LABORATORY 9

Now that's classy

Objective

This week in lab you will work with an existing class type. The objective of the laboratory is to further explain how a class is constructed, to demonstrate how to overload an operator, and to show how a properly constructed class can be extended in useful ways without affecting any client programs that already use the class.

Key Concepts

- Abstract data type (ADT)
- Facilitators
- Constructors
- **`const`** member functions
- Inspectors
- Auxiliary functions and operators
- Mutators
- Operator overloading

9.1 GETTING STARTED

- Using the procedures in the introductory laboratory handout, create the working directory `\cpplab` on the appropriate disk drive and obtain a copy of self-extracting archive `lab09.exe`. The copy should be placed in the `cpplab` directory. Execute the copy to extract the files necessary for this laboratory.
- Many of the activities that are performed in the laboratory can be done in groups but you should work the exercises yourself.

9.2 ABSTRACT DATA TYPES

Sophisticated problem solving requires that we develop our own representations for the information to be manipulated. In object-oriented programming terminology, the representation and the operations to be performed on the representation form a *data abstraction*.

Data abstractions are developed using classes, functions, and operators. In developing an abstraction, we normally follow the information-hiding principle. Enforcing information hiding through encapsulation helps to maintain the integrity of the data (e.g., preventing an errant client application from setting the denominator of rational number to zero). In addition, because client programs use public methods, they are generally immune to changes in the implementation of the abstraction.

A well-defined abstraction allows its objects to be created and used in an intuitive manner. Therefore, the programming syntax for the definition and manipulation of objects of an abstraction should have a form analogous to fundamental-type and standard-class objects doing comparable activities.

A well-defined class using the information-hiding principle coupled with the appropriate library functions is an *abstract data type* or ADT.

The class `Rational` is an example of an abstract data type. In the following code segment, we display the result of summing 1/2 and 1/3.

```
Rational a(1,2);          // a = 1/2
Rational b(2,3);          // b = 2/3
cout << a << " + " << b << " = " << a + b << endl;
```

`Rational` addition and insertion have the same form as the corresponding display of the sum of two **`int`** or **`float`** objects would have. This analogous form would not be the case in traditional languages such as C or Pascal. In traditional languages, a programmer can neither have objects with methods nor extend existing operators to work with new types of objects. The programmer is forced to define functions and additional temporary objects. The resulting code is generally unnatural and awkward.

Your first task in the lab is to define two member functions `Subtract()` and `Divide()` to support `Rational` subtraction and division. The members are used by `Rational` auxiliary operators: minus (`-`) and slash (`/`). The two members `Subtract()` and `Divide()` are necessary because the auxiliary operators are not members and therefore have no access to the data member values.

- Open the project file `rational.dsw`. Open the file `ratextra.cpp`. Scan through `ratextra.cpp` to see the definitions of auxiliary operators `-` and `/`. See that they invoke `Rational` member functions `Subtract()` and `Divide()`.
- Open the file `rational.h`. Make sure that `Subtract()` and `Divide()` are listed in the class division as public member functions. Make sure that

- and / have been prototyped as auxiliary operators. Notice that they are prototyped outside the class definition.

- The function body of the `Rational` member function `Subtract()` is incomplete. Remember the operation `a/b - c/d` equals `(ad - bc)/bd`. Complete this function.
- The `Rational` member function `Divide()` needs to be completely written. Add this new function to `ratextra.cpp`. Remember the operation `(a/b) / (c/d)` equals `ad / bc`.
- Open the file `ratmain.cpp` from the project `rational.dsw`. Observe what function `main()` does.
- ✎ Make and run project `rational.dsw`. Use the following inputs to test the program. ✓

```
4/5 1/5
3/2 7/9
5/5 2/2
0/5 -1/5
```

- Observe that 4/5 - 1/5 did not produce 3/5. Instead, it produced 15/25, which is equivalent to 3/5.
- Close and save your modified `rational.dsw` project.

9.3 REDUCING RATIONAL NUMBERS

In a *reduced* rational the numerator and denominator do not have a common divisor other than the factor 1. For example, the rational number 9/10 is reduced, while the rational number 8/10 is not. 8/10 can be reduced to 4/5. Producing rational numbers that are not reduced is unsatisfactory. With nonreduced numbers, determining whether two rationals are equal is difficult. In addition, the numerators and denominators of rationals that are being computed can get larger than they need to be, which can cause unnecessary overflow problems.

You are to modify the `Rational` class so that all arithmetic operations return a reduced result. A simple way to reduce a fraction is to divide both the numerator and denominator by their greatest common divisor (GCD). A simple algorithm for computing the GCD of two positive integers m and n follows.

1. Let r be the remainder of m divided by n.
2. If r is zero, the algorithm terminates and n is the GCD. If r is not zero, then set $m = n$ and $n = r$; and return to step 1.

A C++ function that implements this algorithm is contained in `gcd.cpp`, and the interface is in `gcd.h`. Examine the function and make sure it corresponds to the algorithm. Before using a function, you should make sure that the function works as advertised. One way to do so is to supply a test harness and test the function. Before making the modifications to the rational class, write a test harness and test gcd.cpp. Your test program should prompt for two integers and print the greatest common divisor.

- Open the file gcdtest.cpp and put your the test code inside function main().
- Make and run the project gcd.dsw. This project uses gcdtest.cpp.
- Run your program on the following inputs to make sure function gcd() is working correctly.

```
12 6
55 44
15 0
461952 116298
```

- ✎ Explain the results to your laboratory instructor. ✓
- Open the reduce.dsw project.
- Make the changes to rational.h and rational.cpp so that reduced rationals are used. You will need to develop a new protected member Rational function Reduce(), whose class prototype is

```
void Reduce();
```

Recall that the implementation of Reduce() should go in the file rational.cpp, while the interface to Reduce() is part of the Rational class definition in rational.h.

- Add the prototype for Reduce() to the class Rational in rational.h.
- Include the library gcd.h at the beginning of rational.cpp.
- Add the implementation of Reduce() to the file rational.cpp. Note that in the implementation file when defining Reduce() you must begin the definition in the following manner:

```
void Rational::Reduce()
```

  Because the implementation of Reduce() is not contained with its class definition, you need to prepend the function name Reduce with its class name Rational through the use of the scope resolution operator. This syntax tells the compiler which function Reduce() we are defining—in this case the one that is a member function of the class Rational.

- ✎ Show your laboratory instructor the change you made to rational.h and your implementation of Reduce() in rational.cpp. Remember that gcd() expects that both of its parameters are positive. Therefore, some case analysis is required in Reduce(). ✓

The next step is to decide where and when to call Reduce(). Our implementation of the Rational class requires only two calls of Reduce() in the Rational member functions. One of these calls is in the nondefault constructor.

- Examine the Rational implementation and determine where to place the other call to Reduce(). Tell your laboratory instructor where you want to place this call. Hint: the call is not added to `SetNumerator()` or `SetDenominator()`. ✓

- After making all the necessary changes, demonstrate that your implementation of the `Rational` class produces reduced results. Build and run the project `reduce.dsw` on appropriate inputs. The test harness for this project is `rattest.cpp`. ✓
- Close the project `reduce.dsw`.

9.4 OVERLOADING OPERATORS

For the next part of the laboratory, you are to develop relational auxiliary operators for the class `Rational`.

Rationals *a/b* and *c/d* are equal if *ad* equals *bc*. Rational *a/b* is less than rational *c/d* if *ad* is less than *bc*. Rational *a/b* is greater than rational *c/d* if *ad* is greater than *bc*.

The overloaded relational auxiliary operators will use a public member facilitator `Compare()` that expects a single constant reference parameter `r` as its parameter. Facilitator `Compare()` returns a negative value if the invoking object is less than `r`, it returns 0 if the invoking object is equal to `r`, and it returns a positive value if the invoking object is greater than `r`.

You can use member function `Compare()` to implement the relational operators ==, < and >. For example `Rational` object `s` is less than `Rational` object `t` if `s.Compare(t)` is negative.

- Open the project `comp.dsw`.
- Make the changes necessary to both `rational.h` and `rational.cpp` to make `Compare()` a member function of the Rational ADT. Remember that the interface (i.e., the prototype) goes in `rational.h` and the implementation of the overloading goes in `rational.cpp`.
- Make the changes necessary to both `rational.h` and `rational.cpp` to make ==, < and > auxiliary relational operators of the Rational ADT. Remember that the interface (i.e., the prototype) goes in `rational.h` and the implementation of the overloading goes in `rational.cpp`.
- Test your auxiliary operators by making and running the project `comp.dsw`. Demonstrate your program to a laboratory instructor. ✓

9.5 FINISHING UP

- Copy any files you wish to keep to your own drive.
- Delete the directory `\cpplab`.
- Hand in your check-off sheet.