C++ Functions

A function is a group of statements that together perform a task. A function is a block of code which only runs when it is called. You can pass data, known as parameters, into a function. Every C++ program has at least one function, which is **main()**, and all the most trivial programs can define additional functions.

Functions are used to perform certain actions, and they are important for reusing code: Define the code once and use it many times.

Create a Function

C++ provides some pre-defined functions, such as `main()`, which is used to execute code. But you can also create your own functions to perform certain actions.

To create (often referred to as *declare*) a function, specify the name of the function, followed by parentheses **()**:

Syntax

```
void myFunction() {
  // code to be executed
}
```

Example Explained

- `myFunction()` is the name of the function
- `void` means that the function does not have a return value.
- inside the function (the body), add code that defines what the function should do

Call a Function

Declared functions are not executed immediately. They are "saved for later use", and will be executed later, when they are called.

To call a function, write the function's name followed by two parentheses `()` and a semicolon `;`

In the following example, `myFunction()` is used to print a text (the action), when it is called:

Example

```
// Create a function
void myFunction() {
  cout << "I just got executed!";
}

int main() {
  myFunction(); // call the function
  return 0;
}

// Outputs "I just got executed!"
```

A function can be called multiple times:

Example

```
void myFunction() {
  cout << "I just got executed!\n";
}

int main() {
  myFunction();
  myFunction();
  myFunction();
  return 0;
}

// I just got executed!
// I just got executed!
// I just got executed!
```

Function Declaration and Definition

A C++ function consist of two parts:

- **Declaration:** the function's name, return type, and parameters (if any)
- **Definition:** the body of the function (code to be executed)

```
void myFunction() { // declaration
  // the body of the function (definition)
}
```

Note: If a user-defined function, such as `myFunction()` is declared after the `main()` function, **an error will occur**. It is because C++ works from top to bottom; which means that if the function is not declared above `main()`, the program is unaware of it:

Example

```
int main() {
  myFunction();
  return 0;
}

void myFunction() {
  cout << "I just got executed!";
}

// Error
```

However, it is possible to separate the declaration and the definition of the function - for code optimization.

You will often see C++ programs that have function declaration above `main()`, and function definition below `main()`. This will make the code better organized and easier to read:

Example

```
// Function declaration
void myFunction();
```

```
// The main method
int main() {
  myFunction();  // call the function
  return 0;
}

// Function definition
void myFunction() {
  cout << "I just got executed!";
}
```

C++ Function Parameters

Parameters and Arguments

Information can be passed to functions as a parameter. Parameters act as variables inside the function.

Parameters are specified after the function name, inside the parentheses. You can add as many parameters as you want, just separate them with a comma:

Syntax

```
void functionName(parameter1, parameter2, parameter3) {
  // code to be executed
}
```

The following example has a function that takes a `string` called **fname** as parameter. When the function is called, we pass along a first name, which is used inside the function to print the full name:

Example

```
void myFunction(string fname) {
  cout << fname << " Mensah\n";
}

int main() {
  myFunction("Kwame");
  myFunction("Jenny");
```

```
  myFunction("Anna");
  return 0;
}

// Kwame Mensah
// Jenny Mensah
// Anna Mensah
```

When a **parameter** is passed to the function, it is called an **argument**. So, from the example above: `fname` is a **parameter**, while `Kwame`, `Jenny` and `Anna` are **arguments**.

Default Parameter Value

You can also use a default parameter value, by using the equals sign (`=`).

If we call the function without an argument, it uses the default value ("Norway"):

Example

```
void myFunction(string country = "Norway") {
  cout << country << "\n";
}

int main() {
  myFunction("Sweden");
  myFunction("India");
  myFunction();
  myFunction("USA");
  return 0;
}

// Sweden
// India
// Norway
// USA
```

A parameter with a default value, is often known as an "**optional parameter**". From the example above, `country` is an optional parameter and `"Norway"` is the default value.

C++ Multiple Parameters

Multiple Parameters

Inside the function, you can add as many parameters as you want:

Example

```
void myFunction(string fname, int age) {
  cout << fname << " Refsnes. " << age << " years old. \n";
}

int main() {
  myFunction("Liam", 3);
  myFunction("Jenny", 14);
  myFunction("Anja", 30);
  return 0;
}

// Liam Refsnes. 3 years old.
// Jenny Refsnes. 14 years old.
// Anja Refsnes. 30 years old.
```

Note that when you are working with multiple parameters, the function call must have the same number of arguments as there are parameters, and the arguments must be passed in the same order.

Example

// Program to sum three integers

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

int sum (int a, int b, int c){
 return(a+b+c);
}
int main( ) {

      cout << " The ans is : " << sum(8,9,3)<< endl;
      return 0;
}
```

Example

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

int sum(int a, int b, int c) {
      int results;
      results = (a+b)-c;
            return results;
}
    int main () {
      int a, b, c;
      cout << "Enter first integer" << endl;
      cin >> a;

      cout << "Enter second integer" << endl;
      cin >> b;
      cout << "Enter third integer" << endl;
      cin >> c;

       int answer;
   answer = sum (a, b, c);
 cout << "The answer is  " << answer;
 return 0;
}
```

Example

// Program to calculate the Area of a triangle

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

int area (int height, int base){
 return((height * base)/2);
}
int main( ) {

      cout << " The ans is : " << area(8,8)<< endl;
```

```
    return 0;
}
```

Advantages of using functions in programming

- It makes it easier to maintain and debug program codes
- It enhances the ability to reuse codes by other programs
- It reduces the length of codes
- It makes codes more readable and clear.
- It reduces coding time
- Avoid repetition of codes.
- Divide a complex problem into many simpler problems.
- Reduce chances of error.
- Makes unit testing possible

Return Values

The `void` keyword, used in the previous examples, indicates that the function should not return a value. If you want the function to return a value, you can use a data type (such as `int`, `string`, etc.) instead of `void`, and use the `return` keyword inside the function:

Example

```
int myFunction(int x) {
  return 5 + x;
}

int main() {
  cout << myFunction(3);
  return 0;
}

// Outputs 8 (5 + 3)
```

This example returns the sum of a function with **two parameters**:

Example

```
int myFunction(int x, int y) {
  return x + y;
}

int main() {
  cout << myFunction(5, 3);
  return 0;
}

// Outputs 8 (5 + 3)
```

You can also store the result in a variable:

Example

```
int myFunction(int x, int y) {
  return x + y;
}

int main() {
  int z = myFunction(5, 3);
  cout << z;
  return 0;
}
// Outputs 8 (5 + 3)
```

Function Overloading

With **function overloading**, multiple functions can have the same name with different parameters:

Example

```
int myFunction(int x)
float myFunction(float x)
double myFunction(double x, double y)
```

Consider the following example, which have two functions that add numbers of different type:

Example

```
int plusFuncInt(int x, int y) {
  return x + y;
}

double plusFuncDouble(double x, double y) {
  return x + y;
}

int main() {

  cout << "Int: " << plusFuncInt(8, 5) << endl;
  cout << "Double: " << plusFuncDouble(4.3, 6.26) << endl;

  return 0;
}
```

Example

```
int plusFuncInt(int x, int y) {
  return x + y;
}

double plusFuncDouble(double x, double y) {
  return x + y;
}

int main() {
  int myNum1 = plusFuncInt(8, 5);
  double myNum2 = plusFuncDouble(4.3, 6.26);
  cout << "Int: " << myNum1 << "\n";
  cout << "Double: " << myNum2;
  return 0;
}
```

Instead of defining two functions that should do the same thing, it is better to overload one.

In the example below, we overload the `plusFunc` function to work for both `int` and `double`:

Example

```
int plusFunc(int x, int y) {
  return x + y;
}

double plusFunc(double x, double y) {
  return x + y;
}

int main() {
  int myNum1 = plusFunc(8, 5);
  double myNum2 = plusFunc(4.3, 6.26);
  cout << "Int: " << myNum1 << "\n";
  cout << "Double: " << myNum2;
  return 0;
}
```