Chapter 7 - Testing

Its time for a little break from learning about C++ proper to focus on development practices. About 13 years ago, a new methodology emerged (or an old methodology was rediscovered): Test Driven Development or TDD.

In TDD, you build your code, a test at a time, writing as little code as possible to get the test to pass. Each test should focus on a single unit of code, and avoid any external dependencies. Writing in a TDD style tends to produce designs which favor small components with focused responsibilities. And, that, in turn, promotes good engineering choices.

Regardless of whether you practice TDD or just write tests eventually, you need a good framework for doing testing. And there are plenty of them out there. Historically, dynamic languages such as javascript and ruby have been blessed with particularly expressive frameworks, and languages like C++ have lagged behind. However, there are now some pretty decent libraries out there, and we are going to be focusing on one called Catch.

Getting Started

Laying out the project

The first thing that you need to do is lay out your project structure. For this project, I am going to use the following setup:

```
/Project_name
    /src
    /include
    /test
    CMakeLists.txt
```

Obivously, the `Project_name` directory should be changed to the actual name of the project.
The `src` directory will hold our cpp files.
The `include` directory will hold our includes.
THe `test` directory will hold our tests.

Download Catch

Catch Github Repo:
https://github.com/philsquared/Catch

Next, we are going to need to do is download Catch. So, follow the link above to the github repo and clone it or download a zip from the page. Then copy

`catch.cpp` from the `single_include` directory into a our new project's `include` directory.

Create the CMakeLists.txt file

Now, we are going to have to modify the master CMakeLists.txt file. Just like before, we are going to:

- set a minimum version of cmake
- set the project name
- set the include path to include the `include` directory
- add the `test` directory

Your top level CMakeLists.txt file should end up looking something like this:

```
cmake_minimum_required(VERSION 3.6)
project(catch_eg)
include_directories(include)
add_subdirectory(test)
```

Create the test/CMakeLists.txt file

Of course, our main CMakeLists.txt file is expecting to find a CMakeLists.txt file in the `test` directory. (because we used the `add_subdirectory` function), so lets create it.

```
file(GLOB cpps *.cpp )
file(GLOB hpps
../include/*.hpp )

add_executable(catchtest
        ${cpps}
        ${hpps})
```

Here I have used the GLOB variant of the file command to set `cpps` and `hpps` variables. Alternatively, we could have used the `set` command, followed by the variable name and each file we wished to include, but I am far too lazy for that.

Add a main cpp

Now we are going to add a couple of files in our test folder. The first will be the main.cpp, which will include catch.hpp and house a main function. We do this, per the instructions, to speed up compilation.

In test/main.cpp:

```
#define CATCH_CONFIG_MAIN
#include "catch.hpp"
```

The `#define CATCH_CONFIG_MAIN` is a preprocessor directive which Catch provides; it is responsible for setting up the main function and wiring up Catch.

Add a test.cpp

Now we are going to add a cpp to house our tests. In a real application, we might have a number of tests split across multiple cpp files. However for this demonstration, we are going to create a single test file and name it `test.cpp`.

In this file, we are going to `#include` a couple of files like so:

```
#include "catch.hpp"
#include "RingBuffer.hpp"
```

The first `#include` will appear in every test cpp you write. RingBuffer.hpp is the name of the cpp we are going to be testing in this exercise. So lets go add it now.

Add a RingBuffer.hpp in the include directory

In the include directory, create a file called RingBuffer.hpp.

RingBuffers

Since our TDD exercise will focus on creating a ring buffer, we should probably define what that is exactly. A ring buffer is a circular buffer with a fixed size that you can insert data into. If you insert more data that the fixed size of the ring buffer, then you will begin to overwrite the oldest data in the ring.

Our implementation won't be particularly sophisticated. We won't worry about concurrency, for example. And we will develop it using a BDD methodology. We will be using what we have learned in past lessons however. So, lets get started.

Create a RingBuffer template class

In RingBuffer.cpp add the following:

```
template <typename T>
class RingBuffer {

};
```

Now we start testing

Ok, we have a shell of a class. Lets start testing. In TDD you grow your design by adding a failing test for a particular feature, then work to make it pass. Rinse. Repeat. Lets start by adding our first test.

Catch supports two different syntaxes for Test Driven Development: TDD and BDD. Each style uses a different set of macros to describe the tests. We are going to go with BDD for now as it is a bit more descriptive. (BDD stands for Behavior Driven Development. You can google it, but really, I just like the more descriptive macros)

We start by describing a `SCENARIO`, which is just the name to organize all of the tests underneath it.

In test/test.cpp

```
SCENARIO("RingBuffer") {

}
```

We then add an invariant for our tests using the GIVEN macro, and an assertion about the result using the REQUIRE macro.

```
SCENARIO("RingBuffer") {
    GIVEN("an empty RingBuffer") {
        auto rb = RingBuffer<int>();
        THEN("the size will be zero")
            REQUIRE(rb.size() == 0);

    }
}
```

Get that sucker to pass

If you build the test now, it will fail when you run it. We obviously need to add a `size` method to our class.

```
// for size_t
#include <cstddef>

template <typename T>
class RingBuffer {
public:
    size_t size() const { return 0; };
};
```

We return a `size_t` which is an unsigned data type defined in the C++ standard which represents the capacity of a container. We do the minimum to get our test to pass, which is to return a value that satisfies the test.

Build your project and run your test. If you are not using an IDE that supports Cmake, you are going to be creating a build directory, cd'ing into that directory and running `cmake ...` Then you are going to run `make`. This should produce a `catchtest` executable, which calls your tests. Run that.

Whoo it passes. Ok we are done. Good job, you have implemented a ring buffer.

Keeping track of size and adding a constructor

Ok. Not so fast. Lets go back and fix this up. We obviously need to store the size of the ring buffer. At some point, we are going to add some storage for the ringbuffer and allocate a fixed size. However, we will need to keep track of the used size separately. So lets add a variable to keep track of the size, a backing vector, as well as a constructor to set this up.

When we construct an instance of the class, we are going to need to set a fixed size for the ring, as well as a default value to initialize the ring with. We will use an std::vector as the backing storage, and add a constructor which takes a size, as well as a default value. Fortunately, std::vector has a constructor which takes a size and value, doing that part of the work for us.

```
#include <vector>
// for size_t
#include <cstddef>

template <typename T>
class RingBuffer {
    size_t m_size;
    std::vector<T> m_storage;

public:
    // constructor
    RingBuffer(size_t capacity, const T& default_value)
    : m_size{0},
     m_storage{capacity, default_value }
     {}

    size_t size() const { return m_size; }
};
```

Update our test

Now we need to update our test to handle construction. We are going to chose a small number to set our default to - say, 4.

```
SCENARIO("RingBuffer") {
    GIVEN("a RingBuffer with a fixed size of 4 and default values of 0") {
        auto rb = RingBuffer<int>(4, 0);
        THEN("the size will be 4")
            REQUIRE(rb.size() == 4);

    }
}
```

Adding a second test

Now, how about testing what happens when we add a value to the ring buffer? Even before we add this to our class, we write a test. We will expect a method called `push_back` which takes a value of type `T` and adds it to the end of the buffer.

```
SCENARIO("RingBuffer") {
    GIVEN("a RingBuffer with a fixed size of 4 and default values of 0") {
        auto rb = RingBuffer<int>(4, 0);
        THEN("the size will be 4")
            REQUIRE(rb.size() == 4);
        WHEN("a value is appended to the ring buffer") {
            rb.push_back(23);
            THEN("the size becomes 1")
                REQUIRE(rb.size() == 1);
        }
    }
}
```

Get the test to pass

Ok, we obviously need to add a `push_back` method to RingBuffer. To get the test to pass, we need to increment the size when we push back.

```
#include <vector>
// for size_t
#include <cstddef>

template <typename T>
class RingBuffer {
    size_t m_size;
```

```
    std::vector<T> m_storage;

public:
    // constructor
    RingBuffer(size_t capacity, const T& default_value)
    : m_size{0},
     m_storage{capacity, default_value }
     {}

    size_t size() const { return m_size; }

    // add an element
    void push_back(T const& val) {
        m_size++;
    }
};
```

Now run the test. Notice that we didn't even bother to add the val to m_vector. We are just concerned with getting our test to pass.

Add another test

Lets add a test to pop a value off the front of the stack. We will call the method `pop_front`. Back in our test:

```
SCENARIO("RingBuffer") {
    GIVEN("a RingBuffer with a fixed size of 4 and default values of 0") {
        auto rb = RingBuffer<int>(4, 0);
        THEN("the size will be 4")
            REQUIRE(rb.size() == 4);
        WHEN("a value is appended to the ring buffer") {
            int val = 23;
            rb.push_back(val);
            THEN("the size becomes 1")
                REQUIRE(rb.size() == 1);
            AND_WHEN("we pop the value we just added off the front") {
                auto popped = rb.pop_front();
                THEN("the size decrements and the value is returned") {
                    REQUIRE(rb.size() == 0);
                    REQUIRE(popped == val);
                }
            }
        }
    }
}
```

A couple things to notice about Catch here. We nested the AND_WHEN under WHEN so that it lives in the same scope. Any setup done in a particular scope is only valid for that scope.

Running this test - Failure. Lets get it to pass.

Getting our new test to Pass

At this point, lets fix up push_back to store the value we are adding into our `m_storage`.

```
    void push_back(T const& val) {
        // postfix ++ increments after evaluating the expression
        m_storage[m_size++] = val;
    }
```

And add a pop_front which will return the front value.

```
    T pop_front() {
        // prefix -- decrements before evaluating the expression
        return m_storage[--m_size];
    }
```

Now lets run our tests. They should pass.

Add another test

Ok, something looks fishy with our implementation. What happens if we push a couple of values onto the buffer and then pop the front? Will it still work? Lets add a test to find out.

```
SCENARIO("RingBuffer") {
    GIVEN("a RingBuffer with a fixed size of 4 and default values of 0") {
        auto rb = RingBuffer<int>(4, 0);
        THEN("the size will be 4")
            REQUIRE(rb.size() == 4);
        WHEN("a value is appended to the ring buffer") {
            int val = 23;
            rb.push_back(val);
            THEN("the size becomes 1")
                REQUIRE(rb.size() == 1);
            AND_WHEN("we pop the value we just added off the front") {
                auto popped = rb.pop_front();
                THEN("the size decrements and the value is returned") {
                    REQUIRE(rb.size() == 0);
```

```
                REQUIRE(popped == val);
            }
        }
    }
    WHEN("multiple values are appended to the ring buffer") {
                int val = 23;
                rb.push_back(val);
                rb.push_back(256);
                THEN("the size becomes 2")
                    REQUIRE(rb.size() == 2);
                AND_WHEN("we pop the value off the front of the ring") {
                    auto popped = rb.pop_front();
                    THEN("the size decrements and the original value is returned") {
                        REQUIRE(rb.size() == 0);
                        REQUIRE(popped == val);
                    }
                }
            }

    }
}
```

Run the test.

And...

Blam.

Ok. We clearly need to modify our design. How can we get this to work? We are clearly going to have to do more book keeping. Ideas?

Exercise for the Reader

Modify the design to get the test to pass...
(hint: to solve this problem, you are going to have to add an index to the first element in the buffer, and an index to the back element of the buffer)

Exercise #2

Add a test to verify that the appropriate value is returned after pushing on more values than the actual capacity of the underlying array. Remember, the point of a ring buffer is to allow for this behavior. When exceeding the size of the buffer, you should start to replace the oldest members first.
(Hint: those indices you added for Exercise #1 should come in handy)

Make sure you test the size as well. The size should never exceed the capacity of the underlying vector (which you can test for using the `size` method on the

vector)