Unit Test Framework

Tutorial: Unit Test Framework

In this tutorial, you will learn how to write test cases using a lightweight framework that functions similarly to unit test frameworks used in real-life software development.

Setting Up

First, you will need the file `unit_test_framework.hpp`. This file is included with the starter code for EECS 280 projects where you are expected to use it. You can also download it directly (e.g. for this tutorial) with the following command.

```
$ wget
https://raw.githubusercontent.com/eecs280staff/unit_test_framework/main/unit_test_f
```

For this tutorial, you'll need three starter files, `tutorial.hpp`, `tutorial.cpp`, and `Makefile`.

```
$ wget https://eecs280staff.github.io/unit_test_framework/tutorial.hpp
$ wget https://eecs280staff.github.io/unit_test_framework/tutorial.cpp
$ wget https://eecs280staff.github.io/unit_test_framework/Makefile
```

These functions contain implementations of two functions (`slideRight()` and `flip()`) that work with vectors. The implementations each contain a bug! You'll catch the bugs by writing tests.

Your tests should go in a new file, called `tutorial_tests.cpp`. Add the following code to `tutorial_tests.cpp`:

```
#include "tutorial.hpp"
#include "unit_test_framework.hpp"

// We define a test case with the TEST(<test_name>) macro.
// <test_name> can be any valid C++ function name.
TEST(true_is_true) {
  ASSERT_TRUE(true);
}

TEST(numbers_are_equal) {
  ASSERT_EQUAL(42, 42);
}
```

```
13
14  TEST_MAIN() // No semicolon!
```

These are just example tests… we'll get to "real" tests for our vector functions soon.

You're probably wondering why some of the syntax in this code looks unusual. That's because this testing framework uses preprocessor macros to achieve functionality that wouldn't be possible with the plain C++ you're used to seeing. Preprocessor macros are beyond the scope of this course and in general should be used sparingly, so here's all you need to know:

- The `TEST(<test_name>)` essentially gets replaced (by the preprocessor) with a test function called `<test_name>`, where `<test_name>` is any valid C++ function name and `<test_name>` will be the name of the new test function. You do **NOT** need to put quotes around `<test_name>`, and if you do you'll get a compiler error.
- `TEST_MAIN()` gets replaced by a `main()` function that detects and runs all of the test cases you defined using the `TEST()` macro. Unlike in Project 1 where you had to explicitly call your test functions from `main()`, this framework handles that for you!
- `ASSERT_TRUE()` is one of several special test assertion preprocessor macros that you can use to check conditions in your test cases. You'll be using these instead of `assert()` in your unit tests. These will be demonstrated in more detail in the next section.

Compile and run this test case with the following two commands:

```
1  $ make tutorial_tests.exe
2  $ ./tutorial_tests.exe
3  Running test: numbers_are_equal
4  PASS
5  Running test: true_is_true
6  PASS
7
8  *** Results ***
9  ** Test case 'numbers_are_equal': PASS
10 ** Test case 'true_is_true': PASS
11 *** Summary ***
12 Out of 2 tests run:
13 0 failure(s), 0 error(s)
```

Another nice feature of the framework is that we can tell it to run only a subset of our test cases. If we wanted to only run the test `numbers_are_equal`, we could do it with this command:

```
1  $ ./tutorial_tests.exe numbers_are_equal
2  Running test: numbers_are_equal
3  PASS
```

```
4
5  *** Results ***
6  ** Test case 'numbers_are_equal': PASS
7  *** Summary ***
8  Out of 1 tests run:
9  0 failure(s), 0 error(s)
```

You can pass any number of test names as command line arguments, and it will only run the ones you've specified.

Special Test Assertions

One of the main reasons for using the special assertions provided by the framework is that they allow the framework to run all of your tests and report which ones passed and which ones failed. As you may have noticed, when you use regular `assert()` in your test cases, they automatically stop at the first failure. This can make it difficult to debug errors in one test that are actually caused by a function whose test cases didn't get a chance to run yet.

Here is a summary of all the special assertions that the framework provides:

Assertion	Description
`ASSERT_EQUAL(`*`first, second`*`)`	If `first == second` evaluates to false, the test will fail. **Note:** Do not use this if `first` and `second` are not comparable using the == operator. Other than this restriction, `first` and `second` may be any type.
`ASSERT_NOT_EQUAL(`*`first, second`*`)`	If `first != second` evaluates to false, the test will fail. **Note:** Do not use this if `first` and `second` are not comparable using the != operator. Other than this restriction, `first` and `second` may be any type.
`ASSERT_TRUE(`*`bool value`*`)`	If `value` is false, the test will fail.
`ASSERT_FALSE(`*`bool value`*`)`	If `value` is true, the test will fail.
`ASSERT_ALMOST_EQUAL(`*`double first, double second, double precision`*`)`	If `first` and `second` are not equal within `precision`, the test will fail.
`ASSERT_SEQUENCE_EQUAL(`*`first, second`*`)`	`first` and `second` must be sequences (e.g. arrays, vectors, lists). If `first` and `second` do not have equal elements, the test will fail.

Example: Tests for an `add()` function

Consider the `add()` function, declared in `tutorial.hpp` and defined in `tutorial.cpp`:

```
double add(double first, double second) {
  return first + second;
}
```

A thorough set of tests for this function would include several tests for basic functionality, as well as any special cases, such as adding 0, negative numbers, or non-integer floating point numbers. Here's an example:

```
TEST(add_basic) {
  ASSERT_EQUAL(add(2, 2), 4);
  ASSERT_EQUAL(add(2, 3), 5);
  ASSERT_EQUAL(add(5, 0), 5);
  ASSERT_EQUAL(add(0, 0), 0);
}

TEST(add_negative) {
  ASSERT_EQUAL(add(2, -2), 0);
  ASSERT_EQUAL(add(3, -5), -2);
  ASSERT_EQUAL(add(-3, 5), 2);
  ASSERT_EQUAL(add(-5, -5), -10);
}

TEST(add_floating_point) {
  ASSERT_ALMOST_EQUAL(add(0.1, 0.2), 0.3, 0.001);
  ASSERT_ALMOST_EQUAL(add(0.1, 0.1), 0.2, 0.001);
  ASSERT_ALMOST_EQUAL(add(0.1, -0.1), 0.0, 0.001);
  ASSERT_ALMOST_EQUAL(add(-0.1, -0.1), -0.2, 0.001);
}
```

Note that the `add_floating_point` test case uses `ASSERT_ALMOST_EQUAL()` with a tolerance of `0.001` instead of `ASSERT_EQUAL()` or `==`. You should always use the "almost" version when comparing results that might be slightly different due to limited numeric precision. (For example, `0.1` + `0.2` == `0.3` will yield `false` for most C++ implementations.)

Feel free to add these tests to `tutorial_tests.cpp` and run them to see how they work. For example:

```
$ make tutorial_tests.exe
$ ./tutorial_tests.exe
Running test: add_basic
```

```
PASS
Running test: add_floating_point
PASS
Running test: add_negative
PASS
Running test: numbers_are_equal
PASS
Running test: true_is_true
PASS

*** Results ***
** Test case "add_basic": PASS
** Test case "add_floating_point": PASS
** Test case "add_negative": PASS
** Test case "numbers_are_equal": PASS
** Test case "true_is_true": PASS
*** Summary ***
Out of 5 tests run:
0 failure(s), 0 error(s)
```

Exercise: Writing Unit Tests for `slideright()` and `flip()` vector functions

Now, let's add some real test cases for `slideRight()` and `flip()` to `tutorial_tests.cpp`.

For example, here's one test for each function (you can replace your existing code in `tutorial_tests.cpp` with the code below). Note the use of `ASSERT_SEQUENCE_EQUAL()` to verify the contents of two vectors are equal.

```cpp
#include <vector>
#include "tutorial.hpp"
#include "unit_test_framework.hpp"

using namespace std;

TEST(test_slide_right_1) {
  vector<int> v = { 4, 0, 1, 3, 3 };
  vector<int> expected = { 3, 4, 0, 1, 3 };
  slideRight(v);
  ASSERT_SEQUENCE_EQUAL(v, expected);
}

TEST(test_flip_1) {
```

```
  vector<int> v = { 4, 0, 1, 3, 3 };
  vector<int> expected = { 3, 3, 1, 0, 4 };
  flip(v);
  ASSERT_SEQUENCE_EQUAL(v, expected);
}

TEST_MAIN() // No semicolon!
```

Compile and run the tests with the following commands:

```
$ make tutorial_tests.exe
$ ./tutorial_tests.exe
Running test: test_flip_1
PASS
Running test: test_slide_right_1
FAIL

*** Results ***
** Test case "test_flip_1": PASS
** Test case "test_slide_right_1": FAIL
In ASSERT_SEQUENCE_EQUAL(v, expected), line 11:
{ 4, 4, 4, 4, 4 } != { 3, 4, 0, 1, 3 } (elements at position 0 differ: 4 != 3)

*** Summary ***
Out of 2 tests run:
1 failure(s), 0 error(s)
```

A failed test indicates there's a bug in our `tutorial.cpp` code. The tests above caught the bug in `slideRight` but not the one in `flip` (recall we mentioned there is a bug in both). You'll need to write more tests to create a thorough testing suite! Ideally, you should have enough tests that any reasonable bug will cause at least one test to fail.

On the other hand, when run against a correct implementation, all of your tests should pass. (Otherwise, it would be giving a false positive for detecting a bug.) If a test passes on a correct implementation, we call it *valid*.

Once you feel your tests are thorough, submit `tutorial_tests.cpp` to the unit test framework tutorial autograder. The autograder will check your tests against a set of buggy implementations of `slideRight` and `flip`, similar to the buggy versions functions provided with this tutorial. To earn points, your tests must detect (i.e. fail when run against) each of the bugs. Note that the autograder will discard any tests that are not valid when checked against a correct solution.

Note: Because these functions work with vectors, it's possible that a buggy version might go outside the bounds of the vector when given one of your tests, causing a segfault or other crash. For example:

```
$ ./tutorial_tests.exe
Running test: test_flip_1
PASS
Running test: test_flip_2
Segmentation fault
```

This still "counts" as catching the bug, because the program exits with a non-zero exit status (as is the case for a failed `ASSERT`) and we are alerted to the presence of a bug.