Assignment Operator Overloading and Class Templates: Overloading C++ syntatic operators and class templates for user defined data types

COSC 2336: Data Structures and Algorithms

Summer 2021

Objectives

- Gaining more experience with the concept of an Abstract Data Type (ADT) and how user defined data types are like adding a new type to the language.
- More practice with classes and arrays in C++.
- Practice writing member functions of C++ classes.
- Introduction to operator overloading, and extending the idea of a user defined type as a new type by adding basic support in the language for syntatic operators on the new type.
- More practice with dynamic memory management in order to support list types that can grow and shrink dynamically.
- Experience working with and defining class templates, which are important mechanisms in C++ and other languages for writing generic container data types.

Description

We have two related but separate goals in this assignment. First of all in this assignment we will be emphasizing the idea that defining a class is like adding a new user defined data type to the C++ language. One way this can be increadibly powerful is when we define operator so that user defined classes can be used in similar ways as built in data types. This gives ways of much more naturally and expressively being able to work with classes you define to use in your code, as hopefully you will better understand by doing this assignment. We have actually already seen some examples of operator overloading in previous assignments, before we talked about it in this weeks unit. For example, you have been using the overloaded `operator[]` a lot which defined an indexing operator into our user defined `List` data type. This week you will continue adding onto the `List` data type some other operators that will allow us to append and prepend items on lists, and concatentate lists together to make other lists.

In addition another goal of this assignment is to get some experience with defining class templates. Class templates are used extensively by C++ (see the Standard Template Library or STL) to provide generic container data types. For example, in previous assignments you have been given a `List` user defined class that initially held lists of `int` integer values. This was kind of boring, so we switched it up a bit and defined basically the same `List` but one that held and managed lists of `string` values. It is very common to want to have powerful data types that can hold and manage different types of values, floats, ints, strings, even other user defined data types. So far, the only solution you would have if you wanted to have our `List` that could sometime manage lists of `int` values, and have some others to manage lists of `string` values, would be to define two mostly identical classes with different names, say `ListInt` and `ListString` to hold lists of integers and lists of strings respectively. Class templates allow us to parameterize the type of data being held in a container, so that we can define the operations of the container separate from the type of data we want to be able to put into and manage with the container.

So in this assignment, you will start again with the `List` of strings container, and add some additional overloaded operators to it. Then we will attempt to 'templatize' your `List` class, so that you can use it to hold and manage lists of any desired data type.

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/assg-tests.cpp`	Unit tests for the overloaded operators you are to implement.
`include/List.hpp`	Header file for the declarations of the `List` class you will be modifying and its defined API.
`src/List.cpp`	Implementation file for the `List` member functions that implement the API and class functionality.

This week you will be adding in several new member functions to the `List` class. So all of your work will be to add in code into the `List.[hpp|cpp]` header and implementation files.

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 Recursion' for our current class semester and section.
2. Clone the repository using the SSH url to your local class DevBox development environment.
3. Checkout the 'origins/feedback' branch to your locak working DevBox repository.
4. Configure the project by running the `configure` script from a terminal.
5. 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.
6. 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. Also you should close the initial Pull request that should be automatically opened for you, so that you can open your own when commiting your work.

Assignment Tasks

Notice before beginning the assignment that there are 3 overloaded operators already defined for the `List`: `operator[]()`, `operator==()` and `operator<<()`. You used these in previous assignments, expecially the indexing `operator[]`, though we mostly took them for granted as working at the time. Notice how they work. For example, the `operator==()` allows us to test two `List` items to see if they contain all of the exact same values:

```
int values2[] = {1, 3, -2, -4, 7};
List l2(5, values2);

int values3[] = {1, 3, -2, -4, -7};
List l3(5, values3);

l2 == l3;
```

This should return `false` because while the lists are both of the same size, and most values are equal, the last value of the two lists differ. But the point is that, given the overloaded boolean comparision operator for our `List` class, we can perform boolean comparisions that yield `true` / `false` results just like the built in data types like `int` or `float`. If you wanted, you could add in other boolean comparision operators for the `List` class, like `<`, `>=`, etc. Question: how would you define `<` for our `List` class? We will not be adding in more boolean operators in this assignment, but doing something different.

Notice also the overloading of the `operator[]` which allows us to index `List` instances as if they were regular C/C++ arrays. This is powerful, as it allows us a way to look into our lists, iterate over them, and even modify our list contents. But also notice, recall that this is powerful because it allows us to add in some bounds checking on index requests, so that we can throw an exception if someone tries and access a value beyond the end of our list, instead of causing memory corruption which is what will happen if you make bounds errors when accessing regular C/C++ arrays.

As was discussed in this weeks unit, the `operator<<` to overload the output stream operator is a bit different from the previous two operators. It is actually a `friend` function rather than a member function. There are technical reasons why this is necessary, which were discussed in some of this weeks learning materials. Basically we need the instance of the class we want to send to send to the output stream on the right hand size of the output stream operator:

```
List l1;

cout << l1;
```

Here the output stream object `cout` is on the left hand side, and the `List` object `l1` is on the right hand side of the `operator<<()` stream operator. Because the `List` needs to be on the right hand side for stream output, it cannot be handled as a regular member overloaded operator, as that requires the object to be on the left hand side of the operator.

Also before starting to implement this weeks assignment, note the new `growListIfNeeded()` private member function. This week we are enhancing the `List` to be able to dynamically grow and shrink, but being able to add and remove items from the list. Because of this requirement, the memory management of our list of `values` is a bit more complicated now. We need to be able to grow the size of the dynamically allocated array of `values` if it is currently full. This is where the new `growListIfNeeded()` function comes in. There is also a new member variable named `allocationSize`.

For this implementation of the `List` user defined type, the `size` of the list will not necessarily be the same as the amount of memory we have allocated to hold values, the `allocationSize`. The memory allocation can be larger than is currently being used. The strategy is that, whenever the list of values becomes full (e.g. `size == allocationSize`), we will grow the list. We do this by doubling the current memory allocation, and copying all of the values from the original array into the newly allocated array. So the `growListIfNeeded()` function will be useful for the methods you will implement where you need to add in a value and the list is currently full.

We double the new allocation of memory by default in this class. This is a common tradeoff between performance and memory size/usage. It is a very expensive operation to have to grow the list size, we have to allocate the new memory, and copy all of the values from the old allocation to the new allocation. This is a $\mathcal{O}(n)$ complexity operation, which will take an amount of time roughly proportional to the current number of items n or `size` that are in the `List`. But adding an item or items to the end of the list can be done in constant $\mathcal{O}(1)$ time whenever the new value or values will fit into the current allocation, we just add them to the end of the `values` array and increase the `List size` by 1. So insertion at the end can usually be done fast, but sometimes when we need to grow the list it will be slow. Doubling the size when we need to grow the list usually ensures that we won't have to grow too soon, and by doubling we ensure we only need to perform the growth $\log_2 n$ times for a final large size of n of values that might be managed by the `List`

Task 1: Implement `append()` and Overload `operator>>()`

As usual make sure that you create Task 1 issue on GitHub and are ready to create your Pull request for this assignment before beginning.

For the first task you will implement the ability to append a new value onto the end of our `List` of values. You will first do this by implementing a regular member function named `append()`. This function takes an `int` value as input, which is the value to append to the end of the list. This function should actually return a reference to a `List` as its result. You will not be creating a new `List` to return, but you will simply be returning a reference to the instance from the member function. The reason for this will become apparent when you implement the overloaded operator. To return a reference to this instance, simply do a

```
    return *this;
```

at the end of your member function. The result of calling this function is that the new value will be appended to the end of the list, and the list may have grown if it was full when the append was attempted.

As usual it is recommended that you start by adding the member function signature to the `List` class header file, and a stub function to the implementation file, uncommenting the second set of unit tests, and ensuring that your project can compile and run the tests before proceeding with the implementation.

It is required that you use the `growListIfNeeded()` private member function to implement your append and prepend operations. If you call this function before attempting to add the item to the list of `values`, it will grow the list if needed, so that you have enough room to append on the new value.

Once the `append()` member function is working, you should then add in an overloaded operator for the `List` class to allow appending of values to the list. The next set of unit tests will test the overloading of the operator for appending values. We chose to reuse the `operator>>()` operator to denote appending values to the list. The `>>` operator normally performs input stream operations in C/C++. There is no real operator that means something like 'append' for the built in basic data types. It is not uncommon when doing operator overloading to do something like this, to repurpose an existing syntatic operation to mean something useful for your own user defined data type. Overloading this operator to mean appending a value allows us to write code like the following:

```
List l1;

l1 >> 5;
```

This will append the new value of 5 onto the end of the initially empty `l1` list. Likewise, because we return a reference to the `List` object from the append operator, it allows us to chain append operations, for example:

```
l1 >> 5 >> 3 >> 7;
```

This will first append 5 to `l1` and it returns `l1` back as its result. Then `l1 >> 3` would be performed which appends 3 to the end now, and so on.

Your `operator>>()` will have the same signature as the `append()` funciton you just completed. And in fact, this method is trivial, as you should simply reuse and call the `append()` implementation you already have to implement this overloaded operator.

Once you are satisfied with your implementation of the overloaded append operator perform the usual actions. Commit your changes and push them to the `feedback` branch of your repository.

Task 2: Implement `prepend()` and Overload `operator<<()`

As usual make sure that you create the Task 2 issue on GitHub and are updating your Pull request with work for this assignment.

Your `List` user defined data type you are creating uses a regular, though dynamically allocated, array as the actual storage of the data values being managed by the `List`. If your append operation is working correctly, it most of the time only takes $\mathcal{O}(1)$ time to append a new item onto the end of the list, though at times when the list needs to grow you ned $\mathcal{O}(n)$ time to allocate the new memory and copy the n values from the old memory to the new memory.

What if we want the ability to be able to push items onto the front of the list? Can we do that with our current data structure? The only way we can do this is to first shift all of the current items up 1 index to make room at index 0 of the array to prepend the new value. This means that the prepend operation will always be at least $\mathcal{O}(n)$ in time, though you also still need to grow the array before shifting if it is full, so that time can double if the array first needs to grow. Keep in mind the performance of append and prepend in our next unit where we talk about Linked Lists, which give a way to allow $\mathcal{O}(1)$ constant time performance to append and prepend items to the list.

Besides the need to first shift all items up 1 index for the prepend to index 0, this task will end up being pretty similar to Task 1. You will first create a `prepend()` member function. It will have the same signature as `append()` and will return the same reference to `this` List instance as its result. Of course after checking if the list needs to grow, you first need to shift all items up by 1 index before putting the new value into index 0 of the `values` array.

Then likewise the `operator<<()` which we have choosen to overload for the prepend operation can be defined, and its implemention is simply to call the `prepend()` function you implemented first.

NOTE: it would probably syntatically make more sense to specify prepend as

```
5 << l1;
```

with the value to prepend on the left hand side of the operator. But in order to do this, we would instead have to create a friend function that expects an `int` value as the first parameter and a `List` as the second, similar to the `operator<<()` friend function. We leave this as an exercise for the interested student.

As usual it is recommended that you first create the signature for the `append()` method and a stub function, uncomment the first test case testing the `append()` and make sure you project compiles and runs the tests before you start implementing. It is also suggested you complete `append()` first, and then work on defining the overloaded `operator<<()`.

Once you are satisfied with your implementation of the append operation for the `List` class, commit your changes and push them to the `feedback` branch of your repository.

Task 3: Implement `concatenate()` and Overload `operator+()`

Make sure that you create the Task 3 issue on GitHub and are updating your Pull request with work for this assignment, and that the previous tasks have appropriate commits to document them before proceeding.

We would like to support the concatenation of two lists together to create a new list. For example, if we have

```
List l1;
l1 >> 1 >> 2 >> 3;

List l2;
l2 >> 4 >> 5 >> 6;

List l3 = l1 + l2;
```

We would like the result to be that `List l3` has the values `1, 2, 3, 4, 5, 6` in it. This is the concatenation of the two lists to create a new third list. Also notice the above implies that `l1` and `l2` should not me modified as a result of this operation, they are still both lists with 3 values after the concatenation. This further implies that a new list needs to be dynamically created and returned in the `operator+()` that holds the result of the concatentation.

There are some subtle difficulties with memory management that we must take into consideration here. We could try using, for example, the copy constructor and the append or prepend methods, but these approaches will end up being more difficult than they might at first seem.

First of all, the above example indicates that your `concatenate()` function should take a `const` reference to another `List`. This would be the right hand side `l2` list that is passed into your `concatenate()` function in the previous example. Also we said that `l1` should not be modified by doing the concatenate, so your `concatenate()` function should be declared as a `const` member function as well. The function should return a reference to a new `List` that will need to be created dynamically.

So the approach you should use for `concatenate` is as follows.

1. Start by allocating a new array of `int` values that is large enough to hold both `this` size plus the right hand side `size` values once you concatenate them together. Then
2. Copy all of the values of `this` from its array of `values` to the beginning of this new array of values.
3. Likewise copy all of the values from the right hand side list after these in the new array.
4. Then it is important to dynamically create a new `List` of values that is initialized with this array of the concatenated values you just created.
5. Once you have dynamically allocated your new `List` you don't need the array from step 1 anymore, so `delete` it.
6. You should return th newly dynamically allocated `List` object as a reference from your function.

For step 5, lets say you created an array of integers named `concatenatedValues` that has is of size `concatenatedSize`. Then you would dynamically allocate a new list using the array based constructor, like

```
List* concatenatedList = new List(concatenatedSize, concatenatedValues);
```

The result as shown here, as always, is a pointer to the newly allocated List that is now on the heap. You can return this dynamically allocated reference by dereferencing the pointer

```
return *concatenatedList;
```

Once you have the `concatenate()` member function working, overloading the `operator+()` for list concatenation should be relatively simple and basically the same as you did in the previous 2 tasks. You should be able to define the overloaded operator and simply call your working `concatenate()` function to actually perform the concatenation of the lists into a new list that you return.

Once you are satisfied with your implementation of the `concatenate()` member function and overloaded operator, commit your work and push this commit to the `feedback` branch of your repository.

Task 4: Templatize `List` Class

With task 4 we will be shifting gears and turn our attention to turning your List class into a template class, so that we can manage lists of types other than `int` types as we have now.

We will be us

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 autograder 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.

- Lecture U07-1 Object Inheritance and Composition
- Lecture U07-2 Overloading
- Lecture U07-3 Templates
- C++ Classes and Objects
- C++ Inheritance
- C++ Overloading
- C++ Templates