Lecture 10
Functions

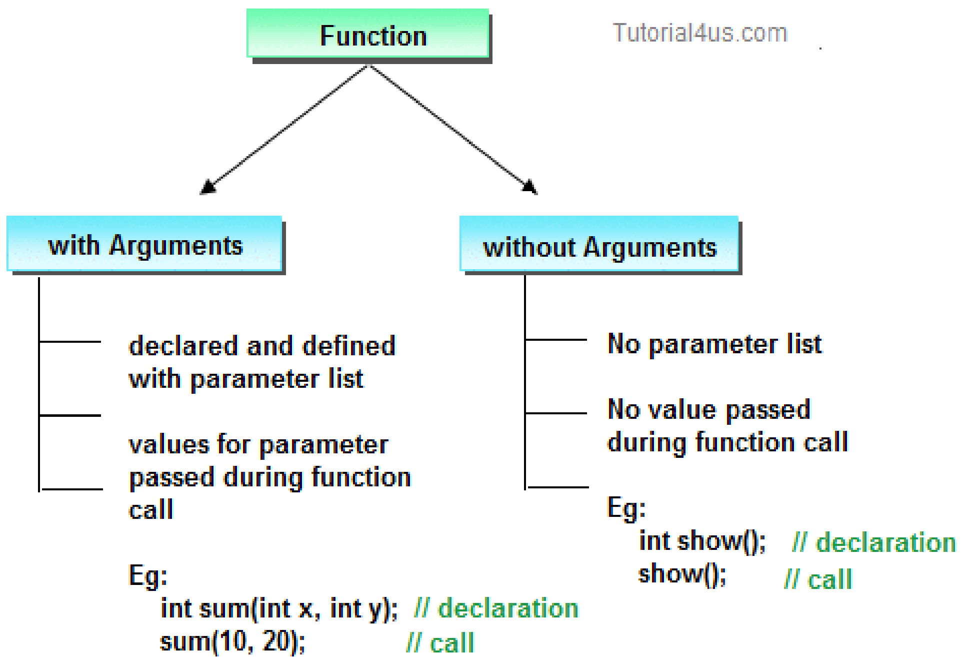
A *function* is a group of statements that together perform a task. Every C++ program has at least one function, which is main(), and all the most trivial programs can define additional functions. You can divide up your code into separate functions.

A C++ function definition consists of a function header and a function body. Here are all the parts of a function :

- *Return Type* − A function may return a value. The return_type is the data type of the value the function returns. Some functions perform the desired operations without returning a value. In this case, the return_type is the keyword void.
- *Function Name* − This is the actual name of the function. The function name and the parameter list together constitute the function signature.
- *Parameters* − A parameter is like a placeholder. When a function is invoked, you pass a value to the parameter. This value is referred to as actual parameter or argument. The parameter list refers to the type, order, and number of the parameters of a function. Parameters are optional; that is, a function may contain no parameters.
- *Function Body* − The function body contains a collection of statements that define what the function does.

Syntax of Function

```
return-type function-name (parameters)

{

 // function-body

}
```


Function
Tutorial4us.com
with Arguments
without Arguments
declared and defined with parameter list
values for parameter passed during function call
Eg:
int sum(int x, int y); // declaration
sum(10, 20); // call
No parameter list
No value passed during function call
Eg:
int show(); // declaration
show(); // call

//Declaring, Defining and Calling Function

```
#include <iostream>
using namespace std;
int sum (int x, int y);   //declaring function
int main()
{
 int a = 10;
 int b = 20;
 int c = sum (a, b);   //calling function
 cout << c;
return 0;}
int sum (int x, int y)   //defining function
{
 return (x + y);
}
```

```
// Program to find the maximum number between two
numbers and print it
#include <iostream>
using namespace std;
```

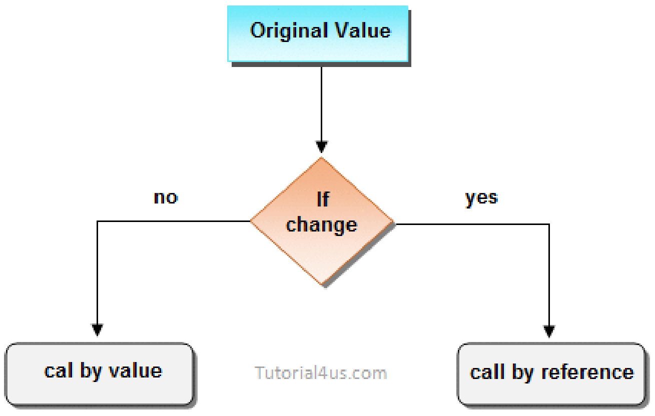
```
// function declaration
int max(int num1, int num2);

int main () {
   // local variable declaration:
   int a = 100;
   int b = 200;
   int ret;

   // calling a function to get max value.
   ret = max(a, b);
   cout << "Max value is : " << ret << endl;

   return 0;
}
// function returning the max between two numbers
int max(int num1, int num2) {
   // local variable declaration
   int result;

   if (num1 > num2)
      result = num1;
   else
      result = num2;

   return result;
}
```

Calling a Function

Functions are called by their names. If the function is without argument, it can be called directly using its name. But for functions with arguments, we have two ways to call them,

1.Call by Value

2.Call by Reference



Call by Value

In this calling technique we pass the values of arguments which are stored or copied into the formal parameters of functions. Hence, the original values are unchanged only the parameters inside function changes.

```
 void calc(int x);
int main()
 {
    int x = 10;
   calc(x);
   cout<<x;
  }

   void calc(int x)
 {
 x = x + 10 ;
}
```

Output : 10

In this case the actual variable x is not changed, because we pass argument by value, hence a copy of x is passed, which is changed, and that copied value is destroyed as the function ends(goes out of scope). So the variable x inside main() still has a value 10.

But we can change this program to modify the original x, by making the function calc() return a value, and storing that value in x.

```
 int calc(int x);
 int main()
       {
  int x = 10;
 x = calc(x);
cout<< x;
}

int calc(int x)
{
 x = x + 10 ;
 return x;
}
```

Output : 20

Call by Reference

In this we pass the address of the variable as arguments. In this case the formal parameter can be taken as a reference or a pointer, in both the case they will change the values of the original variable.

```
void calc(int *p);
int main()
{
 int x = 10;
 calc(&x);     // passing address of x as argument
 cout<< &x;
}

void calc(int *p)
{
 *p = *p + 10;
}
```

Output : 0x7fff31d5107c

The same example with little change:

```
void calc(int *p);
int main()
{
 int x = 10;
 calc(&x);     // passing address of x as argument
 cout<< x;
}

void calc(int *p)
{
 *p = *p + 10;
}
```

Output : 20

Default value

```
int sum(int a,int b=10,int c=20);
int main(){
cout<<sum(1)<<endl;
cout<<sum(1,2)<<endl;
cout<<sum(1,2,3)<<endl;
return 0}
int sum(int a, int b, int c){
int z;
z=a+b+c;
return z;}
```

Output:

```
31
23
6
```

Procedural programming

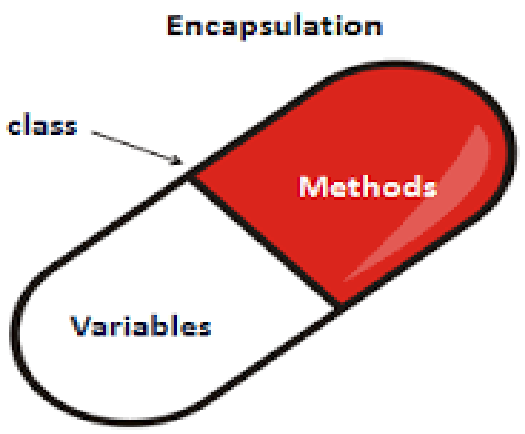
A procedural language is a type of computer programming language that specifies a series of well-structured steps and procedures within its programming context to compose a program. It contains a systematic order of statements, functions, subroutines and commands to complete a computational task or program.

The procedural language separates a program within variables, functions, statements and conditional operators. Procedures or functions are implemented on the data and variables to perform a task. These procedures can be called/invoked anywhere between the program hierarchy, and by other procedures as well. A program written in procedural language contains one or more procedures.

Under this definition, we get:

- C
- C++
- Java
- Python
- JavaScript
- BASIC
- Perl

Encapsulation



Encapsulation is a process of combining data members and functions in a single unit called class. This is to prevent the access to the data directly, the access to them is provided through the functions of the class. It is one of the popular feature of Object Oriented Programming(OOPs) that helps in data hiding.

How Encapsulation is achieved in a class?

To do this:
1) Make all the data members private.
2) Create public setter and getter functions for each data member in such a way that the set function set the value of data member and get function get the value of data member.

```
#include<iostream.h>

class sum
{
private: int a,b,c;

public:
void add()
{
clrscr();
cout<<"Enter any two numbers: ";
cin>>a>>b;
c=a+b;
cout<<"Sum: "<<c;
}
};
int main()
{
sum s;
s.add();
}
```

Output:

```
Enter any two number:
4
5
Sum: 9
```

Above class adds numbers together, and returns the sum. The public member add is the interface to the outside world and a user needs to know it to use the class. The private members a, b and c are something that are hidden from the outside world, but are needed for the class to operate properly.