Functions

A function is a single comprehensive unit that performs a specified task. This specified task is repeated each time the function is called. Functions break large programs into smaller tasks. They increase the modularity of the programs and reduce code redundancy.
Like in C, C++ programs also should contain a main function, where the program always begins execution. The main function may call other functions, which in turn will again call other functions.
When a function is called, control is transferred to the first statement of the function body. Once the statements of the function get executed (when the last closing bracket is encountered) the program control return to the place from where this function was called.

Function Prototype (Function declaration)
Function prototype lets the compiler know the structure of function in terms of its name, number and type of arguments and its return type.

Syntax:
```
        return-type function-name(datatype1, datatype2, …,datatype n);
```

Function Call
Function call is the process of making use of function by providing it with the parameters it needs. We call a function as follows.
```
        function-name (argument1, argument2, .. ,argument n);
```

Function Definition
Function definition is a process of defining how it does what it does or in other words, during function definition, we list the series of codes that carry out the task of the function. A function is defined as follows,

```
return-type function-name(datatype1 variable1, datatype2 var2, …., datatype n var n)
{
        ……………… ;
        ………………….; //body of the function
        ……………….;
}
```

Default Arguments

In C++, a function can be called without specifying all its arguments. But it does not work on any general function. The function declaration must provide default values for those arguments that are not specified. When the arguments are missing from function call, default value will be used for calculation.

```
#include<iostream.h>
float interest(int p, int t = 5, float r = 5.0);
main()
{
        float rate, i1,i2,i3;
        int pr , yr;
```

```
	cout<<"Enter principal, rate and year";
	cin>>pr>>rate>>yr;
	i1=interest(pr ,yr ,rate);
	i2=interest(pr , yr);
	i3=interest(pr);
	cout<<i1<<i2<<i3;
	return(0);
}

float interest(int p, int t, float r)
{
	return((p*t*r)/100);
}
```

In the above program, t and r has default arguments. If we give, as input, values for pr, rate and yr as 5000, 10 and 2, the output will be

```
		1000 500 1250
```

NOTE: The default arguments are specified in function declaration only and not in function definition. Only the trailing arguments can have default values. We must add defaults from right to left. We cannot provide a default value to a particular argument at the middle of an argument list. Default arguments are used in the situation where some arguments have same value. For eg., interest rate in a bank remains same for all customers for certain time.

Inline Functions

We say that using function s in a program is to save some memory space because all the calls to the functions cause the same code to be executed. However, every time a function is called, it takes a lot of extra time in executing a series of instructions. Since the tasks such as jumping to the function, saving registers, pushing arguments into the stack and returning to the calling function are carried out when a function is called. When a function is small, considerable amount of time is spent in such overheads.

C++ has a solution for this problem. To eliminate the cost of calls to small functions, C++ proposed a new feature called INLINE function.
When a function is defined as inline, compiler copies it s body where the function call is made, instead of transferring control to that function.
A function is made inline by using a keyword "inline" before the function definition.

Eg.

```
		inline void calculate_area(int l,int b)
		{
			return(l * b);
		}
```

It should be noted that, the inline keyword merely sends request, not a command, to a compiler. The compiler may not always accept this request. Some situations where inline expansion may not work are

- for functions having loop, switch or goto statements
- for recursive functions
- functions with static variables
- for functions not returning values, if a return statement exists

Inline functions must be defined before they are called.

Eg.

```
#include<iostream.h>

inline float lbtokg(float lbs)
{
        return (0.453 * lbs);
}


main()
{
        float lbs, kgs;
        cout<<"Enter weight in lbs:";
        cin>>lbs;
        kgs=lbtokg(lbs);
        cout<<"Weight in kg is "<<kgs;
        return (0);
}
```

Exercise:

When do we use inline function? Explain with example.
When do we use default argument? Explain with example.

Function Overloading

Function that share the same name are said to be **overloaded functions** and the process is referred to as **function overloading**. i.e. function overloading is the process of using the same name for two or more functions. Each redefinition of a function must use different type of parameters, or different sequence of parameters or different number of parameters. The number, type or sequence of parameters for a function is called the function signature. When we call the function, appropriate function is called based on the parameter passed. Two functions differing only in their return type can not be overloaded. For eg-
int add(int , int) and float add(int, int)

A function call first matches the declaration having the same number and type of arguments and then calls the appropriate function for execution. A best match must be unique. The function selection will involve the following steps:

- the compiler first tries to find an exact match in which the types of actual arguments are the same and uses that function
- if an exact match is not found, the compiler uses the integral promotion to the actual parameters, such as,
    - char to int
    - float to double to find the match
- If both of the above fail, the compiler tries to use the built-in conversions and then uses the function whose match is unique.

```
#include<iostream.h>
//function declaration
float perimeter(float);
int perimeter(int,int);
int perimeter(int,int,int);

main()
{
        cout<<"Perimeter of a circle: "<<perimeter(2.0)<<endl;
        cout<<"Perimeter of a rectangle: "<<perimeter(10,10)<<endl;
        cout<<"Perimeter of a triangle: "<<perimeter(5,10,15);
        return (0);
}

//function definition
float perimeter(float r)
{
        return(2*3.14*r);
}
int perimeter(int l,int b)
{
        return(2*(l+b));
}
int perimeter(int a,int b,int c)
{
        return(a+b+c);
}
```

In the above program, a function "perimeter" has been overloaded. The output will be as follows:

```
        Perimeter of a circle 12.56
        Perimeter of a rectangle 40
        Perimeter of a triangle 30
```

Recursive function
Recursion is a powerful technique of writing complex algoriths in an easy way. It defines the problem in terms of itself. In this technique , a large problem is divided into smaller problem of similar nature as original problem.so that the smaller problem is easy to solve and in the most case they can be solved easily. Hence to implement this techinique, a programming language support the function that is capable of calling itself. C++ support such function and these function are called recursive functions

For example: to find the factorial of a given number

```
Int mani()
{
        int num;
        cout<<"Enter a number";
        cin>>num;
        int f=fact(num);
        cout>>"the factorial of given number is"<<f;
        getch();
        return 0;

}

int fact(int num)
{
        If(num==0)
                Return 1;
        Else
                Return (num*fact(num-1));
}
```

Passing Arguments to the Function

We can pass arguments to the function in three ways: Pass by value, Pass by reference, and pass by pointer:

Pass By Value
While calling a function, when we pass values by copying variables, it is known as Pass by Value. In this method, a copy of the variable is passed. Changes made in a copy of variable never modify the value of variable outside the function. Values of variables are passed using a straightforward method.
//Program to find area and perimeter of the rectangle using pass by value method

```
#include<iostream>
using namespace std;
int area(int l,int b)
{
  return l*b;
}
int peri(int l,int b)
{
    return 2*(l+b);
}
int main()
{
    int l=5,b=5;
    cout<<"\nThe area of the rectangle is "<<area(l,b);
    cout<<"\nThe perimeter of the rectangle is "<<peri(l,b);
}
```

Pass by Reference

While calling a function, in programming language instead of copying the values of variables, the address of the variables is used which is known as Pass By References. In this method, a variable itself is passed. Allows you to make changes in the values of variables by using function calls.

```
//Finding area and perimeter of a rectangle using pass by reference
#include<iostream>
using namespace std;
void cal(int l,int b,int &a,int &p)
{
    a=l*b;
    p=2*(l+b);
}
int main()
{
    int l=5,b=2,area,peri;
    cal(l,b,area,peri);
    cout<<"\nThe area of a rectangle is "<<area;
    cout<<"\nThe perimeter of a rectangle is "<<peri;
}
```

Pass by Pointer

Working principle of pass by pointer is same as the pass by reference. Only the difference is that instead of using the reference variable we use the pointer variable in function definition and pass address of the variable from function call statement. Therefore, while calling a function, in programming language instead of copying the values of variables, the address of the variables is used in pass by pointer.

```
//passing by pointer
#include<iostream>
using namespace std;
void cal(int l,int b,int *a,int *p)
{
   *a=l*b;
   *p=2*(l+b);
}
int main()
{int l=5,b=2,area,peri;
cal(l,b,&area,&peri);
cout<<"\nThe area of a rectangle is "<<area;
cout<<"\nThe perimeter of a rectangle is "<<peri;
}
```

Returning from Function

Function can return value as well as memory location. We can return values from function in three ways: return by value, by reference and by pointer. Returning values from a function to its caller by value, address, or reference works almost exactly the same was asp passing parameters to a function does. The primary difference between the two is simply that the direction of data flow is reversed.

Return by Value

Return by value is the simplest and safest return type to use. When a value is returned by value, a copy of that value is returned to the caller. As with pass by value, you can return by value literals, variables, or expressions, which makes return by value very flexible.

```
//A program to find the square of the given number using return by value
#include<iostream>
using namespace std;
int square(int num)
{
   return num*num;
}
int main()
{int num=5;
cout<<"The square of the given number is "<<square(num);
  }
```

Return by Reference

Similar to pass by address, values returned by reference must be variables (you should not return a reference to a literal or an expression that resolves to a temporary value, as those will go out of scope at the end of the function and you'll end up returning a dangling reference). When a variable is returned by reference, a reference to the variable is passed back to the caller. The caller can then use this reference to continue modifying the variable, which can be useful at times. Return by reference is also fast, which can be useful when returning structs and classes.

```
//A program to find the square of the number using return by reference
#include<iostream>
using namespace std;
 /*
 int* square(int num,int *sq)
 {*sq=num*num;
    return sq ;
 }
 int main()
 {int num=5,sq;
 cout<<"The square is "<<*square(num,&sq);
 cout<<"\nThe value of sq:"<<sq;
  }
```

Return by Pointer or Address

Returning by address involves returning the address of a variable to the caller. Similar to pass by pointer, return by pointer can only return the address of a variable, not a literal or an expression (which don't have addresses). Because return by address just copies an address from the function to the caller, return by address is fast.

```
//A program to find the square of the given number using Return by pointer
#include<iostream>
using namespace std;
 int* square(int num,int *sq)
 {*sq=num*num;
     return sq ;
 }
 int main()
 {int num=5,sq;
 cout<<"The square is "<<*square(num,&sq);
 cout<<"\nThe value of sq:"<<sq;
  }
```