Basics of C++

**Structure of a program**

Probably the best way to start learning a programming language is by writing a program. Therefore, here is our

First program:



The first panel shows the source code for our first program. The second one shows the result of the program once

compiled and executed. The way to edit and compile a program depends on the compiler you are using.

Depending on whether it has a Development Interface or not and on its version. Consult the compilers section and the manual or help included with your compiler if you have doubts on how to compile a C++ console program.

The previous program is the typical program that programmer apprentices write for the first time, and its result is the printing on screen of the "Hello World!" sentence. It is one of the simplest programs that can be written in C++, but it already contains the fundamental components that every C++ program has. We are going to look line by line at the code we have just written:

**// my first program in C++**

This is a comment line. All lines beginning with two slash signs (//) are considered comments and do not

have any effect on the behavior of the program. The programmer can use them to include short

explanations or observations within the source code itself. In this case, the line is a brief description of

what our program is.

**#include <iostream>**

Lines beginning with a hash sign (#) are directives for the preprocessor. They are not regular code lines

with expressions but indications for the compiler's preprocessor. In this case the directive #include

<iostream> tells the preprocessor to include the iostream standard file. This specific file (iostream)

includes the declarations of the basic standard input-output library in C++, and it is included because its

functionality is going to be used later in the program.

**using namespace std;**

All the elements of the standard C++ library are declared within what is called a namespace, the

namespace with the name std. So in order to access its functionality we declare with this expression that

we will be using these entities. This line is very frequent in C++ programs that use the standard library,

and in fact it will be included in most of the source codes included in these tutorials.

**int main ()**

This line corresponds to the beginning of the definition of the main function. The main function is the point

by where all C++ programs start their execution, independently of its location within the source code. It

does not matter whether there are other functions with other names defined before or after it - the

instructions contained within this function's definition will always be the first ones to be executed in any

C++ program. For that same reason, it is essential that all C++ programs have a main function.

The word main is followed in the code by a pair of parentheses (()). That is because it is a function

declaration: In C++, what differentiates a function declaration from other types of expressions are these

parentheses that follow its name. Optionally, these parentheses may enclose a list of parameters within

them.

Right after these parentheses we can find the body of the main function enclosed in braces ({}). What is

contained within these braces is what the function does when it is executed.

**Comments**

Comments are parts of the source code disregarded by the compiler. They simply do nothing. Their purpose is only to allow the programmer to insert notes or descriptions embedded within the source code. C++ supports two ways to insert comments:

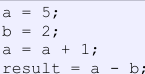
****

The first of them, known as line comment, discards everything from where the pair of slash signs (//) is found up to the end of that same line. The second one, known as block comment, discards everything between the /\* characters and the first appearance of the \*/ characters, with the possibility of including more than one line. We are going to add comments to our second program:



**Variables.**

The usefulness of the "Hello World" programs shown in the previous section is quite questionable. We had to write several lines of code, compile them, and then execute the resulting program just to obtain a simple sentence written on the screen as result. It certainly would have been much faster to type the output sentence by ourselves. However, programming is not limited only to printing simple texts on the screen. In order to go a little further on and to become able to write programs that perform useful tasks that really save us work we need to introduce the concept of variable. Let us think that I ask you to retain the number 5 in your mental memory, and then I ask you to memorize also the number 2 at the same time. You have just stored two different values in your memory. Now, if I ask you to add 1 to the first number I said, you should be retaining the numbers 6 (that is 5+1) and 2 in your memory. Values that we could now for example subtract and obtain 4 as result. The whole process that you have just done with your mental memory is a simile of what a computer can do with two variables. The same process can be expressed in C++ with the following instruction set:



**DATATYPES.**

**1)primitive datatypes**

int myNum = 5; // Integer (whole number)

float myFloatNum = 5.99; // Floating point number

char myLetter = 'D'; // Character

double myDoubleNum = 9.98;

bool myBoolean = true; // Boolean

**2) Derived datatypes**

**Array:**

int myNumbers[5] = {1, 2, 3, 4, 5}; // Array of integers

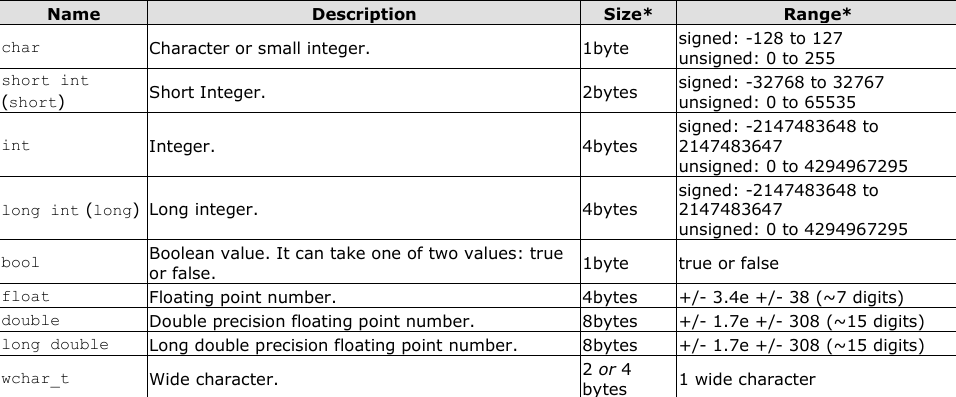
**pointer:**

int \*ptr; // Pointer to an integer

**reference:**

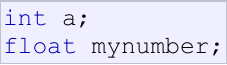
int myNum = 10;

int &ref = myNum; // Reference to myNum



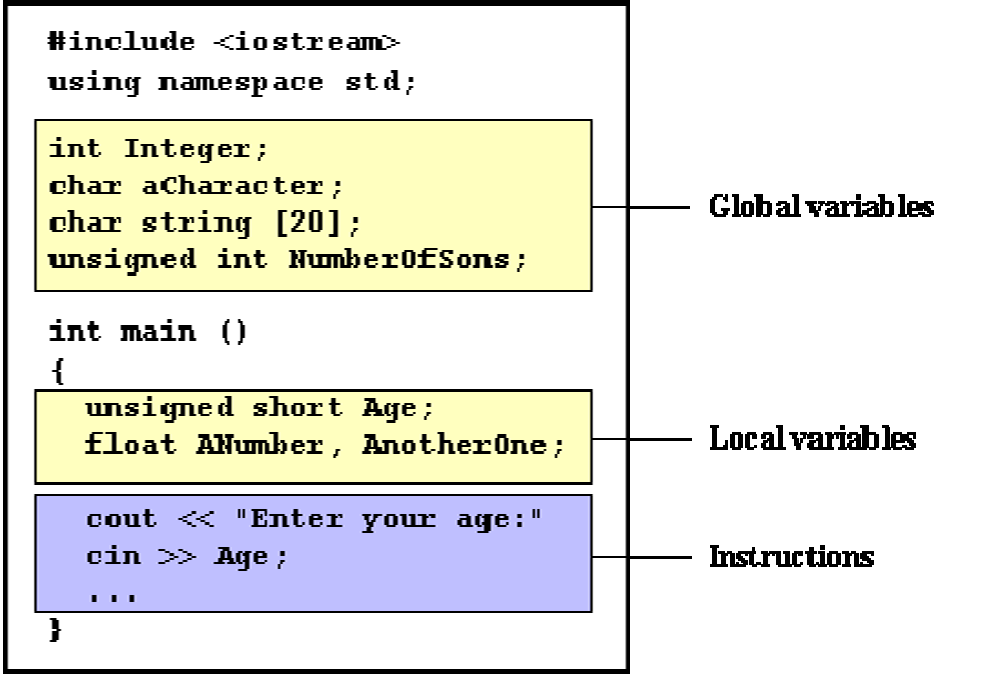
**Declaration of variables**

In order to use a variable in C++, we must first declare it specifying which data type we want it to be. The syntax to declare a new variable is to write the specifier of the desired data type (like int, bool, float...) followed by a valid variable identifier. For example:



**Scope of variables**

All the variables that we intend to use in a program must have been declared with its type specifier in an earlier point in the code, like we did in the previous code at the beginning of the body of the function main when we declared that a, b, and result were of type int. A variable can be either of global or local scope. A global variable is a variable declared in the main body of the source code, outside all functions, while a local variable is one declared within the body of a function or a block.



Global variables can be referred from anywhere in the code, even inside functions, whenever it is after its declaration.

The scope of local variables is limited to the block enclosed in braces ({}) where they are declared. For example, if they are declared at the beginning of the body of a function (like in function main) their scope is between its declaration point and the end of that function. In the example above, this means that if another function existed in addition to main, the local variables declared in main could not be accessed from the other function and vice versa.

**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. You do not have to memorize all the content of this page. Most details are only provided to serve as a later reference in case you need it.

**1)Assignment (=)**

The assignment operator assigns a value to a variable.

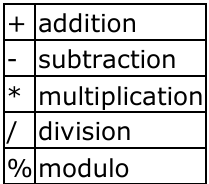


This statement assigns the integer value 5 to the variable a. The part at the left of the assignment operator (=) is known as the lvalue (left value) and the right one as the rvalue (right value). The lvalue has to be a variable whereas the rvalue can be either a constant, a variable, the result of an 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:



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

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



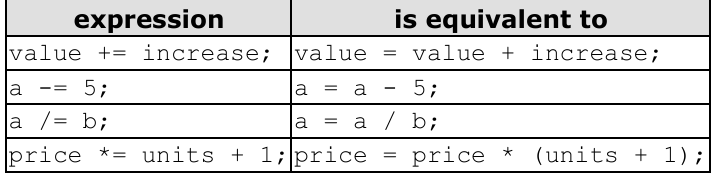
Operations of addition, subtraction, multiplication and division literally correspond with their respective mathematical operators. The only one that you might not be so used to see is modulo; whose operator is the percentage sign (%). Modulo is the operation that gives the remainder of a division of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



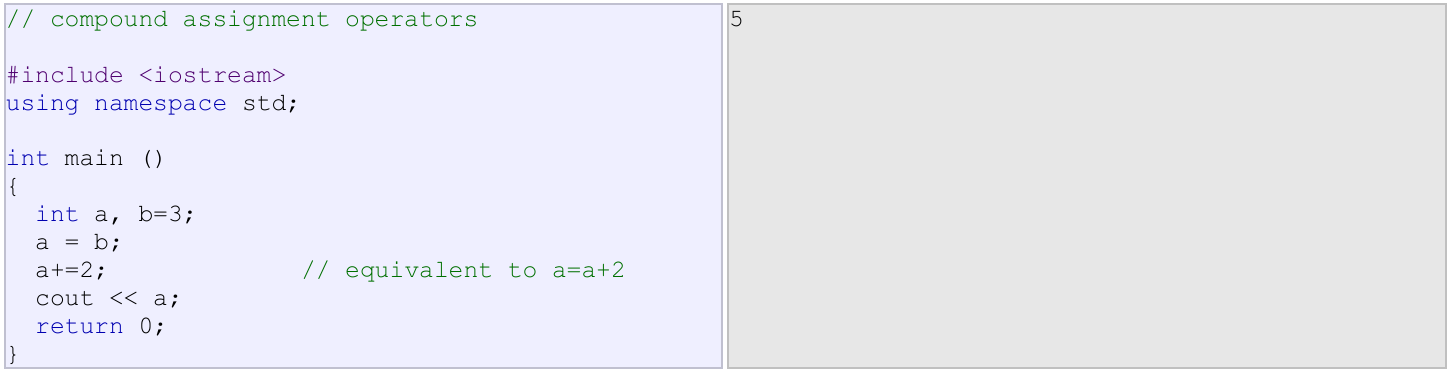
the variable a will contain the value 2, since 2 is the remainder from dividing 11 between 3.

**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:



and the same for all other operators. For example:



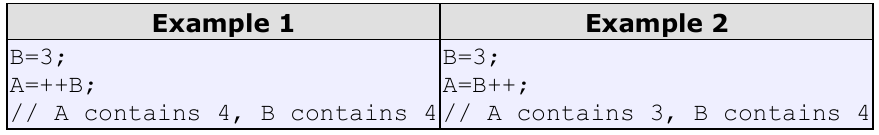
**4)Increase and decrease (++, --)**

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:



are all equivalent in its functionality: the three of them increase by one the value of c.

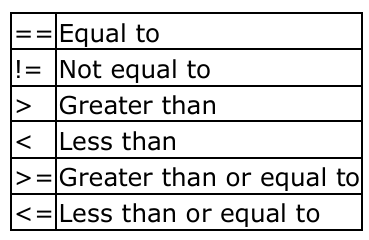
A characteristic of this operator is that it can be used both as a prefix and as a suffix. That means that it can be written either before the variable identifier (++a) or after it (a++). Although in simple expressions like a++ or ++a both have exactly the same meaning, in other expressions in which 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:



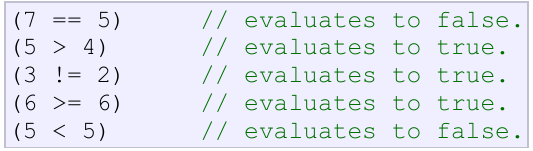
In Example 1, B is increased before its value is copied to A. While in Example 2, the value of B is copied to A and then B is increased.

**5)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++:

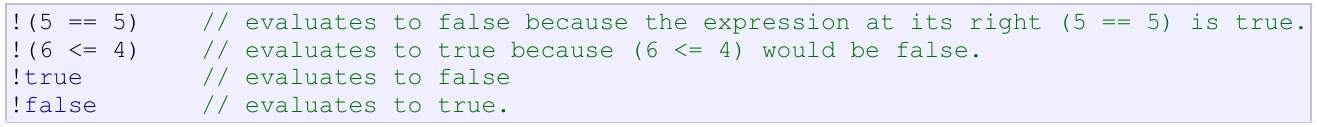


Here there are some examples:

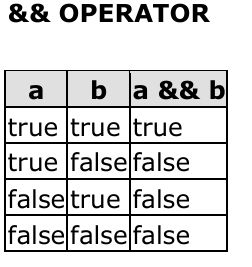


**6)Logical operators ( !, &&, || )**

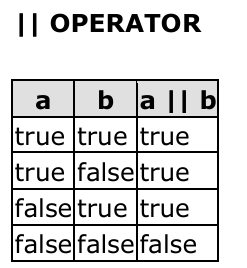
The Operator ! is the C++ operator to perform the Boolean operation NOT, it has only one operand, located at its right, and the only thing that it does is to inverse the value of it, producing false if its operand is true and true if its operand is false. Basically, it returns the opposite Boolean value of evaluating its operand. For example:



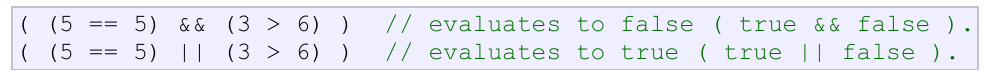
The logical operators && and || are used when evaluating two expressions to obtain a single relational result. The operator && corresponds with Boolean logical operation AND. This operation results true if both its two operands are true, and false otherwise. The following panel shows the result of operator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



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:



For example:

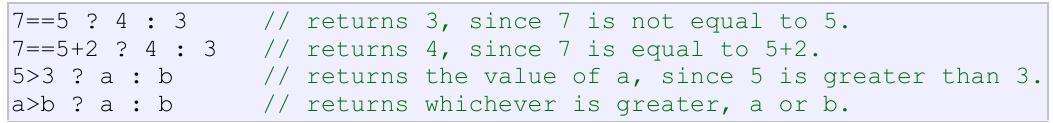


**7) Conditional operator ( ? )**

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

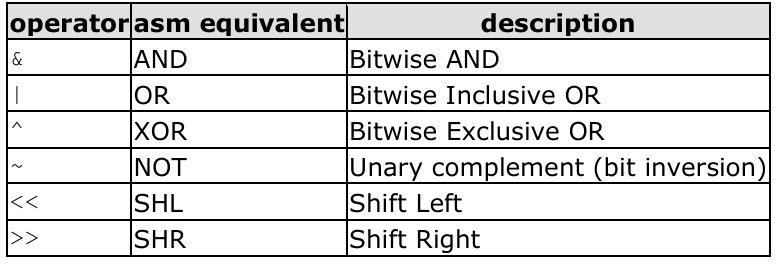
condition ? result1 : result2

If condition is true the expression will return result1, if it is not it will return result2.



**8) Bitwise Operators ( &, |, ^, ~, <> )**

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



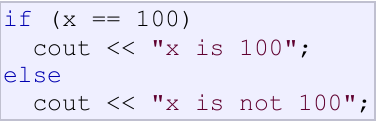
**Conditional structure: if and else**

The if keyword is used to execute a statement or block only if a condition is fulfilled. Its form is: if (condition) statement Where condition is the expression that is being evaluated. If this condition is true, statement is executed. If it is false, statement is ignored (not executed) and the program continues right after this conditional structure. For example, the following code fragment prints x is 100 only if the value stored in the x variable is indeed 100:



We can additionally specify what we want to happen if the condition is not fulfilled by using the keyword else. Its form used in conjunction with if is:

**if (condition) statement1 else statement2**



prints on the screen x is 100 if indeed x has a value of 100, but if it has not -and only if not- it prints out x is not 100.

**Iteration structures (loops)**

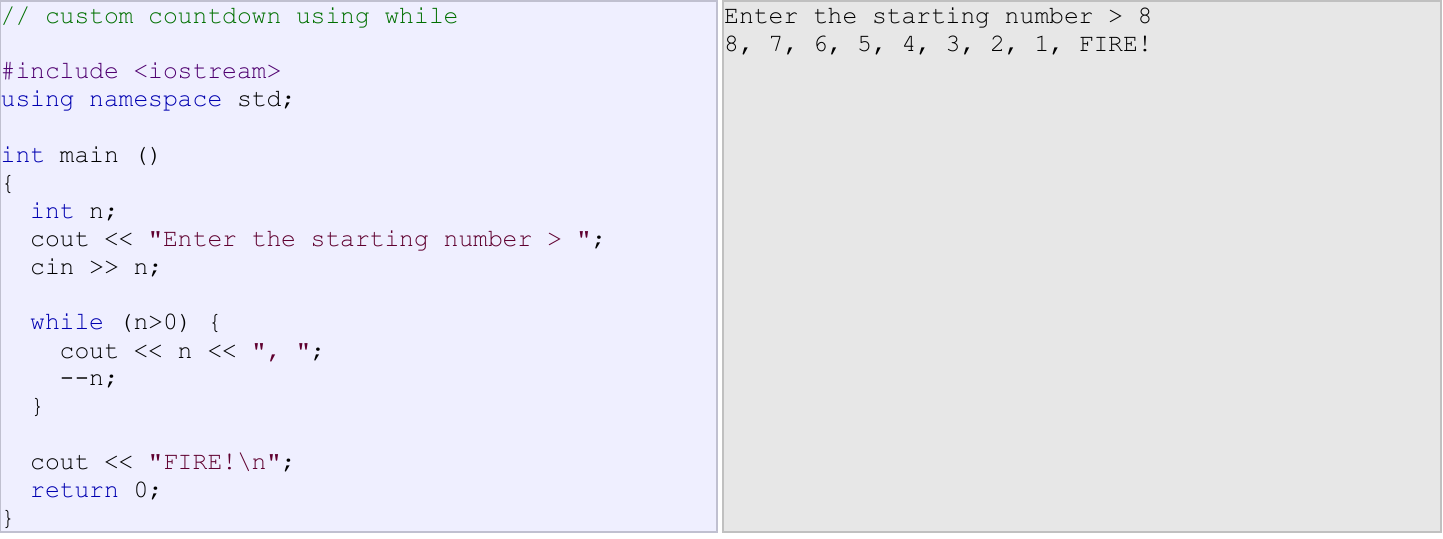
Loops have as purpose to repeat a statement a certain number of times or while a condition is fulfilled.

**The while loop**

Its format is:

**while (expression) statement**

and its functionality is simply to repeat statement while the condition set in expression is true. For example, we are going to make a program to countdown using a while-loop:



The whole process of the previous program can be interpreted according to the following script (beginning in main):

**1.** User assigns a value to n

**2**. The while condition is checked (n>0). At this point there are two posibilities:

\* condition is true: statement is executed (to step 3)

\* condition is false: ignore statement and continue after it (to step 5)

**3**. Execute statement:

cout << n << ", ";

n;

(prints the value of n on the screen and decreases n by 1)

**4**. End of block. Return automatically to step 2

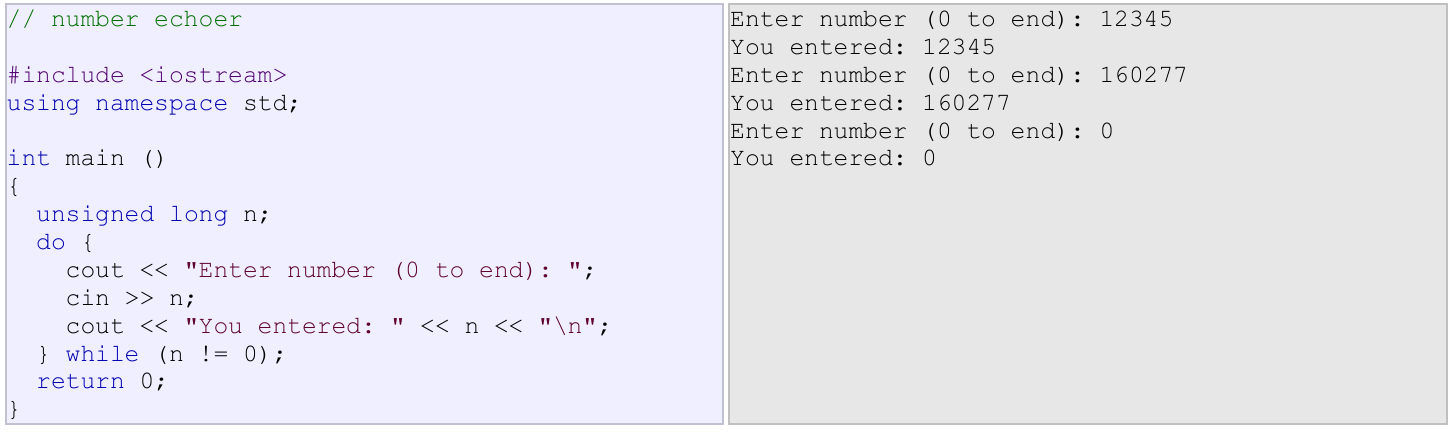
**5**. Continue the program right after the block: print FIRE! and end program.

**The do-while loop**

Its format is:

**do statement while (condition);**

Its functionality is exactly the same as the while loop, except that condition in the do-while loop is evaluated after the execution of statement instead of before, granting at least one execution of statement even if condition is never fulfilled. For example, the following example program echoes any number you enter until you enter 0.



The do-while loop is usually used when the condition that has to determine the end of the loop is determined within the loop statement itself, like in the previous case, where the user input within the block is what is used to determine if the loop has to end. In fact if you never enter the value 0 in the previous example you can be prompted for more numbers forever.

**The for loop**

Its format is:

**for (initialization; condition; increase) statement;**

and its main function is to repeat statement while condition remains true, like the while loop. But in addition, the for loop provides specific locations to contain an initialization statement and an increase statement. So this loop is specially designed to perform a repetitive action with a counter which is initialized and increased on each iteration.

It works in the following way:

**1.** initialization is executed. Generally it is an initial value setting for a counter variable. This is executed

only once.

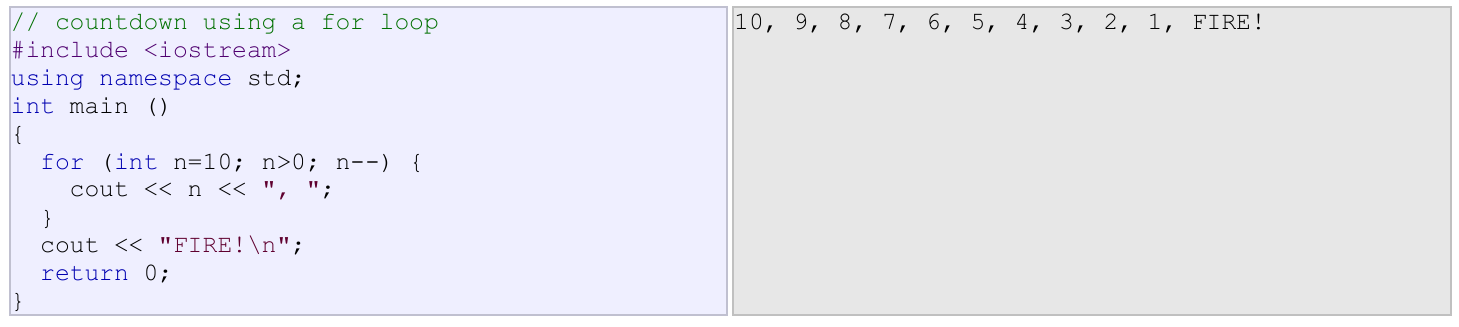
**2.** condition is checked. If it is true the loop continues, otherwise the loop ends and statement is skipped

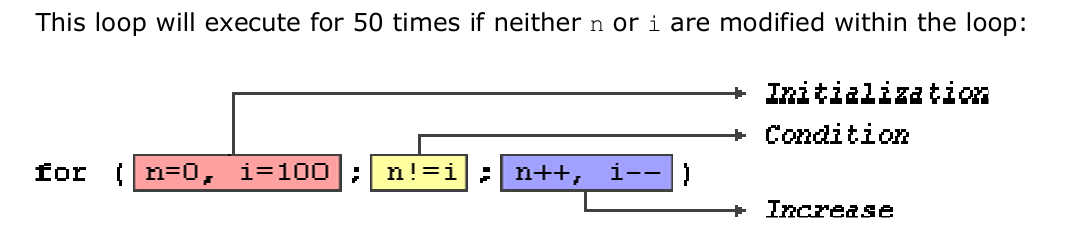
(not executed).

**3**. statement is executed. As usual, it can be either a single statement or a block enclosed in braces { }.

**4**. finally, whatever is specified in the increase field is executed and the loop gets back to step 2.

Here is an example of countdown using a for loop:





**Nested loops:**

Nested loops are loops within loops. They are often used when you need to perform a repetitive task within another repetitive task.

Eg:

#include <iostream>

using namespace std;

int main() {

for (int i = 1; i <= 3; i++) { // Outer loop

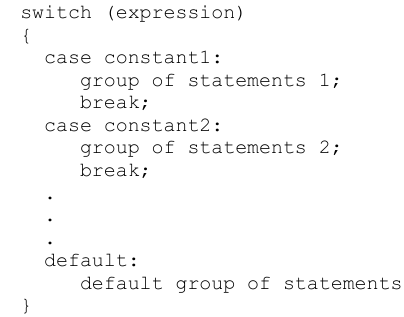
for (int j = 1; j <= 3; j++) { // Inner loop

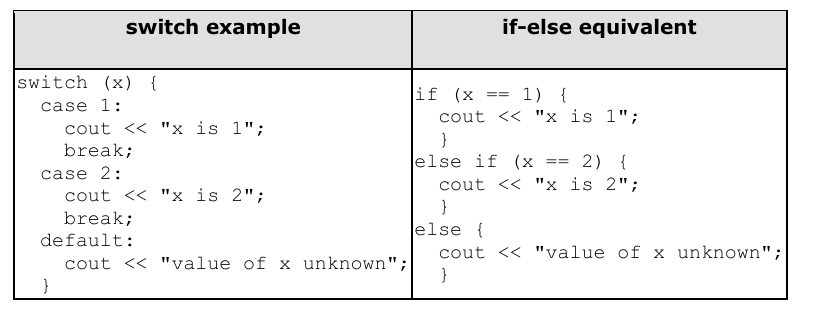
cout << "i = " << i << ", j = "

}

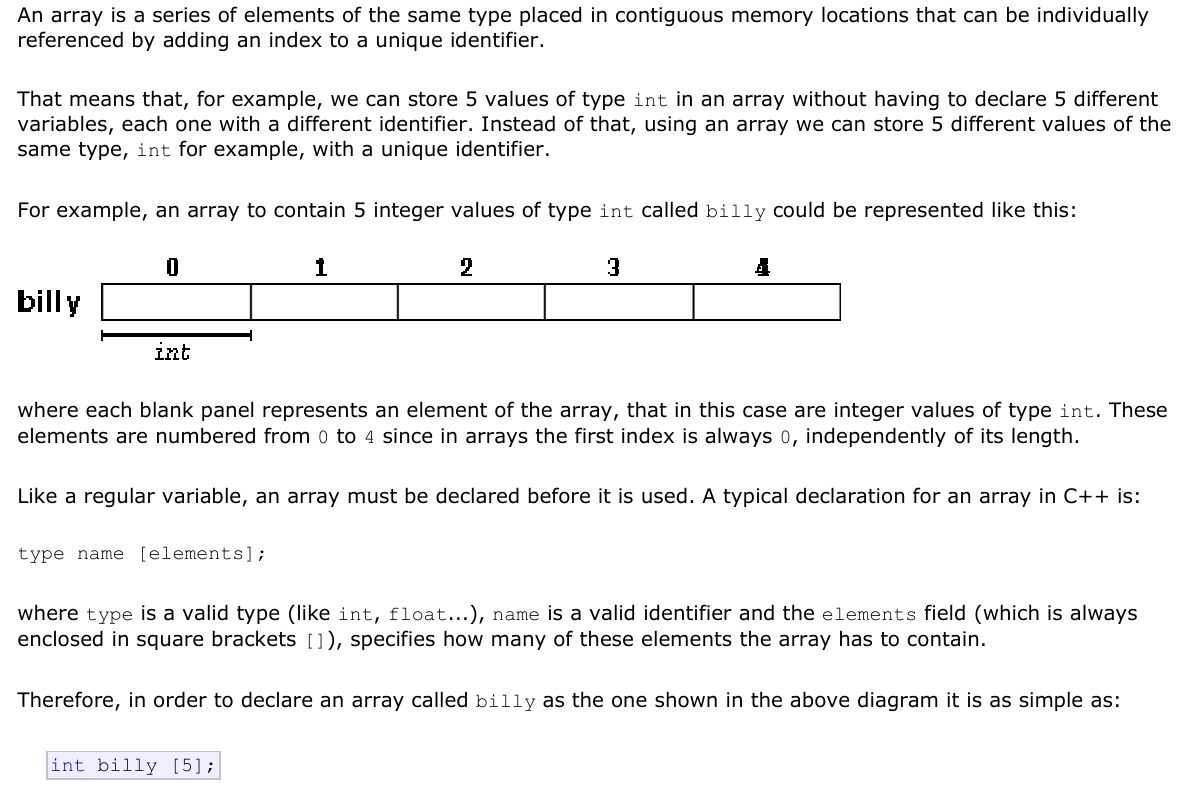
**Switch cases:**

The syntax of the switch statement is a bit peculiar. Its objective is to check several possible constant values for an expression. Something similar to what we did at the beginning of this section with the concatenation of several if and else if instructions. Its form is the following:

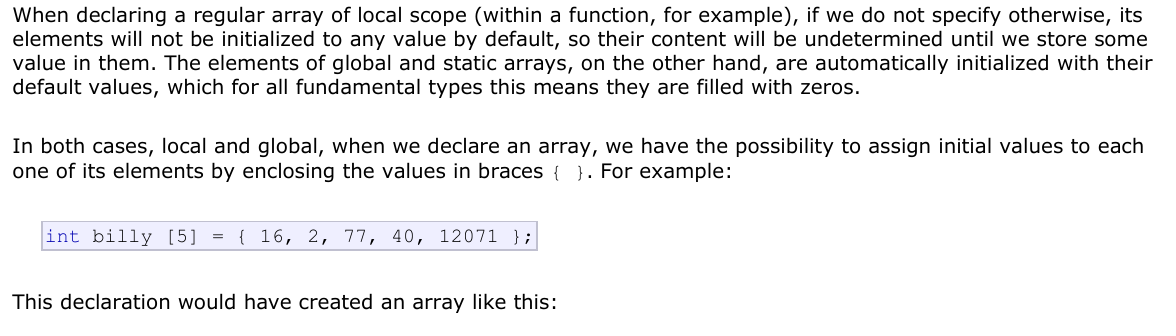


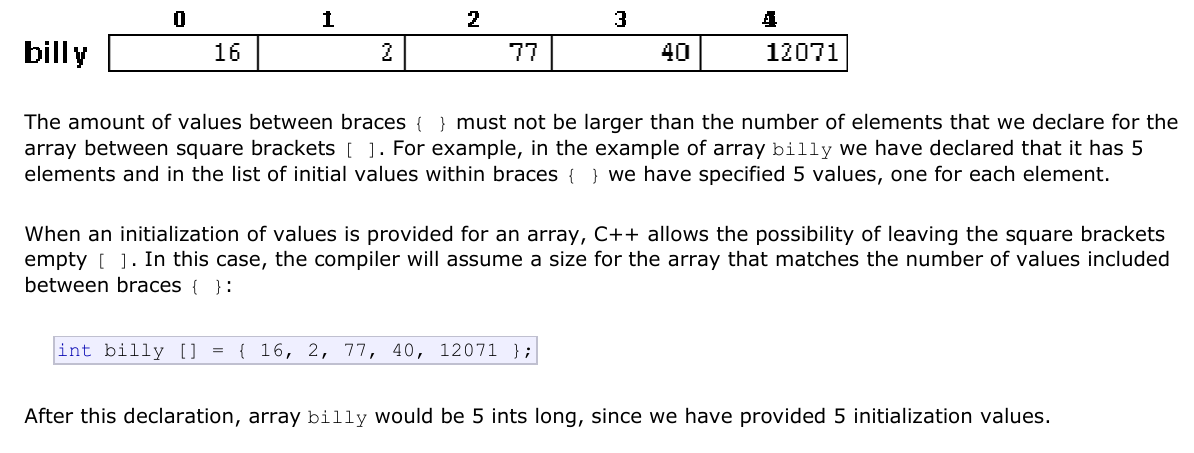


**Arrays:**

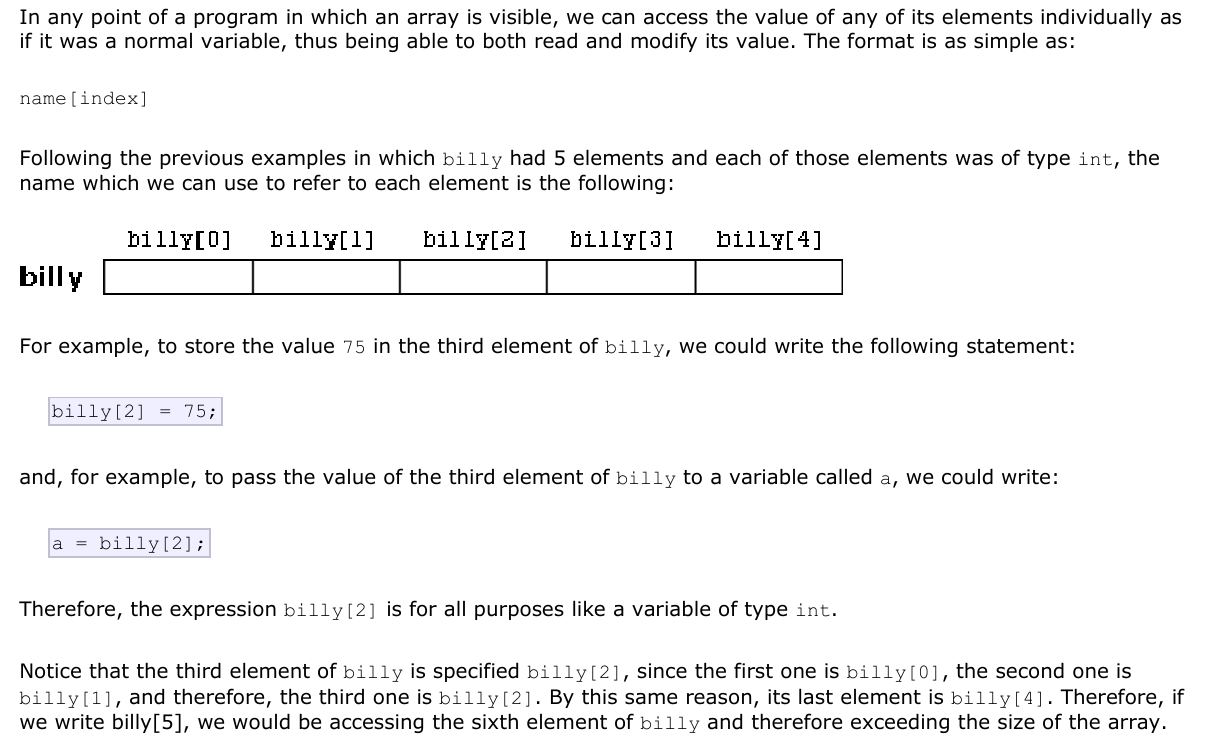
****

**Initializing arrays:**

****

****

**Accessing the values of an array:**

****

**2D ARRAYS:**

A 2D array, or two-dimensional array, is a data structure that stores elements in a grid-like format with rows and columns. It’s essentially an array of arrays, where each element is itself an array. Here’s a breakdown:

**Declaration**

To declare a 2D array, you specify the data type, the array name, and the number of rows and columns. For example, in C:

int array[3][4];

This creates a 2D array with 3 rows and 4 columns.

**Initialization**

You can initialize a 2D array in several ways. Here are two common methods:

1. **Using an initializer list:**

int array[3][4] = {

{1, 2, 3, 4},

{5, 6, 7, 8},

{9, 10, 11, 12}

};

1. **Using nested loops:**

int array[3][4];

for (int i = 0; i < 3; i++) {

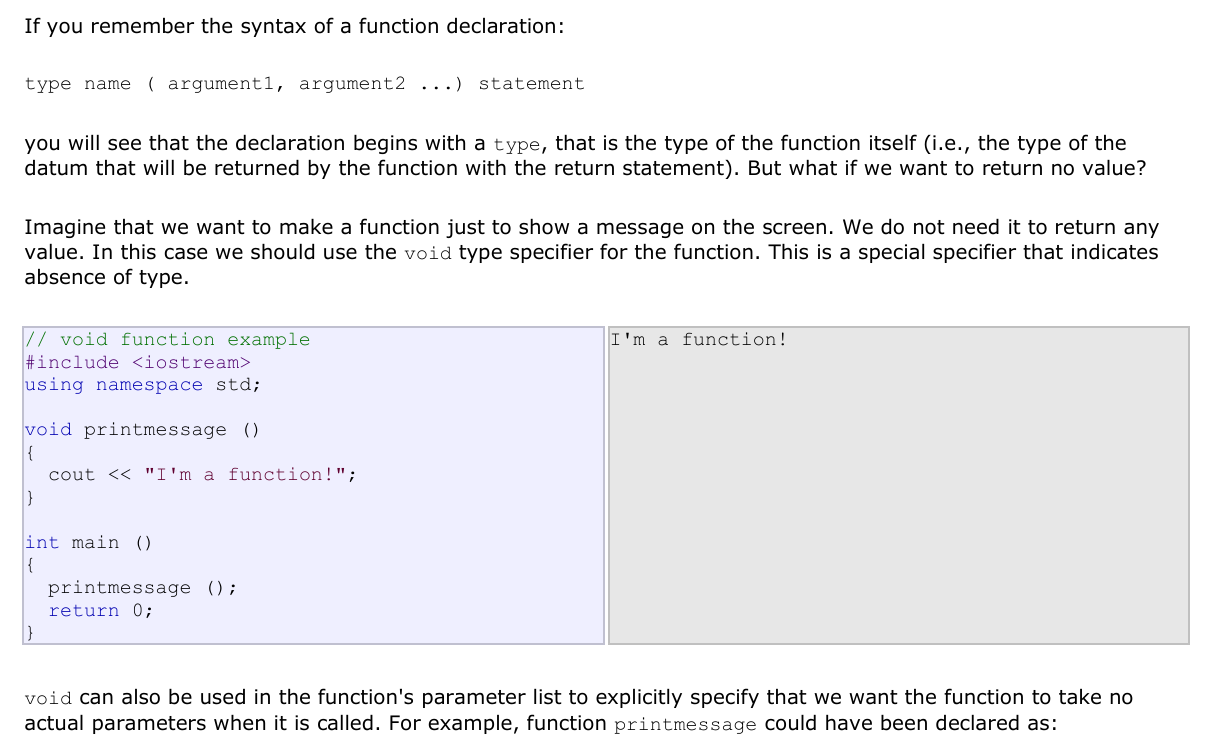
for (int j = 0; j < 4; j++) {

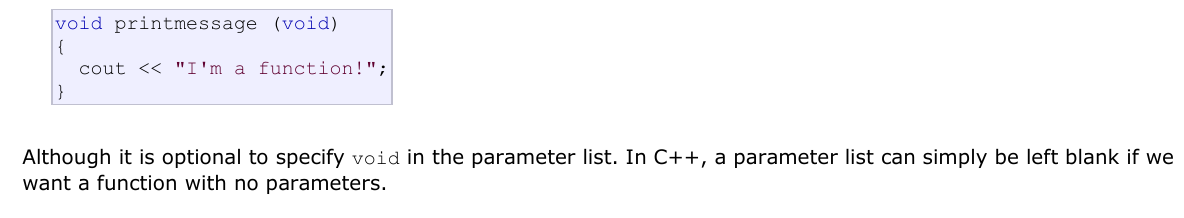
array[i][j] = i + j;

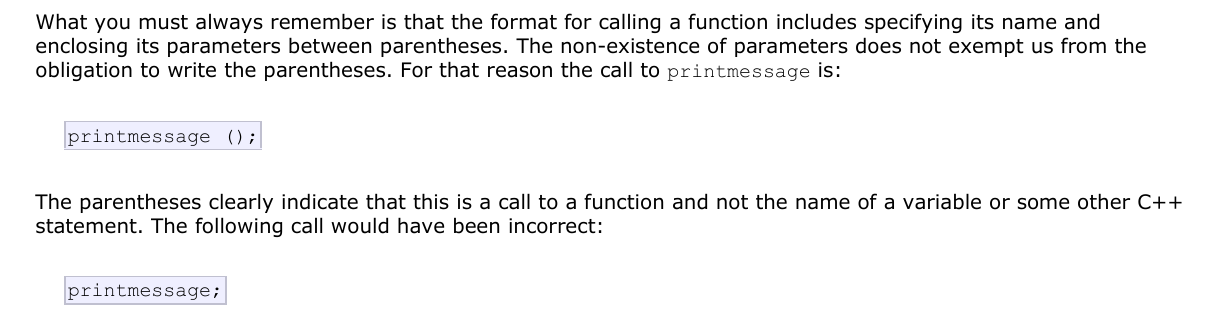
}

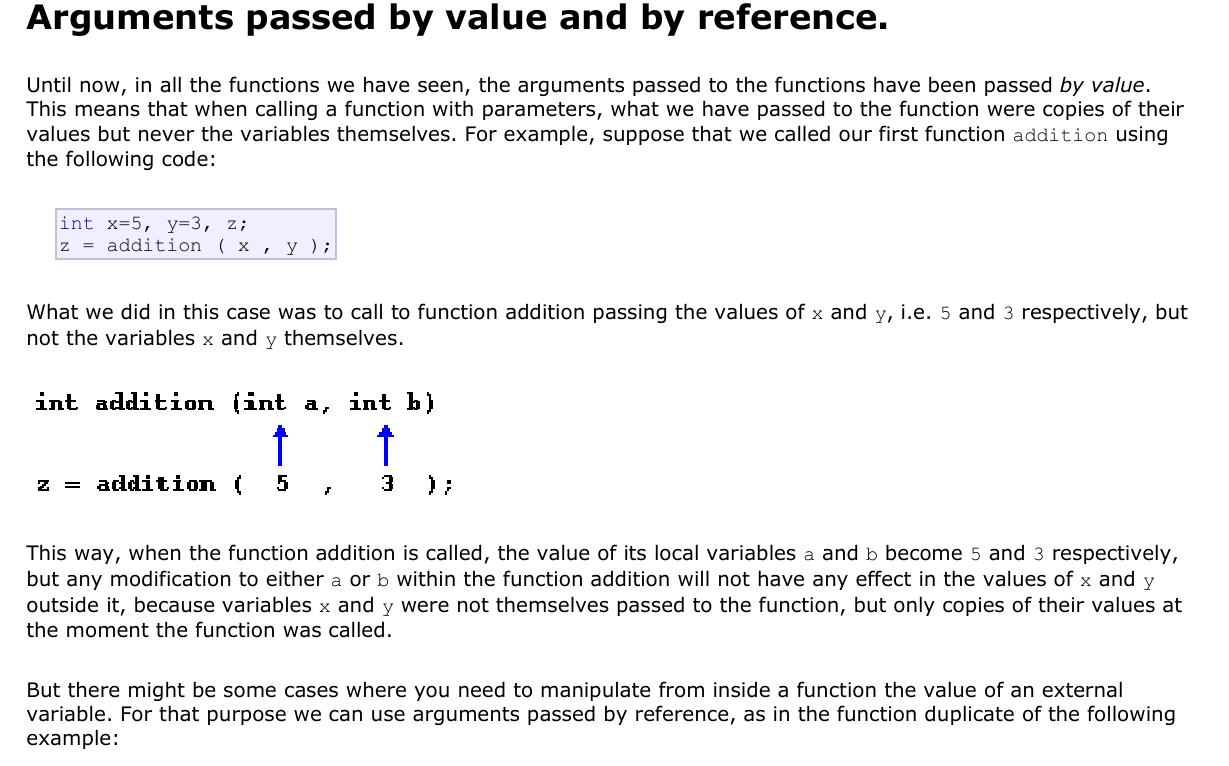
}

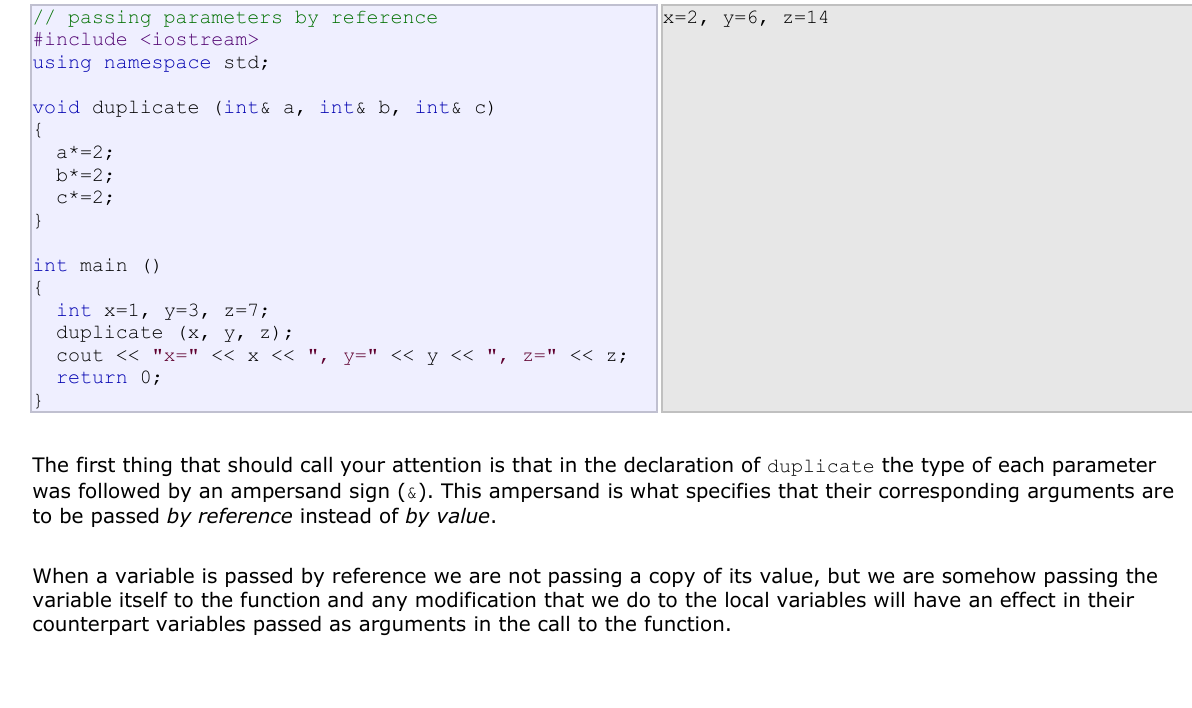
**Functions:**

****

****

****

****

****