# Unidad N3: OPERADORES Y EXPRESIONES

## 3.1 Operadores y Expresiones

Una *expresión* se compone de uno o más **operandos** que se combinan entre sí mediante **operadores**. En la práctica una expresión es una secuencia de operaciones y operandos que especifica un cálculo y en consecuencia devuelve un resultado. La forma más simple de una **expresión** consta de una única constante o variable. Las expresiones más complicadas se forman a partir de un operador y uno o más operandos.

Cada expresión produce un **resultado**. En el caso de una expresión sin operador/es el resultado es el propio operando; por ejemplo, una constante o una variable.

El resultado de expresiones que implican operadores se determina aplicando cada operador a sus operandos. Los **operadores** son símbolos que expresan operaciones que se aplican a uno o varios operandos y que devuelven un valor.

## 3.2 Expresiones

Al igual que en otros lenguajes, C++ permite formar expresiones utilizando variables constantes y **operadores** **aritméticos**: **+** (suma), **–** (resta), **×** (multiplicación, **/** (división) y **%** (resto módulos). Estas expresiones se pueden utilizar en cualquier lugar que sea legal utilizar un valor del tipo resultante de la expresión.

Una **expresión** es una operación que produce un valor con la excepción de expresiones void. Casi todos los tipos de sentencias utilizan expresiones de una u otra manera.

**Obsérvese que las expresiones no son igual que las sentencias. Las sentencias indican al compilador que realice alguna tarea y termina con un punto y coma, mientras que las expresiones especifican un cálculo. En una sentencia puede haber varias expresiones.**

## 3.3 Operadores

El significado de un operador —operación que realiza y tipo de resultado— depende de los tipos de sus operandos. Hasta que no se conoce el tipo de operando/s, no se puede conocer el significado de la expresión.

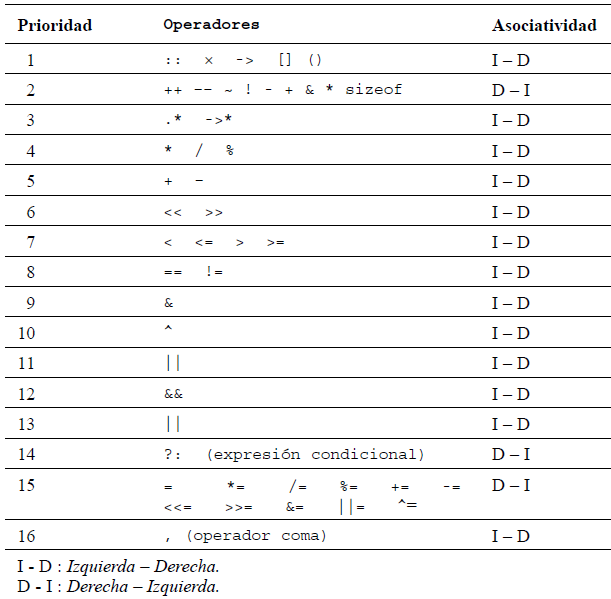
Los operadores se clasifican en: *unitarios* (*"unarios"*), binarios o ternarios. Operadores unitarios, tales como el operador de dirección (&) o el de *desreferencia* (\*) que actúan sobre un operando. Operadores binarios, tales como suma (+) y resta (–) que actúan sobre dos operadores. Operadores ternarios, tales como, el operador condicional (?:) que actúa sobre tres operandos. Algunos operadores pueden actuar como *"unitarios"* o como binarios; un ejemplo es el operador \* que puede actuar como el operador de multiplicación (binario) o como operador de *desreferencia* (indirección).

Los operadores imponen los requisitos sobre el tipo/s de su/s operando/s. Así, cuando un operador binario se aplica a operandos de tipos predefinidos o tipos compuestos, se requiere normalmente que sean del mismo tipo o tipos que se puedan convertir a un tipo común. Por ejemplo, se puede convertir un entero a un tipo de coma flotante y viceversa; sin embargo, no es posible convertir un tipo puntero a un tipo en coma flotante.

Los operadores se clasifican también según la posición del operador y de los operandos: *prefijo* (si va delante), *infijo* (si va en el interior) o *postfijo* (si va detrás). Otra propiedad importante de los operadores es la **aridad** (número de operandos sobre los que actúa):

* **Procedencia** (*prioridad de evaluación*). Indica la prioridad del operador respecto a otros a la hora de calcular el valor de una expresión.
* **Asociatividad**. Determina el orden en que se asocian los operandos del mismo tipo en ausencia de paréntesis.
* **Asociatividad por la derecha** (D-I). Si dos operandos que actúan sobre el mismo operando tienen la misma precedencia se aplica primero al operador que está más a la derecha (operadores primarios, terciarios y asignación).
* **Asociatividad por la izquierda** (I-D). En este caso se aplica primero al operador que está a mano izquierda (operadores binarios).

La prioridad o precedencia de operadores determina el orden en el que se aplican los operadores a un valor. Los operadores C++ vienen en una tabla con 16 grupos. Los operadores del grupo 1 tienen mayor prioridad que los del grupo 2, y así sucesivamente:

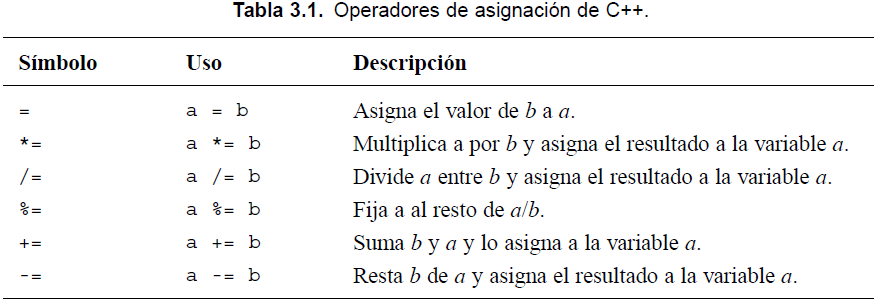


## 3.4 Operadores de asignación

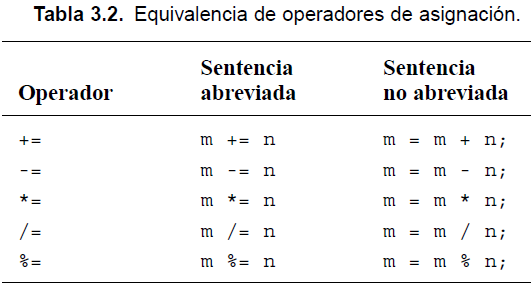
El operador de asignación = asigna el valor de la expresión derecha a la variable situada a su izquierda. expresión puede ser una variable, una constante o una expresión aritmética más complicada

variable **operador** = expresión

**Asignación compuesta**: Este operador es asociativo por la derecha, eso permite realizar asignaciones múltiples. Así, a = b = c = 45



**Precedencia**: Muchas expresiones implican más de un operador. En estos casos es necesario saber qué operando se aplica primero para obtener el valor final. C++ proporciona un operador abreviado de asignación (\*=), que realiza una asignación equivalente: i \*= 10; *equivale a* i = i \* 10;



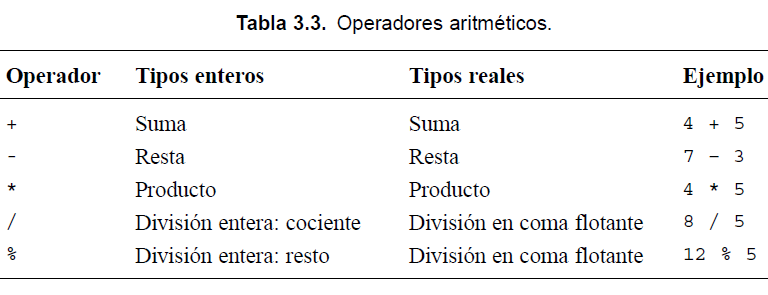
**Operadores de asignación compuesta**: La sintaxis general de un operador de asignación compuesta es a op= b; donde op= puede ser cualquiera de los siguientes diez operadores:

+= -= \*= /= %= //operadores aritméticos

<<= >>= &= ^= |= //operadores de bits

## 3.5 Operadores aritméticos

Los operadores aritméticos sirven para realizar operaciones aritméticas básicas. Los operadores aritméticos C++ siguen las reglas algebraicas típicas de jerarquía o prioridad. Estas reglas especifican la precedencia de las operaciones aritméticas.



## 3.6 Operadores de incremento y decremento

De las muchas características de C++ heredadas de C, una de las más útiles son los operadores de incremento ++ y decremento ––. Los operadores ++ y ––, denominados de *incrementación* y *decrementación*, suman o restan 1 a su argumento, respectivamente, cada vez que se aplican a una variable.

Si los operadores ++ y -- están de prefijos, la operación de incremento se efectúa antes que la operación de asignación; si los operadores ++ y -- están de sufijos, la asignación se efectúa en primer lugar y la incrementación o decrementación a continuación.

## 3.7 Operadores relacionales

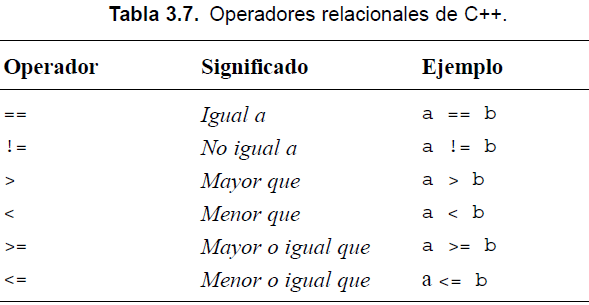
ANSI C++ soporta el tipo bool que tiene dos literales false y true. Una expresión booleana es, por consiguiente, una secuencia de operandos y operadores que se combinan para producir uno de los valores true y false.

C++ no tiene tipos de datos lógicos o booleanos, como Pascal, para representar los valores verdaderos (true) y falso (false). En su lugar se utiliza el tipo int para este propósito, con el valor entero 0 que representa a falso y distinto de cero a verdadero.

Los operadores relacionales se usan normalmente en sentencias de selección (if) o de iteración (while, for), que sirven para comprobar una condición. Utilizando operadores relacionales se realizan operaciones de igualdad, desigualdad y diferencias relativas. La Tabla 3.7 muestra los operadores relacionales que se pueden aplicar a operandos de cualquier tipo de dato estándar: char, int, float, double, etc.

Cuando se utilizan los operadores en una expresión, el operador relacional produce un 0, o un 1, dependiendo del resultado de la condición. 0 se devuelve para una condición *falsa*, y 1 se devuelve para una condición *verdadera*.

Un error típico, incluso entre programadores experimentales, es confundir el operador de asignación (=) con el operador de igualdad (==).



## 3.8 Operadores lógicos

Estos operadores se utilizan con expresiones para devolver un valor verdadero (cualquier entero distinto de cero) o un valor falso (0). Los operadores lógicos se denominan también *operadores booleanos*, en honor de George Boole, creador del álgebra de Boole.

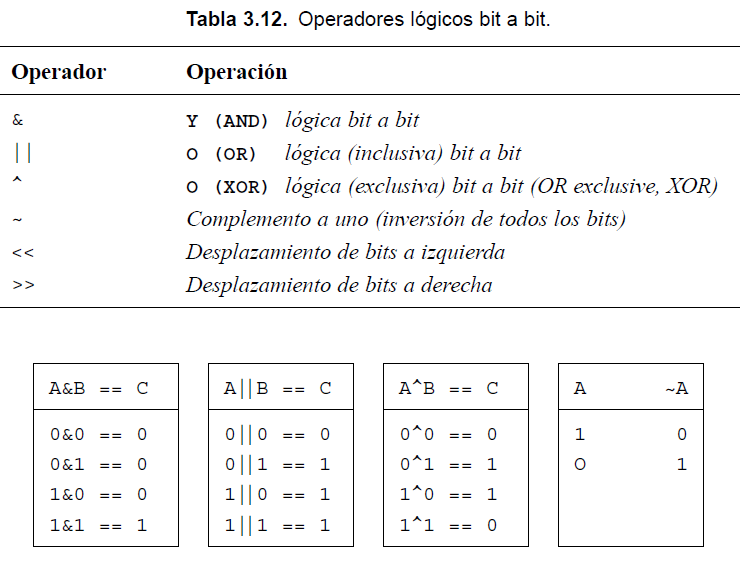
Los operadores lógicos de C++ son: **not (!), and (&&)** y **or (||)**. El operador lógico ! (**not***, no*) produce *falso* si su operando es *verdadero* (distinto de cero) y viceversa. El operador lógico **&&** (and*, y*) produce verdadero *sólo* si ambos operandos son *verdadero* (no cero); si cualquiera de los operandos es falso produce *falso*. El operador lógico **||** (**or**, *o*) produce verdadero si cualquiera de los operandos es verdadero (distinto de cero) y produce falso sólo si ambos operandos son falsos.

## 3.9 Operadores de manipulación de bits

Los operadores de manipulación o tratamiento de bits *(bitwise)* ejecutan operaciones lógicas sobre cada uno de los bits de los operandos. Estas operaciones son comparables en eficiencia y en velocidad a sus equivalentes en lenguaje ensamblador.

Cada operador de manipulación de bits realiza una operación lógica bit a bit sobre datos internos. Los operadores de manipulación de bits se aplican sólo a variables y constantes char, int y long, y no a datos en coma flotante. Dado que los números binarios constan de 1,s y 0,s (denominados *bits*), estos 1 y 0 se manipulan para producir el resultado deseado para cada uno de los operadores.

Las siguientes tablas de verdad describen las acciones que realizan los diversos operadores sobre los diversos patrones de bit de un dato int (char o long).



## 3.10 Operador condicional

El operador condicional, ?:, es un operador ternario que devuelve un resultado cuyo valor depende de la condición comprobada. Tiene asociatividad a derechas. Al ser un operador ternario requiere tres operandos. El operador *condicional* se utiliza para reemplazar a la sentencia if-else lógica en algunas situaciones.

## 3.11 Operador coma

El *operador coma* (,) de secuencia o evaluación permite combinar dos o más expresiones separadas por comas en una sola línea. Se evalúa primero la expresión de la izquierda y luego las restantes expresiones de izquierda a derecha.

## 3.12 Operadores especiales

## 3.12.1 Operador ()

El operador () es el operador de llamada a funciones. Sirve para encerrar los argumentos de una función, efectuar conversiones explícitas de tipo, indicar en el seno de una declaración que un identificador corresponde a una función, resolver los conflictos de prioridad entre operadores.

## 3.12.2 Operador []

Sirve para designar un elemento de un array. También se puede utilizar en unión con el operador **delete**; en este caso, indica el tamaño del array a destruir.

**3.12.3 Operador : :**

Este operador es específico de C++ y se denomina *operador de ámbito de resolución*, y permite especificar el alcance o ámbito de un objeto.

## 3.13 Operador SIFEOF

Con frecuencia, su programa necesita conocer el tamaño en bytes de un tipo de dato o variable. C++ proporciona el operador sizeof, que toma un argumento, bien un tipo de dato o bien el nombre de una variable (escalar, array, registro, etc.).

El operador sizeof se denomina también *operador en tiempo de compilación*, ya que, en tiempo de compilación, el compilador sustituye cada ocurrencia de sizeof en su programa con un valor entero sin signo (unsigned).

## 3.14 Conversiones de tipos

En C++ se puede convertir un valor de un tipo en un valor de otro tipo. Tal acción se denomina *conversión de tipos*. Esta característica es necesaria en numerosas expresiones aritméticas y asignaciones, debido a que las operadoras binarias requieren operandos del mismo tipo.

En la práctica se realizan dos tipos de conversiones *implícitas* (ejecutadas automáticamente, algunas de las citadas anteriormente), o *explícitas* (solicitadas especialmente por el programador C++).

* **Conversiones implícitas**: Cuando se intenta realizar operaciones con tipos de datos de diferente precisión, el compilador define automáticamente —sin intervención del programador— de modo que convierte los operandos a un tipo común antes de realizar cualquier operación. Es decir, necesitamos asignar un a un numero fraccionario. A esto se denomina implícito
* **Conversiones aritméticas:** aseguran que los operandos de un operador binario, tal como un operador aritmético o lógico, se convierten a un tipo común antes de que se evalúe el operador. Por ejemplo, si un operando es de tipo long double, entonces el otro operando se convierte a long double independientemente del tipo que contiene.
* **Conversiones explicitas:** Una *conversión de tipos* “*cast*” modelado de tipos (*moldes*) es una conversión explícita de tipos. C++ ofrece diferentes formas de modelar una expresión a un tipo diferente.

El compilador automáticamente realizará las conversiones de acuerdo a ciertas reglas. Para proteger la información, normalmente, sólo se permiten conversiones de menor tamaño (menos ricas) a mayor tamaño (más ricas) y no se pierde información; esta operación se llama promoción; por ejemplo, de char a int (no se pierde información). En otros casos se pierde información, como la conversión de float a int.

Si el compilador no puede efectuar la conversión por dudas en la misma, el programador puede hacerla explícita.