Functions In C++

- Function is a module which performs a specific task
- Functions are called by name
- Rules for giving function name is same as variable name
- Function can take 0 or more parameters
- Function can return single value
- Void function don't return any value
- Default return type is int

Simple Function:

```
#include <iostream>
using namespace std;
int maxim(int a, int b){
    return a>b? a:b;
}
int main()
{
   cout<<maxim(12,13);
}
```

Example of Simple Function

```
#include <iostream>
using namespace std;
void display()
```

```
{
cout<<"Hello";
}
int main()
{
display();
return 0;
}
```

Example of Function with Arguments

```
#include<iostream>
using namespace std;
float add(float x,float y)
{
float z;
z=x+y;
return z;
}
int main()
{
float x=2.3,y=7.9,z;
z=add(x,y);
cout<<z<<endl;
return 0;
}
```

Example of function to find Maximum of 3 number

```
#include<iostream>
using namespace std;
int maxim(int a,int b,int c)
{
if(a>b && a>c)
return a;
else if(b>c)
return b;
else
return c;
}
int main()
{
int a,b,c,d;
cout<<"Enter three no.s ";
cin>>a>>b>>c;
d=maxim(a,b,c);
cout<<"Maximum is "<<d<<endl;
return 0;
}
```

Function Overloading :

Function with the same name but with different parameters or signatures.

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

```
int add(int a, int b){
   return a+b;
}
float add(float a, float b){
   return a+b;
}
double add( double a, double b){
   return a+b;
}
int main()
{
  cout<<add(12,13)<<endl;
  cout<<add(12.3,34.4)<<endl;
}
```

Function Overloading

• If More than one functions can have same name, but different parameter list, then they

are overloaded functions

• Return the is not considered in overloading

• Function overloading is used for achieving compile time polymorphism

Program to Demonstrate Function Overloading using Sum function

```
#include<iostream>
using namespace std;
int sum(int a,int b)
{
return a+b;
```

```
}
float sum(float a,float b)
{
return a+b;
}
int sum(int a,int b,int c)
{
return a,b,c;
}
int main()
{
cout<<sum(10,5)<<endl;
cout<<sum(12.5f,3.4f)<<endl;
cout<<sum(10,20,3)<<endl;
return 0;
}
```

Function Template

- Function template are used for defining generic functions
- They work for multiple datatypes
- Datatype is decided based on the type of value passed
- Datatype is a template variable
- Function can have multiple template variables

Simple Program

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

```
template <class T>
T add(T a, T b){
   return a+b;
}

int main()
{
  cout<<add(12,13)<<endl;
  cout<<add(12.3,34.4)<<endl;
}
```

Default Arguments

• Parameters of a function can have default values

• If a parameter is default then , passing its value is options

• Function with default argument can be called with variable number of argument

• Default values to parameters must be given from right side parameter

• Default arguments are much useful in constructors

• Default arguments are useful for defining overloaded functions

Example of Default Arguments:

```
#include <iostream>
using namespace std;
template <class T>
T add(T a, T b=0){
   return a+b;
}
```

```
int main()
{
  cout<<add(12,13)<<endl;
  cout<<add(12.3,34.4)<<endl;
}
```

Parameter Passing Methods

Three parameter passing methods are supported by C++

Pass-By-Value : values of Actual parameters are passed to formal parameters. Actual

parameters cannot be modified by function

Pass-By-Address: Address of Actual Parameters are passed to a function, formal

parameters must be pointers. Function can indirectly access actual parameters.

Pass-By-Reference: Actual parameters are passed as reference to formal parameters,

function can modify actual parameters.

Program for Call by Value

• Value of actual parameters are copied in formal parameters

• If any changes done to formal parameters in function, they will not modify actual

parameters

```
Void swap(int a, int b)
{
int temp;
temp=a;
a=b;
b=temp;
}
```

```
Int main()
{
int x=10, y=20;
swap(x,y);
cout<<x<<y;
}
```

Call by Address

• Address pf actual parameters are passed.

• Formal parameters must be pointers

• Formal parameters ca indirectly access actual parameters.

• Changes done using formal parameters will reflect in actual parameters

```
Void swap(int *x, int *y)
{
int temp;
temp=*x;
*x=*y;
*y=temp;
}
Int main()
{
int a=10, b=20;
swap(&a,&b);
cout<<a<<b;
}
```

Call by Reference

• Actual parameters are passed as reference

• Formal parameters can directly access actual parameters

• Function call is converted into inline function, if not possible it will become call by

address

• Reference don’t take extra memory

• Syntax is same as Call by Value except, formal parameters are reference

```
Void swap(int &a, int &b)
{
int temp;
temp=a;
a=b;
b=temp;
}
Int main()
{
int x=10, y=20;
swap(x,y);
cout<<x<<y;
}
```

Return by Address

• A function can return address of memory

• It should not return address of local variables, which will be disposed after function ends

• It can return address of memory allocated in heap

```
Int * fun(int n)
{
int *p=new int[n];
for(int i=0;i<n;i++)
p[i]=i+1;
return p;
}
Int main()
{
int *ptr=fun(5);
for(int i=0;i<5;i++)
cout<<i<<endl;

}
```

Return by Reference

• A function cal return reference

• It should not return reference of its local variables

• It can return formal parameters if they are reference

```
Int & fun(int &a)
{
cout<<a;
return a;
}
Int main()
```

```
{
int x=10;
fun(x)=25;
cout<<x;
}
```

Static variables

• They have local scope but remain in memory thru out the execution of program

• They are created in code section

• They are history-sensitive

```
Void fun()
{
static int v=0;
int a=10;
v++:
cout<<a<<" "<<v;
}
Int main()
{
fun();
fun();
fun();
}
```

Program for Linear Search using Functions

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

```
int Search(int A[],int n,int key)
{
for(int i=0;i<n;i++)
if(key==A[i])
return i;
return 0;
}
int main()
{
int A[]={2,4,5,7,10,9,13};
int k;
cout<<"Enter an Element to be Searched:";
cin>>k;
int index=Search(A,7,k);
cout<<"Element found at index :"<<index<<endl;

}
```

ThankYou☺