ECE 3574: Applied Software Design: Unit Testing using Catch

Chris Wyatt

The goal of today's meeting it to learn about a very important part of programming, testing.

- Unit tests
- Integration tests
- Testing functional requirements
- Error and Recovery testing
- Performance testing
- Usability testing
- How to test
- Unit Testing using Catch
- Test coverage

Testing

Testing is like exercise and eating healthy. You know its good for you but most people don't like it.
But testing prevents technical debt, keeping your code healthy, and can actually improve your productivity (like exercise and a good diet).
The mantra is: "Test Early, Test Often, Test Automatically"

Testing should be integrated into your workflow.

Almost everyone does this at some level.
"code a little, test a little"
But rather than manually testing each compile cycle, take the time to write tests.

What to test: Unit tests

Unit tests exercise each module, generally a class and associated functions.
Treat the public interface as a contract. Your test code checks the contract.

Simple example: puzzle

Suppose that we wanted to write a C++ class, `Puzzle`, that models an eight-tile sliding puzzle. You have probably seen these, a square array of tiles with numbers, letters, or part of an image printed on them, and one blank space. The tiles can slide left-right and up-down within the puzzle, exchanging positions with the empty location.

The goal is, from a scrambled state, slide the tiles around until the tiles show a particular image or spell some text.

Consider a 3 by 3 puzzle with eight tiles

It is addressed by row and column with 0-based indexing, with labels ‘A’ through ‘H’ and an empty spot denoted by the label ‘’(space). For example:

```
ABC
DEF
GH
```

is empty at position (2,2), has tile ‘D’ in position (1,0), tile ‘B’ on position (0,1), etc.

Lets begin by defining how our class should behave, it's *specification*.

`Puzzle` should support:

- Construction of a default puzzle instance with the layout in the example above,
- a `move` method taking two position arguments from and to, throwing an exception if either position or the move is invalid
- and a `get` method taking a position argument and returning the tile label at that position, throwing an exception if the position is invalid.

This specification is pretty detailed but it still has some missing information.

For example what types should the position, labels, and exceptions be?
We can nail down the specification further and define a set of tests that tell us how well we are doing implementing Puzzle by writing a test *before* we write the Puzzle class.
This is called Test-Driven-Development or **TDD**.

In it's simplest form unit tests are just a program that tries to use the code being tested.

So we might write a file `puzzle_test.cpp`:

```
#include "puzzle.h"

void run_tests();

int main()
{
  run_tests();

  return 0;
}
```

where the function `run_tests` has yet to be implemented and the puzzle.h file does not exist yet.

Lets implement the first version of our test by appending the following to `puzzle_test.cpp`

```
void run_tests()
{
  Puzzle p;
}
```

All this function does (at this point) is attempt to create an instance (an object/variable named p) of type Puzzle. If we try to compile this we get the error along the lines of

```
fatal error: 'puzzle.h' file not found
```

Congratulations!, we have written our first failing test (it will not even compile!).

So let us fix the problem.

We clearly need to create a file name "puzzle.h" defining a type Puzzle, like so:

```
class Puzzle {};
```

Now if we compile `puzzle_test.cpp` it gives no errors, and it even runs.
But clearly the test is not very good, we say it does not *cover* the functionality of the Puzzle specification.
Notice we have started to define the puzzle class, but it is what we call a *stub*, it is just a placeholder to get the tests to at least compile.

Improving the tests: what is the contract?

we need test code that calls and checks the constructor, the get method, and the move method.

```
void test_constructor()
{
  Puzzle p;

  assert(p.get(0,0) == Puzzle::A);
  assert(p.get(0,1) == Puzzle::B);
  assert(p.get(0,2) == Puzzle::C);
  assert(p.get(1,0) == Puzzle::D);
  assert(p.get(1,1) == Puzzle::E);
  assert(p.get(1,2) == Puzzle::F);
  assert(p.get(2,0) == Puzzle::G);
  assert(p.get(2,1) == Puzzle::H);
  assert(p.get(2,2) == Puzzle::EMPTY);
}
```

Getting our first test to compile

To get this to compile we will need to extend our stub to define the type for the label and the default values. We can use an enum for this:

```
class Puzzle
{
public:
  enum LabelType {A,B,C,D,E,F,G,H,EMPTY};

  LabelType get(int row, int col){
    return A;
  }
};
```

Compiling and running this gives us what we expect:

```
Assertion failed: (p.get(0,1) == Puzzle::B),
      function run_tests, file puzzle_test.cpp, line 19.
```

Moving on: another test

Lets test our move function by making a legal move and checking that it actually occurred.

```
void test_correct_move()
{
  Puzzle p;

  p.move(2,1,2,2); // slide H to the right
  assert(p.get(2,1) == Puzzle::EMPTY);
  assert(p.get(2,2) == Puzzle::H);
}
```

We would want to extend these tests to include invalid moves, etc.

So now we have some basic unit tests, which fail.

This gives us a few important things:

- We have a goal to work toward, namely to implement the methods of Puzzle so that all the tests pass.
- When all the tests pass, we have a reasonable belief the code is correct. Remember "UNTESTED code is BROKEN code".
- Further, there are automated tools that can check how many of the statements we write in our implementations are covered by the tests. This is called the test code coverage. Particularly in critical applications e.g. avionics, medical devices, 100% code coverage is needed.
- We can spot potential design flaws in the detailed design of the code early on in the process.

Designing code that can be tested

You can think of unit tests similar to IC hardware tests. Testing hardware requires forethought.
The same goes for code: define a contract and test it.
An issues that comes up: how do I test internal code (private methods) or methods I invoke?
Answer: split it out into its own module. Demeter's law helps here.

Integration tests

Once all your unit tests are written and pass you build up to test larger blocks of code.
In the above example we might have:

- puzzle module
- I/O module
- search module
- solver module

The solver module uses the search and puzzle module, thus defining a contract for it and testing it, implicitly tests the integration of puzzle and search.

Functional Tests

These are what you probably think of as tests. Does the entire program do what it is supposed to?
This is what the customer (however defined) cares about.
How you go about this depends on the kind of program it is.

- Is it interactive?
- Is it text-mode?
- Is it GUI driven?

We will talk more about this later.

Testing error handling

It is obvious you should test error handling, invalid input: bad files, etc.
But you should also test

- memory allocation failures
- incorrect permissions
- out of disk space
- network connectivity
- screen resolution
- and importantly *user interruption of a program*

When you can't proceed, cleanup and save as much state as you can – *fail gracefully*.
Don't just die or lock-up, dropping or corrupting users data. This can be trickier that it seems.

Performance testing

Performance testing covers update rates. For example:

- time to completion for tasks
- frame rate in a game or simulation
- network transfer speeds
- requests handled per second

These can help you diagnose problems and check what features need to be **non-blocking** and can be interrupted by the user.

Security testing

Code whose operation has security concerns has special considerations:

- test authentication and permissions
- test user input carefully, especially things like scripts that directly affect execution
- sanitize all input, add security checks to I/O, and test them

So now we know the what, how do we test

Regression testing: define data, input-output pairs, then test using the input, check against the output.
This works for all the tests we have looked at. The data can be

- synthetic: often generated by the testing code itself or from other programs
- real-world: collected and curated databases

An important source of real-world regression data is the bug/issue database.
For each bug found, write a test case that triggers it. This ensures that the bug never returns from the grave.

Another strategy is called *Fuzzifying*

Take real-world or synthetic test data, which is limited, and make random changes to it to generate larger test sets.
For example you might take a known good input file and corrupt it to see if you code handles it.
Similarly you can corrupt date and time fields, toggle version numbers, flip encoding of binary files (big/little endian).

How do we know when we are done?

Ideally we would like to put the code in every possible state, but this is impossible. Testing is never *done*.
As a surrogate we can measure code coverage: how many lines of code are executed over all the tests.
This requires instrumenting the code and is compiler dependent.
Examples:

- Visual C++ has a `/PROFILE` switch
- GCC and clang have the `--coverage` flag.
- External tools: for example Bullzeye Code Coverage (commercial)

These generate data that can be used to create reports of coverage, in summary, and line-by-line.

How do we test automatically

A measure of code quality is both the extent of tests and how easy they are to run.
Test Early, Test *Often*, implies Test **Automatically**
It can be handy to divide tests roughly by the time required to run them:

- short, quick running tests, which should be most of them, run all the time
- longer tests, for example some kinds of functionality tests, are run less often

Slow tests will not get run.

Finally, a word about *when* to test

- Run unit and integration tests very often (every compilation)
- run all test before committing changes to a repo
- long-running tests can be scheduled to run periodically or overnight

Unit Testing using Catch

Catch is a header-only "multi-paradigm automated test framework for C++ and Objective-C".

- It is very easy to use
- It supports a wide range of testing styles

Lets look at an example.

Exercise 07: Catch

See Website.

Next Actions and Reminders

- Project 1. You should have *at least* a working parser by now.
- Read CMake Tutorial