Due Date: Apr 13, 2022 @ 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 `unsigned int` in an STL vector, and your job will be to implement some commonly used operators on your new infinite integer class.

Your will submit your work to Gradescope and it will be both autograded and human graded. The autograder will compile your code with g++-7.5.0.

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 Integer 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. As with the other assignments, you will also submit a `Makefile`, and a `README.md`.

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.
  - 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 the functionality, and be sure to cite any resource (site or person) that you used in your implementation.
- Be sure to follow the best practices for operator overloading especially using friend when appropriate.
- Each function 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.
  - In your class declarations (and header files in general), only inline appropriate functions.

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. If an operation would go negative, the value returned should be 0.
  - We've restricted the main operators to basic comparisons, addition, and subtraction.
- All the function bodies not defined in the header file should be here.
- Don't clutter your code with useless inline comments (it is a code smell [Inline Comments in the Code is a Smell, but Document the Why], unless it is the why).
- Follow normal programming practices for spacing, naming, etc.: Be reasonable and consistent.
- Be sure to avoid redundant code.

For classes, in general:

- Remember the rule of 3: Explicitly define the destructor, copy operator=, and copy constructor if needed (i.e. for RAII).
- Use `= default` to explicitly use the default version of default constructor, copy constructor, destructor, and copy operator=.
- Reuse the code better by calling base class functions and constructors from derived classes.
- Do not make a member variable public or protected, unless it makes sense to do so.
- Use the keyword `override` for all functions that override a parent's method.
- Use *initializer list* for constructor delegation, constructor chaining and setting class member values.

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 behaviour.
- You can use the `demo.cpp` found on the Canvas site 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.

You have been provided a `demo.cpp` on Canvas with some tests to get you started.

Functions to Implement:

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.

For `demo.cpp`:

`int main()` You are required to test each operator at least once in `main()` found in `demo.cpp`. 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.

For `InfiniteInt`:

`std::vector<unsigned int>* getDigits()` A getter function to return the pointer to the digits vector.

`InfiniteInt()` Create a default constructor for `InfiniteInt` that sets the integer value 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> &d)` Create a parameterized constructor for `InfiniteInt` that sets the starting value equal to the digits of `d`.

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

`InfiniteInt& operator=(const InfiniteInt &)` 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. 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 parse the input stream by skipping any whitespace, then parse all digits into the vector of digits, stopping when a non-digit is encountered. If no digits are encounter, the value should be set to 0.

`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 `InfiniteInt`s. 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 `InfiniteInt`s. 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 `InfiniteInt`s and when the destructor is called.
- 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`).
  - Mark return values as return-by-reference when appropriate.
  - Be sure to use friend methods for the operator overloading when appropriate.
- We'll be manually grading your function headers and checking for return by value/reference as well as `const`-correctness, and proper use of friend methods.

Makefile

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

- Compile using the demo executable with the `make` command.
- Use appropriate variables for the compiler and compiler options.
- Have a rule to build the object file, with appropriate dependencies, for each module (header and source pairs).
- 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. There are no limits on the number of submissions. See the syllabus for the details about late submissions.

Grading

For this assignment, 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.

Rubric

- Autograder Tests - 40
- README File - 5
  - Name / student id
  - Typeset/organized with markdown
  - Program description
  - Compilation instructions
  - Run instructions
  - Code organization description
- Makefile - 5
  - File header comments including name/id
  - Demo build rule with appropriate named executable
  - InfiniteInt.o build rule
  - Clean build rule
  - Correct file name
- InfiniteInt.h - 12
  - File header comments including name/id
  - Include guards
  - All includes are required for header file
  - Function header comments with function description, return value and parameters
  - Comment detailing other elements like structs
  - Function prototypes include default values as per the specification
  - Only inline function bodies are defined in header
  - Correct use of `const`
  - Follows best practice for operator overload especially correct use of friend vs member
  - Follows rule of big 3, uses default where appropriate
  - Uses override when appropriate
- InfiniteInt.cpp - 13
  - File header comments including name/id
  - No unused/unneeded includes
  - Consistent style (braces, spacing, etc)
  - Avoid inline comments
  - Avoid duplicate code, call parent methods and use helper functions as needed
  - Avoid c-style casting
  - Avoid gotos
  - Well-designed functions

Tackling Assignment 5

1. Create a directory for your assignment 5 project, and create the required files: `InfiniteInt.h/.cpp`, `README.md`, and `Makefile`. Download the `demo.cpp` provide in Canvas to help get you started.
2. Get things compiling: Set up the InfinitInteger class and the function stub details. Then, set up your makefile. (Pro tip: When adding you function prototypes, also do the function header comment). For each function:
   - Make sure to follow the best practices for operator overloading: should it be a friend or a member?
   - Is everything `const` that can be `const`?
3. Get the constructors and memory management (big 3) working, and the output operator.
4. Next move on to the comparisons. Start with $<$ or $>$ – all the rest should follow. You can test these in gradescope.
5. Next, work on $+$. You'll have to do the addition digit by digit and worry about the carry. Once $+$ is done, the other addition operators should follow. You can test these in gradescope.
6. Next, work on $-$. You'll have to do the subtraction digit by digit and worry about the borrowing. Once $-$ is done, the other subtraction operators should follow. Remember if the subtraction goes negative, the result is just calculated as 0. You can test these in gradescope.
7. Finally, get the input operator working.
8. Don't forget to do the `README.md` file.
9. Final submission: make sure you have file comments, and name on all files. Double check the rubric, and do your final submission to Gradescope.

HAPPY CODING!