**C++ Functions**

In this tutorial, we will learn about the C++ function and function expressions with the help of examples.

A function is a block of code that performs a specific task.

Suppose we need to create a program to create a circle and color it. We can create two functions to solve this problem:

* a function to draw the circle
* a function to color the circle

Dividing a complex problem into smaller chunks makes our program easy to understand and reusable.

There are two types of function:

1. **Standard Library Functions:** Predefined in C++
2. **User-defined Function:** Created by users

In this tutorial, we will focus mostly on user-defined functions.

**C++ User-defined Function**

C++ allows the programmer to define their own function.

A user-defined function groups code to perform a specific task and that group of code is given a name (identifier).

When the function is invoked from any part of the program, it all executes the codes defined in the body of the function.

**C++ Function Declaration**

The syntax to declare a function is:

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

// function body

}

Here's an example of a function declaration.

// function declaration

void greet() {

cout << "Hello World";

}

Here,

* the name of the function is greet()
* the return type of the function is void
* the empty parentheses mean it doesn't have any parameters
* the function body is written inside {}

**Note:** We will learn about returnType and parameters later in this tutorial.

**Calling a Function**

In the above program, we have declared a function named greet(). To use the greet() function, we need to call it.

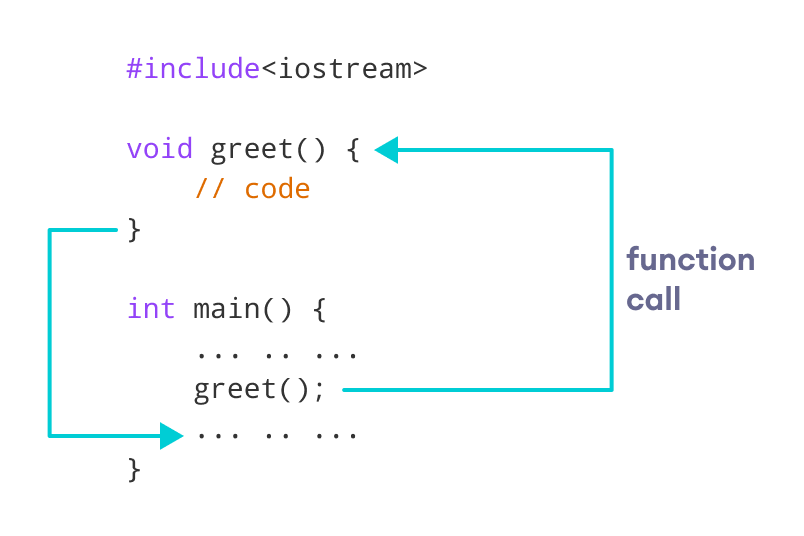
Here's how we can call the above greet() function.

int main() {

// calling a function

greet();

}

How Function works in C++

**Example 1: Display a Text**

#include <iostream>

using namespace std;

// declaring a function

void greet() {

cout << "Hello there!";

}

int main() {

// calling the function

greet();

return 0;

}

**Output**

Hello there!

**Function Parameters**

As mentioned above, a function can be declared with parameters (arguments). A parameter is a value that is passed when declaring a function.

For example, let us consider the function below:

void printNum(int num) {

cout << num;

}

Here, the int variable num is the function parameter.

We pass a value to the function parameter while calling the function.

int main() {

int n = 7;

// calling the function

// n is passed to the function as argument

printNum(n);

return 0;

}

**Example 2: Function with Parameters**

// program to print a text

#include <iostream>

using namespace std;

// display a number

void displayNum(int n1, float n2) {

cout << "The int number is " << n1;

cout << "The double number is " << n2;

}

int main() {

int num1 = 5;

double num2 = 5.5;

// calling the function

displayNum(num1, num2);

return 0;

}

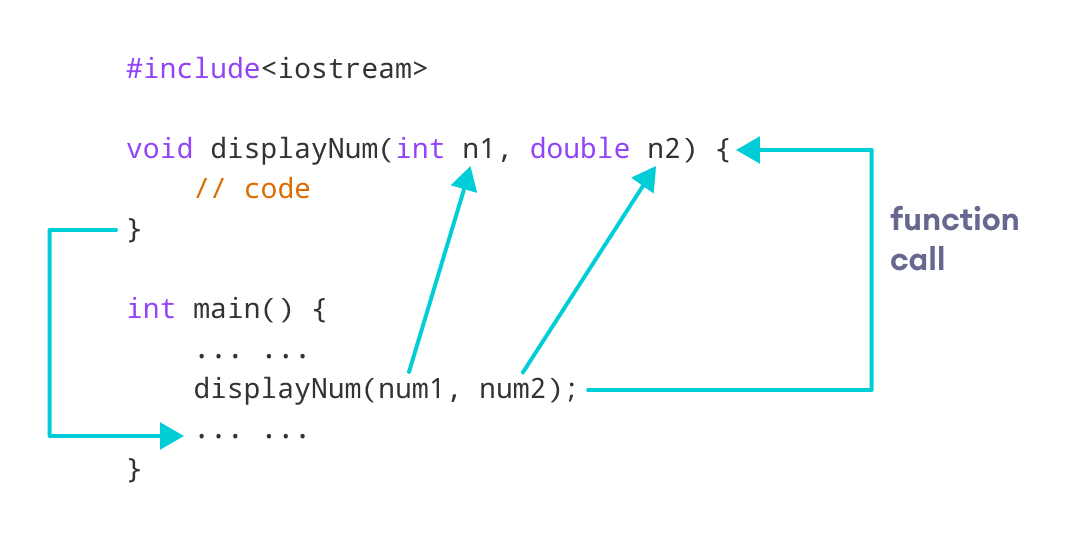
**Output**

The int number is 5

The double number is 5.5

In the above program, we have used a function that has one int parameter and one double parameter.

We then pass num1 and num2 as arguments. These values are stored by the function parameters n1 and n2 respectively.

C++ function with parameters

**Note:** The type of the arguments passed while calling the function must match with the corresponding parameters defined in the function declaration.

**Return Statement**

In the above programs, we have used void in the function declaration. For example,

void displayNumber() {

// code

}

This means the function is not returning any value.

It's also possible to return a value from a function. For this, we need to specify the returnType of the function during function declaration.

Then, the return statement can be used to return a value from a function.

For example,

int add (int a, int b) {

return (a + b);

}

Here, we have the data type int instead of void. This means that the function returns an int value.

The code return (a + b); returns the sum of the two parameters as the function value.

The return statement denotes that the function has ended. Any code after return inside the function is not executed.

**Example 3: Add Two Numbers**

// program to add two numbers using a function

#include <iostream>

using namespace std;

// declaring a function

int add(int a, int b) {

return (a + b);

}

int main() {

int sum;

// calling the function and storing

// the returned value in sum

sum = add(100, 78);

cout << "100 + 78 = " << sum << endl;

return 0;

}

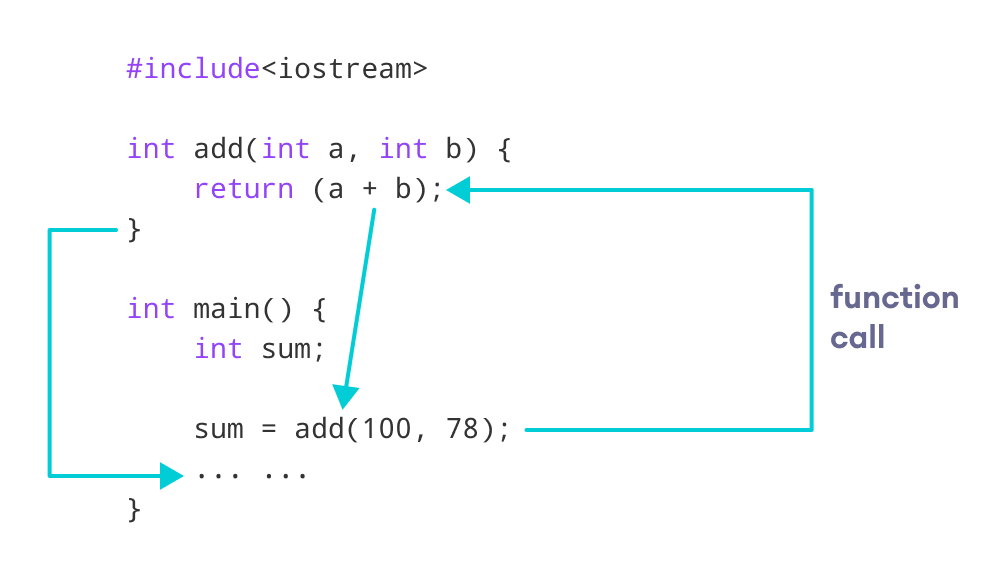
**Output**

100 + 78 = 178

In the above program, the add() function is used to find the sum of two numbers.

We pass two int literals 100 and 78 while calling the function.

We store the returned value of the function in the variable sum, and then we print it.

Working of C++ Function with return statement

Notice that sum is a variable of int type. This is because the return value of add() is of int type.

**Function Prototype**

In C++, the code of function declaration should be before the function call. However, if we want to define a function after the function call, we need to use the function prototype. For example,

// function prototype

void add(int, int);

int main() {

// calling the function before declaration.

add(5, 3);

return 0;

}

// function definition

void add(int a, int b) {

cout << (a + b);

}

In the above code, the function prototype is:

void add(int, int);

This provides the compiler with information about the function name and its parameters. That's why we can use the code to call a function before the function has been defined.

The syntax of a function prototype is:

returnType functionName(dataType1, dataType2, ...);

**Example 4: C++ Function Prototype**

// using function definition after main() function

// function prototype is declared before main()

#include <iostream>

using namespace std;

// function prototype

int add(int, int);

int main() {

int sum;

// calling the function and storing

// the returned value in sum

sum = add(100, 78);

cout << "100 + 78 = " << sum << endl;

return 0;

}

// function definition

int add(int a, int b) {

return (a + b);

}

**Output**

100 + 78 = 178

The above program is nearly identical to **Example 3**. The only difference is that here, the function is defined **after** the function call.

That's why we have used a function prototype in this example.

**Benefits of Using User-Defined Functions**

* Functions make the code reusable. We can declare them once and use them multiple times.
* Functions make the program easier as each small task is divided into a function.
* Functions increase readability.

**C++ Library Functions**

Library functions are the built-in functions in C++ programming.

Programmers can use library functions by invoking the functions directly; they don't need to write the functions themselves.

Some common library functions in C++ are sqrt(), abs(), isdigit(), etc.

In order to use library functions, we usually need to include the header file in which these library functions are defined.

For instance, in order to use mathematical functions such as sqrt() and abs(), we need to include the header file cmath.

**Example 5: C++ Program to Find the Square Root of a Number**

#include <iostream>

#include <cmath>

using namespace std;

int main() {

double number, squareRoot;

number = 25.0;

// sqrt() is a library function to calculate the square root

squareRoot = sqrt(number);

cout << "Square root of " << number << " = " << squareRoot;

return 0;

}

**Output**

Square root of 25 = 5

In this program, the sqrt() library function is used to calculate the square root of a number.

The function declaration of sqrt() is defined in the cmath header file. That's why we need to use the code #include <cmath> to use the sqrt() function.

To learn more, visit [C++ Standard Library functions](https://www.programiz.com/cpp-programming/library-function).

**Types of User-defined Functions in C++**

In this tutorial, you will learn about different approaches you can take to solve a single problem using functions.

For better understanding of arguments and return in functions, user-defined functions can be categorised as:

* [Function with no argument and no return value](https://www.programiz.com/cpp-programming/user-defined-function-types#no_argument_no_return)
* [Function with no argument but return value](https://www.programiz.com/cpp-programming/user-defined-function-types#no_argument_yes_return)
* [Function with argument but no return value](https://www.programiz.com/cpp-programming/user-defined-function-types#yes_argument_no_return)
* [Function with argument and return value](https://www.programiz.com/cpp-programming/user-defined-function-types#yes_argument_yes_return)

Consider a situation in which you have to check prime number. This problem is solved below by making user-defined function in 4 different ways as mentioned above.

**Example 1: No arguments passed and no return value**

# include <iostream>

using namespace std;

void prime();

int main()

{

// No argument is passed to prime()

prime();

return 0;

}

// Return type of function is void because value is not returned.

void prime()

{

int num, i, flag = 0;

cout << "Enter a positive integer enter to check: ";

cin >> num;

for(i = 2; i <= num/2; ++i)

{

if(num % i == 0)

{

flag = 1;

break;

}

}

if (flag == 1)

{

cout << num << " is not a prime number.";

}

else

{

cout << num << " is a prime number.";

}

}

In the above program, prime() is called from the main() with no arguments.

prime() takes the positive number from the user and checks whether the number is a prime number or not.

Since, return type of prime() is void, no value is returned from the function.

**Example 2: No arguments passed but a return value**

#include <iostream>

using namespace std;

int prime();

int main()

{

int num, i, flag = 0;

// No argument is passed to prime()

num = prime();

for (i = 2; i <= num/2; ++i)

{

if (num%i == 0)

{

flag = 1;

break;

}

}

if (flag == 1)

{

cout<<num<<" is not a prime number.";

}

else

{

cout<<num<<" is a prime number.";

}

return 0;

}

// Return type of function is int

int prime()

{

int n;

printf("Enter a positive integer to check: ");

cin >> n;

return n;

}

In the above program, prime() function is called from the main() with no arguments.

prime() takes a positive integer from the user. Since, return type of the function is an int, it returns the inputted number from the user back to the calling main() function.

Then, whether the number is prime or not is checked in the main() itself and printed onto the screen.

**Example 3: Arguments passed but no return value**

#include <iostream>

using namespace std;

void prime(int n);

int main()

{

int num;

cout << "Enter a positive integer to check: ";

cin >> num;

// Argument num is passed to the function prime()

prime(num);

return 0;

}

// There is no return value to calling function. Hence, return type of function is void. \*/

void prime(int n)

{

int i, flag = 0;

for (i = 2; i <= n/2; ++i)

{

if (n%i == 0)

{

flag = 1;

break;

}

}

if (flag == 1)

{

cout << n << " is not a prime number.";

}

else {

cout << n << " is a prime number.";

}

}

In the above program, positive number is first asked from the user which is stored in the variable num.

Then, num is passed to the prime() function where, whether the number is prime or not is checked and printed.

Since, the return type of prime() is a void, no value is returned from the function.

**Example 4: Arguments passed and a return value.**

#include <iostream>

using namespace std;

int prime(int n);

int main()

{

int num, flag = 0;

cout << "Enter positive integer to check: ";

cin >> num;

// Argument num is passed to check() function

flag = prime(num);

if(flag == 1)

cout << num << " is not a prime number.";

else

cout<< num << " is a prime number.";

return 0;

}

/\* This function returns integer value. \*/

int prime(int n)

{

int i;

for(i = 2; i <= n/2; ++i)

{

if(n % i == 0)

return 1;

}

return 0;

}

In the above program, a positive integer is asked from the user and stored in the variable num.

Then, num is passed to the function prime() where, whether the number is prime or not is checked.

Since, the return type of prime() is an int, 1 or 0 is returned to the main() calling function. If the number is a prime number, 1 is returned. If not, 0 is returned.

Back in the main() function, the returned 1 or 0 is stored in the variable flag, and the corresponding text is printed onto the screen.

**Which method is better?**

All four programs above gives the same output and all are technically correct program.

There is no hard and fast rule on which method should be chosen.

The particular method is chosen depending upon the situation and how you want to solve a problem.

# C++ Function Overloading

#### In this tutorial, we will learn about the function overloading in C++ with examples.

In C++, two functions can have the same name if the number and/or type of arguments passed is different.

These functions having the same name but different arguments are known as overloaded functions. For example:

// same number different arguments

int test() { }

int test(int a) { }

float test(double a) { }

int test(int a, double b) { }

Here, all 4 functions are overloaded functions.

Notice that the return types of all these 4 functions are not the same. Overloaded functions may or may not have different return types but they must have different arguments. For example,

// Error code

int test(int a) { }

double test(int b){ }

Here, both functions have the same name, the same type, and the same number of arguments. Hence, the compiler will throw an error.

## Function Overloading using Different Types of Parameter

// Program to compute absolute value

// Works for both int and float

#include <iostream>

using namespace std;

// function with float type parameter

float absolute(float var){

if (var < 0.0)

var = -var;

return var;

}

// function with int type parameter

int absolute(int var) {

if (var < 0)

var = -var;

return var;

}

int main() {

// call function with int type parameter

cout << "Absolute value of -5 = " << absolute(-5) << endl;

// call function with float type parameter

cout << "Absolute value of 5.5 = " << absolute(5.5f) << endl;

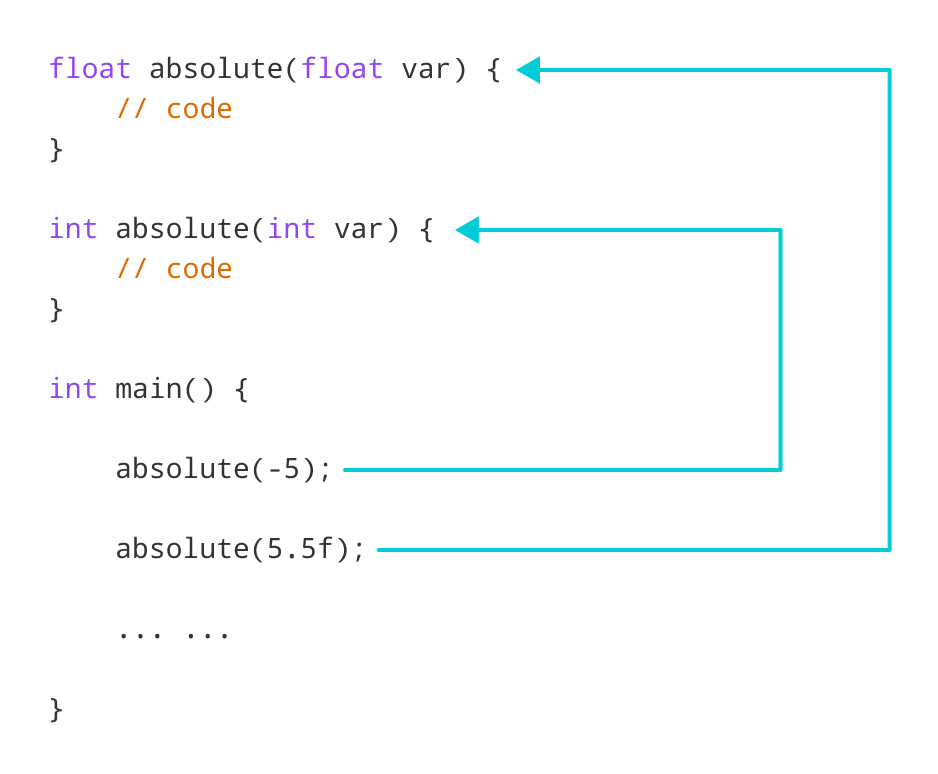
return 0;

}

**Output**

Absolute value of -5 = 5

Absolute value of 5.5 = 5.5

Working of overloading for the absolute() function

In this program, we overload the absolute() function. Based on the type of parameter passed during the function call, the corresponding function is called.

## Function Overloading using Different Number of Parameters

#include <iostream>

using namespace std;

// function with 2 parameters

void display(int var1, double var2) {

cout << "Integer number: " << var1;

cout << " and double number: " << var2 << endl;

}

// function with double type single parameter

void display(double var) {

cout << "Double number: " << var << endl;

}

// function with int type single parameter

void display(int var) {

cout << "Integer number: " << var << endl;

}

int main() {

int a = 5;

double b = 5.5;

// call function with int type parameter

display(a);

// call function with double type parameter

display(b);

// call function with 2 parameters

display(a, b);

return 0;

}

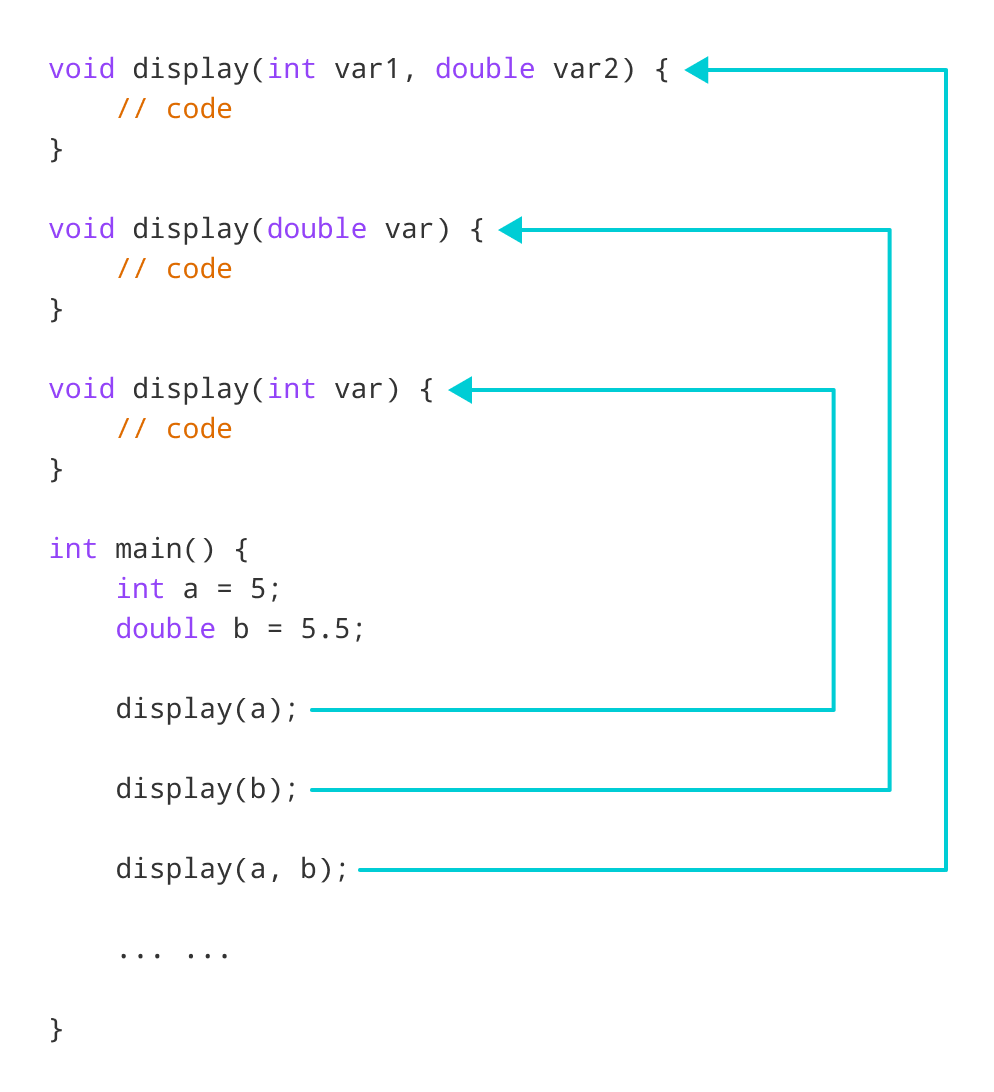
**Output**

Integer number: 5

Float number: 5.5

Integer number: 5 and double number: 5.5

Here, the display() function is called three times with different arguments. Depending on the number and type of arguments passed, the corresponding display() function is called.

Working of overloading for the display() function

The return type of all these functions is the same but that need not be the case for function overloading.

**Note:** In C++, many standard library functions are overloaded. For example, the sqrt() function can take double, float, int, etc. as parameters. This is possible because the sqrt() function is overloaded in C++.

**C++ Programming Default Arguments (Parameters)**

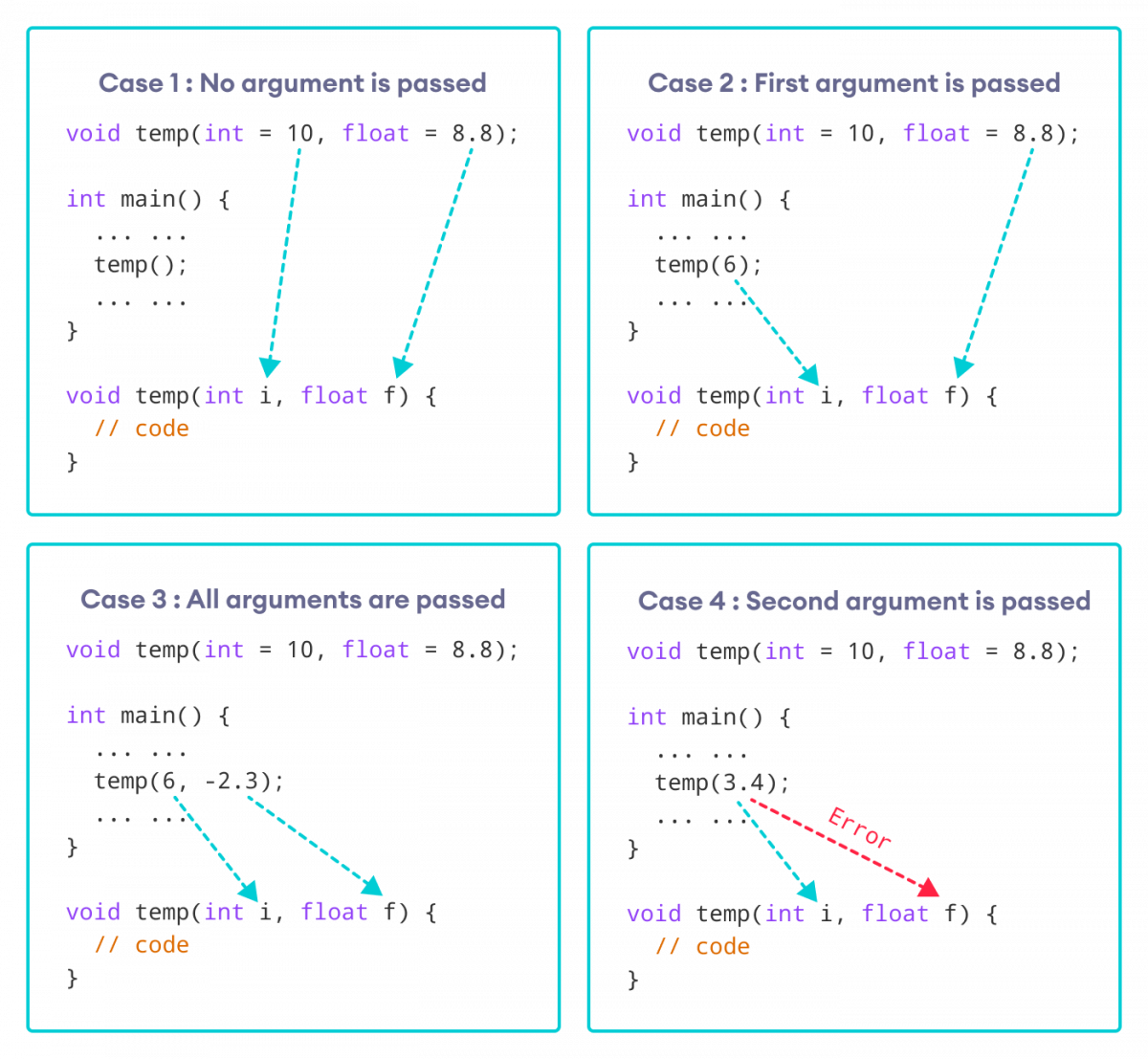
In this tutorial, we will learn C++ default arguments and their working with the help of examples.

In C++ programming, we can provide default values for [function](https://www.programiz.com/cpp-programming/function) parameters.

If a function with default arguments is called without passing arguments, then the default parameters are used.

However, if arguments are passed while calling the function, the default arguments are ignored.

**Working of default arguments**

How default arguments work in C++

We can understand the working of default arguments from the image above:

1. When temp() is called, both the default parameters are used by the function.
2. When temp(6) is called, the first argument becomes 6 while the default value is used for the second parameter.
3. When temp(6, -2.3) is called, both the default parameters are overridden, resulting in i = 6 and f = -2.3.
4. When temp(3.4) is passed, the function behaves in an undesired way because the second argument cannot be passed without passing the first argument.  
     
   Therefore, 3.4 is passed as the first argument. Since the first argument has been defined as int, the value that is actually passed is 3.

**Example: Default Argument**

#include <iostream>

using namespace std;

// defining the default arguments

void display(char = '\*', int = 3);

int main() {

int count = 5;

cout << "No argument passed: ";

// \*, 3 will be parameters

display();

cout << "First argument passed: ";

// #, 3 will be parameters

display('#');

cout << "Both arguments passed: ";

// $, 5 will be parameters

display('$', count);

return 0;

}

void display(char c, int count) {

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

{

cout << c;

}

cout << endl;

}

**Output**

No argument passed: \*\*\*

First argument passed: ###

Both arguments passed: $$$$$

Here is how this program works:

1. display() is called without passing any arguments. In this case, display() uses both the default parameters c = '\*' and n = 1.
2. display('#') is called with only one argument. In this case, the first becomes '#'. The second default parameter n = 1 is retained.
3. display('#', count) is called with both arguments. In this case, default arguments are not used.

We can also define the default parameters in the function definition itself. The program below is equivalent to the one above.

#include <iostream>

using namespace std;

// defining the default arguments

void display(char c = '\*', int count = 3) {

for(int i = 1; i <= count; ++i) {

cout << c;

}

cout << endl;

}

int main() {

int count = 5;

cout << "No argument passed: ";

// \*, 3 will be parameters

display();

cout << "First argument passed: ";

// #, 3 will be parameters

display('#');

cout << "Both argument passed: ";

// $, 5 will be parameters

display('$', count);

return 0;

}

**Things to Remember**

1. Once we provide a default value for a parameter, all subsequent parameters must also have default values. For example,
2. // Invalid
3. void add(int a, int b = 3, int c, int d);
4. // Invalid
5. void add(int a, int b = 3, int c, int d = 4);
6. // Valid

void add(int a, int c, int b = 3, int d = 4);

1. If we are defining the default arguments in the function definition instead of the function prototype, then the function must be defined before the function call.
2. // Invalid code
3. int main() {
4. // function call
5. display();
6. }
7. void display(char c = '\*', int count = 5) {
8. // code

}

**C++ Storage Class**

In this article, you'll learn about different storage classes in C++. Namely: local, global, static local, register and thread local.

Every variable in C++ has two features: type and storage class.

Type specifies the type of data that can be stored in a variable. For example: int, float, char etc.

And, storage class controls two different properties of a variable: lifetime (determines how long a variable can exist) and scope (determines which part of the program can access it).

Depending upon the storage class of a variable, it can be divided into 4 major types:

* [Local variable](https://www.programiz.com/cpp-programming/storage-class#local_variable)
* [Global variable](https://www.programiz.com/cpp-programming/storage-class#global_variable)
* [Static local variable](https://www.programiz.com/cpp-programming/storage-class#static_variable)
* [Register Variable](https://www.programiz.com/cpp-programming/storage-class#register%20variable)
* [Thread Local Storage](https://www.programiz.com/cpp-programming/storage-class#thread_local_storage)

**Local Variable**

A variable defined inside a function (defined inside [function](https://www.programiz.com/cpp-programming/function) body between braces) is called a local variable or automatic variable.

Its scope is only limited to the function where it is defined. In simple terms, local variable exists and can be accessed only inside a function.

The life of a local variable ends (It is destroyed) when the function exits.

**Example 1: Local variable**

#include <iostream>

using namespace std;

void test();

int main()

{

// local variable to main()

int var = 5;

test();

// illegal: var1 not declared inside main()

var1 = 9;

}

void test()

{

// local variable to test()

int var1;

var1 = 6;

// illegal: var not declared inside test()

cout << var;

}

The variable var cannot be used inside test() and var1 cannot be used inside main() function.

Keyword auto was also used for defining local variables before as: auto int var;

But, after [C++11 auto](http://en.cppreference.com/w/cpp/language/auto) has a different meaning and should not be used for defining local variables.

**Global Variable**

If a variable is defined outside all functions, then it is called a global variable.

The scope of a global variable is the whole program. This means, It can be used and changed at any part of the program after its declaration.

Likewise, its life ends only when the program ends.

**Example 2: Global variable**

#include <iostream>

using namespace std;

// Global variable declaration

int c = 12;

void test();

int main()

{

++c;

// Outputs 13

cout << c <<endl;

test();

return 0;

}

void test()

{

++c;

// Outputs 14

cout << c;

}

**Output**

13

14

In the above program, c is a global variable.

This variable is visible to both functions main() and test() in the above program.

**Static Local variable**

Keyword static is used for specifying a static variable. For example:

... .. ...

int main()

{

static float a;

... .. ...

}

A static local variable exists only inside a function where it is declared (similar to a local variable) but its lifetime starts when the function is called and ends only when the program ends.

The main difference between local variable and static variable is that, the value of static variable persists the end of the program.

**Example 3: Static local variable**

#include <iostream>

using namespace std;

void test()

{

// var is a static variable

static int var = 0;

++var;

cout << var << endl;

}

int main()

{

test();

test();

return 0;

}

**Output**

1

2

In the above program, test() function is invoked 2 times.

During the first call, variable var is declared as static variable and initialized to 0. Then 1 is added to var which is displayed in the screen.

When the function test() returns, variable var still exists because it is a static variable.

During second function call, no new variable var is created. The same var is increased by 1 and then displayed to the screen.

**Output of above program if var was not specified as static variable**

1

1

**Register Variable (Deprecated in C++11)**

Keyword register is used for specifying register variables.

Register variables are similar to automatic variables and exists inside a particular function only. It is supposed to be faster than the local variables.

If a program encounters a register variable, it stores the variable in processor's register rather than memory if available. This makes it faster than the local variables.

However, this keyword was deprecated in C++11 and should not be used.

**Thread Local Storage**

Thread-local storage is a mechanism by which variables are allocated such that there is one instance of the variable per extant thread.

Keyword thread\_local is used for this purpose.

Learn more about [thread local storage](http://www.codeproject.com/Articles/8113/Thread-Local-Storage-The-C-Way).

**C++ Recursion**

In this tutorial, we will learn about recursive function in C++ and its working with the help of examples.

A [function](https://www.programiz.com/cpp-programming/function) that calls itself is known as a recursive function. And, this technique is known as recursion.

**Working of Recursion in C++**

void recurse()

{

... .. ...

recurse();

... .. ...

}

int main()

{

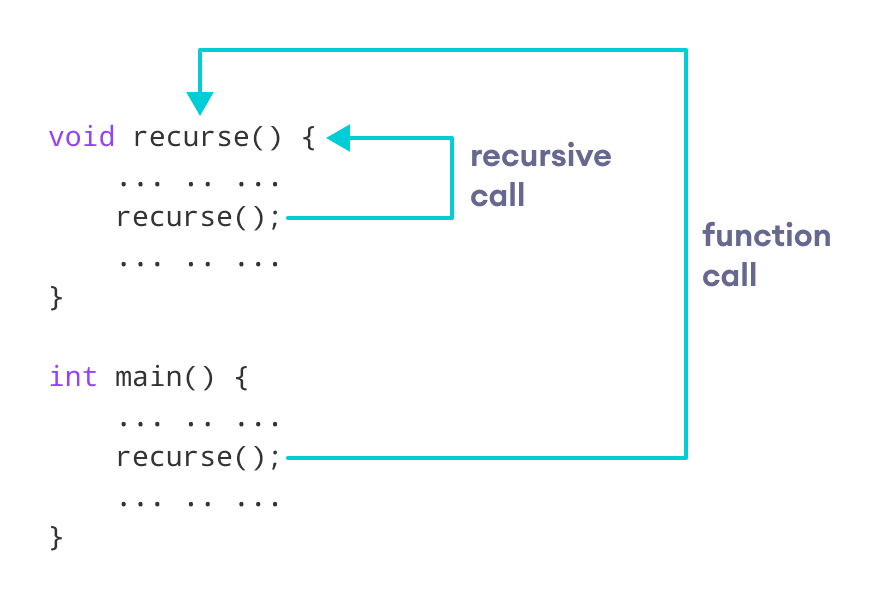
... .. ...

recurse();

... .. ...

}

The figure below shows how recursion works by calling itself over and over again.

How recursion works in C++ programming

The recursion continues until some condition is met.

To prevent infinite recursion, [if...else statement](https://www.programiz.com/cpp-programming/if-else) (or similar approach) can be used where one branch makes the recursive call and the other doesn't.

**Example 1: Factorial of a Number Using Recursion**

// Factorial of n = 1\*2\*3\*...\*n

#include <iostream>

using namespace std;

int factorial(int);

int main() {

int n, result;

cout << "Enter a non-negative number: ";

cin >> n;

result = factorial(n);

cout << "Factorial of " << n << " = " << result;

return 0;

}

int factorial(int n) {

if (n > 1) {

return n \* factorial(n - 1);

} else {

return 1;

}

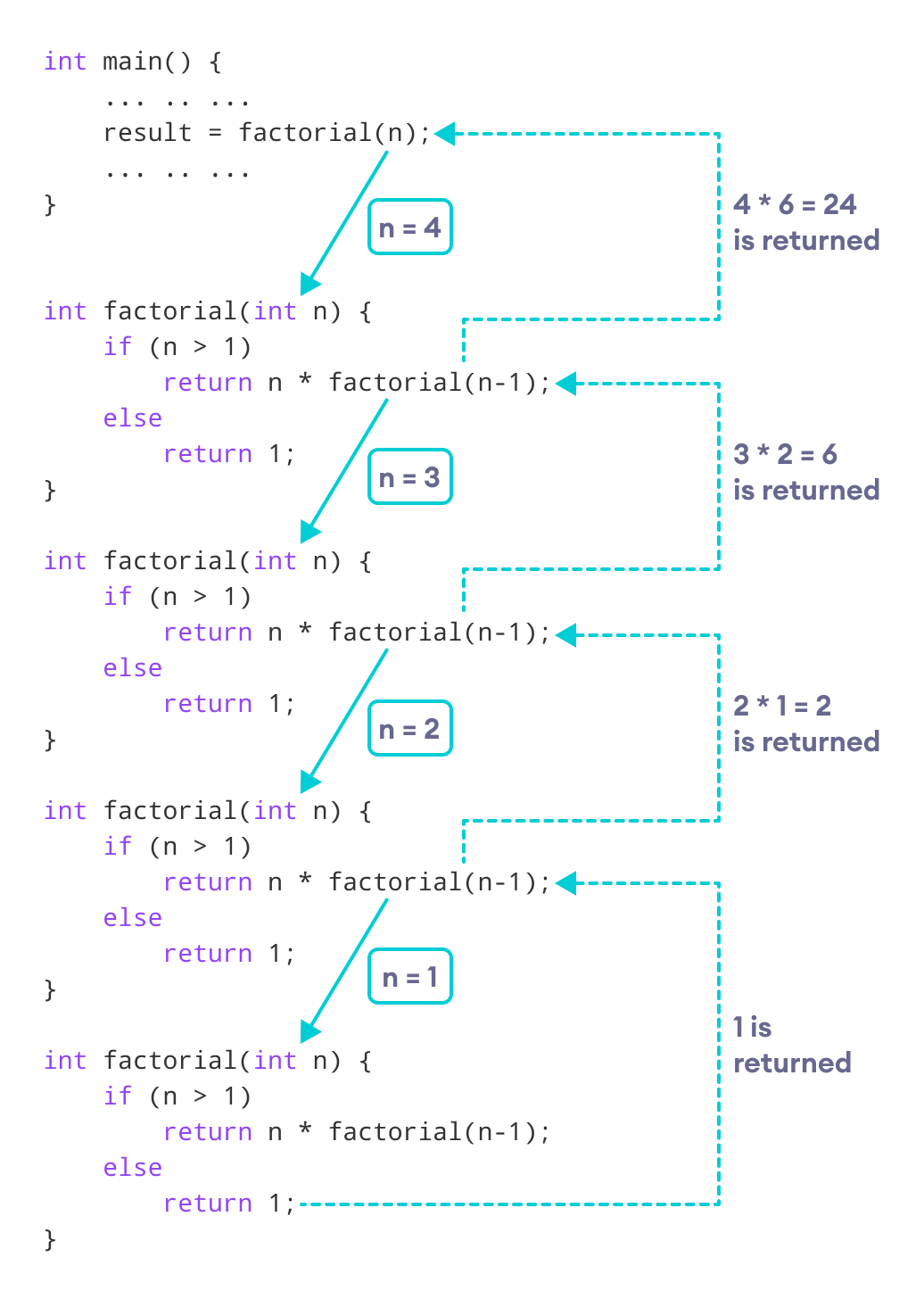
}

**Output**

Enter a non-negative number: 4

Factorial of 4 = 24

**Working of Factorial Program**

How this C++ recursion program works

As we can see, the factorial() function is calling itself. However, during each call, we have decreased the value of n by 1. When n is less than 1, the factorial() function ultimately returns the output.

**Advantages and Disadvantages of Recursion**

Below are the pros and cons of using recursion in C++.

**Advantages of C++ Recursion**

* It makes our code shorter and cleaner.
* Recursion is required in problems concerning data structures and advanced algorithms, such as Graph and Tree Traversal.

**Disadvantages of C++ Recursion**

* It takes a lot of stack space compared to an iterative program.
* It uses more processor time.
* It can be more difficult to debug compared to an equivalent iterative program.

**C++ Return by Reference**

In this article, you'll learn how to return a value by reference in a function and use it efficiently in your program.

In C++ Programming, not only can you pass values by reference to a [function](https://www.programiz.com/cpp-programming/function) but you can also return a value by reference.

To understand this feature, you should have the knowledge of:

* [Global variables](https://www.programiz.com/cpp-programming/storage-class#global_variable)

**Example: Return by Reference**

#include <iostream>

using namespace std;

// Global variable

int num;

// Function declaration

int& test();

int main()

{

test() = 5;

cout << num;

return 0;

}

int& test()

{

return num;

}

**Output**

5

In program above, the return type of function test() is int&. Hence, this function returns a reference of the variable num.

The return statement is return num;. Unlike return by value, this statement doesn't return value of num, instead it returns the variable itself (address).

So, when the **variable** is returned, it can be assigned a value as done in test() = 5;

This stores 5 to the variable num, which is displayed onto the screen.

**Important Things to Remember When Returning by Reference.**

* Ordinary function returns value but this function doesn't. Hence, you cannot return a constant from the function.
* int& test() {
* return 2;

}

* You cannot return a local variable from this function.
* int& test()
* {
* int n = 2;
* return n;

}