**Functions.**

Functions allow to structure programs in segments of code to perform individual tasks.

In C++, a function is a group of statements that is given a name, and which can be called from some point of the program. The most common syntax to define a function is:

*type name ( parameter1, parameter2, ...) { statements }*

Where:  
- *type* is the type of the value returned by the function.  
- *name* is the identifier by which the function can be called.  
- *parameters* (as many as needed): Each parameter consists of a type followed by an identifier, with each parameter being separated from the next by a comma. Each parameter looks very much like a regular variable declaration (for example: *int x*), and in fact acts within the function as a regular variable which is local to the function. The purpose of parameters is to allow passing arguments to the function from the location where it is called from.

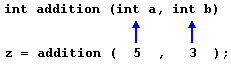
*- statements* is the function's body. It is a block of statements surrounded by braces {} that specify what the function actually does.

Let's have a look at an example:

|  |  |  |  |
| --- | --- | --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 | // function example  #include <iostream>  using namespace std;  int addition (int a, int b)  {  int r;  r= a + b;  return r;  }  int main ()  {  int z;  z = addition (5, 3);  cout << "The result is " << z;  } | The result is 8 | [Edit & Run](https://cplusplus.com/doc/tutorial/functions/) |

This program is divided in two functions: *addition* and *main*. Remember that no matter the order in which they are defined, a C++ program always starts by calling *main*. In fact, *main* is the only function called automatically, and the code in any other function is only executed if its function is called from *main* (directly or indirectly).

In the example above, *main* begins by declaring the variable *z* of type *int*, and right after that, it performs the first *function call*: it calls *addition*. The call to a function follows a structure very similar to its declaration. In the example above, the call to addition can be compared to its definition just a few lines earlier:



The parameters in the function declaration have a clear correspondence to the arguments passed in the function call. The call passes two values, 5 and 3, to the function; these correspond to the parameters *a* and *b*, declared for function addition.

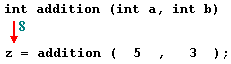
At the point at which the function is called from within main, the control is passed to function *addition*: here, execution of *main* is stopped, and will only resume once the *addition* function ends. At the moment of the *function call*, the value of both arguments (5 and 3) are copied to the local variables *int a* and *int b* within the function.

Then, inside *addition*, another local variable is declared (*int r*), and by means of the expression *r = a + b*, the result of *a* plus *b* is assigned to *r*; which, for this case, where *a* is 5 and *b* is 3, means that 8 is assigned to *r*.

The final statement within the function:

|  |  |  |
| --- | --- | --- |
|  | return r; |  |

Ends function *addition*, and returns the control back to the point where the function was called; in this case: to function *main*. At this precise moment, the program resumes its course on *main* returning exactly at the same point at which it was interrupted by the *call* to *addition*. But additionally, because *addition* has a *return type*, the *call* is evaluated as having a value, and this value is the value specified in the *return* statement that ended *addition*: in this particular case, the value of the local variable *r*, which at the moment of the *return statement* had a value of 8.



Therefore, the *call to addition* is an expression with the value returned by the function, and in this case, that value, *8*, is assigned to *z*. It is as if the entire function call (*addition(5,3)*) was replaced by the value it returns (i.e., 8).

Then main simply prints this value by calling:

|  |  |  |
| --- | --- | --- |
|  | cout << "The result is " << z; |  |

A function can actually be called multiple times within a program, and its argument is naturally not limited just to literals:

|  |  |  |  |
| --- | --- | --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 | // function example  #include <iostream>  using namespace std;  int subtraction (int a, int b)  {  int r;  r=a-b;  return r;  }  int main ()  {  int x=5, y=3, z;  z = subtraction (7,2);  cout << "The first result is " << z << '\n';  cout << "The second result is " << subtraction (7,2) << '\n';  cout << "The third result is " << subtraction (x,y) << '\n';  z= 4 + subtraction (x,y);  cout << "The fourth result is " << z << '\n';  } | The first result is 5  The second result is 5  The third result is 2  The fourth result is 6 | [Edit & Run](https://cplusplus.com/doc/tutorial/functions/) |

Similar to the addition function in the previous example, this example defines a *subtract* function, that simply returns the difference between its two parameters. This time, main calls this function several times, demonstrating more possible ways in which a function can be called.

Let's examine each of these calls, bearing in mind that each function call is itself an expression that is evaluated as the value it returns. Again, you can think of it as if the function call was itself replaced by the returned value:

|  |  |  |
| --- | --- | --- |
| 1 2 | z = subtraction (7,2);  cout << "The first result is " << z; |  |

If we replace the function call by the value it returns (i.e., 5), we would have:

|  |  |  |
| --- | --- | --- |
| 1 2 | z = 5;  cout << "The first result is " << z; |  |

With the same procedure, we could interpret:

|  |  |  |
| --- | --- | --- |
|  | cout << "The second result is " << subtraction (7,2); |  |

as:

|  |  |  |
| --- | --- | --- |
|  | cout << "The second result is " << 5; |  |

since 5 is the value returned by *subtraction (7,2)*.

In the case of:

|  |  |  |
| --- | --- | --- |
|  | cout << "The third result is " << subtraction (x,y); |  |

The arguments passed to *subtraction* are variables instead of literals. That is also valid, and

works fine. The function is called with the values x and y have at the moment of the call: 5 and 3 respectively, returning 2 as result.

The fourth call is again similar:

|  |  |  |
| --- | --- | --- |
|  | z = 4 + subtraction (x,y); |  |

The only addition being that now the function call is also an operand of an addition operation. Again, the result is the same as if the function call was replaced by its result: 6. Note, that thanks to the commutative property of additions, the above can also be written as:

|  |  |  |
| --- | --- | --- |
|  | z = subtraction (x,y) + 4; |  |

With exactly the same result. Note also that the semicolon does not necessarily go after the function call, but, as always, at the end of the whole statement. Again, the logic behind may be easily seen again by replacing the function calls by their returned value:

|  |  |  |
| --- | --- | --- |
| 1 2 | z = 4 + 2; // same as z = 4 + subtraction (x,y);  z = 2 + 4; // same as z = subtraction (x,y) + 4; |  |