        ...
```

LABORATORIO (OPCIONAL)

1. Modifique el programa agenda del laboratorio anterior, para que cuando se trate de registrar clientes se almacene ademas teléfono de oficina, fax y nombre de la empresa que representa; para los amigos hay que almacenar solamente el telefono de su casa y la fecha de cumpleaños.

Apuntadores a estructuras

Se pueden definir apuntadores a estructuras para poder referenciarlas indirectamente.

Desde un apuntador también se pueden accesar los atributos de una estructura.

Los apuntadores a estructuras se pueden utilizar para manejar paso de parámetros por referencia cuando se utilizan estructuras como parámetros.

Los apuntadores de estructuras son la base de la implementación más eficiente de estructuras de datos como pilas, listas lineales, gráficas y árboles.

La forma natural de accesar los atributos de una estructura por medio de un apuntador es un poco confusa:

```
EMPLEADO *p;
...

(*p).tipo = 'D';    /* Son necesarios los parentesis, por la presedencia de los operadores */
```

Una forma alterna, y la más utilizada es con el operador "->":

```
EMPLEADO *p;
...

p->tipo = 'D';
```

Ejemplo:

```
/*
**      lab6_2.c
**
**      Archivo main del programa de nomina.
*/
/*
**      Programa 6_2
**
**      Este programa genera un reporte de sueldos de una empresa. En la empresa
**      existen tres tipos de empleados: de confianza, administrativos y vendedores. La
**      forma de calcular el sueldo varia dependiendo del tipo de empleado.
**
*/
#include      <stdio.h>
#include      "emp.h"

main()
{
      EMPLEADO nomina[MAX_EMP];
      float          pagoEmp,
                     totalNomina=0;
      int            i, numEmp;

      printf("Cuantos empleados forman la nomina? ");
      scanf("%d",&numEmp);
      fflush(stdin);
      for(i=0 ; i < numEmp ; i++)
            leeDatos(&nomina[i]);
      printf("Reporte de NOMINA\n\n");
      printf("No. Cta.\tEmpleado\t\tPago\n\n");
      for(i=0 ; i < numEmp ; i++)
      {
            pagoEmp = calculaPago(nomina[i]);
            totalNomina += pagoEmp;
            printf("%-12.12s\t%-12.12s\t%10.2f\n",nomina[i].noCta,
                  nomina[i].nombre, pagoEmp);
      }
      printf("\n\n\t\tTOTAL: %12.2f\n",totalNomina);
```

```
}

/*
**      emp.c
**
**      Archivo de definicion de funciones para calculo de los pagos de empleados.
*/
#include      <stdio.h>
#include      "emp.h"
/*
**      leeDatos
**
**      Funcion que lee los datos de un empleado.
**
**      PARAMETROS:
**              empleado            empleado para el que se leera su informacion.
*/
void    leeDatos(EMPLEADO      *empleado)
{
        /*
        **      Datos comunes a todos los empleados:
        */
        printf("No. Cta   =>");
        gets(empleado->noCta);
        printf("Nombre    =>");
        gets(empleado->nombre);
        printf("Direccion  =>");
        gets(empleado->direccion);
        printf("Tel.      =>");
        gets(empleado->tel);
        printf("Sexo      =>");
        empleado->sexo = getchar();
        printf("Edad      =>");
        scanf("%d",&empleado->edad);
        printf("A&o ingreso=>");
        scanf("%d",&empleado->anioIngreso);
        fflush(stdin);
        printf("Tipo (C confianza, A administrativo, V vendedor) =>");
        empleado->tipo = getchar();
        fflush(stdin);
```

```
        /*
        **      Datos particulares dependiendo del tipo de empleado.
        */
        if(empleado->tipo == 'C' || empleado->tipo == 'c')
        {
                printf("Sueldo    =>");
                scanf("%f",&empleado->salario.confianza.sueldoBase);
                printf("Bono      =>");
                scanf("%f",&empleado->salario.confianza.bono);
                empleado->salario.confianza.gastosAuto = GASTOS_AUTO;
                fflush(stdin);
        }
        else if (empleado->tipo == 'A' || empleado->tipo == 'a')
        {
                printf("Sueldo    =>");
                scanf("%f",&empleado->salario.admon.sueldoBase);
                printf("Horas Ext. =>");
                scanf("%d",&empleado->salario.admon.horasExt);
                fflush(stdin);
        }
        else
        {       printf("Sueldo    =>");
                scanf("%f",&empleado->salario.vendedor.sueldoBase);
                printf("Comisiones =>");
                scanf("%d",&empleado->salario.vendedor.comision);
                printf("Premios   =>");
                scanf("%d",&empleado->salario.vendedor.premio);
                fflush(stdin);
        }
}
```

```
/*
**      calculaPago
**
**      Funcion encargada de calcular el sueldo de un empleado en base a sus datos de
**      nomina. A los empleados de confianza se les descuenta el 15% por concepto
**      de impuesto sobre la suma que resulte de su sueldo base mas su ayuda para
**       gastos de auto, su bono no tiene ningun descuento. A los administrativos se les
**      descuenta solamente el 8% sobre su sueldo base y cada hora extra trabajada se
**      paga de acuerdo a una tarifa establecida con el sindicato. A los vendedores se les
**      paga sueldo base, comisiones y premios, pero todos estos conceptos causan 10%
**      de impuesto retenido.
**
**      PARAMETROS
**              empleado            informacion del empleado
**
**      RETURN:
**              pago mensual del empleado
*/
float   calculaPago(EMPLEADO  empleado)
{
        float   pago;

        if(empleado.tipo == 'C' || empleado.tipo == 'c')
                pago = (empleado.salario.confianza.sueldoBase +
                        empleado.salario.confianza.gastosAuto) * 0.85 +
                        empleado.salario.confianza.bono;

        else if(empleado.tipo == 'A' || empleado.tipo == 'a')
                pago = (empleado.salario.admon.sueldoBase +
                    empleado.salario.admon.horasExt * PAGO_HORA_EXT) * 0.92;

        else
                pago = (empleado.salario.vendedor.sueldoBase +
                        empleado.salario.vendedor.comision +
                        empleado.salario.vendedor.premio) * 0.9;
        return pago;
}
```

```
/*
**      emp.h
**
**      Archivo de definicion de tipos, prototipos y simbolos para las funciones de calculo
**      de la nomina de empleados.
*/

#ifndef     EMP_H
#define     EMP_H

/*
**      Estructura que define los datos particulares de un empleado de confianza.
*/
typedef  struct
{
        float    sueldoBase;
        float    bono;
        float    gastosAuto;
}  CONFIANZA;

/*
**      Estructura que define los datos particulares de un empleado administrativo.
*/
typedef struct
{
        float    sueldoBase;
        int  horasExt;
}  ADMON;

/*
**      Estructura que define los datos particulares de un vendedor.
*/
typedef struct
{
        float    sueldoBase;
        float    comision;
        float    premio;
} VENDEDOR ;
```

```
/*
**      Union que define la informacion del salario de un empleado, puede ser
**      informacion de un vendedor, de un administrativo o de un empleado de
**      confianza.
*/
typedef union
{
        CONFIANZA     confianza;
        ADMON         admon;
        VENDEDOR      vendedor;
} SALARIO;

/*
**      Estructura para definir la informacion de un empleado.
*/
typedef struct
{
    char        noCta[8],
                nombre[35],
                direccion[35],
                tel[10],
                sexo,
                tipo;
    int         edad,
                anioIngreso;
    SALARIO         salario;
} EMPLEADO;

#define     MAX_EMP         20              /* Numero maximo de empleados */
#define     GASTOS_AUTO     550             /* Ayuda para gastos de automovil */
#define     PAGO_HORA_EXT   50              /* Pago por hora extra para los empleados administrativos */

/*
**      Prototipos
*/
void    leeDatos(EMPLEADO *);
float   calculaPago(EMPLEADO);
void    leeFloat(int *);

#endif
```

RESUMEN DE OPERADORES

Operador	Asociatividad
() [] ->	izquierda a derecha
! ~ ++ -- + - * & sizeof	derecha a izquierda
* / %	izquierda a derecha
+ -	izquierda a derecha
>> <<	izquierda a derecha
> >= < <=	izquierda a derecha
== !=	izquierda a derecha
&	izquierda a derecha
^	izquierda a derecha
|	izquierda a derecha
&&	izquierda a derecha
||	izquierda a derecha
?:	derecha a izquierda
= += -= &= *= /= %= ^=	derecha a izquierda
|= <<= >>=	
, (operador coma)	izquierda a derecha

ARCHIVOS

Manejo de archivos

Las funciones estandar que proporciona el lenguaje C permiten el acceso de archivos en forma secuencial y directa (a alguna posición en particular en el archivo). Si se quiere manejar otro tipo más complicado de acceso como lo sería algun indexado, se tienen que implementar funciones con los algoritmos apropiados.

Por otra parte, las funciones estandar de C para manejo de archivos no utilizan el concepto de lectura/escritura de registros, un archivo es simplemente un flujo de bytes de entrada o salida.

El sistema operativo proporciona tres archivos estándar, definidos como:

- stdin: archivo estándar de entrada (teclado)
- stdout: archivo estándar de salida (pantalla)
- stderr: archivo estándar de errores (pantalla)

Estos archivos pueden ser utilizados como cualquier otro.

printf

La función *printf* es utilizada para escribir a la salida estandar de datos. Esta función permite dar formato a los datos que se generan a la salida.

Esta función recibe como parámetros una cadena de formato y una lista variable de parámetros:

Sintaxis:

printf(cadena, parametro1, parametro2, ..., parametroN)

El primer parámetro, la cadena de formato, contiene caracteres ordinários, que son escritos a la salida, y especificaciones de conversión, cada una de las cuales causa la conversión de los siguientes argumentos sucesivos de printf.

Cada una de las especificaciones de conversión comienzan con % y terminan con uno de los caracteres mostrados en la siguiente tabla.

CARACTER	FORMA EN LA QUE ES IMPRESO EL ARGUMENTO
d	Número decimal.
o	Número en formato octal.
x	Número en formato hexadecimal.
c	Caracter.
s	Cadena de caracteres.
f	Número con parte fraccionaria sin exponente.
e	Número de punto flotante con exponente.
u	Entero decimal sin signo.

Entre el % y el caracter de conversión puede aparecer, en orden:

- Un signo menos, que indica justificación a la izquierda del argumento convertido.

- Un número que indica el ancho mínimo del campo.

- Un punto, que separa el ancho de campo de la precisión.

- Un número que indica el número de dígitos después del punto decimal para un valor numérico, o el número máximo de caracteres de una cadena.

Ejemplos:

especificación	**significado**
%10s	Se escribe la cadena en campo de 10 caracteres esta especificación no se respeta si la cadena tiene una longitud mayor. La justificación es a la derecha.
%-10.10s	Se escribe la cedena en un campo de 10 caracteres, si la cadena es mayor a 10 caracteres, a la salida solo se escriben los 10 primeros. La justificación es a la izquierda.
%8d	El valor numérico se escribe en un campo de 8 caracteres.
%10.2f	El valor se convierte a float y se escribe en un campo de 10 caracteres (incluyendo parte entera, fraccionaria y punto) con dos digitos a la derecha del punto.

scanf

La función scanf permite la lectura de datos de la entrada estandar.

Esta función recibe como parámetros una cadena de formato y una lista variable de parámetros.

La cadena de formato contiene caracteres ordinarios, que son leídos de la entrada. Los caracteres de entrada se convierten en valores de acuerdo con las especificaciones de conversión.

La lista de parámetros consiste en una lista de variables apuntadores.

Cada una de las especificaciones de formato comienzan con % y terminan con uno de los caracteres mostrados en la siguiente tabla.

CARACTER	FORMA EN LA QUE ES LEIDA LA ENTRADA
d	Número decimal.
o	Número en formato octal.
x	Número en formato hexadecimal.
c	Caracter.
s	Cadena de caracteres.
f	Número con parte fraccionaria.
e	Equivalente a f.
u	Entero decimal sin signo.

Escritura/lectura de archivos

Las funciones estandar para manejo de archivos del lenguaje C, permiten el acceso a archivos através de una estructura definida como FILE (la definición de esta estructura se encuentra en el archivo estándar de encabezados *stdio.h*).

La estructura FILE contiene campos o atributos que describen el estado actual del archivo, por ejmplo la fecha de creación, el tamaño del archivo, la posición física de inicio, los permisos de acceso al archivo, etc.

Para abrir un archivo se utiliza la función *fopen:*

FILE *fopen(char *fileName, char *modo)

El primer parámetro es el nombre del archivo. Si no se especifica toda la ruta del archivo, este se busca en el directorio de trabajo actual. En MS-DOS, hay que considerar que las rutas se indican como: C:\cursos\c\labs\lab7_1.c y que el caracter '\' es el caracter de escape, por lo tanto la forma correcta de indicar el nombre del archivo es:

C:\\cursos\\c\\labs\\lab7_1.c

El segundo parámetro es el modo que puede ser alguno de los siguientes:

r	lectura
w	escritura (crea el archivo)
a	agregar

Recuerde que los dos parametros que recibe la función *fopen* son de tipo *char* *, por lo que si se quiere indicar constantes se deben delimitar por comillas:

fopen("C:\\cursos\\c\\labs\\datos", "r")

El valor de regreso de la función es una apuntador a FILE, mediante el cual se hacen las referencias posteriores al archivo. Si el valor de regreso es NULL se debe a que existe algún error, algunas de las posibles causas son:

- No se tiene permisos de escritura en el archivo.
- Se trata de leer un archivo que no existe.
- Se trata de crear un archivo en un directorio protegido.
- No se tiene permisos de lectura.

Los archivos deben ser cerrados cuando ya no son utilizados, para ello se utiliza la función **fclose**:

```
fclose(FILE *fp)
```

El parámetro que recibe la función es el apuntador a FILE del archivo que se desea cerrar.

Las funciones fprintf y fscanf se utilizan para escribir y leer de un archivo respectivamente, en la misma forma que se utilizan las funciones printf y scanf para la salida y entrada estandar respectivamente.

La sintaxis para utilizarlas es la siguiente:

```
fprintf(fp, s, arg1, arg2,..., argN)
fscanf(fp, s, arg1, arg2,..., argN)
```

donde:

fp = apuntador a FILE del archivo previamente abierto

s = cadena de formato

arg1,.. , argN = lista variable de parámetros

Otras funciones de entrada/salida

int fgetc(FILE *fp)

Lee un caracter del archivo, regresa EOF cuando es fin de archivo.

int getc(FILE *fp)

Es equivalente a fgetc pero es una macro.

int getchar(void)

Es una macro, construida como getc(stdin).

int fputc(int c, FILE *fp)

Escribe el carácter c en el archivo. Regresa el carácter escrito o EOF en caso de error.

int putc(int c, FILE *fp)

Es equivalente a fputc pero es una macro.

int putchar(int c)

Es una macro, construida como putc(s, stdout);

Ejemplo:

```
/*
**      Programa 7_1
**
**      Programa que demuestra el manejo de archivos con funciones de lectura de
**      caracteres. Se lleva a cabo la copia de dos achivos, el nombre de los archivos se
**       indica en la linea de comandos.
**      La forma de ejecutar el programa es la siguiente:
**
**      lab7_1  archivo_origen   archivo_copia
**
*/
#include <stdio.h>

main(int argc, char **argv)
{
        FILE            *fileRead,
                        *fileWrite;
        int             c;

        if (argc !=3)
        {
                fprintf(stderr,"Uso: %s file1 file2\n", argv[0]);
                exit(1);
        }
        if ((fileRead = fopen(argv[1],"r")) == NULL)
        {
                fprintf(stderr,"%s: error al abrir el archivo %s\n",
                        argv[0], argv[1]);
                exit(1);
        }
        if ((fileWrite = fopen(argv[2],"w")) == NULL) {
                fprintf(stderr,"%s: error al abrir el archivo %s\n",
                        argv[0], argv[2]);
                exit(1);
        }
        while((c = getc(fileRead)) != EOF )
                putc(c, fileWrite);
        fclose(fileRead);
        fclose(fileWrite);
```

}

Funciones de entrada/salida con cadenas

char *fgets(char *s, int n, FILE *fp)

Se leen los n-1 caracteres del archivo apuntado por fp o hasta que exista un carácter nuevalínea '\n', lo que suceda primero. Los caracteres leídos se colocan en s, lo mismo sucede con '\n', si este es leído. La cadena s se termina con '\0'. El valor de retorno es s, o NULL si existe fin de archivo u ocurre un error.

int fputs(char *s, FILE *fp)

Escribe la cadena s en el archivo. Regresa un valor no negativo o EOF en caso de error.

char *gets(char *s)

Lee la siguiente línea de la entrada estándar y la coloca en s. Reemplaza el carácter '\n' por '\0'. Regresa s, o NULL en caso de que se lea fin de archivo.

int puts(char *s)

Escribe la cadena s en la salida estándar además, agrega un carácter '\n'. Regresa EOF en caso de error.

Ejemplo:

```
/*
**      Programa 7_2
**
**      Programa que demuestra el manejo de archivos con funciones de lectura de
**      cadenas. Se lleva a cabo la copia de dos achivos, el nombre de los archivos se
**       indica en la linea de comandos.
**      La forma de ejecutar el programa es la siguiente:
**
**      lab7_2  archivo_origen   archivo_copia
**
*/
#include      <stdio.h>
#define       LONG_REG 80

main(int argc, char **argv)
{
        FILE            *fileRead,
                        *fileWrite;
        char            reg[LONG_REG];

        if (argc !=3) {
                fprintf(stderr,"Uso: %s file1 file2\n", argv[0]);
                exit(1);
        }
        if ((fileRead = fopen(argv[1],"r")) == NULL) {
                fprintf(stderr,"%s: error al abrir el archivo %s\n",
                        argv[0], argv[1]);
                exit(1);
        }
        if ((fileWrite = fopen(argv[2],"w")) == NULL) {
                fprintf(stderr,"%s: error al abrir el archivo %s\n",
                        argv[0], argv[2]);
                exit(1);
        }
        while(fgets(reg, LONG_REG, fileRead) != NULL )
                fputs(reg, fileWrite);
        fclose(fileRead);
        fclose(fileWrite);
}
```

LABORATORIO

1. Modifique el programa agenda del capítulo anterior para que los datos de entrada los obtenga de un archivo.

MATERIAL

COMPLEMENTARIO

13 DE MARZO DE 1998

Introducción a la Programación en Lenguaje C

J. Antonio Chávez F.

Mayo de 1995

CONTENIDO

4. Funciones y el Preprocesador

- 4.1 Funciones
  - 4.1.1 Declaración y definicón deuna función
  - 4.1.2 Valores de regreso
  - 4.1.3 Paso de parámetros
  - 4.1.4 Variables automáticas
  - 4.1.5 Variables externas
  - 4.1.6 Variables estáticas
  - 4.1.7 Inicialización de las variables
  - 4.1.8 Reglas de alcance
  - 4.1.9 Recursividad
  - 4.1.10 Laboratorio
- 4.2 El preprocesador
  - 4.2.1 **#include**
  - 4.2.2 **#define**
  - 4.2.3 Compilación condicional
  - 4.2.4 División de un programa en varios archivos
  - 4.2.5 Laboratorio

5. Arreglos y apuntadores

- 5.1 Arreglos
  - 5.1.1 Inicialización de arreglos
  - 5.1.2 Arreglos de caracteres
  - 5.1.3 Arreglos multudimensionales
  - 5.1.4 Laboratorio
- 5.2 Apuntadores
- 5.3 Apuntadores como parámetros defunciones
- 5.4 Apuntadores y arreglos
- 5.5 Arreglos como parámetros de funciones
- 5.6 Aritmética de apuntadores
- 5.7 Funciones para manejo de caracteres y cadenas
- 5.8 Laboratorio
- 5.9 Arreglos de apuntadores
- 5.10 Parámetros para main
- 5.11 Laboratorio

6. Estructuras y Uniones

6.1 **typedef**
6.2 Estructuras
6.3 Inicialización de estructuras
6.4 Arreglos de estructuras
6.5 Laboratorio
6.6 Uniones
6.7 Apuntadores a estructuras
6.8 Resumen de operadores

7. Manejo de archivos

7.1 Manejo de archivos
7.2 **printf**
7.3 **scanf**
7.4 Escritura y lectura de archivos
7.5 Funciones de entrada y salida
7.6 Laboratorio

BIBLIOGRAFIA

Brian W. Kernighan, Dennis M. Ritchie
El lenguaje de preogramación C.
Segunda Edición
Prentice Hall

Al Kelley, Ira Pohl
Introducción al lenguaje C
Addison Wesley

Aaron M. Tenenbaum, Yedidyah Langsam
Estructuras de datos en C
Prentice Hall

Herbert Schildt
C, Guía para usuarios expertos
McGraw Hill

INTRODUCCION

Historia

Su origen esta en los lenguajes BCPL (Martin Richards) y B (Ken Thompson).

Fué diseñado por Dennis Ritchie en la Laboratorios Bell de AT&T en 1972.

El sistema operativo UNIX fué originalmente escrito en C por el mismo grupo de investigadores de AT&T.

Su definición formal aparece en 1978 en el apéndice "C Reference Manual" del libro "The C Programming Language" de Brian W. Kernighan y Dennis M. Ritchie.

En 1983 el Instituto Nacional Americano de Estandares (ANSI) establece un comité para proporcionar una definición estandar denominada, el estándar ANSI o "ANSI C".

Características de C

- Es un lenguaje de propósito general.
- Es un lenguaje pequeño en cuento al número de instrucciones que lo componen.
- Es muy poderoso, debido a sus capacidades de lenguaje de bajo nivel.
- Es fácil de aprender.
- Existe una estrecha relación con UNIX.
- Es portátil.
- Es elegante.

Compilación de un programa

La edición del programa fuente se puede hacer desde cualquier editor del sistema operativo.

El nombre de un archivo fuente en C debe terminar con ".c" (en UNIX no existe el concepto de extensión, sin embargo es necesario que el nombre de los archivos terminen con ".c").

Un programa en C esta compuesto de uno o más archivos fuente (solamente debe de existir una función main).

Cada archivo fuente puede ser compilado independientemente.

El ligado del código objeto de los archivos que conforman el programa con las librerias necesarias dan como resultado el código ejecutable del programa.

Unix

Edición:

```
% vi agenda.c
```

Compilación exclusivamente:

```
% cc -c agenda.c
% cc -c listas.c
% cc -c utilsStrings.c
```

Compilación y ligado:

```
% cc agenda.c listas.c utilsStrings.c
% cc agenda.c listas.c utilsStrings.c -o agenda
```

Ligado:

```
% cc agenda.o listas.o utilsStrings.o -o agenda
```

Ejecución:

```
% a.out
% archivo
```

El programa en C más famoso

```
/*
**      Programa 1_1
**
**      Este es el primer ejemplo de un programa en C
*/

main()                                                  /* Funcion principal */
{                                                       /* Inicio de la funcion */
        printf("Hola Mundo\n");         /* Imprime una cadena finalizada con salto de linea */
}                                                       /* Fin de la funcion */
```

Ejemplo No. 2

```
/*
**      Programa 1_2
**
**      Obtiene el mayor de 2 números
**
*/

#include <stdio.h>                          /* Sustituye el contenido del archivo stdio.h*/
                    /* El archivo stdio.h contiene definiciones y definiciones para las funciones de I/O */

main()
{
    int n1, n2;                             /* Se definen dos variables de tipo int */

    printf("Proporciona 2 números: ");  /* Se imprime la cadena a la salida estandar de datos */
    scanf("%d",&n1);                    /* Se lee el valor para la variable n1 de la entrada estandar */
    scanf("%d",&n2);
    if (n1 > n2)                            /* Si el valor de n1 es mayor al de n2 */
        printf("El número mayor es: %d\n", n1);
    else                                    /* Si el valor de n1 no es mayor al de n2 */
        printf("El número mayor es: %d\n", n2);
}
```

Ejemplo No. 3

```
/*
**      Programa 1_3
**
**      Obtiene el mayor y menor de N números.
**
*/

#include <stdio.h>

#define     N     5     /* Define el simbolo N, a partir de esta definición, dicho simbolo sera
reemplazado por 5 */

main()
{
        int     max, /* La definición de variables del mismo tipo puede ser en una lista o por separado */
                min,
                i;
        int     numero;

        printf("Dame el primer numero: ");
        scanf("%d",&numero);
        max = numero;
        min = numero;
        i = 2;                                          /* Asignacion a la variable i */
        while (i <= N)                                  /* Mientras el valor de i sea menor al de N */
        {                                               /* Inicio del bloque de instrucciones a realizar en el ciclo */
                printf("Dame el numero %d: ",i);        /* Manejo de formatos en la funcion printf: el
                                                           formato %d es sustituido por el valor de i */
                scanf("%d",&numero);
                if (numero > max)
                        max = numero;
                if (numero < min)
                        min = numero;
                i = i + 1;
        }                                               /* Fin del bloque de instrucciones del ciclo */
        printf("\n\tEl número mayor es: %d\n\ty el menor: %d", max,min);
}
```

Ejemplo No. 4

```
/*
**      Programa 1_4
**
**      Cuenta los caracteres de la entrada de datos (entrada estandar).
**
*/

#include <stdio.h>

main()
{
        int     n=0;                                    /* Inicialización de variable */

                                                /* La funcion getchar lee un caracter de la entrada estandar */
        while(getchar() != EOF)
                n++;                                    /* Se incrementa en una unidad el valor de n */
        printf("\n\tTecleaste %d caracteres\n",n);
}
```

Ejemplo No. 5

```
/*
**      Programa 1_5
**
**      Cuenta los caracteres de la entrada. Versión utilizando una construccion for.
**
*/

#include <stdio.h>

main()
{
        int     n;

        for(n=0 ; getchar() != EOF; n++)
                ;                                               /* Sentencia vacia */
        printf("\n\tTecleaste %d caracteres\n",n);
}
```

Ejemplo No. 6

```
/*
**      Programa 1_6
**
**      Cuenta lineas, palabras y caracteres de la entrada.
*/
#include <stdio.h>

main()
{
        int     c,nc, np, nl;

        nc = np = nl = 0;       /* El operador de asignación se puede asociar de derecha a izquierda */
        while((c = getchar()) != EOF)       /* Se asigna a c el valor regresado por getchar y se
                                               compara con EOF */
        {
                nc++;
                if (c == '\n')                          /* El operador de igualdad es == */
                        nl++;

                /*
                ** Si el caracter leido es una letra, es el comienzo de una palabra.
                */
                if (c >= 'a' && c<= 'z'  || c>= 'A' && c<= 'Z')
                {
                        np++;

                        /*
                        ** Se leen todos los caracteres de la palabra. Como se lee el
                        ** siguiente despues de la palabra, este se regresa a la entrada.
                        */
                        while ((c=getchar()) >= 'a' && c <= 'z' || c>= 'A' && c<= 'Z')
                                nc++;
                        ungetc(c,stdin);/* La funcion ungetc regresa un caracter a la entrada estandar */
                }
        }
        printf("\n\nTotales %d %d %d\n",nl,np,nc);
}
```

LABORATORIO

1. Escriba un programa que imprima su nombre, dirección, telefono y edad en tres líneas separadas (Haciendo uso de la función printf).

2. Escriba un programa que presente una tabla de la suma progresiva de los primeros N números. La salida deberá ser como la siguiente:

Numero	Suma progresiva
1	1
2	3
3	6
...	...

3. Modifique el programa 1_6 para que se cuenten cualquier tipo de palabras (palabras que incluyan números y subguión _).

4. Haga un programa que convierta los caracteres proporcionados como entrada de datos a mayúsculas.

Página intencionalmente blanca.

CONTROL DE FLUJO

Expresiones y sentencias

Una expresión puede ser:

- una variable o constante:

  135.4 a 'A' "hola"

- una llamada a una función:

  strcat(cad1,cad2) sin(1)

- una asignación:

  a = 5

- una combinación de operandos y operadores:

  numero >= limInf && numero <=limSup

Las expresiones en el lenguaje, son generalmente utilizadas en las llamadas a funciones, asignaciones o en la formación de nuevas expresiones.

Todas las expresiones al evaluarlas regresan un valor y, como consecuencia, tienen asociado un tipo, el cual depende totalmente de los operandos y operadores involucrados:

- Una variable o constante, como expresión, tienen asociado el tipo y el valor que como variable o constante representan en un momento dado.

- Una llamada a una función, como expresión, tiene asociado el tipo y el valor de retorno de la función.

- Una asignación, como expresión, tiene asociado el tipo y el valor asignado a la variable.

- Una expresión aritmética, tiene asociado el tipo y el valor que resulte de la evaluación de los operadores.

- Una expresión lógica, siempre tiene asociado un tipo entero y el valor puede ser verdadero (diferente de cero) o falso(cero).

Una sentencia es una expresión terminada con ";" , ejemplos:

```
a = 5;
sin(a);
a++;
```

Un bloque es una colección de sentencias agrupadas por "{" y "}" que se les considera como una sola sentencia, ejemplo:

```
{
        a = 5;
        sin(a);
        a++;
}
```

La sentencia vacía se representa como:

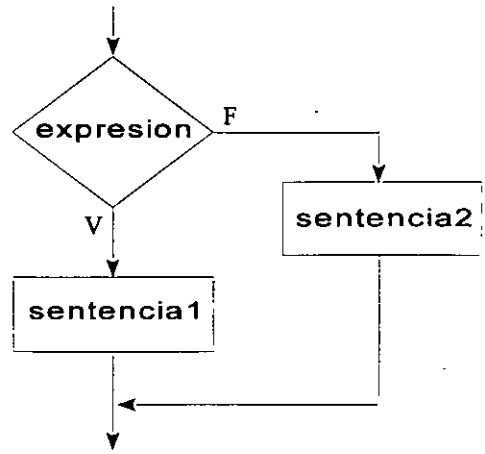
```
;
```

if

Un if es **una sentencia condicional**.

Su sintaxis es la siguiente:

```
if (expresión)
        sentencia1
else
        sentencia2
```



La parte ***else*** es opcional.

En construcciones anidadas, la parte ***else*** termina el if más interno, el compilador no toma en cuenta el sangrado:

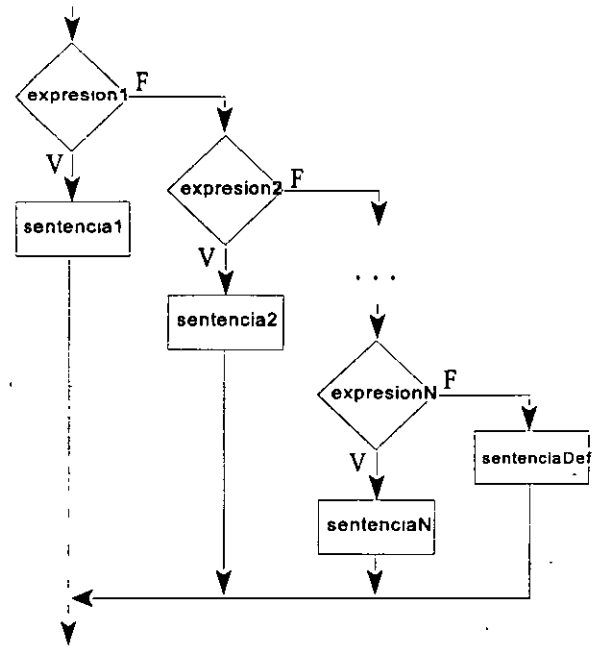
```
...

if (n>b)                                                    /* Primer IF*/
        if (n > c)                                          /* Segundo IF*/
                z = c;
else                                    /* La condicion else pertenece al segundo IF*/
        z = 0;
...
```

Existen errores comunes como el siguiente:

```
...

/*
** En este caso, la expresión es una asignación que siempre es verdadera
**  (su valor es 5).
 */
if ( x=5)
        printf("valor correcto\n");
else
        printf("valor incorrecto");
...
```

Una decisión múltiple puede implementarse con una serie de if anidados; sin embargo, el sangrar cada una de las sentencias provocaría que el tamaño de la línea creciera demasiado, para ello se emplea una construcción como la siguiente:

```
if (expresión1)
        sentencia1
else if (expresión2)
        sentencia2
else if (expresión3)
        sentencia3
. . .

else if (expresiónN)
        sentenciaN
else
        sentenciaDef
```



En la construcción anterior, las expresiones se evalúan en orden, cuando alguna de ellas es verdadera, la sentencia asociada se ejecuta y con esto se termina la construcción.

La sentencia del último else (sentenciaDef) se ejecuta cuando ninguna expresión es verdadera.

Ejemplo:

```
/*
**      Programa 3_1
**
**      Este programa determina el signo zodiacal de una persona en base al
**      día y mes de nacimiento.
**
*/

#include <stdio.h>

/*
** calculaDiaDelAnio
**
** Obtiene el dia del anio de una fecha determinada por el mes y dia del mes.
**
** PARAMETROS
**      mes
**      dia
**
** RETURN
**      dia del a&o
*/
int     calculaDiaDelAnio(int mes, int dia)
{
        int     diasPorMes[12] =    {31, 29, 31, 30, 31, 30,
                                     31, 31, 30, 31, 30, 31};
        int     diasTanscurridos;
        int     i;

        diasTranscurridos = 0;
        for (i=0 ; i < mes - 1 ; i++)
                diasTranscurridos += diasPorMes[i];

        return diasTranscurridos + dia;
}
```

```
main()
{
    int     mes, dia, diaDelAnio;
    enum  meses{ enero=1, febrero, marzo, abril, mayo, junio, julio, agosto, septiembre,
                octubre, noviembre, diciembre };

    printf("Mes de nacimiento [1-12]: ");
    scanf("%d",&mes);
    printf("Dia [1-31]: ");
    scanf("%d",&dia);
    diaDelAnio = calculaDiaDelAnio(mes, dia);

    if (diaDelAnio <= calculaDiaDelAnio(enero,21))
        printf ("Eres CAPRICORNIO***\n");
    else if (diaDelAnio <= calculaDiaDelAnio(febrero,21))
        printf ("Eres ACUARIO\n");
    else if (diaDelAnio <= calculaDiaDelAnio(marzo,21))
        printf ("Eres PISCIS***\n");
    else if (diaDelAnio <= calculaDiaDelAnio(abril,21))
        printf ("Eres ARIES\n");
    else if (diaDelAnio <= calculaDiaDelAnio(mayo,21))
        printf ("Eres TAURO\n");
    else if (diaDelAnio <= calculaDiaDelAnio(junio,21))
        printf ("Eres GEMINIS***\n");
    else if (diaDelAnio <= calculaDiaDelAnio(julio,21))
        printf ("Eres CANCER\n");
    else if (diaDelAnio <= calculaDiaDelAnio(agosto,21))
        printf ("Eres LEO\n");
    else if (diaDelAnio <= calculaDiaDelAnio(septiembre,21))
        printf ("Eres VIRGO***\n");
    else if (diaDelAnio <= calculaDiaDelAnio(octubre,21))
        printf ("Eres LIBRA***\n");
    else if (diaDelAnio <= calculaDiaDelAnio(noviembre,21))
        printf ("Eres ESCORPION\n");
    else if (diaDelAnio <= calculaDiaDelAnio(diciembre,21))
        printf ("Eres SAGITARIO\n");
    else
        printf ("Eres CAPRICORNIO\n");
}
```

Operador Condicional

El operador condicional permite la implementación de una expresión condicional en una sola línea.

Su sintaxis es la siguiente:

expresión1 ? expresión2 : expresión3

En una expresión condicional:

- Primero se evalúa la expresión1.
- Si la expresión1 es verdadera, se evalúa la expresión2.
- Si la expresión1 es falsa, se evalúa la expresión3.

El resultado y tipo de la expresión condicional es el resultado y tipo de la expresión que se evalúa al último (expresión2 o expresión3).

Ejemplo:

```
/*
**      Programa 3_2
**
**      Programa que imprime el mayor de dos números
**
*/

#include <stdio.h>

main()
{
        int     x, y;

        printf("Proporciona dos numeros: \n");
        printf("1 => ");
        scanf("%d",&x);
        printf("2 => ");
        scanf("%d",&y);
        printf("%d es el número mayor entre %d y %d\n",
                ((x >y) ? x : y), x, y);
}
```

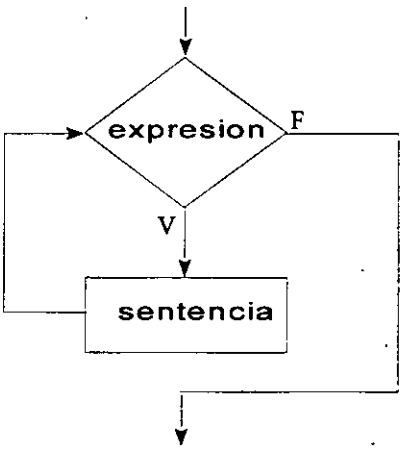
Los operadores condicionales se pueden anidar, ejemplo:

```
/*
**      Programa 3_3
**
**      Programa que imprime el mayor de tres números
**
*/

#include <stdio.h>

main()
{
        int     x, y, z;

         printf("Proprociona 3 numeros:\n");
        printf("1=> ");
        scanf("%d",&x);
        printf("2=> ");
        scanf("%d",&y);
        printf("3=> ");
         scanf("%d",&z);
        printf("%d es el numero mayor entre %d, %d y %d\n",
                ((x >y) ? ((x>z) ? x : z) : ((y>z) ? y : z) ),
                x, y,z);
}
```

while

Sintaxis:

while (expresión)
- sentencia



La sentencia es ejecutada mientras la expresión sea verdadera.

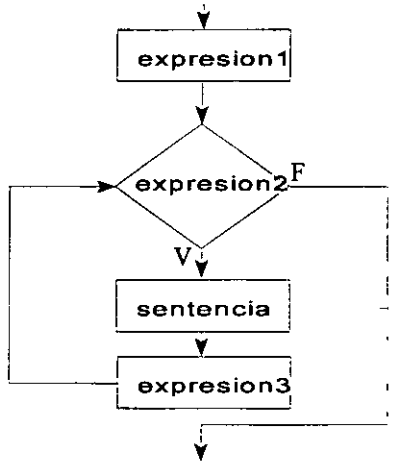
Ejemplo:

```
/*
**      Programa 3_4
**
**      Este programa genera una letra mayuscula del alfabeto aleatoria (tomando
**      como base la funcion rand) para que sea adivinada. Esta tarea se realiza
**      en un ciclo que termina cuando el usuario ya no desee continuar.
**      Version utilizando while.
*/
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>

main()
{
        char    respuesta='s';
        char    car,                                            /* Caracter generado*/
                apuesta;                                /* Caracter proporcionado por el usuario*/

        while ( respuesta == 's' || respuesta == 'S')
        {
                randomize();                    /* Se inicializa la semilla para numeros aleatorios */
                car = 'A' + rand() % 26;        /* Con esta formula se obtiene alguna de las 26 letras */

                printf("Adivina el caracter: ");
                apuesta = getchar();
                getchar();                      /* Esta lectura sirve para vaciar el buffer de la entrada, */
                                                        /* la cual  contiene el caracter nueva linea */
                if (islower(apuesta))                   /* Si el caracter proporcionado es minuscula */
                        apuesta -= 32;                          /* se convierte a mayuscula */
                if (car == apuesta)
                        printf("FELICIDADES ADIVINASTE EL CARACTER!!!\n");
                else
                        printf("Perdiste ..., el caracter es=> %c\n",car);
                printf("\nContinuas [s/n]? ");
                respuesta = getchar();
                getchar();
        }
}
```

for

Sintaxis:

for(expresión1;expresión2;expresión3)
sentencia



La secuencia de ejecución es la siguiente:

1. Se ejecuta la expresión1.
2. Se evalúa la expresión2:

   si la evaluación es falsa, termina el for.
   si la evaluación es verdadera, se continua en el paso 3.

3. Se ejecuta la sentencia.
4. Se evalúa la expresión3.
5. Se regresa al paso 2.

Cualquiera de las expresiones se puede omitir.

Si se omite la segunda expresión, se trata de un ciclo infinito.

Ejemplo:

```
/*
**      Programa 3_5
**
**      Este programa calcula el numero de dias transcurridos a partir de una fecha,
**      por ejemplo desde la fecha de nacimiento.
**
*/

#include <stdio.h>

/*
** diasDelAnio
**
** Obtiene el numero de dias de un anio.
**
*/
int     diasDelAnio(int anio)
{
        /*
        ** Si el anio es biciesto, el numero de dias es 366.
        */
        return (anio % 4 ? 365 : 366);
}
```

```
/*
** diaDelAnio
**
** Obtiene el dia del anio de una fecha determinada por el anio, mes y dia del mes.
**
*/
**
int     diaDelAnio(int anio, int mes, int dia)
{
        int     diasPorMes[12] =  {31, 28, 31, 30, 31, 30,
                                   31, 31, 30, 31, 30, 31};
        int     i,diasTranscurridos;

        diasTranscurridos = 0;

        for ( i  = 0 ; i < mes -1 ; i++)
                diasTranscurridos += diasPorMes[i];

        diasTranscurridos += dia;

        /*
        ** Para anios biciestos hay que considerar un dia mas, si se busca un dia
        ** en un mes mayor al segundo (febrero).
        */
        if (!(anio%4) && mes >2)
                diasTranscurridos ++;

        return diasTranscurridos;
}
```

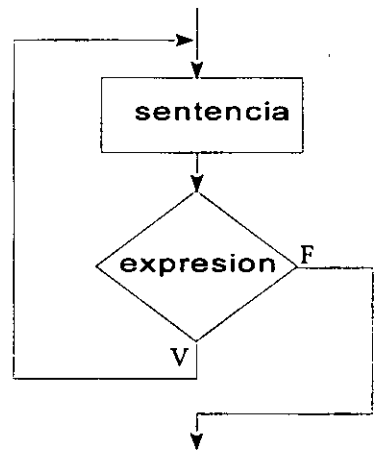
```
main()
{
        int     diaNac, mesNac, anioNac,
                diaHoy, mesHoy, anioHoy,
                diasVividos;

        printf("\tDia de Nacimiento [1-31]: ");
        scanf("%d",&diaNac);
        printf("\tMes [1-12]: ");
        scanf("%d",&mesNac);
        printf("\tAnio: ");
        scanf("%d",&anioNac);
        printf("\tDia actual [1-31]: ");
        scanf("%d",&diaHoy);
        printf("\tMes [1-12]: ");
        scanf("%d",&mesHoy);
        printf("\tAnio: ");
        scanf("%d",&anioHoy);

        diasVividos = 0;
        for ( anio = anioNac ;  anio < anioHoy ; anio++)
                /*
                ** Los dias vividos en el primer a&o se cuentan a partir
                ** del dia de nacimiento.
                */
                if (anio == anioNac)
                        diasVividos = diasDelAnio(anioNac) -
                                        diaDelAnio(anioNac,mesNac,diaNac);
                else
                        diasVividos += diasDelAnio(anio);

        if ( anioNac != anioHoy)
                /*
                ** Si el anio de nacimiento no es el actual, le sumamos los dias
                ** transcurridos de este anio.
                */
                diasVividos += diaDelAnio(anioHoy, mesHoy, diaHoy);
        else
                diasVividos = diaDelAnio(anioHoy, mesHoy, diaHoy) -
                        diaDelAnio(anioNac, mesNac, diaNac);
```

}

do

Sintaxis:

```
do
        sentencia
while (expresión);
```



La secuencia de ejecución es la siguiente:

1. Se ejecuta la sentencia.

2. Se evalúa la expresión:

   si la evaluación es falsa termina el ciclo.
   si la evaluación es verdadera se vuelve al paso 1.

Ejemplo:

```
/*
**      Programa 3_6
**
**      Este programa genera una letra mayuscula del alfabeto aleatoria (tomando
**      como base la funcion rand) para que sea adivinada. Esta tarea se realiza
**      en un ciclo que termina cuando el usuario ya no desee continuar.
**      Version utilizando do-while.
*/
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>

main()
{
        char    respuesta='s';
        char    car,                                            /* Caracter generado*/
                apuesta;                                /* Caracter proporcionado por el usuario*/

        do
        {
                randomize();                    /* Se inicializa la semilla para numeros aleatorios */
                car = 'A' + rand() % 26;        /* Con esta formula se obtiene alguna de las 26 letras */

                printf("Adivina el caracter: ");
                apuesta = getchar();
                getchar();                      /* Esta lectura sirve para vaciar el buffer de la entrada, */
                                                        /* la cual  contiene el caracter nueva linea */
                if (islower(apuesta))           /* Si el caracter proporcionado es minuscula */
                        apuesta -= 32;                          /* se convierte a mayuscula */
                if (car == apuesta)
                        printf("FELICIDADES ADIVINASTE EL CARACTER!!!\n");
                else
                        printf("Perdiste ..., el caracter es=> %c\n",car);
                printf("\nContinuas [s/n]? ");
                respuesta = getchar();
                getchar();
        } while ( respuesta == 's' || respuesta == 'S');
}
```

Operador comma

Este operador sirve para agrupar dos expresiones en una sola, frecuentemente es utilizado en la sentencia *for* para colocar expresiones múltiples en la *expresión1* o en la *expresión3,* para procesamiento de indices en paralelo.

La sintaxis es la siguiente:

expresión1, expresión2

El resultado y tipo de la expresión anterior son el resultado y tipo de la *expresión2.*

Ejemplo:

```
/*
**      Programa 3_7
**
**      Programa que despliega dos columnas de números, una en forma ascendente
**      y otra en forma descendente.
**
*/

#include <stdio.h>

#define     N     10

main()
{
        int     i, j;

        for(i=0, j=N; i<=N && j>=0; i++, j--)
                printf("\t%d\t%d\n",i,j);
```

switch

La proposición switch permite la implementación de decisiones múltiples con valores enteros.

Sintaxis:

```
switch (expresión) {
        case  exp-const:   sentencia1
                           sentencia2
                           . . .
        case  exp-const2:  sentencia2_1
                           sentencia2_2
                           . . .
        default:           sentenciaN_1
                           sentenciaN_2
                           . . .
}
```

donde: exp-const = expresión constante entera
sentencia1,... = cualquier sentencia válida

La expresión se evalúa y el resultado se compara con las expresiones constantes; si alguna de ellas coincide, el control del programa se traslada a ese punto.

La expresiones constantes deben ser enteras y no se deben repetir.

Las sentencias después de la expresión constante no se necesitan agrupar como bloque.

La cláusula **default** es opcional e indica el lugar a donde se traslada el control del programa en el caso en que ninguna de las etiquetas **case** coincidan con el valor de la expresión.

Ejemplo:

```
/*
**      Programa 3_8
**
**      Este programa muestra el comportamiento de la estructura switch.
**
*/
main()
{
        int     opcion;

        printf("Proporciona tu opcion: ");
        scanf("%d",&opcion);
        switch ( opcion )
        {
                case 1: printf("OPCION 1\n");
                case 2: printf("OPCION 2\n");
                case 3: printf("OPCION 3\n");
                case 4: printf("OPCION 4\n");
                default:printf("OPCION DEFAULT\n");
        }
}
```

¿Cuál es el resultado del segmento de programa anterior, cuando la opcion proporcionada es 3?

break

Un ***break*** causa una salida inmediata de las siguientes construcciones:

- while
- for
- do
- switch

continue

En las sentencias ***do*** y ***while*** un ***continue*** provoca la evaluación de la **expresión.**

En un ***for*** el control del programa pasa a la **expresión3.**

Ejemplo:

```
/*
**      Programa 3_9
**
**      Este programa simula un juego semajante a la loteria.
**      El jugador debe adivinar la letra generada por el programa, si la adivina, ganara la
**      parte proporcional del premio mayor de acuerdo a la apuesta hecha. El premio
**      mayor es de 10,000 y el billete cuesta 2000.
**      De inicio el jugador cuenta con cierta cantidad de dinero, de la cual puede disponer
**      para sus apuestas.
**
*/
#include    <stdio.h>
#include     <stdlib.h>
#include     <ctype.h>
#define     MINIMO     100                    /*Cantidad minima que se puede apostar */
#define     PREMIO     10000                  /* Premio mayor */
#define     BILLETE    2000                   /* Valor del billete */

main()
{
        char    respuesta;
        char    car,                           /* Caracter generado por el programa */
                apuesta;                       /* Caracter apostado por el jugador */
        long    banco,                         /* Dinero con el que cuenta el banco */
                jugador;                       /* Dinero con el que cuenta el jugador*/
        int     dineroApostado,    /* Dinero apostado por el jugador (menor o igual al billete) */
                dineroGanado;                  /* Dinero ganado en la apuesta */
        float   porcentaje,    /* Porcentaje con respecto al valor del billete que el jugador apuesta */

        banco = 100000;
        jugador = 10000;
        printf("*****************************************\n");
        printf("Inicialmente cuentas con N$%10.2ld\n",jugador);
        printf("El banco tiene N$%12.2ld\n",banco);
        printf("El premio mayor es de => %d\n", PREMIO);
        printf("El billete cuesta => %d",BILLETE);
        printf("*****************************************\n\n");
        respuesta = 's';
```

```
randomize();                    /* Se inicializa la semilla de los numeros aleatorios */
while ( respuesta == 's' || respuesta == 'S')
{
        /*
        ** Se genera una letra mayuscula en base a un numero aleatorio.
        */
        car = 'A' + rand() % 26;
        printf("Cuanto apuestas? ");
        scanf("%d",&dineroApostado);
        fflush(stdin);          /* Se vacia el buffer, ya que el caracter nueva linea queda en el */

        /*
        ** El jugador debe contar con el dinero apostado y no puede ser mayor al
        ** precio del billete.
        */
        if (dineroApostado > jugador || dineroApostado > BILLETE)
                continue;
        printf("Adivina el caracter: ");
        apuesta = getchar();
        fflush(stdin);
        if (islower(apuesta))   /* La letra propuesta por el jugador se convierte a mayuscula */
                apuesta -= 32;
        porcentaje = ((float)BILLETE)/dineroApostado;
        if (car == apuesta)
        {
                dineroGanado = PREMIO * porcentaje;
                jugador += dineroGanado;
                banco -= dineroGanado;
                printf("FELICIDADES ADIVINASTE EL CARACTER!!!\n");
        }
        else
        {
                jugador -= dineroApostado;
                banco += dineroApostado;
                printf("Perdiste ..., el caracter es=> %c\n",car);
        }
        printf("\n\nBANCO    => N$%12.2ld\n",banco);
        printf("JUGADOR  => N$%12.2ld\n\n",jugador);
```

```
            if (jugador < MINIMO)          /* Si el jugador ya no tiene dinero suficiente */
            {
                printf("\nLo siento no eres sujeto de credito, ADIOS.\n");
                break;
            }
            printf("\nContinuas [s/n]? ");
            respuesta = getchar();
            fflush(stdin);
        }
}
```

LABORATORIO

1. Una persona que recibe pagos por honorarios desea hacer un programa que le calcule automáticamente el monto de sus impuestos de acuerdo a los ingresos mensuales que percibe. Lo único que el sabe es que sus impuestos se calculan en base a la siguiente tabla:

Limite inferior	limite superior	Cuota fija	Porcentaje
0	800	0	0
800	1500	80	10
1501	2500	180	15
2501	4500	400	20
4501	6500	950	25
6501	-	1600	30

La formula utilizada es la siguiente:

impuesto = cuota fija + (salario - limite_inf) * porcentaje/100

Si el ingreso mensual de una persona es de 4750, entonces el monto por el pago de sus impuestos que debera pagar es:

impuesto = 950 + (4750 - 4501)*(25/100) = 1012.25

2. Escriba un programa que genere los primeros N números de la secuencia de Fibonacci. Los dos primeros número de la secuencia son: 0 , 1 y a partir de estos los siguientes se generan en base a la siguiente formula:

$$numero_N = numero_{N-1} + numero_{N-2}$$

Los primeros 5 número de la serie son: 0 1 1 2 3 ....

3. Escriba un programa que imprima los primeros N números primos. Un número primo es aquel que es divisible solamente por el mismo y la unidad. Los primeros 5 número primos son: 1, 2, 3, 5, 7.

Página intencionalmente blanca.