Lab No.7 Friend Function and Friend Class

7.1 Introduction

Encapsulation and data hiding is a key component in object oriented programming. It ensures that class members are not accessible openly. An access to class members is provided through a regulated mechanism in which all accesses are accountable and legal.

Friend function and friend class provide a way through which access specifiers can be relaxed and provide direct access. This is only available in C++ and not supported in Java and Python. This lab covers friend function and friend class in detail.

7.2 Objectives of the lab:

1. Understand the difference between a regular function and a friend function
2. Explain the concept of friend function.
3. Develop a friend function.
4. Explain the concept of friend class.
5. Develop a friend class.

7.3 Pre-Lab

7.3.1 Friend Functions

1. A regular C-style function accessing the non-public members of the objects of a class to which it is a friend
   - Not a member function
   - Can access non-public members of the objects to which it is a friend
2. Declaration inside the class preceded by keyword **friend**
   - Keyword friend is not used with definition. //Compiler error
3. Definition outside the class but like the definition of normal C-style functions
4. **this** is NOT Available to friend Functions
5. A friend function is not called in the manner member functions are called. Friend function is called like normal C-style functions and takes the object of the class to which it is a friend
6. A friend can be declared in **public**, **private**, or **protected** portions.
   - Member access specifiers have no effect on friend function
   - Friend function is not a member
7. Friend functions are not inherited

7.3.2 Example: FriendFtn.cpp

```
#include <iostream>
```

```
using namespace std;

class test {
private:
    int       data;
public:
    test(): data(0){}

    void show(){
            cout<<"data:  "<<data<<endl;
    }

    friend void set_data(test &);
};

void set_data(test &obj){
    obj.data=3;
}

int main(){
    test      t;
    set_data(t);       //called like normal function
    t.show();
    return 0;
}
```

Output:

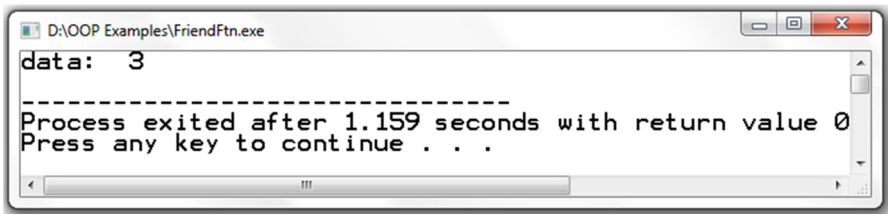

Figure 7.1: Output FriendFtn.cpp

7.3.3 Friend Classes

1 Not directly associated with the Class

2 All the member functions of the friend class can access non-public data members of the original class

- If class A is the friend of class B then all its member functions can access the nonpublic members of class B

3 Declaration

```
class B{
friend class A;
…
```

```
    }
```

4 Can be friend of more than one classes
5 Violation of encapsulation
6 SHOULD BE USED ONLY WHEN REQUIRED

7.3.4 Example: FriendCls.cpp

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

class A{
private:
    int       x;
    friend class B;
public:
    A(): x(0){}

    void showA(){
            cout<<"x: "<<x<<endl;
    }
};

class B{
private:
    int       y;
public:
    B(): y(0){}

    void showB(A &a){
            a.x=4;
            cout<<"x: "<<a.x<<endl;
            cout<<"y: "<<y<<endl;
    }
};

int main(){
    A         obja;
    cout<<"Original Data of A:"<<endl;
    obja.showA();

    B         objb;
    cout<<"\nAfter calling showB()..."<<endl;
    objb.showB(obja);

    cout<<"\nModified Data of A:"<<endl;
```

```
    obja.showA();

    return 0;
}
```

Output:

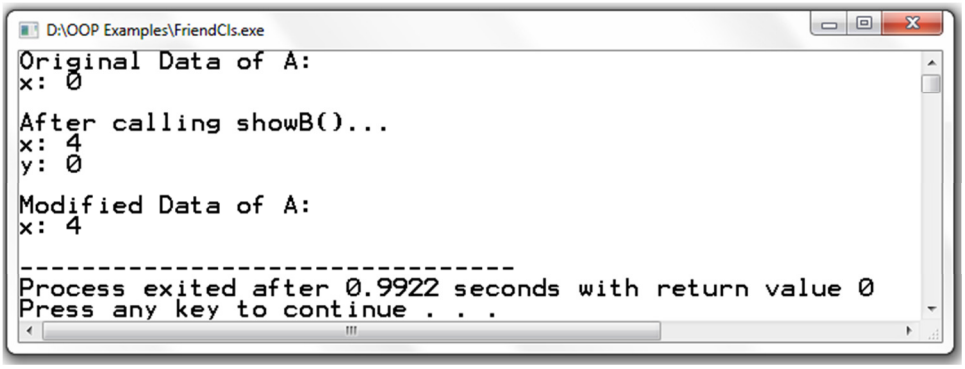

Figure 7.1: Output FriendCls.cpp

Q. What would happen if we write the statement friend class B; in public or protected portions? Would it have any effect on behavior of the class?

7.4 Activities

Perform these activities in C++ only.

7.4.1 Activity

Create a class **RationalNumber** that stores a fraction in its original form (i.e. without finding the equivalent floating pointing result). This class models a fraction by using two data members: an integer for numerator and an integer for denominator. For this class, provide the following functions:

a) A **no-argument constructor** that initializes the numerator and denominator of a fraction to some fixed values.

b) A **two-argument constructor** that initializes the numerator and denominator to the values sent from calling function. This constructor should prevent a 0 denominator in a fraction, reduce or simplify fractions that are not in reduced form, and avoid negative denominators.

c) A **showRN()** function to display a fraction in the format a/b.

d) Provide the following operator functions as **non-member friend functions.**

   i. An overloaded **operator +** for addition of two rational numbers.

      Two fractions a/b and c/d are added together as:

$$\frac{a}{b}+\frac{c}{d}=\frac{(a*d)+(b*c)}{b*d}$$

ii. An overloaded **operator -** for subtraction of two rational numbers.

Two fractions a/b and c/d are subtracted from each other as:

$$\frac{a}{b}-\frac{c}{d}=\frac{(a*d)-(b*c)}{b*d}$$

iii. An overloaded **operator *** for subtraction of two rational numbers.

Two fractions a/b and c/d are multiplied together as:

$$\frac{a}{b}*\frac{c}{d}=\frac{a*c}{b*d}$$

iv. An overloaded **operator /** for division of two rational numbers.

If fraction a/b is divided by the fraction c/d, the result is

$$\frac{a}{b}\Big/\frac{c}{d}=\frac{a*d}{b*c}$$

v. Overloaded relational operators

a. **operator >**: should return a variable of type **bool** to indicate whether 1st fraction is greater than 2nd or not.

b. **operator <**: should return a variable of type **bool** to indicate whether 1st fraction is smaller than 2nd or not.

c. **operator >=**: should return a variable of type **bool** to indicate whether 1st fraction is greater than or equal to 2nd or not.

d. **operator <=**: should return a variable of type **bool** to indicate whether 1st fraction is smaller than or equal to 2nd or not.

vi. Overloaded equality operators for **RationalNumber** class

a. **operator==**: should return a variable of type **bool** to indicate whether 1st fraction is equal to the 2nd fraction or not.

b. **Operator!=:** should a **true** value if both the fractions are not equal and return a **false** if both are equal.

7.4.2 Activity

Create a class called **Time** that has separate int member data for hours, minutes, and seconds. Provide the following member functions for this class:

a) A **no-argument constructor** to initialize hour, minutes, and seconds to 0.

b) A **3-argument constructor** to initialize the members to values sent from the calling function at the time of creation of an object. Make sure that valid values are provided for all the data members. In case of an invalid value, set the variable to 0.

c) A member function **show** to display time in 11:59:59 format.

d) Provide the following functions as **friends**

   a. An overloaded **operator+** for addition of two Time objects. Each time unit of one object must add into the corresponding time unit of the other object. Keep in view the fact that minutes and seconds of resultant should not exceed the maximum limit (60). If any of them do exceed, subtract 60 from the corresponding unit and add a 1 to the next higher unit.

   b. Overloaded operators for **pre- and post- increment**. These increment operators should add a 1 to the **seconds** unit of time. Keep track that **seconds** should not exceed 60.

   c. Overload operators for **pre- and post- decrement**. These decrement operators should subtract a 1 from **seconds** unit of time. If number of seconds goes below 0, take appropriate actions to make this value valid.

A **main()** programs should create two initialized **Time** objects and one that isn't initialized. Then it should add the two initialized values together, leaving the result in the third **Time** variable. Finally it should display the value of this third variable. Check the functionalities of ++ and -- operators of this program for both pre- and post-incrementation.

7.5 Testing

Test Cases for Activity 7.4.1

Sample Inputs	Sample Outputs
Declare two RationalNumber Objects r1 and r2. Take both object values from user. Display the content of r1 and r2 using showRN() Add r1 and r2 using + and store in r3. Display the content of r3 using showRN() Subtract r1 and r2 using - and store in r3. Display the content of r3 using showRN()	Enter First Rational Number: 1 1 Enter Second Rational Number: 1 3 Input Rational Numbers: R1 = 1 R2 = 1/3

Multiply r1 and r2 using * and store in r3. Display the content of r3 using showRN() Divide r1 and r2 using / and store in r3. Display the content of r3 using showRN() Check rational number r1 is greater than r2 using > and display the respective message. Check rational number r1 is less than r2 using < and display the respective message. Check rational number r1 is greater than or equal to r2 using >= and display the respective message. Check rational number r1 is less than or equal to r2 using <= and display the respective message. Check rational number r1 is equal to r2 using == and display the respective message. Check rational number r1 is not equal to r2 using != and display the respective message.	

Test Cases for Activity 7.4.2

Sample Inputs	Sample Outputs
Declare two Time Objects t1 and t2 and initialize using three-argument constructor. Display the content of t1 and t2 using showTime() Add t1 and t2 using + and store in t3. Display the content of d3 using showTime() Pre-increment t1 and display the content using showTime(). Post- increment t1 and display the content using showTime(). Pre-decrement t2 and display the content using showTime(). Post- decrement t2 and display the content using showTime().	 First time input: 4:30:50 Second time input: 9:45:30 Sum of two time inputs: 2:16:20 Pre-incrementing first time input: 4:30:51 Post-incrementing first time input: 4:30:52 Pre-decrementing second time input: 9:45:29 Post-decrementing second time input: 9:45:28

7.6 References:

1. **Class notes**
2. **Object-Oriented Programming in C++ by *Robert Lafore***
3. **How to Program C++ by *Deitel & Deitel***
4. **Programming and Problem Solving with Java by *Nell Dale & Chip Weems***
5. **Murach's Python Programming by *Micheal Urban & Joel Murach***