Object-Oriented Programming (OOP)

Week – 09

April 13-17, 2020

Instructor: **Basit Ali**

Email: basit.jasani@nu.edu.pk

Object-Oriented Programming (OOP)

Friend Function & Friend Class

Friend Function

- A friend function of a class is defined outside that class' scope but it has the right to access all private and protected members of the class.

- Even though the prototypes for friend functions appear in the class definition, friends are not member functions.

Friend Function

- To declare a function as a friend of a class, precede the function prototype in the class definition with keyword **friend** as follows

Implementation

```
class A
{
        int x;

        public:
            display()
            {
                cout<<x<<endl;
            }

            friend void set();

            friend void set2(A a);
};
```

```
void set()
{
    A a;
    a.x = 10;
    a.display();
}

void set2(A a)
{
    a.x = 15;
    a.display();
}
```

Friend Class

- like a friend function, a class can also be made a friend of another class using keyword.
- A friend class can access all the private and protected members of other class.
- In order to access the private and protected members of a class into friend class we must pass on object of a class to the member functions of friend class.
- When a class is made a friend class, all the member functions of that class becomes friend functions.

Example!

```
2   #include<iostream>
3   using namespace std;
4
5   class A
6   {
7       private:
8       int x;
9
10      public:
11
12      friend class B;
13
14      set_x(int a)
15      {
16          x = a;
17      }
18  };
19
```

```
20  class B
21  {
22      public:
23
24      display(A a)
25      {
26          cout<<a.x;
27      }
28
29  };
```

Continue..

```
31  int main()
32  {
33      A a;
34      B b;
35      a.set_x(10);
36      b.display(a);
37      return 1;
38  }
```

Generic Programming!

- Generic Programming is the idea to allow type (Integer, String, … etc and user-defined types) to be a parameter to methods, classes and interfaces.

- The method of Generic Programming is implemented to increase the efficiency of the code

Generic Programming!

- Generic Programming enables the programmer to write a general algorithm which will work with all data types.

- It eliminates the need to create different algorithms if the data type is an integer, string or a character.

The advantages of Generic Programming

- Code Reusability
- Avoid Function Overloading
- Once written it can be used for multiple times and cases.

Generics

- Generics can be implemented in C++ using **Templates**.

Templates!

Function Templates

- The general form of a template function definition is:

```
template <class T>
ret-type function-name(parameters)
{
   // body of function
}
```

T is a placeholder that the compiler will automatically replace with an actual data type

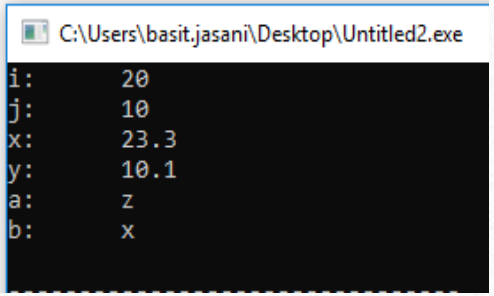
Example

```
6   template <class X>
7   void  SimplePrint (X a)
8   {
9       cout << "Parameter is: " << a <<endl;
10  }
11
12  int main()
13
14  {
15      int i = 20;
16      char c = 'M';
17      float f = 5.5;
18
19      SimplePrint ( i );
20      SimplePrint ( c );
21      SimplePrint ( f );
22  }
```

Example

```
6   template <class T>
7   void swapargs(T &a, T &b)
8   {
9       T temp;
10      temp = a;
11      a = b;
12      b = temp;
13  }
```

```
16  int main()
17  {
18      int i=10;
19      int j=20;
20      double x=10.1;
21      double y=23.3;
22      char a='x';
23      char b='z';
24
25      swapargs(i, j); // swap integers
26      swapargs(x, y); // swap floats
27      swapargs(a, b); // swap chars
28
29      cout<<"i:   "<<i<<endl;
30      cout<<"j:   "<<j<<endl;
31      cout<<"x:   "<<x<<endl;
32      cout<<"y:   "<<y<<endl;
33      cout<<"a:   "<<a<<endl;
34      cout<<"b:   "<<b<<endl;
35  }
```



Template Function with Two Generic Types

- You can define more than one generic data type in the template statement by using a comma-separated list

```
template <class T1, class T2>
   void myfunc(T1 a, T2 b)
   {
     cout << a << “  &  ” << b << '\n';
   }
```

Specialized Template

```
4   template <class T>
5   void fun(T a)
6   {
7       cout << "The main template fun(): "
8            << a << endl;
9   }
10
11  template<>
12  void fun(int a)
13  {
14      cout << "Specialized Template for int type: "
15           << a << endl;
16  }
```

```
18  int main()
19  {
20      fun<char>('a');
21      fun<int>(10);
22      fun<float>(10.14);
23  }
```

```
C:\Users\basit.jasani\Desktop\Untitled2.exe
The main template fun(): a
Specialized Template for int type: 10
The main template fun(): 10.14

--------------------------------
Process exited after 0.1619 seconds with
```

Overloading a Generic Function

- In addition to creating explicit, overloaded versions of a generic function, you can also overload the template specification itself

- To do so, simply create another version of the template that differs from any others in its parameter list

Example

```
// First version of f() template
   template
template <class X>
void f(X a)
{
   cout << "Inside f(X a)";
}
```

```
// Second version of f()

template <class X, class Y>
void f(X a, Y b)
{
cout << "Inside f(X a, Y b)";
}
```

Using Normal Parameters in Generic Functions

- You can mix *non-generic parameters* with *generic parameters* in a template function:

```
template<class X> void func(X a, int b){
    cout << "General Data:  " << a;
    cout << "Integer Data:  "  << b;
}
```

Generic Classes

- In addition to generic functions, you can also define a *generic class*
- The actual type of the data being used (in class) will be specified as a parameter when objects of that class are created
- Generic classes are useful when a class uses logic that can be generalized e.g. Stacks, Queues

Generic Classes

- The general form of a generic class declaration is shown here:

template <class T> class class-name

{

. . .

}

Generic Classes

- If necessary, we can define more than one generic data type using a comma-separated list
- We create a specific instance of that class using the following general form:

class-name <type> ob;

Example

```
template <class T1, class T2> class myclass {
    T1 i;
    T2 j;
    public:
    myclass (T1 a, T2 b) { i = a; j = b; }
    void show( ) { cout << i << " & " << j; }
};
```

Example (cont.)

```
int main(){
    myclass<int, double> ob1(10, 0.23);
    myclass<char, char *> ob2('X', “Hello”);

    ob1.show(); // show int, double
    ob2.show(); // show char, char *
}
```

Using Non-Type Arguments with Generic Classes

- In a generic class, we can also specify non-type arguments:

```
template <class T, int size> class MyClass
{
   T arr[size]; // length of array is passed in size
   // rest of the code in class
}
```

Example (cont.)

```
int main()
{
    atype<int, 10> intob;
    atype<double, 15> doubleob;
}
```