Testing

CMSC 240

Unit Tests

- Unit tests verify that a focused collection of code (e.g., function or class) behave as intended
  - Want these tests to isolate unit being tested from its dependencies (though this may be difficult)
  - If tested unit depends on other unit, sometimes use *mocks* (fake objects) as stand in during tests
    - Mocks are only used for testing

Mocks

- Can be used to simulate fine-grained control over how the dependencies behave during test
- Can also test how unit is interacting with mocks, to ensure this is correct
- Can use mocks to simulate rare events (e.g., out of memory) by programming them to throw exceptions

Types of Unit Tests

- Integration Tests: Test a collection of units together
  - Can also refer to testing interactions between software and hardware
  - NOT a replacement for individual unit tests, but complement them

Types of Unit Tests

- Acceptance Tests: Verify that software meets customer requirements
- Can be used to guide development
- Once acceptance tests passed, software is deliverable
- These tests become part of code base, so built-in protection against refactoring or feature regression
  - Feature regression: breaking an old feature when adding new

Types of Unit Tests

- Performance Tests: Just what it sounds like
  - Does code meet speed requirements?
  - Does code meet memory requirements?
  - Does code meet power consumption requirements?
- Typically have an idea where problems will occur, but can't be sure without testing

Types of Unit Tests

- Performance Tests
- Can’t know whether optimizations are working unless you measure after implementing
- Instrumentation: instrument code to provide relevant measures
  - Also detect errors, log program execution

Intrumentation

- Often part of customer requirements
  - E.g., procedure must execute in under 100ms and/or use less than 1MB of memory
  - By making this part of code, can automate checks as further optimizations are implemented

Test-Driven Development (TDD)

- We'll try implementing auto braking service using TDD
- So, the idea: if you're going to be coding unit tests anyway, why not code them first?
- TDD or Not TDD: Something of a religious war
  - Like vim vs emacs, where prens go, big endian vs little endian

TDD Advantages

- Key notion: write the code that tests a requirement *before* implementing solution
- Proponents claim:
  - Code is more modular, robust, clean, and well designed
- Good tests are excellent documentation
- Good test suite is a working set of examples that prevents regression

TDD Advantages

- Key notion: write the code that tests a requirement *before* implementing solution
- Great way to submit bug reports
  - Found by failed unit test
  - Once fixed, stays fixed, because test and code that fixes bug becomes part of the test suite

TDD: Red–Green–Refactor

- Red: First implement a failing test
  - Why? Make sure you're actually testing something!
- Green: Implement code that makes the test pass (no more, no less)
- Refactor: restructure existing code without changing functionality
  - E.g., replace code with library, rewrite for performance, elegance
  - If it breaks, test suite will tell you

Assertions

- Essential element of a unit test
- An assertion tests that some condition is met
  - If not met, test fails

```
#include <stdexcept>
constexpr void assert_that(bool statement, const char* message) {
  if(!statement) throw std::runtime_error{ message };
}

int main() {
  assert_that(1 + 2 > 2, "Something is profoundly wrong with the universe.");
  assert_that(24 == 42, "This assertion will generate an exception!");
}
```

What does constexpr mean?

Assertions

- Essential element of a unit test
- An assertion tests that some condition is met
  - If not met, test fails

```
#include <stdexcept>
constexpr void assert_that(bool statement, const char* message) {
  if(!statement) throw std::runtime_error{ message };
}

int main() {
  assert_that(1 + 2 > 2, "Something is profoundly wrong with the universe.");
  assert_that(24 == 42, "This assertion will generate an exception!");
}
```

What does constexpr mean? It instructs the compiler to evaluate the expression at compile time, if possible,

Assertions

- Essential element of a unit test
- An assertion tests that some condition is met
  - If not met, test fails

```
#include <stdexcept>
constexpr void assert_that(bool statement, const char* message) {
  if(!statement) throw std::runtime_error{ message };
}

int main() {
  assert_that(1 + 2 > 2, "Something is profoundly wrong with the universe.");
  assert_that(24 == 42, "This assertion will generate an exception!");
}
```

```
libc++abi.dylib: terminating with uncaught exception of type std::runtime_error: This
assertion will generate an exception!
Abort trap: 6
```

Test Harness

- Test harness: code that executes unit tests
- Idea: create code that invokes unit tests, but handles failed assertions gracefully
    - E.g., doesn’t crash on failed test(s)

Test Harness

- Test harness: code that executes unit tests

```
#include <string>
#include <exception>

--snip --

void run_test(void (*unit_test)(), string name) {
  try {
    unit_test();
    cout << "[+] Test \"" << name << "\" successful!" << endl;
  } catch (exception& e) {
    cout << "[-] Test failure in \"" << name << "\" " << e.what() << endl;
  }
}
```

Test Harness

- To make a *unit-test program* that will run all of the unit tests, place `run_test` inside the main function of a new program…

```cpp
#include <string>
#include <exception>
#include "Stack.h"

using namespace std;

void pushes_and_pops_work_correctly();
void moved_from_stack_has_null_values_variable();
void run_test(void (*)(), string);

void assert_that(bool statement, string message) {
  if (!statement) {
    throw runtime_error{message};
  }
}

int main() {
  //assert_that(1 + 2 > 2, "Something is profoundly wrong with the universe!");
  //  assert_that(24 == 42, "This assertion will generate an exception!");
  //  pushes_and_pops_work_correctly();
  run_test(pushes_and_pops_work_correctly, "pushes and pops work correctly");
  run_test(moved_from_stack_has_null_values_variable, "moved-from stack has null values variable");
}

void run_test(void (*unit_test)(), string name) {
  try {
    unit_test();
    cout << "[+] Test \"" << name << "\" successful!" << endl;
  } catch (exception& e) {
    cout << "[-] Test failure in \"" << name << "\" " << e.what() << endl;
  }
}
```

```cpp
#include <string>
#include <exception>
#include "Stack.h"

using namespace std;

void pushes_and_pops_work_correctly();
void moved_from_stack_has_null_values_variable();
void run_test(void (*)(), string);

void assert_that(bool statement, string message) {
  if (!statement) {
    throw runtime_error{message};
  }
}

int main() {
  //assert_that(1 + 2 > 2, "Something is profoundly wrong with the universe!");
  //  assert_that(24 == 42, "This assertion will generate an exception!");
  //  pushes_and_pops_work_correctly();
  run_test(pushes_and_pops_work_correctly, "pushes and pops work correctly");
  run_test(moved_from_stack_has_null_values_variable, "moved-from stack has null values variable");
}

void run_test(void (*unit_test)(), string name) {
  try {
    unit_test();
    cout << "[+] Test \"" << name << "\" successful!" << endl;
  } catch (exception& e) {
    cout << "[-] Test failure in \"" << name << "\" " << e.what() << endl;
  }
}
```

```
(base) m1-mcs-dszajda:week8 dszajda$ ./StackTesterAssertions
[-] Test failure in "pushes and pops work correctly" push or pop not working correctly
[+] Test "moved-from stack has null values variable" successful!
```

Mocking Dependencies

- Mock class (think ”mock up”): a special implementation that you generate for the purpose of testing a class that depends on the mock
  - That is, your class depends, say, on class `foo`. But you may not have the full `foo` implementation (perhaps it isn’t even coded yet)
  - Use the mock to test interactions with your class

Mocking Dependencies

- You have *complete* control over the mock -- you can do just about anything you want with it
  - Can record arbitrarily detailed info about how the mock gets called
    - E.g., number of times the mock is called and with which parameters
  - Can perform arbitrary computation in the mock

Mocking Dependencies

- You have *complete* control over the mock -- you can do just about anything you want with it. E.g.
  - How does your class respond to an out of memory error?
  - How many times did your class invoke methods in the dependent?
  - Etc.

One Note:

- Mocks are very useful, but if you end up refactoring your class(es), you'll likely have to refactor your unit tests as well
  - No way around that, unless the interface to your class doesn't change

Unit Testing and Mocking Frameworks

- Unit-testing frameworks make unit testing easier, just as IDEs can help make coding easier
  - Provide commonly used functions and the scaffolding necessary to tie tests into a user-friendly program
  - Functionality to help create consice, expressive tests

The Catch Unit-Testing Framework

- Catch Unit Testing Framework: One of three described in your text
- Very straightforward
- Written by Phil Nash
- Available at https://github.com/catchorg/Catch2/
- Header only library
  - So you can download the single-header version and #include in each unit-testing translation unit

Catch

- Easiest way to use this
  - Download single `catch.hpp` header file
    - https://raw.githubusercontent.com/catchorg/Catch2/v2.x/single_include/catch2/catch.hpp
  - Put it in your project directory
  - Be sure to `#include` it in unit test code

Catch

- Defining an entry point
  - Provide your test binary's entry point with `#define CATCH_CONFIG_MAIN`
  - That's it: Within the catch.hpp header file, it looks for `CATCH_CONFIG_MAIN` preprocessor definition
  - When found, Catch will add a `main` function (so you don't have to)
  - Automatically grabs all unit tests you have defined and wraps them in a test harness

```cpp
#define CATCH_CONFIG_MAIN
#include "catch.hpp"
#include <iostream>
#include <stdexcept>
#include <string>
#include <exception>
#include "Stack.h"

using namespace std;

TEST_CASE("Move Constructor") {
  // This is the setup.  The init here is run before each test.
  // Conceptually, this code is glued into the start of each SECTION
  Stack stack1{};
  stack1.push(-5);
  stack1.push(3);
  Stack stack2{std::move(stack1)};

  SECTION("moved from stack has null \"values\" variable") {
    REQUIRE(stack1.getValues() == nullptr);
  }

}

TEST_CASE("Push and pop") {
  Stack stack1{};
  Stack stack2{};

  SECTION("initial push() works correctly") {
    stack1.push(-5);
    REQUIRE(stack1.pop() == -5);
  }

  SECTION("pop() off empty stack causes thrown exception") {
    REQUIRE_THROWS(stack2.pop());
  }

  SECTION("stack resizes without error") {
    for (int i = 0; i < 100; ++i) {
      stack1.push(i);
    }
    REQUIRE(stack1.pop() == 99);
  }
}
```

Catch

- Building: just build the executable as usual
  - E.g., this is from my Makefile

```
StackTesterCatch: Stack.cpp Stack.h StackTesterCatch.cpp
	g++ $(CFLAGS) -o StackTesterCatch StackTesterCatch.cpp Stack.cpp
```

  - Note `StackTesterCatch.cpp` has no `main` method

- Running `StackTesterCatch` (after changing `Stack` to not throw exception on empty stack)

```
(base) m1-mcs-dszajda:week8 dszajda$ ./StackTesterCatch

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
StackTesterCatch is a Catch v2.13.1 host application.
Run with -? for options

-------------------------------------------------------------------------------
Push and pop
  pop() off empty stack causes thrown exception
-------------------------------------------------------------------------------
StackTesterCatch.cpp:34
...............................................................................

StackTesterCatch.cpp:35: FAILED:
  REQUIRE_THROWS( stack2.pop() )
because no exception was thrown where one was expected:

===============================================================================
test cases: 2 | 1 passed | 1 failed
assertions: 3 | 2 passed | 1 failed
```

Recall…

- Earlier, we defined separate functions for each unit test
- Passed a pointer to each function as the first parameter to `run_test`
- Passed name of the test as the second parameter
  - Which is redundant if you named unit test function well
- Implemented an `assert` function for each unit test

Catch

- Catch does all of that implicitly
- For each unit test, use `TEST_CASE` macro and Catch does all of the integration for you

Catch: Making Assertions

- Catch comes with a built-in assertion, with two distinct families of macros
  - `REQUIRE`: will fail a test immediately
  - `CHECK`: will allow test to run to completion, but still cause a failure
    - Useful if a group of related assertions can help lead the programmer toward a bug
  - Also, macros for assertions that should be false
    - `REQUIRE_FALSE`
    - `CHECK_FALSE`

Catch: Making Assertions

- Usage: wrap a Boolean expression with `REQUIRE` macro
  - If expression evaluates to false, assertion fails
  - You provide *assertion expression* that evaluates to true if assertion passes, false if it doesn’t
- Syntax: `REQUIRE(assertion-expression);`

Testing: Summary

- Unit tests
- Mocks
- Test-driven development
- Assertions
- Mocks
- Unit-testing frameworks