Operators

Once introduced to variables and constants, we can begin to operate with them by using operators. What follows is a complete list of operators. At this point, it is likely not necessary to know all of them, but they are all listed here to also serve as reference.

Assignment Operator (=)

The assignment operator assigns a value to a variable.

```
x = 5;
```

This statement assigns the integer value 5 to the variable x. The assignment operation always takes place from right to left, and never the other way around:

```
x = y;
```

This statement assigns to variable x the value contained in variable y. The value of x at the moment this statement is executed is lost and replaced by the value of y.

Consider also that we are only assigning the value of y to x at the moment of the assignment operation. Therefore, if y changes at a later moment, it will not affect the new value taken by x.

For example, let's have a look at the following code - I have included the evolution of the content stored in the variables as comments:

```
// assignment operator
#include <iostream>
using namespace std;

int main ()
{
  int a, b;         // a:?,  b:?
  a = 10;           // a:10, b:?
  b = 4;            // a:10, b:4
  a = b;            // a:4,  b:4
  b = 7;            // a:4,  b:7

  cout << "a:";
  cout << a;
  cout << " b:";
  cout << b;
}
```

Execute:

```
a:4 b:7
```

The following expression is also valid in C++:

```
x = y = z = 5;
```

It assigns 5 to the all three variables: x, y and z; always from right-to-left.

Arithmetic operators (+, -, *, /, %)

The five arithmetical operations supported by C++ are:

operator	description
+	addition
-	subtraction
*	multiplication
/	division
%	modulo

Operations of addition, subtraction, multiplication and division correspond literally to their respective mathematical operators. The last one, modulo operator, represented by a percentage sign (%), gives the remainder of a division of two values. For example:

```
x = 11 % 3;
```

results in variable x containing the value 2, since dividing 11 by 3 results in 3, with a remainder of 2.

Compound assignment (+=, -=, *=, /=, %=, >>=, <<=, &=, ^=, |=)

Compound assignment operators modify the current value of a variable by performing an operation on it. They are equivalent to assigning the result of an operation to the first operand:

expression	equivalent to...
y += x;	y = y + x;
x -= 5;	x = x - 5;
x /= y;	x = x / y;
price *= units + 1;	price = price * (units+1);

and the same for all other compound assignment operators. For example:

```
// compound assignment operators
#include <iostream>
using namespace std;

int main ()
{
  int a, b=3;
  a = b;
  a+=2;               // equivalent to a=a+2
  cout << a;
}
```

Execute:

```
5
```

Increment and decrement (++, --)

Some expression can be shortened even more: the increase operator (++) and the decrease operator (--) increase or reduce by one the value stored in a variable. They are equivalent to +=1 and to -=1, respectively. Thus:

```
1  ++x;
2  x+=1;
3  x=x+1;
```

are all equivalent in its functionality; the three of them increase by one the value of x.

In the early C compilers, the three previous expressions may have produced different executable code depending on which one was used. Nowadays, this type of code optimization is generally performed automatically by the compiler, thus the three expressions should produce exactly the same executable code.

A peculiarity of this operator is that it can be used both as a prefix and as a suffix. That means that it can be written either before the variable name (++x) or after it (x++). Although in simple expressions like x++ or ++x, both have exactly the same meaning; in other expressions in which the result of the increment or decrement operation is evaluated, they may have an important difference in their meaning: In the case that the increase operator is used as a prefix (++x) of the value, the expression evaluates to the final value of x, once it is already increased. On the other hand, in case that it is used as a suffix (x++), the value is also increased, but the expression evaluates to the value that x had before being increased. Notice the difference:

Example 1	Example 2

`x = 3;`
`y = ++x;`
`// x contains 4, y contains 4`
	`x = 3;`
	`y = x++;`
	`// x contains 4, y contains 3`

In *Example 1*, the value assigned to y is the value of x after being increased. While in *Example 2*, it is the value x had before being increased.

Relational and comparison operators (==, !=, >, <, >=, <=)

Two expressions can be compared using relational and equality operators. For example, to know if two values are equal or if one is greater than the other.

The result of such an operation is either true or false (i.e., a Boolean value).

The relational operators in C++ are:

operator	Description
==	Equal to
!=	Not equal to
<	Less than
>	Greater than
<=	Less than or equal to
>=	Greater than or equal to

Here there are some examples:

```
(7 == 5)      // evaluates to false
(5 > 4)       // evaluates to true
(3 != 2)      // evaluates to true
(6 >= 6)      // evaluates to true
(5 < 5)       // evaluates to false
```

Of course, it's not just numeric constants that can be compared, but just any value, including, of course, variables. Suppose that `a=2`, `b=3` and `c=6`, then:

```
(a == 5)      // evaluates to false, since a is not equal to 5
(a*b >= c)    // evaluates to true, since (2*3 >= 6) is true
(b+4 > a*c)   // evaluates to false, since (3+4 > 2*6) is false
((b=2) == a) // evaluates to true
```

Be careful! The assignment operator (operator `=`, with one equal sign) is not the same as the equality comparison operator (operator `==`, with two equal signs); the first one (`=`) assigns the value on the right-hand to the variable on its left, while the other (`==`) compares whether the values on both sides of the operator are equal. Therefore, in the last expression (`(b=2) == a`), we first assigned the value `2` to `b` and then we compared it to `a` (that also stores the value 2), yielding `true`.

Logical operators (!, &&, ||)

The operator ! is the C++ operator for the Boolean operation NOT. It has only one operand, to its right, and inverts it, producing false if its operand is true, and true if its operand is false. Basically, it returns the opposite Boolean value of evaluating its operand. For example:

```
!(5 == 5)    // evaluates to false because the expression at its right (5 == 5) is true
!(6 <= 4)    // evaluates to true because (6 <= 4) would be false
!true        // evaluates to false
!false       // evaluates to true
```

The logical operators && and || are used when evaluating two expressions to obtain a single relational result. The operator && corresponds to the Boolean logical operation AND, which yields true if both its operands are true, and false otherwise. The following panel shows the result of operator && evaluating the expression a&&b:

&& OPERATOR (and)		
a	**b**	**a && b**
true	true	True
true	false	False
false	true	False
false	false	False

The operator || corresponds to the Boolean logical operation OR, which yields true if either of its operands is true, thus being false only when both operands are false. Here are the possible results of a||b:

|| OPERATOR (or)		
a	b	a || b
true	true	True
true	false	True
false	true	True
false	false	False

For example:

```
( (5 == 5) && (3 > 6) )  // evaluates to false ( true && false )
( (5 == 5) || (3 > 6) )  // evaluates to true ( true || false )
```

When using the logical operators, C++ only evaluates what is necessary from left to right to come up with the combined relational result, ignoring the rest. Therefore, in the last example ((5==5)||(3>6)), C++ evaluates first whether 5==5 is true, and if so, it never checks whether 3>6 is true or not. This is known as *short-circuit evaluation*, and works like this for these operators:

operator	short-circuit
&&	if the left-hand side expression is false, the combined result is false (the right-hand side expression is never evaluated).
||	if the left-hand side expression is true, the combined result is true (the right-hand side expression is never evaluated).

This is mostly important when the right-hand expression has side effects, such as altering values:

```
if ( (i<10) && (++i<n) ) { /*...*/ }   // note that the condition increments i
```

Here, the combined conditional expression would increase i by one, but only if the condition on the left of && is true, because otherwise, the condition on the right-hand side (++i<n) is never evaluated.

Conditional ternary operator (?)

The conditional operator evaluates an expression, returning one value if that expression evaluates to true, and a different one if the expression evaluates as false. Its syntax is:

```
condition ? result1 : result2
```

If condition is true, the entire expression evaluates to result1, and otherwise to result2.

```
7==5 ? 4 : 3      // evaluates to 3, since 7 is not equal to 5.
7==5+2 ? 4 : 3    // evaluates to 4, since 7 is equal to 5+2.
5>3 ? a : b       // evaluates to the value of a, since 5 is greater than 3.
a>b ? a : b       // evaluates to whichever is greater, a or b.
```

For example:

```
// conditional operator
#include <iostream>
using namespace std;

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

  a=2;
  b=7;
  c = (a>b) ? a : b;

  cout << c << '\n';
}
```

Execute:

7

In this example, a was 2, and b was 7, so the expression being evaluated (a>b) was not true, thus the first value specified after the question mark was discarded in favor of the second value (the one after the colon) which was b (with a value of 7).

Comma operator (,)

The comma operator (,) is used to separate two or more expressions that are included where only one expression is expected. When the set of expressions has to be evaluated for a value, only the right-most expression is considered. For example, the following code:

```
a = (b=3, b+2);
```

would first assign the value 3 to b, and then assign b+2 to variable a. So, at the end, variable a would contain the value 5 while variable b would contain value 3.

Bitwise operators (& , | , ^ , ~ , << , >>)

Bitwise operators modify variables considering the bit patterns that represent the values they store.

Operator	Description	Example
&	Binary AND Operator copies a bit to the result if it exists in both operands.	(A & B) = 12, i.e., 0000 1100
|	Binary OR Operator copies a bit if it exists in either operand.	(A | B) = 61, i.e., 0011 1101
^	Binary XOR Operator copies the bit if it is set in one operand but not both.	(A ^ B) = 49, i.e., 0011 0001
~	Binary One's Complement Operator is unary and has the effect of 'flipping' bits.	(~A) = ~(60), i.e,. 1100 0011
<<	Binary Left Shift Operator. The left operands value is moved left by the number of bits specified by the right operand.	A << 2 = 240 i.e., 1111 0000
>>	Binary Right Shift Operator. The left operands value is moved right by the number of bits specified by the right operand.	A >> 2 = 15 i.e., 0000 1111

Example

Try the following example to understand all the bitwise operators available in C

```
#include <iostream>
using namespace std;
int main()
{

   int a = 60;    /* 60 = 0011 1100 */
   int b = 13;    /* 13 = 0000 1101 */
   int c = 0;

   c = a & b;       /* 12 = 0000 1100 */
   cout << "1- Value of c is  " << c << endl;

   c = a | b;       /* 61 = 0011 1101 */
   cout << "2- Value of c is  " << c << endl;

   c = a ^ b;       /* 49 = 0011 0001 */
   cout << "3- Value of c is  " << c << endl;

   c = ~a;          /*-61 = 1100 0011 */
   cout << "4- Value of c is  " << c << endl;

   c = a << 2;     /* 240 = 1111 0000 */
   cout << "5- Value of c is  " << c << endl;

   c = a >> 2;     /* 15 = 0000 1111 */
   cout << "6 - Value of c is  " << c <<
endl;

return 0;
}
```

```
Execute:
1- Value of c is 12
2- Value of c is 61
3- Value of c is 49
4- Value of c is -61
5- Value of c is 240
6- Value of c is 15
```

Explicit type casting operator

Type casting operators allow to convert a value of a given type to another type. There are several ways to do this in C++. The simplest one, which has been inherited from the C language, is to precede the expression to be converted by the new type enclosed between parentheses (()):

```
1  int i;
2  float f = 3.14;
3  i = (int) f;
```

The previous code converts the floating-point number 3.14 to an integer value (3); the remainder is lost. Here, the typecasting operator was (int). Another way to do the same thing in C++ is to use the functional notation preceding the expression to be converted by the type and enclosing the expression between parentheses:

```
i = int (f);
```

Both ways of casting types are valid in C++.

Precedence of operators

A single expression may have multiple operators. For example:

```
x = 5 + 7 % 2;
```

In C++, the above expression always assigns 6 to variable x, because the % operator has a higher precedence than the + operator, and is always evaluated before. Parts of the expressions can be enclosed in parenthesis to override this precedence order, or to make explicitly clear the intended effect. Notice the difference:

```
1  x = 5 + (7 % 2);     // x = 6 (same as without parenthesis)
2  x = (5 + 7) % 2;     // x = 0
```

From greatest to smallest priority, C++ operators are evaluated in the following order:

Level	Precedence group	Operator	Description	Grouping
1	Scope	`::`	scope qualifier	Left-to-right
2	Postfix (unary)	`++ --`	postfix increment / decrement	Left-to-right
		`()`	functional forms	
		`[]`	subscript	
		`. ->`	member access	
3	Prefix (unary)	`++ --`	prefix increment / decrement	Right-to-left
		`~ !`	bitwise NOT / logical NOT	
		`+ -`	unary prefix	
		`& *`	reference / dereference	
		`new delete`	allocation / deallocation	
		`Sizeof`	parameter pack	
		(*type*)	C-style type-casting	
4	Pointer-to-member	`.* ->*`	access pointer	Left-to-right
5	Arithmetic: scaling	`* / %`	multiply, divide, modulo	Left-to-right
6	Arithmetic: addition	`+ -`	addition, subtraction	Left-to-right
7	Bitwise shift	`<< >>`	shift left, shift right	Left-to-right
8	Relational	`< > <= >=`	comparison operators	Left-to-right
9	Equality	`== !=`	equality / inequality	Left-to-right
10	And	`&`	bitwise AND	Left-to-right
11	Exclusive or	`^`	bitwise XOR	Left-to-right
12	Inclusive or	`|`	bitwise OR	Left-to-right
13	Conjunction	`&&`	logical AND	Left-to-right
14	Disjunction	`||`	logical OR	Left-to-right
15	Assignment-level expressions	`= *= /= %= += -= >>= <<= &= ^= |=`	assignment / compound assignment	Right-to-left
		`?:`	conditional operator	
16	Sequencing	`,`	comma separator	Left-to-right

When an expression has two operators with the same precedence level, *grouping* determines which one is evaluated first: either left-to-right or right-to-left.

Enclosing all sub-statements in parentheses (even those unnecessary because of their precedence) improves code readability.