LAB 10

Summary

Items	Description
Course Title	Programming Fundamentals
Lab Title	Functions- Call by reference and recursive call
Duration	3 Hours
Operating System /Tool/Language	Visual studio
Objective	To get familiar with use of functions call by reference in C++

Syntax:

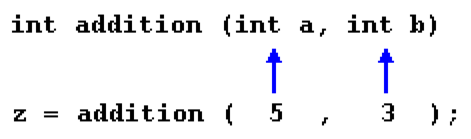
//DEFINTION
type name (& parameter1, & parameter2,….)
{ statement(s) }

//CALLING
name (parameter 1 , parameter2 ,…..);

Until now, in all the functions we have seen, the arguments passed to the functions have been passed *by value*. This means that when calling a function with parameters, what we have passed to the function were copies of their values but never the variables themselves. For example, suppose that we called our first function addition using the following code:

```
int x=5, y=3, z;
z = addition ( x , y );
```

What we did in this case was to call to function addition passing the values of x and y, i.e. 5 and 3 respectively, but not the variables x and y themselves.



```
int addition (int a, int b)

z = addition (  5  ,   3  );
```

This way, when the function addition is called, the value of its local variables a and b become 5 and 3 respectively, but any modification to either a or b within the function addition will not have any effect in the values of x and y outside it, because variables x and y were not themselves passed to the function, but only copies of their values at the moment the function was called.

<u>Example</u>

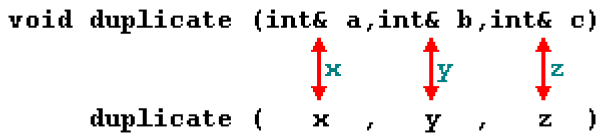
// passing parameters by reference

Sample program #01 :

```
// pf10_1.cpp : Defines the entry point for the console application.
//

#include "stdafx.h"
#include<iostream>
using namespace std;
void duplicate (int& a, int& b, int& c)
{
  a=a*2;
  b=b*2;
  c=c*2;
}


int _tmain(int argc, _TCHAR* argv[])
{
       int x=1, y=3, z=7;
        duplicate (x, y, z);
       cout << "x=" << x << ", y=" << y << ", z=" << z;
       system("pause");
       return 0;
}
```

The declaration of duplicate the type of each parameter was followed by an ampersand sign (&). This ampersand is what specifies that their corresponding arguments are to be passed *by reference* instead of *by value*.

When a variable is passed by reference we are not passing a copy of its value, but we are somehow passing the variable itself to the function and any modification that we do to the local variables will have an effect in their counterpart variables passed as arguments in the call to the function.



To explain it in another way, we associate a, b and c with the arguments passed on the function call (x, y and z) and any change that we do on a within the function will affect the value of x outside it. Any change that we do on b will affect y, and the same with c and z.

If when declaring the following function:

```
void duplicate (int& a, int& b, int& c)
```

we had declared it this way:

```
void duplicate (int a, int b, int c)
```

i.e., without the ampersand signs (&), we would have not passed the variables by reference, but a copy of their values instead, and therefore, the output on screen of our program would have been the values of x, y and z without having been modified.

Passing by reference is also an effective way to allow a function to return more than one value.

Sample program #02:

```
// pf10_2.cpp : Defines the entry point for the console application.
//

#include "stdafx.h"
#include<iostream>
using namespace std;
void prevnext (int x, int& prev, int& next)
{
  prev = x-1;
  next = x+1;
}


int _tmain(int argc, _TCHAR* argv[])
{
        int x=100, y, z;
  prevnext (x, y, z);
  cout << "Previous=" << y << ", Next=" << z;

       system("pause");
       return 0;
}
```

Default values in Parameters

When declaring a function we can specify a default value for each of the last parameters. This value will be used if the corresponding argument 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.

Sample program #03:

```
#include "stdafx.h"
#include<iostream>
using namespace std;
int divide (int a, int b=2)
{
  int r;
  r=a/b;
  return r;
        }


int _tmain(int argc, _TCHAR* argv[])
{
       cout << divide (12);
       cout << endl;
       cout << divide (20,4);
       system("pause");
       return 0;
}
```

Overloaded Functions

In C++ two different functions can have the same name if their parameter types or number are different. That means that you can give the same name to more than one function if they have either a different number of parameters or different types in their parameters.

Sample program #04:

```
// pf10_4.cpp : Defines the entry point for the console application.
//

#include "stdafx.h"
#include<iostream>
using namespace std;
int operate (int a, int b)
{
  return a*b;
}
```

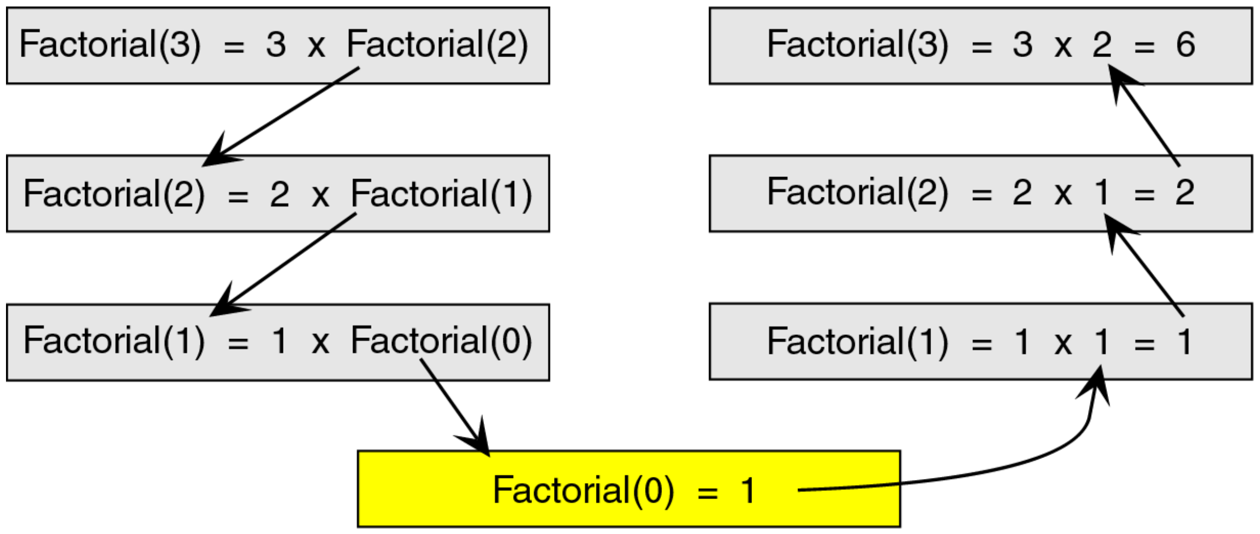
```
float operate (float a, float b)
{
  return a/b;
}



int _tmain(int argc, _TCHAR* argv[])
{

       int x=5,y=2;
       float n=5.0,m=2.0;
       cout << operate (x,y);
       cout << "\n";
        cout << operate (n,m);
       cout << "\n";

       system("pause");
       return 0;
}
```

In this case we have defined two functions with the same name, operate, but one of them accepts two parameters of type int and the other one accepts them of type float. The compiler knows which one to call in each case by examining the types passed as arguments when the function is called. If it is called with two ints as its arguments it calls to the function that has two int parameters in its prototype and if it is called with two floats it will call to the one which has two float parameters in its prototype.

The behavior of a call to operate depends on the type of the arguments passed because the function has been *overloaded*. Notice that a function cannot be overloaded only by its return type. At least one of its parameters must have a different type.

Recursivity

It is the property that functions have to be called by themselves. It is useful for many tasks, like sorting or calculate the factorial of numbers. For example, to obtain the factorial of a number (n!) the mathematical formula would be:
n! = n * (n-1) * (n-2) * (n-3) ... * 1
more concretely, 5! (factorial of 5) would be:
3! = 3 * 2 * 1 = 6


Factorial(3) = 3 x Factorial(2)
Factorial(2) = 2 x Factorial(1)
Factorial(1) = 1 x Factorial(0)
Factorial(0) = 1
Factorial(1) = 1 x 1 = 1
Factorial(2) = 2 x 1 = 2
Factorial(3) = 3 x 2 = 6

Sample program #05:

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

int fact (int a)
{
  if (a==0)
   return 1;
  else
   return a*fact(a-1);
}


int _tmain(int argc, _TCHAR* argv[])
{
       int n;
  cout << "Please type a number: ";
  cin >> n;
  cout << n << "! = " << fact (n);


       system("pause");
       return 0;
}
```

LAB TASKS

TASK # 01

Compile all sample programs

TASK # 02

Create a program with a function which calculate the square of both the values entered by user. (Using call be reference)

TASK # 03

Write a program with a function volume( ) to calculate the volume of a cube. Use Function Overloading concept **.**Call this function with zero, one, two and three arguments and display the volume returned in the main( ). Use length of side =1 for definition with no arguments.

v=$sxsxs = s^3$,s = length of side

TASK # 04

Perform Task # 03 by using **Default value concept** call the function with 0,1, 2 and 3 Arguments

TASK # 05

Create a program with a function which calculate the power of a number, both number and power should be entered by user at run time. (Note : Use Recursion for this program)