Constants

Constants are expressions with a fixed value.

You can define your own names for constants that you use very often without having to resort to memory consuming variables, simply by using the #define preprocessor directive. Its format is:

#define identifier value

For example:

```
#define PI 3.14159
#define NEWLINE '\n'
```

This defines two new constants: PI and NEWLINE. Once they are defined, you can use them in the rest of the code as if they were any other regular constant, for example:

```
// defined constants: calculate circumference
#include <iostream>
using namespace std;
#define PI 3.14159
#define NEWLINE '\n'
int main ()
{
double r=5.0; // radius
double circle;
circle = 2 * PI * r;
cout << circle;
cout << NEWLINE;
return 0;
}
```

In fact the only thing that the compiler preprocessor does when it encounters *#define* directives is to literally replace any occurrence of their identifier (in the previous example, these were PI and NEWLINE) by the code to which they have been defined (3.14159 and '\n' respectively). The #define directive is not a C++ statement but a directive for the preprocessor; therefore it assumes the entire line as the directive and does not require a semicolon (;) at its end. If you append a semicolon character (;) at the end, it will also be appended in all occurrences within the body of the program that the preprocessor replaces.

Declared constants (const)

With the const prefix you can declare constants with a specific type in the same way as you would do with a variable:

```
const int pathwidth = 100;

const char tabulator = '\t';
```

Here, pathwidth and tabulator are two typed constants. They are treated just like regular variables except that their values cannot be modified after their definition.

Operators

Once we know of the existence of variables and constants, we can begin to operate with them. For that purpose, C++ integrates operators. Unlike other languages whose operators are mainly keywords, operators in C++ are mostly made of signs that are not part of the alphabet but are available in all keyboards. This makes C++ code shorter and more international, since it relies less on English words, but requires a little of learning effort in the beginning.

Assignment (=)

The assignment operator assigns a value to a variable.

```
a = 5;
```

This statement assigns the integer value 5 to the variable a. The part at the left of the assignment operator (=) is known as the *lvalue* (left value) and the right one as the *rvalue* (right value). The lvalue has to be a variable whereas the rvalue can be a constant, a variable, the result of an operation or any combination of these. The most important rule when assigning is the *right-to-left* rule: The assignment operation always takes place from right to left, and never the other way:

```
a = b;
```

This statement assigns to variable a (the lvalue) the value contained in variable b (the rvalue). The value that was stored until this moment in a is not considered at all in this operation, and in fact that value is lost. Consider also that we are only assigning the value of b to a at the moment of the assignment operation. Therefore a later change of b will not affect the new value of a.

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

```
#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;

return 0;

}
```

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

The five arithmetical operations supported by the C++ language are:

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

Operations of addition, subtraction, multiplication and division literally correspond with their respective mathematical operators. The only one that you might not be so used to see is *modulo*; whose operator is the percentage sign (%). Modulo is the operation that gives the remainder of a division of two values. For example, if we write:

a = 11 % 3;

the variable a will contain the value 2, since 2 is the remainder from dividing 11 between 3.

Compound assignment (+=, -=, *=, /=, %=)

When we want to modify the value of a variable by performing an operation on the value currently stored in that variable we can use compound assignment operators:

Expressions	**Equivalent to**
value += increase;	value = value + increase;
a -= 5;	a = a - 5;
a /= b;	a = a / b;
price *= units + 1;	price = price * (units + 1);

and the same for all other 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;
return 0;
}
```

Increase and decrease (++, --)

Shortening even more some expressions, 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:

```
c++;
```

```
c+=1;
```

```
c=c+1;
```

are all equivalent in its functionality: the three of them increase by one the value of c. A characteristic 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 identifier (++a) or after it (a++). Although in simple expressions like a++ or ++a both have exactly the same meaning, in other expressions in which the result of the increase or decrease operation is evaluated as a value in an outer expression they may have an important difference in their meaning: In the case that the increase operator is used as a prefix (++a) the value is increased before the result of the expression is evaluated and therefore the increased value is considered in the outer expression; in case that it is used as a suffix (a++) the value stored in a is increased after being evaluated and therefore the value stored before the increase operation is evaluated in the outer expression. Notice the difference:

Example 1

```
B=3;
A=++B;
// A contains 4, B contains 4
```

Example 2

```
B=3;
A=B++;
// A contains 3, B contains 4
```

In Example 1, B is increased before its value is copied to A. While in Example 2, the value of B is copied to A and then B is increased.

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

In order to evaluate a comparison between two expressions we can use the relational and equality operators. The result of a relational operation is a Boolean value that can only be true or false, according to its Boolean result. We may want to compare two expressions, for example, to know if they are equal or if one is greater than the other is. Here is a list of the relational and equality operators that can be used in C++:

- == Equal to
- != Not equal to

- > Greater than
- < Less than
- >= Greater than or equal to
- <= Less 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, instead of using only numeric constants, we can use any valid expression, including variables. Suppose that a=2, b=3 and c=6:

```
(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 operator = (one equal sign) is not the same as the operator == (two equal signs), the first one is an assignment operator (assigns the value at its right to the variable at its left) and the other one (==) is the equality operator that compares whether both expressions in the two sides of it are equal to each other. Thus, 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, so the result of the operation is true.

Logical operators (!, &&, ||)

The Operator ! is the C++ operator to perform the Boolean operation NOT, it has only one operand, located at its right, and the only thing that it does is to inverse the value of 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 with Boolean logical operation AND. This operation results true if both its two operands are true, and false otherwise. The following panel shows the result of operator && evaluating the expression a && b:

&& OPERATOR

a	b	a && b
true	true	true
true	false	false
false	true	false
false	false	false

The operator || corresponds with Boolean logical operation OR. This operation results true if either one of its two operands is true, thus being false only when both operands are false themselves. Here are the possible results of a|| b:

|| OPERATOR

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).

Basic Input/Output

Standard Output (cout)

By default, the standard output of a program is the screen, and the C++ stream object defined to access it is cout.
cout is used in conjunction with the *insertion operator*, which is written as << (two "less than" signs).

```
cout << "Output sentence"; // prints Output sentence on screen
cout << 120; // prints number 120 on screen
cout << x; // prints the content of x on screen
```

The << operator inserts the data that follows it into the stream preceding it. In the examples above it inserted the constant string Output sentence, the numerical constant 120 and variable x into the standard output stream cout. Notice that the sentence in the first instruction is enclosed between double quotes (") because it is a constant string of characters. Whenever we want to use constant strings of characters we must enclose them between double quotes (") so that they can

be clearly distinguished from variable names. For example, these two sentences have very different results:

```
cout << "Hello";      // prints Hello
cout << Hello;       // prints the content of Hello variable
```

The insertion operator (<<) may be used more than once in a single statement:

```
cout << "Hello, " << "I am " << "a C++ statement";
```

This last statement would print the message *Hello, I am a C++ statement* on the screen. The utility of repeating the insertion operator (<<) is demonstrated when we want to print out a combination of variables and constants or more than one variable:

```
cout << "Hello, I am " << age << " years old and my zipcode is " << zipcode;
```

If we assume the age variable to contain the value 24 and the zipcode variable to contain 90064 the output of the previous statement would be*: Hello, I am 24 years old and my zipcode is 90064*
It is important to notice that cout does not add a line break after its output unless we explicitly indicate it, therefore, the following statements:

```
cout << "This is a sentence.";
cout << "This is another sentence.";
```

will be shown on the screen one following the other without any line break between them: *This is a sentence.This is another sentence* even though we had written them in two different insertions into cout. In order to perform a line break on the output we must explicitly insert a new-line character into cout. In C++ a new-line character can be specified as \n (backslash, n):

```
cout << "First sentence.\n ";
cout << "Second sentence.\nThird sentence.";
```

This produces the following output:

First sentence.
Second sentence.
Third sentence.

Additionally, to add a new-line, you may also use the endl manipulator. For example:

```
cout << "First sentence." << endl;
cout << "Second sentence." << endl;
```

would print out:

First sentence.
Second sentence.

Standard Input (cin).

The standard input device is usually the keyboard. Handling the standard input in C++ is done by applying the overloaded operator of extraction (>>) on the cin stream. The operator must be followed by the variable that will store the data that is going to be extracted from the stream. For example:

```
int age;
cin >> age;
```

The first statement declares a variable of type int called age, and the second one waits for an input from cin (the keyboard) in order to store it in this integer variable. cin can only process the input from the keyboard once the RETURN key has been pressed. Therefore, even if you request

a single character, the extraction from cin will not process the input until the user presses RETURN after the character has been introduced.
You must always consider the type of the variable that you are using as a container with cin extractions. If you request an integer you will get an integer, if you request a character you will get a character and if you request a string of characters you will get a string of characters.

```
// i/o example
#include <iostream>
using namespace std;
int main ()
{
int i;
cout << "Please enter an integer value: ";
cin >> i;
cout << "The value you entered is " << i;
cout << " and its double is " << i*2 << ".\n";
return 0;
}
```

You can also use cin to request more than one data input from the user:

```
cin >> a >> b;
```

is equivalent to:

```
cin >> a;
cin >> b;
```

In both cases the user must give two data, one for variable a and another one for variable b that may be separated by any valid blank separator: a space, a tab character or a newline.