Assignment Classes: Classes and User Defined Data Types

COSC 2336: Data Structures and Algorithms

Summer 2021

Objectives

- 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++.
- Review of writing functions and C/C++ expressions.
- Continue practicing test driven development and git project workflow

Description

In this assignment you will implement your own version of a `Set` data type, in the mathematical sense of a set of objects. A `Set` is like a list of items, except all items are unique in the set, we never have duplicates of items in the set. We will implement a simple `Set` data type using a C++ `class`. To keep things relatively simple, your set will only keep track of a set of integer values, and we will declare a maximum number of items that can be in your set so that we can use a statically defined array of integers to represent the items currently in the set (we will look at dynamic memory allocation next week to solve this issue of allowing sets of an arbitrary size to be represented).

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
`assg-tests.cpp`	Unit tests for the Set class you are to implement.
`Set.hpp`	Header file for the declarations of the `Set` class and its members.
`Set.cpp`	Implementation file for the member functions of your `Set` class.

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' 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

Task 1: Create Stubs for Accessor Methods so Unit Tests can be Run

If you haven't already, create the Task 1 issue on GitHub and close the initial Pull request if it is still open.

Because of the dependencies of the member functions, it will be hard to test the accessor methods until we are doing some actual work, like adding and removing items from the set. So for this first task, your goal is to write stub implementations for the following accessor member functions:

- `isEmpty()`, returns a `bool` result, whether the set is empty or not. Initially the set is empty, so you should initially just return `true`.
- `getSetSize()`, returns an `int` result, the current size of the set. Initially the set is empty, so you should just return 0 for the set size.
- `containsItem()`, returns a `bool` result. This function is the only one of these accessor methods that takes an integer parameter. It takes a single `int` as input. The method is supposed to return `true` if the item is in the set and `false` otherwise. Again since the set is initially empty, just return `false` for this method stub.
- `str()`, returns a c++ `string` result. Since the set is initially empty, simply return a string that looks like "[ ]" (note there is a space between the square brackets here), as that is what the first unit test is expecting.

In all cases you are to implement a stub member function that returns a simple hard-coded value that is correct for the initial case where the `Set` is empty. All of these member functions, except for `containsItem()` require no input parameters. `containsItem()` takes an integer parameter as input, the item to test whether it is in the set or not. **NOTE**: also all of these member functions should be declared as `const` member functions. None of these functions will actually modify the `Set`, they simply return information about the set. If you don't know what a `const` member function is, you should review this weeks lecture videos and materials.

I suggest you do each of these four methods one at a time. You should

1. Uncomment the test case, but comment out all of the tests except for the function you are adding, for example `isEmpty()`.
2. Add the function signature/prototype into the class declaration file `include/Set.hpp`. Make sure the function is declared to return the correct type, and take an input parameter in the case of the `containsItem()` member function. Also don't forget that all of these functions should be declared to be `const` member functions.
3. Add in the stub implementation of the member function into the `src/Set.cpp` implementation file. You should hard-code the appropriate return value to pass back a result that will pass the first unit tests, and is appropriate for when the `Set` is empty. Remember that these are member functions of the `Set` class. So in `src/Set.cpp` you need to specify they are members of the `Set` class, like this:

   ```
   bool Set::isEmpty() const
   {
     // implementation goes here.
   }
   ```

   Make sure that you put the member function implementation in the correct location, after the member function documentation describing that function. As with the previous assignment, function documentation has already been written for you for the member functions for this assignment.

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 pass. If they don't, fix them so you can build and run the tests before adding in the next stub member function.

When you are satisfied that your stubs are working and you can compile and pass the first set of tests, you should commit your changes and push them to the repository using the `feedback` branch, and then document with a pull request and merge the changes to the main branch.

Task 2: Implement `addItem()` Member Function

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 will actually implement the `addItem()` member function in this task, and we will fix the implementation stubs for the first 4 accessor methods as well. Uncomment the second set of unit tests in the test file, and add in the declaration and a stub implementation of the `addItem()` function in the `Set.[hpp|cpp]` files. `addItem()` is a void member function, it doesn't return a result. It should take a single `int` parameter as input, which is the item to add to the set. Unlike the previous 4 methods, this method cannot be a `const` member method, because you should be modifying the set when you add a new item into it.

Make sure after adding the stub that your project compiles and runs. The first set of tests should all still pass. The second test case tests should run, but many of them will fail until we provide actual implementations of `addItem()` and the other accessor methods we will use to test that adding an item is working.

You should implement `addItem()` in the following way. The `Set` class is declared to have 2 member variables, the current `setSize` and a static array of `setItem` integer values. `setSize` is initialized to 0 in the constructor for the `Set` class.

To begin with, work on getting `isEmpty()` and `setSize()` accessor methods to work. Whenever you add a new item, the size of the set should increase. So if you add 1 to the current `setSize()`, you can then give correct implementations for the `setSize()` and `isEmpty()` accessor methods. `setSize()` is straight forward, you should return the current `setSize` of this `Set` when asked for it. A set `isEmpty()` when the `setSize` is 0. When `setSize` is greater than 0, the `Set` is no longer empty. Implement this and check that all of the tests for `setSize()` and `isEmpty()` now pass for the second test case.

To implement `containsItem()` you actually have to add the item to your set when `addItem()` is called. You should do this by adding the new item onto the end of the `setItem` array. You need to use the `setSize` member variable as your index. For example, when the set is initially empty, `setSize` will be 0, so if you are adding a new item is should be put into index 0 of the `setItem` array and then `setSize` should be incremented to 1. And then you can do the same thing when adding the second item since `setSize` is now 1 and you want to add the second item into index 1 of the member array.

Once you implement `addItem()` correctly, you should be able to write an implementation of the `containsItem()` function. This function takes an integer value that you need to determine if this value is in the current set or not. You do this by searching the `setItem` array for the value being asked about. So you will need to write a loop to search the member array. If you find the item in your `setItem` array, you should return `true` that the `Set` contains the item. If you search all of the array and don't find the asked for item, you should return `false`.

Finally you should implement your `str()` method. This method should create and return a string representation of the items in the `Set`. Notice from the tests that all items are proceeded and followed by a single space in the expected string. You should use an output string stream `ostringstream` to create and return your `string` item from this function. There were examples of using string streams in this weeks video lectures and materials that you can follow to perform this task.

Once you have implemented `addItem()` and the 4 accessor member functions, your 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` branch, and close out your Task #2 and the pull request you should have created for this task.

Task 3: Make `addItem()` work as a `Set` add item

Make sure that you have created Task 3 and are ready for a new pull request for this set of work.

You may or may not have been aware that we actually have a problem with the implementation of `addItem()` that we didn't mention in Task 2. The items in a `Set` are supposed to be unique. So whenever we add in a new item, we need to ensure we don't add in duplicate items to the `Set`.

Start by uncommenting the tests for the third test case. All of these tests should still compile and run, because thy are not using any member functions that you haven't implemented yet. But make sure that is the case by compiling and running the tests after you uncomment them.

Though they run, they will fail because we will be testing what happens if you add duplicate items to the `Set` which you should not be handling correctly yet.

The implementation of `addItem()` to handle duplicates should be relatively straight forward. However, you should practice code reuse. We already have a member function that should be working that can answer the question of if the `Set` contains an item already or not that we want to add. So you should modify your `addItem()` function to first

check if it already contains the item it is being asked to add. If it doesn't already contain the item, then you should add it as you were doing before. If it already contains the item, then `addItem()` should just silently do nothing, as it already has the item in its set.

Modify your `addItem()` to correctly handle requests to add in duplicate items already contained in the set. Once you have done this, the tests in the third test case testing when duplicates are added 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 `removeItem()` Function

Perform the usual prerequisite steps before starting task 4.

Our sets would not be too useful if all we could do is add items in to them. We want to be able to remove items from the set, as well as perform other operations. The next task is to then add and implement the `removeItem()` member function. This function has the same signature as `addItem()`, it takes an integer parameter as input, which is the item to search for and remove if it is in the set. It is a void function, it does not return an explicit result. This function is also not a `const` member function, for the same reason that `addItem()` is not a const member function. As usual, it is suggested that you add in the function signature to the header file, and a stub implementation first, then uncomment the fourth set of test cases, and make sure that your project can compile and run the tests before moving on to trying to implement this function.

Here are some hints on how to implement `removeItem()`. Like `addItem()` you should first check if the set actually contains the item to be removed, and if it doesn't you should just silently do nothing and ignore the request. But if the set has the item, it has to be removed from the `setItem` member array somehow. Of course it is easy to update the `setSize`, just reduce it by 1. But how do you remove the actual item for the `setItem` array? You could search for the item and replace it with a flag, like -1, to indicate the item is no longer a valid item. But this complicates some of your other accessor methods (you now should probably search for empty locations instead of just always adding new items on the end), and wastes space. Instead you should remove the item by performing shifting. This is a bit complicated. The general idea is, you know the item is in the `setItem` array, because you checked that first. So you first have to search your array to find the index where the item is located that you need to remove. Then once you find it, you will remove it by shifting all items down one space in the array (and don't forget to decrement `setSize` when done).

An alternative method that avoids the need to shift all of the items is to just swap the item with the item at the end of the array. However, the tests you are given assume that the items when displayed by `str()` will always appear in the order they were added, even when you remove things in the middle of the list of the set items. So if you did implement by swapping, you might find that some of the tests will be failing. We leave it as an exercise to implement this more efficient swapping method to remove items, and modify the tests correctly to expect this method, though you should do that only after finishing the required tasks for this assignment.

Once your implementation of `removeItem()` is working, 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 `operatorUnion()` Member Function

Perform the usual prerequisite steps before starting task 5.

Besides adding and removing items from a set, the most common operations we want to perform is to compute the union and intersection between two (or more) sets. The usual way we would define this would be two specify two sets as operands, and compute the operation (union or intersection), and return the result in a new third set. We will look at ways to perform this more fully in future classes where we cover operator overloading and dynamic memory allocation and memory management.

For now, we will implement union and intersection by taking another set as the input parameter to a given set. And we will compute the result by modifying the `Set` that the operation is called on to hold the result of the union or intersection.

As usual, start by uncommenting the test case for the `operatorUnion()` function. There is also a set of test fixtures in the test file right before the tests of `operatorUnion()`. You should uncomment the test fixture as well. Test fixtures are common items for performing unit testing. The fixture in this case creates a couple of different sets that can be used to test the union and intersection operations. Also there are actually 5 test cases of the `operatorUnion()` function. So you should uncomment and test all 5 of these test cases for task 5.

You should add in the member function prototype and a stub function implementation for `operatorUnion()`, and make sure your code still compiles and runs the tests. Int his case, `operatorUnion()` is a `void` function, it doesn't return a result. The input parameter for the union and intersection is interesting. For both, you want to take another `Set` as input for the operation, which is a perfectly valid and legal thing to do. Note however, that you will not be modifying this `Set` that is given as input to the `operatorUnion()` function. So this parameter should be declared as a `const` parameter for this function. Be aware that `const` parameter is different from being a `const` member function. If you are not sure about the difference, you should review our course materials for this unit and find out.

The following are some hints and discussion of how you should implement the `operatorUnion()` function. As we already stated, the union function takes `otherSet` as an input parameter. The union of two sets is the set of all values that appear in either of the two sets. So a very simple approach, that reuses your existing member functions, is the following. You can iterate over all of the items in the `otherSet` and add each of these items to `this` set that the unit was called on. You should reuse your `addItem()` member function here because recall it will ignore duplicates if you ask it to add them and will end up only adding items that are in the `otherSet` that `this` set doesn't already have. NOTE even though the `setSize` and `setItem` member variables are private, you can actually access them directly in a member function, even if they belong to `otherSet`, thus it is easy to iterate over the item in the `otherSet` by accessing its `setItem` array.

Once you implement the `operatorUnion()` 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` branch, and then close and merge your pull request and issue associated with Task 5.

Task 6: Implement `operatorIntersect()` Member Function

Perform the usual prerequisite steps before starting task 6.

You should have uncommented the test fixture needed for implementing the `operatorIntersect()` member function already. There are again 5 test cases for this task, uncomment them all. And then add in the member function prototype and a stub member function implementation, and make sure you can compile and run the tests before proceeding.

The signature of the `operatorIntersect()` member function is the same as for the union operator. It is a void function, that takes a `const Set` as its input parameter.

The set intersection operations is conceptually similar to the union, but a bit trickier. The intersection of two sets are all items that appear in BOTH sets. So to implement the intersection, one approach is to iterate through `this` set this time. Any item that you find in `this` set that is not also in the `otherSet` you need to remove from `this` set. However as a warning you need to be careful when doing this. You should reuse the `removeItem()` member function to remove items. But if you are iterating through your items, calling `removeItem()` will actually shift the items you are iterating over, and you can easily miss or skip items. It is possible to do it correctly if you iterating over the items in the usual forward direction, from index 0 to the end of the array. However, as a hint, consider instead iterating from the last item up to item 0. The reason this is easier is because, when you call `removeItem()` it will only shift items at the end that you already checked. So there is not a lot of special checking you need to do if you iterate in reverse and use `removeItem()` to remove and shift from the parts you already iterated over.

Once you implement the `operatorIntersect()` 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` branch, and then close and merge your pull request and issue associated with Task 6.

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.

- Classes (I)

- Classes (II)
- C++ Classes and Objects
- Lecture U02-1 C++ Structures
- Lecture U02-2 C++ Classes and Data Abstraction