Department of Electrical and Computer Engineering
University of Puerto Rico
Mayagüez Campus

ICOM 4035 - Data Structures - Spring 2002

Project 1: Large Integers
Due: 11:59 PM - Monday, February 4, 2002

1 Objectives

1. Understand the process necessary to implement a C++ class from the specification of its interface.
2. Understand the idea of operator overloading.
3. Gain experience with object-oriented programming concepts, especially constructors.

2 Overview

In this project you will implement, and test a class that represents very large integers. Notice that most computers are limited by how big an integer value can be. In this project, you will work with a class named `Integer` that can be configured to represent integers that cannot be represented by built-in integer types in most computers.

Class `Integer` contains an fixed-size array of integers representing the digits in an integer value. The number of digits is also stored in the class, as well as a flag indicating the sign (positive or negative). Each digit is stored in one of the entries in the array, starting with the least significant value which is stored at position 0, and ending with the most significant at position number of digits - 1. For example, if we have integer:

$$10234567899$$

then, the digit 9 will be stored at entry 0 of the digits array, and digit 1 will be stored at entry 10.

You must implement several constructors to create `Integer` objects from numbers and strings. Moreover, you must overload all the arithmetic (+ , += , - , -= , * , *= , / , /= , % , %=), and relational operators (==, !=, $>$, $>=$, $<$, $<=$) to work correctly with objects of type `Integer`. You will need to recall the algorithms you learned in Elementary School to implement these arithmetic operations.

3 Integer Class Interface

3.1 Private Member Values

3.1.1 Constants

The following constants are provided in the `Integer` class interface:

- `MAX_DIGITS` - maximum number of digits to be found in an `Integer` object.
- `POSITIVE` - flag indicating a positive sign for an `Integer` object.
- `NEGATIVE` - flag indicating a negative sign for an `Integer` object.

3.1.2 Variables

The following variables are provided in the `Integer` class interface:

- `num_digits` - number of digits found in an `Integer` object.
- `digits` - array of `int` storing the digits in an `Integer` object.
- `sign` - the sing of an `Integer` object.

3.2 Private Member Methods

You are free to add as many private member methods as you feel necessary to implement your version of the `Integer` class. It is a good idea to have:

- A constructor that creates a new `Integer` object from a number of digits, an array with the digits, and a sign.
- A method that prints the object to an output stream as a string. This can be used to implement the $<<$ operator (see below).

3.3 Public Member Methods

3.3.1 Integer-based Constructor

Your class must provide a constructor that can create a new `Integer` instance from a value of C++ type `int`. This constructor will also work as the default constructor for the class. The prototype for this constructor is as follows:

```
Integer(int num=0);
```

Here variable `num` holds the value of the integer to be represented by the object instance. As you can see, the parameter defaults to 0, thus any variable of type `Integer` has a default value of 0.

3.3.2 String-based Constructor

Your class must provide a constructor that can create a new `Integer` instance from a C++ `string`. You can assume that any such string will start with either a number, the '+' character, or the '-' character. In the first two cases the string will represent a positive number, and the latter corresponds to a negative value. If the string starts with any other symbol, and if any non-digit character is found in subsequent positions of the string, then you must catch these errors with an assertion. The interface for this constructor is as follows:

```
Integer(string num);
```

3.3.3 Copy Constructor

You do not need to provide a copy constructor. The default copy constructor provided by C++ is sufficient. Recall that the interface for the copy constructor is as follows:

```
Integer(const Integer& num);
```

3.3.4 Destructor

You do not need to provide a destructor. The default destructor provided by C++ is sufficient. Recall that the interface for the destructor is as follows:

```
~Integer();
```

3.3.5 Copy Assignment Operator

You do not need to provide a copy assignment operator. The default copy assignment operator provided by C++ is sufficient. The interface for the copy assignment operator is as follows:

```
const Integer& operator=(const Integer& N);
```

Here, N is the object whose value will now become the value of the object that activates its = operator.

3.3.6 Absolute Value Method

You must implement a method called `abs`() which returns the absolute value of an object of type `Integer`. The object returned must be of type `Integer` and must be independent from the object that activated the call to its `abs`() method. The interface for this method is:

```
Integer abs() const;
```

3.3.7 Operator <<

Overload the << operator to print the contents of an object of type `Integer` to an output stream. If the object is positive or zero, only the digits should be shown. For negative values, the character '-' must be printed before the digits in the number. The interface for this method is:

```
ostream& operator<<(ostream& out, const Integer& N);
```

Here, `out` is the output stream to be written to, and N is the `Integer` value to be displayed.

3.3.8 Operator >>

Overload the >> operator to read an object of type `Integer` from an input stream. This method should simply read a string from the input stream, use the string-based constructor to get an object of type `Integer`, and assign this value to the `Integer` parameter. The interface for this method is:

```
istream& operator>>(istream& in, Integer& N);
```

Here, `in` is the input stream to be read from, and N is the `Integer` value to be read.

3.3.9 Operator ==

You must overload the == operator to determine whether two objects of type `Integer` are equal. This is determined based on the equality of their signs, number of digits, and values at the corresponding positions on the `digits` array. The interface for the == operator is :

```
bool operator==(const Integer& N);
```

Here, N is the object to be compared with the object that activated its == operator.

3.3.10 Operator !=

You must overload the != operator to determine whether two objects of type `Integer` are different. Clearly, this operator is the opposite of the operator ==. The interface for the != operator is:

```
bool operator!=(const Integer& N);
```

Here, N is the object to be compared with the object that activated its != operator.

3.3.11 Operator <

You must overload the < operator to determine whether an object of type `Integer` is less than another object of the same type. This is determined based on signs, and absolute value. Therefore, you implement the `abs()` method before this one, so you can use it. The interface for the < operator is:

```
bool operator<(const Integer& N);
```

Here, N is the object to be compared with the object that activated its operator. The expression represented by this call would be: `*this < N`.

3.3.12 Operator $>$

You must overload the $>$ operator to determine whether an object of type `Integer` is greater than another object of the same type. This is determined based on signs, and absolute value. *Hint: For any two numbers a,b, exactly one of the following must hold: a == b, a < b or a > b.* The interface for the $>$ operator is:

```
bool operator>(const Integer& N);
```

Here, N is the object to be compared with the object that activated its operator. The expression represented by this call would be: $*$`this` $>$ `N`.

3.3.13 Operator $<=$

You must overload the $<=$ operator to determine whether an object of type `Integer` is lesser or equal than another object of the same type. The interface fo the $<=$ operator is:

```
bool operator<=(const Integer& N);
```

Here, N is the object to be compared with the object that activated its operator. The expression represented by this call would be: $*$`this` $<=$ `N`.

3.3.14 Operator $>=$

You must overload the $>=$ operator to determine whether an object of type `Integer` is greater or equal than another object of the same type. The interface fo the $>=$ operator is:

```
bool operator>=(const Integer& N);
```

Here, N is the object to be compared with the object that activated its operator. The expression represented by this call would be: $*$`this` $>=$ `N`.

3.3.15 Operator +=

You must overload the operator += to add an object of type `Integer` to the current value stored by another object of type `Integer`. The object that activates this method **will change** because the resulting `Integer` value will become its new value. The interface for this method is:

```
const Integer& operator+=(const Integer& N);
```

In this case, N is the `Integer` to be added. The expression represented by this method call is: $*$`this` $=$ $*$`this` $+$ `N`.

3.3.16 Operator $- =$

You must overload the operator $- =$ to substract an object of type `Integer` from the current value stored by another object of type `Integer`. The object that activates this method **will change** because the resulting `Integer` value will become its new value. The interface for this method is:

```
const Integer& operator-=(const Integer& N);
```

In this case, N is the `Integer` to be substracted. The expression represented by this method call is: $*\mathtt{this} = *\mathtt{this} - \mathtt{N}$.

3.3.17 Operator $* =$

You must overload the operator $* =$ to multiply an object of type `Integer` to the current value stored by another object of type `Integer`. The object that activates this method **will change** because the resulting `Integer` value will become its new value. The interface for this method is:

```
const Integer& operator*=(const Integer& N);
```

In this case, N is the `Integer` to be multiplied. The expression represented by this method call is: $*\mathtt{this} = *\mathtt{this} * \mathtt{N}$.

3.3.18 Operator /=

You must overload the operator /= to divide the current value stored by an object of type `Integer` by the value of another object of type `Integer`. The object that activates this method **will change** because the resulting `Integer` value (the quotient) will become its new value. The interface for this method is:

```
const Integer& operator/=(const Integer& N);
```

In this case, N is the `Integer` to be used as divisor. The expression represented by this method call is: $*\mathtt{this} = *\mathtt{this}/\mathtt{N}$.
Hint: Think of division as a substraction performed multiple times.

3.3.19 Operator %=

You must overload the operator %= to obtain the remainder after dividing the current value stored by an object of type `Integer` by the value of another object of type `Integer`. The object that activates this method **will change** because the resulting `Integer` value will become its new value. The interface for this method is:

```
const Integer& operator%=(const Integer& N);
```

In this case, N is the `Integer` to be used as divisor. The expression represented by this method call is: $*\mathtt{this} = *\mathtt{this}\%\mathtt{N}$.
Hint: Remainder and division can be implemented with the same algorithm. The difference is that /= keeps the quotient and %= keeps the remainder.

3.4 Non-Member Methods

3.4.1 Operator +

Overload the + operator to add two numbers `N1` and `N2`, returning the computed value in a new object of type `Integer`. Notice that this method cannot modify either of the two operands. The interface for this method is:

```
Integer operator+(const Integer& N1, const Integer& N2);
```

3.4.2 Operator -

Overload the - operator to substract two numbers `N1` and `N2`, returning the computed value in a new object of type `Integer`. Notice that this method cannot modify either of the two operands. The interface for this method is:

```
Integer operator-(const Integer& N1, const Integer& N2);
```

3.4.3 Operator *

Overload the * operator to multiply two numbers `N1` and `N2`, returning the computed value in a new object of type `Integer`. Notice that this method cannot modify either of the two operands. The interface for this method is:

```
Integer operator*(const Integer& N1, const Integer& N2);
```

3.4.4 Operator /

Overload the / operator to divide two numbers `N1` and `N2`, returning the computed value in a new object of type `Integer`. Notice that this method cannot modify either of the two operands. The interface for this method is:

```
Integer operator/(const Integer& N1, const Integer& N2);
```

3.4.5 Operator %

Overload the % operator to get remainder after dividing two numbers `N1` and `N2`, returning the computed value in a new object of type `Integer`. Notice that this method cannot modify either of the two operands. The interface for this method is:

```
Integer operator%(const Integer& N1, const Integer& N2);
```

4 Distribution Files

The following is a summary of the files that you will find in the distribution tar file for this project:

1. `Integer.h` - declaration of the interface for class `Integer`.
2. `Integer.cpp` - implementation of the `Integer` class. YOU MUST IMPLEMENT IN THIS FILE ALL THE REQUIRED METHODS IN THE INTERFACE FOR CLASS INTEGER. ANY OTHER METHOD MUST ALSO BE IMPLEMENTED HERE.
3. `arithmetic.h` - constants and other declaration needed by the main test program.
4. `arithmetic.cpp` - menu-driven main program used to perform user-supplied arithmetic operations on instances of the class `Integer`.
5. `Makefile` - the make file used to compile all modules for this project.
6. `test1.in` - test input file 1.
   NOTE: YOUR PROGRAM MUST PASS ALL TESTS IN THIS FILE WITHOUT ANY ERRORS IN ORDER FOR IT TO BE CONSIDERED A RUNNING PROGRAM.
7. `test1.out` - the expected output from test input file 1.
8. `test2.in` - test input file 2.
9. `test2.out` - the expected output from test input file 2.
10. `test3.in` - test input file 3.
11. `test3.out` - the expected output from test input file 3.
12. `prof_arithmetic` - professor's version of the arithmetic program.