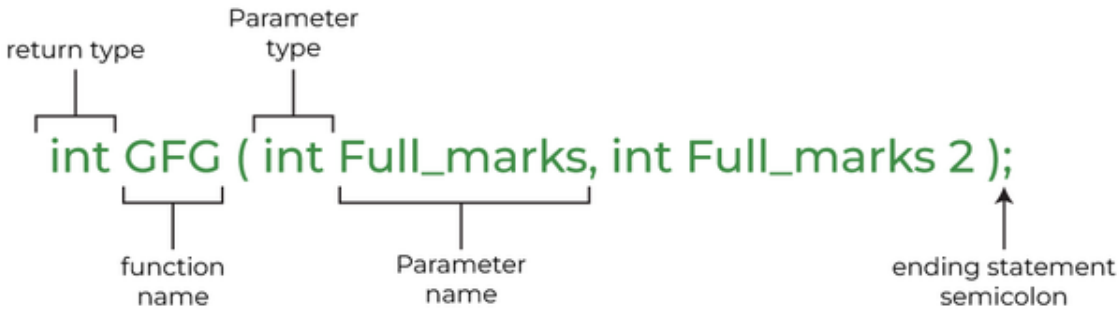
# Functions in C++

Difficulty Level : Easy • Last Updated : 16 Mar, 2023



A function is a set of statements that take inputs, do some specific computation, and produce output. The idea is to put some commonly or repeatedly done tasks together and make a **function** so that instead of writing the same code again and again for different inputs, we can call the function.
In simple terms, a function is a block of code that only runs when it is called.

**Syntax:**




*Syntax of Function*

**Example:**

## C++

```
// C++ Program to demonstrate working of a function
#include <iostream>
using namespace std;

// Following function that takes two parameters 'x' and 'y'
// as input and returns max of two input numbers
int max(int x, int y)
{
```

```
    if (x > y)
        return x;
    else
        return y;
}

// main function that doesn't receive any parameter and
// returns integer
int main()
{
    int a = 10, b = 20;

    // Calling above function to find max of 'a' and 'b'
    int m = max(a, b);

    cout << "m is " << m;
    return 0;
}
```

### Output



```
m is 20
```

**Time complexity:** O(1)

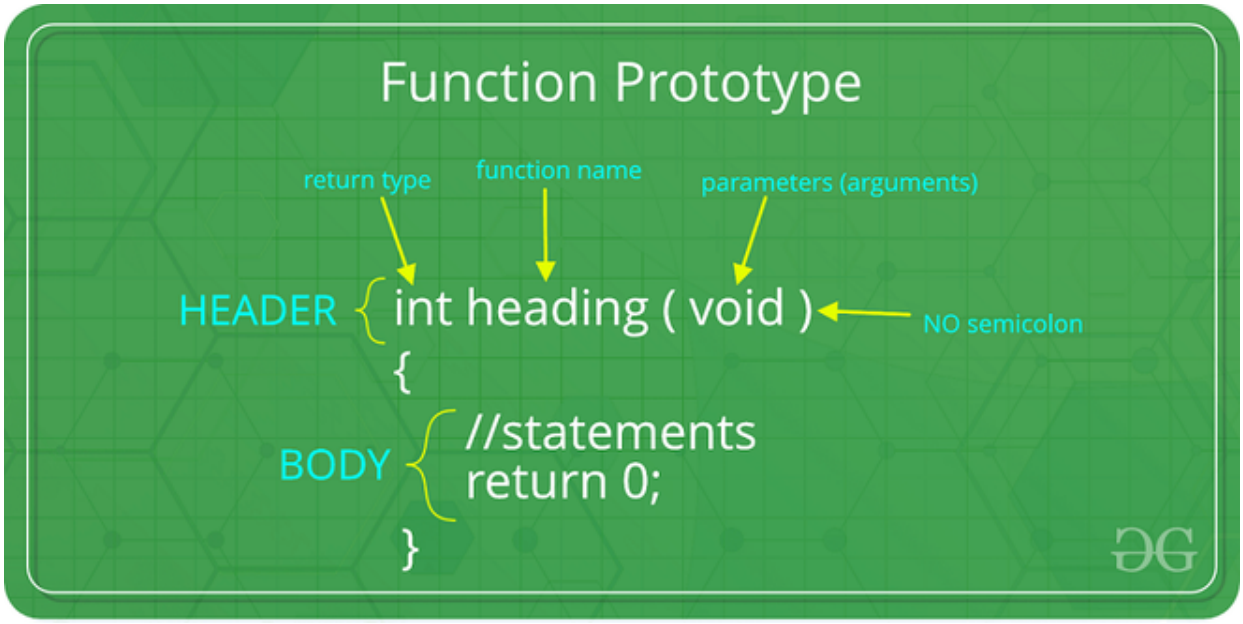
**Space complexity:** O(1)

## Why Do We Need Functions?

- Functions help us in ***reducing code redundancy***. If functionality is performed at multiple places in software, then rather than writing the same code, again and again, we create a function and call it everywhere. This also helps in maintenance as we have to change at one place if we make future changes to the functionality.
- Functions make code ***modular***. Consider a big file having many lines of code. It becomes really simple to read and use the code if the code is divided into functions.
- Functions provide ***abstraction***. For example, we can use library functions without worrying about their internal work.

## Function Declaration

A function declaration tells the compiler about the number of parameters function takes data-types of parameters, and returns the type of function. Putting parameter names in the function declaration is optional in the function declaration, but it is necessary to put them in the definition. Below are an example of function declarations. (parameter names are not there in the below declarations)




*Function Declaration*

**Example:**

## C++

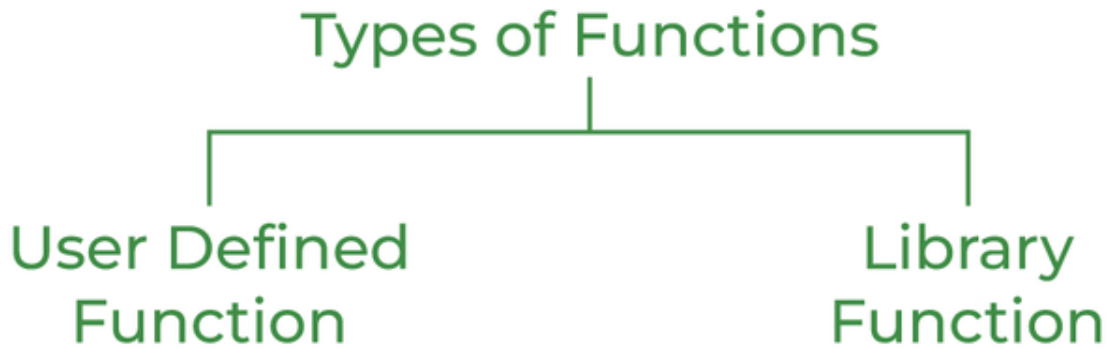
```
// C++ Program to show function that takes
// two integers as parameters and returns
// an integer
int max(int, int);

// A function that takes an int
// pointer and an int variable
// as parameters and returns
// a pointer of type int
int* swap(int*, int);

// A function that takes
// a char as parameter and
// returns a reference variable
char* call(char b);

// A function that takes a
// char and an int as parameters
// and returns an integer
int fun(char, int);
```

## Types of Functions




*Types of Function in C++*

### User Defined Function

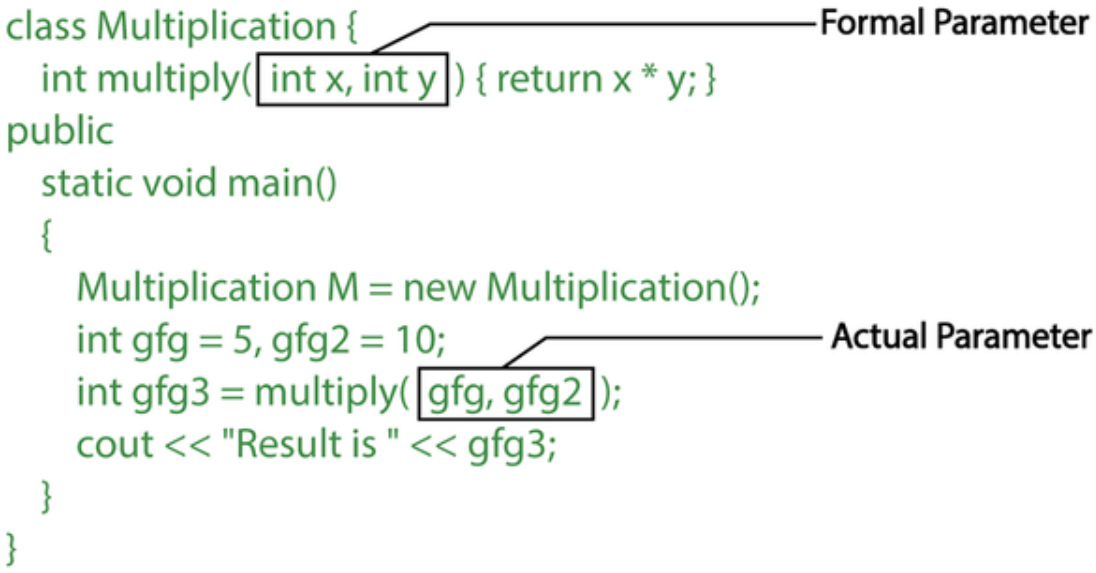
User Defined functions are user/customer-defined blocks of code specially customized to reduce the complexity of big programs. They are also commonly known as "***tailor-made functions***" which are built only to satisfy the condition in which the user is facing issues meanwhile reducing the complexity of the whole program.

### Library Function

Library functions are also called "***builtin Functions***". These functions are a part of a compiler package that is already defined and consists of a special function with special and different meanings. Builtin Function gives us an edge as we can directly use them without defining them whereas in the user-defined function we have to declare and define a function before using them.
**For Example:** sqrt(), setw(), strcat(), etc.

## Parameter Passing to Functions

The parameters passed to function are called ***actual parameters***. For example, in the program below, 5 and 10 are actual parameters.
The parameters received by the function are called ***formal parameters***. For example, in the above program x and y are formal parameters.




*Formal Parameter and Actual Parameter*

**There are two most popular ways to pass parameters:**

1. ***Pass by Value:*** In this parameter passing method, values of actual parameters are copied to the function's formal parameters and the two types of parameters are stored in different memory locations. So any changes made inside functions are not reflected in the actual parameters of the caller.

2. ***Pass by Reference:*** Both actual and formal parameters refer to the same locations, so any changes made inside the function are actually reflected in the actual parameters of the caller.

## Function Definition

***Pass by value*** is used where the value of x is not modified using the function fun().

---

### C++

```
// C++ Program to demonstrate function definition
#include <iostream>
using namespace std;

void fun(int x)
{
    // definition of
    // function
    x = 30;
}

int main()
{
    int x = 20;
    fun(x);
```

```
    cout << "x = " << x;
    return 0;
}
```

**Output**

```
x = 20
```

**Time complexity:** O(1)

**Space complexity:** O(1)

## Functions Using Pointers

The function fun() expects a pointer ptr to an integer (or an address of an integer). It modifies the value at the address ptr. The dereference operator * is used to access the value at an address. In the statement '*ptr = 30', the value at address ptr is changed to 30. The address operator & is used to get the address of a variable of any data type. In the function call statement 'fun(&x)', the address of x is passed so that x can be modified using its address.

---

### C++

```
// C++ Program to demonstrate working of
// function using pointers
#include <iostream>
using namespace std;

void fun(int* ptr) { *ptr = 30; }

int main()
{
    int x = 20;
    fun(&x);
    cout << "x = " << x;

    return 0;
}
```

**Output**

```
x = 30
```

**Time complexity:** O(1)

**Space complexity:** O(1)

## Difference between call by value and call by reference in C++

| Call by value | Call by reference |
|---|---|
| A copy of value is passed to the function | An address of value is passed to the function |
| Changes made inside the function is not reflected on other functions | Changes made inside the function is reflected<br>outside the function also |
| Actual and formal arguments will be created in<br>different memory location | Actual and formal arguments will be created in<br>same memory location. |

## Points to Remember About Functions in C++

**1.** Most C++ program has a function called main() that is called by the operating system when a user runs the program.

**2.** Every function has a return type. If a function doesn't return any value, then void is used as a return type. Moreover, if the return type of the function is void, we still can use the return statement in the body of the function definition by not specifying any constant, variable, etc. with it, by only mentioning the 'return;' statement which would symbolize the termination of the function as shown below:

---

### C++

```
void function name(int a)
{
    ..... // Function Body
        return; // Function execution would get terminated
}
```

**3.** To declare a function that can only be called without any parameter, we should use "**void fun(void)**". As a side note, in C++, an empty list means a function can only be called without any parameter. In C++, both void fun() and void fun(void) are same.

## Main Function

The main function is a special function. Every C++ program must contain a function named main. It serves as the entry point for the program. The computer will start running the code

from the beginning of the main function.

### Types of Main Functions

**1. Without parameters:**

**CPP**

```
// Without Parameters
int main() { ... return 0; }
```

**2. With parameters:**

**CPP**

```
// With Parameters
int main(int argc, char* const argv[]) { ... return 0; }
```

The reason for having the parameter option for the main function is to allow input from the command line. When you use the main function with parameters, it saves every group of characters (separated by a space) after the program name as elements in an array named **argv**.
Since the main function has the return type of **int**, the programmer must always have a return statement in the code. The number that is returned is used to inform the calling program what the result of the program's execution was. Returning 0 signals that there were no problems.

## C++ Recursion

When function is called within the same function, it is known as recursion in C++. The function which calls the same function, is known as recursive function.
A function that calls itself, and doesn't perform any task after function call, is known as tail recursion. In tail recursion, we generally call the same function with return statement.
**Syntax:**

**C++**

```
recursionfunction()
{
    recursionfunction(); // calling self function
}
```

To know more see this article.

## C++ Passing Array to Function

In C++, to reuse the array logic, we can create function. To pass array to function in C++, we need to provide only array name.

```
functionname(arrayname); //passing array to function
```

**Example: Print minimum number**

### C++

```
#include <iostream>
using namespace std;
void printMin(int arr[5]);
int main()
{
    int ar[5] = { 30, 10, 20, 40, 50 };
    printMin(ar); // passing array to function
}
void printMin(int arr[5])
{
    int min = arr[0];
    for (int i = 0; i < 5; i++) {
        if (min > arr[i]) {
            min = arr[i];
        }
    }
    cout << "Minimum element is: " << min << "\n";
}

// Code submitted by Susobhan Akhuli
```

**Output**

```
Minimum element is: 10
```

Time complexity: O(n) where n is the size of array

Space complexity: O(n) where n is the size of array.

## C++ Overloading (Function)

If we create two or more members having the same name but different in number or type of parameter, it is known as C++ overloading. In C++, we can overload:

- *methods,*

- *constructors and*
- *indexed properties*

It is because these members have parameters only.

### Types of overloading in C++ are:

- *Function overloading*
- *Operator overloading*

### C++ Function Overloading

Function Overloading is defined as the process of having two or more function with the same name, but different in parameters is known as function overloading in C++. In function overloading, the function is redefined by using either different types of arguments or a different number of arguments. It is only through these differences compiler can differentiate between the functions.
The advantage of Function overloading is that it increases the readability of the program because you don't need to use different names for the same action.
**Example: changing number of arguments of add() method**

---

## C++

```
// program of function overloading when number of arguments
// vary
#include <iostream>
using namespace std;
class Cal {
public:
    static int add(int a, int b) { return a + b; }
    static int add(int a, int b, int c)
    {
        return a + b + c;
    }
};
int main(void)
{
    Cal C; // class object declaration.
    cout << C.add(10, 20) << endl;
    cout << C.add(12, 20, 23);
    return 0;
}

// Code Submitted By Susobhan Akhuli
```

**Output**

```
30
55
```

**Time complexity:** O(1)

**Space complexity:** O(1)

Example: when the type of the arguments vary.

---

## C++

```
// Program of function overloading with different types of
// arguments.
#include <iostream>
using namespace std;
int mul(int, int);
float mul(float, int);

int mul(int a, int b) { return a * b; }
float mul(double x, int y) { return x * y; }
int main()
{
    int r1 = mul(6, 7);
    float r2 = mul(0.2, 3);
    cout << "r1 is : " << r1 << endl;
    cout << "r2 is : " << r2 << endl;
    return 0;
}

// Code Submitted By Susobhan Akhuli
```

**Output**

```
r1 is : 42
r2 is : 0.6
```

**Time Complexity:** O(1)

**Space Complexity:** O(1)

### Function Overloading and Ambiguity

When the compiler is unable to decide which function is to be invoked among the overloaded function, this situation is known as *function overloading*.
When the compiler shows the ambiguity error, the compiler does not run the program.
**Causes of Function Overloading:**

- *Type Conversion.*
- *Function with default arguments.*
- *Function with pass by reference.*

**Type Conversion:-**

**C++**

```
#include <iostream>
using namespace std;
void fun(int);
void fun(float);
void fun(int i) { cout << "Value of i is : " << i << endl; }
void fun(float j)
{
    cout << "Value of j is : " << j << endl;
}
int main()
{
    fun(12);
    fun(1.2);
    return 0;
}

// Code Submitted By Susobhan Akhuli
```

The above example shows an error "*call of overloaded 'fun(double)' is ambiguous*". The fun(10) will call the first function. The fun(1.2) calls the second function according to our prediction. But, this does not refer to any function as in C++, all the floating point constants are treated as double not as a float. If we replace float to double, the program works. Therefore, this is a type conversion from float to double.

**Function with Default Arguments:-**

**C++**

```
#include <iostream>
using namespace std;
void fun(int);
void fun(int, int);
void fun(int i) { cout << "Value of i is : " << i << endl; }
void fun(int a, int b = 9)
{
    cout << "Value of a is : " << a << endl;
    cout << "Value of b is : " << b << endl;
}
int main()
{
    fun(12);
```

```
    return 0;
}

// Code Submitted By Susobhan Akhuli
```

The above example shows an error "*call of overloaded 'fun(int)' is ambiguous*". The fun(int a, int b=9) can be called in two ways: first is by calling the function with one argument, i.e., fun(12) and another way is calling the function with two arguments, i.e., fun(4,5). The fun(int i) function is invoked with one argument. Therefore, the compiler could not be able to select among fun(int i) and fun(int a,int b=9).

**Function with Pass By Reference:-**

---

## C++

```
#include <iostream>
using namespace std;
void fun(int);
void fun(int&);
int main()
{
    int a = 10;
    fun(a); // error, which fun()?
    return 0;
}
void fun(int x) { cout << "Value of x is : " << x << endl; }
void fun(int& b)
{
    cout << "Value of b is : " << b << endl;
}

// Code Submitted By Susobhan Akhuli
```

The above example shows an error "*call of overloaded 'fun(int&)' is ambiguous*". The first function takes one integer argument and the second function takes a reference parameter as an argument. In this case, the compiler does not know which function is needed by the user as there is no syntactical difference between the fun(int) and fun(int &).

Friend Function

- A friend function is a special function in C++ which in-spite of not being member function of a class has privilege to access private and protected data of a class.
- A friend function is a non member function or ordinary function of a class, which is declared as a friend using the keyword "friend" inside the class. By declaring a function as a friend, all the access permissions are given to the function.

- The keyword "friend" is placed only in the function declaration of the friend function and not in the function definition.
- When friend function is called neither name of object nor dot operator is used. However it may accept the object as argument whose value it want to access.
- Friend function can be declared in any section of the class i.e. public or private or protected.

Declaration of friend function in C++

Syntax :

```
class <class_name>
{
        friend  <return_type>  <function_name>(argument/s);
};
```

Example_1: Find the largest of two numbers using Friend Function

---

## C++

```
#include<iostream>
using namespace std;
class Largest
{
    int a,b,m;
    public:
         void set_data();
         friend void find_max(Largest);
};

void Largest::set_data()
{
    cout<<"Enter the First No:";
    cin>>a;
    cout<<"Enter the Second No:";
    cin>>b;
}

void find_max(Largest t)
{
    if(t.a>t.b)
         t.m=t.a;
    else
         t.m=t.b;

     cout<<"Maximum Number is\t"<<t.m;
}

main()
{
    Largest l;
```

```
    l.set_data();
    find_max(l);
    return 0;
}
```

### Output

```
Enter the First No:Enter the Second No:Maximum Number is    2117529904
```

461

## Related Articles

1. Static functions in C
2. Write one line functions for strcat() and strcmp()
3. Can Static Functions Be Virtual in C++?
4. Functions that cannot be overloaded in C++
5. Functions that are executed before and after main() in C
6. Pure Functions
7. Can Virtual Functions be Private in C++?
8. Virtual Functions and Runtime Polymorphism in C++
9. Can Virtual Functions be Inlined in C++?
10. Macros vs Functions

Previous Next

### Article Contributed By :

GeeksforGeeks

# Default Arguments in C++

Difficulty Level : Easy • Last Updated : 27 Jan, 2023



A default argument is a value provided in a function declaration that is automatically assigned by the compiler if the calling function doesn't provide a value for the argument. In case any value is passed, the default value is overridden.

**1)** The following is a simple C++ example to demonstrate the use of default arguments. Here, we don't have to write 3 sum functions; only one function works by using the default values for 3rd and 4th arguments.

## CPP

```
// CPP Program to demonstrate Default Arguments
#include <iostream>
using namespace std;

// A function with default arguments,
// it can be called with
// 2 arguments or 3 arguments or 4 arguments.
int sum(int x, int y, int z = 0, int w = 0) //assigning default values to z,w as 0
{
    return (x + y + z + w);
}

// Driver Code
int main()
{
    // Statement 1
    cout << sum(10, 15) << endl;

    // Statement 2
    cout << sum(10, 15, 25) << endl;

    // Statement 3
    cout << sum(10, 15, 25, 30) << endl;
```

```
    return 0;
}
```

**Output**

```
25
50
80
```

**Time Complexity:** O(1)



**Space Complexity:** O(1)

**Explanation:** In statement 1, only two values are passed, hence the variables z and w take the default values of 0. In statement 2, three values are passed, so the value of z is overridden with 25. In statement 3, four values are passed, so the values of z and w are overridden with 25 and 30 respectively.

**2)** If function overloading is done containing the default arguments, then we need to make sure it is not ambiguous to the compiler, otherwise it will throw an error. The following is the modified version of the above program:

---

## CPP

```
// CPP Program to demonstrate Function overloading in
// Default Arguments
#include <iostream>
using namespace std;

// A function with default arguments, it can be called with
// 2 arguments or 3 arguments or 4 arguments.
int sum(int x, int y, int z = 0, int w = 0)
{
    return (x + y + z + w);
}
int sum(int x, int y, float z = 0, float w = 0)
{
    return (x + y + z + w);
}
// Driver Code
```

```
int main()
{
    cout << sum(10, 15) << endl;
    cout << sum(10, 15, 25) << endl;
    cout << sum(10, 15, 25, 30) << endl;
    return 0;
}
```

**Error:**

```
prog.cpp: In function 'int main()':
prog.cpp:17:20: error: call of overloaded
'sum(int, int)' is ambiguous
  cout << sum(10, 15) << endl;
                    ^
prog.cpp:6:5: note: candidate:
int sum(int, int, int, int)
 int sum(int x, int y, int z=0, int w=0)
     ^
prog.cpp:10:5: note: candidate:
int sum(int, int, float, float)
 int sum(int x, int y, float z=0, float w=0)
     ^
```

**3)** A constructor can contain default parameters as well. A default constructor can either have no parameters or parameters with default arguments.

---

## C++

```
// CPP code to demonstrate use of default arguments in
// Constructors

#include <iostream>
using namespace std;

class A {
    private:
        int var = 0;
    public:
        A(int x = 0): var(x){}; // default constructor with one argument
                                // Note that var(x) is the syntax in c++ to do : "va
        void setVar(int s){
            var = s; // OR => this->var = s;
            return;
        }
        int getVar(){
            return var; // OR => return this->var;
        }
};
```

```
int main(){
    A a(1);

    a.setVar(2);

    cout << "var = " << a.getVar() << endl;

    /* ANOTHER APPROACH:
    A *a = new A(1);

    a->setVar(2);

    cout << "var = " << a->getVar() << endl;
    */
}

// contributed by Francisco Vargas #pt
```

**Explanation**: Here, we see a default constructor with no arguments and a default constructor with one default argument. The default constructor with argument has a default parameter x, which has been assigned a value of 0.

**Key Points:**

- Default arguments are different from constant arguments as constant arguments can't be changed whereas default arguments can be overwritten if required.
- Default arguments are overwritten when the calling function provides values for them. For example, calling the function sum(10, 15, 25, 30) overwrites the values of z and w to 25 and 30 respectively.
- When a function is called, the arguments are copied from the calling function to the called function in the order left to right. Therefore, sum(10, 15, 25) will assign 10, 15, and 25 to x, y, and z respectively, which means that only the default value of w is used.
- Once a default value is used for an argument in the function definition, all subsequent arguments to it must have a default value as well. It can also be stated that the default arguments are assigned from right to left. For example, the following function definition is invalid as the subsequent argument of the default variable z is not default.

```
// Invalid because z has default value, but w after it doesn't have a
default value
int sum(int x, int y, int z = 0, int w).
```

**Advantages of Default Arguments:**

- Default arguments are useful when we want to increase the capabilities of an existing function as we can do it just by adding another default argument to the function.
- It helps in reducing the size of a program.

- It provides a simple and effective programming approach.
- Default arguments improve the consistency of a program.

**Disadvantages of Default Arguments:**

- It increases the execution time as the compiler needs to replace the omitted arguments by their default values in the function call.

203

## Related Articles

1. Templates and Default Arguments
2. Some Interesting facts about default arguments in C++
3. Default Arguments and Virtual Function in C++
4. When do we pass arguments by reference or pointer?
5. Template non-type arguments in C++
6. Command Line Arguments in C/C++
7. Does C++ compiler create default constructor when we write our own?
8. Default constructor in Java
9. C++ default constructor | Built-in types for int(), float, double()
10. Default Assignment Operator and References in C++

Previous Next

**Article Contributed By :**

GeeksforGeeks

**Vote for difficulty**



# Inline Functions in C++

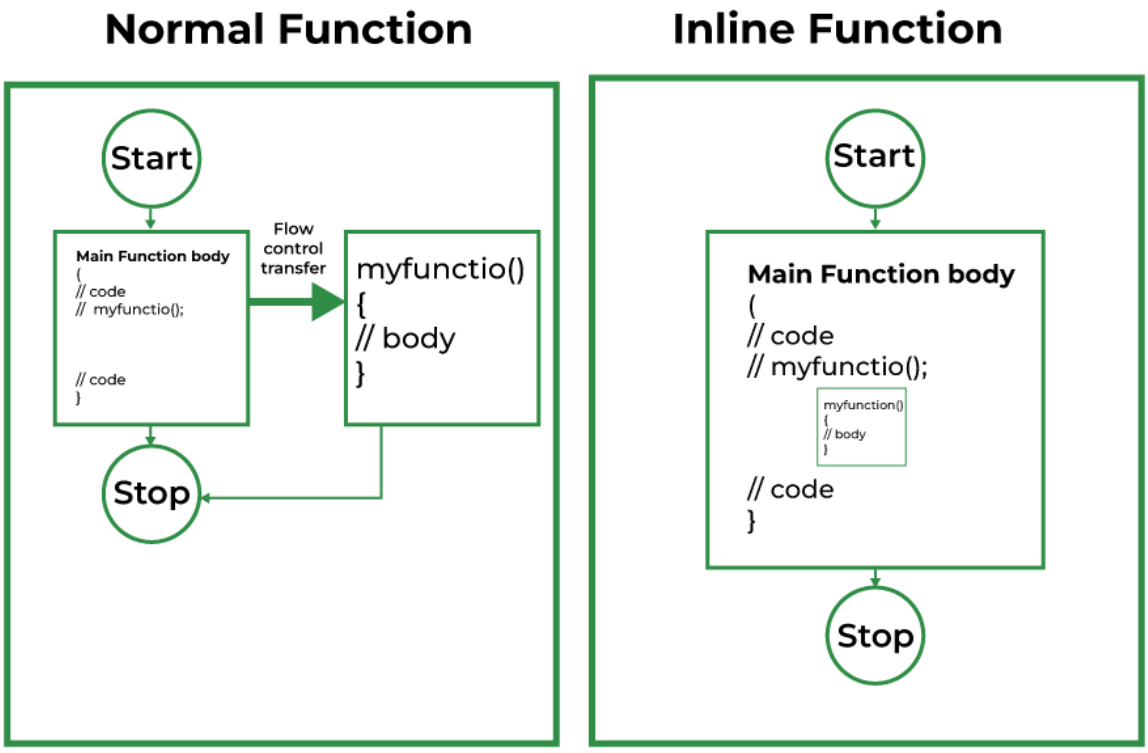
Difficulty Level : Medium • Last Updated : 05 Mar, 2023



C++ provides inline functions to reduce the function call overhead. An inline function is a function that is expanded in line when it is called. When the inline function is called whole code of the inline function gets inserted or substituted at the point of the inline function call. This substitution is performed by the C++ compiler at compile time. An inline function may increase efficiency if it is small.

**Syntax:**

```
inline return-type function-name(parameters)
{
    // function code
}
```




Remember, inlining is only a request to the compiler, not a command. The compiler can ignore the request for inlining.

**The compiler may not perform inlining in such circumstances as:**



1. If a function contains a loop. (*for, while and do-while*)
2. If a function contains static variables.
3. If a function is recursive.
4. If a function return type is other than void, and the return statement doesn't exist in a function body.
5. If a function contains a switch or goto statement.

## Why Inline Functions are Used?

When the program executes the function call instruction the CPU stores the memory address of the instruction following the function call, copies the arguments of the function on the stack, and finally transfers control to the specified function. The CPU then executes the function code, stores the function return value in a predefined memory location/register, and returns control to the calling function. This can become overhead if the execution time of the function is less than the switching time from the caller function to called function (callee).

For functions that are large and/or perform complex tasks, the overhead of the function call is usually insignificant compared to the amount of time the function takes to run. However, for small, commonly-used functions, the time needed to make the function call is often a lot more than the time needed to actually execute the function's code. This overhead occurs for small functions because the execution time of a small function is less than the switching time.

## Inline functions Advantages:

1. Function call overhead doesn't occur.
2. It also saves the overhead of push/pop variables on the stack when a function is called.
3. It also saves the overhead of a return call from a function.
4. When you inline a function, you may enable the compiler to perform context-specific optimization on the body of the function. Such optimizations are not possible for normal function calls. Other optimizations can be obtained by considering the flows of the calling context and the called context.
5. An inline function may be useful (if it is small) for embedded systems because inline can yield less code than the function called preamble and return.

## Inline function Disadvantages:

1. The added variables from the inlined function consume additional registers, After the in-lining function if the variable number which is going to use the register increases then they may create overhead on register variable resource utilization. This means that when the inline function body is substituted at the point of the function call, the total number of variables used by the function also gets inserted. So the number of registers going to be used for the variables will also get increased. So if after function inlining variable numbers increase drastically then it would surely cause overhead on register utilization.
2. If you use too many inline functions then the size of the binary executable file will be large, because of the duplication of the same code.
3. Too much inlining can also reduce your instruction cache hit rate, thus reducing the speed of instruction fetch from that of cache memory to that of primary memory.

4. The inline function may increase compile time overhead if someone changes the code inside the inline function then all the calling location has to be recompiled because the compiler would be required to replace all the code once again to reflect the changes, otherwise it will continue with old functionality.
5. Inline functions may not be useful for many embedded systems. Because in embedded systems code size is more important than speed.
6. Inline functions might cause thrashing because inlining might increase the size of the binary executable file. Thrashing in memory causes the performance of the computer to degrade. The following program demonstrates the use of the inline function.

**Example:**

---

**C++**

```
#include <iostream>
using namespace std;
inline int cube(int s) { return s * s * s; }
int main()
{
    cout << "The cube of 3 is: " << cube(3) << "\n";
    return 0;
}
```

**Output**

```
The cube of 3 is: 27
```

## Inline function and classes

It is also possible to define the inline function inside the class. In fact, all the functions defined inside the class are implicitly inline. Thus, all the restrictions of inline functions are also applied here. If you need to explicitly declare an inline function in the class then just declare the function inside the class and define it outside the class using the inline keyword.

**Syntax:**

```
class S
{
public:
    inline int square(int s) // redundant use of inline
    {
        // this function is automatically inline
        // function body
```

```
    }
};
```

The above style is considered a bad programming style. The best programming style is to just write the prototype of the function inside the class and specify it as an inline in the function definition.

**For Example:**

```
class S
{
public:
    int square(int s); // declare the function
};

inline int S::square(int s) // use inline prefix
{
}
```

**Example:**

---

## C++

```
// C++ Program to demonstrate inline functions and classes
#include <iostream>

using namespace std;

class operation {
    int a, b, add, sub, mul;
    float div;

public:
    void get();
    void sum();
    void difference();
    void product();
    void division();
};
inline void operation ::get()
{
    cout << "Enter first value:";
    cin >> a;
    cout << "Enter second value:";
    cin >> b;
}

inline void operation ::sum()
{
```

```
    add = a + b;
    cout << "Addition of two numbers: " << a + b << "\n";
}

inline void operation ::difference()
{
    sub = a - b;
    cout << "Difference of two numbers: " << a - b << "\n";
}

inline void operation ::product()
{
    mul = a * b;
    cout << "Product of two numbers: " << a * b << "\n";
}

inline void operation ::division()
{
    div = a / b;
    cout << "Division of two numbers: " << a / b << "\n";
}

int main()
{
    cout << "Program using inline function\n";
    operation s;
    s.get();
    s.sum();
    s.difference();
    s.product();
    s.division();
    return 0;
}
```

**Output:**

```
Enter first value: 45
Enter second value: 15
Addition of two numbers: 60
Difference of two numbers: 30
Product of two numbers: 675
Division of two numbers: 3
```

## What is wrong with the macro?

Readers familiar with the C language know that the C language uses macro. The preprocessor replaces all macro calls directly within the macro code. It is recommended to always use the inline function instead of the macro. According to Dr. Bjarne Stroustrup, the creator of C++ macros are almost never necessary in C++ and they are error-prone. There

are some problems with the use of macros in C++. Macro cannot access private members of the class. Macros look like function calls but they are actually not.

**Example:**

---

## C++

```
// C++ Program to demonstrate working of macro
#include <iostream>
using namespace std;
class S {
    int m;

public:
    // error
#define MAC(S::m)
};
```

**Output:**

```
Error: "::" may not appear in macro parameter list
 #define MAC(S::m)
```

C++ compiler checks the argument types of inline functions and necessary conversions are performed correctly. The preprocessor macro is not capable of doing this. One other thing is that the macros are managed by the preprocessor and inline functions are managed by the C++ compiler. Remember: It is true that all the functions defined inside the class are implicitly inline and the C++ compiler will perform inline calls of these functions, but the C++ compiler cannot perform inline if the function is virtual. The reason is called to a virtual function is resolved at runtime instead of compile-time. Virtual means waiting until runtime and inline means during compilation, if the compiler doesn't know which function will be called, how it can perform inlining? One other thing to remember is that it is only useful to make the function inline if the time spent during a function call is more compared to the function body execution time.

**An example where the inline function has no effect at all:**

```
inline void show()
{
    cout << "value of S = " << S << endl;
}
```

The above function relatively takes a long time to execute. In general, a function that performs an input-output (I/O) operation shouldn't be defined as inline because it spends a

considerable amount of time. Technically inlining of the show() function is of limited value because the amount of time the I/O statement will take far exceeds the overhead of a function call. Depending upon the compiler you are using the compiler may show you a warning if the function is not expanded inline.

Programming languages like Java & C# don't support inline functions. But in Java, the compiler can perform inlining when the small final method is called because final methods can't be overridden by subclasses, and the call to a final method is resolved at compile time.

In C# JIT compiler can also optimize code by inlining small function calls (like replacing the body of a small function when it is called in a loop). The last thing to keep in mind is that inline functions are a valuable feature of C++. Appropriate use of inline functions can provide performance enhancement but if inline functions are used arbitrarily then they can't provide better results. In other words, don't expect a better performance of the program. Don't make every function inline. It is better to keep inline functions as small as possible.

This article is contributed by **Meet Pravasi**. Please write comments if you find anything incorrect, or if you want to share more information about the topic discussed above.

338

## Related Articles

1. Mathematical Functions in Python | Set 1 (Numeric Functions)
2. Mathematical Functions in Python | Set 2 (Logarithmic and Power Functions)
3. Difference between Inline and Macro in C++
4. Difference Between Inline and Normal Function in C++
5. Difference between virtual function and inline function in C++
6. C++ Inline Namespaces and Usage of the "using" Directive Inside Namespaces
7. Can Static Functions Be Virtual in C++?
8. Virtual Functions in Derived Classes in C++
9. Functions that cannot be overloaded in C++

# Return From Void Functions in C++

Difficulty Level : Easy • Last Updated : 27 Jan, 2023



Void functions are known as **Non-Value Returning functions**. They are "void" due to the fact that they are not supposed to return values. True, but not completely. We cannot return values but there is something we can surely return from void functions. **Void functions do not have a return type, but they can do return values.** Some of the cases are listed below:

**1) A Void Function Can Return:** We can simply write a return statement in a void fun(). In fact, it is considered a good practice (for readability of code) to write a return; statement to indicate the end of the function.

## CPP

```cpp
// CPP Program to demonstrate void functions
#include <iostream>
using namespace std;

void fun()
{
    cout << "Hello";

    // We can write return in void
    return;
}

// Driver Code
int main()
{
    fun();
    return 0;
}
```

### Output

```
Hello
```

**Time Complexity:** O(1)

**Space Complexity:** O(1)



**2) A void fun() can return another void function:** A void function can also call another void function while it is terminating. For example,

## CPP

```cpp
// C++ code to demonstrate void()
// returning void()
#include <iostream>
using namespace std;

// A sample void function
void work()
{
    cout << "The void function has returned "
            " a void() !!! \n";
}

// Driver void() returning void work()
void test()
{
    // Returning void function
    return work();
}

// Driver Code
int main()
{
    // Calling void function
    test();
    return 0;
}
```

### Output

```
The void function has returned  a void() !!!
```

**Time Complexity:** O(1)

**Space Complexity:** O(1)

The above code explains how void() can actually be useful to return void functions without giving errors.

**3) A void() can return a void value:** A void() cannot return a value that can be used. But it can return a value that is void without giving an error. For example,

---

## CPP

```cpp
// C++ code to demonstrate void()
// returning a void value
#include <iostream>
using namespace std;

// Driver void() returning a void value
void test()
{
    cout << "Hello";

    // Returning a void value
    return (void)"Doesn't Print";
}

// Driver Code
int main()
{
    test();
    return 0;
}
```

**Output**

```
Hello
```

**Time Complexity:** O(1)

**Space Complexity:** O(1)

This article is contributed by **Manjeet Singh**. If you like GeeksforGeeks and would like to contribute, you can also write an article using write.geeksforgeeks.org or mail your article to review-team@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

# Functors in C++

Difficulty Level : Medium • Last Updated : 27 Jan, 2023



Please note that the title is **Functors** (Not Functions)!!

Consider a function that takes only one argument. However, while calling this function we have a lot more information that we would like to pass to this function, but we cannot as it accepts only one parameter. What can be done?

One obvious answer might be global variables. However, good coding practices do not advocate the use of global variables and say they must be used only when there is no other alternative.

**Functors** are objects that can be treated as though they are a function or function pointer. Functors are most commonly used along with STLs in a scenario like following:



Below program uses transform() in STL to add 1 to all elements of arr[].

## CPP

```
// A C++ program uses transform() in STL to add
// 1 to all elements of arr[]
#include <bits/stdc++.h>
using namespace std;
```

```
int increment(int x) {  return (x+1); }

int main()
{
    int arr[] = {1, 2, 3, 4, 5};
    int n = sizeof(arr)/sizeof(arr[0]);

    // Apply increment to all elements of
    // arr[] and store the modified elements
    // back in arr[]
    transform(arr, arr+n, arr, increment);

    for (int i=0; i<n; i++)
        cout << arr[i] <<" ";

    return 0;
}
```

Output:

```
2 3 4 5 6
```

**Time Complexity:** O(n)

**Space Complexity:** O(n) where n is the size of the array.

This code snippet adds only one value to the contents of the arr[]. Now suppose, that we want to add 5 to contents of arr[].

See what's happening? As transform requires a unary function(a function taking only one argument) for an array, we cannot pass a number to increment(). And this would, in effect, make us write several different functions to add each number. What a mess. This is where functors come into use.

A functor (or function object) is a C++ class that acts like a function. Functors are called using the same old function call syntax. To create a functor, we create a object that overloads the *operator()*.

```
The line,
MyFunctor(10);

Is same as
MyFunctor.operator()(10);
```

Let's delve deeper and understand how this can actually be used in conjunction with STLs.

---

## CPP

```
// C++ program to demonstrate working of
// functors.
#include <bits/stdc++.h>
using namespace std;

// A Functor
class increment
{
private:
    int num;
public:
    increment(int n) : num(n) {  }

    // This operator overloading enables calling
    // operator function () on objects of increment
    int operator () (int arr_num) const {
        return num + arr_num;
    }
};

// Driver code
int main()
{
    int arr[] = {1, 2, 3, 4, 5};
    int n = sizeof(arr)/sizeof(arr[0]);
    int to_add = 5;

    transform(arr, arr+n, arr, increment(to_add));

    for (int i=0; i<n; i++)
        cout << arr[i] << " ";
}
```

Output:

```
6 7 8 9 10
```

**Time Complexity:** O(n)

**Space Complexity:** O(n) where n is the size of the array.

Thus, here, Increment is a functor, a c++ class that acts as a function.

**The line,**

```
transform(arr, arr+n, arr, increment(to_add));
```

**is the same as writing below two lines,**

```
// Creating object of increment
increment obj(to_add);
```

```
// Calling () on object
transform(arr, arr+n, arr, obj);
```

Thus, an object *a* is created that overloads the *operator()*. Hence, functors can be used effectively in conjunction with C++ STLs. This article is contributed by **Supriya Srivatsa**. If you like GeeksforGeeks and would like to contribute, you can also write an article and mail your article to review-team@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

185

## Related Articles

1. Sort an array according to absolute difference with given value using Functors
2. C++ - Difference Between Functors and Functions
3. How to Use Binder and Bind2nd Functors in C++ STL?
4. main Function in C
5. printf in C
6. C Program to Implement Max Heap
7. C Program to Implement Min Heap
8. C++ Error - Does not name a type
9. Execution Policy of STL Algorithms in Modern C++
10. C++ Program To Print Pyramid Patterns

Previous Next

**Article Contributed By :**

GeeksforGeeks