Contents

Chapter 1

Introduction

1.1 Welcome

Welcome to `unit_test`, and easy-to-use but powerful unit testing framework. The goal of `unit_test` is to allow you to spend as much time as possible working on and testing your code, without having to spend unnecessary overhead on administrative details.

However, as with anything complex, `unit_test` requires a slight bit of acclimation. This document is intended to bring you up to speed so you can start testing and debugging ASAP. But, before diving into testing, it's good to be clear on what the `unit_test` framework is, how it works, how to use it, and what its limitations are.

1.2 What is unit_test?

`unit_test` is a script which runs test functions that you have written in a testing file. It not only runs your tests, it also runs `valgrind` on your tests to check for memory leaks/errors right away, and then reports the results. `unit_test` can also run tests to `diff` your code's output with a reference automatically.

1.2.1 What does the unit_test script do?

The pseudocode for the `unit_test` script is (roughly) as follows:

1. Extract all of the test function names from your test file

2. Create a driver file named `unit_test_driver.cpp` that keeps track of these names. This driver file has `main()` inside of it.
3. Run each test as its own process. If a test finishes execution, it is considered successful. For each test that succeeds:
   - If `diff` tests have been setup for this test, then `diff` the code's `stdout` with the testing output file.
   - run `valgrind` on that test
4. Report the results of the tests.

1.3 Installing unit_test

You can either install the script directly, or use the associated `VSCode` extension.

1.3.1 Using the Script Directly

The `unit_test` script contains the entirety of what is needed to run `unit_test`. It simply needs to be on your `PATH`. For students in `CS15`, please perform the following steps:

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

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

- Either run the following command, or simply login again.

```
source ~/.cshrc
```

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

1.3.2 VSCode Installation

If you're running `VSCode` and would like to use the `unit_test` extension, perform the following steps:

- Open `VSCode` and, via Remote-SSH, connect to the homework server as your remote host.
- Click the 'Extensions' tab, and search for `AutumnMoon.cpp-unit_test`
- You should see an extension. Install it!
- Note that there are one or two differences with using this on VSCode - after reading the sections below on how the framework works, please read the documentation in VSCode as well, which will help you get started there. (All necessary files for the framework, however, are provided for you by the lab.)
- Here are also links to videos created by TA Amy Bui that demonstrate how to use `unit_test` on `VSCode` as well: video 1 link, and video 2 link

Chapter 2

Using unit_test

2.1 Where do I write tests?

All tests are written in a testing file named `unit_tests.h`. This file should be in the same working directory as your code.

2.2 How do I write tests?

`unit_test` runs test functions that you will write in the testing file. In order for `unit_test` to find and run your tests, each test function must:

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

For example, the following is a valid test signature:

```
1
2 void my_first_test()
```

2.3 When is a test successful?

A test is considered successful if:

- it successfully executes without crashing
- it passes `valgrind`

- if a `diff` file has been setup for the test, the test's `stdout` produces an empty `diff` against the `diff` file (see the `diff` tests section for details).

2.3.1 assert

Often in unit testing, use of the `assert` function is helpful. `assert` is a `C++` library function which both takes and returns a `Boolean`. If the `Boolean` passed in evaluates to `true`, then the program continues as usual. Otherwise, the program will crash. Thus, it is often convenient to `assert` conditions which you expect in your tests. See below for an example test that creates an empty `ArrayList` and asserts that the size of the list is 0.

```
1
2  #include <iostream>
3
4  #include ``ArrayList.h''
5
6
7
8  void test_ArrayList_size() {
9
10     ArrayList mylist;
11
12     assert(mylist.size() == 0);
13
14 }
```

2.4 How do I diff tests with reference output?

In order to setup a `diff` test with `unit_test`, you need to:

- create a folder in your current working directory named `stdout`
- create one file per test that you would like to `diff` - each such file needs to have the same name as the test in your `unit_tests.h` file.

2.5 Can my tests take input from stdin?

Yes! In order to do this, you need to:

- create a folder in your current working directory named `stdin`

- create one file per test that you would like to have input from `stdin` - each such file needs to have the same name as the test in your `unit_tests.h` file. If there is no file with the same name as a test in the `stdin` directory, nothing will be sent to `stdin` for that test.

Then, when `unit_test` runs that test, the contents of the file with the same name as the test will be sent to `stdin`.

2.6 How do I compile my tests?

In order to compile your tests, you need a `Makefile` with a target named `unit_test`. This target must depend on a compiled driver file named `unit_test_driver.o`. **This driver file is built and compiled by the `unit_test` program, and contains the main() function.** See an example below for an ArrayList class.

```
CXX = clang++

CXXFLAGS = -Wall -Wextra



unit_test: unit_test_driver.o ArrayList.o

    $(CXX) unit_test_driver.o ArrayList.o



ArrayList.o: ArrayList.cpp ArrayList.h

    $(CXX) -c ArrayList.cpp
```

2.7 How do I run tests?

Once you've written your tests and your `Makefile` as discussed above, you simply run the following command:

```
unit_test
```

At that point, your code will be compiled, all your tests will be run, `valgrind` will be run on all passing tests, and `diff` will be run on any output files

matching names of any tests, and output will be provided for you. **There is no need to run `make` or to compile your code by hand! `unit_test` runs `make` for you.**

2.8 How do I write good tests?

`unit_test` is intended to help you write tests for your code *as you write it*. That is to say, after you write a function, you write one or more tests for that function. One of the great benefits here is that your code will be being checked for memory errors/leaks *as you work*, rather than at the end of your development cycle.

So, what might constitute a valid test for the constructor of an `ArrayList` class? How about this:

```
1
2 void test_constructor() {
3
4     ArrayList my_list;
5
6 }
```

While this may seem overly simplistic, 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.

```
1
2 void test_constructor() {
3
4     ArrayList my_list;
5
6     assert(my_list.size() == 0);
7
8 }
```

Clearly, this 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.

unit-test limitations

Given this style of testing, as mentioned above, often it can be difficult to write tests that only explicitly 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.

Chapter 3

How does unit_test actually work?

3.1 Intro

Great question! The contents of this chapter are completely optional, but if you're curious, read on!

`unit_test` is a `Python` script which does the following:

1. Extract the names of your test functions using regualr expressions.
2. Using a preexisting template, build the driver file. [more details on this below]
3. Compile and link your code with the driver.
4. For each test, run it. If it passes, run Valgrind on the test.
5. If the test has a file in a `stdout` folder with the same name, run a `diff` test on that folder.
6. report the results.

3.2 Driver File Details

The driver file template looks exactly like this:

```
/* unit_test_driver.cpp
 * Matt Russell
 * COMP15 2020 Summer
 * Updated 12/16/2020
 *
 * This file is used as the driver for unit testing.
 *
 * The 'tests' map will be auto-populated in the form:
 *
 *  { "test_name", test_name }
 *
 * Where "test_name" maps to the associated test function in
    unit_tests.h.
 */

#include <map>
#include <string>
#include <iostream>
#include "unit_tests.h"

typedef void (*FnPtr)();

int main(int argc, char **argv) {

    /* will be filled in by the unit_test script */
    std::map<std::string, FnPtr> tests {

    };

    /* first argument to main() is the string of a test
       function name */
    if (argc <= 1) {
        std::cout << "No test function specified. Quitting" <<
            std::endl;
        return 1;
    }

    /* extract the associated fn pointer from "tests", and run
       the test */
    FnPtr fn = tests[argv[1]];
    fn();

    return 0;
}
```

As the comment mentions at the top, the key insight here is that each test name from the `unit_tests.h` file is being turned into an entry of a `std::map<std::string, FnPtr>`. Let's break this down.

Note the declaration of the `typedef`'d type at the top of the file - `typedef void (*FnPtr)();` This is is a user-defined type named `FnPtr`, which represents a pointer-to-a-function which returns void and takes no arguments.

Thus, each key in the `map` is a `std::string`, and each value in the `map` is a **function which returns void and takes no arguments** (or, an object of type `FnPtr`). Okay, but how is this useful? Recall that the `unit_test` script fills in the details of this file for you!

So, after discovering the function names in the first step, it fills them in here as the form:

```
...
std::map<std::string, FnPtr> tests {
    { "my_first_test", my_first_test },
    { "my_scnd_test", my_scnd_test },
    ...
    { "my_last_test", my_last_test }
};
...
```

Now, let's see what else `main()` does. The key next bit is:

```
FnPtr fn = tests[argv[1]];
fn();
```

So, assuming that a valid test name is passed to `main()` as an argument from the command line, the program extracts from the map the associated pointer-to-the-function with the same name, and then runs it! (Note that, just as with the `[]` operator, no `->` is needed to dereference the pointer).

Thus, after parsing the names, building the driver file, and compiling/linking the code, `unit_test` simply calls `a.out testname` one time for each test - each test is thus run as its own process, but only one compilation is required to produce the test code.

There is certainly more nuance here in the `Python` script, but the logic of running the `diff` tests, etc. are relatively straightforward.

Chapter 4

Wrapping up

4.1 Getting Help

If you'd like more help, please see the following resources:

- See the review video here
- Ask questions on Piazza!
- Come to office hours!

4.2 Learning More

If you're interested in learning the ins and outs of `unit_test`, check out the script itself! It's located on the CS server at: `/g/15/2021f/bin/unit_test`. You're more than welcome to copy this file to your home directory and edit it if you'd like!

4.3 Contributing

If you're interested in contributing to the `unit_test` framework in some way, please reach out to me directly at - `mrussell at cs dot tufts dot edu` - there are opportunities to write code in `Python`, `TypeScript` (for the `VSCode` application), and generally to have lots of good experience working with code that's in-production now!