1. Introduction. We modify the approach of Gerard Meszaros's *xUnit Test Patterns*. The common features xUnit test frameworks share, they provide a way to perform the following tasks:

1. Specify a test,
2. Specify the expected results within a test in the forms of function calls to assertion methods,
3. Aggregate the tests into test suites that can be run as a single operation,
4. Run one or more tests to get a report on the results of the test run.

2. Caution: Abuse of Language. For object-oriented languages, there is some more boiler plate. The basic idea is to organize tests into suites. For our purposes, we have a "test case" for a single unit test, organized into a "test suite", which is then executed by a "test runner". Phrased differently, a test runner iterates through each test suite, and for each suite the runner iterates through each test case in the suite.

We deviate from the terminology standard in the xUnit framework, because a system under test has all its tests organized into a TestCase class. Its methods are precisely the tests to be performed.

However, JUnit followed a different pattern, using TestCase classes as command paterns (c.f., Cook's Tour), as a wrapper for a single test, and organize them into test suites. Each TestCase produces a TestResult upon running.

If we were more faithful in our terminology, we would have a TestCase be a collection of Test structures, a data structure wrapping around a function pointer. The TestSuite would be a collection of TestCases.

3. Our basic design will try to abstract away all the details, so the user just has to use macros to declare a new `TEST`(*name*) which are then collected in a `SUITE`(*name*) later in the test file. We will have a family of different assertion macros, because we need to pretty-print the arguments passed failed to satisfy the assertion. Ideally, there is one assertion in a `TEST`( ).

We also want to compartmentalize test suites, so if a test "blows up" crashing the program, we protect ourselves against interrupting testing the remaining suites. Towards this end, we could treat a `SUITE`(*name*) as a **void** *main*( ) function declaration — i.e., each test suite is a separate program which passes test result information back to its caller. We fork the test runner to execute a test suite, collect the test result information, and incorporate it back to be reported to the user. If one test suite has an *abort*( ) or *exit*( ) function call, then the test runner can continue on regardless.

The only difficulty is reporting back the results of a `SUITE`(*name*) back to the runner. If we are content with being restricted to UNIX or POSIX, then we could use *popen*( ) to communicate via pipes. The only alternative which springs to mind is use temporary files, which requires communicating the filename through parameters passed in the *execve*( ) function call.

4. Public declarations. The xunit header contains all the public-facing function declarations.

```
⟨ xunit.h 4 ⟩ ≡
#ifndef XUNIT_H
#define XUNIT_H
  ⟨ TestCase declarations 5 ⟩
  ⟨ TestSuite declarations 6 ⟩
  ⟨ Assertion macros 7 ⟩
  ⟨ Test macros 13 ⟩
#endif    /* XUNIT_H */
```

5. The test case declarations amount to just a typedef. It really encodes two things: the function pointer to the test itself, and metadata tracking the test results. The constructor expects only a function pointer, and returns a newly allocated TestCase object. Dually we free TestCase objects by reference. We should also track the name of the test case, to help the programmer look up failed tests.

```
⟨ TestCase declarations 5 ⟩ ≡
  typedef struct TestCase TestCase;
  TestCase *test_new(void(*test)(TestCase *this), const char *test_name);
  void test_free(TestCase *this);
```

See also section 8.

This code is used in section 4.

6. The test suite declarations include typedefs as well as the boiler-plate constructor and destructor functions. A test suite is a linked-list of either nested test suites or test cases. In either case, we abstract away the implementation details by just providing a method to add a suite to a given test suite, and another to add a test case to a test suite.

```
⟨ TestSuite declarations 6 ⟩ ≡
  typedef struct TestSuite TestSuite;
  TestSuite *suite_new(const char *name);
  void suite_free(TestSuite *this);
  void suite_addTest(TestSuite *this, TestCase *test);
  void suite_addSuite(TestSuite *this, TestSuite *suite);
```

This code is used in section 4.

7. Assertions. Assertions update the test result to reflect the outcome of the test: failure, success, exception. Although it is tempting to have success correspond to the "true" value (which is defined to be 1 in `stdbool.h`), we would paint ourselves into a corner. We only ask for the default value reflect an "untested" result.

```
⟨ Assertion macros 7 ⟩ ≡
  enum TestResult {
    RESULT_UNTESTED = 0,
    RESULT_SUCCESS = 1,
    RESULT_SKIPPED = 2,
    RESULT_FAILURE = 3,
    RESULT_EXCEPTION = 4
  };
```

This code is used in section 4.

8. We need to add a way to update a **TestCase** object to store its **enum TestResult** value after running. Conversely, we need a way to get the result from a test (which should always be defined).

```
⟨ TestCase declarations 5 ⟩ +≡
  void test_setResult(TestCase *this, enum TestResult result);
  enum TestResult test_result(TestCase *this);
```

9. Assertions come in a variety of forms. There are two basic assertions we care about: assert two expressions are "equal", or assert two expressions are "not equal". Three arguments must be provided — the left-hand side, the right-hand side, and the equality predicate. This causes some problems when we want to assert two numbers are equal, or two pointers are identical, because ≡ is not a function in C.

```
#define xstr(s) #s
#define ASSERT_EQUALS(lhs, rhs, equals) ASSERT_UPDATE((equals)((lhs),
        (rhs)), #equals##"("#lhs##", "#rhs##") expected, found: \
        "###equals##"("#(xstr(lhs))##", "##xstr(rhs)##")")
#define ASSERT_IDENTICAL(lhs, rhs) ASSERT_UPDATE((lhs) ≡ (rhs),
        #lhs##" == "###rhs##" expected, found: "##xstr(lhs)##" == "##xstr(rhs))
```

10. We need to sweep away the code to update the current test case under the rug of abstraction. Fortunately, our `TEST(name)` macro will make sure the test function expects a reference to a **TestCase** object where the results will be stored. The `SET_CURRENT_TEST_RESULT` macro will simply set the results for that particular **TestCase** object.

```
#define SET_CURRENT_TEST_RESULT(result) test_setResult(test, (result))
```

11. As far as updating the test result, we should report a *message* upon failure. Our conventions will be to have this message *only* reported upon failure, otherwise "No news is good news". Since this is a macro, we use the usual trick of wrapping it in a **do** { } **while** (*false*) which lets us use it as a statement.

One design decision we must make: what to do if we have a test with two assertions, and the first assertion failed? We should not "just **return**" after marking failure, because some objects may have been *malloc*( )-ed and ought to be freed. (One way out of this is to just use a garbage collector.) The other approach is to not update results after failing, or even run the test code. This seems simpler and more robust.

```
#define ASSERT_UPDATE(result_bool, fail_message)
        do {
          if (test_result(test) ≤ RESULT_SUCCESS) {
            if (result_bool) {
              SET_CURRENT_TEST_RESULT(RESULT_SUCCESS);
            } else {
              SET_CURRENT_TEST_RESULT(RESULT_FAILURE);
              report(report_stream, fail_message);
            }
          }
        } while (0)
```

12. We have some globally declared variables which parametrize what stream we report failures to, and the function useful for reporting them. Ostensibly we could use something even more general, like **void**(∗*report*)(**FILE**, **const char** ∗, ...), but I cannot think of a use-case where this would be more useful at the moment. I may revisit this as necessary, which would just require further revising our previous macro to have signature ASSERT_UPDATE(*success*, *format*, ...).

There are a variety of different output formats we could choose when reporting test results. This freedom motivates using a function pointer for (∗*report*)-ing results. The two principal choices are the TAP ("Test Anything Protocol") and the JUnit XML report. Usually the **FILE** *stream* is one end of a pipe, which is then communicated to the "master chef" test runner program.

```
    /* still in xunit.h */
  extern FILE report_stream;
  extern void(∗report)(FILE stream, const char ∗message);
```

13. User Macros. Now the user would use these macros to write their unit tests. The first macro we need simply handles declaring a function used in test cases.

⟨ Test macros 13 ⟩ ≡
```
#define TEST(name) void name(TestCase *test)
```
This code is used in section 4.

14. Index.

⟨ Assertion macros 7 ⟩ Used in section 4.
⟨ Test macros 13 ⟩ Used in section 4.
⟨ TestCase declarations 5, 8 ⟩ Used in section 4.
⟨ TestSuite declarations 6 ⟩ Used in section 4.
⟨ `xunit.h` 4 ⟩

XUNIT FRAMEWORK FOR C