Assignment Classes and Memory: Classes, Pointers and Dynamic Memory

COSC 2336: Data Structures and Algorithms

Summer 2021

Objectives

- Practice using pointers and dynamically allocating memory
- Look at separating class declaration from implementation
- Learn how to define classes over a multi-file project, with class declarations in the class header file and member function implementations in the implementation file.
- Practice defining classes and class member functions in C++.
- Write member functions for classes to implement an abstract data type.
- Continue practicing test driven development and git project workflow

Description

In C++ the largest int value is 2147483647. So, an integer larger than this cannot be stored and processed as an integer. Similarly if the sum or product of two positive integers is greater than 2147483647, the result will be incorrect.

One way to store and manipulate large integers is to store each individual digit of the number in an array. In this assignment, you will design a class named `LargeInteger` such that an object of this class can store an integer of any number of digits. Your abstract data type `LargeInteger` will support member functions to `add()` two `LargeInteger` objects together and return a new resulting `LargeInteger` object. We will only implement an unsigned integer, we will not worry about handling negative large integers in this assignment. Likewise, subtraction and especially multiplication are a bit more complicated, and we will leave those operations for later. But in addition to the `add()` member function, you will implement several other member functions and constructors, that will be useful to implementing addition of the `LargeInteger` values.

In order to support arbitrarily large integers, you will also get a chance to practice dynamic memory allocation. We will be managing a block of memory to hold the digits of the large integer, and this block will be able to grow as needed to handle any number of digits (that is until we exhaust memory, which would be a lot of digits indeed).

Overview and Setup

For this assignment you will be given the following files that you will be using and adding code to for this assignment.

File Name	Description
`src/test-LargeInteger.cpp`	Unit tests for the `LargeInteger` class you are to implement.
`include/LargeInteger.hpp`	Header file for the declarations of the `LargeInteger` class and its defined API.
`src/LargeInteger.cpp`	Implementation file for the `LargeInteger` member functions that implement the API and class functionality.

As usual, before starting on the assignment tasks proper, you should make sure you have completed the following setup steps.

1. Copy the assignment repository on GitHub using the provided assignment invitation link for 'Assignment Classes

and Memory' for our current class semester and section.
2. Clone the repository using the SSH url to your local class DevBox development environment.
3. Configure the project by running the `configure` script from a terminal.
4. Confirm that the project builds and runs, though no tests will be defined or run initially. If the project does not build on the first checkout, please inform the instructor.
5. You should create the issue for Task 1 and/or for all tasks for the assignment now before beginning the first task. On your GitHub account, go to issues, and create it/them from the issue templates for the assignment.

Assignment Tasks

Task 1: Implement `tostring()` Member Method

If you haven't already, create the Task 1 issue on GitHub and associate the task 1 issue with the `Feedback` pull request in your class repository.

For the first task you will provide an implementation of the `tostring()` method. We need this method to perform some of our unit testing on the `LargeInteger` class.

This function will return a string representation of the `LargeInteger`. You should use a `<sstream>` output string streams to implement this function. There were some examples of using string streams in this weeks lecture videos and materials.

Your `LargeInteger` class will have an array of integer values. This array will be stored in the member variable named `digit`. If you look in the `LargeInteger.hpp` header file you were given that gives the declarations of this class, you will notice that `digit` is defined as:

```
/// @brief digits A dynamically allocated array of integers.  This array
///   holds the digits of the large integer this object represents.  The
///   digits in the array are ordered such that the 1's place (10^0) is in
///   index 0 of the array, the 10's place (10^1) is in the index 1, and so
///    on.
int* digit;
```

This is a pointer to an `int`. However, as discussed in our materials for this unit, we can allocate a block of integers that `digit` will point to, and after we do that we can use it just like a regular C array of `int` values, e.g. `digit[0]` to access the first digit of our `LargeInteger`, etc.

However, note that the digits of the `LargeInteger` are to be stored in reverse of their display order. That is to say that the 1's place (10^0) is in index 0 of `digit[]`, the 10's place (10^1) is in index 1, etc. So to be clear, if we are representing the integer value 487 in our `LargeInteger`, we should end up with

```
digit[0] = 7 // The ones place  10^0 has a 7's digit
digit[1] = 8 // The 10's place  10^1 has an 8's digit
digit[2] = 4 // The 100's place 10^2 has a 4's digit
```

You should be clear about this representation. This makes it easy to interpret the individual digits according to their correct magnitude as a power of 10, as the index in the array corresponds to the power of 10 of the digit.

So back to implementing the `tostring()` method. Given a value of 487, notice that you will have to display the 4 first in the output string, followed by 8 and 7. So you will have to iterate through the `digit[]` array in reverse order. Notice that the other member variable, named `numDigits` will tell you how many digits are in your `LargeInteger` currently.

Your `tostring()` is a member method of the `LargeInteger` class. So don't forget when you add the method to the implementation file, you will need to indicate that the method is a member of the `LargeInteger` class. The `tostring()` member function does not take any input, and it should return a `C++` string as its result. Also you should be aware that the `tostring()` method does not modify the value of the `LargeInteger` if it is called. Thus it is an accessor method that simply returns information about the `LargeInteger`. Any accessor method that doesn't cause the object to change needs to be declared as a `const` member function. We discussed `const` member functions in this unit when we looked at class members. If you are unclear on the concept, you should review what it means to be a `const` member function, and how to declare a function to be a `const` member function.

It is suggested, as usual, that you start by uncommenting the first test case in the unit test file, which will be testing basics of the `tostring()` method. A constructor has already been implemented for you that allows for `LargeInteger` objects to be created with an initial value. You should start by adding in the correct prototype of your member function into the header file, and adding a stub function that simply returns an empty string `""` initially. Do this to make sure that your project can compile and run the tests (though they will be failing initially) before continuing. Don't forget that your `tostring()` must be a member of the `LargeInteger` class. This means that you have to indicate it is a member function in the implementation file, something like this:

```
string LargeInteger::tostring() const
{
  return ""
}
```

This is an example of the stub function you need so your project can compile and run the tests after uncommenting the first test case. Once you add the stub implementation, build your project and run the tests. The project should compile, and the test(s) for the stub you just added should fail. After that you can do the actual implementation of the `tostring()` member method so that is passes the first set of tests.

When you are satisfied with your work and you can compile and pass the first set of tests, you should commit your changes and push them to the `Feedback` pull request of your class repository.

Task 2: Implement Array Based `LargeInteger` Constructor

Make sure you have created Issue 2 for this task. And it would be best to close and merge your first pull request before continuing on to Task 2.

You were given two different class constructors for the `LargeInteger` class, both of which were used in the first set of unit tests. One constructor, the default constructor, simply initializes the `LargeInteger` object to have a single digit with a value of 0. The other constructor initializes itself from a regular `int` type provided as a parameter. It does this by parsing the individual digits of the supplied `int` parameter in order to dynamically allocate and fill up the `digit[]` array initially.

You should study these two existing class constructors, and make sure you understand them. Especially, notice how both of them are actually dynamically allocating an array of `int` values and initially assigning and constructing the digits in the `digit[]` array for the `LargeInteger` object.

There is one more constructor we would like to have that will be very useful in our implementation of some of the `LargeInteger` operators. We would like to be able to construct a `LargeInteger` if given an existing array of `int` values. In many ways this is a simpler task than constructing the array from a single `int` value, we don't have to do any parsing of the digits, simply copy the digits from the given input array into our `digit[]` array.

Start by creating a stub implementation of your new constructor. You should uncomment second unit test in our test file, which has the tests for the constructor you are to create here. I would suggest you copy and paste the default constructor to begin with. Add in the function prototype to the header file, and an empty implementation in the implementation file. This constructor differs from the other existing constructors because of the input parameter it expects. Your constructor here should be expecting two parameters. The first parameter is the size of the array of integers it will be given to initialize the `LargeInteger` object with. The second parameter is the actual array of integer digits to use for the initialization. You will have to get the signature of the constructor correct in order to compile your project. But your implementation can do nothing, or be a copy of the default constructor, initially. Do this to make sure you can compile your project and run the second set of tests before trying to implement the function.

Your implementation will be similar to the constructor that constructs the digits from an existing integer. You will need to dynamically allocate a new array of `digit[]` integers. However, unlike the existing constructor, you are given explicitly as the first parameter the size that this array needs to be when you construct it. Once the array has been dynamically allocate, you can then simply copy over the digits from the given input array, into your `digit[]` array.

Once you have implemented your new constructor, the tests in the second test case should now all be able to pass. When you are satisfied this is all working so far, commit your changes and push them to the `Feedback` pull request.

Task 3: Implement `maxDigits()` Member Function

Make sure that you have created Task 3 and are ready for a new pull request before beginning this next task.

The third unit test test the `maxDigits()` member function you are to implement next, though it is proceeded by a test framework of some `LargeInteger` instances we will need to use in this and the remaining tests, so uncomment those as well. As usual, you should then create the function prototype in the header file, and create a stub function for `maxDigits()` that returns 0, so you can be sure you project is compiling and running.

The purpose of the `maxDigits()` function is to compare the number of digits of `this` instance to the number of digits in another `LargeInteger` instance, and return the number of digits contained in the larger of the two. This method will be useful in implementing some of our operators in a bit. Thus this method will return a simple integer result. It needs to take another `LargeInteger` as its input parameter.

NOTE: however, the parameter should be passed in with the following modifications. First of all, we should never be modifying this other `LargeInteger`, so the parameter should be declared to be a `const` parameter. Also, it can be expensive to copy in an object like the `LargeInteger` by value, because in theory if the object has a large number of digits, these would need to be copied before we call the function. So you should be passing in this parameter by reference, instead of by value. We discussed `const` parameters, and passing parameters by reference vs. passing them by value in the previous unit and current units.

NOTE: also, this member function should be a `const` member function, like the `tostring()` you already implemented. Again there is a difference between a parameter being a `const` parameter, and a member function being a `const` member function. If you are unclear of the difference, you should review those concepts now. This function should not cause either the input `LargeInteger` object to be changed, nor `this LargeInteger` object to be changed, thus the parameter should be constant, and this member function needs to be a constant member function.

The implementation of this function should hopefully not be that difficult once you get the function signature declaration correct. It should be noted that when you are in a class member function, you can not only access the `private` member variables of `this` instance, but you can also access `private` members of other instances that are passed in as parameters. This is useful in this function. To determine which instance has the larger number of digits, you simply need to compare the `numDigits` of the two instances and return the large of the two as the result of this function.

Implement your `maxDigits` member function to determine and return the larger of the number of digits in the two instances. Once you have done this, the tests in the third test case testing this function should now be passing. When you are satisfied with your work, and the project still compiles and now passes all of the unit tests in the first 3 test cases, do the usual to create and push a commit, update the pull request, merge in the changes and close the third issue covering this task.

Task 4: Implement `digitAtPosition()` Member Function

Perform the usual prerequisite steps before starting task 4.

We will also need the `digitAtPosition()` member function in our implementation of the `add()` operator, which is what we are working towards with this project.

The `digitAtPosition()` member function simply returns the digit at the indicated position. For example if our `LargeInteger` object contains the digits for the number `376`, which would be represented in the `digit` array as:

```
digit[0] = 6  // the 10^0 or one's digit
digit[1] = 7  // the 10^1 or 10's digit
digit[2] = 3  // the 10^2 or 100's digit
```

we could ask this function for example "what is the digit at position 2 (the 10^2 digit)" and it should return an answer of `3` because we have 3 100's in the 10^2 place.

So this function takes an integer index as input, and it returns an integer digit as its result. This function should not modify the `LargeInteger` object, so it again should be a `const` member function. As usual, it is suggested that you uncomment the test case for this member function, create a stub function and function prototype, and make sure the project still compiles and runs the tests before beginning to implement the function.

Also there is a slight wrinkle that your function is required to handle. In the above example, we have 3 digits in our `LargeInteger`, `376`, and the `numDigits` should be 3 for this object. If your function is asked to return the digit at place 3, this is actually the 10^3 or 1000s digit. But your `digit[]` array is dynamically allocated to only have 3 digits, so the valid indexes are from 0 to 2, there is no digit at index 3, this is beyond the bounds of the `digit[]` array.

If a request is made for a digit place beyond the bounds of the array, you should return 0 as the answer. This is true for this `LargeInteger` example, there are 0 1000s in this digit position. Likewise, if a request is made for a negative position, like `digitAtPosition(-1)` you should also return 0. This check both guards your class from illegally trying to access beyond the bounds of your array, and it will also be a useful feature later when we implement the `add()` operation. Both of these conditions are also checked in the unit tests for this function.

Once you are satisfied with your implementation of `digitAtPosition()`, you should perform the usual steps to finish Task 4 by committing and pushing your implementation to your repository, and closing and merging the pull request and issue for this task.

Task 5: Implement `appendDigit()` Member Function

Perform the usual prerequisite steps before starting task 5.

This function again will be reused to implement your `add()` operation in a clean way. The purpose of this function, as its name suggests, is to append a new digit onto the `LargeInteger`. So for example, if the `LargeInteger` instance currently has the value of `376` and you ask it to `appendDigit(1)`, its new value will be `1376`. Notice that the new digit becomes the new most significant digit of the `LargeInteger` instance.

This function will take a single integer parameter as input, which is the new digit to be appended to the `LargeInteger`. This function doesn't return any explicit result, so it should be a `void` function. This function will be modifying the `LargeInteger` instance, so it will not be a `const` member function in this case. As usual it is strongly suggested that you first add in a stub function and uncomment the unit tests testing the `appendDigit()` function and making sure you project compiles and still runs all of the tests (and is passing all of your previous tests still), before starting on your implementation.

This method will give you more practice in managing memory and creating memory dynamically. When you implemented your constructor, you had to allocate a new block of memory correctly. Here we need the size of the `LargeInteger` to grow by 1, so that our `digit[]` array is 1 bigger so that it can hold the new digit we are appending.

The suggested algorithm you should perform is as follows:

1. Allocate a new array of digits of size numDigits+1 (e.g. enough space to hold one more digit).
2. Copy the old digits to the new array of digits you created.
3. Append the new passed in digit to the end of the array.
4. Free up the old original array of digits.
5. Re-point your `digit` member variable to your newly allocated space that has grown in size by 1.

To understand how this works correctly, you have to understand both dynamic memory allocation, and how pointers work in C/C++. Recall that `digit` member variable is a `int*`, we point it to a block of integer values, and can then treat it as a regular C array. However, if we want to, we can create a new block of memory and re-point `digit` to that new block. If you are fuzzy about any of this, you should review this units materials on pointers and memory management concepts.

One final note, you are required to detect if a request to append a digit of 0 is made. If a 0 were to be appended onto the `LargeInteger` `376`, the result would be `0376`. However we normally do not keep track of nor represent 0's past the last most significant non-zero digit. If a 0 is asked to be appended, just simply ignore the request and do nothing. This again will make the implementation of the `add()` operation easier.

Once you implement the `appendDigit()` as described, it should be able to pass all of the given unit tests for this task 5. As usual when you are satisfied, create a commit and push it to your `Feedback` pull request.

Task 6: Implement `add()` Operation Member Function

Perform the usual prerequisite steps before starting task 6.

We will be reusing most all of the previous functions to implement the `add()` operation. As usual, before you start working on this function, first uncomment the final set of test cases, add in the prototype for the new member function, and create a stub function, so you can be sure you project is still compiling and running the tests.

Lets give a quick example, though you can look at the unit tests to see similar examples of how `add()` is supposed to work.

```
LargeInteger li1(123);
LargeInteger li2(987);
LargeInteger result;

result = li1.add(li2);
```

The result will be a `LargeInteger` with the digits $123 + 987 = 1110$.

Notice that the `add()` member function takes another `LargeInteger` as its input parameter. Similar to the `maxDigits()` function you implemented previously, the other `LargeInteger` should not be modified when it is used as an operand, so it should be passed in as a `const` parameter. Also again, we don't want to possibly have to make an expensive copy of this parameter, so the parameter needs to be passed in by reference.

Also notice that a new `LargeInteger` is returned as the result of performing the `add()` operation. So your `add()` member function needs to return a reference to a `LargeInteger` as its result, and you will need to create the new instance to be returned inside of this member function dynamically. Again for the same reason as before, it is important that you are returning a `LargeInteger` reference, not an actual value. This will avoid a copy of the object having to be made on the return. But this forces you to create the object dynamically to be returned in your `add()` function.

Here is the suggested algorithm you should try to implement for the `add()` operator:

1. Dynamically allocate a new array to hold the resulting sum of this and the other `LargeInteger`. Reuse the `maxDigits()` member function to determine the size needed for this new array. The size of the resulting sum will either be equal to the number of digits of the larger of the 2 numbers, or this size + 1 if there is a carry on the most significant sum. We handle needing to grow the result in the last step of the algorithm. For example if this large integer has the value 4537 (4 digits) and the other has the value of 23 (2 digits), the result will fit in 4 digits, which is what `maxDigits()` should return. Only if we have a carry would we need an extra digit. For example if this is 4537 and the other is 7242 the result of adding them together would be 11779, which needs 5 digits. But as mentioned, we will grow to accommodate a final carry in the last step of the algorithm.
2. Perform the addition, from least significant digit to most significant digit, handling carry as needed. Use the `digitAtPosition()` member function here, as this will determine if each number has a digit, or will return a 0 if you are beyond the size of each number. The resulting digits should be stored in the new array you allocated in step 1. Also make sure you keep track of the carry, and at the end, you should know if a 0 or 1 was carried from the addition of the last most significant digits.
3. Dynamically allocate a new `LargeInteger()` object, using the constructor you created that takes an array as its parameter. You will pass in the array of new digits you summed up in step 2 and its size that you determined and dynamically allocated in step 1.
4. If there was a carry on the last most significant digit, use the `appendDigit()` member function to add the carry value as the new most significant digit to the new `LargeInteger()` you created in step 3. Since your `appendDigit()` function should ignore a request to append a '0' digit, you can always just append the final carry without checking if it is 0 or 1, your `appendDigit()` should grow and append if needed, or ignore if the carry was a 0.
5. The array you dynamically allocated is no longer needed, it was copied when you created your new result instance. So free up the array so we don't leak memory.
6. Finally the `add()` function should return a reference to the new `LargeInteger()` that contains the calculated sum you just computed and assigned to it.

Once you implement the `add()` operator as described, it should be able to pass all of the given unit tests for this task 6. As usual when you are satisfied, create a commit and push it to your `Feedback` pull request. This is the final task, so hopefully if you check the GitHub auto grading actions, you should now be passing all tests and getting 100/100 from the auto grader.

Assignment Submission

For this class, the submission process is to correctly create pull request(s) with changes committed and pushed to your copied repository for grading and evaluation. For the assignments, you may not be able to complete all tasks and have all of the tests successfully finishing. This is OK. However, you should endeavor to have as many of the tasks completed before the deadline for the assignment as possible. Also, try and make sure that you only push commits that are building and able to run the tests. You may loose points for pushing a broken build, especially if the last build you submit is not properly compiling and running the tests.

In this problem, up to 40 points will be given for having at least 1 commit that compiles and runs the tests (and at least some attempt was made to work on the first task). Thereafter 10 points are awarded for completing each of the 6 tasks. However you should note that the auto grader awards either all point for passing all tests, or no points if any test is failing for one of the tasks. Also note that even if you pass all tests, when the instructor evaluates your assignment, they may remove points if you don't follow the requirements for implementing the code (e.g. must reuse functions here as described, need to correctly declare parameters or member functions as `const` where needed, must have function documentation correct). You may also loose points for style issues. The instructor may give back comments in pull requests and/or create new issues for you if you have issues such as these, so it is good to have work committed early before the due date, so that the instructor may give feedback requesting you to fix issues with your current submission.

Program Style

At some point you will be required to follow class style and formatting guidelines. The VSCode environment has been set up to try and format your code for some of these guidelines automatically to conform to class style requirements. But not all style issues can be enforced by the IDE/Editor. The instructor may give you feedback in your pull comments and/or create issues for you for the assignment that you need to address and fix. You should address those if asked, and push a new commit that fixes the issue (or ask for clarification if you don't understand the request). In general the following style/formatting issues will be required for programs for this class:

1. All programs must be properly indented. All indentation must be consistent and lined up correctly. Class style requires 2 spaces with no embedded tabs for all code indentation levels. The editor style checker should properly indent your code when you save it, but if not you may need to check or correct this if code is misaligned or not properly indented.
2. Variable and function names must use `camelCaseNameingNotation`. All variable and function names must begin with a lowercase letter. Do not use underscores between words in the variable or function name. Often function names will be given to you, but you will need to create variables, and maybe some functions, that conform to the naming conventions.
    - Global constants should be used instead of magic numbers. Global constants are identified using `ALL_CAPS_UNDERLINE_NAMING`.
    - User defined types, such as classes, structures and enumerated types should use camel case notation, but should begin with an initial upper case letter, thus `MyUserDefinedClass`.
3. You are required to use meaningful variable and function names. Choosing good names for code items is an important skill. The code examples and starting code tries to give examples of good and meaningful names. In general, do not use abbreviations. Single variable names should be avoided, except maybe for generic loop index variables `i`, `j`, etc. Make your code readable, think of it as writing a document to communicate with other developers (and with your instructor who will be evaluating your code).
4. There are certain white space requirements. In general there should usually never be more than 1 blank line in a row in your code. Likewise there should usually not be more than 1 blank space on a line. There should be 1 blank space before and after all binary operators like `+`, `*`, `=`, `or`.
5. Function documentation is required for all regular functions and all class member functions. You need to follow the correctly formatted Doxygen function documentation format. We will use function documentation generation, and you should be sure your documentation can be built without emitting warnings or errors. Likewise all files should have a file header documentation at the top. You should edit the file header of files where you add in new code (not simply uncommenting existing code). Make sure the information has your correct name, dates, and other information.
6. Practice using proper Git commit messages. You should refer to issues and tasks correctly in commit messages.

Additional Information

The following are suggested online materials you may use to help you understand the tools and topics we have introduced in this assignment.

- Classes (I)
- Classes (II)
- C++ Classes and Objects
- Lecture U02-1 C++ Structures
- Lecture U02-2 C++ Classes and Data Abstraction
- Lecture U03-1 C++ Pointer Variables
- Lecture U03-2 Dynamic Memory