C++ Functions

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

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.

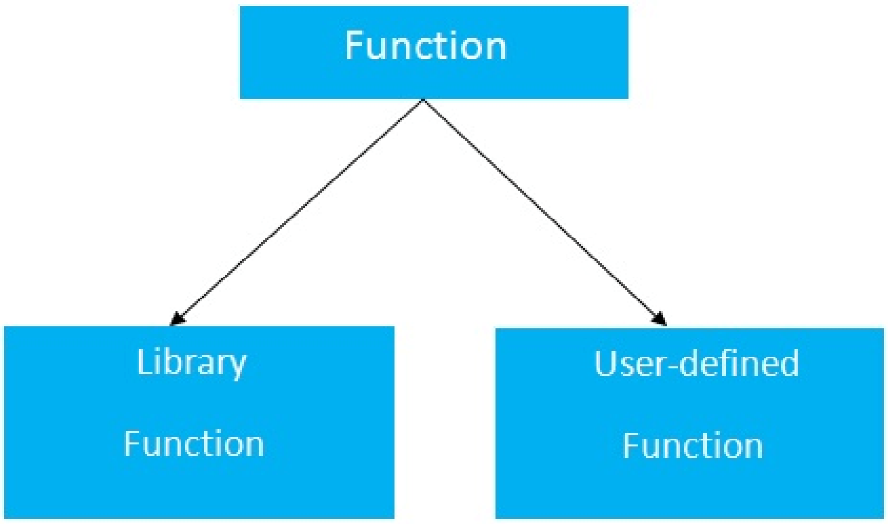
But if you use functions, you need to write the logic only once and you can reuse it several times.

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:

```
return_type function_name(data_type parameter...)
{
//code to be executed
}
```

C++ Function Example

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

```
#include <iostream>
using namespace std;
void func() {
   static int i=0; //static variable
   int j=0; //local variable
   i++;
   j++;
   cout<<"i=" << i<<" and j=" <<j<<endl;
}
```

```
int main()
{
 func();
 func();
 func();
}
```

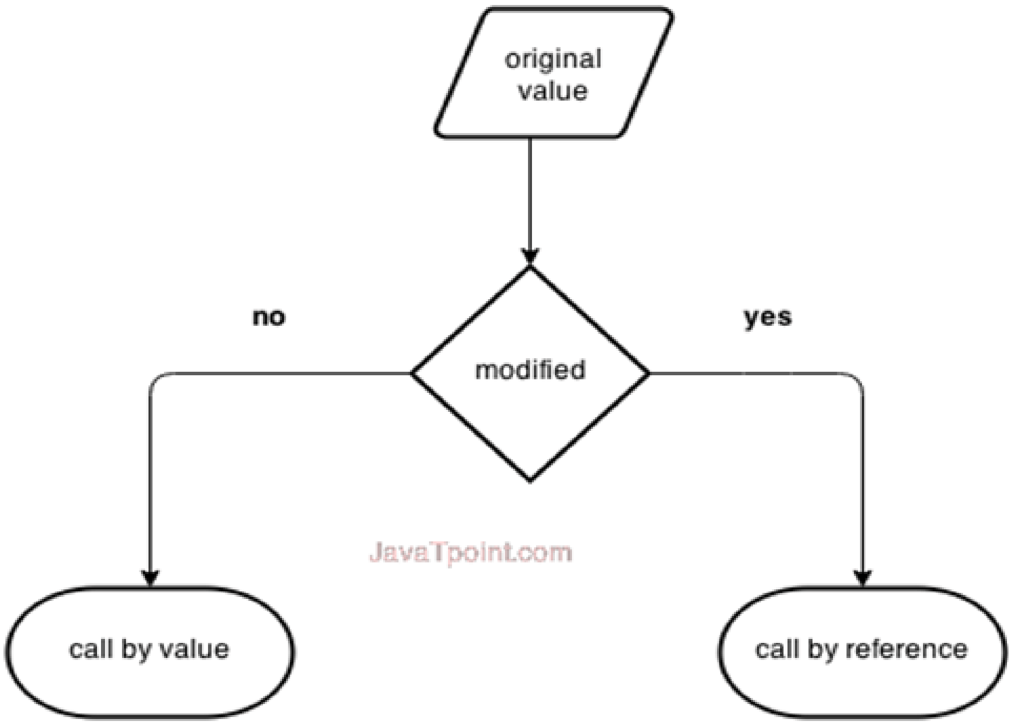
Output:

```
i= 1 and j= 1
i= 2 and j= 1
i= 3 and j= 1
```

← Prev Next →

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().

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>
using namespace std;
void swap(int *x, int *y)
{
 int swap;
 swap=*x;
 *x=*y;
 *y=swap;
}
int main()
{
 int x=500, y=100;
 swap(&x, &y);  // passing value to function
 cout<<"Value of x is: "<<x<<endl;
 cout<<"Value of y is: "<<y<<endl;
 return 0;
}
```

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

← Prev Next →

AD

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.

Let's see a simple example of recursion.

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

C++ Recursion Example

Let's see an example to print factorial number using recursion in C++ language.

```
#include<iostream>
using namespace std;
int main()
{
int factorial(int);
int fact,value;
cout<<"Enter any number: ";
cin>>value;
fact=factorial(value);
cout<<"Factorial of a number is: "<<fact<<endl;
return 0;
}
int factorial(int n)
{
if(n<0)
return(-1); /*Wrong value*/
if(n==0)
return(1);  /*Terminating condition*/
else
{
return(n*factorial(n-1));
}
}
```

Output:

```
Enter any number: 5
Factorial of a number is: 120
```

We can understand the above program of recursive method call by the figure given below:

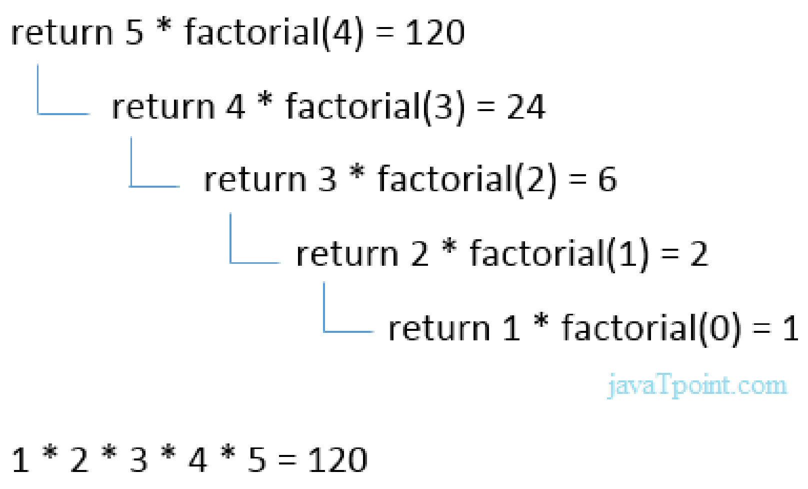

return 5 * factorial(4) = 120
return 4 * factorial(3) = 24
return 3 * factorial(2) = 6
return 2 * factorial(1) = 2
return 1 * factorial(0) = 1
javaTpoint.com
1 * 2 * 3 * 4 * 5 = 120

Fig: Recursion

← Prev Next →

C++ Storage Classes

Storage class is used to define the lifetime and visibility of a variable and/or function within a C++ program.

Lifetime refers to the period during which the variable remains active and visibility refers to the module of a program in which the variable is accessible.

There are five types of storage classes, which can be used in a C++ program

1. Automatic
2. Register
3. Static
4. External
5. Mutable

Storage Class	Keyword	Lifetime	Visibility	Initial Value
Automatic	auto	Function Block	Local	Garbage
Register	register	Function Block	Local	Garbage
Mutable	mutable	Class	Local	Garbage
External	extern	Whole Program	Global	Zero
Static	static	Whole Program	Local	Zero

Automatic Storage Class

It is the default storage class for all local variables. The auto keyword is applied to all local variables automatically.

```
{
auto int y;
float y = 3.45;
}
```

The above example defines two variables with a same storage class, auto can only be used within functions.

Register Storage Class

The register variable allocates memory in register than RAM. Its size is same of register size. It has a faster access than other variables.

It is recommended to use register variable only for quick access such as in counter.

Note: We can't get the address of register variable.

```
register int counter=0;
```

Static Storage Class

The static variable is initialized only once and exists till the end of a program. It retains its value between multiple functions call.

The static variable has the default value 0 which is provided by compiler.

```
#include <iostream>
using namespace std;
void func() {
   static int i=0; //static variable
   int j=0; //local variable
   i++;
   j++;
   cout<<"i=" << i<<" and j=" <<j<<endl;
}
int main()
{
 func();
 func();
 func();
}
```

Output:

```
i= 1 and j= 1
i= 2 and j= 1
i= 3 and j= 1
```

External Storage Class

The extern variable is visible to all the programs. It is used if two or more files are sharing same variable or function.

```
extern int counter=0;
```

← Prev Next →

Youtube For Videos Join Our Youtube Channel: Join Now

Feedback

- Send your Feedback to feedback@javatpoint.com