NATIONAL UNIVERSITY OF COMPUTER AND EMERGING SCIENCES

OBJECT ORIENTED PROGRAMMING - LAB

Instructors: Muhammad Sudais

Email: muhammad.sudais.v@nu.edu.pk

LAB 08

Outline

- Operator Overloading
- Friend Functions
- Friend Classes

Operator Overloading

In C++, we can change the way operators work for user-defined types like objects and structures. This is known as operator overloading.

Suppose we have created three objects c1, c2 and result from a class named Complex that represents complex numbers.

Since operator overloading allows us to change how operators work, we can redefine how the + operator works and use it to add the complex numbers of c1 and c2 by writing the following code:

result = c1 + c2;

instead of something like

result = c1.addNumbers(c2);

This makes our code intuitive and easy to understand.

Syntax for C++ Operator Overloading

To overload an operator, we use a special operator function. We define the function inside the class or structure whose objects/variables we want the overloaded operator to work with.

```
class className {
   ... .. ...
   public
     returnType operator symbol (arguments) {
        ... .. ...
     }
   ... .. ...
};
```

Here,

- **returnType** is the return type of the function.
- **operator** is a keyword.
- **symbol** is the operator we want to overload. Like: +, <, -, ++, etc.
- **arguments** is the arguments passed to the function.

Unary Operator Overloading

```
// Overload ++ when used as prefix

#include <iostream>
using namespace std;

class Count {
  private:
   int value;

  public:

   // Constructor to initialize count to 5
   Count() : value(5) {}

   // Overload ++ when used as prefix
   void operator ++ () {
      ++value;
   }
   // Overload ++ when used as postfix
      void operator ++ (int) {
          value++;
      }

   void display() {
      cout << "Count: " << value << endl;
   }
};

int main() {
   Count count1;

   // Call the "void operator ++ ()" function
   ++count1;
count1++;
   count1.display();
   return 0;
}
```

This code doesn't work if we try to do something like this:

```
Count count1, result;
// Error
result = ++count1;
```

This is because the return type of our operator function is void. We can solve this problem by making Count as the return type of the operator function.

```
// return Count when ++ used as prefix
Count operator ++ () {
   // code
}
// return Count when ++ used as postfix
Count operator ++ (int) {
  // code
}
```

Binary Operator Overloading

Binary operators work on two operands. For example,

```
result = num + 9;
```

Here, + is a binary operator that works on the operands num and 9.

When we overload the binary operator for user-defined types by using the code:

```
obj3 = obj1 + obj2;
```

The operator function is called using the obj1 object and obj2 is passed as an argument to the function.

C++ Binary Operator Overloading

```
// C++ program to overload the binary operator +
// This program adds two complex numbers

#include <iostream>
using namespace std;

class Complex {
   private:
    float real;
    float imag;

   public:
    // Constructor to initialize real and imag to 0
    Complex() : real(0), imag(0) {}

    void input() {
        cout << "Enter real and imaginary parts respectively: ";
        cin >> real;
        cin >> imag;
    }

    // Overload the + operator
    Complex operator + (const Complex& obj) {
        Complex temp;
        temp.real = real + obj.real;
        temp.imag = imag + obj.imag;
        return temp;
    }
```

FRIEND FUNCTION

Non-member functions of a class will not have access to the private data of another class. There could be situations where we want two classes to share some functions and the data members. In that case, we can make the function a friend of these classes, and that will enable the function to access the private and protected data members of the classes. In this article, we will look at how to make a non-member function friend to a class.

A friend function is a function that is specified outside a class but has the ability to access the class members' protected and private data. A friend can be a member's function, function template, or function, or a class or class template, in which case the entire class and all of its members are friends.

Why we need it?

In special cases when a class's private data needs to be accessed directly without using objects of that class, we need friend functions. For instance, let's consider two classes: Director and Doctor. We may want the function gross_salary to operate the objects of both these classes. The function does not need to be a member of either of the classes.

They are also used in operator overloading because they are more intuitive. The binary arithmetic operator that is commonly used can be overloaded the friend function way. Go ahead and check out operator overloading using a friend function for more information.

Special features of friend functions:

- A friend function does not fall within the scope of the class for which it was declared as a friend. Hence, functionality is not limited to one class.
- The friend function can be a member of another class or a function that is outside the scope of the class.
- A friend function can be declared in the private or public part of a class without changing its meaning.
- Friend functions are not called using objects of the class because they are not within the class's scope.
- Without the help of any object, the friend function can be invoked like a normal member function.
- Friend functions can use objects of the class as arguments.
- A friend function cannot explicitly access member names directly. Every member name has to use the object's name and dot operator. . For example, Doctor.pay where pay is the object name.

Syntax

To make a function that is declared outside the class "friendly" to that class, we have to declare the function as a friend function, as seen below:

```
class className{
  // Other Declarations
  friend returnType functionName(arg list);
};
```

As we can see above, the friend function should be declared inside the class whose private and protected members are to be accessed.

Let's breakdown the syntax:

- `friend` is a keyword to denote that this function is a friend function.
- `returnType` is the function's return type.
- `functionName` is the name of the function being made a friend of the class.
- `arg list` is the arguments that are passed.

The friend function definition is found outside the class like a normal member function. The friend function is not defined using the friend keyword or use the scope resolution operator:: as it is not a member of the class in which it has been declared. A friend function can be declared in several classes.

Friend function declaration

```
// Program to illustrate friend function
#include<iostream>
using namespace std;
class integer
{
  int a, b;
  public:
    void set_value()
    {
    a=50;
    b=30;
    }
  friend int mean(integer s);  //declaration of friend function
};

int mean(integer s)
{
  return int(s.a+s.b)/2.0; //friend function definition
}
int main()
{
  integer c;
  c.set_value();
  cout<< "Mean value:" <<mean(c);
  return 0;
}
```

The output of our program will be: Mean value: 40

Friend Class

A friend class can have access to the data members and functions of another class in which it is declared as a friend. They are used in situations where we want a certain class to have access to another class's private and protected members.

Classes declared as friends to any another class will have all the member functions become friend functions to the friend class. Friend functions are used to work as a link between the classes.

Syntax of friend class:

```
class S; //forward declaration

class P{
  // Other Declarations
  friend class S;
};

class S{
  // Declarations
};
```

In the illustration above, class S is a friend of class P. As a result class S can access the private data members of class P. However, this does not mean that class P can access private data members of class S. A forward declaration informs the compiler about an entity's existence before the entity is explicitly defined.

We have declared class S using forward declaration to inform the compiler of its existence, allowing us to use the objects of class S in class P.

Note: Class friendship is neither inherited nor mutual unless we make it so. This means that because class S is a friend of class P, it will be a friend of the sub-classes of class P.

Example of a program to illustrate friend class:

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

// forward declaration
class ClassY;

class ClassX {
  int digit1;

  // friend class declaration
  friend class ClassY;

  public:
      // constructor to initialize num1 to 10
      ClassX() : digit1(10) {}
};
```

```
class ClassY {
    int digit2;

    public:
        // constructor to initialize num2 to 5
        ClassY() : digit2(5) {}

    // member function to multiply num1
    // from ClassX with num2 from ClassY
    int multiply() {
        ClassX m;
        return m.digit1 * digit2;
    }
};

int main() {
    ClassY n;
    cout << "Multiplication: " << n.multiply();
    return 0;
}
```

The output will be: `Multiplication: 50`

In the program above, we have declared two classes: X and Y. `ClassY` is a friend class of `ClassX`. Therefore, `ClassY` has access to the member function of `ClassX`. In `ClassY`, we have created a function `multiply()` that returns the multiplication of `digit1` and `digit2`.

`ClassY` being a friend class enables us to create objects of `ClassX` inside of `ClassY`. This is possible through forward declaration of the `ClassY`.

Here are some important points on friend functions and classes:

- Friend functions should be used for restricted purposes only. Having excessive friend functions and classes can reduce the object-oriented programming feature of encapsulation in a program.
- Friendship is not reciprocal. If class X is Y's friend, then Y does not automatically become X's friend.
- Friendship cannot be inherited.

EXERCISE

Question 1

Create a class Home which has the mentioned attributes. Create the operator overloading functions for **+**, **-**, **++** and **/**. The final outcome of the operator overloading will be the changes in attributes values.

For instance, if we are using add operator it should add number of rooms, area and other relevant values.

ATTRIBUTES	ADD +	SUBTRACT -	INCREMENT ++	DECREMENT --
NOOFROOMS	H1+H2	H1-H2	H1++	H1--
HOMEID	Assign new Id	Assign new Id	Assign new Id	Assign new Id
AREA	H1+H2	H1-H2	H1++	H1--
ADDRESS	Give new address	Give new address	Give new address	Give new address

You have to return an object of same class in every operator overloading functions.

```
Home H1;
Home H2;
Home H3 = H1+H2;
```

Sample Input Homes for Add:
Home H1 = 2 Rooms, 2821, 240 yds, ABC Street 123 Road.
Home H2 = 3 Rooms, 2213, 120 yds, ABC Street 000 Road.
Sample Output Home for Add:
Home H3 = 5 Room, 2346, 360 yds, XYZ Street 122 Road.

Sample Input Homes for Increment:
Home H1 = 2 Rooms, 2821, 240 yds, ABC Street 123 Road.
Sample Output Home for Add:
Home H3 = 3 Room, 2346, 241 yds, XYZ Street 122 Road.

Question 2

Make a friend function of already created Class Home named “MyFriend()” which displays a newly added private data member named “FamilyMembers” of Class Home.

Question 3

Complete the below implementation to experiment friend classes.

```
#include <iostream>s
using namespace std;
class Square
{
    // declaring Rectangle as friend class
    int side;
    public:
        Square ( int s )
        {
            side = s;
        }
};

class Rectangle
{
    // declare variables
    public:
    int getArea()
    {
        return length * breadth;
    }
    void shape( ?? )
    {
        // Length and Breadth are the size of Square a's side.
    }
};

int main()
{
    // Square Object Create
    // Rectangle Object Create
    rectangle.shape(square);
    cout << rectangle.getArea() << endl;
    return 0;
}
```

Question 4

Implement the mentioned concept of Friend Classes in the scope of your Course Project.

Question 5

Write a class Time which represents time. the class should have three fields for hours, minutes and seconds. It should have constructor to initialize the hours, minutes and seconds. A function print Time() to print the current time. Overload the following operators:

- Plus operator (+) to add two time objects based on 24-hour clock.
- Operator< to compare two time objects.