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.

一旦介绍了变量和常量，我们就可以开始使用*操作符*来操作它们。下面是操作符的完整列表。在这一点上，可能没有必要了解所有这些内容，但在这里列出它们也是作为参考。

Assignment operator (=)

The assignment operator assigns a value to a variable.

赋值操作符将一个值赋给变量。

```
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:

该语句将整数值' 5 '赋给变量' x '。赋值操作总是从右向左进行，而不是从左向右:

```
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`.

该语句将变量y中包含的值赋给变量x，执行该语句时x的值将丢失并被y的值替换。

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`.

还要考虑，我们只是在赋值操作时将y的值赋给x。因此，如果y稍后发生变化，不会影响x所取的新值。

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:

例如，让我们看一下下面的代码——我已经将存储在变量中的内容的演变包括为注释:

```
// 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;
}
```

This program prints on screen the final values of `a` and `b` (4 and 7, respectively). Notice how `a` was not affected by the final modification of `b`, even though we declared `a = b` earlier.

这个程序在屏幕上输出a和b的最终值(分别为4和7)。注意a是如何不受b的最终修改的影响的，即使我们之前声明了a = b。

Assignment operations are expressions that can be evaluated. That means that the assignment itself has a value, and -for fundamental types- this value is the one assigned in the operation. For example:

赋值操作是可以计算的表达式。这意味着赋值本身有一个值，并且—对于基本类型—该值是在操作中赋值的值。例如:

```
y = 2 + (x = 5);
```

In this expression, `y` is assigned the result of adding 2 and the value of another assignment expression (which has itself a value of 5). It is roughly equivalent to:

在这个表达式中，y被赋值为2加上另一个赋值表达式的值(该表达式本身的值为5)。它大致相当于:

```
x = 5;
y = 2 + x;
```

With the final result of assigning 7 to `y`.

最终的结果是把7赋给y。

The following expression is also valid in C++:

下面的表达式在c++中也是有效的:

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

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

它给所有三个变量赋值5:' x ', ' y '和' z ';总是从从右到左。

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

The five arithmetical operations supported by C++ are:

c++支持的五种算术运算是:

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:

加、减、乘、除运算字面上对应于它们各自的数学运算符。最后一个是*模运算符*，由百分号(' % ')表示，给出两个值的除法的余数。例如:

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

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

结果变量' x '包含值2，因为11除以3得到3，余数为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:

复合赋值操作符通过对变量执行操作来修改变量的当前值。它们相当于将一个操作的结果赋值给第一个操作数:

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:

对于所有其他复合赋值运算符也是一样的。例如:

```
// 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;
}
```

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:

有些表达式还可以进一步简化:递增操作符(' ++ ')和递减操作符('——')将存储在变量中的值增加或减少一个。它们分别等价于' +=1 '和' -=1 '。因此:

```
++x;
x+=1;
x=x+1;
```

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

在功能上都是相同的;它们三个都是x的值增加1。

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.

在早期的C编译器中，根据使用的是哪一个表达式，前面的三个表达式可能会产生不同的可执行代码。现在，这种类型的代码优化通常由编译器自动执行，因此这三个表达式应该生成完全相同的可执行代码。

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:

该操作符的一个特点是既可以用作前缀，也可以用作后缀。这意味着它可以写在变量名(++x)之前，也可以写在变量名(x++)之后。尽管在像x++或++x这样的简单表达式中，它们有完全相同的含义;在其他表达式的结果计算盈亏操作,他们可能有一个重要区别的意义:如果增加操作符作为前缀(+ + x)的价值,表达式的求值结果为最终x的值,一旦它已经增加了。另一方面，如果它被用作后缀(x++)，值也会增加，但表达式的计算结果是x在增加之前的值。注意区别:

```
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.

例1中，赋给y的值是x增加后的值。而在例2中，它是x在被增加之前的值。

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.

两个表达式可以使用关系和相等操作符进行比较。例如，知道两个值是否相等或是否一个大于另一个。

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

这种操作的结果要么是真，要么是假(即，一个布尔值)。

The relational operators in C++ are:

c++中的关系操作符是:

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:

当然，可以比较的不只是数值常量，而是任何值，当然包括变量。假设a=2, b=3, c=6，则:

```
(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`.

小心!赋值操作符(带一个等号的操作符' = ')与相等比较操作符(带两个等号的操作符' == ')不一样;第一个(' = ')将右边的值赋给左边的变量，而另一个(' == ')比较运算符两边的值是否相等。因此，在最后一个表达式(' (b=2) == a ')中，我们首先将值' 2 '赋给' b '，然后将它与' a '(也存储值2)进行比较，得到' 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:

操作符!是Boolean操作NOT的c++操作符。它只有一个操作数，在它的右边，并反转它，如果它的操作数为真，则产生false，如果它的操作数为假，则产生true。基本上，它返回与运算数相反的布尔值。例如:

```
!(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`:

逻辑运算符' && '和' || '用于计算两个表达式以获得单个关系结果。运算符' && '对应于布尔逻辑操作AND，如果它的两个操作数都是' true '，则产生' true '，否则产生' false '。下面的面板显示了运算符' && '求表达式' a&&b '的结果:

&& 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`:

操作符' || '对应于布尔逻辑运算OR，如果其中一个操作数为'真'，则产生'真'，因此只有当两个操作数都为假时才为假。以下是' a||b '可能的结果:

|| 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:

在使用逻辑运算符时，c++只计算从左到右所需的结果，而忽略其余的结果。因此，在最后一个例子('(5==5)||(3>6)')中，c++首先计算' 5==5 '是否为'真'，如果是，它从不检查' 3>6 '是否为'真'。这被称为“短路计算”，对这些操作符的工作原理如下:

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:

当右表达式有副作用时，例如改变值时，这一点非常重要:

```
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.

在这里，组合条件表达式将' i '增加1，但只有在' && '左边的条件为' true '的情况下，因为否则，右边的条件(' ++i<n ')永远不会被计算。

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:

条件操作符计算一个表达式，如果表达式计算结果为“true”返回一个值，如果表达式计算结果为“false”返回另一个值。它的语法是:

```
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';
}
```

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).

在这个例子中，' a '是2，而' b '是7，因此正在计算的表达式(' a>b ')不是' true '，因此在问号之后指定的第一个值被丢弃，取而代之的是第二个值(冒号后面的值)，即' b '(值为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.

逗号操作符(', ')用于分隔两个或多个表达式，这些表达式包含在只需要一个表达式的地方。当必须对表达式集求值时，只考虑最右边的表达式。

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.

首先将值3赋给' b '，然后将' b+2 '赋给变量' a '。因此，在最后，变量a将包含值5，而变量b将包含值3。

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

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

按位操作符根据表示变量存储值的位模式来修改变量。

operator	asm equivalent	description
`&`	`AND`	Bitwise AND
`|`	`OR`	Bitwise inclusive OR
`^`	`XOR`	Bitwise exclusive OR
`~`	`NOT`	Unary complement (bit inversion)
`<<`	`SHL`	Shift bits left
`>>`	`SHR`	Shift bits right

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 (()):

类型强制转换操作符允许将给定类型的值转换为另一类型。在c++中有几种方法可以做到这一点。最简单的一个是从C语言继承来的，是在要被括号(())括起来的新类型转换的表达式之前:

```
int i;
float f = 3.14;
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:

前面的代码将浮点数' 3.14 '转换为整数值(' 3 ');剩下的部分丢失了。这里，类型转换操作符是' (int) '。在c++中做同样事情的另一种方法是在要被类型转换的表达式之前使用函数表示法，并将表达式括在括号中:

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

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

这两种类型转换方法在c++中都是有效的。

sizeof

This operator accepts one parameter, which can be either a type or a variable, and returns the size in bytes of that type or object:

该操作符接受一个形参，形参可以是类型或变量，并返回该类型或对象的大小(以字节为单位):

```
x = sizeof (char);
```

Here, `x` is assigned the value `1`, because `char` is a type with a size of one byte.

在这里，' x '被赋值为' 1 '，因为' char '是一个大小为一个字节的类型。

The value returned by `sizeof` is a compile-time constant, so it is always determined before program execution.

' sizeof '返回的值是一个编译时常量，所以它总是在程序执行之前确定。

Other operators

Later in these tutorials, we will see a few more operators, like the ones referring to pointers or the specifics for object-oriented programming.

在这些教程的后面，我们将看到更多的操作符，比如指向指针的操作符或面向对象编程的具体操作符。

Precedence of operators

A single expression may have multiple operators. For example:

一个表达式可以有多个操作符。例如:

```
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:

在c++中，上面的表达式总是将6赋值给变量' x '，因为' % '操作符的优先级高于' + '操作符，并且总是在前面求值。表达式的部分可以括在括号中，以覆盖此优先顺序，或显式地明确预期的效果。注意区别:

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

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

从最大优先级到最小优先级，c++操作符的计算顺序如下:

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.

当一个表达式有两个具有相同优先级的操作符时，*grouping*决定哪个操作符先被计算:是从左到右还是从右到左。

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

将所有子语句都包含在括号中(即使是那些没有必要的语句，因为它们具有优先级)可以提高代码的可读性