Assignment Classes and User Defined Data Types

COSC 2336: Data Structures and Algorithms

Spring 2024

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.
- Review of writing functions and C/C++ expressions.
- Understand basics of defining a new data type (`Set`) and adding it to the C++ language as a type that can be declared and used to hold information.

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
`src/assg02-tests.cpp`	Unit tests for the Set class you are to implement.
`include/Set.hpp`	Header file for the declarations of the `Set` class and its member functions and variables.
`src/Set.cpp`	Implementation file for the member functions of your `Set` class.

As usual, before starting on the assignment tasks proper, you will need to finish the following setup steps.

1. Accept the assignment invitation and create your assignment repository on GitHub.
2. Clone the repository to your DevBox using the SSH URL. Make sure that you open the cloned folder and restart inside of a linux Dev Container.
3. Confirm that the project builds and runs, though no tests may be defined or run initially. If the project does not build on the first checkout, please inform the instructor.
4. 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 them from the issue templates for the assignment. Also make sure you link the issue(s) with the `Feedback` pull request.

Assignment Tasks

Task 1: Add initial stub functions

We will start by practicing a type of incremental test driven development in this assignment. We ultimately want to implement a function to be able to add new items to our set. To do this, we would like to have some other member functions that we can use to get information about the set, so that we can test the `Set` properly. We will start by adding some "stub" or empty functions that don't yet work completely. We will implement these stub functions more fully as we continue working on implementing the full functionality of the `Set` data type.

Start by defining the first test `task1_1` in the `assg02-tests.cpp` file. This test tries to call an `isEmpty()` member function of a `Set` instance. Initially when a new `Set` is created, it will be empty. So we can create a "stub" `isEmpty()` member function that simply returns `true`, since it will be true that sets are initially empty. Define `task1_1`, and create a stub `isEmpty()` function that simply returns `true`. To create the function, you need to declare it in the `Set.hpp` header file. The `isEmpty()` function should be a `public` function that returns a `bool` result, and takes no parameters as input. Also, the `isEmpty()` member function should be a `const` member function, because this function only returns information about the `Set`, it does not change the set. You declare a member function to be a `const` member function by putting the `const` keyword at the end of the function declaration. Once you declare your function in `Set.hpp`, you also need to add an implementation of the function into the `Set.cpp` implementation file. Make sure that you indicate this function is a member of the `Set` class by prepending the name of member functions with `Set::`. You should initially just return `true` as the implementation of this function.

Once you get the `isEmpty()` function added and the test passing, do the same to create stub functions for the `getSetSize()`, `containsItem()` and `str()` member functions. Define each subtask, add the stub function, and check that the code compiles and the test case passes.

All of these functions are returning information about the `Set`. The `getSetSize()` returns the number of items currently in the set. So it should return an `int` result, and should initially be stubbed out to return 0 since sets will start out empty and thus will have a size of 0. This function is a `const` member function, so you need to declare it as such when you implement it.

The `containsItem()` member function will return a boolean result of `true` if the item it is asked about is currently in the set, and `false` if it is not. Thus `containsItem()` returns a `bool` result. This function, however, has an input parameter, an `int` value which is the item that we should check to see if it is in the set or not. Like before, this method is an informational method, so it should be declared as a `const` member function.

Finally the `str()` method will return a `string` representation of the current items in the set, so this method should return a value of type `string`. This method takes no input parameters, but it is again a `const` member function. This method should initially return a string with the value "[ ]" (notice the space between the two square brackets). Later on the items that are currently in the set will need to be put into the string that is returned from this method.

For practice you might want to make a commit once you have each individual stub function implemented and passing its small unit tests. At a minimum, you need to make at least 1 commit before moving on to the next task that has all 4 of these informational stub functions implemented and their tests passing.

Task 2: Implement `addItem()` Member Function

Now the real work begins. If you completed the first task, you have functions named `isEmpty()`, `getSetSize()`, `containsItem()` and `str()` that return default values. Now we will implement a function that will need you to do some real work to implement it and the previous stubbed out functions.

Define the task 2 in the assg02 tests file. This test case tests the `addItem()` member function. This is a `void` function, it does not return a result. But `addItem()` does take an input parameter, an `int` which is the item to be added to the set. Unlike the previous methods, the `addItem()` function will actually be modifying the set, so it should not be a `const` member function this time.

Write a stub function that does nothing, and make sure your code still compiles. If your code compiles it should still be passing the previous tests, but it will fail now on the first test of `isEmpty()` of the most recent test cases since we no longer expect the set to be empty after adding in our first item.

You need to add code into `addItem()` first. For example, you could simply increment the `setSize` member variable by 1 each time an item is added to the set. This would allow you to correctly implement the `isEmpty()` function now

because you can return `true` from `isEmpty()` when the `setSize` is 0, and you can return `false` when the `setSize` is a non zero value. Do this to get the first test in this case of your `isEmpty()` to correctly pass.

By incrementing the `setSize` member variable, you should be able to correctly implement `isEmpty()` but also can easily correctly implement `getSetSize()` now as well, by simply returning the current `setSize` of the set.

However, getting `containsItem()` to work will require some more implementation. For `containsItem()` it is not enough to simply know how many items are supposed to be in the set, we will have to keep track of and search the actual items in the set to determine if the requested item is in the set or not. So you should start by saving the item in your `addItem()` method to index 0 of the `setItem` member array. You can think ahead about how to correctly add the next item to index 1 of the array, e.g. you could use the `setSize` variable which will allow you to know which index should be used to store new items that are added. Once you store the item in the array somehow, work on your `containsItem()` function. Your implementation should simply search through your `setItem` array member variable, and return true if you find the item is currently in your set, and false if not. Implement your `containsItem()` so that it passes the unit test here. It should also still correctly pass the next test that if we ask if the set contains the item 0, which it does not, the answer is `false`.

Finally, if you got to this point, you have now implemented `addItem()` and also fixed your stubs of `isEmpty()`, `getSetSize()` and `containsItem()` to all be working correctly. The only remaining work is to fully implement the `str()` member function. You should use a `ostringstream` string stream object to construct a string stream as expected to represent the items currently in the set. Watch the lecture videos from this week where we cover using string streams if you have not used them before.

At this point you should be passing the tests in the first section for the `addItem()` member function. Check that all of the tests in the next section pass as well after adding 3 more items, and fix any issues/bugs if these tests are not passing before moving on.

Once you are satisfied that your `addItem()` and all the implemented accessor methods are working, you should commit and push your changes to the `Feedback` pull request of your repository. Remember that you are required to have at least 1 commit for each task, so you need to push a commit here of your implementation of `addItem()` that works when inserting non-duplicate items. As usual, make sure that you check the autograder on GitHub for this commit.

Task 3: Fix `addItem()` to Handle Duplicate Items

By definition a set contains a unique collection of the items, there should be no duplications of items in the sset. However, if you implemented your `addItem()` function as suggested, it will not behave as a proper `Set`. If you add a duplicate item and then look at your set (for example using the `str()` method), you will see that the duplicate item occurs multiple times, and that the set size counts the duplicates.

In this task we need to modify/fix this bug so that the `Set` is properly maintained to contain only a unique collection of the items. Define the task 3 tests. This test case is still testing the `addItem()` member function, so you don't need to add a new function for task 3. But here we test that if you add a duplicate item to the set, nothing happens and only 1 unique value of each integer item is kept in the set.

You should implement this behavior using the following algorithm. You should reuse your now working `containsItem()` member function. Before you actually add the item into your `setItem` array, you will first check whether or not the set already contains the item by calling `containsItem()`. If you already have the item, you can just return immediately and do nothing (i.e. silently ignore the request to add a duplicate item). Only if the item is not already in the set should you then proceed to add it to the end of the array of your set items. If you implement this correctly, all of the tests of adding duplicate items should now be passing in this test case.

Notice that you are required to reuse the `containsItem()` member method here. You could write a loop again to check if the item is already in the set or not, but this would be repeating the functionality of the `containsItem()` member function. This is a fundamental principle of programming, Don't Repeat Yourself (DRY). You write functions or member functions to encapsulate a piece of functionality, and you should always reuse this functionality instead of writing another implementation of it in your code.

Once you are satisfied that your new implementation of `addItem()` correctly handles insertions of duplicate items, commit you changes and push them to the `Feedback` pull request of your class repository before continuing to the next task.

Task 4: Implement `removeItem()` Member Function

In the remaining tasks, we will add more common functionality to our `Set` data type. To be useful, we need to be able to also remove items from our set.

Define the task 4 to test the `removeItem()` member function. As usual first create a stub function that does nothing, and make sure your code still compiles and still passes all of the unit tests up to this point. The `removeItem()` member function is a `void` member function, but it takes an `int` parameter as input, which is the item to search for and remove from the set.

The `removeItem()` should simply silently do nothing if asked to remove an item that is not in the set. You can do this again by first checking using your `containsItem()` member function, and just returning if the set doesn't contain the item you were asked to remove.

If the item is in the set, you need to search for and remove the item from your array of set items. You need to do this using the following algorithm. First find the location of the item in the array. Then you should shift down all of the items at higher indexes by 1 value. This effectively removes the item, and ensures there are no holes left in your array of set items. Make sure you also correctly update your `setSize` count by decreasing it by 1. The tests in this unit test check that you correctly handle cases like when you try and remove the last item of the set of items, or when you try to remove the item at index 0 in the array, etc. If you correctly shift the items and update the `setSize` you should be able to pass all of the unit tests of `removeItem()` here.

Once you complete this task, commit your changes that implement the `removeItem()` member function and push them to the `Feedback` pull request of your repository.

Task 5: Implement `operatorUnion()` Member Function

Set union and intersection are fundamental operations that most users of your `Set` will expect you to support. The union of two sets is a new set that contains all of the items either in set 1 or set 2 (or in both of them). We don't quite have all of the tools yet to create and return a new `Set` dynamically, so we will implement the union of two sets by passing a `Set` into this member function and updating this `Set` to add in any items from the other set that don't already exist in this set. The result is that this `Set` (that we call `operatorUnion()` on) will actually be modified to be the union of the original 2 sets.

Define the task 5 tests in order to develop and test the `operatorUnion()` member function. As usual you should create a stub function and make sure you code compiles, runs and passes all of the previous tests. The union operation will take a `otherSet` as a parameter and the result will be the union of the two sets that ends up in the set you call the `operatorUnion()` member function on.

The algorithm for the `operatorUnion()` is relatively simple. You should iterate over all of the items in the `otherSet` and call your own `addItem()` to add each item of the `otherSet`. If your `addItem()` is implemented correctly, it will do nothing if the item is already in the set, but it will add the item that it doesn't have from the other set if it is missing. Note that you are required to reuse `addItem()` here in your implementation of `operatorUnion()`. The result of doing this is the union of the two sets.

Note it is required that the `otherSet` be passed in as a reference parameter. This is done for performance reasons. But since the `otherSet` should not be modified when the union operation is performed, it should be passed in as a `const` parameter.

Once you are satisfied with your work, add and commit your work on the `operatorUnion()` function and push them to the `Feedback` pull request of your repository.

Task 6: Implement `operatorIntersect()` Member Function

To complete our `Set` data type we also need an intersection operation. The intersection of two sets is defined as only those items that appear in both sets.

Define the task 6 tests to enable the unit tests for the `operatorIntersect()` member function. Again start by creating a stub function and making sure your code still compiles, runs and is still passing all of the tests up to this point.

The intersection operation is a bit more tricky than the union operation. To implement intersection you will have to remove any item in the set that is not also in the other set. There are many different ways you could accomplish

this. My recommendation is that you implement this in the following way. Iterate **BACKWARDS** through your items in THIS `setItem` array, e.g. start with the last item and work backwards to the item at index 0. For each item test if it is in the `otherSet` by calling the `containsItem()` member function on the other set. If the item is not also in the other set, you need to remove it from your set. Do this by calling your own `removeItem()` member function to remove the item. The reason why you work backwards through your items is because, if you do so, you will not get messed up by by your `removeItem()` implementation shifting items in the array. Only items you have already checked will get shifted down.

As usual, when finished with this task, add and commit your changes and push them to the `Feedback` pull request of your class repository. This is the final task of this assignment.

If you have gotten all of the tasks completed to this point, you should be able to run and pass all of the original tests once your intersect and union operators are implemented. When satisfied you should commit and push your changes. Check your final GitHub actions and autograder at this point. If your work is committed correctly, you should see you now get a green check mark on all tests and the autograder gives a score of 100/100.

Assignment Submission

For this class, the submission process is to correctly create a 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 50 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 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). 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.

- C/C++ Functions
- C/C++ Arrays and Arrays as Function Parameters
- cmath Library Reference
- Lecture U01-1 User Defined Functions
- Lecture U01-2 User Defined Data Types
- Lecture U01-3 C++ Arrays