C++ Functions

A function is a group of statements that together perform a task. Every C++ program has at least one function, which is **main()**, and all the most trivial programs can define additional functions.

You can divide up your code into separate functions. How you divide up your code among different functions is up to you, but logically the division usually is such that each function performs a specific task.

A function **declaration** tells the compiler about a function's name, return type, and parameters. A function **definition** provides the actual body of the function.

The C++ standard library provides numerous built-in functions that your program can call. For example, function **strcat()** to concatenate two strings, function **memcpy()** to copy one memory location to another location and many more functions.

A function is known with various names like a method or a sub-routine or a procedure etc.

Defining a Function

The general form of a C++ function definition is as follows −

```
return_type function_name( parameter list ) {
   body of the function
}
```

A C++ function definition consists of a function header and a function body. Here are all the parts of a function −

- **Return Type** − A function may return a value. The **return_type** is the data type of the value the function returns. Some functions perform the desired operations without returning a value. In this case, the return_type is the keyword **void**.
- **Function Name** − This is the actual name of the function. The function name and the parameter list together constitute the function signature.
- **Parameters** − A parameter is like a placeholder. When a function is invoked, you pass a value to the parameter. This value is referred to as actual parameter or argument. The parameter list refers to the type, order, and number of the parameters of a function. Parameters are optional; that is, a function may contain no parameters.
- **Function Body** − The function body contains a collection of statements that define what the function does.

Example

Following is the source code for a function called **max()**. This function takes two parameters num1 and num2 and return the biggest of both −

```
// function returning the max between two numbers

int max(int num1, int num2) {
   // local variable declaration
   int result;

   if (num1 > num2)
      result = num1;
   else
      result = num2;

   return result;
}
```

Function Declarations

A function **declaration** tells the compiler about a function name and how to call the function. The actual body of the function can be defined separately.

A function declaration has the following parts −

```
return_type function_name( parameter list );
```

For the above defined function max(), following is the function declaration −

```
int max(int num1, int num2);
```

Parameter names are not important in function declaration only their type is required, so following is also valid declaration −

```
int max(int, int);
```

Function declaration is required when you define a function in one source file and you call that function in another file. In such case, you should declare the function at the top of the file calling the function.

Calling a Function

While creating a C++ function, you give a definition of what the function has to do. To use a function, you will have to call or invoke that function.

When a program calls a function, program control is transferred to the called function. A called function performs defined task and when it's return statement is executed or when its function-ending closing brace is reached, it returns program control back to the main program.

To call a function, you simply need to pass the required parameters along with function name, and if function returns a value, then you can store returned value. For example −

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

// function declaration
int max(int num1, int num2);

int main () {
   // local variable declaration:
   int a = 100;
   int b = 200;
   int ret;

   // calling a function to get max value.
   ret = max(a, b);
   cout << "Max value is : " << ret << endl;

   return 0;
}

// function returning the max between two numbers
int max(int num1, int num2) {
   // local variable declaration
   int result;

   if (num1 > num2)
      result = num1;
   else
      result = num2;

   return result;
}
```

I kept max() function along with main() function and compiled the source code. While running final executable, it would produce the following result −

```
Max value is : 200
```

Function Arguments

If a function is to use arguments, it must declare variables that accept the values of the arguments. These variables are called the **formal parameters** of the function.

The formal parameters behave like other local variables inside the function and are created upon entry into the function and destroyed upon exit.

While calling a function, there are two ways that arguments can be passed to a function −

Sr.No	Call Type & Description
1	Call by Value This method copies the actual value of an argument into the formal parameter of the function. In this case, changes made to the parameter inside the function have no effect on the argument.
2	Call by Pointer This method copies the address of an argument into the formal parameter. Inside the function, the address is used to access the actual argument used in the call. This means that changes made to the parameter affect the argument.
3	Call by Reference This method copies the reference of an argument into the formal parameter. Inside the function, the reference is used to access the actual argument used in the call. This means that changes made to the parameter affect the argument.

By default, C++ uses **call by value** to pass arguments. In general, this means that code within a function cannot alter the arguments used to call the function and above mentioned example while calling max() function used the same method.

Default Values for Parameters

When you define a function, you can specify a default value for each of the last parameters. This value will be used if the corresponding argument is left blank when calling to the function.
This is done by using the assignment operator and assigning values for the arguments in the function definition. If a value for that parameter is not passed when the function is called, the default given value is used, but if a value is specified, this default value is ignored and the passed value is used instead. Consider the following example −

```
#include <iostream>
using namespace std;
int sum(int a, int b = 20) {
   int result;
   result = a + b;
   return (result);
}
int main () {
   // local variable declaration:
   int a = 100;
   int b = 200;
   int result;
   // calling a function to add the values.
   result = sum(a, b);
   cout << "Total value is :" << result << endl;
   // calling a function again as follows.
   result = sum(a);
   cout << "Total value is :" << result << endl;
   return 0;
}
```

When the above code is compiled and executed, it produces the following result −

```
Total value is :300
Total value is :120
```

C++ function call by value

The **call by value** method of passing arguments to a function copies the actual value of an argument into the formal parameter of the function. In this case, changes made to the parameter inside the function have no effect on the argument.

By default, C++ uses call by value to pass arguments. In general, this means that code within a function cannot alter the arguments used to call the function. Consider the function **swap()** definition as follows.

```
// function definition to swap the values.
void swap(int x, int y) {
   int temp;
   temp = x; /* save the value of x */
   x = y;    /* put y into x */
   y = temp; /* put x into y */
   return;
}
```

Now, let us call the function **swap()** by passing actual values as in the following example

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

// function declaration
void swap(int x, int y);
int main () {
   // local variable declaration:
   int a = 100;
   int b = 200;
   cout << "Before swap, value of a :" << a << endl;
   cout << "Before swap, value of b :" << b << endl;
   // calling a function to swap the values.

   swap(a, b);

   cout << "After swap, value of a :" << a << endl;
   cout << "After swap, value of b :" << b << endl;
   return 0;
}
```

When the above code is put together in a file, compiled and executed, it produces the following result −

```
Before swap, value of a :100
Before swap, value of b :200
After swap, value of a :100
After swap, value of b :200
```

Which shows that there is no change in the values though they had been changed inside the function.

C++ function call by pointer

The **call by pointer** method of passing arguments to a function copies the address of an argument into the formal parameter. Inside the function, the address is used to access the actual argument used in the call. This means that changes made to the parameter affect the passed argument.

To pass the value by pointer, argument pointers are passed to the functions just like any other value. So accordingly you need to declare the function parameters as pointer types as in the following function **swap()**, which exchanges the values of the two integer variables pointed to by its arguments.

```
// function definition to swap the values.
void swap(int *x, int *y) {
   int temp;
   temp = *x; /* save the value at address x */
   *x = *y; /* put y into x */
   *y = temp; /* put x into y */
   return;
}
```

To check the more detail about C++ pointers, kindly check C++ Pointers chapter.

For now, let us call the function **swap()** by passing values by pointer as in the following example −

```
#include <iostream>
using namespace std;
// function declaration
void swap(int *x, int *y);
int main () {
   // local variable declaration:
   int a = 100;
   int b = 200;
   cout << "Before swap, value of a :" << a << endl;
   cout << "Before swap, value of b :" << b << endl;
   /* calling a function to swap the values.
      * &a indicates pointer to a ie. address of variable a and
      * &b indicates pointer to b ie. address of variable b. */
   swap(&a, &b);
   cout << "After swap, value of a :" << a << endl;
   cout << "After swap, value of b :" << b << endl;
   return 0;
}
```

When the above code is put together in a file, compiled and executed, it produces the following result −

```
Before swap, value of a :100
Before swap, value of b :200
After swap, value of a :200
After swap, value of b :100
```

C++ function call by reference

The **call by reference** method of passing arguments to a function copies the reference of an argument into the formal parameter. Inside the function, the reference is used to access the actual argument used in the call. This means that changes made to the parameter affect the passed argument.

To pass the value by reference, argument reference is passed to the functions just like any other value. So accordingly you need to declare the function parameters as reference types as in the following function **swap()**, which exchanges the values of the two integer variables pointed to by its arguments.

```
// function definition to swap the values.
void swap(int &x, int &y) {
   int temp;
   temp = x; /* save the value at address x */
   x = y;    /* put y into x */
   y = temp; /* put x into y */
   return;
}
```

For now, let us call the function **swap()** by passing values by reference as in the following example −

```
#include <iostream>
using namespace std;
// function declaration
void swap(int &x, int &y);
int main () {
   // local variable declaration:
   int a = 100;
   int b = 200;

   cout << "Before swap, value of a :" << a << endl;
   cout << "Before swap, value of b :" << b << endl;

   /* calling a function to swap the values using variable
reference.*/

   swap(a, b);
   cout << "After swap, value of a :" << a << endl;
   cout << "After swap, value of b :" << b << endl;
   return 0;
}
```

When the above code is put together in a file, compiled and executed, it produces the following result −

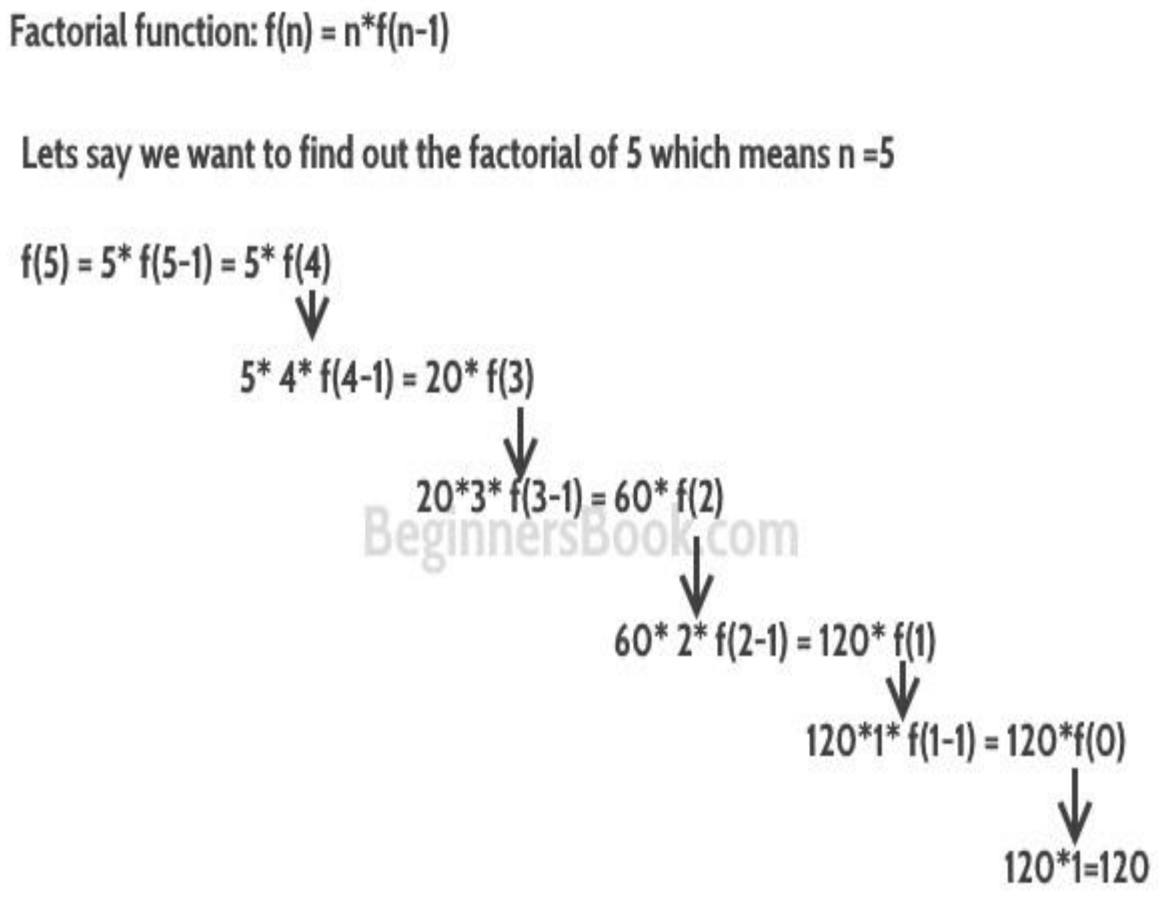
```
Before swap, value of a :100
Before swap, value of b :200
After swap, value of a :200
After swap, value of b :100
```

C++ Recursion

The process in which a function calls itself is known as recursion and the corresponding function is called the **recursive function**. The popular example to understand the recursion is factorial function.

Factorial function: f(n) = n*f(n-1), base condition: if n<=1 then f(n) = 1. Don't worry we wil discuss what is base condition and why it is important.

In the following diagram. I have shown that how the factorial function is calling itself until the function reaches to the base condition.


Factorial function: f(n) = n*f(n-1)
Lets say we want to find out the factorial of 5 which means n =5
f(5) = 5* f(5-1) = 5* f(4)
5* 4* f(4-1) = 20* f(3)
20*3* f(3-1) = 60* f(2)
BeginnersBook.com
60* 2* f(2-1) = 120* f(1)
120*1* f(1-1) = 120*f(0)
120*1=120

Lets solve the problem using C++ program.

C++ recursion example: Factorial

```
#include <iostream>
using namespace std;
//Factorial function
int fact(int n){
   /* This is called the base condition, it is
    * very important to specify the base condition
    * in recursion, otherwise your program will throw
    * stack overflow error.
    */
   if (n <= 1)
        return 1;
   else
       return n*fact(n-1);
}
int main(){
   int num;
   cout<<"Enter a number: ";
   cin>>num;
   cout<<"Factorial of entered number: "<<fact(num);
   return 0;
}
```

Output:

```
Enter a number: 5
Factorial of entered number: 120
```

Base condition

In the above program, you can see that I have provided a base condition in the recursive function. The condition is:

```
if (n <= 1)
        return 1;
```

The purpose of recursion is to divide the problem into smaller problems till the base condition is reached. For example in the above factorial program I am solving the factorial function f(n) by calling a smaller factorial function f(n-1), this happens repeatedly until the n value reaches base condition(f(1)=1). If you do not define the base condition in the recursive function then you will get stack overflow error.

Direct recursion vs indirect recursion

Direct recursion: When function calls itself, it is called direct recursion, the example we have seen above is a direct recursion example.

Indirect recursion: When function calls another function and that function calls the calling function, then this is called indirect recursion. For example: function A calls function B and Function B calls function A.

Indirect Recursion Example in C++

```
#include <iostream>
using namespace std;
int fa(int);
int fb(int);
int fa(int n){
   if(n<=1)
      return 1;
   else
      return n*fb(n-1);
}
int fb(int n){
   if(n<=1)
      return 1;
   else
      return n*fa(n-1);
}
int main(){
   int num=5;
   cout<<fa(num);
   return 0;
}
```

Output:

```
120
```