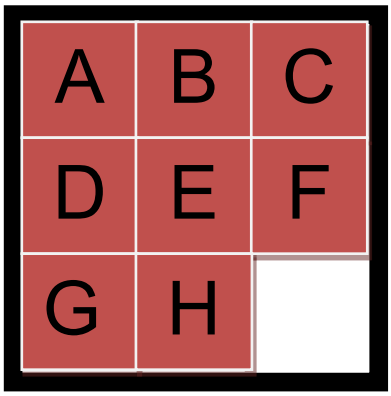
Writing Unit Tests with Catch and CMake

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, 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:



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

Our goal here is **not to write** a program to solve the puzzle (although it can be a fun way to hone your C++ OOP skills), but to just model the puzzle itself, a component of a larger puzzle solver program. But we want to ensure that the code works, perhaps before we write either the puzzle class or the solver, so we need to write some *unit tests*, code that tests individual units of other code, in this case the `Puzzle` class.

Let's 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 its 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. Let's 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

```
puzzle_test.cpp:1:10: fatal error: 'puzzle.h' file not found
#include "puzzle.h"
         ^
1 error generated.
```

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

Ok, let's get more serious about our tests. In order to test the specification, we need test code that calls and checks the constructor, the get method, and the move method. To test the constructor, we can adapt our simple test above to check, or *assert*, the instantiated object p has the correct default contents:

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

This requires adding the include `cassert` at the top of the `puzzle_test.cpp` file to use assert from the standard library. 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;
  }
};
```

Note: You can read more about *enum* from this link:
https://www.programiz.com/cpp-programming/enumeration

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.
```

since the stub always returns the label A. It may seem odd but **we are not concerned with the tests passing at this point** just writing the tests to cover the functionality desired and the minimal stub necessary to get the tests to compile and run.

Now let's test our move function by making a legal move and checking that it actually occurred. So that our testing code does not get too messy, lets *refactor* the tests into separate test functions with more meaningful names

```
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);
 }
 void test_move()
 {
   Puzzle         p;
   p.move(2,1,2,2);
   assert(p.get(2,1)  == Puzzle::EMPTY);
   assert(p.get(2,2)  == Puzzle::H);
 }
 int main()
 {
   test_constructor();
   test_move();
   return 0;
 }
```

To get this to compile requires defining a stub method inside the Puzzle class of `puzzle.h`

```
  void move(int from_row, int from_col, int to_row, int to_col)
  {
    // do nothing
  }
```

So now we have some basic unit tests, most of which fail (the first test in test_constructor passes by accident). 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. There is a saying, often attributed to Fred Brooks, "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.

An example of the latter is in the move test, where the call to slide tile H to the right looks like `p.move(2,1,2,2);`. A function/method call with that many arguments is a *code smell*, one of many we will see during the course. Unless I am looking at the definition of the method or some documentation how do I remember if it is from then to, to then from, and if the row is first or the column? Such code is ripe for being used incorrectly and causing a bug. So, we might add a separate type defining position and modify the call. Note this happens early in the coding process, before we have put even minimal effort into implementing the methods. That does not mean that design flaws do not show up later, but we catch them earlier this way, when refactoring is easier. Paradoxically, the time put into writing test and the stub saves time overall. This process of test writing, stub writing, refactoring continues until all the functionality is covered and the design is in good shape. The result is a complete specification. Only then do we worry about implementing the methods to get the tests to pass.

Catch and CMake

Our testing approach using assert is not very fancy. It stops at the first test that fails and does not give us very good diagnostic information about which test is failing. We could print out additional information and write our own version of assert that does not abort right away, however this is so common a task that there are many libraries that provide support for this. The one we will use is call Catch. It is a header only library, consisting of a single (large) file `catch.hpp` for convenience. We simply create a new source file, include the catch header, and start writing tests. This process is setup to compile and run as a test in cmake.

Most assignments will have a directory structure similar to the following

```
|-- CMakeLists.txt
|-- foo.hpp
|-- foo.cpp
|-- code_using_foo.cpp
|-- catch.hpp
`-- foo_test.cpp
```

Your tests for a fictional module defined in `foo.h` and implemented in `foo.cpp` are written in `foo_test.cpp`. The `CMakeLists.txt` file sets up the test so that when run to create the build directory a test target is automatically created. The exact form this takes depends on the build generator. For the makefile generators you can execute the tests using `make test`. For Visual Studio it creates a project called `RUN_TESTS`. Building this project will run the tests and show the status in the output window. It will look something like (for a failing test)

```
Running tests...
Test project _build
    Start 1: unittest_foo
1/1 Test #1: unittest foo .....................***Failed    0.00 sec


0% tests passed, 1 tests failed out of 1


Total Test time (real) =   0.01 sec


The following tests FAILED:
      1 - unittest_foo (Failed)
Errors while running CTest
```

This is just the overall status of the tests. Detailed testing output is placed in the text file `Testing\Temporary\LastTest.log` of the build directory.

Writing tests using catch takes care of the boilerplate code for you. For more, see the following tutorial and documentation:

- https://baptiste-wicht.com/posts/2014/07/catch-powerful-yet-simple-cpp-test-framework.html
- https://github.com/catchorg/Catch2/blob/master/docs/Readme.md