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

CURSOS INSTITUCIONALES

LENGUAJE DE PROGRAMACION "C"

Del 3 al 19 de Octubre de 1994

ERICSSON, S.A. DE C.V.

L E N G U A J E " C "

PARTE I

Ing. Jesica Briseño
México, D.F.

1994

LENGUAJE DE PROGRAMACION C

TEMARIO

1. Introducción

2. Tipos, operadores y expresiones

3. Control de flujo

4. Funciones

5. Arreglos y apuntadores

6. Estructuras

7. Manejo de archivos

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.

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).

Un programa en C esta compuesto de uno o más archivos fuente.

Cada archivo fuente puede ser compilado independientemente.

Edición:

vi archivo.c

Compilación exclusivamente:

cc -c archivo.c

Compilación y ligado:

cc -c archivo.c

cc archivo.o -o archivo

cc archivo1.c archivo2.c archivo3.o

Ejecución:

a.out

archivo

El programa en C más famoso

```
/*
        Programa No. 1

        Este es el primer ejemplo de un programa en C
*/

#include <stdio.h>

main()
{
        printf("Hola Mundo\n");
}
```

Ejemplo No. 2

```
/*
  Programa No. 2

  Obtiene el mayor de 2 números

*/


#include <stdio.h>

main()
{
        int n1, n2;

        printf("Proporciona 2 números: ");
        scanf("%d",&n1);
        scanf("%d",&n2);
        if (n1 > n2)
                printf("El número mayor es: %d\n", n1);
        else
                printf("El número mayor es: %d\n", n2);
}
```

Ejemplo No. 3

```
/*
  Programa No. 3

  Obtiene el mayor y menor de N números

*/


#include <stdio.h>

#define         N       5

main()
{
        int max,min;
        int i;
        int numero;

        i = 1;
        printf("Dame el numero %d: ",i);
        scanf("%d",&numero);
        max = numero;
        min = numero;
        while (i<N) {
                i = i + 1;
                printf("Dame el numero %d: ",i);
                scanf("%d",&numero);
                if (numero > max)
                        max = numero;
                if (numero < min)
                        min = numero;
        }
        printf("\n\tEl número mayor es: %d\n\ty el menor: %d",                max,min);
}
```

Ejemplo No. 4

```
/*
  Programa No. 4

  Cuenta los caracteres de la entrada

*/


#include <stdio.h>

main()
{
        int     n=0;

        while(getchar() != EOF)
                n++;
        printf("\n\tTecleaste %d caracteres\n",n);
}
```

Ejemplo No. 5

```
/*
  Programa No. 5

  Cuenta los caracteres de la entrada. Versión 2

*/

#include <stdio.h>

main()
{
        int     n=0;

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

Ejemplo No. 6

```
/* Programa No. 6

  Cuenta lineas, palabras y caracteres de la entrada.

*/

#include <stdio.h>

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

    nc = np = nl = 0;
    while((c = getchar()) != EOF) {
        nc++;
        if (c == '\n')
            nl++;
        if (c >= 'a' && c<= 'z'  || c>= 'A' && c<= 'Z') {
            np++;
            while ((c=getchar()) >= 'a' && c <= 'z' || c>= 'A' && c<= 'Z')
                    nc++;
          ungetc(c,stdin);
            nc--;
        }
    }
    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.

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 6 para que se cuenten cualquier tipo de palabras (palabras que incluyan numeros, palabras con mayúsculas, etc.).

4. Haga un programa que convierta la entrada de datos a mayúsculas

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.

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

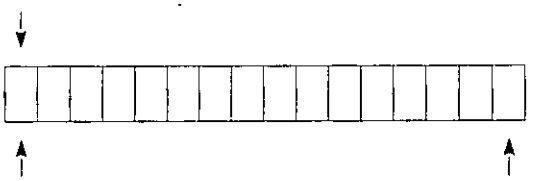
Enteros

El tamaño de los tipos enteros depende de la máquina

Los tipos enteros signados son:

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

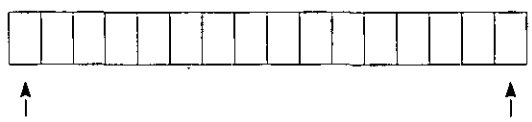
Representación:



Los enteros sin signo son:

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

Representación:



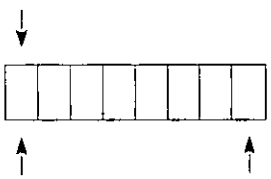
Carácter

Los tipos carácter son:

- [unsigned] char

Sus valores son enteros

Representación:

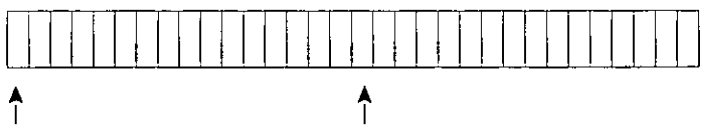


Punto flotante

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 se representan por más de un carácter:

```
'\n'      '\t'      '\f'
'\a'      '\b'
```

- También se pueden representar: '\033', '\0xff'.

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 = zeta;
```

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 1

   Este programa determina el tamaño de los tipos básicos

*/

#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      car = 'a';

     ...

}
```

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	float double	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.

Los operadores de igualdad son los siguientes:

```
!=         ==
```

Lo 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 1, 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 3

   Este programa muestra el comportamiento de los operadores
lógicos

*/

#include <stdio.h>

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

     x = 0 && (i = j = 999);
     printf("%d  %d  %d\n",i,j,x);  /* se imprime 0 0 0 */
     y = 1 || (i = j++);
     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 pueden aplicar únicamente a variables.

```
/*
   Programa 4

  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);  /* se imprime 2 1 1 */
     a = b++ + c++;
     printf("\n%d  %d  %d", a,b,c);  /* se imprime 2 2 2 */
     a = ++b + c++;
     printf("\n%d  %d  %d", a,b,c);  /* se imprime 5 3 3 */
     a = b-- + --c;
     printf("\n%d  %d  %d", a,b,c);  /* se imprime 5 2 2 */
     a = ++c + c;
     printf("\n%d  %d  %d", a,b,c);  /* depende de la maquina
*/
     a = ++c +++b;
     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 operadores:
+, -, *, /, %,
<<, >>, &, ^, |

expresión = cualquier expresión

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

var operador= expresión

No debe existir blanco entre operador y =

```
/*
   Programa 5

   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) */

}
```

Operadores para manejo de bits

Operadores binarios lógicos de bits

Los operadores binarios lógicos de bits son: & (and) , | (or) y ^ (xor).

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

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: >> y <<

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.
- 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. Pruebe el programa No. 1.

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

3. 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.

4. 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.

CONTROL DE FLUJO

Expresiones y sentencias

Una expresión puede ser:

- una variable o constante
- una llamada a una función
- una combinación de operandos y operadores

Ejemplos de expresiones:

a=5 a++ 135.4 a*b/c sin(a)

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++;
}
```

if

Un if es en si una sentencia condicional.

Su sintaxis es la siguiente:

```
if (expresión)
        sentencia
else
        sentencia
```

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)
        if (n > c)
                z = c;
else
        z = 0;
...
```

Existen errores comunes como el siguiente:

```
...

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 (expresión)
        sentencia
else if (expresión)
        sentencia
else if (expresión)
        sentencia
else if (expresión)
        sentencia
else if (expresión)
        sentencia
else
        sentencia
```

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 se ejecuta cuando ninguna expresión es verdadera

Ejemplo:

```
...

if (x > y)
        printf("%d es mayor que %d\n",x,y);
else if (y > x)
        printf("%d es mayor que %d\n",y,x);
else
        printf("%d y %d son iguales\n",x,y);

...
```

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 No. 1

        Programa que imprime el mayor de dos números

*/
#include <stdio.h>

main()
{
        int     x=5, y=8;

        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 No. 2

        Programa que imprime el mayor de tres números

*/
#include <stdio.h>

main()
{
        int     x=5, y=8, z=2;

        printf("%d es el n·mero mayor entre %d, %d y %d\n",
                ((x >y) ? ((x>z) ? x : (y>z) ? z : y)) : ((y>z) ? y : z) ),
                 x, y,z);
}
```

while

Sintaxis:

```
while (expresión)
        sentencia
```

La sentencia se ejecuta mientras la evaluación de la expresión sea verdadera

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.

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.

Operador comma

Este operador sirve para agrupar dos expresiones como 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 expresión2.

Ejemplo:

```
/*
        Programa No. 3

        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("%d      %d\n",i,j);
}
```

switch

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

Sintaxis:

```
switch (expresión) {
        case exp-const:   sentencias
        case exp-const2: sentencias
        default: sentencias
}
```

donde: exp-const = expresión constante entera

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:

```
...

int     x = 3;

switch (x) {
        case 1: printf("*");
        case 2: printf("**");
        case 3: printf("***");
        case 4: printf("****");
}

...
```

¿Cuál es el resultado del segmento de programa anterior?

La proposición break provoca una salida inmediata del switch.

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.

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/10

2. Escriba un programa que genere los primeros N números de la secuencia de Fibonacci.

0 1 1 2 3 ....

3. Escriba un programa que imprima los primeros N números primos.

FUNCIONES Y EL PREPROCESADOR

FUNCIONES

Las funciones son elementos que permiten el desarrollo de programas modulares.

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

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

Las funciones pueden presentarse en cualquier orden dentro del archivo fuente, y el archivo fuente se puede dividir en varios archivos, mientras las funciones no se dividan.

En el lenguaje C, los módulos de un programa se implementan por medio de funciones; no existe el concepto de subrutina o procedimiento.

Todas las funciones se definen al mismo nivel, no se puede definir una función en otra.

Las funciones pueden ser recursivas, salvo main.

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

```
tipo_retorno    nombre (lista_parámetros ) {
        declaraciones y sentencias
}
```

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 se puede especificar como tipo *void*.

Ejemplo:

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

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.

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

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 la llamada actual.

Ejemplo:

```
double  maximo(double, double);

main() {

        double x=5, y=8, z;

        z = maximo(x,y);

   ...

}


double  maximo(double x, double y) {

...
}
```

Valores de regreso

Una función puede regresar un valor asociado al tipo de retorno.

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.

La expresión del return debe de ser del mismo tipo que el especificado en el tipo de retorno, o bien, debe de existir una conversión explícita.

En una función pueden existir más de una cláusulas return, en caso contrario la función termina al alcanzar su última sentencia y el valor de retorno es indefinido.

Ejemplo:

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

        if (x > y)
                return x;
        return y;
}
```

Paso de parámetros

Los parámetros actuales de una función son la lista de valores con los que se hace una llamada a una función.

Los parámetros formales de una función son la lista de variables que se definen en la definición de la función.

El paso de parámetros es por valor.

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 necesario) 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:

```
int incrementa(int);

main() {

        int     i=3, j;

        ...

        j = incrementa(i);
        printf("\n%d\t%d\n",i,j);

}

int incrementa(int x) {

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

El ejemplo anterior genera la siguiente salida:

```
4
3       4
```

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;
        auto int  x,y;

        ...

        for (i=0; i< N ; i++) {

                int     a,b;

                ...

        }
}
```

Variables externas

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

Para estas funciones, 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*.

Solamente debe existir una definición de una variable externa.

Todas las funciones que aparecen después de la definición de una variable externa pueden accesar a esta.

Variables estáticas

Las variables estáticas son automáticas a un bloque o función; pero retienen su valor una vez que termina la llamada a la función.

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

Ejemplo:

```
int     fun1(void);

main() {

        ...

        for(i=0; i < 10 ; i++)
                fun1();
        ...

}


int fun1() {

        static int  x=1;

        printf("\t%d"x);
        x++;
}
```

La salida del programa anterior es:

1 2 3 4 5 6 7 8 9 10

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:

```
int fun(int n) {

    int x = n;
    int i=0, j=0;

    ...

}
```

Reglas de alcance

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

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. 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

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

Ejemplo:

```
int     fun1(int);
int     fun2(int);

int     x=5, y;

main() {

        int x=10;

        x = fun1(x);
        y = fun1(y);
        printf("\n%d %d",x,y);
        x = fun2(y);
        printf("\n%d %d",x,y);

}

int fun1(int  y) {

        x+=y++;

        printf("\n%d %d",x,y);
        return x++;

}

int fun2(int  x) {

        int     y=x;

        y+= y;
        printf("\n%d %d",x,y);
        return y;

}
```

¿Cuál es la salida del programa anterior?

Recursividad

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

Una función recursiva es aquella que se llama así misma directa o indirectamente.

Cada llamada recursiva reserva espacio para las variables automáticas que se definen en ella.

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

Ejemplo:

```
int     factorial(int n) {

        if (!n)
                return 1;
        return n*factorial(n-1);
}
```

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.

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.

EL PREPROCESADOR

C proporciona ciertas facilidades por medio de un preprocesador, que actúa antes que el compilador y ejecuta las instrucciones que comienzan con el carácter #.

El efecto de las líneas de control o instrucciones del preprocesador van desde el lugar que aparecen hasta el final del archivo en donde se encuentran.

include

Sintaxis:

```
#include <archivo>
#include "archivo"
```

Esta instrucción del preprocesador sustituye el contenido del archivo a partir del lugar en donde aparece la instrucción.

Cuando el nombre de archivo esta limitado por <>, el preprocesador busca el archivo en un directorio asignado por default, generalmente /usr/include.

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.

define

Sintaxis:

```
#define      id      token_string
#define      id(id, ..., id)   token_string
```

1. #define id token_string

El preprocesador sustituye cualquier ocurrencia del id por el token_string en el archivo en donde aparece esta instrucción, a excepción de los comentarios y cadenas de caracteres encerradas por "".

Ejemplo:

```
#define      SEG_X_DIA   (60 * 60 * 24)
```

El preprocesador sustituye todas las ocurrencias de SEG_X_DIA por (60 * 60 * 24), de tal forma que el compilador no tiene conocimiento de la existencia de SEG_X_DIA.

La expresión que reemplaza no se evalúa.

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:

```
#undef       id
```

2. #define id(id, ..., id) token_string

Sirve para la definición de macros:

#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), esta secuencia se sustituirá por ((3) * (3)).

Los paréntesis parecen excesivos, sin embargo son necesarios, suponga:

#define CUADRADO(x) (x * x)

si existe una línea como:

CUADRADO(7 + i)

la sustitución será:

(7 + i * 7 + i)

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

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.

Compilación condicional

El preprocesador incluye algunas instrucciones que permiten llevar a cabo tareas de rastreo en los programas.

Las líneas de control

```
#ifdef  id
#ifndef         id
```

inician la compilación condicional del texto que les sigue hasta encontrar la línea de control #endif.

Este tipo de instrucciones sirve para que un archivo de encabezado pueda incluirse sin ningún problema en cualquier programa, evitando dobles definiciones:

```
#ifndef         EOF
#define         EOF    -1

....

#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.

Desde main pueden hacerse llamadas a funciones definidas en otros archivos, pero es recomendable que antes se declaren en todos los archivos en donde se utilicen.

Ejemplo:

prog1.c

```
/*
   Programa No. 1

   Programa que calcula el numero mayor y menor de una
   secuencia de N números

*/

#include <stdio.h>
#include "maxMin.h"

#define        N       10

main() {

        int     i, max, min, n;

        printf("\tProporciona 10 numeros:\n");
        scanf("%d",&n);
        max = min  = n;
        for(i=0 ; i < N ; i++) {
                scanf("%d", &n);
                max = maximo(max, n);
                min = minimo(min, n);
        }
        printf("%d es el mayor y %d el menor %d\n", max, min);
}
```

maxMin.c

```
/*  maxMin

        Archivo que contiene la definición de las funciones maximo y minimo

*/

#include <stdio.h>
#include "maxMin.h"

int maximo(int max, int x) {

        if (max > x)
                return max;
        return x;
}

int minimo(int min, int x) {

        if (min < x)
                return min;
        return x;
}
```

maxMin.h

```
/* maxMin.h

        Archivo de prototipos y definición de constantes para las funciones maximo y
        minimo

*/

int     maximo(int, int);
int     minimo(int, int);
```

makefile

```
maxMin.o: maxMin.c maxMin.h
        cc -c maxMin.c

prog1:  maxMin.o prog1.c maxMin.h
        cc prog1.c maxMin.o -o prog1
```

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];
```

El identificador de un arreglo es un apuntador constante, que guarda la dirección de inicio del arreglo.

Ejemplo:

```
int x[10];
```

Para accesar a los elementos de un arreglo hay que hacer referencia a su índice.

El primer elemento de un arreglo es el que tiene el índice cero.

En el ejemplo anterior, el primer elemento es x[0] y el último x[9].

Cuando se define un arreglo se reservan localidades continuas de memoria para almacenarlo, aún cuando el arreglo sea multidimensional.

Ejemplo:

```
/* Programa 1

        Este programa muestra el manejo de un arreglo en un programa.

*/

#include <stdio.h>
#define         N       100

main()
{
        int vector[N], i;

        for(i=0 ; i < N; i++)
                vector[i] = i;
        for(i=0 ; i < N; i++)
                printf("%c%3d", i%10 ? ' ' : '\n', vector[i]);
}
```

Inicialización de arreglos

Los arreglos externos y estáticos de enteros inicializan sus elementos con cero.

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

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

en este caso:

- 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.

- Cuando se inicializa un arreglo no es necesario especificar su dimensión, se definición será de acuerdo al número de inicializadores:

```
int x[] = { 1, 5, 5, 7};
```

Arreglos de caracteres

En el lenguaje C no existe el tipo "string" o "cadena".

Una cadena puede ser representada con un arreglo de caracteres

Para la manipulación de cadenas, por convención, el termino de esta se indica con el carácter '\0'.

Por lo tanto, para un arreglo de tamaño N, la longitud máxima de la cadena será de N-1.

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

Las constantes cadena se escriben entre comillas, por ejemplo, "esto es una cadena", es un arreglo de caracteres de tamaño 19, donde el último elemento es '\0'.

Por lo tanto, las constantes "A" y 'a' no son iguales, la primera de ellas representa un arreglo de dos elementos y la segunda es un carácter.

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

```
char   cadena[] = { 'c', 'u', 'r', 's', 'o', '\0'};
```

o bien:

```
char      cadena[] = "curso";
```

Ejemplo:

```
/* Programa 2

        Este programa muestra el manejo de un arreglo de caracteres.

*/

#include <stdio.h>
#define         N       100

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

        printf("\tDame tu nombre: ");
        while ((c = getchar()) != '\n')
                nombre[i++] = c;
        nombre[i] = '\0';
        printf("\nMuchas gracias %s por haber dado tu nombre\n",
                nombre);
}
```

Arreglos multidimensionales

El lenguaje C permite la definición de arreglos multidimensionales, que en realidad son arreglos de arreglos.

En un arreglo multidimensional, las localidades de memoria que se reservan, al igual que en un arreglo unidimensional, son continuas.

La forma de definir un arreglo multidimensional, por ejemplo de dos dimensiones, sería:

```
int  x[10][10];
```

Los arreglos multidimensionales más utilizados son aquellos de dos dimensiones, que permiten la representación de matrices. 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 representará a las columnas.

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

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

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

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

Cuando se define un arreglo multidimensional, es necesario indicar todas sus tamaños a excepción del primero, de modo que el compilador pueda determinar la función correcta de transformación de almacenamiento.

La función de transformación de almacenamiento se utiliza para calcular la localidad de un elemento 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

Para un arreglo definido como: int x[][10], el elemento x[5][4] se localizará 10*5 + 4 posiciones después del inicio del arreglo.

LABORATORIO 1

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 o -1 si no se encuentra.

2. Escriba el programa número 1 del laboratorio del capítulo 3 para calculo de impuestos utilizando un arreglo para almacenar la tabla.

APUNTADORES

Todas las variables se almacenan en un cierto número de bytes a partir de cierta dirección de memoria en la máquina.

Un apuntador es una variables que almacena la dirección de memoria de otra variable.

El tipo de la variable para la cual se almacena la dirección puede ser cualquiera y determina el tipo de apuntador: apuntador a entero, apuntador a carácter, 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; sin embargo, la dirección que contiene, hasta después de la definición es una dirección desconocida.

Para asignar a un apuntador una dirección válida se puede utilizar el operador de dirección &. Por ejemplo, supongase que x es una variable entera:

```
p = &x;
```

asigna a p la dirección de x y se puede accesar el valor de la variable x 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 dirección que contiene un apuntador, se utiliza el operador de desreferencia o indirección *. Por ejemplo, para cambiar el valor de la variable entera x en forma indirecta:

```
*p = 15;
```

en este caso *p es una variable entera y se puede utilizar en cualquier contexto que acepte valores enteros:

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

p = &x;
y = *p + 1;
d = sqrt(*p);

*p = 0;
*p += 1;
(*p)++;

q = p;
```

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 */
```

A un apuntador de le puede asignar la dirección de una localidad de memoria que se reserva al momento de ejecución del programa:

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

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 almacenados de los parámetros actuales.

Para que una función cambie el valor de una variable a la cuál se puede hacer referencia desde la función que hace la llamada, es necesario que esta sea definida como externa:

Ejemplo:

```
/* Programa 3

        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 l a sentencia return en una función

Ejemplo:

```
/* Programa 3.1

        Este programa muestra el regreso de valores en la cláusula return.

*/
#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);
```

}

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 una función que intercambia el valor de sus dos parámetros:

Ejemplo:

```
/* Programa 4.1

        Programa que implementa la función swap sin manejo de apuntadores

*/

#include <stdio.h>

int swap(int,int);

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

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

int 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 del programa ?

El mismo ejemplo, implementando paso de parámetros por referencia con el uso de apuntadores:

Ejemplo:

```
/* Programa 4.2

        Programa que implementa la función swap con manejo de apuntadores

*/

#include <stdio.h>

int swap(int *, int *);

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

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

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

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

En el ejemplo:

- Se deben declarar los parámetros como apuntadores.

- Hay que utilizar el operador de indirección en la definición de la función.

- El parámetro actual en la llamada a la función es 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, es decir, es un apuntador constante.

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 continuas de 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          */
```

¿ Cuál es la salida del siguiente programa ?

```
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));
}
```

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 declarado como arreglo a como apuntador.

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

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.

¿ Es correcto el siguiente programa ?

¿ Cuál es la salida ?

```
#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);

}

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 1876; la expresión p + 1 no es necesariamente la dirección 1877.

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

La siguiente tabla muestra una serie de funciones que 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 como valor cero, si el carácter es del tipo que se esta validando y un valor diferente 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.

Las cadenas son un arreglo de caracteres terminados con '\0'.

Se debe de indicar el header 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 un valor 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 un valor 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 s1.
Regresa 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 2

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

Arreglos de apuntadores

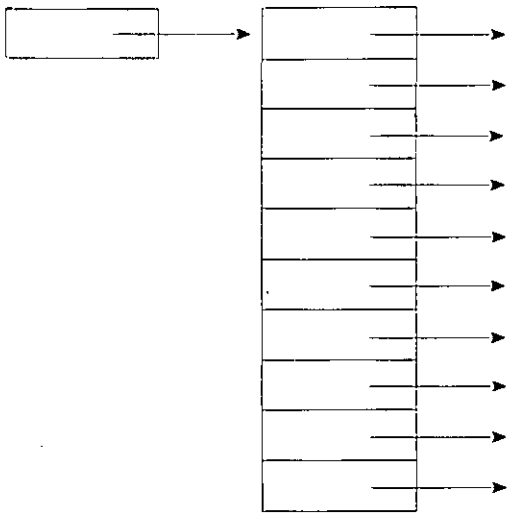
El lenguaje C permite la definición de tipos a partir de los ya definidos, de esta forma se pueden crear arreglos de apuntadores, arreglos de arreglos de apuntadores, etc.

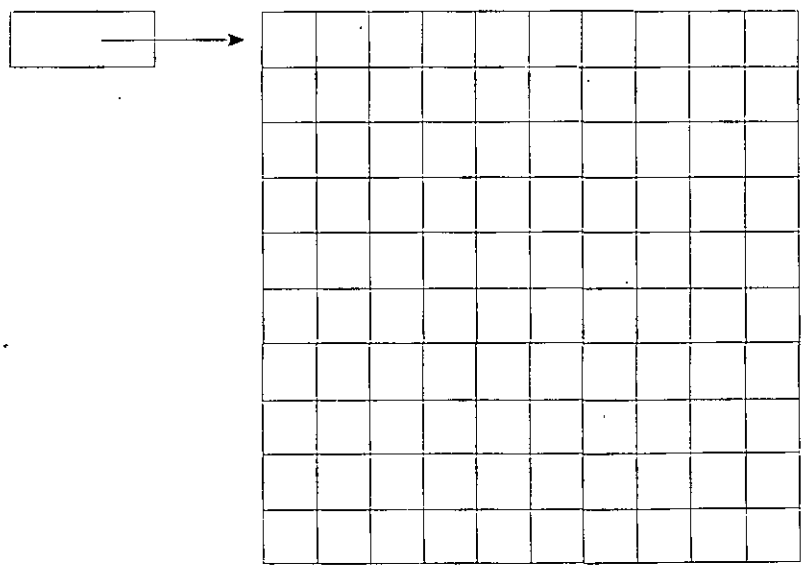
Los arreglos de apuntadores son mucho más flexibles que los arreglos multidimensionales, además de que generalmente, requieren de menos memoria.

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.





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(sizeof(N));
```

De esta forma cada uno de los elementos se puede considerar como un arreglo de N caracteres y se puede hacer referencia a cada elemento de un arreglo con la notación utilizada para arreglos bidimensionales.

Ejemplo:

```
/*
   Programa 5

      Este programa muestra el manejo de arreglos de
      apuntadores.

      Se leen n cadenas de caracteres y se reserva memoria a
      tiempo de ejecución.

*/

#include <stdio.h>

#define     MAX     20  /* Máximo número de elementos en */
                        /* el arreglo                    */
#define     MAX_NOM 35  /* Máximo número de caracteres   */
                        /* una cadena                    */

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) {
                agenda[i] = (char *)malloc(sizeof(strlen(nombre)+1));
                strcpy(agenda[i], nombre);
                i++;
        }
   ordena(agenda, i);
   printf("\n\nLista Ordenada:\n");
   imprime(agenda, i);
}
```

```
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--)
         if ((strcmp(x[j-1], x[j])) > 0) {
                                          aux = x[j-1];
                                          x[j-1] = x[j];
                                          x[j] = aux;
         }
}

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 6

            Este programa despliega los parámetros que se le pasan en la línea de comandos

*/

#include <stdio.h>

main(int argc, char **argv) {

        int i;

        for(i=0 ; i < argc; i++)
                printf("%s ", argv[i]);
}
```

ESTRUCTURAS Y UNIONES

TYPEDEF

C es un lenguaje que puede ampliarse con facilidad.

La extensión del lenguaje se puede llevar a cabo mediante los #define y creando funciones de propósito general para uso de todos los usuarios.

También puede ampliarse al definir tipos de datos que se construyen con los tipos estándar.

También se pueden definir tipos con componentes no homogéneos con el uso de estructuras.

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 asociación explícita de un tipo con un identificador.

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 o funciones, de la misma forma en como se utilizan los tipos estándar:

```
STRING      lista[N];
MATRIZ      a,b,c;
```

Las definiciones de tipo permiten la documentación de programas, ya que normalmente las definiciones de tipo junto con los prototipos y las definiciones de símbolos con #define se colocan en archivos header.

Además, cuando existen declaraciones sensibles al sistema, como el caso de int, que en los sistemas UNIX es de cuatro bytes y en otros es de dos, y si estas diferencias son críticas para el programa, el empleo de typedef permite que los programas sean transportables.

ESTRUCTURAS

Las estructuras permiten la representación de elementos de diferentes tipos por medio de un identificador.

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

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

Se podría utilizar una estructura para definir un tipo que representara a un empleado:

```
struct emp {
        char        noCta[8],
                    nombre[35],
                    dirección[35],
                    teléfono[10],
                    sexo,
                    tipo;
        float       salario;
};
```

De esta forma se pueden definir variables de tipo struct emp, así como arreglos de ese tipo:

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

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

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

Con la definición anterior las definiciones de variables serían:

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

En la definición de el tipo EMPLEADO, la etiqueta emp después de struct es opcional, cuando se coloca permite que se pueda utilizar el 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 miembro ".".

Una construcción de la forma:

variable_estructura.nombre_miembro

se utiliza como variable de la misma forma que se utiliza una variable simple o un elemento del mismo tipo.

Por ejemplo, para asignar valores a la variable empleado de tipo EMPLEADO:

```
strcpy(empleado.noCta, "811CAFA");
strcpy(empleado.nombre,"C. Jéssica Briseño C.");
empleado.salario = 6500;
```

El valor de una estructura se puede asignar directamente, por ejemplo, si empleado y empleado1 son estructuras del mismo tipo:

empleado1 = empleado

Las funciones pueden recibir parámetros de algún tipo de estructura o regresar el valor de alguna estructura.

Ejemplo:

```
/*
        Programa 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:\n");
        scanf("%f",&a.real);
        scanf("%f",&a.imaginaria);
        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);
}
```

```
COMPLEJO   suma(COMPLEJO  x, COMPLEJO y) {

                COMPLEJO   z;

                z.real = x.real + y.real;
                z.imaginaria = x.imaginaria + y.imaginaria;
                return z;
}


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',
                       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        lista[N];
```

Así, para poder leer una lista de información de empleados:

```
for (i = 0; i < N; i++) {
                printf("Num. de Cta.: ");
                gets(lista[i].noCta);
                printf("Nombre: ");
                gets(lista[i].nombre);
                printf("Dirección: ");
                gets(lista[i].dirección);
                printf("Teléfono: ");
                gets(lista[i].teléfono);
                printf("Sexo: ");
                scanf("%c",&lista[i].sexo);
                printf("Tipo: ");
                scanf("%c",&lista[i].tipo);
                printf("Salario: ");
                scanf("%f",&lista[i].salario);
}
```

LABORATORIO 1

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

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

UNIONES

Una unión al igual que una estructura, es un tipo derivado.

Las uniones tienen la misma sintaxis que las estructuras, pero comparten el almacenamiento.

Una unión define a un conjunto de valores alternos que pueden almacenarse en una porción compartida de la memoria.

El compilador asigna una porción de almacenamiento que pueda acomodar al más grande de los miembros especificados.

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

El sistema interpreta los valores almacenados de acuerdo al miembro seleccionado, elegir el correcto es responsabilidad del programador.

Ejemplo:

Suponga que se quiere representar la información de los empleados de una empresa; sin embargo, existen diferentes tipos de empleados (vendedor, administrativo, directivo, 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 y un bono adicional, los vendedores perciben además de su sueldo base comisiones y premios, a los administrativos se les paga las horas extras.

Los tipos utilizados serían los siguientes:

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


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


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


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;
        SALARIO   salario;
} EMPLEADO;
```

De esta forma en base al campo tipo se podría determinar como manejar la unión:

```
EMPLEADO        emp;

...

if (emp.tipo = 'V') {
        emp.salario.vendedor.comision = 1000;
        emp.salario.vendedor.premio = 500;
}

...
```

LABORATORIO 2 (OPCIONAL)

1. Modifique el programa agenda del laboratorio anterior, para que cuando se trate de registrar amigas se almacene su teléfono y fecha de nacimiento y cuando sean amigos el teléfono de su oficina.

APUNTADORES A ESTRUCTURAS

Se pueden definir apuntadores a estructuras para poder referenciarlas indirectamente.

Desde un apuntador también se pueden accesar los miembros 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 miembros de una estructura por medio de un apuntador es un poco confusa:

```
EMPLEADO  *p;

...

(*p).tipo = 'D';
```

Debido a que son operaciones muy utilizadas, se utiliza el operador "->":

```
EMPLEADO  *p;

...

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

Ejemplo:

```
/*
        Programa 2

      Programa que implementa el manejo de números complejos
      con apuntadores a estructuras.

*/
#include <stdio.h>
#define                 SIGNO(x)      ((x) >= 0) ? '+' : ' '

typedef struct{
      float    real,
               imaginaria;
} COMPLEJO;
COMPLEJO  suma(COMPLEJO, COMPLEJO);
COMPLEJO  resta(COMPLEJO, COMPLEJO);

main() {
   COMPLEJO  a,b,*c;

   c = (COMPLEJO *)malloc(sizeof(COMPLEJO));
   printf("\n\nProporciona dos numeros complejos:\n");
   scanf("%f",&a.real);
   scanf("%f",&a.imaginaria);
   scanf("%f",&b.real);
   scanf("%f",&b.imaginaria);
   *c = suma(a,b);
   printf("\nLa suma de los numeros:"
                        "\n%5.2f %c %5.2fi\n%5.2f %c %5.2fi\nes:\n\n"
                        "%5.2f %c %5.2fi\n",
                        a.real, SIGNO(a.imaginaria), a.imaginaria,
                        b.real, SIGNO(b.imaginaria), b.imaginaria,
                        c->real, SIGNO(c->imaginaria), c->imaginaria);
   *c = resta(a,b, c);
    printf("\n\ny la resta:\n\n%5.2f %c %5.2fi\n",
                        c->real, SIGNO(c->imaginaria), c->imaginaria);
}
```

```
COMPLEJO  suma(COMPLEJO  x, COMPLEJO y) {
        COMPLEJO   z;

        z.real = x.real + y.real;
        z.imaginaria = x.imaginaria + y.imaginaria;
        return z;
}

COMPLEJO  resta(COMPLEJO  x, COMPLEJO y) {
        COMPLEJO   z;

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

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

Printf

Para tener control del formato que se da a la salida, se puede utilizar la instrucción.

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

La cadena de formato contiene caracteres ordinarios, que son copiados 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 formato 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 especificación 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 una cadena de caracteres.

Scanf

La función scanf permite la lectura de variables.

Esta función recibe como parámetros una cadena de formato y una lista 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 separadas por coma.

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.

Manejo de archivos

Un archivo es accesible a través de una estructura definida como FILE en el archivo estándar de encabezamiento stdio.h.

Esta estructura contiene miembros que describen el estado actual del archivo.

Un archivo se considera como un flujo de caracteres que se procesa secuencialmente.

El sistema proporciona tres archivos estándar:

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

Para abrir un archivo se utiliza la función fopen:

```
fp = fopen("filename","w");
```

El primer parámetro es el nombre del archivo. Se puede especificar toda la ruta.

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

"r"	lectura
"w"	escritura
"a"	agregar

El modo "w" crea un archivo.

El valor de regreso de la función es una apuntador a FILE, mediante el cual se hacen las referencias posteriores al archivo.

El valor de regreso es NULL si existe algún error, el cuál se puede deber a las siguientes causas:

- El archivo que se quiere para escritura no tiene permisos de escritura para quien ejecuta el programa.

- 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 para el archivo que se intenta leer.

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

```
fclose(fp);
```

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

Escritura/lectura de archivos

Las funciones fprintf y fscanf se utilizan para escribir y leer de un archivo respectivamente.

La forma de 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,.. = lista 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 cómo 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 1

        Programa que demuestra el manejo de archivos

*/
#include <stdio.h>

main(int argc, char **argv) {

        FILE        *fileRead,
                    *fileWrite;

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

Funciones de entrada/salida de 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 y se colocan en s. Si se lee '\n', este se coloca en s. La cadena s se termina con '\0'. Regresa 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 de fin de archivo.

int puts(char *s)

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

Ejemplo:

```
/*
        Programa 2

        Programa que demuestra el manejo de archivos

*/
#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",
                        argv[0], argv[1]);
                exit(1);
        }
        if ((fileWrite = fopen(argv[2],"w")) == NULL) {
                fprintf(stderr,"%s: error al abrir el archivo %s",
                        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.