FAST NATIONAL UNIVERSTIY OF COMPUTER AND EMERGING SCIENCES, PESHAWAR

CL217 – OBJECT ORIENTED PROGRAMMING LAB

LAB MANUAL # 05

INSTRUCTOR: MUHAMMAD ABDULLAH

DEPARTMENT OF COMPUTER SCIENCE

SEMESTER SPRING 2021

OBJECT ORIENTED PROGRAMMING LANGUAGE

Table of Contents

Functions:

Functions are like building blocks. They let you divide complicated programs into manageable pieces. They have other advantages, too:

1. While working on one function, you can focus on just that part of the program and construct it, debug it, and perfect it.
2. Different people can work on different functions simultaneously.
3. If a function is needed in more than one place in a program or in different programs, you can write it once and use it many times.
4. Using functions greatly enhances the program's readability because it reduces the complexity of the function main .

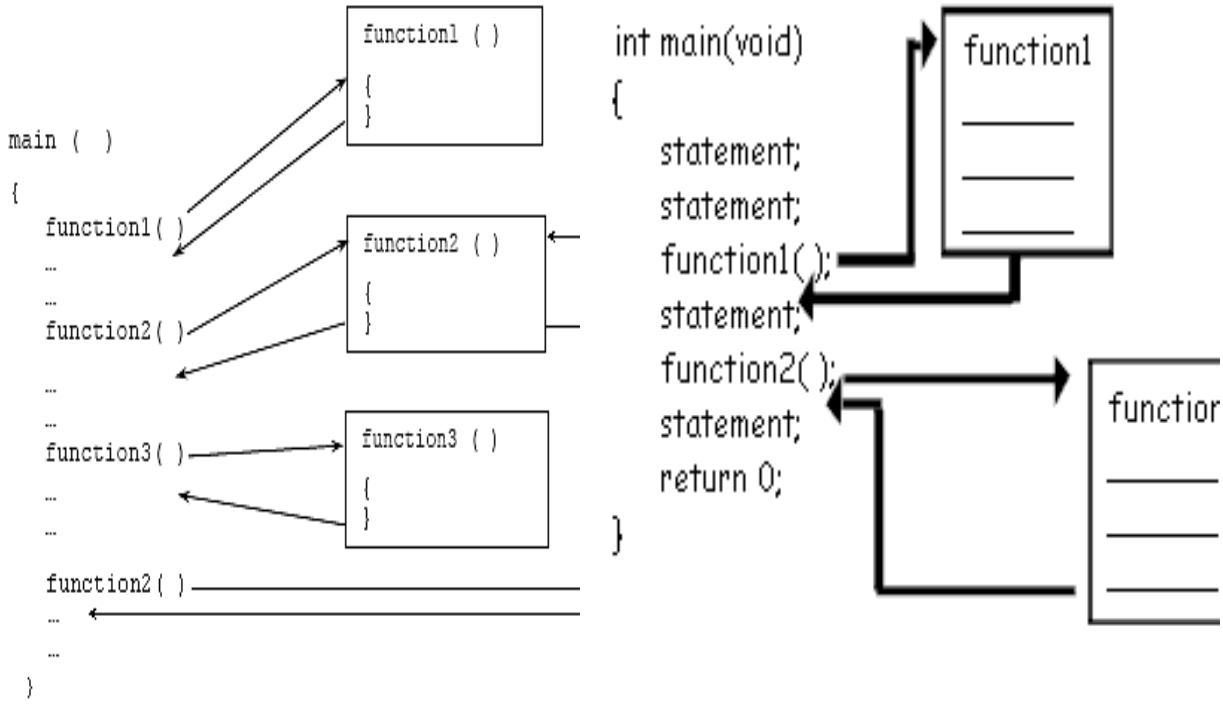

Figure 1: Functions

```
#include <iostream>
#include <cctype>
#include <iomanip>
using namespace std;
double getNumber();
char identifyInput();
double getKilos (double);
double getMeters (double);
int main()
{
```

```
double numberIn = 0.0, kilos = 0.0, meters = 0.0, answer = 0.0 ;
char choice;
numberIn = getNumber();
choice = identifyInput();
if (choice =='P')
{
answer = getKilos (numberIn);
cout << numberIn << " pounds are equivalent to " << answer << "
Kilograms.\n";
}
else
{
answer = getMeters (numberIn);
cout << numberIn << " yards are equivalent to " << answer << "
Meters.\n";
}
system("pause");
return 0;
}
double getNumber()
{
double input;
do
{
cout << "Enter a number greater than 0 that represents a measurement
in either pounds or yards: ";
cin >> input;
}while (input <= 0);
return input;
}
char identifyInput()
{
char option;
do
{
cout << "Enter a P if the number you input was pounds and Y if the
number input was yards: ";
cin >> option;
option = toupper(option);
}while(!(option == 'P' || option == 'Y'));
return option;
}
double getKilos (double lbs)
{
double result;
result = lbs * 0.45359237;
```

```
return result;
}
double getMeters ( double yds)
{
double result;
result = yds * 0.9144;
return result;
}
```

Output

```
/* Sample Run-I:
Enter a number greater than 0 that represents a measurement in either pounds or yards: 512
Enter a P if the number you input was pounds and Y if the number input was yards: P
512.00 pounds are equivalent to 232.24 Kilograms.

Sample Run-II:
Enter a number greater than 0 that represents a measurement in either pounds or yards: 512
Enter a P if the number you input was pounds and Y if the number input was yards: Y
512.00 yards are equivalent to 468.17 Meters.
Press any key to continue . . .
*/
```

Functions Overloading

Creating several functions with the same name but different formal parameters.
Two functions are said to have different formal parameter lists if both functions have:

- A different number of formal parameters or
- If the number of formal parameters is the same, then the data type of the formal parameters, in the order you list them, must differ in at least one position.

If a function's name is overloaded, then all of the functions in the set have the same name. Therefore, all of the functions in the set have different signatures if they have different formal parameter lists. Thus, the following function headings correctly overload the function functionXYZ:

```
void functionXYZ()
void functionXYZ(int x, double y)
void functionXYZ(double one, int y)
void functionXYZ(int x, double y, char ch)
```

Consider the following function headings to overload the function functionABC:

```
void functionABC(int x, double y)
int functionABC(int x, double y)
```

Both of these function headings have the same name and same formal parameter list. Therefore, these function headings to overload the function functionABC are incorrect. In this case, the compiler will generate a syntax error.

```
#include <iostream>
#include <conio.h>
using namespace std;
void repchar();
void repchar(char);
void repchar(char, int);
void main()
{
repchar();
repchar('=');
repchar('+', 30);
system("pause");
}
void repchar()
{
for(int j=0; j<45; j++)
cout << '*';
cout << endl;
}
void repchar(char ch)
{
for(int j=0; j<45; j++)
cout << ch;
cout << endl;
}
void repchar(char ch, int n)
{
for(int j=0; j<n; j++)
cout << ch;
cout << endl;
}
```

Output

```
*********************************************
=============================================
++++++++++++++++++++++++++++++
```

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

void sum(int x, int y)  // formal arguments
    {
    cout<<"sum of int is: "<<(x+y)<<endl;
    }
void sum(double x, double y)  // formal arguments
    {
    cout<<"sum of double is: "<<(x+y)<<endl;
    }
void sum(int x, double y)  // formal arguments
    {
    cout<<"sum of int & double is: "<<(x+y)<<endl;
    }
void sum(double x, int y)  // formal arguments
    {
    cout<<"sum of double & int is: "<<(x+y)<<endl;
    }

int main() {
    sum(3,5);
    sum(3.3,5.6);
    sum(3,5.4);
    sum(3.6,5);

  return 0;
}
```

Output

Output:
sum of int is: 8
sum of double is: 8.9
sum of int & double is: 8.4
sum of double & int is: 8.6

Functions with Default Parameters

- The default parameter is a way to set default values for function parameters a value is not passed in (i.e. it is undefined).
- In C++ programming, we can provide default values for function parameters.

- If a function with default arguments is called without passing arguments, then the default parameters are used.
- However, if arguments are passed while calling the function, the default arguments are ignored.

```
#include <iostream>
#include <conio.h>
using namespace std;
void repchar(char='*', int=45);
int main()
{
repchar();
repchar('=');
repchar('+', 30);
system("pause");
}
void repchar(char ch, int n)
{
for(int j=0; j<n; j++)
cout << ch;
cout << endl;
}
```

Output

```
*********************************************
=============================================
++++++++++++++++++++++++++++++
Press any key to continue . . .
```

How to pass arguments to functions

We can pass arguments to functions by two ways:

1. Passing by values.
2. Passing by references.

1. Pass by Values

Passing arguments in such a way where the function creates copies of the arguments passed to it , it is called passing by value.

When an argument is passed by value to a function, a new variable of the data type of the argument is created and the data is copied into it. The function accesses the value in the newly created variable and the data in the original variable in the calling function is not changed.

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

using namespace std;
void sum(int x, int y)

{

    int sum =x+y;

    cout<<"Result is: "<<sum;

}
int main()

{

   int a=5;

    int b=6;

    sum(a,b);

    return 0;

}
```

Output:

Result is: 11

2. Pass by Reference

The data can also be passed to a function by reference of a variable name and that contains data.

The reference provides the second name (or **alias**) for a variable name.

Alias: Two variables refer to the same thing or entity. Alias are the alternate name for referring to the same thing.

When a variable is passed by reference to a function, no new copy of the variable is created. Only the address of the variable is passed to the function.

The original variable is accessed in the function with reference to its second name or alias. Both variables use the same memory location.

Thus, any change in the reference variable also changes the value in the original variable.

The reference parameters are indicated by an ampersand (&) sign after the data type both in the function prototype and in the function definition.

```
#include <iostream>

using namespace std;


void swapNums(int &x, int &y) {
  int z = x;

  x = y;

  y = z;
  }


int main() {

  int firstNum = 10;

  int secondNum = 20;
  cout << "Before swap: " << "\n";

  cout << firstNum << secondNum << "\n";
  swapNums(firstNum,  secondNum);
  cout << "After swap: " << "\n";

  cout << firstNum << secondNum << "\n";
  return 0;

}
```

Output:

```
Before swap:
10 20

After swap:
20 10
```

The ampersand sign (&) is also used with the data type of **"x"** and **"y"** variables. The **x** and **y** are the aliases of "firstNum" and "secondNum" variables respectively.

The memory location of **"x"** and "firstNum" is the same and similarly, memory location of **"y"** and "secondNum" is same.

How to pass 1D array to Function

```
#include<iostream>

using namespace std;
void arrayIterationFunction(int test[])

{

    for (int i = 0; i <6 ; i++)

    {

        cout<<"myNum["<<i<<"]="<<test[i]<<endl;

    }

}
int main()

{

    int myNum[]= {1,3,4,5,6,7};

    arrayIterationFunction(myNum);
}
```

Output:

myNum[0]=1
myNum[1]=3
myNum[2]=4
myNum[3]=5
myNum[4]=6
myNum[5]=7

```
#include<iostream>

using namespace std;
int array_size;    // initialization at the top

void arrayIterationFunction(int test[])

{

   for (size_t i = 0; i <array_size; i++) //loop for insertion in array

    {

        cout<<"Enter value at test["<<i<<"]=";

        cin>>test[i];

    }

    for (int i = 0; i <array_size; i++) //loop to show value of array

    {

        cout<<"myNum["<<i<<"]="<<test[i]<<endl;

    }

}

int main()

{

    cout<<"Enter Array Size: ";

    cin>>array_size;

    int myNum[array_size];
    arrayIterationFunction(myNum);
}
```

Output:

```
Enter Array Size: 4
Enter value at test[0]=3
Enter value at test[1]=2
Enter value at test[2]=1
Enter value at test[3]=6

myNum[0]=3
myNum[1]=2
myNum[2]=1
myNum[3]=6
```

Find Maximum Value in 1 D Array

```
#include <iostream>

using namespace std;
int main() {


int array_size= 5;

int myNum[array_size]= {11,10,33,4,5};

int max =myNum[0];


for (int i = 0; i <array_size; i++)

{

    if (myNum[i] > max)

    {

        max=myNum[i];

    }

}

cout<<"Maximum number is: "<<max;


  return 0;

}
```

Output:

```
Maximum number is: 33
```

Find Minimum Value in 1 D Array

```
#include <iostream>

using namespace std;
int main() {


int array_size= 5;

int myNum[array_size]= {11,10,33,4,5};

int min =myNum[0];


for (int i = 0; i <array_size; i++)

{

    if (myNum[i] < min)

    {

        min=myNum[i];

    }


}

cout<<"Manimum number is: "<<min;
  return 0;

}
```

Output:

```
Manimum number is: 4
```

Inline Functions

The inline function is special function used for small functions. It is a user-defined function but in it the function prototype is omitted. The keyword "**inline**" is used in the declarator on the function definition. The "**inline**" function is defined before the main function.

When a compiler compiles the source code, it generates a special code at the function call to jump to the function definition. It again generates a code at the end of the function definition to jump back to the calling function. Thus, time is spent is passing the control to the function body and then to the calling function during execution of the program.

The code of the function can be inserted at each of the function call to save the jumping time during program execution. By using the keyword "**inline**" in the declarator of the function definition, the code of the function body is inserted at the place of function call during compilation. Such types of functions are called inline functions.

Note: The inline functions are normally used for small functions.

Inline Functions Example

Write a program to find the exponential power of a given number.

```
#include <iostream>

using namespace std;
inline int expp(int b, int e)

{

    int c;

    int pp=1;

    for(c=1; c<=e; c++)

    {

        pp*=b;  // pp=pp*b;

    }

    return pp;

}


int main()
```

```
{
    int n, p, res;
    cout<<"Enter any number?"<<endl;
    cin>>n;
    cout<<"Enter raise to the power?"<<endl;
    cin>>p;

    res= expp(n, p);
    cout<<"The exponential power of given number is ="<<res;

    return 0;
}
```

Output:

```
Enter any number?
2
Enter raise to the power?
3
The exponential power of given number is =8
```

C++ Templates

In this topic, you'll learn about templates in C++. You'll learn to use the power of templates for generic programming.

Templates are powerful features of C++ which allows you to write generic programs. In simple terms, you can create a single function or a class to work with different data types using templates.

Templates are often used in larger codebase for the purpose of code reusability and flexibility of the programs.

The concept of templates can be used in two different ways:

- Function Templates

- Class Templates

Function Templates

A function template works in a similar to a normal function, with one key difference.

A single function template can work with different data types at once but, a single normal function can only work with one set of data types.

Normally, if you need to perform identical operations on two or more types of data, you use function overloading to create two functions with the required function declaration.

However, a better approach would be to use function templates because you can perform the same task writing less and maintainable code.

In function overloading more than one function with the same name are declared and defined. This makes the program more complex and lengthy. To overcome this problem, function template is used.

Function Template Declaration

In a function template a single typeless function is defined such that arguments of any standard data type can be passed to the function.

The general syntax of a function template is:

```
template <class T>
T function_name(T  argument(s))
{
  Statement(s);
}
```

template The keyword template is used to define the function template.

class In function templates a general data name is defined by using the keyword “class”. In the above syntax, it is represented by word “T”. Any character or word can be used after the keyword “class” to specify the general data name.

function_name It specifies the name of the function.

argument(s) These are the arguments that are to be passed to the function. The arguments are declared of type T. As explained earlier, the type T specifies any standard data type.

statement(s) It specifies the actual statements of the function.

For example, to write a function to calculate a sum of three values either integers or float-points, the function template is written as:

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

Example 1: Function Template to sum of three values

Write a program to calculate and print the sum of three integer and floating-point values. Use the function template.

```
#include <iostream>
//Program to swap data using function templates.
#include <iostream>
using namespace std;

template <class T>
T sum(T a, T b, T c)
{
   return (a+b+c);
}

int main()
```

```
{
    cout<<"Sum of 3 integer values ="<<sum(33,44,55)<<endl;
    cout<<"Sum of 3 float values ="<<sum(33.5,44.55,55.66)<<endl;

    return 0;
}
```

Output:

```
Sum of 3 integer values =132
Sum of 3 float values =133
```

Note: In template functions, the function prototype is generally omitted.

Example 2: Function Template to find the largest number

//Program to display largest among two numbers using function templates.

```
// If two characters are passed to function template, character with larger ASCII
value is displayed.
#include <iostream>
using namespace std;
// template function
template <class T>
T Large(T n1, T n2)
{
        return (n1 > n2) ? n1 : n2;
}

int main()
{
        int i1, i2;
        float f1, f2;
        char c1, c2;
        cout << "Enter two integers:\n";
        cin >> i1 >> i2;
        cout << Large(i1, i2) <<" is larger." << endl;

        cout << "\nEnter two floating-point numbers:\n";
```

```
        cin >> f1 >> f2;
        cout << Large(f1, f2) <<" is larger." << endl;

        cout << "\nEnter two characters:\n";
        cin >> c1 >> c2;
        cout << Large(c1, c2) << " has larger ASCII value.";
        return 0;
}
```

Output

```
Enter two integers:
5
10
10 is larger.

Enter two floating-point numbers:
12.4
10.2
12.4 is larger.

Enter two characters:
z
Z
z has larger ASCII value.
```

- In the above program, a function template Large() is defined that accepts two arguments n1 and n2 of data type T. T signifies that argument can be of any data type.
- Large() function returns the largest among the two arguments using a simple conditional operation.
- Inside the main() function, variables of three different data types: int, float and char are declared. The variables are then passed to the Large() function template as normal functions.
- During run-time, when an integer is passed to the template function, compiler knows it has to generate a Large() function to accept the int arguments and does so.
- Similarly, when floating-point data and char data are passed, it knows the argument data types and generates the Large() function accordingly.

- This way, using only a single function template replaced three identical normal functions and made your code maintainable.

Example 3: Swap Data Using Function Templates

//Program to swap data using function templates.

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

template <typename T>
void Swap(T &n1, T &n2)
{
        T temp;
        temp = n1;
        n1 = n2;
        n2 = temp;
}

int main()
{
        int i1 = 1, i2 = 2;
        float f1 = 1.1, f2 = 2.2;
        char c1 = 'a', c2 = 'b';

        cout << "Before passing data to function template.\n";
        cout << "i1 = " << i1 << "\ni2 = " << i2;
        cout << "\nf1 = " << f1 << "\nf2 = " << f2;
        cout << "\nc1 = " << c1 << "\nc2 = " << c2;

        Swap(i1, i2);
        Swap(f1, f2);
        Swap(c1, c2);

        cout << "\n\nAfter passing data to function template.\n";
        cout << "i1 = " << i1 << "\ni2 = " << i2;
        cout << "\nf1 = " << f1 << "\nf2 = " << f2;
        cout << "\nc1 = " << c1 << "\nc2 = " << c2;

        return 0;
}
```

Output

```
Before passing data to function template.

i1 = 1

i2 = 2

f1 = 1.1

f2 = 2.2

c1 = a

c2 = b

After passing data to function template.

i1 = 2

i2 = 1

f1 = 2.2

f2 = 1.1

c1 = b

c2 = a
```

In this program, instead of calling a function by passing a value, a call by reference is issued.

The Swap() function template takes two arguments and swaps them by reference.