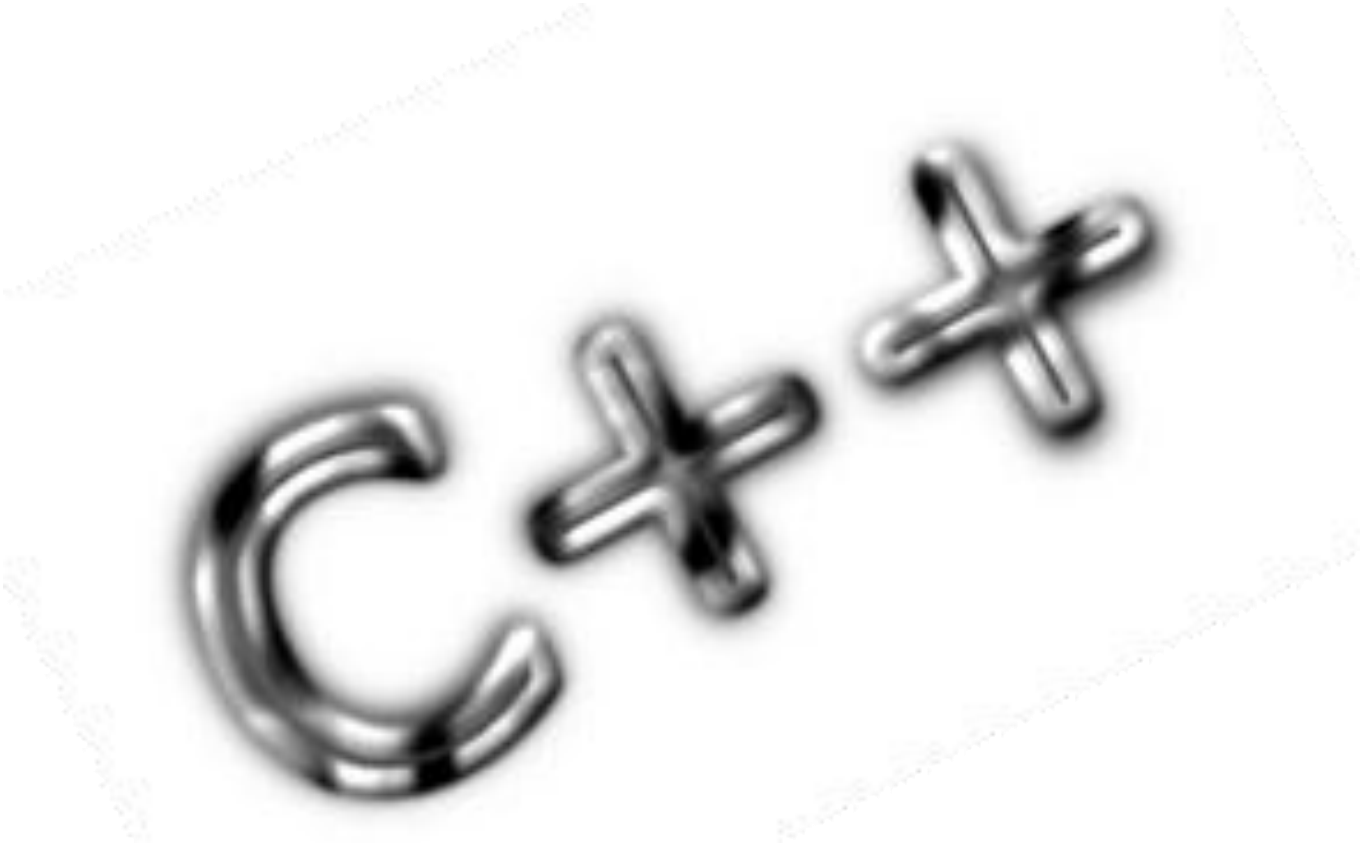



# OPERADORES EN C++

## TRANSVERSAL DE PROGRAMACIÓN BÁSICA

### INGENIERÍA DE SISTEMAS

En el presente documente se explica el uso de operadores en el lenguaje de c++. Así mismo se presente información correspondiente a la precedencia que tiene estos, para ser interpretados por la máquina y las nociones básicas que se deben tener a la hora de programar en este lenguaje

CRISTIAN GUILLERMO GARCÍA
MONITORIA 2012-3 | Universidad Distrital francisco José de Caldas

# OPERADORES EN C++

## Tabla de contenido


1. INTRODUCCIÓN ..... 2
2. ¿Qué es un OPERADOR? ..... 2
   2.1. OPERADORES DE ASIGNACIÓN ..... 2
   2.2. OPERADORES ARITMÉTICOS ..... 3
   2.3. OPERADORES RELACIONALES ..... 5
   2.4. OPERADORES LÓGICOS ..... 6
      2.4.1. AND lógico ..... 6
      2.4.2. OR Lógico ..... 7
      2.4.3. Negación Lógica ..... 7
   2.5. OPERADORES DE BITS [2] ..... 8
      2.5.1. Complemento ..... 8
      2.5.2. Desplazamiento A Izquierda ..... 10
      2.5.3. Desplazamiento A Derecha ..... 10
      2.5.4. AND ..... 11
      2.5.5. XOR ..... 11
      2.5.6. OR ..... 12
   2.6. OPERADORES DE DIRECCIÓN ..... 12
   2.7. OTROS OPERADORES ..... 13
      2.7.1. OPERADOR CONDICIONAL ..... 14
3. PRECEDENCIA DE OPERADORES ..... 14
      3.1. PRECEDENCIA ..... 15
      3.2. ASOCIATIVIDAD ..... 16
4. REFERENCIAS ..... 19

# OPERADORES EN C++

## 1. INTRODUCCIÓN

El presente documento pretende servir de guía para la enseñanza en la programación bajo el lenguaje C++. Pese a que se ha desarrollado siguiendo los lineamientos establecidos en el syllabus de la asignatura transversal de programación básica de la universidad distrital Francisco José de Caldas, es posible utilizarla como un manual en cualquier campo o ámbito siempre y cuando se relación con el aprendizaje del lenguaje mencionado. Adicionalmente, se recomienda que en caso de ser estudiante, se cuente con algunos conceptos básicos sobre el lenguaje o haber revisado el material de introducción C++.

## 2. ¿Qué es un OPERADOR?

"Un operador es un elemento de programa que se aplica a uno o varios operandos en una expresión o instrucción. Los operadores que requieren un operando, como el operador de incremento se conocen como operadores unarios. Los operadores que requieren dos operandos, como los operadores aritméticos (+,-,*,/) se conocen como operadores binarios. Un operador, el operador condicional (?:), utiliza tres operandos y es el único operador ternario de C++" [1].

Existen 6 tipos de operadores según su función, que son aritméticos, relacionales, de asignación, lógicos, de dirección y de manejo de Bits.

### 2.1. OPERADORES DE ASIGNACIÓN

"Tal y como su nombre lo indica, este tipo de operadores permiten la asignación de un valor especifico a una variable. En C++ se encuentran disponibles los siguientes operadores:

| Operador | Acción | Ejemplo | Resultado |
|---|---|---|---|
| = | Asignación Básica | X = 6 | X vale 6 |
| *= | Asigna Producto | X *= 5 | X vale 30 |
| /= | Asigna División | X /= 2 | X vale 3 |
| += | Asigna Suma | X += 4 | X vale 10 |
| -= | Asigna Resta | X -= 1 | X vale 5 |
| %= | Asigna Modulo | X %= 5 | X vale 1 |
| <<= | Asigna Desplazamiento Izquierda | X <<= 1 | X vale 12 |
| >>= | Asigna Desplazamiento Derecha | X >>= 1 | X vale 3 |
| &= | Asigna AND entre Bits | X &= 1 | X vale 0 |
| ^= | Asigna XOR entre Bits | X ^= 1 | X vale 7 |
| \|= | Asigna OR entre Bits | X \|= 1 | X vale 7 |

# OPERADORES EN C++

Todos ellos son operadores binarios, de los cuales, " = " es el único de asignación simple, los demás son operadores de asignación compuestos, puesto que están conformados por más de un símbolo, por ejemplo "+=" se compone del operador "+ " y el operador " = ".

Los seis primeros aceptan operandos de distinto tipo, mientras que los cinco últimos: <<=, >>=, &=, ^= y |=, implican manejo de bits, por lo que sus operandos deben ser numero int en sus distintas variantes. El funcionamiento de estos operadores se encuentra descrito en la sección de operadores de bits.

La parte izquierda (que tiene que ser una variable no constante) adquiere el valor señalado en la expresión de la derecha, pero se mantiene el tipo original de la variable de la parte izquierda. En caso necesario se realiza una conversión de tipo (con pérdida de precisión en su caso) del izquierdo al derecho.

Es necesario resaltar que el operador C++ de asignación simple (=) se distingue de otros lenguajes como Pascal que utilizan el símbolo := para este operador. Observe también que la asignación simple (=) utiliza un símbolo distinto del operador relacional de igualdad (==)" [2]. Además, en los operadores compuesto no debe haber espacios de la forma "+ =" y que el igual siempre va a la derecha del resto de operandos.

## 2.2. OPERADORES ARITMÉTICOS

Los operadores aritméticos se usan para realizar cálculos y operaciones con números reales y punteros [2]. Básicamente permiten hacer cualquier operación aritmética que se necesite. Los operadores más comunes son [3]:

| Operador | Acción | Ejemplo | Resultado |
|---|---|---|---|
| - | Resta | X = 5 - 3 | X vale 2 |
| + | Suma | X = 5 + 3 | X vale 8 |
| * | Multiplicación | X = 2 * 3 | X vale 6 |
| / | División | X = 6 / 3 | X vale 2 |
| % | Módulo | X = 5 % 2 | X vale 1 |
| -- | Decremento | X = 1; X-- | X vale 0 |
| ++ | Incremento | X = 1; X++ | X vale 2 |

La operación modulo corresponde a obtener el residuo de la división, de modo que al dividir 5 entre 2 tendremos como resultado 2 y como residuo 1, por tanto 5 % 2 corresponde al 1 que sobra de la división exacta.

# OPERADORES EN C++

Es necesario hacer una observación acerca de los operadores de incremento y decremento, ya que dependiendo de su ubicación con respecto a la variable, se tienen acciones diferentes. Si el operador precede a la variable, se conoce como pre-incremento o pre-decremento y se dice que el operador está en su forma prefija. Por el contrario, si el operador es posterior a la variable se encuentra en la forma posfija y se le llama pos-incremento o pos-decremento según el caso.

"Cuando un operador de incremento o decremento precede a su variable, se llevará a cabo la operación de incremento o de decremento antes de utilizar el valor del operando", tal y como se muestra en el siguiente ejemplo:

```
int x,y;
x = 2004;
y = ++x;
/* x e y valen 2005. */
```

En el caso de los post-incrementos y post-decrementos pasa lo contrario: se utilizará el valor actual del operando y luego se efectuará la operación de incremento o decremento" [3].

```
int x,y
x = 2004;
y = x++;
/* y vale 2004 y x vale 2005 */
```

Tal y como se presentó anteriormente, también existen operadores para los punteros, sin embargo estos solo aplican para aquellos apuntadores a matrices, arreglos o listas de elementos, y aunque se muestran a continuación, se explicaran de una mejor forma en el material de apuntadores o punteros en c++ [2].

| Operador | Acción | Ejemplo |
|---|---|---|
| - | Desplazamiento descendente | pt1 - n |
| + | Desplazamiento ascendente | pt1 + n |
| - | Distancia entre elementos | pt1 - pt2 |
| -- | Desplazamiento descendente de 1 elemento | pt1-- |
| ++ | Desplazamiento ascendente de 1 elemento | pt1++ |

# OPERADORES EN C++

## 2.3. OPERADORES RELACIONALES

"Los operadores relacionales, también denominados operadores binarios lógicos y de comparación, se utilizan para comprobar la veracidad o falsedad de determinadas propuestas de relación (en realidad se trata respuestas a preguntas). Las expresiones que los contienen se denominan expresiones relacionales. Aceptan diversos tipos de argumentos, y el resultado, que es la respuesta a la pregunta, es siempre del tipo cierto/falso, es decir, producen un resultado booleano.

Si la propuesta es cierta, el resultado es ***true*** (un valor distinto de cero), si es falsa será ***false*** (cero). C++ dispone de los siguientes operadores relacionales:

| Operador | Relación | Ejemplo | Resultado |
|---|---|---|---|
| < | Menor | X = 5; Y = 3;<br>if(x < y) x+1; | X vale 5<br>Y vale 3 |
| > | Mayor | X = 5; Y = 3;<br>if(x > y) x+1; | X vale 6<br>Y vale 3 |
| <= | Menor o igual | X = 2; Y = 3;<br>if(x <= y) x+1; | X vale 3<br>Y vale 3 |
| >= | Mayor o igual | X = 5; Y = 3;<br>if(x >= y) x+1; | X vale 6<br>Y vale 3 |
| == | Igual | X = 5; Y = 5;<br>if(x == y) x+1; | X vale 6<br>Y vale 5 |
| ¡= | Diferente | X = 5; Y = 3;<br>if(x ¡= y) y+1; | X vale 5<br>Y vale 4 |

Como puede verse, todos ellos son operadores binarios (utilizan dos operandos), de los cuales, dos de ellos son de igualdad: == y !=, y sirven para verificar la igualdad o desigualdad entre valores aritméticos o punteros. Estos dos operadores pueden comparar ciertos tipos de punteros, mientras que el resto de los operadores relacionales no pueden utilizarse con ellos.

Cualquiera que sea el tipo de los operandos, por definición, un operador relacional, produce un **bool** (**true** o **false**) como resultado, aunque en determinadas circunstancias puede producirse una conversión automática de tipo a valores **int** (**1** si la expresión es cierta y **0** si es falsa).

En las expresiones relacionales E1 <operador> E2, los operandos deben cumplir alguna de las condiciones siguientes:

- E1 y E2 son tipos aritméticos.
- E1 y E2 son punteros a versiones cualificadas o no cualificadas de tipos compatibles.

- Uno de ellos es un puntero a un objeto, mientras que el otro es un puntero a una versión cualificada o no cualificada de void" [2].
- Uno de los dos es un puntero, mientras que el otro es un puntero nulo constante.

## 2.4. OPERADORES LÓGICOS

"Los operadores lógicos producen un resultado booleano, y sus operandos son también valores lógicos o asimilables a ellos (los valores numéricos son asimilados a cierto o falso según su valor sea cero o distinto de cero). Por el contrario, las operaciones entre bits producen valores arbitrarios.

Los operadores lógicos son tres, dos de ellos son binarios y el último (negación) es unario:

| Operador | Acción | Ejemplo | Resultado |
|---|---|---|---|
| && | AND Lógico | A && B | Si ambos son verdaderos se obtiene verdadero(true) |
| \|\| | OR Lógico | A \|\| B | Verdadero si alguno es verdader |
| ! | Negación Lógica | !A | Negación de a |

### 2.4.1. AND lógico

Devuelve un valor lógico true si ambos operandos son ciertos. En caso contrario el resultado es false. La operatoria es como sigue: El primer operando (de la izquierda) es convertido a bool. Para ello, si es una expresión, se evalúa para obtener el resultado (esta computación puede tener ciertos efectos laterales). A continuación, el valor obtenido es convertido a bool cierto/falso siguiendo las reglas de conversión estándar. Si el resultado es false, el proceso se detiene y este es el resultado, sin que en este caso sea necesario evaluar la expresión de la derecha (recuérdese que en el diseño de C++ prima la velocidad).

Si el resultado del operando izquierdo es cierto, se continúa con la evaluación de la expresión de la derecha, que también es convertida a bool. Si el nuevo resultado es true, entonces el resultado del operador es true. En caso contrario el resultado es false.

```
int m[3] = {0,1,2};
int x = 0;
if (m && x) cout << "Cierto.";
else cout << "Falso.";
```

El valor m, que es interpretado como un puntero al primer elemento de la matriz, es transformado a un bool. Como es distinto de cero (no es un puntero nulo) el resultado es cierto. A continuación, el valor x es convertido también a bool. En este caso la conversión produce falso, con lo que este es el resultado del paréntesis de la sentencia if" [2].

### 2.4.2. OR Lógico

Este operador binario devuelve true si alguno de los operandos es cierto. En caso contrario devuelve false. Este operador sigue un funcionamiento análogo al anterior. El primer operando (izquierdo) es convertido a bool. Para ello, si es una expresión, se evalúa para obtener el resultado (esta computación puede tener ciertos efectos laterales). A continuación el valor obtenido es convertido a bool cierto/falso siguiendo las reglas de conversión estándar. Si el resultado es true, el proceso se detiene y este es el resultado, sin que en este caso sea necesario evaluar la expresión de la derecha (recuérdese que en el diseño de C++ prima la velocidad).

Si el resultado del operando izquierdo es false, se continúa con la evaluación de la expresión de la derecha, que también es convertida a bool. Si el nuevo resultado es true, entonces el resultado del operador es true. En caso contrario el resultado es false.

### 2.4.3. Negación Lógica

Este operador es denominado también No lógico y se representa en el texto escrito por la palabra inglesa NOT (otros lenguajes utilizan directamente esta palabra para representar el operador en el código). El operando (que puede ser una expresión que se evalúa a un resultado) es convertido a tipo bool, con lo que solo puede ser uno de los valores cierto/falso. A continuación el operador cambia su valor; Si es cierto es convertido a falso y viceversa.

Resulta por tanto, que el resultado de este operador es siempre un tipo bool, aunque al existir una conversión estándar por la que un cero es convertido a false, y cualquier valor distinto de cero a true, coloquialmente se dice que este operador convierte un operando 0 en 1 y uno no-cero en 0. En otras palabras: este operador devuelve cierto (true) si la expresión se evalúa a distinto de cero, en caso contrario devuelve falso (false).

Si "E" es una expresión, "!E" es equivalente a "(0 == E)". Como consecuencia, las expresiones que siguen son equivalentes dos a dos:

```
if (! valid);
if (valid == 0);
...
if (valid);
if (valid != 0);
```

# OPERADORES EN C++

## 2.5. OPERADORES DE BITS [2]

“Los operadores de movimiento son operadores a nivel de bits, y lo que hacen es convertir una determinada cantidad a su equivalente en bits para posteriormente realizar un desplazamiento de dicho valor. Estos operadores son:

| Operador | Acción | Ejemplo | Resultado |
|---|---|---|---|
| << | Desplazamiento a Izquierda | a << b | X vale 2 |
| >> | Desplazamiento a Derecha | X = 5 + 3 | X vale 8 |
| ~ | Complemento | X = 2 * 3 | X vale 6 |
| & | AND | X = 2 & -2 | X vale 2 |
| ^ | XOR | X = 7 ^ -2 | X vale -7 |
| \| | OR | X = 6 \| 13 | X vale 15 |

A pesar de ser "Operadores para manejo de bits", todos ellos exigen operandos de tipo entero, que puede ser de cualquiera de sus variantes (short, long, signed o unsigned) y enumeraciones. Es decir, el material de partida son bytes, uno o varios, dependiendo del tipo de entero utilizado.

Si los operandos no son enteros el compilador realiza la conversión pertinente, por lo que el resultado es siempre un entero del mismo tipo que los operandos.

No se debe confundir los operadores de bits, & y |, con los operadores lógicos && y ||.

En lo relativo al tratamiento del signo, &, >>, << son sensibles al contexto.

& puede ser también el operador de referencia de punteros, y declarador de referencia.

La librería Estándar C++ ha sobrecargado los operadores << y >> para los tipos básicos, de forma que pueden ser utilizados como operadores de salida y entrada.

El resultado de los operadores AND, XOR y OR es independiente del orden de colocación de sus operandos. Los operadores que gozan de esta propiedad se denominan asociativos. Viene a ser equivalente a la propiedad conmutativa de ciertos operadores aritméticos.

### 2.5.1. Complemento

Es el único operador unario en cuanto a manejo de bits, y básicamente invierte cada bit del operando; 0 es convertido en 1 y viceversa. También es posible usar su funcionalidad a tráves de la palabra reservada **compl.**

# OPERADORES EN C++

```
signed int s1 = ~2;          // equivale a:
signed int s1 = compl 2;
signed int s2 = ~s1 + 2;
```

En la primera línea, el complemento a uno de 2 es asignado al entero con signo s1. Tenga en cuenta que el resultado de este operador cambia el signo del operando, de ahí el "signed".

La representación binaria de los complementos a uno de los decimales 0, 1 y 2 son los que se expresan (para simplificar los representamos como un octeto):

```
0  == 0000 0000  ⇨  ~ 0  == 1111 1111
1  == 0000 0001  ⇨  ~ 1  == 1111 1110
2  == 0000 0010  ⇨  ~ 2  == 1111 1101
```

Es necesario resaltar que los tipos negativos se representan internamente como complemento a dos, de forma que la representación interna de -1, -2 y -3 es:

```
-1  == 1111 1110 + 0000 0001  == 1111 1111
-2  == 1111 1101 + 0000 0001  == 1111 1110
-3  == 1111 1100 + 0000 0001  == 1111 1101
```

Por lo tanto, al ejecutar el siguiente código de la derecha, se tendrá la salida de la izquierda:

```
#include <iostream.h>

  short signed cero = 0, uno = 1, dos = 2;

int main (void) {
  cout << "~0 == " << ~cero << endl;
  cout << "~1 == " << ~uno << endl;
  cout << "~2 == " << ~dos << endl;
}
```

Salida:

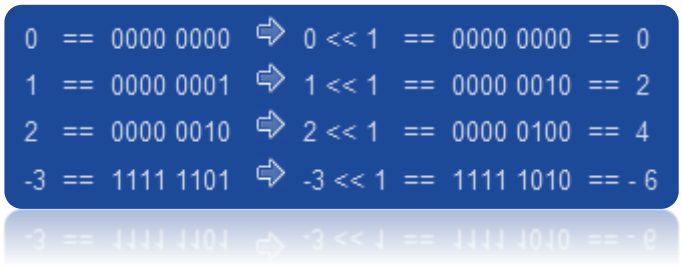
```
~0 == -1
~1 == -2
~2 == -3
```

### 2.5.2. Desplazamiento A Izquierda

Este operador binario realiza un desplazamiento de bits a la izquierda. El bit más significativo (más a la izquierda) se pierde, y se le asigna un 0 al menos significativo (el de la derecha). El operando derecho indica el número de desplazamientos que se realizarán.

Recuérdese que los desplazamientos no son rotaciones; los bits que salen por la izquierda se pierden, los que entran por la derecha se rellenan con ceros. Este tipo de desplazamientos se denominan lógicos en contraposición a los cíclicos o rotacionales.

Los desplazamientos izquierda de valor unitario aplicados sobre los números 0, 1, 2 y -3, producen los siguientes resultados:

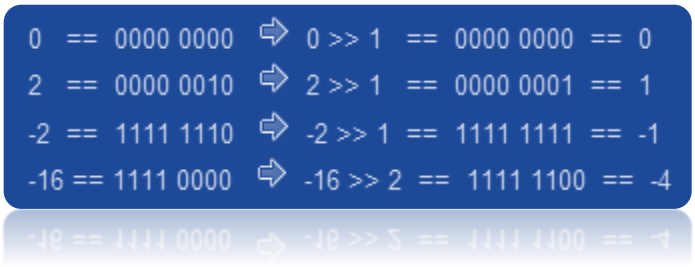



Como es notorio, el desplazamiento unitario a izquierda equivale a multiplicar por dos el valor del operando desplazado.

### 2.5.3. Desplazamiento A Derecha

El bit menos significativo (a la derecha) se pierde, pero hay que advertir que si la expresión desplazada es un entero con signo y es negativo, el resultado depende de la implementación. Además, es necesario resaltar que al igual que con el desplazamiento a derecha, el segundo operando o factor de desplazamiento debe ser positivo y de longitud menor que la del primer operando.

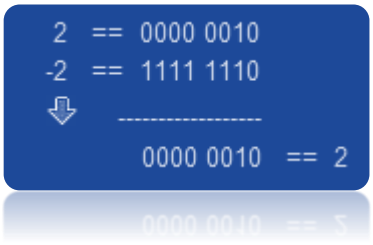
Por lo demás, el comportamiento de este operador es análogo al anterior (desplazamiento izquierda). Por ejemplo:



En contraposición al desplazamiento a izquierda, el desplazamiento unitario a derecha equivale a dividir el primer operando en 2.

### 2.5.4. AND

Este operador binario compara ambos operandos bit a bit, y como resultado devuelve un valor construido de tal forma, que cada bit es 1 si los bits correspondientes de los operandos están a 1. En caso contrario, el bit es 0:



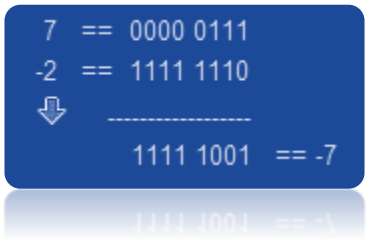

También es posible usar la funcionalidad del operador "&" a través de la palabra reservada **bitand**, tal y como se muestra a continuación:

```
int x = 10, y = 20;
int z = x & y;          // equivale a: int z = x bitand y;
```

En algunos compiladores las palabras reservadas como **bitand** o **compl** pueden estar desactivadas o sin soporte por lo que es recomendable usar siempre el operador.

### 2.5.5. XOR

El funcionamiento de este operador binario es parecido al AND lógico , salvo que en este caso el resultado es 1 si ambos bits son complementarios (uno es 0 y el otro 1). En caso contrario devuelve 0.




Al igual que el operador "&", el operador "^" cuenta con una palabra reservada con la que es posible hacer uso de su funcionalidad:

```
int x = 10, y = 20;
int z = x ^ y;        // equivale a: int z = x xor y;
```

### 2.5.6. OR

Este operador binario tiene un funcionamiento parecido a los anteriores (AND y XOR), salvo que en este caso el resultado es 1 si alguno de ellos está a 1. En caso contrario devuelve 0 (ver ejemplo). Se puede hacer uso de la palabra **bitor** para reemplazar el operador "|":

```
int x = 10, y = 20;
int z = x | y;        // equivale a:
int z = x bitor y;
```

El operador "|" aplicado entre los valores 6 y 13 resultaría" [2]:

```
6   == 0000 0110
13  == 0000 1101
    ----------------
       0000 1111  == 15
```

## 2.6. OPERADORES DE DIRECCIÓN

Además de los operadores aritméticos, de los que existen versiones para los tipos numéricos y para los punteros, C++ dispone de dos operadores específicos para estos últimos (punteros de cualquier tipo): la indirección "*" y la referencia "&" . Además existen dos operadores específicos para punteros-a-clases [2].

| Operador | Nombre | Ejemplo |
|---|---|---|
| * | Deferencia o indirección | int* ptr = 0 |
| & | Referencia o dirección | int* p = &x; |
| .* | Indirección de puntero a miembro | (*pc).*pmint |
| -> | indirección de puntero-a-clase | pc->x |

# OPERADORES EN C++

Dado que estos operadores trabajan con punteros o apuntadores, solo se mencionan en este documento, si se desea profundizar en su uso y aplicación, es necesario recurrir al material de apuntadores.

## 2.7. OTROS OPERADORES

En C++ también existen otros operadores que no fueron mencionados en este documento pero que se consideran importantes o de uso frecuente. La siguiente tabla [4] muestra algunos de ellos, así como otros que ya fueron anteriormente mencionados:

| Nombre del operador | Sintaxis |
|---|---|
| Asignación básica | `a = b` |
| Llamada a función | `a()` |
| Índice de Array | `a[b]` |
| Indirección (Desreferencia) | `*a` |
| Dirección de (Referencia) | `&a` |
| Miembro de puntero | `a->b` |
| Miembro | `a.b` |
| Desreferencia a miembro por puntero | `a->*b` |
| Desreferencia a miembro por objeto | `a.*b` |
| Conversión de tipo | `(tipo) a` |
| Coma | `a , b` |
| Condicional ternario | `a ? b : c` |
| Resolución de ámbito | `a::b` |
| Puntero a función miembro | `a::*b` |
| Tamaño de | `sizeof a`<br>`sizeof(tipo)` |
| Identificación de tipo | `typeid(a)`<br>`typeid(tipo)` |
| Asignar almacenamiento | `new tipo` |
| Asignar almacenamiento (Vector) | `new tipo[n]` |
| Desasignar almacenamiento | `delete a` |
| Desasignar almacenamiento (Vector) | `delete[] a` |

### 2.7.1. OPERADOR CONDICIONAL

"El operador condicional es el único operador ternario de la gramática C++ y sirve para tomar decisiones. Proporciona un resultado entre dos posibilidades en función de una condición.

El operador condicional "? :" produce un resultado. En la expresión E1 ? E2 : E3, E1 es una expresión relacional que se evalúa primero. Si el resultado es cierto, entonces se evalúa E2 y este es el resultado. En caso contrario (si E1 resulta falso), entonces se evalúa E3 y este es el resultado. Si la premisa E1 es cierta, entonces no llega a evaluarse la expresión E3.

E2 y E3 deben seguir las reglas siguientes:

1. Si E2 y E3 son de tipos distintos, se realiza una conversión de tipo estándar, de forma que el resultado será siempre del mismo tipo, con independencia de E1.
2. Si E2 y E3 son de tipos unión o estructuras compatibles. El resultado es una unión o estructura del tipo de E2 y E3.
3. Si E2 y E3 son de tipo void, el resultado es void.
4. Ambos operandos son punteros a versiones cualificadas o no cualificadas de tipos compatibles. El resultado es un puntero a cualquiera de los tipos de ambos operandos.
5. Un operando es un puntero y el otro es un puntero nulo. El resultado es un puntero que puede señalar a un tipo del primero o del segundo operando.
6. Un operando es un puntero a un objeto o tipo incompleto, y el otro es un puntero a una versión cualificada o no cualificada de void. El tipo resultante es el del puntero distinto de void.

En el siguiente ejemplo se muestra el uso del operador condicional, donde a la variable y se le asigna el valor de 100" [2]:

```
x = 10;
y = x > 9 ? 100 : 200;
```

## 3. PRECEDENCIA DE OPERADORES

"En C++ existen 4 aspectos que indican el orden de ejecución de un programa. "Este orden viene determinado por cuatro condicionantes:

1. Presencia de paréntesis que obligan a un orden de evaluación específico.
2. Naturaleza de los operadores involucrados en la expresión (asociatividad).
3. Orden en que están colocados (precedencia).
4. Providencias (impredecibles) del compilador relativas a la optimización del código.

En cuanto al primero, aunque el paréntesis es un signo de puntuación, podría considerarse como el operador de precedencia más alta. Si existen paréntesis, el compilador los evalúa en primer lugar.

El segundo es especialmente importante, porque como veremos a continuación, es precisamente su naturaleza la que establece dos propiedades importantes de los operadores: la asociatividad y la precedencia

El punto tercero es influyente porque a igualdad de precedencia, unos operadores se ejecutan en el orden en que aparecen escritos en el código (de izquierda a derecha), y en otros casos es al contrario (dependiendo de su asociatividad). A su vez el punto cuarto encierra decisiones que son dependientes de la plataforma. Se refieren a medidas del compilador tendentes a la optimización del código de la sentencia, que resultan incontrolables para el programador a no ser que adopte medidas específicas. Estas medidas suelen consistir en no simplificar demasiado las expresiones, y obtener resultados intermedios, que solo son necesarios para obligar a una forma determinada de obtener el resultado.

## 3.1. PRECEDENCIA

La precedencia indica cual es el orden de ejecución de los operadores cuando existen varios. Por ejemplo, en la expresión:

```
a * b + c++;      // L.1
```

La precedencia determina que se ejecutará primero el operador postincremento ++ sobre c. A continuación se aplicará el operador de multiplicación * entre a y b. Finalmente se aplicará el operador suma + entre los resultados anteriores. Así pues, la expresión es equivale a:

```
(a * b) + (c++);  // L.2
```

Este orden "natural" del compilador no necesita paréntesis de forma que las sentencias L1 y L2 producen el mismo resultado. Cualquier otro debe ser forzado específicamente mediante la utilización de los paréntesis correspondientes.

## 3.2. ASOCIATIVIDAD

La asociatividad indica el orden de ejecución cuando en una expresión existen diversos operadores de igual precedencia. Puede ser de dos tipos: izquierda (→) o derecha (←) . Por ejemplo, la suma binaria + tiene asociatividad izquierda, lo que significa que en una expresión como la de la izquierda, se seguiría el orden de la derecha:

```
a + b + c + d;
```

```
(((a + b) + c) + d);
```

Los
operadores unarios y el de asignación (=), tienen asociatividad derecha (←). Todos los demás la tienen izquierda (→). En consecuencia, si @ representa un operador binario, ante una expresión como:

```
a @ b @ c @ d;
```

El orden de evaluación es desde la izquierda hacia la derecha. Pero si la expresión es del tipo:

```
(...) @ (...) @ (...) @ (...);
```

El orden de evaluación de los paréntesis es indefinido. Aunque una vez obtenidos todos los resultados parciales, la computación sigue el orden indicado en el punto anterior. Si existen paréntesis anidados se procede desde dentro hacia fuera" [2].

## 3.3. TABLA DE RESUMEN

La tabla siguiente es una lista que muestra el orden de precedencia y la asociatividad de todos los operadores del lenguaje de programación C++. Están listados de arriba a abajo por orden de precedencia descendente y con la misma descendencia en la misma celda (puede haber varias filas de operadores en la misma celda). La precedencia de los operadores no cambia por la sobrecarga.

# OPERADORES EN C++

Una tabla de precedencias, aunque adecuada, no puede resolver todos los detalles. Por ejemplo, el operador ternario permite expresiones arbitrarias como operador central independientemente de la precedencia del resto de operadores. Así a ? b , c : d es interpretado como a ? (b, c) : d en vez de (a ? b), (c : d). También hay que tener en cuenta que el resultado sin paréntesis de una expresión de conversión en C no puede ser el operando de *sizeof*. Por eso *sizeof* (*int*) * x es interpretado como (*sizeof(int*)) * x y no como *sizeof* ((*int*) *x) [4].

| Operador | Descripción | Asociatividad |
|---|---|---|
| `::` | Resolución de ámbito (solo C++) | Izquierda a derecha |
| `++ --`<br>`()`<br>`[]`<br>`.`<br>`->`<br>`typeid()`<br><br>`const_cast`<br>`dynamic_cast`<br>`reinterpret_cast`<br>`static_cast` | Post- incremento y decremento<br>Llamada a función<br>Elemento de vector<br>Selección de elemento por referencia<br>Selección de elemento con puntero<br>Información de tipo en tiempo de ejecución (solo C++)<br><br>Conversión de tipo (solo C++)<br>Conversión de tipo (solo C++)<br>Conversión de tipo (solo C++)<br>Conversión de tipo (solo C++) | |
| `++ --`<br>`+ -`<br>`! ~`<br>`(type)`<br>`*`<br>`&`<br>`sizeof`<br>`new new[]`<br>`delete delete[]` | Pre- incremento y decremento<br>Suma y resta unitaria<br>NOT lógico y NOT binario<br>Conversión de tipo<br>Indirección<br>Dirección de<br>Tamaño de<br>Asignación dinámica de memoria (solo C++)<br>Desasignación dinámica de memoria (solo C++) | Derecha a izquierda |
| `.* ->*` | Puntero a miembro (solo C++) | Izquierda a derecha |
| `* / %` | Multiplicación, división y módulo | |

# OPERADORES EN C++

| | | |
|---|---|---|
| + - | Suma y resta | |
| << >> | Operaciones binarias de desplazamiento | |
| < <=<br>> >= | Operadores relaciones "menor que", "menor o igual que", "mayor que" y "mayor o igual que" | |
| == != | Operadores relaciones "igual a" y "distinto de" | |
| & | AND binario | |
| ^ | XOR binario | |
| \| | OR binario | |
| && | AND lógico | |
| \|\| | OR lógico | |
| *c*?*t*:*f* | Operador ternario | Derecha a izquierda |
| =<br>+= -=<br>*= /= %=<br><<= >>=<br>&= ^= \|= | Asignaciones | |
| throw | Operador Throw (lanzamiento de excepciones, solo C++) | |
| , | Coma | Izquierda a derecha |

## 4. REFERENCIAS

[1] Microsoft, «MSDN,» [En línea]. Available: http://msdn.microsoft.com/es-es/library/ms173145.aspx. [Último acceso: 04 11 2012].

[2] Z. Systems, «Curso C++,» [En línea]. Available: http://www.zator.com/Cpp/E_Ce.htm. [Último acceso: 08 11 2012].

[3] R. P. Vivanco, «Curso C,» [En línea]. Available: http://laurel.datsi.fi.upm.es/~rpons/personal/trabajos/curso_c. [Último acceso: 08 11 2012].

[4] Wikipedia, «Wikipedia,» [En línea]. Available: http://es.wikipedia.org/wiki/Anexo:Operadores_de_C_y_C%2B%2B. [Último acceso: 08 11 201(