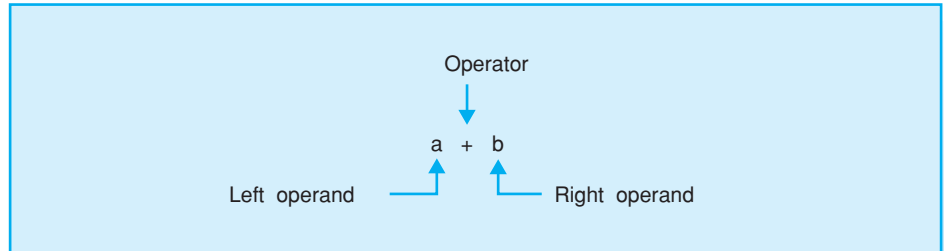


# Operators for Fundamental Types

In this chapter, operators needed for calculations and selections are introduced. Overloading and other operators, such as those needed for bit manipulations, are introduced in later chapters.

## BINARY ARITHMETIC OPERATORS

### Binary operator and operands



### The binary arithmetic operators

Operator	Significance
+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Remainder

### Sample program

```
#include <iostream>
using namespace std;
int main()
{
    double x, y;
    cout << "\nEnter two floating-point values: ";
    cin >> x >> y;
    cout << "The average of the two numbers is: "
         << (x + y)/2.0 << endl;
    return 0;
}
```

### Sample output for the program

```
Enter two floating-point values:  4.75   12.3456
The average of the two numbers is: 8.5478
```

If a program is to be able to process the data input it receives, you must define the operations to be performed for that data. The operations being executed will depend on the type of data — you could add, multiply, or compare numbers, for example. However, it would make no sense at all to multiply strings.

The following sections introduce you to the most important operators that can be used for arithmetic types. A distinction is made between *unary* and *binary* operators. A unary operator has only one operand, whereas a binary operator has two.

## □ Binary Arithmetic Operators

*Arithmetic operators* are used to perform calculations. The opposite page shows an overview. You should be aware of the following:

- *Divisions* performed with integral operands will produce integral results; for example,  $7/2$  computes to 3. If at least one of the operands is a floating-point number, the result will also be a floating-point number; e.g., the division  $7.0/2$  produces an exact result of 3.5.
- *Remainder division* is only applicable to integral operands and returns the remainder of an integral division. For example,  $7\%2$  computes to 1.

## □ Expressions

In its simplest form an expression consists of only one constant, one variable, or one function call. Expressions can be used as the operands of operators to form more complex expressions. An expression will generally tend to be a combination of operators and operands.

Each expression that is not a void type returns a value. In the case of arithmetic expressions, the operands define the type of the expression.

**Examples:**

```
int a(4);    double x(7.9);
a * 512      // Type int
1.0 + sin(x) // Type double
x - 3        // Type double, since one
              // operand is of type double
```

An expression can be used as an operand in another expression.

**Example:** `2 + 7 * 3` // Adds 2 and 21

Normal *mathematical rules* (multiplication before addition) apply when evaluating an expression, i.e. the `*`, `/`, `%` operators have higher precedence than `+` and `-`. In our example,  $7*3$  is first calculated before adding 2. However, you can use parentheses to apply a different precedence order.


**Example:** `(2 + 7) * 3` // Multiplies 9 by 3.

## ■ UNARY ARITHMETIC OPERATORS

### The unary arithmetic operators

Operator	Significance
+   -	Unary sign operators
++	Increment operator
--	Decrement operator

### Precedence of arithmetic operators

Precedence	Operator	Grouping
<div style="text-align: center;">           High              Low         </div>	++   --   (postfix)	left to right
	++   --   (prefix) +   -   (sign)	right to left
	*   /   %	left to right
	+   (addition) -   (subtraction)	left to right

### Effects of prefix and postfix notation

```
#include <iostream>
using namespace std;
int main()
{
    int i(2), j(8);

    cout << i++ << endl;      // Output: 2
    cout << i << endl;        // Output: 3
    cout << j-- << endl;      // Output: 8
    cout << --j << endl;      // Output: 6

    return 0;
}
```

There are four unary arithmetic operators: the sign operators `+` and `-`, the increment operator `++`, and the decrement operator `--`.

## □ Sign Operators

The *sign operator* `-` returns the value of the operand but inverts the sign.

**Example:** `int n = -5;    cout << -n;    // Output: 5`

The *sign operator* `+` performs no useful operation, simply returning the value of its operand.

## □ Increment / Decrement Operators

The increment operator `++` modifies the operand by adding 1 to its value and cannot be used with constants for this reason.

Given that `i` is a variable, both `i++` (*postfix notation*) and `++i` (*prefix notation*) raise the value of `i` by 1. In both cases the operation `i = i + 1` is performed.

However, prefix `++` and postfix `++` are two different operators. The difference becomes apparent when you look at the value of the expression; `++i` means that the value of `i` has already been incremented by 1, whereas the expression `i++` retains the original value of `i`. This is an important difference if `++i` or `i++` forms part of a more complex expression:

<code>++i</code>	<code>i</code> is incremented first and the new value of <code>i</code> is then applied,
<code>i++</code>	the original value of <code>i</code> is applied before <code>i</code> is incremented.

The decrement operator `--` modifies the operand by reducing the value of the operand by 1. As the sample program opposite shows, prefix or postfix notation can be used with `--`.

## □ Precedence

How is an expression with multiple operators evaluated?

**Example:** `float val(5.0);    cout << val++ - 7.0/2.0;`

*Operator precedence* determines the order of evaluation, i.e. how operators and operands are grouped. As you can see from the table opposite, `++` has the highest precedence and `/` has a higher precedence than `-`. The example is evaluated as follows: `(val++) - (7.0/2.0)`. The result is 1.5, as `val` is incremented later.

If two operators have equal precedence, the expression will be evaluated as shown in column three of the table.

**Example:** `3 * 5 % 2` is equivalent to `(3 * 5) % 2`

## ■ ASSIGNMENTS

### Sample program

```
// Demonstration of compound assignments

#include <iostream>
#include <iomanip>
using namespace std;

int main()
{
    float x, y;

    cout << "\n Please enter a starting value: ";
    cin >> x;

    cout << "\n Please enter the increment value: ";
    cin >> y;

    x += y;

    cout << "\n And now multiplication! ";
    cout << "\n Please enter a factor: ";
    cin >> y;

    x *= y;

    cout << "\n Finally division.";
    cout << "\n Please supply a divisor: ";
    cin >> y;

    x /= y;

    cout << "\n And this is "
        << "your current lucky number: "
            // without digits after
            // the decimal point:
        << fixed << setprecision(0)
        << x << endl;

    return 0;
}
```

## □ Simple Assignments

A *simple* assignment uses the assignment operator = to assign the value of a variable to an expression. In expressions of this type the variable must be placed on the left and the assigned value on the right of the assignment operator.

**Examples:** `z = 7.5;`  
`y = z;`  
`x = 2.0 + 4.2 * z;`

The assignment operator has low precedence. In the case of the last example, the right side of the expression is first evaluated and the result is assigned to the variable on the left.

Each assignment is an expression in its own right, and its value is the value assigned.

**Example:** `sin(x = 2.5);`

In this assignment the number 2.5 is assigned to `x` and then passed to the function as an argument.

*Multiple* assignments, which are always evaluated from right to left, are also possible.

**Example:** `i = j = 9;`

In this case the value 9 is first assigned to `j` and then to `i`.

## □ Compound Assignments

In addition to simple assignment operators there are also compound assignment operators that simultaneously perform an arithmetic operation and an assignment, for example.

**Examples.** `i += 3;` is equivalent to `i = i + 3;`  
`i *= j + 2;` is equivalent to `i = i * (j+2);`

The second example shows that compound assignments are implicitly placed in parentheses, as is demonstrated by the fact that the precedence of the compound assignment is just as low as that of the simple assignment.

Compound assignment operators can be composed from any binary arithmetic operator (and, as we will see later, with bit operators). The following compound operators are thus available: `+=`, `-=`, `*=`, `/=`, and `%=`.


You can modify a variable when evaluating a complex expression by means of an assignment or the `++`, `--` operators. This technique is referred to as a *side effect*. Avoid use of side effects if possible, as they often lead to errors and can impair the readability of your programs.

## ■ RELATIONAL OPERATORS

### The relational operators

Operator	Significance
<	less than
<=	less than or equal to
>	greater than
>=	greater than or equal to
==	equal
!=	unequal

### Precedence of relational operators

Precedence	Operator
High	arithmetic operators
	<   <=   >   >=
	==   !=
	assignment operators
Low	

### Examples for comparisons:

Comparison	Result
5 >= 6	false
1.7 < 1.8	true
4 + 2 == 5	false
2 * 4 != 7	true



## □ The Result of Comparisons

Each comparison in C++ is a `bool` type expression with a value of `true` or `false`, where `true` means that the comparison is correct and `false` means that the comparison is incorrect.

**Example:** `length == circuit // false or true`

If the variables `length` and `circuit` contain the same number, the comparison is `true` and the value of the relational expression is `true`. But if the expressions contain different values, the value of the expression will be `false`.

When individual characters are compared, the character codes are compared. The result therefore depends on the character set you are using. The following expression results in the value `true` when ASCII code is used.

**Example:** `'A' < 'a' // true, since 65 < 97`

## □ Precedence of Relational Operators

Relational operators have lower precedence than arithmetic operators but higher precedence than assignment operators.

**Example:** `bool flag = index < max - 1;`

In our example, `max - 1` is evaluated first, then the result is compared to `index`, and the value of the relational expression (`false` or `true`) is assigned to the `flag` variable. Similarly, in the following

**Example:** `int result;  
result = length + 1 == limit;`

`length + 1` is evaluated first, then the result is compared to `limit`, and the value of the relational expression is assigned to the `result` variable. Since `result` is an `int` type, a numerical value is assigned instead of `false` or `true`, i.e. 0 for `false` and 1 for `true`.

It is quite common to assign a value before performing a comparison, and parentheses must be used in this case.

**Example:** `(result = length + 1) == limit`

Our example stores the result of `length + 1` in the variable `result` and then compares this expression with `limit`.

### ✓ NOTE

You cannot use the assignment operator `=` to compare two expressions. The compiler will not generate an error message if the value on the left is a variable. This mistake has caused headaches for lots of beginners when troubleshooting their programs.

## ■ LOGICAL OPERATORS

### “Truth” table for logical operators

A	B	A && B	A    B
true	true	true	true
true	false	false	true
false	true	false	true
false	false	false	false

A	!A
true	false
false	true

### Examples for logical expressions

x	y	Logical Expression	Result
1	-1	<code>x &lt;= y    y &gt;= 0</code>	false
0	0	<code>x &gt; -2 &amp;&amp; y == 0</code>	true
-1	0	<code>x &amp;&amp; !y</code>	true
0	1	<code>!(x+1)    y - 1 &gt; 0</code>	false



#### NOTE

A numeric value, such as `x` or `x+1`, is interpreted as “false” if its value is 0. Any value other than 0 is interpreted as “true.”

The logical operators comprise the *boolean operators* `&&` (AND), `||` (OR), and `!` (NOT). They can be used to create compound conditions and perform conditional execution of a program depending on multiple conditions.

A logical expression results in a value `false` or `true`, depending on whether the logical expression is correct or incorrect, just like a relational expression.

## □ Operands and Order of Evaluation

The operands for boolean type operators are of the `bool` type. However, operands of any type that can be converted to `bool` can also be used, including any arithmetic types. In this case the operand is interpreted as `false`, or converted to `false`, if it has a value of 0. Any other value than 0 is interpreted as `true`.

The **OR** operator `||` will return `true` only if at least one operand is true, so the value of the expression

**Example:** `(length < 0.2) || (length > 9.8)`

is true if `length` is less than 0.2 or greater than 9.8.

The **AND** operator `&&` will return `true` only if both operands are true, so the logical expression

**Example:** `(index < max) && (cin >> number)`

is true, provided `index` is less than `max` and a number is successfully input. If the condition `index < max` is not met, the program will not attempt to read a number! One important feature of the logical operators `&&` and `||` is the fact that there is a fixed order of evaluation. The left operand is evaluated first and if a result has already been ascertained, the right operand will not be evaluated!

The **NOT** operator `!` will return `true` only if its operand is false. If the variable `flag` contains the value `false` (or the value 0), `!flag` returns the boolean value `true`.

## □ Precedence of Boolean Operators

The `&&` operator has higher precedence than `||`. The precedence of both these operators is higher than the precedence of an assignment operator, but lower than the precedence of all previously used operators. This is why it was permissible to omit the parentheses in the examples earlier on in this chapter.

The `!` operator is a unary operator and thus has higher precedence. Refer to the table of precedence in the Appendix for further details.