CAPITULO 3: OPERADORES.

1. OPERADORES ARITMETICOS.

Los operadores aritméticos comprenden las operaciones básicas: la suma (+), la resta (-), la multiplicación (*), la división (/) y el operador módulo (%).

El operador módulo: **%** se utiliza para calcular el resto del cociente entre dos enteros, y no puede ser aplicado a variables del tipo float ó double.

Se puede observar que no existen operadores de potencia, radicales, logaritmos, etc. En C todas estas operaciones (y muchas otras) se realizan por medio de llamadas a funciones de librería.

2. OPERADORES RELACIONALES Y LÓGICOS.

Operadores relacionales

Todas las operaciones relacionales dan sólo dos posibles resultados: **falso** ó **verdadero**. En C, **falso** queda representado por cualquier expresión cuyo valor es nulo (cero) y **verdadero** por cualquier número distinto de cero. No hay que confundir el operador relacional igual que (==) con el de operador de asignación igual a (=).

Símbolo	Descripción		Símbolo	Descripción		Símbolo	Descripción
<	Menor que		< =	Menor o igual que		= =	Igual que
>	Mayor que		>=	Mayor o igual que		! =	Distinto que

```
int i=1, j=2, k=3, n;

n=(i<j);                    resultado verdadero: n=1.
n= ( (i+j)>=k+5 ) ;         resultado falso: n=0.
```

Operadores lógicos:

Además de los operadores relacionales, C posee 3 operadores lógicos. El lenguaje interpreta como **verdadero** todo valor distinto de cero, y **falso** cuando es igual a cero.

En las proposiciones lógicas se pueden emplear los operadores aritméticos y relacionales.

Operador	&&	||	!
Descripción	AND (y lógica)	OR (o lógica)	NOT (negación)

El resultado de una operación **AND** es verdadera sólo sí los dos operando son verdaderos.

El resultado de una operación **OR** es verdadera si uno de los operando es verdadero y falso si los dos operandos son falsos.

```
(4>5) || (1==1);        verdadero; el valor de la expresión es 1.
4*5 -3;                 verdadero.
!(3>5);                 verdadero.
4*5 - 20;               falso; el valor de la expresión es 0.
(1==1) && (1>2);        falso.
!(5>3);                 falso.
```

3. OPERADORES DE ASIGNACION.

Sintaxis: identificador = expresión;

La asignación más usada es el (=) , donde el identificador representa generalmente una variable y expresión puede ser una constante, una variable o una expresión más compleja. El identificador se conoce como **lvalue** y la expresión por **rvalue**. Una asignación es simplemente la copia del resultado de una expresión **rvalue** sobre otra **lvalue,** por lo tanto, **lvalue** debe tener lugar es decir poseer una posición de memoria.

```
a =17;          La segunda sentencia no es correcta ya que 17
17 = a;         no posee una ubicación de memoria.
```

Asignación múltiple.

En C, se permite la asignación múltiple, y como es una operación que se evalúa de derecha a izquierda , los paréntesis son superfluos.

```
a=(b=(c=5));
a=b=c=5;                Dos expresiones equivalentes.
```

Asignación condicional

Sintaxis: Identificador = (Expresion_1) ? (expresion _2) : (expresion_3) ;

La expresión **Expresion_1** que es tipo relacional con un resultado **cierto** ó **falso** se evalúa primero. Si el resultado es **Cierto**, se asignará al **Identificador** el valor de **expresion_1** y si el resultado es **falso**, se le asignará al **Identificador** valor de **expresion_3** .

```
menor =(a<b)? a:b ;         Se elige el menor de a y b.
valor =(a>0)? a=a+1:a=a-1;  Si a es positivo, se incrementa sino se
                            decrementa.
```

Otros operadores de asignación.

C posee otros operadores de asignación que se combinan con los operadores aritméticos para abreviar las operaciones. Así una operación como a = a + b, se puede abreviar como: a += b.

No se admite un espacio en blanco entre los símbolos (+=). Ésta asignación se extiende a todos los operadores aritméticos.

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

`int i=5, j=7;` `float f=5.5, g = -3.25;`	
`f -= g;`	`f = f-g;  f = 8.75`
`j *= (i-3);`	`j = j*(i-3); j=13`

4. EL OPERADOR DE MOLDE Y CONVERSIÓN.

Conversión automática.

Cuando dos ó mas tipos de variables distintas se encuentran dentro de una misma operación ó expresión matemática, ocurre una conversión automática del tipo de las variables. Se realizan las siguientes reglas de conversión.

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

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

`long p;`
`unsigned char q;`
`int i;`
 	
`p + q*i;`
	`Las conversiones son las siguientes:`
	`1. q se convierte en unsigned int.`
	`2. i se convierte en unsigned int.`
	`3. q*i es de tipo unsigned int,`
	`pero se convierte en long`
	`4. la suma p + q*i es long.`

Conversión por asignación.

En una asignación, **lvalue** = **rvalue**;

El calculo de **rvalue** se hace de acuerdo con las reglas de conversión automática, pero luego se ajusta al tipo de **lvalue**. Además:

- Un valor un coma flotante puede truncarse si se asigna a un entero; (perdida de la parte fraccionaria).
- Un valor de doble precisión puede ser redondeado si se asigna a un identificador de simple precisión.
- un entero puede ser alterado si se asigna a otro entero más corto. (perdida de los bits más significativos.)

Operador de moldeado (casting).

El lenguaje permite cambiar el tipo de una expresión imponiendo el tipo de variable al resultado de una operación mediante el operador de molde :

(tipo de dato) operando, donde tipo puede ser (**char, int, float, double, long, unsigned,** etc.). El casting no cambia el tipo de dato de las variables o expresiones, sino cambia el contenido a tomar una forma determinada.

`double x, y, z, a=2.25;`
`int i=3;`
 	
`y = a*i;`
`x=(int)(a*i);`
`z=(int)a*i;`
	`1. y es de tipo double; su valor es 6.75;`
	`2. El valor de (int)(a*i) es de tipo int;`
	`su valor es 6 truncando la parte fraccionaria,`
	`pero x es de tipo double con valor 6.0 .`
	`3. (int)a trunca la parte fraccionaria de a`
	`y su valor es 2, (int)a*i = 6,`
	`y el valor de z es 6.0`

5. OPERADORES DE INCREMENTO Y DECREMENTO.

Un operación muy común es incrementar y decrementar una variable, para ello, C proporciona dos operadores unitarios `++` y `--` que incrementan o decrementan en una unidad el valor del operando. Para visualizar la función de los operadores antedichos , digamos que

las sentencias:

n = n + 1; es equivalente a **n++** o **++n ;**
m = m - 1; es equivalente a **m--** o **--m;**

Estos operadores suelen emplearse con variables de tipo **int,** aunque se pueden usar sin problemas con cualquier otro tipo de variable .

La colocación de los operadores como **prefijo** ó **sufijo** son equivalente y tienen el mismo efecto, excepto cuando haya otros operadores; en éste caso, la posición de los operadores (**++, --**) indica el momento de la operación de incremento o decremento con respecto de la otra operación. Veamos el siguiente ejemplo :

`m = 4;  n = ++m;` `m = 4;  n = m++;`	1. Se incrementa m (m = 5) y se asigna su valor a n; (n = m = 5). 2. Se asigna m a n y luego se incrementa, (n = 4 y m = 5). En ambos casos el valor de m es 5, pero el valor de n es distinto.

6. LOS OPERADORES DE PUNTERO (*, &)

Los operadores ***** y **&** se utilizan con los punteros. No se pueden confundir con los operadores de multiplicación y la operación AND ya que se son unitarios y afectan solo a un operando.

El operador **&** permite obtener la dirección de una variable y el operador ***** es el complementario del primero; pues al evaluarse devuelve el contenido de una dirección , por lo tanto el operando tiene que ser un puntero. El operador ***** sirve también para declarar una variable de tipo puntero a un tipo de dato determinado.

`int m=1, n=2,z;`
`int *puntero;`
 	
`puntero = &m;`
`*puntero = 4;`
`puntero = &n;`
`z= *puntero;`
	1. Declaración de 3 variables enteras y un puntero (puntero) a objetos enteros.
	
	2. Se asigna a puntero la dirección de m.
	3. Se asigna 4 a la dirección donde apunta puntero; (m = 4)
	4. Se asigna a puntero la dirección de n.
	5. Se asigna a z el contenido de la dirección apuntada por puntero; (z = 2);

7. OPERADORES [], ( ) y (,).

Los corchetes [].

Los corchetes se utilizan para declarar tablas (array, vector , matrices).

Sintaxis: tipo identificador[n]; donde **n** indica el tamaño de la tabla y los elementos se referencian mediante un índice que empieza desde **0** hasta **n-1**. El operador nos permite declarar tablas múltidimensionales.

En el caso de cadenas de caracteres, el tamaño del array correspondiente se vera aumentado

de uno, pues el compilador añade el carácter **NULL** ('**\0**') al final de la cadena.

```
int tabla[4];
int tabla2[3][3];
char c1, c2;
char texto[]="hola";

tabla[0]=1;
tabla[3]=4;
tabla2[0][0]=1; tabla2[2][2]=9;

c1 = texto[0];
c2 = texto[4];
```

1. Declaración de un array "tabla" de 4 enteros, una matriz "tabla2" de 3X3 enteros y una cadena de caracteres con un tamaño libre.
2. tabla[0] es el primer elemento y tabla[3] es el último del tabla
3. tabla2[0][0] es el primer elemento de y tabla2[2][2] es el último.
4. c1 contiene el primer elemento de texto; c1='h'.
5. c2 contiene el último elemento de texto; c2='\0'.

Los paréntesis ().

Los paréntesis se emplean dentro de las expresiones para modificar el orden de evaluación predeterminado por el lenguaje. (Todos los operadores están ordenados según una precedencia predeterminada), así pues los paréntesis alteran este orden.

También por motivo de claridad es recomendable utilizar los paréntesis e incluir espacios para separar los operadores.

```
x + 2*z;

(x + 2)*z;
```

1. La expresión se evaluará según las precedencias predeterminas como la formula x + (2*z).
2. En la segunda, hemos cambiado el orden de su evaluación.

Operador secuencial (,).

La coma se utiliza para concatenar varias expresiones en una sentencia. El lado izquierdo de la coma siempre se evalúa primero.

```
m=(n=20, n++, n*2);

m=f1(), f2(), f3();
```

1. Se asigna 20 a n, incrementa n y multiplica por 2 y luego asigna;m=42.
2. Invoca a la función f1(), luego a f3(), y después a f3(). Por último asigna a m el valor que devuelve f3().

8. OPERADOR (sizeof)

El operador **sizeof()** es unitario y devuelve la longitud en bytes del operando que puede ser una variable, un tipo de dato como **int** o **chr,** etc, o otro tipo de datos más complejos.

sizeof(m) devuelve 1 si m es de tipo **char**, un 2 si es de tipo **int,** 4 si es un **float**,

En muchos programas es necesario conocer el tamaño que ocupa una variable para fines de reservación dinámica de memoria. Como vimos, este tamaño depende del compilador que se use, lo que producirá, si definimos rigidamente (o sea, con un número dado de bytes) el espacio requerido, un problema serio si luego se quiere compilar el programa con otro distinto del original. Para mantener la portabilidad, es conveniente que cada vez que haya que referirse al **tamaño** en bytes de las variables, hay que hacerlo mediante el operador **sizeof** que calcula sus requerimientos.

9. OPERADOR (typedef)

Con la palabra reservada **typedef,** se asignan nuevos nombres a los tipos de datos existentes; de esta forma se pueden crear nombres para datos más complejos y abreviaturas para declaraciones largas.

Sintaxis; typedef tipo nombre; donde tipo es cualquier tipo de dato válido en C, y nombre es el nuevo nombre de ese tipo. El nuevo nombre nos sirve para declarar variables o funciones, pero no reemplaza al anterior, pues se puede nombrar de ambas formas.

```
typedef unsigned long int Contador;
typedef long double Enorme;
/* Al tipo unsigned long int, le cambiamos de nombre (Contador)y
el tipo long double le cambiamos de nombre como (Enorme) */

Contador n, m;
Enorme x;      /* Declaración de las variables n, m del tipo
Contador y x del tipo Enorme */
```

10. OPERADORES DE MANEJO DE BITS

Estos operadores muestran una de las características más potentes del lenguaje C, la de poder manipular internamente las variables (es decir manipular los **bits** de las variables). Estos operadores sólo se aplican a variables del tipo entero; **char , short , int , long** y **unsigned,** pero no pueden ser usados con los reales (**float ó double**).

Estos operadores comprenden los desplazamientos hacia la izquierda o la derecha, las operaciones lógicas AND, OR, XOR y un operador de complemento a 1.

Las operaciones lógicas vistas anteriormente tratan el valor de las variables o de las expresiones y no los bits de las variables.

Tabla de los operadores de manejo de bits

Operador	**&**	|	^	~	>>	<<
Descripción	AND	OR	XOR	complemento a 1	Desplazamiento D	Desplazamiento I

Tabla de las operaciones lógicas sobre los bits

a	b	a & b	a | b	a ^ b	~a
1	1	1	1	0	0
1	0	0	1	1	0
0	1	0	1	1	1
0	0	0	0	0	1

```
m = 0X6db7; n = 0Xa726;
k = m&n;
m=0110 1101 1011 0111
n=1010 0111 0010 0110
k=0010 0101 0010 0110; k=0x2526

k = m^n;
m=0110 1101 1011 0111
n=1010 0111 0010 0110
k=1100 1010 1001 0001; k=0xca91
```

```
k = m|n;
m=0110 1101 1011 0111
n=1010 0111 0010 0110
k=1110 1111 1011 0111; k=0xefb7

k = ~m ;
m=0110 1101 1011 0111
                     .
k=1001 0010 0100 1000; k=0x9248
```

Veamos las operaciones de desplazamiento; la sentencia **c = a << b;** implica asignarle a **c**, el valor de **a** con sus bits corridos a la izquierda en **b** lugares, los bits que van "saliendo" por la izquierda , se desechan y los bits que van quedando libres a la derecha se completan con cero . Se procede de la misma manera a la derecha **>>,** teniendo en cuenta que C considera dos tipos de desplazamientos; lógicos y aritméticos dependiendo del tipo de dato. (con o sin signo).

Los desplazamientos lógicos se realizan sobre datos **unsigned** y no tiene ninguna consideración sobre el signo (Si se desplaza una variable de tipo **unsigned char** de 8 veces, el resultado es 0).

Los desplazamientos aritméticos se realizan sobre tipos enteros con signo y mantienen el signo de las variables (el bit más alto). Con los desplazamientos a la izquierda, el bit no se altera, pero con los desplazamientos a la derecha, se copia el bit más significativo en todas las posiciones libres.

```
unsigned char n =1;
signed char m = -1;

n << 1;
n << 4;
n >> 1;
m << 2;
m >> 4;
```

```
n = 0000 0001;
m = 1000 0001;

n = 0000 0010; Desplazamiento lógico.
n = 0010 0000;
n = 0001 0000;
m = 1000 0100; Desplazamiento aritmético.
m = 1111 1000;
```

Enmascaramiento.

El enmascaramiento son operaciones sobre los bits de una variable para cambiar de patrón, extraer información, etc. El operador **|** se utiliza para activar bits (poner unos) y el operador **&** se usa para la desactivación de los bits (poner cero) y extraer información empleando una máscara.
Vemos el siguiente ejemplo para ver como funciona el proceso

`mascara1 = 0xf0; mascara2 = 0xff;` `mascara3 = 0x01; mascara4 = 0xe0;`	
`x = x & mascara1;`	1. pone ceros en los 4 primeros bits.
`x = x | mascara2;`	2. pone unos en los 8 bits.
`y = x & mascara3;`	3. extrae el contenido del primer bit.
`y = x & mascara4;`	4. extrae el contenido de los 3 primeros bits.

11. PRECEDENCIA DE LOS OPERADORES.

Tabla de precedencia de los operadores ordenada de mayor a menor

Operadores	Asociativid
()[] -> .	I -> D
sizeof (tipo) ! ~ ++ -- * &	D -> I
* / %	I -> D
+ -	I -> D
>> <<	I -> D
>= > < <=	I -> D
== !=	I -> D
&	I -> D
^	I -> D
|	I -> D
&&	I -> D
||	I -> D
?:	D -> I
= += -= *= etc	D -> I
,	I -> D

Los operadores de C se agrupan jerárquicamente de acuerdo con su precedencia. En una expresión, las operaciones de mayor precedencia se efectuarán antes que las de menor precedencia. Entre los operadores aritméticos, (*, /, %) se encuentran dentro del mismo grupo de precedencia y (**+, -)** se encuentran en otro. El primer grupo tiene mayor precedencia que el segundo, por lo tanto, las operaciones de multiplicación y división se efectuarán antes que la suma y la resta. Otra consideración es el orden en que se efectuarán las operaciones del mismo grupo. Esto se conoce como la asociatividad de los operadores (generalmente es de izquierda a derecha).

Sin embargo, se puede alterar el orden natural de evaluación mediante el uso de los paréntesis permitiendo que las operaciones se hagan en el orden en que se desee, ya que es el operador que tiene la mayor precedencia.

Nota: En el renglón de los operadores de precedencia cero hemos agregado ubicándolos a la derecha del mismo para diferenciarlos, tres operadores , **->** **y .** que serán analizados más adelante.

Consideramos el ejemplo siguiente: **a - b / c * d.** En primer lugar primero se efectúa la división (**b/c**) ya que tiene mayor precedencia, el cociente resultante se multiplica por **d** a causa de la asociatividad de izquierda a derecha, y finalmente se resta el producto a **a.**

Otro ejemplo: **m = ~n++.** Como los operadores son del mismo grupo cuya asociatividad es de D ->I, primero se incrementa **n** y luego se complementa a 1 y luego se asigna a m.