Assignment 5: MyInt Class

Assigned: November 03, 2016
Due: November 16, 2016, 11:59:59pm

Purpose

The purpose of this assignment is to provide you with more experience in working with operator overloading and managing dynamic memory allocation inside a class.

Project Task

One common limitation of programming languages is that the built-in types are limited to smaller finite ranges of storage. For instance, the built-in `int` type in C++ is 4 bytes in many systems today. This allows for about 4 billion different numbers. The regular `int` splits this range between positive and negative numbers, but even a 4 byte `unsigned int` is limited to the range 0 through 4,294,967,295. Heavy scientific computing applications often have the need for computing with larger numbers than the capacity of the normal integer types provided by the system. In C++, the largest integer type is `long`, but this still has an upper limit.

Your task will be to create a class called `MyInt` which will allow storage of any non-negative integer (theoretically without an upper limit – although there naturally is an eventual limit to storage in a program). You will also provide some operator overloads so that objects of type `MyInt` will act like regular integers to some extent (to make them act completely like regular integers would take overloading most of the operators available which we will not do here). There are many ways to go about creating such a class, but all will require dynamic allocation since the variables of this type should not have a limit on their capacity.

The class, along with the required operator overloads, should be written in the files `myint.h` and `myint.cpp`. I have provided starter versions of these files along with some of the declarations you will need. You will need to add in others, and define all of the necessary functions. You can get a copy of the starter files on my website. I have also provided a sample main program to assist in testing, along with a link to some sample test runs from my version of the class.

Documentation

In addition to the project implementation, you will also need to provide a `README` file. Here you will need to document at least 5 errors you encountered and how you solved them. The errors can be some combination of compile errors, linker errors, runtime, and runtime errors. However, you can not use the same errors you used for the first two assignments. The format needs to be a numbered list where each item describes the error type, the error description, and the steps taken to fix the error. For the description, you can just describe the g++ message for compile/linker errors or the crash / wrong output for runtime errors.

You can also include documentation for places where your implementation is unfinished. If you document buggy or unfinished code and describe what the problem is and and general approach to

fixing it, then point deductions will be less severe. This shows me that you are aware of problems in your code and that you at least have an idea of how to fix them. This is completely optional, however, it can only help you as I'm not going to change my test code based on your documentation. That is, if you report an error I would have otherwise missed, you won't lose any points for it.

Part 1: Design a Storage Model

Your class must allow for storage of non-negative integers of any (theoretical) size. You will need to use a dynamic array of integers to represent this. There are many ways to structure this. This simplest is to use each integer to represent a single digit, but this will be expensive in terms of memory consumption. A more memory efficient model would be to use the full range of values of each integer, but implementation is trickier. Other factors to consider are indexing (i.e. do you start indexing with the most significant or least significant digit), and the integer type (e.g. char, short, int, long) to use for each bucket.

In addition, you will need at least one other variable to represent the size of the array. In the first model, this can double as the number of digits (assuming there are no holes in the array). You can use other variables as needed, but make sure they are relevant and that they reside in the private section of your class.

When you decide on a model, make sure to document the details somewhere (e.g. the header file). This gives you something to refer back to when you start implementing your methods, and helps to avoid designing different methods to different targets. Before you begin your implementations, make sure to work things out by hand, thinking about how the steps will map to C++ code.

Part 2: Conversion Constructors

There should be a constructor that expects a regular int parameter, which a default value of 0 so that it also acts as a default constructor. If a negative parameter is provided, set the object to represent the value 0. Otherwise, set the object to represent the value provided in the parameter.

There should be a second constructor that expects a C-string (null-terminated array of characters) as a parameter. If the string provided is empty or contains any characters other than digits ('0' through '9'), set the object to represent the value 0. Otherwise, set the object to represent the value of the number given in the string (which may be longer than what a normal int can hold).

These two constructors will act as conversion constructors. This will allow automatic type conversions to take place, in this case from `int` to `MyInt` and from `char*` to `MyInt`. This makes our operator overloads more versatile as well. For example, the conversion constructor allows the following statements to work:

MyInt x = 1234;
MyInt y = “12345”;
MyInt z = x + 12;

Part 3: Other Constructors

Since dynamic allocation is necessary, you will need to write appropriate definitions of the special

functions: destructor, copy constructor, assignment operator. The destructor should clean up any dynamic memory when a MyInt object is deallocated. The copy constructor and assignment operator should both be defined to make a “deep copy” of the object (i.e. copies all dynamic data in addition to regular memory data) using appropriate techniques. Make sure that none of these functions will ever allow memory leaks in a program (you can check with the Valgrind tool).

Part 4: Resize

Since you are using a dynamic array, you will need to allow for resizing the array when needed. There should never be more than 5 unused array slots left in the array at the end of any public operation. All array resizing functions should be declared in the private section as the user shouldn't have direct control over the data array.

Part 5: Insertion Operator

Create an overload of the insertion operator << for output of numbers. This should print the number in the regular decimal (base 10) format. Do not print extra formatting (e.g. no newlines, spaces, etc).

Part 6: Extraction Operator

Create an overload of the extraction operator >> for reading integers from an input stream. This operator should ignore any leading white space before the number, then read consecutive digits until a non-digit is encountered (this is the same way that >> works for a normal int). This operator should only extract and store the digits in the object. The first non-digit encountered after the number may be part of the next input and should not be extracted. You may assume that the first non-whitespace character in the input will be a digit (i.e. you do not have to error check for entry of an inappropriate type like a letter).

Example: Suppose the following code is executed and the input typed is “ 12345   7894H”.

```
MyInt x, y;
char ch;
cin >> x >> y >> ch;
```

The value of x should now be 12345, the value of y should be 7894, and the value of ch should be 'H'.

Part 7: Comparison Operators

Create overloads for all 6 of the comparison operators (< , > , <= , >= , == , !=). Each of these operations should test two obejcts of type MyInt and return an indication of true or false. You are testing the MyInt numbers for order and/or equality based on the usual meaning of order and equality for integer numbers.

Part 8: Addition Operator

Create an overload version of the + operator to add two MyInt objects. The meaning of + is the usual meaning of addition on integers, and the function should return a single MyInt object representing the sum.

Part 9: Multiplication Operator

Create an overload version of the * operator to multiply two MyInt objects. The meaning of * is the usual meaning of multiplication on integers, and the function should return a single MyInt object representing the product. The function needs to work in a reasonable amount of time, even for large numbers. This means that implementation of x*y where you add x to itself y times will take too long for large numbers.

Part 10: Increment Operators

Create the overloads of the pre-increment and post-increment operators (++). These operators should work just like they do with integers. Remember that the pre-increment operator returns a reference to the newly incremented object, but the post-increment operator returns a copy of the original value (before the increment).

General Requirements

- As usual no global variables other than constants
- All member data should be private
- Use appropriate good programming practices as denoted on previous assignments
- Since the only output involved with your class will be in the <<operator, your output much match mine exactly when running test programs.
- You may not use classes from the STL (Standard Template Library) – this include the class <string> – as the whole point of this assignment is for you to learn how to manage dynamic memory issues inside of a class yourself, not rely on STL classes to do it for you.
- You may use standard I/O libraries like iostream and iomanip as well as the common C libraries like cstring and cctype.

Extra Credit

1) Create an overloaded version of the – operator to subtract two MyInt objects. The meaning of – is the usual meaning of subtraction on integers, with one exception. Since MyInt does not store negative numbers, any attempt to subtract a larger number from a smaller one should result in the answer 0. The function should return a single Myint object representing the difference.

2) Create an overloaded / operator and a % operator for division of two MyInt objects. The usual meaning of integer division should apply (i.e. / gives the quotient and % gives the remainder). Both functions should return their result in a MyInt object.

Note: The extra credit operations must also run in a reasonable amount of time. Long lags of multiple seconds on large values will not be acceptable.

Submitting

- Remove all binary files, executables, and driver code
- Use the tar utility to archive your `myint.cpp`, `myint.h`, and `README` files
- Name the tar file according to the scheme: a5_<*last_name*>_<*first_name*>.tar

- Submit the tar file to the appropriate blackboard assignment link

Tips

Storage: The suggested storage is a dynamic array in which each slot represent on digit of the number. One way to do this is use an array of integers, another would be an array of characters. Both have advantages and disadvantages. An integer array would be easier for arithmetic calculations, but a character array would be easier for input purposes. You may choose your preference, but remember that the integer digit (e.g. 6) and its corresponding character representation (e.g. '6') are not the same value.

In case you want to use them, here are two functions you can use to converting between single digit integers and their corresponding character codes (These functions are only for use with single digits). You may use them if you like, as long as you cite the source in your comments. These are stand-alone functions (not members of any class). Copes of these functions are already placed in your starter myint.cpp file.

```
int C2I (char c)
// converts character into integer (returns -1 for error)
{
    if (c < '0' || c > '9')     return -1;   // error
    return (c - '0');                        // success
}

char I2C(int x)
// converts single digit integer into character (returns '\0' for
error)
{
    if (x < 0 || x > 9)         return '\0'; //error
    return (static_cast<char>(x) + '0');     //success
}
```

Input: For the >> operator overload, there are some issues to be careful about. You will not be able to just use the normal version of >> for integers because it attempts to read consecutive digits, until a non-digit is encountered. The problem here is that we will be entering numbers that go beyond the capacity of a normal int. So you will probably find it necessary to read one digit at a time (which means one byte or character). Because of this, you may find the above conversion functions useful.

You already know of some istream member functions like getline, and you have used many versions of the >> operator on basic types. One thing worth keeping in mind is that when the >> operator is used for built-in types, any leading white space is automatically ignored. However, there are a couple of other istream functions you might find useful:

```
int get();
int peek();
```

The get function extracts and returns the next character on the stream, even if it is a white space character like newline. This function does not skip white space like the built-in >> functions do. The peek function just looks at and returns the next character on the stream, but it does not extract it. This

function is good for seeing what the next character is (e.g. if it is a digit) without actually picking it up.

Comparison Overloads: Remember that once a function is written, other functions can call it. For example, once you have written the details of comparing two numbers with the < operator, it should be very easy to define the > operator by calling the < operator to do the work. You can't write them all this way, because you would get stuck in an infinite loop, but you can make the job easier by defining some of these in terms of others that are already written.

Addition Operator: This is a more difficult function than it sounds like. However, the algorithm should be conceptually easy. You'll probably find it most helpful to break this algorithm down to the grade school step-by-step level, performing an addition digit by digit. And don't forget that some additions can cause a carry.

Multiplication Operator: This one will be a little more complex than the addition overload, but again, you should think back to the algorithm you learned for multiplying numbers in grade school. The same algorithm can be applied here. Don't forget the carries.