CS 15: Unit Testing, ArrayLists Lab

Introduction

Welcome to CS 15! The purpose of this lab is twofold:

- To familiarize you with the concept of unit testing and `unit_test`, a unit testing framework which will help you easily test your code *as you write it*.
- To write a simplified `ArrayList` class and test it with the `unit_test` framework.

Note: If you are unfamiliar with accessing a terminal, using `ssh`, etc., please read the `System Setup` guide on the course website now.

Unit Testing

Testing code is important! We are strict and serious about testing because it is an essential tool for a computing professional. Writing tests while you write your code will save you time by helping you catch bugs early — when they are often easier to figure out and fix. The general rule is: as soon as you write a function, write some basic tests for it.[1]

What Makes a Good Test

Good tests are small functions which test **precise** areas of your code. They have easily understandable function names and often use helpful print state-

[1] There is even a software engineering paradigm where you write the tests *before* you write the code it tests! If you're curious, it's called test-driven devleopment (TDD)

ments. Small test functions that test a single specific thing are useful because if something goes wrong you know where in the code to begin looking.

For example, if a test function named `removeFromEmptyArray` fails, you know immediately that, when you call the remove function on an empty array, something goes wrong. You can comb through your remove function to find the bug.

Often, such tests need not do too much—a few lines of code is great. The goal is to be clear on specifically what you're trying to test.

At first glance, each of the functions you'll write can be tested "at face value"—i.e., if you're writing a function to insert elements into an `ArrayList`, you'll want to write a function that ensures that elements are inserted as you expect them to be. But that is just where testing begins!

Edge Cases

What happens, for instance, when you try to insert an element into an `ArrayList` when the underlying Array is already at maximum capacity?

For many of the functions you will write in this class, there will be cases like this one which are very important—while they don't necessarily occur all the time, if they're not handled properly, they will wreak havoc on your underlying implementation.

In the 'biz, these kinds of cases are commonly called "edge cases" or "corner cases". Indeed, edge cases are not just something you'll face in this class, but are very common in industry - and are painful for everyone. Learning to write code that is resilient to edge cases will make your work more reliable, resilient, and robust, not only for yourself, but for the other programmers who will use/work with your code, and for the end users as well!

Therefore, our autograders will be testing for such situations whenever possible. Common edge-cases situations include:

- Empty containers, especially removing from them
- Full containers, especially inserting into them
- Off-by-one errors when iterating through a collection
- Memory leaks/errors (and we'll see more about these during the term)
- Searching for non-existent things
- Out of range accesses

- Invalid input (either accidental or malicious) from the user

But there are certainly many more...

Coming Up With Tests

While you work on various functions for the assignments in this class, questions or ideas might arise in your mind. One way to work is to jot these down in comments (which you will remove later)—they can often become useful tests.

For example, if you are unsure whether your for loop should have `i < array_length` or `i <= array_length`, write that down and make a test for it! **Trust your instincts: if you are not sure about some code, write extra tests to prove it to yourself.** And even if you are sure about some code, write some tests to prove it to yourself anyway! You'll be surprised how many bugs you catch this way. Writing a few extra tests early on is relatively easy in terms of time, effort, and overall pain; debugging off-by-one errors when you're segfaulting "for no reason", on the other hand, is a nightmare.

Regarding edge-case testing, a good way to catch these is to imagine what might cause the program to segfault or throw an exception. In other words, how would you crash your own program?

For this lab, you may not be able to test all of the cases mentioned above and that is fine. For now, focus on writing short test functions that target specific aspects of your code.

Testing Logistics

Testing code is difficult! Not only is it challenging to come up with tests, but keeping things organized so your testing process and development workflow is as smooth as possible presents its own challenge as well.

There are many strategies you might use to approach testing. Although we are going to be using our `unit_test` framework for this lab, and encourage you to use it for the future, here are some of the methods you might consider on your own—along with some of their pros and cons.

1. You could have a single very long `main` function that interacts with and tests your program. This is often difficult to read, particularly when you have a lot of tests. However, it is relatively easy to compile and run your work. That said, the output of the testing program is

very likely going to be difficult to read and managing all of your tests as you run them together becomes a bit of a headache.

2. You could have one `main` function, which calls other individual test functions. This is certainly better than the previous option, as you can manage the complexity of a single main function more easily, and don't have to comment out large blocks of code. However, you'll still need to remember to run valgrind manually to check for memory leaks—a common problem is that students forget to do this often, and then miss critical bugs until they're nearly ready to submit! And, again, your testing output is likely going to be difficult to parse.

3. You could have many different testing `main.cpp` files (i.e. `main00.cpp`, `main01.cpp`, etc.). This is nice in that each test is individual, and isolated from all others (in fact, our autograder works this way!). Then you'd use some kind of shell script to run them in sequence. This shell script could also run valgrind on your tests automatically to check for memory leaks / errors. Downsides of this approach are that jumping between many testing files will be quite annoying, each test will need to be compiled separately from the others, which takes time, and, again, your testing output will be difficult to parse.

4. You can use our `unit_test` framework! This framework attempts to ameliorate the various problems listed above, but will require you to organize your work a little differently than how you've been used to. Read on below for details!

unit_test

What is unit_test?

`unit_test` is a unit testing framework that we've created to make your development workflow as smooth as possible. More precisely, it is a script (note: it is **not** only a `C++` program) which will:

1. Create a testing driver program (which contains `main()`).

2. Run each of your tests individually (one failed test won't stop the others from running).

3. Run valgrind on your tests.

4. Output the results of your tests, including error messages, number of passing/failing tests, etc..

In order to use the framework, your testing code will need to be organized in a way that the framework can "understand." This will take a bit of getting used to (for instance, you won't have a `main()` function when using `unit_test`!), but will pay off in the long run. Specifically, in order to run `unit_test`, you will have to:

1. Write your tests.
2. Edit your Makefile (we do this for you early on in the semester).
3. Run `unit_test`.

See below for details.

Organizing your tests

The way `unit_test` works, you are required to put your tests into a **single testing file** named `unit_tests.h`. Each test is a function which must:

- have return type `void`
- take no arguments
- have a unique function name

For example, a valid test signature might be:

```
void my_first_test()
```

`unit_test` will find and run each function that matches the above parameters, and each test that finishes execution will be considered successful—see the section "What makes a successful test" for details.

Note! The way that `unit_test` works is each test function will be run as its own process. This way, one failed test will not stop the others from running. The details of how this happens are beyond the scope of this document, so, for now, just think of each test in your `unit_tests.h` file as being its own `main()`. Indeed, the `unit_test` script has a file which it builds for you which has `main()` in it, which is run one time for each of your testing functions. Thus, **your testing code cannot have a main()!** Again, thinking of each testing function as its own `main` is the right way to approach the situation.

Editing the Makefile

In order to use `unit_test`, your `Makefile` must have a target named `unit_test` which links your code with a file named `unit_test_driver.o`. This driver file will be built for you by the framework; however, you must compile your `.cpp` files into `.o` files. Early on in the semester, we will be supplying you with a `Makefile` which has the appropriate targets; later on you will be responsible for writing them yourself.

Running unit_test

Great! You're ready to go. Once you have your `unit_tests.h` file and `Makefile` in place, you just need to run the command `unit_test` from terminal. When you run this command, your code is compiled, the tests are run, and the results are reported to you.

What makes a successful test?

Great question! **A test is considered successful by `unit_test` if it finishes execution.** Consider writing a test for a constructor of an `ArrayList` class. Here's a first draft:

```
void test_constructor() {
    ArrayList my_list;
}
```

While this may seem quite simple, this is actually a great first test of the default constructor. If this test is successful, then you know that your code does not crash when initializing an `ArrayList` with a default constructor, and you'll also be sure that your code doesn't have a memory leak when the list's destructor is called as the list goes out of scope. This is a lot of information for a one-line test! That said, what else might we do? Here's another test:

```
void test_constructor() {
    ArrayList my_list;
    assert(my_list.size() == 0);
}
```

Notice the use of the `assert` function. This function asserts that the boolean expression inside is `true`. If it is `true`, the program continues; if it is `false`, then the program crashes immediately. Recall that a test is considered successful by the `unit_test` program if it finishes completion. Thus, **the `assert` function will be a very useful tool in your unit testing.** (Note that to use this function, you'll need to add `#include <cassert>` at the top of your testing file.)

Clearly, this second test is a bit more thorough—it ensures that the size of the list is both correctly set and reported. Technically, if it fails, you might have a bug in either the constructor or the `size` function, but it certainly alerts you to an issue with relatively narrow scope.

Test overlap

Given this style of testing, often it can be difficult to write tests that **only** test one function. For instance, how can you test the `pushAtBack` function of the `ArrayList` class without also testing the `toString` function? In these cases, just do the best you can to make the tests as specific as possible. Some overlap is perfectly fine. The key idea here is that tests which are as precise as possible will help you debug problems before they get out of hand and cause you major headaches down the road.

Initial setup to run unit_test on the Halligan server

The `unit_test` program is located at `comp/15/bin/unit_test`. In order to run it, please perform the following steps:

- `ssh` to the homework server.
- run the command (include the quotes!):

```
echo "use -q comp15\n" >> ~/.cshrc
```

- Now, either log in again, or run the command:

```
source ~/.cshrc
```

The `use -q comp15` command will now be run automatically every time you log in to the homework server. Among other things, this puts `/comp/15/bin` into your `PATH`, which means you can run the `unit_test` command without specifying a directory.

The Lab

Getting Started

You should have a `cs15` folder. `cd` into it, make a `lab1` directory (`mkdir lab1`), and run the following command:

```
cp /comp/15/files/lab_arraylists/* .
```

This will copy the starter code for the lab. To help you with the lab, we have provided you with all of the necessary files to make `unit_test` work:

- A testing file named `unit_tests.h`.
- The beginnings of an `ArrayList` class.
- A `Makefile` which is used by `unit_test`.
  - We won't discuss the `Makefile` in detail for this lab—that discussion is for another day. You should know, however, that you **do not need to run `make` with `unit_test`. `unit_test` runs `make` for you!** So, just run `unit_test`.

Introduction

Start by familiarizing yourself with the `ArrayList.h` file. You will see a very simple interface for an `ArrayList` class with a few functions defined for you. Notice also that the data type that the `ArrayList` holds is an integer.

Next, open the `ArrayList.cpp` file. Inside you will find an incomplete implementation. The functions have been left blank intentionally, except for the `toString()` function, which is only partially implemented. It is your task to complete and test these functions using the given `unit_tests.h` file.

Now, open the `unit_tests.h` file. Here you will find a few tests implemented for you. However, the `ArrayList.cpp` code is not implemented, so at least one of the tests will fail at first!

To-Do Items

Your tasks are:

1. Add any necessary elements to `ArrayList.h` (see the TODO in the private section there).
2. Implement each of the unimplemented functions in `ArrayList.cpp`
3. After you write each function in `ArrayList.cpp`, think of and write at least one test function for it in `unit_tests.h`.
4. Whenever you're ready to run a test, just run the command `unit_test` in your terminal. Your code will be compiled, tests will run, and the output will show you the results.

5. Fill in the missing sections of the `README` provided for you.

Tips

- Recall that an `ArrayList` is a data structure which dynamically resizes itself as elements are inserted/removed. Notice that in `ArrayList.h`, we have **not** given you the data member that is required to ***hint*** point to the actual data members of the list—you must add this yourself. Refer to lecture if you need help with this.
- For this lab, we will have an `ArrayList` which does not shrink—it only expands. This should make your life easier. Remember that the `ArrayList` expands when the `size` is equal to the `capacity` - in that case, you must:
  - Allocate a new array of the current `capacity * 2 + 2`.
  - Copy each element of the old array to the new array.
  - Free the memory of the old array (using `delete []`).
  - Don't forget to update the `capacity` variable, and to update any other private member variables!
- The `toString` function provided has some starter code which uses `std::stringstream`. Don't be afraid of `std::stringstream`s! You can use it just as you would `std::cout`, but then calling `.str()` on a `std::stringstream` object produces a `std::string`. How cool! Again, all of the setup code is there for you, so you can just use the provided `ss` (short for 'stringstream') variable as you would normally use `std::cout`.
- Just because you wrote a lot of tests doesn't mean they're good! Be careful about thinking that passing your own tests will pass the autograder for homeworks. Think hard about good tests to run!

Part 4: Submitting Your Lab

You will need to submit the following files:

```
ArrayList.h
ArrayList.cpp
Makefile
```

```
unit_tests.h
README
```

You must submit them using Gradescope to the assignment `lab_arraylists`. Submit these files directly, do not try to submit them in a folder.

Reference the style guide for what should be included in your README, but keep in mind that lab README's can be brief.

A Note on Lab Grading

Labs are for hands-on practice in a supervised setting so you can develop your skill. They are designed to be low-pressure and fun. Come to lab, do your best, submit your work at the end of the period (your best effort given the time), and you will get a high score. If for some reason you miss your lab session, you can go to another one (provided there is space, see the course syllabus). If you cannot attend any session for a whole week, do it in your own time so you will gain the skills. See the syllabus for course policies on labs.