C++ Functions

The function in C++ language is also known as procedure or subroutine in other programming languages.

A function is a block of code which only runs when it is called.

You can pass data, known as parameters, into a function.

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

To perform any task, we can create function. A function can be called many times. It provides modularity and code reusability.

Advantage of functions in C

There are many advantages of functions.

**1) Code Reusability**

By creating functions in C++, you can call it many times. So we don't need to write the same code again and again.

**2) Code optimization**

It makes the code optimized, we don't need to write much code.

Suppose, you have to check 3 numbers (531, 883 and 781) whether it is prime number or not. Without using function, you need to write the prime number logic 3 times. So, there is repetition of code.

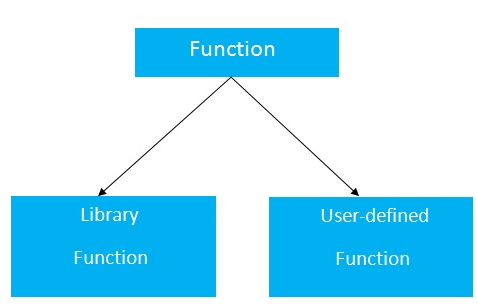
But if you use functions, you need to write the logic only once and you can reuse it several times.

Types of Functions

There are two types of functions in C programming:

**1. Library Functions:** are the functions which are declared in the C++ header files such as ceil(x), cos(x), exp(x), etc.

**2. User-defined functions:** are the functions which are created by the C++ programmer, so that he/she can use it many times. It reduces complexity of a big program and optimizes the code.



Declaration of a function

The syntax of creating function in C++ language is given below:

1. return\_type function\_name(data\_type parameter...)
2. {
3. //code to be executed
4. }

C++ Function Example

Let's see the simple example of C++ function.

1. #include <iostream>
2. **using** **namespace** std;
3. **void** func() {
4. **static** **int** i=0; //static variable
5. **int** j=0; //local variable
6. i++;
7. j++;
8. cout<<"i=" << i<<" and j=" <<j<<endl;
9. }
10. **int** main()
11. {
12. func();
13. func();
14. func();
15. }

Output:

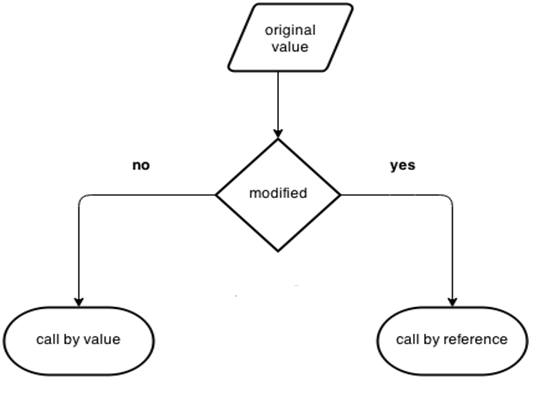
i= 1 and j= 1

i= 2 and j= 1

i= 3 and j= 1

Call by value and call by reference in C++

**There are two ways to pass value or data to function in C language: call by value and call by reference. Original value is not modified in call by value but it is modified in call by reference.**



Let's understand call by value and call by reference in C++ language one by one.

Call by value in C++

In call by value, **original value is not modified.**

In call by value, value being passed to the function is locally stored by the function parameter in stack memory location. If you change the value of function parameter, it is changed for the current function only. It will not change the value of variable inside the caller method such as main().

Hello Java Program for Beginners

Let's try to understand the concept of call by value in C++ language by the example given below:

#include <iostream>

**using** **namespace** std;

**void** change(**int** data);

**int** main()

{

**int** data = 3;

change(data);

cout << "Value of the data is: " << data<< endl;

**return** 0;

}

**void** change(**int** data)

{

data = 5;

}

Output:

Value of the data is: 3

Call by reference in C++

In call by reference, original value is modified because we pass reference (address).

Here, address of the value is passed in the function, so actual and formal arguments share the same address space. Hence, value changed inside the function, is reflected inside as well as outside the function.

**Note:** To understand the call by reference, you must have the basic knowledge of pointers.

Let's try to understand the concept of call by reference in C++ language by the example given below:

#include<iostream>

1. **using** **namespace** std;
2. **void** swap(**int** \*x, **int** \*y)
3. {
4. **int** swap;
5. swap=\*x;
6. \*x=\*y;
7. \*y=swap;
8. }
9. **int** main()
10. {
11. **int** x=500, y=100;
12. swap(&x, &y);  // passing value to function
13. cout<<"Value of x is: "<<x<<endl;
14. cout<<"Value of y is: "<<y<<endl;
15. **return** 0;
16. }

Output:

Value of x is: 100

Value of y is: 500

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

|  |  |  |
| --- | --- | --- |
| **No.** | **Call by value** | **Call by reference** |
| 1 | A copy of value is passed to the function | An address of value is passed to the function |
| 2 | Changes made inside the function is not reflected on other functions | Changes made inside the function is reflected outside the function also |
| 3 | Actual and formal arguments will be created in different memory location | Actual and formal arguments will be created in same memory location |

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.

C++ Function Overloading Example

Let's see the simple example of function overloading where we are changing number of arguments of add() method.

// program of function overloading when number of arguments vary.

1. #include <iostream>
2. **using** **namespace** std;
3. **class** Cal {
4. **public**:
5. **static** **int** add(**int** a,**int** b){
6. **return** a + b;
7. }
8. **static** **int** add(**int** a, **int** b, **int** c)
9. {
10. **return** a + b + c;
11. }
12. };
13. **int** main(**void**) {
14. Cal C;                                                    //     class object declaration.
15. cout<<C.add(10, 20)<<endl;
16. cout<<C.add(12, 20, 23);
17. **return** 0;
18. }

**Output:**

30

55

Let's see the simple example when the type of the arguments vary.

// Program of function overloading with different types of arguments.

1. #include<iostream>
2. **using** **namespace** std;
3. **int** mul(**int**,**int**);
4. **float** mul(**float**,**int**);

7. **int** mul(**int** a,**int** b)
8. {
9. **return** a\*b;
10. }
11. **float** mul(**double** x, **int** y)
12. {
13. **return** x\*y;
14. }
15. **int** main()
16. {
17. **int** r1 = mul(6,7);
18. **float** r2 = mul(0.2,3);
19. std::cout << "r1 is : " <<r1<< std::endl;
20. std::cout <<"r2 is : "  <<r2<< std::endl;
21. **return** 0;
22. }

**Output:**

r1 is : 42

r2 is : 0.6

Inline function in C++

If make a function as inline, then the compiler replaces the function calling location with the definition of the inline function at compile time.

Any changes made to an inline function will require the inline function to be recompiled again because the compiler would need to replace all the code with a new code; otherwise, it will execute the old functionality.

**Syntax for an inline function:**

1. **inline** return\_type function\_name(parameters)
2. {
3. // function code?
4. }

**Let's understand the difference between the normal function and the inline function.**

Triggers in SQL (Hindi)

Inside the **main()** method, when the function fun1() is called, the control is transferred to the definition of the called function. The addresses from where the function is called and the definition of the function are different. This control transfer takes a lot of time and increases the overhead.

When the inline function is encountered, then the definition of the function is copied to it. In this case, there is no control transfer which saves a lot of time and also decreases the overhead.

**Let's understand through an example.**

1. #include <iostream>
2. **using** **namespace** std;
3. **inline** **int** add(**int** a, **int** b)
4. {
5. **return**(a+b);
6. }
7. **int** main()
8. {
9. cout<<"Addition of 'a' and 'b' is:"<<add(2,3);A
10. **return** 0;
12. }

Once the compilation is done, the code would be like as shown as below:

1. #include<iostream>
2. **using** **namespace** std;
3. **inline** **int** add(**int** a, **int** b)
4. {
5. **return**(a+b);
6. }
7. **int** main()
8. {
9. cout<<"Addition of 'a' and 'b' is:"<<**return**(2+3);
10. **return** 0;
11. }

**Why do we need an inline function in C++?**

The main use of the inline function in C++ is to save memory space. Whenever the function is called, then it takes a lot of time to execute the tasks, such as moving to the calling function. If the length of the function is small, then the substantial amount of execution time is spent in such overheads, and sometimes time taken required for moving to the calling function will be greater than the time taken required to execute that function.

C++ has provided one solution to this problem. In the case of function calling, the time for calling such small functions is huge, so to overcome such a problem, a new concept was introduced known as an inline function. When the function is encountered inside the main() method, it is expanded with its definition thus saving time.

We cannot provide the inlining to the functions in the following circumstances:

* If a function is recursive.
* If a function contains a loop like for, while, do-while loop.
* If a function contains static variables.
* If a function contains a switch or go to statement

**When do we require an inline function?**

An inline function can be used in the following scenarios:

* An inline function can be used when the performance is required.
* It can be used over the macros.
* We can use the inline function outside the class so that we can hide the internal implementation of the function.

### Advantages of inline function

* In the inline function, we do not need to call a function, so it does not cause any overhead.
* It also saves the overhead of the return statement from a function.
* It does not require any stack on which we can push or pop the variables as it does not perform any function calling.
* An inline function is mainly beneficial for the embedded systems as it yields less code than a normal function.

Default arguments in C++

In a function, arguments are defined as the values passed when a function is called. Values passed are the source, and the receiving function is the destination.

Now let us understand the concept of default arguments in detail.

**Definition**

A default argument is a value in the function declaration automatically assigned by the compiler if the calling function does not pass any value to that argument.

Java Try Catch

**Characteristics for defining the default arguments**

Following are the rules of declaring default arguments -

* The values passed in the default arguments are not constant. These values can be overwritten if the value is passed to the function. If not, the previously declared value retains.
* During the calling of function, the values are copied from left to right.
* All the values that will be given default value will be on the right.

**Example**

* void function(int x, int y, int z = 0)  
  Explanation - The above function is valid. Here z is the value that is predefined as a part of the default argument.
* Void function(int x, int z = 0, int y)  
  Explanation - The above function is invalid. Here z is the value defined in between, and it is not accepted.

**Code**

1. #include<iostream>
2. **using** **namespace** std;
3. **int** sum(**int** x, **int** y, **int** z=0, **int** w=0) // Here there are two values in the default arguments
4. { // Both z and w are initialised to zero
5. **return** (x + y + z + w); // return sum of all parameter values
6. }
7. **int** main()
8. {
9. cout << sum(10, 15) << endl; // x = 10, y = 15, z = 0, w = 0
10. cout << sum(10, 15, 25) << endl; // x = 10, y = 15, z = 25, w = 0
11. cout << sum(10, 15, 25, 30) << endl; // x = 10, y = 15, z = 25, w = 30
12. **return** 0;
13. }

**Output**

25

50

80

**Explanation**

In the above program, we have called the sum function three times.

* Sum(10,15)  
  When this function is called, it reaches the definition of the sum. There it initializes x to 10 y to 15, and the rest values are zero by default as no value is passed. And all the values after sum give 25 as output.
* Sum(10, 15, 25)  
  When this function is called, x remains 10, y remains 15, the third parameter z that is passed is initialized to 25 instead of zero. And the last value remains 0. The sum of x, y, z, w, is 50 which is returned as output.
* Sum(10, 15, 25, 30)  
  In this function call, there are four parameter values passed into the function with x as 10, y as 15, z is 25, and w as 30. All the values are then summed up to give 80 as the output.

**Storage Classes in C++ with Examples**

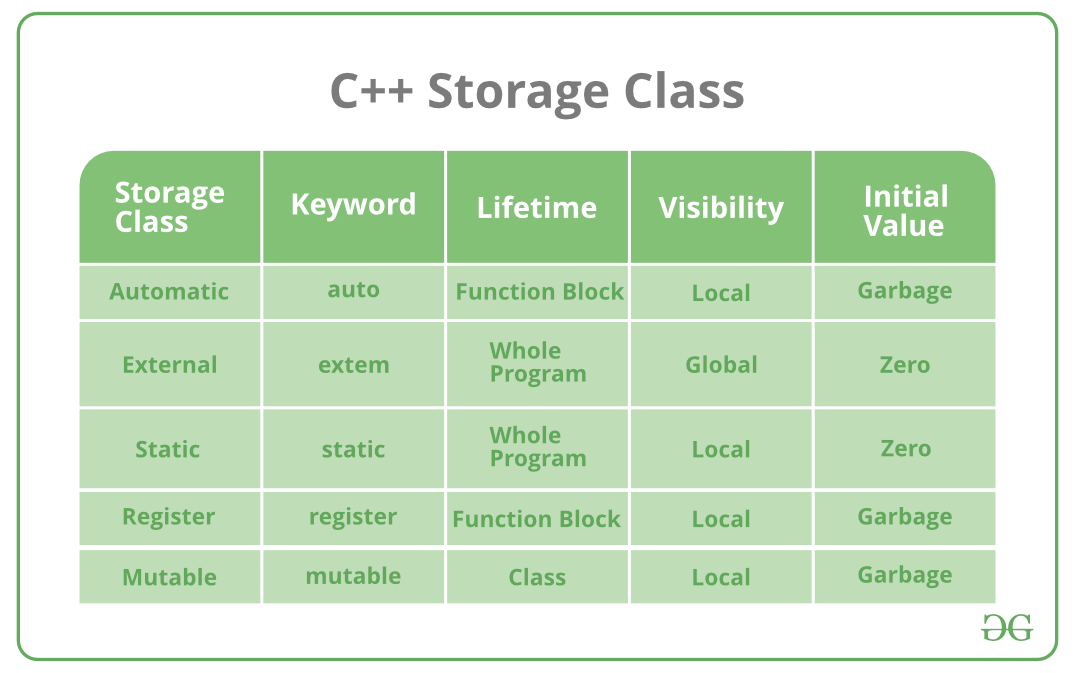
**Storage Classes** are used to describe the features of a variable/function. These features basically include the scope, visibility and life-time which help us to trace the existence of a particular variable during the runtime of a program. To specify the storage class for a variable, the following syntax is to be followed:

**Syntax:**

storage\_class var\_data\_type var\_name;

C++ uses 5 storage classes, namely:

1. auto
2. register
3. extern
4. static
5. mutable



Below is the detailed explanation of each storage class:

* **auto**: The auto keyword provides type inference capabilities, using which automatic deduction of the data type of an expression in a programming language can be done. This consumes less time having to write out things the compiler already knows. As all the types are deduced in compiler phase only, the time for compilation increases slightly but it does not affect the run time of the program. This feature also extends to functions and non-type template parameters. Since C++14 for functions, the return type will be deduced from its return statements. Since C++17, for non-type template parameters, the type will be deduced from the argument.

**Example:**

|  |
| --- |
| #include <iostream>  using namespace std;    void autoStorageClass()  {        cout << "Demonstrating auto class\n";        // Declaring an auto variable      // No data-type declaration needed      auto a = 32;      auto b = 3.2;      auto c = "GeeksforGeeks";      auto d = 'G';        // printing the auto variables      cout << a << " \n";      cout << b << " \n";      cout << c << " \n";      cout << d << " \n";  }    int main()  {        // To demonstrate auto Storage Class      autoStorageClass();        return 0;  } |

**Output:**

Demonstrating auto class

32

3.2

GeeksforGeeks

G

* **extern**: Extern storage class simply tells us that the variable is defined elsewhere and not within the same block where it is used. Basically, the value is assigned to it in a different block and this can be overwritten/changed in a different block as well. So an extern variable is nothing but a global variable initialized with a legal value where it is declared in order to be used elsewhere. It can be accessed within any function/block. Also, a normal global variable can be made extern as well by placing the ‘extern’ keyword before its declaration/definition in any function/block. This basically signifies that we are not initializing a new variable but instead we are using/accessing the global variable only. The main purpose of using extern variables is that they can be accessed between two different files which are part of a large program.
* **Example:**

|  |
| --- |
| #include <iostream>  using namespace std;    // declaring the variable which is to  // be made extern an initial value can  // also be initialized to x  int x;  void externStorageClass()  {        cout << "Demonstrating extern class\n";        // telling the compiler that the variable      // x is an extern variable and has been      // defined elsewhere (above the main      // function)      extern int x;        // printing the extern variables 'x'      cout << "Value of the variable 'x'"           << "declared, as extern: " << x << "\n";        // value of extern variable x modified      x = 2;        // printing the modified values of      // extern variables 'x'      cout          << "Modified value of the variable 'x'"          << " declared as extern: \n"          << x;  }    int main()  {        // To demonstrate extern Storage Class      externStorageClass();        return 0;  } |

**Output:**

Demonstrating extern class

Value of the variable 'x'declared, as extern: 0

Modified value of the variable 'x' declared as extern:

2

* **static**: This storage class is used to declare static variables which are popularly used while writing programs in C++ language. Static variables have a property of preserving their value even after they are out of their scope! Hence, static variables preserve the value of their last use in their scope. So we can say that they are initialized only once and exist until the termination of the program. Thus, no new memory is allocated because they are not re-declared. Their scope is local to the function to which they were defined. Global static variables can be accessed anywhere in the program. By default, they are assigned the value 0 by the compiler.

|  |
| --- |
| #include <iostream>  using namespace std;    // Function containing static variables  // memory is retained during execution  int staticFun()  {      cout << "For static variables: ";      static int count = 0;      count++;      return count;  }    // Function containing non-static variables  // memory is destroyed  int nonStaticFun()  {      cout << "For Non-Static variables: ";        int count = 0;      count++;      return count;  }    int main()  {        // Calling the static parts      cout << staticFun() << "\n";      cout << staticFun() << "\n";      ;        // Calling the non-static parts        cout << nonStaticFun() << "\n";      ;      cout << nonStaticFun() << "\n";      ;      return 0;  } |

**Output:**

For static variables: 1

For static variables: 2

For Non-Static variables: 1

For Non-Static variables: 1

* **register**: This storage class declares register variables which have the same functionality as that of the auto variables. The only difference is that the compiler tries to store these variables in the register of the microprocessor if a free register is available. This makes the use of register variables to be much faster than that of the variables stored in the memory during the runtime of the program. If a free register is not available, these are then stored in the memory only. Usually, a few variables which are to be accessed very frequently in a program are declared with the register keyword which improves the running time of the program. An important and interesting point to be noted here is that we cannot obtain the address of a register variable using pointers.

**Example:**

|  |
| --- |
| #include <iostream>  using namespace std;    void registerStorageClass()  {        cout << "Demonstrating register class\n";        // declaring a register variable      register char b = 'G';        // printing the register variable 'b'      cout << "Value of the variable 'b'"           << " declared as register: " << b;  }  int main()  {        // To demonstrate register Storage Class      registerStorageClass();      return 0;  } |

**Output:**

Demonstrating register class

Value of the variable 'b' declared as register: G

* **mutable:** Sometimes there is a requirement to modify one or more data members of class/struct through const function even though you don’t want the function to update other members of class/struct. This task can be easily performed by using the mutable keyword. The keyword mutable is mainly used to allow a particular data member of const object to be modified. When we declare a function as const, this pointer passed to function becomes const. Adding mutable to a variable allows a const pointer to change members.

**Example:**

|  |
| --- |
| #include <iostream>  using std::cout;    class Test {  public:      int x;        // defining mutable variable y      // now this can be modified      mutable int y;        Test()      {          x = 4;          y = 10;      }  };    int main()  {      // t1 is set to constant      const Test t1;        // trying to change the value      t1.y = 20;      cout << t1.y;        // Uncommenting below lines      // will throw error      // t1.x = 8;      // cout << t1.x;      return 0;  } |

**Output:**

20