Introduction

A function is a group of statements that together perform a task. Every C++ program has at least one function, which is main(), and can have as many user defined function as necessary. A function declaration tells the compiler about a function's name, return type, and parameters.

Depending on whether a function is predefined or created by programmer; there are two types of function:

- Library Function
- User-defined Function

Library Function

Library functions are the built-in function in C++ programming. Programmer can use library function by invoking function directly; they don't need to write it themselves.

Example 1: Library Function

```
#include <iostream>
#include <cmath>

using namespace std;

int main()
{
   double number, squareRoot;
   cout << "Enter a number: ";
   cin >> number;

   // sqrt() is a library function to calculate square root
   squareRoot = sqrt(number);
   cout << "Square root of " << number << " = " << squareRoot;
   return 0;
}
```

Output

```
Enter a number: 26
```

Square root of 26 = 5.09902

In the example above, `sqrt()` library function is invoked to calculate the square root of a number. Like wise several other library functions in C++ are getch(), clrscr(), pow(base, exponent), strlen(string) etc.

User-defined Function

C++ allows 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 program, it all executes the codes defined in the body of function.

Example 1: Checking prime number using C like approach

```
#include <iostream>
using namespace std;

bool checkPrimeNumber(int);

int main()
{
  int n;

  cout << "Enter a positive  integer: ";
  cin >> n;

  if(checkPrimeNumber(n) == false)
    cout << n << " is a prime number.";
  else
    cout << n << " is not a prime number.";
  return 0;
}
```

```
bool checkPrimeNumber(int n)
{
  bool flag = false;

  for(int i = 2; i <= n/2; ++i)
  {
     if(n%i == 0)
     {
        flag = true;
        break;
     }
  }
  return flag;
}
```

Output:

```
Enter a positive  integer: 23
23 is a prime number.
```

Example 2: Checking prime number using class and object

```
#include<iostream>
using namespace std;

class prime
{

	public:
		int n;
		bool flag=false;

		bool checkPrimeNumber(int num){

			cout<<num;

			for(int i=2;i<n/2;i++)
			{
				if(n%i==0)
				{
					flag=true;
					break;
				}
			}
			return flag;

		}
};

int main(){
	prime p;
	cout<<"Enter a positive integer:";
```

```
	cin>>p.n;
	if(p.checkPrimeNumber(p.n)==false)
		cout<<p.n<<" is a prime number.";
	else
		cout<<p.n<<" isn't' a prime number.";
	return 0;
}
```

Output:

same as above

Example 3: Checking for a leap year

```
#include<iostream>
using namespace std;

class LeapYear
{

	private:
		int year;
	public:

		void getYear()
		{
			cout<<"Enter year:";
			cin>>year;
		}

		void checkForLeapYear()
```

```
		{
			if(year%4==0)
			{
				if(year%100==0){
					if(year%400==0)
					{
						cout<<"Leap year";
					}
					else
					{
						cout<<"Not a leap year";
					}
				}
				else{
					cout<<"Leap Year";
				}

			}
			else{
				cout<<"Not a leap year";
			}
		}
};

int main()
{
	LeapYear l;
	l.getYear();
	l.checkForLeapYear();
	return 0;
}
```

Example 4: Finding distance between two 2D points.

```
#include<iostream>
#include<math.h>
using namespace std;

class test{

	private:
		int x1,x2,y1,y2;
	public:
		void getData(){
			cout<<"Enter first coordinate X1 and Y1"<<endl;
			cin>>x1>>y1;
			cout<<"Enter second coordinate X2 and Y2"<<endl;
			cin>>x2>>y2;
		}

		void calculate()
		{
			float distance= sqrt(pow(x2-x1,2)+pow(y2-y1,2));
			cout<<"Distance:"<<distance;
		}
};

int main()
{
	test dist;
	dist.getData();
	dist.calculate();
	return 0;
}
```

Passing Arguments to Function

Passing information from calling function (Method) to the called function (method) is known as argument passing, by using argument passing, we can share information from one scope to another in C++ programming language.

We can pass arguments into the functions according to requirement. C++ supports three types of argument passing:

- Pass by Value
- Pass by Reference
- Pass by Address (Pass by Pointer)

Pass by value

In case of pass by value, Copies of the arguments are passed to the function not the variables themselves.

For example, suppose that we called our function exchange using the following code:

```
#include <iostream>
using namespace std;

void exchange(int a, int b)
{
	int temp;
	temp=a;
	a=b;
	b=temp;
	cout<<"Value of a:"<<a<<endl;
	cout<<"Value of b:"<<b<<endl;
}

int main()
{
```

```
    int x=5,y=3;
    exchange(x,y);
    cout<<"Value of X:"<<x<<endl;
    cout<<"Value of Y:"<<y<<endl;
    return 0;
}
```

Output:

```
Value of a:3
Value of b:5
Value of X:5
Value of Y:3
```

Pass by reference

In case of pass by reference, reference of the variable is passed to the function.

For example: Suppose that we called our function exchange using following code.

```
#include <iostream>
using namespace std;

void exchange(int &a, int &b)
{
    int temp;
    temp=a;
    a=b;
    b=temp;
    cout<<"Value of a:"<<a<<endl;
    cout<<"Value of b:"<<b<<endl;
}
```

```
int main()
{
	int x=5,y=3;
	exchange(x,y);
	cout<<"Value of X:"<<x<<endl;
	cout<<"Value of Y:"<<y<<endl;
	return 0;
}
```

Output:

```
Value of a:3
Value of b:5
Value of X:3
Value of Y:5
```

Pass by address

In case of pass by address, address of the variable is passed to the function.

For example: Suppose that we called our function exchange using following code.

```
#include <iostream>
using namespace std;

void exchange(int *a, int *b)
{
	int temp;
	temp=*a;
	*a=*b;
	*b=temp;
	cout<<"Value of a:"<<*a<<endl;
```

```
        cout<<"Value of b:"<<*b<<endl;
}

int main()
{
        int x=5,y=3;
        exchange(&x,&y);
        cout<<"Value of X:"<<x<<endl;
        cout<<"Value of Y:"<<y<<endl;
        return 0;
}
```

Output:

```
Value of a:3
Value of b:5
Value of X:3
Value of Y:5
```

Returning From Function:

Functions can return value as well as memory address. We can return value from functions in three ways:

- Return by Value
- Return by reference
- Return by address (pointer)

Return by value

Return by value is the simplest and safest return type to use. When a value is returned by value, copy of that value is returned to the caller.

Example: An example to find minimum between two numbers

```
#include<iostream>
using namespace std;

int     min(int a, int b)
{
        if(a<b)
                return a;
        else
                return b;
}

int main()
{
        int x,y;
        cout<<"Enter two numbers:"<<endl;
        cin>>x>>y;
        int minValue=min(x,y);
```

```
	cout<<"Min value="<<minValue;
	return 0;
}
```

Output:

```
Enter two numbers:
56
78
Min value=56
```

Return by reference

In return by reference, a reference of a variable is returned to the calling function. Return by reference is comparatively fast than return by value.

Example: An example to find minimum between two numbers

```
#include<iostream>
using namespace std;

int& min(int &a, int &b)
{
	if(a<b)
		return a;
	else
		return b;
}

int main()
{
	int x,y;
	cout<<"Enter two numbers:"<<endl;
```

```
        cin>>x>>y;
        int minValue=min(x,y);
        cout<<"Min value="<<minValue;
        return 0;
}
```

Output:

```
Enter two numbers:
56
78
Min value=56
```

Return by address

Return by address returns the address of variable to the calling function. Just like pass by address, return address can only return the address of a variable.

Example: An example to find minimum between two numbers

```
#include<iostream>
using namespace std;

int* min(int *a, int *b)
{
        if(*a<*b)
                return a;
        else
                return b;
}

int main()
{
```

```
        int x,y;
        cout<<"Enter two numbers:"<<endl;
        cin>>x>>y;
        int *minValue=min(&x,&y);
        cout<<"Min value="<<*minValue;
        return 0;
}
```

Output:

```
Enter two numbers:
56
78
Min value=56
```

Function Overloading:

Function overloading is a C++ programming feature that allows us to have more than one function having the same name but different parameters. They should have different type, number or sequence of parameters.Function overloading is a compile-time polymorphism.

For example: We can have following functions in the same program.

we can have following functions in the same scope.

sum(int num1, int num2)
sum(int num1, int num2, int num3)
sum(int num1, double num2)

Different ways to Overload a Function

- By changing number of Arguments.
- By having different types of argument.

Function Overloading: Different Number of Arguments

In this type of function overloading we define two functions with same names but different number of parameters of the same type.

For example, in the below mentioned program we have made two area() functions to return area of rectangle and square

```
#include<iostream>
using namespace std;

int area(int length, int breadth)
{
	return (length*breadth);
}

int area(int length)
{
	return length*length;
}

int main()
{
	int sa,ra,l,b;
	cout<<"Enter length and breadth of a rectangle:"<<endl;
	cin>>l>>b;
	ra=area(l,b);
	cout<<"Enter length of a side of an square:"<<endl;
	cin>>l;
	sa=area(l);
	cout<<"Area of rectangle:"<<ra<<endl;
	cout<<"Area of square:"<<sa<<endl;
```

```
    return 0;
}
```

output:

```
Enter length and breadth of a rectangle:
5
6
Enter length of a side of an square:
9
Area of rectangle:30
Area of square:81
.
```

Function Overloading: Different Datatype of Arguments

In this type of overloading we define two or more functions with same name and same number of parameters, but the type of parameter is different.

For example: In this program, we have two multiply() function, first one gets two integer arguments and second one gets two float arguments.

```
#include<iostream>
using namespace std;

int mul(int a, int b)
{
```

```
        return (a*b);
}

float mul(float a, float b)
{
        return (a*b);
}

int main()
{
        int x=5,y=2;
        float n=7.2,m=3.4;
        cout<<"Product of integers:"<<mul(x,y)<<endl;
        cout<<"Product of reals:"<<mul(n,m)<<endl;
        return 0;

}
```

Output:

```
Product of integers:10
Product of reals:24.48
```

Default Arguments

When declaring a function we can simply a default value for each of the last parameters which are called default arguments. This value will be used if the corresponding arguments is left blank when calling to the function. To do that, we simply have to use the assignment operator and a value for the arguments in the function declaration. If a value for that parameter is not passed when the function

is called, the default value is used, but if a value is specified this default value is ignored and the passed value is used instead.

For example:

```
#include<iostream>
using namespace std;

int divide (int a, int b=2)
{
	int r;
	r=a/b;
	return r;
}

int main()
{
	cout<<divide(12)<<endl;
	cout<<divide(20,4);
	return 0;

}
```

Output:

```
6
5
```

Common mistakes when using Default argument

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

   The above function will not compile. You cannot miss a default argument in between two arguments.
   In this case, `c` should also be assigned a default value.

2. `void add(int a, int b = 3, int c, int d);`

   The above function will not compile as well. You must provide default values for each argument after `b`.
   In this case, `c` and `d` should also be assigned default values.
   If you want a single default argument, make sure the argument is the last one. `void add(int a, int b, int c, int d = 4);`

Inline Function

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 function is less than the switching time from the caller function to called function (callee).

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

The syntax for defining the function inline is:

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

The inline keyword sends a request to compiler not a command. Sometimes compiler may ignore the request if

- The function definition is too long or too complicated.
- If a return statement exists.
- If a loop, a switch or a goto statement exists.
- If function contains static variable.
- If inline functions are recursive.

Example:

```
#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 inline function in the class then just declare the function inside the class and define it outside the class using inline keyword.

For example:

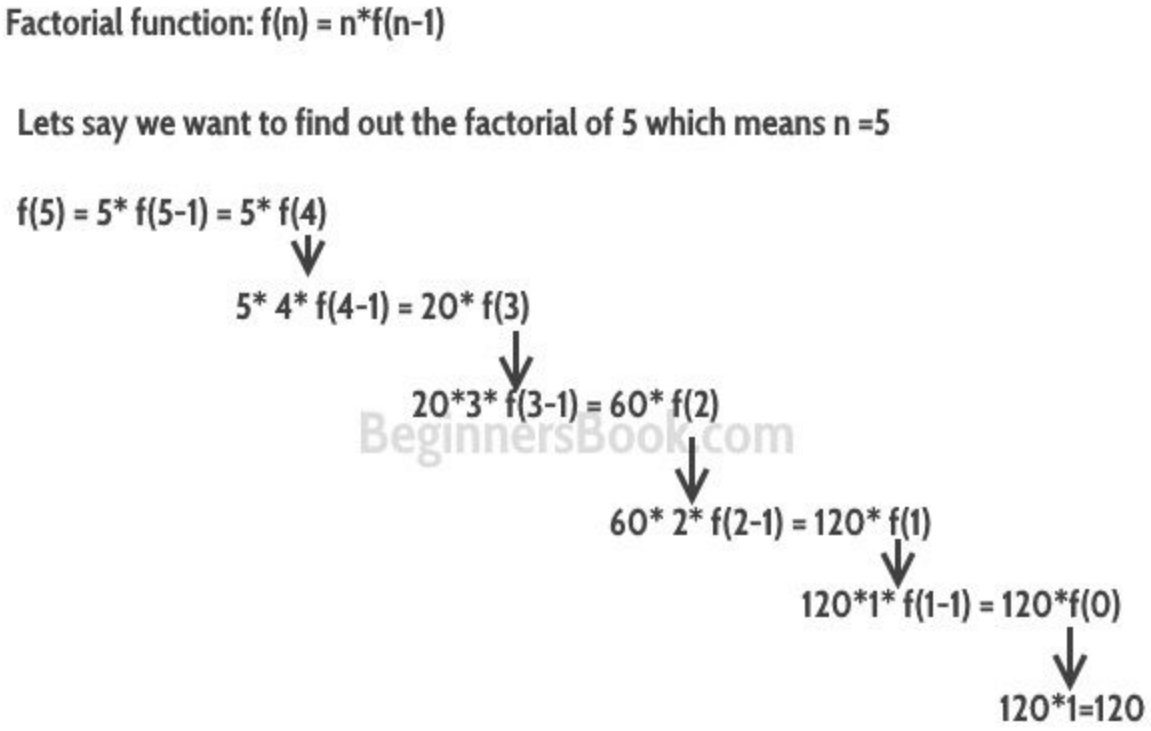
```
class S
{
public:
   int square(int s); // declare the function
};

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

}
```

Advantages:

- Function Overhead doesn't occur.
- It saves the overhead of push/pop variables on the stack when the function is called.
- It also saves overhead of a return call from a function.
- Increases the locality of reference by utilizing instruction cache.

Recursive Function:

The process in which a function calls itself is known as recursion and the corresponding function is called the recursive function. The popular example to understand the recursion is factorial function.



Example: C++ program to find the factorial

```
#include <iostream>
using namespace std;

int f(int n){
  if (n <= 1)
      return 1;
  else
     return n*f(n-1);
}
```

```
int main(){
  int num;
  cout<<"Enter a number: ";
  cin>>num;
  cout<<"Factorial of entered number: "<<f(num);
  return 0;
}
```

Output:

```
Enter a number: 5
Factorial of entered number: 120
```

Example: C++ program to find the Fibonacci series

```
#include <iostream>
using namespace std;
int fib(int x) {
  if((x==1)||(x==0)) {
    return(x);
  }else {
    return(fib(x-1)+fib(x-2));
  }
}
int main() {
  int x , i=0;
  cout << "Enter the number of terms of series : ";
  cin >> x;
  cout << "\nFibonnaci Series : ";
  while(i < x) {
```

```
    cout << " " << fib(i);
    i++;
  }
  return 0;

}
```

output:

Enter the number of terms of series : 15
Fibonnaci Series : 0 1 1 2 3 5 8 13 21 34 55 89 144 233 377