Due Date: Apr 3, 2020 @ 11:59:59

Operator Overloading - InfiniteInt

This assignment will focus on using a few features of C++ operator overloading as well as continuing your exploration of classes. You may already know from previous courses that a 32-bit unsigned integer has a maximum positive value of 4,294,967,295. In this assignment, you will write a class which represents a non-negative number with an "unlimited" number of digits. Each digit will be stored as a separate int in an STL vector, and your job will be to implement some commonly used operators on your new infinite int class.

You will submit your work to Gradescope and it will be both autograded and human graded. The autograder will compile your code with g++-7.4.0. Be sure to watch Piazza for any clarifications or updates to the assignment.

Don't forget the Assignment 5 Canvas Quiz! Your assignment submission is not complete without the quiz.

InfiniteInt Overview

Your new Infinite Int implementation must consist of 3 files: `InfiniteInt.h`, `InfiniteInt.cpp`, and `demo.cpp`. The first being the header file, the second being the implementation of the operators, and the third being the `main()` function implementation which tests the functionality of your operators.

For `InfiniteInt.h`:

- Declare your class member variables and functions.
  - InfiniteInt requires two private member variables:
    1. A container to store the digits of the InfiniteInt: `std::vector<unsigned int>* digits`
       - Store the digits where the most significant digit is in the lowest index position. For example, the number `42` would be stored with `4` in the 0th index position and `2` in the 1st index position.
       - InfiniteInt is unable to represent a number less than 0, so the smallest value should be represented with a vector of size 1 containing 1 digit: `0`.
       - Your vector should not have any leading 0s in the most significant digit positions and the size of the vector shouldn't be larger than necessary (ex: `42` should have a vector of size 2, `412` should have a vector of size 3, etc.)
    2. A constant unsigned integer to represent the base of your number: `const unsigned int radix`. For this assignment, all of our InfiniteInts are base 10, but with generics, this could later be extended to represent a number in any radix. Use this variable in your implementation of operators instead of hard-coding the number 10.
  - There are no private functions required, but you may create private helper functions if you wish.
- Be sure to have include guards.
- Don't import anything that is unnecessary.

- Include file header comments that have your name, student id, a brief description of your new class, and be sure to cite any resource (site or person) that you used in implementing this assignment.
- Each function operator should have appropriate function header comments.
  - You can use javadoc style (see doxygen) or another style, but be consistent.
  - Be complete: good description of the function, parameters, and return values.

For `InfiniteInt.cpp`:

- There are 17 operators, 4 constructors, and 1 destructor that you will have to implement. This may seem like a lot but we've simplified a lot of the functionality for you:
  - Many of the operators can use implementations of other operators to get their results (how can use you use your addition operator to implement the pre and post-increment operators?)
  - Because InfiniteInt is an unsigned number, you don't have to worry about negative cases for your operators.
  - We've restricted the main operators to basic comparisons, addition, and subtraction.
- All the function bodies should be here (except `main()`).
- Don't clutter your code with useless inline comments, unless it is the "why".
- Follow normal programming practices for spacing, naming, etc: be reasonable and consistent.
- Be sure to avoid redundant code.

For `demo.cpp`:

- The implementation of `main()` should be here. Only one call to each operator is required, but I would try to test each operator thoroughly and in different ways to make sure each operator has its expected behavior.
- You can use this starter code to test your constructors, copy Op=, and stream insertion operator. These tests are by no means comprehensive, and you should think of other tests for your constructors as well. Add more tests at the end for your remaining operators before submitting.

The following functions should be the only public functions in your class. You are allowed to create other private helper functions, but they should remain private.

Functions to implement:

`int main()` You are required to test each operator at least once in `main()`. Feel free to write more tests to fully test your implementations of the operators. We will not be testing the output of `main()`, only that you call each function once.

`std::vector<unsigned int>* getDigits()` A getter function for the digits member variable.

`InfiniteInt()` Create a default constructor for InfiniteInt that sets the value of your InfiniteInt to a default value of 0. As in all constructors, make sure to manage any heap memory that your InfiniteInt may need.

`InfiniteInt(unsigned long long val)` Create a parameterized constructor for InfiniteInt that sets the starting value to val.

`explicit InfiniteInt(std::vector<unsigned int>* new_digits)` Create a parameterized constructor for InfiniteInt that sets the starting value equal to the digits pointed to by `new_digits`. Think about why this constructor should be explicit and not the one above.

`InfiniteInt(const InfiniteInt &obj)` Create a copy constructor for InfiniteInt that allocates new memory for `digits` and then copies each digit from the obj vector to the newly allocated vector.

`InfiniteInt& operator=(const InfiniteInt &)` Overload the copy-assignment operator. This will look similar to the copy constructor, but make sure to perform a self-assignment test (Hint: look at the rule of 3 example in the Classes lecture slides).

`~InfiniteInt()` Create a destructor for InfiniteInt which frees any allocated heap memory.

`operator<< InfiniteInt` Write an overloaded operator `<<` for ostream and InfiniteInt. This should display the integer value of the InfiniteInt as if it was a regular unsigned int. As discussed in class, do **not** add an `endl` character after printing the number.

NOTE: the ostream object needs to be passed by reference and returned by reference. (Hint: see Classes lecture sample code)

`operator>> InfiniteInt` Write an overloaded operator `>>` for istream and InfiniteInt. This operator should wait for the user to enter a single string of numbers, which will then be parsed into the vector of digits. If the user enters any characters other than valid digit numbers `(0-9)` in their input string, you should print "Invalid entry" with an `endl` character to `cout` and return without modifying the digits of the InfiniteInt.

`InfiniteInt < InfiniteInt` Write an overloaded operator `<` that returns whether or not the value of the left hand side InfiniteInt is less than the other InfiniteInt.

`InfiniteInt > InfiniteInt` Write an overloaded operator `>` that returns whether or not the value of the left hand side InfiniteInt is greater than the other InfiniteInt.

`InfiniteInt <= InfiniteInt` Write an overloaded operator `<=` that returns whether or not the value of the left hand side InfiniteInt is less than or equal to the other InfiniteInt.

`InfiniteInt >= InfiniteInt` Write an overloaded operator `>=` that returns whether or not the value of the left hand side InfiniteInt is greater than or equal to the other InfiniteInt.

`InfiniteInt == InfiniteInt` Write an overloaded operator `==` that returns whether or not the value of the left hand side InfiniteInt is equal to the other InfiniteInt.

`InfiniteInt != InfiniteInt` Write an overloaded operator `!=` that returns whether or not the value of the left hand side InfiniteInt is not equal to the other InfiniteInt.

`InfiniteInt + InfiniteInt` Write an overloaded operator `+` that returns a InfiniteInt that is the sum of two InfiniteInts. Make sure to carry out and create a new most significant digit when necessary.

`InfiniteInt - InfiniteInt` Write an overloaded operator `-` that returns a InfiniteInt that is the difference of two InfiniteInts. If the result of this subtraction would create a negative number, return an InfiniteInt with a value of 0.

`InfiniteInt += InfiniteInt` Write an overloaded operator `+=` that adds the right operand and the left operand and stores the new value in the left operand.

`InfiniteInt -= InfiniteInt` Write an overloaded operator `-=` that subtracts the right operand from the left operand and stores the new value in the left operand.

`++InfiniteInt` Write an overloaded pre-increment operator, `++`, that adds 1 to the current InfiniteInt and returns the changed InfiniteInt.

`--InfiniteInt` Write an overloaded pre-decrement operator, `--`, that subtracts 1 from the current InfiniteInt and returns the changed InfiniteInt.

`InfiniteInt++` Write an overloaded post-increment operator, `++`, that adds 1 to the current InfiniteInt and returns the original InfiniteInt.

`InfiniteInt--` Write an overloaded post-decrement operator, `--`, that subtracts 1 from the current InfiniteInt and returns the original InfiniteInt.

Notes:

- Don't forget to manage your memory! Because the private member variable `digits` of InfiniteInt is a pointer, you will need to allocate and delete memory when constructing new InfiniteInts and when writing the destructor.
- As you write each function, make sure its prototype is correct. Follow the rules for const-correctness (everything that can be const should be const), and mark return values as return-by-reference when appropriate.
  - We'll be manually grading your function headers and checking for return by value/reference as well as const-correctness.

Makefile

Your submission must include a `Makefile`. The minimal requirements are:

- Have a rule to build an object file for each module (header and source pair). Be sure to have the appropriate dependencies for each rule.

- Compile using the demo executable with the `make` command.
- Use appropriate variables for the compiler and compiler options.
- It must contain a clean rule.
- The demo executable must be named something meaningful (not a.out).

README.md

Your submission must include a `README.md` file (see https://guides.github.com/features/mastering-markdown/). The file should give a brief description of the program, the organization of the code, how to compile the code, and how to run the program.

Submission

You will upload all the required files to Gradescope. Again, don't forget to complete the Assignment 5 Canvas quiz.

There are no limits on the number of submissions. See the syllabus for the details about late submissions.

Grading

For this assignment, roughly half of the marks will come from the autograder. For this assignment, none of the test details have been hidden, but there may be hidden tests in future assignments.

The other half of the marks will come from human-grading of the submission. Your files will be checked for style and to ensure that they meet the requirements described above. In addition, all submitted files should contain a header (comments or otherwise) that, at a minimum, give a brief description of the file, your name, and wisc id.

HAPPY CODING!