

## CS 211, Winter 2004

### Lab Assignment L3: Basic Classes and Operator Overloading Assigned: Monday, Feb. 9, Due: Monday, Feb. 23, 11:00 AM

Brian M. Dennis, Instructor  
Bin Lin, Tom Lechner, Rachel Goldsborough, Teaching Assistants

#### Aims and Goals

The purpose of this assignment is to make you proficient with C++ classes and operator overloading. After you complete this lab, you should be able to do the following:

- Declare and define classes
- Define instances of classes
- Access member variables of a class instance
- Invoke member functions of a class instance
- Declare and define standalone functions for overloading operators.
- Declare and define member functions for overloading operators.

For this assignment, the startup code will be very slight. Mainly we will provide driver programs to demonstrate some of what you should test against. You will have to write the requested classes from scratch.

#### A Complex Number Datatype (10 Points)

Create a class called `Complex` for performing arithmetic with complex numbers.

Complex numbers have the form: `realPart + imaginaryPart * i` where `i` is  $\sqrt{-1}$

Use `double` variables to represent the `private` data of the class. Provide a constructor that enables an object of this class to be initialized when it is declared. The constructor should contain default values in case no initializers are provided. Provide `public` member functions for each of the following:

- Adding two `Complex` numbers: The real parts are added together and the imaginary parts are added together.

- Subtracting two `Complex` numbers: The real part of the right operand is subtracted from the real part of the left operand., and the imaginary part of the right operand is subtracted from the imaginary part of the left operand.
- Printing complex numbers in the form  $(a, b)$ , where  $a$  is the real part and  $b$  is the imaginary part.

In addition, add these extensions to your class:

- Modify the class to enable input and output of complex numbers overloaded `>>` and `<<` operators, respectively. Your input operator should read in the `Complex` output format  $(a, b)$ .
- Overload the multiplication operator to enable multiplication of two complex numbers as in algebra.
- Overload the `==` and `!=` operators to allow comparisons of complex numbers.

## A Rational Number Datatype (20 Points)

Create a class called `Rational` for performing arithmetic with fractions.

Use integer variables to represent the `private` data of the class – the `numerator` and the `denominator`. Provide a constructor that enables an object of this class to be initialized when it is declared. The constructor should contain default values in case no initializers are provided and should store the fraction in reduced form. For example, the fraction

$$\frac{2}{4}$$

would be stored in the object is 1 in `numerator` and 2 in the `denominator`. Provide `public` member functions that perform each of the following tasks:

- Adding two `Rational` numbers. The result should be stored in reduced form.
- Subtracting two `Rational` numbers. The result should be stored in reduced form.
- Multiplying two `Rational` numbers. The result should be stored in reduced form.
- Dividing two `Rational` numbers. The result should be stored in reduced form. Detect and avoid division by zero.
- Printing `Rational` numbers in the form  $a/b$ , where  $a$  is the numerator  $b$  is the denominator.
- Printing `Rational` numbers in floating-point format

Each of the arithmetic operations should return a new `Rational` that is the result of the computation.

*Hint.* Write reduction once as a member function which you can call after doing the obvious thing in all of the other operations.

## An Integer Set Datatype (20 Points)

Create a class `IntegerSet` for which each object can hold integers in the range 0 to 255. A set is represented internally as an array of integers, each of which is a bit vector representing 32 of the values that might be in the set. The default constructor initializes a set to the so-called “empty set”, i.e., a set whose array representation contains all zeros.

Provide member functions for the following set operations:

- `unionofSets`, takes a second set as an argument then creates and returns a third set that is the set-theoretic union of two existing sets.
- `intersectionOfSets`, takes a second set as an argument then creates and returns a third set that is the set-theoretic intersection of two existing sets.
- `insertElement`, that takes an integer between 0 and 255 and inserts it into the set
- `printSet`, prints the set as a list of numbers separated by spaces. Print only those elements that are present in the set. Print `---` for an empty set.
- `isEqualTo`, determines whether two sets are equal

Provide an additional constructor that receives an array of integers and the size of that array and uses the array to initialize a set object.

*Hint.* Think of the array of integers as one big bitvector. For a given index into the bitvector, how do you find the corresponding index into the integer array and the offset into the resulting integer.

## A Huge Integer Datatype (20 Points)

Create a class `HugeInteger` that uses a dynamically allocated array of digits to store arbitrary precision integers. A new instance of `HugeInteger` should be able to represent values with up to 40 decimal digits. Provide member functions `input`, `output`, `add`, `subtract`, `multiply`, `divide`, and `modulus`.

For comparing `HugeInteger` objects, provide functions:

- `isEqualTo`
- `isNotEqualTo`
- `isGreaterThan`
- `isLessThan`
- `isGreaterThanOrEqualTo`
- `isLessThanOrEqualTo`

Each of these functions is a “predicate” function that simply returns `true` if the relationship holds between the two huge integers and returns `false` if the relationship does not hold. Also, provide a predicate function `isZero`.

Your arithmetic operations should deal with overflow by dynamically resizing the array of digits, and copying the old values into the new storage.

*Hints* Work out by hand how many digits might be needed as a result of adding two  $n$  digit numbers in the worst case. Similarly for multiplication. Also, devise a simple method of determining when an addition or multiplication overflows.

### **Extra Credit (20 Points)**

Overload the appropriate arithmetic and comparison operators to subsume all of the member operations for your `HugeInteger` class. Write a driver program to test your operators and demonstrate the customized syntax enabled by overloading these operators.