A function in C++ is a set of statements clubbed together that performs a specific task. The function body is only executed when we call the function. Every C++ program contains at least one function, that is the main function. Program execution starts from the first line of the main function. Creating a function increases reusability and modularity in the program. The whole code can be divided logically into separate functions where each function does a specific task. A function is also called a subroutine or method.

There are two types of functions:

1. User-defined functions
2. Inbuilt library functions

**Scope**

* In this article, we will learn about the requirements and advantages of functions.
* We will see the difference between User-defined and In-built library functions.
* We will also learn how to create and call a function.

**Why do We Need Functions in C++?**

**Reusability** is the main requirement of functions. Functions in C++ are the basic building blocks of a program. As the program can grow to thousands of lines of code, we cannot write the complete code in one file. The code should be broken into smaller, maintainable, and reusable chunks. These reusable chunks are functions.

For example: Suppose we need to calculate the factorial of three numbers. Without using functions, we have to write the logic for calculating factorial three times. If we use functions, then we have to write the logic only once while creating the function, and we can call the function as many times as we want.

Let's understand this with an example.

**Code for Calculating the Factorial of Three Numbers Without using Functions:**

#include<iostream>

using namespace std;

int main()

{

int num1=1;

int fact1= 1;

for(int i=1; i<=num1; i++)

fact1\*= i;

*// Factorial of first number*

cout<<"Factorial of 1 : "<<fact1<<endl;

int num2=2;

int fact2= 1;

for(int i=1; i<=num2; i++)

fact2\*= i;

*// Factorial of second number*

cout<<"Factorial of 2 : "<<fact2<<endl;

int num3=3;

int fact3= 1;

for(int i=1; i<=num3; i++)

fact3\*= i;

*// Factorial of third number*

cout<<"Factorial of 3 : "<<fact3<<endl;

}

**Output**

Factorial of 1: 1

Factorial of 2: 2

Factorial of 3: 6

Here we can see that, for calculating the factorial of three numbers we have written logic for calculating factorial three times. Similarly, if we need to calculate factorial of �*N* numbers (without using functions) then we need to write factorial logic �*N* number of times.

*Code for calculating the factorial of three numbers using functions :*

#include<iostream>

using namespace std;

*// Function for calculating Factorial*

int calculateFactorial( int num)

{

int fact=1;

for(int i=1;i<=num;i++)

fact\*=i;

return fact;

}

int main()

{

*// Calling the function to calculate factorial of 2, 3 and 4*

cout<<"Factorial of 1 : "<<calculateFactorial(1)<<endl;

cout<<"Factorial of 2 : "<<calculateFactorial(2)<<endl;

cout<<"Factorial of 3 : "<<calculateFactorial(3)<<endl;

}

**Output**

Factorial of 1: 1

Factorial of 2: 2

Factorial of 3: 6

Here we can see that factorial logic is written only once and the function for calculating factorial is called three times. We have reused the logic of calculating factorial and also reduced the number of lines of code. Similarly, if we need to calculate factorial of �*N* numbers, we only need to write the logic once and we can call the function �*N* times.

We will learn how to define and call a function in C++ in detail later in this article.

**Advantages of Functions in C++**

1. Code Readability and Maintainability : Large programs are difficult to understand and maintain. Dividing the code into multiple smaller functions makes the code modular. Modular code is easy to organize, read and maintain.
2. Easy debugging: When the program is divided into functions, we can test and debug the functions individually. As each function has different functionality, it makes the complete debugging process easier.
3. Code reusability : We do not need to write the logic for the same task multiple times, as we can call the function whenever required. We can understand it from the factorial example given above. Whenever we need to calculate the factorial of a number, we can just call the function. We do not need to write the logic again and again.
4. Reduces Code Size : As functions are reusable, duplicate lines of code for the same logic are eliminated which reduces the size of the program.

We can understand this from the above factorial example. In the code without using functions, factorial calculation logic is written three times. While in the code using functions factorial calculation logic is written only once, which directly reduces multiple lines of code.

We can call the function from anywhere in the program any number of times as per the requirement.

**Types of Functions in C++**

There are two types of functions in C++:

Explore free courses by our top instructors[View All](https://www.scaler.com/topics/courses/)

TARUN LUTHRA

Java Course - Mastering the Fundamentals

55k+ enrolled

RAHUL JANGHU

Python Certification Course: Master the essentials

41k+ enrolled

PRATEEK NARANG

C++ Course: Learn the Essentials

29k+ enrolled

35,262+ learners have attended these Courses.

**Built-in Functions**

Built-in functions are standard library functions in C++. These functions are already defined in [C++ header files](https://www.scaler.com/topics/cpp/cpp-header-files/) and STL ( Standard template library ). Header files are special files with .h extension. We have to include the header file of the function before calling it using the *include* directive.

For example: pow() function in C++ is defined in **math.h** header file. It returns the result of the first argument raised to the power of the second argument.

#include <iostream>

#include <math.h>

using namespace std;

int main() {

*// Calculates 2 raised to the power 3*

cout << pow(2, 3);

return 0;

}

**Output**

8

**User-defined Functions**

Functions created by the user for custom requirements are called user-defined functions. A user-defined function in C++ performs a specific task and can be called multiple times from anywhere in the program. A program can have multiple user-defined functions.

A user-defined function has:

* Function name
* Return type
* Function parameters
* Function body

Function body contains the code statements that perform the required task. The function body is only executed when the function is called.

Let's see how we can create and call user-defined functions in C++.

**Function Declaration in C++**

The function declaration is made to tell the compiler about the existence of the function. It is also called *function signature* or *function prototype.*

Function prototype tells the compiler about:

* Return type
* Number of parameters
* Type of parameters
* Name of the function
* It also tells the order in which arguments are passed to the function.

In the function prototype, the function body or logic is not defined. This is the difference between function declaration and function definition. In the function declaration, the only prototype is defined while in function definition function body or logic is also defined.

**Syntax for Declaring a Function:**

returnType functionName( parameter1, parameter2,...parameterN ) ;

**Function Name**

Every user-defined function in C++ has a unique name, no two functions can have the same name. It is recommended to name the functions in context with the task it performs.

For example, we can give the name 'calculateFactorial' to the function for calculating the factorial of a given number.

Let's understand the terms return type and parameters one by one.

**Function Return Type in C++**

The return type is the type of value returned by the function. A function in C++ may or may not return a value. If the function does not return a value then its return type is void. Value is returned from a function using the return statement. Control is transferred back to the caller when the **return** statement is executed. If the function returns a value then we need to specify its data type, like int, char, or float.

*Note: Only one value can be returned from a function in C++. It is mandatory to return a value for functions with a non-void return type*.

For example, consider a function *calculateFactorial* which calculates the factorial of a number and returns an integer value. We can see its return type is *int*.

int calculateFactorial( int num)

{

int fact=1;

for(int i=1;i<=num;i++)

fact\*=i;

return fact;

}

**Parameter Passing to Functions in C++**

Parameters define the number and data types of input the function can have. A function in C++ can have zero or multiple parameters.

Values are passed to the function during the function call. These values are used in the function body to process results.

The following information is defined in the parameter list :

* Data type of parameter
* Number of parameters
* Order of values that are passed to the function

For example: Consider a function *calculateSum* which takes an array and its length as input and calculates the sum of all elements. Here we can see that the first parameter is an integer and the second parameter is an array.

int calculateSum( int len, int arr[])

{

'''

'''

'''

}

Let's understand two important terms about parameters :

1. **Formal parameters:** The type and name of parameters defined in the parameter list are called formal parameters.

int calculateSum( int len, int arr[])

*// Here len and arr are formal parameters.*

1. **Actual parameters:** The data or variables passed to the function at the time of calling are called actual parameters. We will learn more about function calling later in the article.

void main()

{

int len=3;

int arr[3]={1, 2, 3};

calculateSum(len,arr);

*// Here len and arr are actual parameters.*

}

**C++ Function Definition**

The function definition is the most important part. It contains the return type, function name, parameter list, and the actual logic of the function. The statements in the function body are executed only when the function is called.

In C++ language, the function body is enclosed in curly braces.

**Syntax for function Definition:**

returnType functionName( parameter1, parameter2,...parameterN )

{

statement1;

statement2;

.

.

.

statementN;

}

C++ language allows us to declare and define the function at the same time.

Note that in the function declaration, it is not important to specify the name of parameters but in the function definition, it is mandatory to write names with the data type.

To understand better, let's look at an example. Here is how we can write a function for calculating the sum of the elements of an array.

int calculateSum( int len, int arr[])

{

int sum=0;

for(int i=0;i<len;i++)

sum+=arr[i];

return sum;

}

**Calling a Function in C++**

During function definition and declaration, we have specified what will the function do. To execute the function we need to call the function.

When a function is called, the control is transferred to the function. After the function gets the control, the statements in the function body are executed. Control is transferred back to the place of function call when a *return* statement is executed or the function body ends.

**Note:** As functions are defined in global space, we can call a function from anywhere in the program.

We can call a function by passing arguments( if there are any) with the function name.

For example, We can call the function calculateSum as follows

int calculateSum( int len, int arr[])

{

int sum=0;

for(int i=0;i<len;i++)

sum+=arr[i];

return sum;

}

void main()

{

int len=3;

int arr[ 3]= {1, 2, 3 };

calculateSum(len, arr ); *// Calling the function*

}

There are two ways to call a function :

**Function Call by Value**

When a function is called by value, changes to the arguments are not reflected outside the function. In this method, actual arguments are copied to formal arguments. The arguments passed to the function are used for performing the task inside the function. Sometimes the value of arguments is changed while performing the task.

In the function call by value method, the arguments retain their original value outside the function. If any changes are made to the arguments inside the function, those changes do not affect the values of arguments outside the function.

For example, consider a function that swaps two numbers. Here we can see that values of num1 and num2 are only changed inside the function. There are no effects on the values of num1 and num2 outside the swap function as the function was called by value.

#include<iostream>

using namespace std;

void swap( int num1, int num2)

{

int temp=num1;

num1=num2;

num2=temp;

*// values of num1 and num2 are swapped inside the function*

cout<<"Inside function: "<<num1<<" "<<num2<<endl;

}

int main()

{

int num1 = 1, num2 = 2;

swap(num1,num2); *// function call by value*

*// values of num1 and num2 were swapped inside the function but*

*// here (outside the function) they have their original values*

cout<<"Outside function: "<<num1<<" "<<num2<<endl;

}

**Output**

Inside function: 2 1

Outside function: 1 2

**Function Call by Reference**

In this method, the address of actual arguments is passed to formal arguments.

The address of parameters is used to access their value. Changes made to the arguments in the function are reflected outside the function too. As we are not passing a copy of actual arguments, any changes made to the arguments during function processing will be made at the actual memory location of the arguments. When the value of a variable is changed at its actual memory location, it is updated everywhere (both inside and outside the function).

For example, let's understand this with the same swap function from the example above.  
We can pass the address of variables using the **&** operator. As the address of a variable is stored in pointers, we will have the parameters of pointer type. We can see that the values of *num1* and *num2* are also changed outside the *swap* function.

#include<iostream>

using namespace std;

void swap( int \*num1, int \*num2)

{

int temp=\*num1;

\*num1=\*num2;

\*num2=temp;

cout<<"Inside function: "<<\*num1<<" "<<\*num2<<endl;

}

int main()

{

int num1 =1, num2 =2;

swap(&num1,&num2); *// function call by reference*

cout<<"Outside function: "<<num1<<" "<<num2<<endl;

}

**Output**

Inside function: 2 1

Outside function: 2 1

**Default Value for Parameters**

When we declare a function in C++, we can specify a default value for parameters. The default value can be assigned using the assignment operator (=).

If a value is not passed during the function call then the default value is used for the corresponding parameter. If the value is specified during the function call then the default value is ignored and the passed value is used.

For example: Consider a function sum which calculates the sum of two numbers. We can see that the default value is used when arguments are not passed during the function call. The default value is ignored when arguments are passed during the function call.

#include<iostream>

using namespace std;

int sum(int x=2, int y=0) *// Here there are two values in the default arguments*

{

return x+y;

}

int main()

{

cout << sum(1, 2) << endl; *// x= 1, y =2 (default values ignored)*

cout << sum(1) << endl; *// x= 1, y =0 (default value for y is used)*

cout << sum() << endl; *// x= 2, y =0 (default values for both x and y are used)*

return 0;

}

**Benefits of User-Defined Functions in C++**

* We can create functions in C++ for custom requirements using user-defined functions. For example: Suppose we need to calculate the factorial of a number. There is no built-in function for calculating factorial, so we can create a user-defined function for that.
* User-defined functions are **reusable**, which helps in reducing code size.
* User-defined functions let us divide the program into smaller modules. Modular code is more readable, easier to understand and debug.
* Workload can be divided among developers as they can work on different functions.

**Example:**

To understand functions in C++ better let's look at the example given below :

#include <iostream>

#include <math.h>

using namespace std;

*// User-defined function for finding factors of a number*

void factorise( int num)

{

for(int i = 1; i <= num; i++)

if (num % i == 0)

cout<<i<<" ";

}

int main()

{

*// Calculates 2 raised to the power 3*

*// In-built library function*

cout << pow(2, 3)<<endl;

*// Calling function factorise*

factorise(10);

return 0;

}

**Output**

8

1 2 5 10

We can see that there is a user-defined function factorise() which takes an integer as input and prints its factors. When factorise() function is called control is transferred from main function to factorise() function.

We can also see that there is an in-built library function pow() which calculates the value of the first argument raised to the power value of the second argument.

We now understand that both user-defined functions and in-built library functions have different use cases. Library functions have the functionality already defined in header files and C++ libraries. User-defined functions provide more flexibility in terms of implementation so that we can create functions according to our requirements.

**Conclusion**

* Functions are the basic building blocks of a program. They make the code reusable, shorter, and more readable.
* Every C++ program has at least one function that is the *main* function. Code execution starts from the main function.
* We can define multiple functions in the program and we can call every function multiple times as per our requirements.