FRUCTOSE – a C++ unit test framework

User Guide

Andrew Peter Marlow
44 Stanhope Road, North Finchley
London N12 9DT, England

andrew@andrewpetermarlow.co.uk

May 28, 2012

Contents

1 Introduction **1**
1.1 Keeping it simple 1
1.2 The license 2

2 Getting Started **3**
2.1 Downloading and installing 3
2.2 Your first test 3
2.3 Your second test 5

3 Writing FRUCTOSE test programs **9**
3.1 The skeleton 9
3.2 The assert macros 9
3.3 The command line 11

4 Features in detail **13**
4.1 Assertions 13
4.2 Floating point assertions 14
4.3 Loop assertions 15
4.4 Support for hamcrest matchers 17
4.5 Exception handling 19
4.6 Setup and teardown 19
4.7 Common functions 20
4.8 Code generation 20

Chapter 1

Introduction

FRUCTOSE is a simple unit test framework for C++. It stands for FRamework for Unit testing C++ for Test Driven development of SoftwarE.

1.1 Keeping it simple

FRUCTOSE is designed to be much smaller and simpler than frameworks such as CppUnit. It offers the ability to quickly create small standalone programs that produce output on standard out and that can be driven by the command line. The idea is that by using command line options to control which tests are run, the level of verbosity and whether or not the first error is fatal, it will be easier to develop classes in conjunction with their test harness. This makes FRUCTOSE an aid to Test Driven Development (TDD).

This is a slightly different way of approaching unit test frameworks. Most other frameworks seem to assume the class has already been designed and coded and that the test framework is to be used to run some tests as part of the overnight build regime. Whilst FRUCTOSE can be used in this way, it is hoped that the class and its tests will be developed at the same time.

FRUCTOSE does not follow the unit test framework pattern known as xUnit. This is to do with the aim of making FRUCTOSE cut down and much simpler than other frameworks.

A simple unit test framework should be concerned with just the functions being tested (they do not have to be class member functions although they typically will be) and the collection of functions the developer has to write to get those tests done. These latter functions can be grouped into a testing class. This article shows by example how this testing class is typically written and what facilities are available.

FRUCTOSE has a simple objective: provide enough functionality such that the developer can produce one standalone command line driven program per implementation file (usually one class per file). This means that most FRUCTOSE programs will use only two classes; the test class and the class being tested. This means that unlike other test frameworks, FRUCTOSE is not designed to be extensible. It was felt that the flexibility of other frameworks comes at the cost of increased size and complexity. Most other frameworks expect the developer to derive other classes from the framework ones to modify or specialise behaviour in some way, e.g. to provide HTML output instead of simple TTY output. They also provide the ability to run multiple suites. FRUCTOSE does not offer this. The test harness is expected to simply consist of a test class with its test functions defined inline, then a brief `main` function to register those functions with test names and run them with any command line options.

1.2 The license

FRUCTOSE is released under the LGPL license (Lesser General Public License). The weaker copyleft provision of the LGPL allows FRUCTOSE to be used in proprietary software because the LGPL allows a program to be linked with non-free modules.

A proprietary project can use FRUCTOSE to unit test its proprietary classes and does not have to release those classes under the GPL or LGPL. However, if such a project releases a modified version of FRUCTOSE then it must do so under the terms of LGPL.

Chapter 2

Getting Started

2.1 Downloading and installing

The homepage for FRUCTOSE is `http://fructose.sourceforge.net/`.

The project summary page is `http://sourceforge.net/projects/fructose`. It may be downloaded from there.

FRUCTOSE is header only implementation so there is no building to do. Once unpacked you are ready to go!

2.2 Your first test

Here is a small but complete test harness. This is example 1 in the examples directory.

```
// Copyright (c) 2011 Andrew Peter Marlow.
// All rights reserved.

#include <fructose/fructose.h>

// This is a very simple set of tests without there even
// being a class to test. These tests should all pass.
// To see the assertion mechanism working, use the -r flag
// to reverse the sense of the tests.

const int life_the_available_tests_and_everything = 42;
const int the_neighbour_of_the_beast = 668;
const int is_alive = 6;

struct simpletest : public fructose::test_base<simpletest> {
    void test42(const std::string& test_name) {
        fructose_assert(life_the_available_tests_and_everything == 6*7);
    }

    void testbeast(const std::string& test_name) {
        fructose_assert(the_neighbour_of_the_beast == 668);
    }

    void testfivealive(const std::string& test_name) {
        const int five = 5;
        fructose_assert_eq(five, is_alive);
    }
};

int main(int argc, char** argv) {
    simpletest tests;
    tests.add_test("test42", &simpletest::test42);
    tests.add_test("testbeast", &simpletest::testbeast);
    tests.add_test("fiveisalive", &simpletest::testfivealive);
    return tests.run(argc, argv);
}
```

When this program is run it produces the output:

```
Error: fiveisalive in ex1.cpp(30): five == is_alive (5 == 6) failed.

Test driver failed: 1 error
```

When the program is run with the command line option -v (for verbose), the following output is produced:

```
Running test case test42
========================


Running test case testbeast
===========================


Running test case fiveisalive
=============================

Error: fiveisalive in ex1.cpp(30): five == is_alive (5 == 6) failed.

Test driver failed: 1 error
```

There a few points to note about this test harness:

- All FRUCTOSE test harnesses need to include `<fructose/fructose.h>`.

- The writer of the test harness only needs to worry about two classes; the one being tested and the one doing the testing. This is felt to be a great simplication compared to other frameworks.
- FRUCTOSE uses the namespace `fructose` to scope all its externally visible symbols. Macros have global scope so they are prepended with `fructose_`. This makes the naming and scoping as consistent as possible. Too many unit test frameworks call their main assert macro `ASSERT`.
- The test class must inherit from the FRUCTOSE base class, `test_base`. This base class provides, amongst other things, the function `add_test`, which takes the name of a test and a function that runs that test.
- The functions that comprise the tests are members of the test class. So `test_base` needs to define a function that takes a member function pointer for the test class. The Curiously Recurring Template Pattern (CRTP) is used so that the base class can specify a function signature that uses the derived class.
- `add_test` is used to register and name the tests. Since the tests are named they can be selected via the command line. By default all tests are run. FRUCTOSE does provide a way to select which tests are run via command line options. Basically, the names of the tests are given on the command line.

  The command line is considered to consist of optional parameters followed by an optional list of test names. During parsing, FRUCTOSE sets various private data members according to the flags seen on the command line. These flags are available via accessors such as `verbose()`. This is another reason why the test class has to inherit from a base class. It provides access to these flags.
- The test program produces minimal output unless the `--verbose` option is used.

2.3 Your second test

Here is example 2 from the examples directory. It uses several different kinds of assert offered by FRUCTOSE.

```
// Copyright (c) 2011 Andrew Peter Marlow.
// All rights reserved.

#include <stdexcept>
#include <cmath>

#include <fructose/fructose.h>
```

```
const int life_the_available_tests_and_everything = 41;
const int the_neighbour_of_the_beast = 668;
const int is_alive = 6;

// These tests are rigged so that some of them fail.
// The tests include exercising floating point comparisons
// and exception handling.

namespace {
    void throw_a_runtime_error() {
        throw std::runtime_error("this is a runtime error");
    }
}

struct simpletest : public fructose::test_base<simpletest> {
    void test42(const std::string& test_name) {
        fructose_assert(life_the_available_tests_and_everything == 6*7);
    }

    void testbeast(const std::string& test_name) {
        fructose_assert(the_neighbour_of_the_beast == 666);
    }

    void testfivealive(const std::string& test_name) {
        const int five = 5;
        fructose_assert_eq(five, is_alive);
    }

    void testexceptions(const std::string& test_name) {
        fructose_assert_exception(throw_a_runtime_error(), std::logic_error);
        fructose_assert_no_exception(throw_a_runtime_error());
    }

    void testloopdata(const std::string& test_name) {
        static const struct {
            int line_number;
            int a;
            int b;
            int expected_result;
        } data[] = {
            {__LINE__, 3, 4, 12}
          , {__LINE__, 1, 50, 50}
          , {__LINE__, 5, 12, 60}
          , {__LINE__, 6, 6, 37}
          , {__LINE__, 7, 10, 70} };
        for (std::size_t i = 0; i < sizeof(data)/sizeof(data[0]); ++i) {
            if (verbose()) {
                std::cout << "Testing to see if "
                          << data[i].a << " * " << data[i].b
                          << " = " << data[i].expected_result
                          << std::endl;
            }
            int result = data[i].a * data[i].b;
            fructose_loop1_assert(data[i].line_number, i,
                                 result == data[i].expected_result);
        }
    }

    void testfloatingpoint(const std::string& test_name) {
        double my_pi = 3.141592654;
        double calc_pi = 4.0 * atan(1.0);
        fructose_assert_double_eq_rel_abs(my_pi, calc_pi, 0.01, 0.01);
        fructose_assert_double_eq(my_pi, calc_pi);
        fructose_assert_double_ne(my_pi, calc_pi);
        fructose_assert_double_ne_rel_abs(my_pi, calc_pi, 0.01, 0.01);
    }
};

int main(int argc, char** argv) {
    simpletest tests;
    tests.add_test("test42", &simpletest::test42);
    tests.add_test("testbeast", &simpletest::testbeast);
    tests.add_test("fiveisalive", &simpletest::testfivealive);
    tests.add_test("exceptions", &simpletest::testexceptions);
    tests.add_test("testloopdata", &simpletest::testloopdata);
    tests.add_test("testfloatingpoint", &simpletest::testfloatingpoint);
    return tests.run(argc, argv);
}
```

When you run this test program the following output is produced:

```
Error: test42 in ex2.cpp(28): life_the_available_tests_and_everything == 6*7 failed.
Error: testbeast in ex2.cpp(33): the_neighbour_of_the_beast == 666 failed.
Error: fiveisalive in ex2.cpp(39): five == is_alive (5 == 6) failed.
exception caught but not of expected type
exception caught as std::exception: this is a runtime error
Error: exceptions in ex2.cpp(44): throw_a_runtime_error()
    throws std::logic_error failed.
Error: An exception was thrown where none expected. Exception is: this is
    a runtime error
Error: exceptions in ex2.cpp(45): no_exception_thrown failed.
```

```
data[i].line_number: 59
index i = 3
Error: testloopdata in ex2.cpp(74): result == data[i].expected_result failed.
Error: testfloatingpoint in ex2.cpp(83): my_pi == calc_pi
  (3.141592654000000e+00 == 3.141592653589793e+00) failed floating point compare.
Error: testfloatingpoint in ex2.cpp(85): my_pi != calc_pi
  (3.141592654000000e+00 != 3.141592653589793e+00) failed floating point compare.
```

This example shows some of the different kinds of assertion failures that FRUCTOSE offers. In each case the test name is given along with the filename and line number where the assertion fails. Unlike some other test frameworks, FRUCTOSE does not halt on the first error – the code continues to show all the errors. This has to be borne in mind when writing the test code.

Run the program with the -h option to get help. This shows the standard features of any test program that uses FRUCTOSE. See section 3.3 for details.

Chapter 3

Writing FRUCTOSE test programs

3.1 The skeleton

Your test program will most likely start by looking like this:

```
#include <fructose/fructose.h>

struct tester : public fructose::test_base<tester> {
    // put the test methods here
}

int main(int argc, char** argv) {
    tester tests;
    // add named tests here
    return tests.run(argc, argv);
```

3.2 The assert macros

The FRUCTOSE assert macros are compared with the classic C assert and the assert macros of other unit test frameworks.

First, there are a couple of minor style points about the naming of the FRUCTOSE macros.

1. Generally one chooses uppercase for macros in order to tip the reader off that the token is a macro. However, where the macro is intended to look and feel like a function call macros are sometimes in lowercase. Well known examples include `assert` and `get_char`. Also, `toupper` and `tolower` are sometimes implemented as macros. The FRUCTOSE macros are expected to be used in a similar way to the standard C `assert` macros

and are designed to look and behave as function calls. Hence they are in lowercase. During the early evolution of FRUCTOSE some developers asked for them to be in uppercase so both are provided. This document always uses the lowercase form.

The `fructose_assert` macro takes a boolean expression. It takes no action if the condition evaluates to true. However, when the condition is false it reports the name of the test that failed, the filename and line number and the assertion expression. It also increments the error report (for a final report at the end of all tests) and optionally halts (depends on whether or not the command line option has been specified to halt on first failure).

Note, unlike some other test frameworks, an assertion failure does not abort the function from which it was invoked. This is deliberate. Other test frameworks take the view that when a test fails all bets are off and the safest thing to do is to abort. FRUCTOSE takes a different view. FRUCTOSE assumes it is being used as part of a TDD effort where the developer will typically be developing the class and its test harness at the same time. This means that the developer will want to execute as many tests as possible so he can see which pass and which fail. It also means that the developer needs to be aware that code immediately after a FRUCTOSE assert will still be executed so it should not rely on the previous lines having worked. One style of writing test cases that is in keeping with this is to use a table of test data. Each row in the table contains the input and the expected output. This allows the test code to loop over the table performing lots of tests without a test having to rely on successful execution of the previous test.

FRUCTOSE offers the following assertions:

- `fructose_assert(X)`
  This is analogous to the C assert macro provided by the standard. It takes a boolean expression and fails if the expression is false.

- `fructose_fail(X)`
  Fail unconditionally. This is useful as a placeholder for incomplete tests.

- `fructose_assert_eq(X, Y)`
  Verify that X == Y.

- `fructose_assert_ne(X, Y)`
  Verify that X != Y.

- `fructose_assert_same_data`
  Verify that two blocks of memory of a specified length are equal.

- Floating point assertions
  See section 4.2 for details.

- Loop assertions
  Like `fructose_assert` but for use within single or doubly nested loops. See section 4.3 for details.

- `fructose_assert_that`
  Similar to `fructose_assert_eq` but using hamcrest matchers. See section 4.4 for details.

- `fructose_assert_exception(X, E)`
  Asserts that upon evaluation of expression X, the specified exception E (or derived class thereof) is thrown.

- `fructose_assert_no_exception(X)`
  Asserts that upon evaluation of expression X, no exception is thrown.

3.3 The command line

The following command line options come as standard for every unit test built using FRUCTOSE:

- -h[elp] provides built-in help. Not only does it produce help on all the options here but also it names the tests available so they can be run selectively.

- -a[ssert_fatal] causes the test harness to exit upon the first failure. Normally the harness would continue through any assertion failures and produce a report on the number of failed tests at the end. The developer needs to be aware of this when coding the tests and ensure that in a given test function that has a number of assertions, the correct working of the tests does not depend on the assertions passing. If the developer finds there are such dependencies these tests should be rearranged into separately named tests. Sometimes this is awkward to do so the flag is provided for these cases. When this flag is enabled, the first assertion failure results in an exception being thrown, which is caught and reported by the `run` function.

- -v[erbose] this sets a flag which can be tested in each test function. This allows test functions to output diagnostic trace when the option has been enabled. This is of particular use during TDD. The developer may find it useful to leave a certain amount of this trace in, even when all the tests pass, in case there is a regression, as a debugging aid.

- -r[everse] reserve the sense of all assertion tests. This is primarly of use when testing the framework itself.

- The remaining command line options are taken to be test names. All supplied test names are checked against the registered test names. Only registered test names are allowed.

- Each test name can optionally be followed by `--args <args>`. This passes per-test command line options to the test.

When the example program above is run with the `-help` option, the following output is produced:

```
USAGE:

    ./ex1 [-h] [-r] [-a] [-v] [--] <testNameString> ...

Where:

   -h,  --help           produces this help

   -r,  --reverse        reverses the sense of test assertions.

   -a,  --assert_fatal   is used to make the first test failure fatal.

   -v,  --verbose        turns on extra trace for those tests that have made use of

   --,  --ignore_rest    Ignores the rest of the labeled arguments following this f


   <testNameString>  (accepted multiple times)
     test names

   Any number of test names may be supplied on the command line. If the
   name 'all' is included then all tests will be run.

Supported test names are:
    beast
```

The above test should pass, so to see the kind of report that is given when a test fails, run the harness with the `-reverse` option. It produces the following output:

```
Error: beast in ex3.cpp(6): neighbour_of_the_beast
              == 668 failed.

Test driver failed: 1 error
```

It it tedious to have to remember to call `add_test` and pick a name once the test method has been written. Forgetting to do this means the test never gets called. Some people prefer implicit test registration for this reason. However, FRUCTOSE does provide a code generation facility that gives some help in this area (see section 4.8).

Chapter 4

Features in detail

4.1 Assertions

Although FRUCTOSE uses words such as 'assert' and 'assertions', one must bear in mind that these are not C-style `assert` statements. They do not cause core dumps. They produce diagnostic output and increment an error count in the event that a tested condition returns `false` (i.e. they are asserting that the supplied condition should be true). They are macros and use the `__FILE__` and `__LINE__` macros to show which file and line the error occurred on. The simple case is the `fructose_assert` macro, which produces an error if the supplied condition is false.

A number of other assertion macros are provided:

- `fructose_assert_eq(X,Y)`
  Assert that X equals Y. If they are not than the names of X and Y and their values are reported. This level of detail would not be present if the developer used
  `fructose_assert(X == Y)` instead.

- `fructose_assert_ne(X,Y)`
  Assert that X does not equal Y. If they are equal than the names of X and Y and their values are reported. This is provided for symmetry with `fructose_assert_eq(X,Y)`.

- `fructose_assert_double_eq(X,Y)`
  A familiy of floating point assertions is provided. Substitute `lt`, `gt`, `le`, or `ge` for `eq` to check for less than, greater than, less than or equal to and greater than or equal to. Floating point assertions are a special case for two reasons:

  1. floating point compares need to be done with configurable relative tolerance and or absolute tolerance levels.

2. the default left shift operator is insufficient to show enough precision when a floating point compare has failed.

The macro family mentioned does floating point compares using default tolerances. The macro family `fructose_assert_double_ <test>_rel_abs(X,Y,rel_tol,abs_tol)` provides the same tests but with the tolerances specified explicitly.

- Loop assertions.
  Macros are provided that help the developer track down the reason for assertion failures for data held in static tables. What is needed in these cases in addition to the file and line number of the assertion is the line number of the data that was tested in the assert and the loop index. There is a family of macros for this named `fructose_loop<n>_assert`, where `<n>` is the of looping subscripts. For example, when the array has one subscript the macro is `fructose_loop1_assert(LN,I,X)` where `X` is the condition, `LN` is the line number of the data in the static table and `I` is the loop counter. `fructose_loop2_assert(LN,I,J,X)` tests condition `X` with loop counters `I` and `J`.

- Exception assertions.
  The test harness may assert that a condition should result in the throwing of an exception of a specified type. If it does not then the assertion fails. Similarly, a harness may assert that no exception is to be thrown upon the evaluation of a condition; if one is then the assertion fails. `fructose_assert_exception(X,E)` asserts that when the condition `X` is evaluated, an exception of type `E` is thrown. `fructose_assert_no_exception(X)` asserts that on evaluation of expression X, no exception will be thrown.

- `fructose_assert_that(value, matcher)`
  This is an optional part of FRUCTOSE that integrates with the C++ port of hamcrest matchers. To enable this part of FRUCTOSE, define the macro `FRUCTOSE_USING_HAMCREST` (see section 4.4).

4.2 Floating point assertions

Comparing floating point numbers for exact equality is not always reliable, given the limitations of precision and the fact that there are some real numbers that can be expressed precisely in decimal but not precisely in binary. The common way around this is to compare floating point numbers with a degree of fuzziness. If the numbers are "close enough" then they are judged to be equal.

A common mistake that is made in fuzzy floating point comparisons is for closeness check to be done using an absolute value for the allowed difference between the two values. The perils of doing this are explained in great detail in section 4.2 of Ref[10], where Knuth explains how to use relative tolerances to overcome problems. Very few unit test harnesses seem to provide much to help here. See the example below for a simple floating point compare assertion with default tolerances:

```
#include "fructose/test_base.h"
#include <cmath>

struct simpletest :
  public fructose::test_base<simpletest> {
    void floating(const std::string& test_name) {
      double mypi = 4.0 * std::atan(1.0);
      fructose_assert_double_eq(M_PI, mypi);
    }
};

int main(int argc, char* argv[]) {
  simpletest tests;
  tests.add_test("float", &simpletest::floating);
  return tests.run(tests.get_suite(argc, argv));
}
```

Since π is equal to four times arctan(1), this assertion will pass. Using the `-reverse` option shows the kind of output that is produced when such a test fails:

```
Error: float in ex3.cpp(8):
   M_PI == mypi (3.141592653589793e+00 ==
         3.141592653589793e+00) failed floating point compare.

Test driver failed: 1 error
```

Note that the numbers are output in scientific format with the maximum number of significant figures available in most implementations of `doubles`.

4.3 Loop assertions

Consider the class `multiplication` which provides a function `times` that returns the constructor arguments x and y multiplied together.

```
class multiplication {
  double m_x, m_y;
public:
  multiplication(double x, double y)
     : m_x(x), m_y(y) {};
  double times() const {return m_x * m_y; };
};
```

A FRUCTOSE unit test can use a table of test data of values for x and y and the expected result: One could use the fructose_assert macro to test that the expected value is equal to the computed value:

```
#include "fructose/test_base.h"

struct timestest :
  public fructose::test_base<timestest> {
    void loops(const std::string& test_name) {
      static const struct {
          int line_number;
          double x, y, expected;
          } data[] = {
        { __LINE__, 3,      4,    12}
      , { __LINE__, 5.2,    6.8,  35.36}
      , { __LINE__, -8.1,   -9.2, 74.52}
      , { __LINE__, 0.1,    90,   9}
      };
      for (unsigned int i = 0;
           i < sizeof(data)/sizeof(data[0]); ++i) {
             multiplication m(data[i].x, data[i].y);
             double result = m.times();
         fructose_assert(result == data[i].expected);
      }
  }
};

int main(int argc, char* argv[]) {
  timestest tests;
  tests.add_test("loops",
                 &timestest::loops);
  return tests.run(argc, argv);
}
```

However, this is not very useful when there is an assertion failure because it doesn't tell you which assertion has failed. One way is to add verbose tracing that gives all the detail:

```
      for (unsigned int i = 0;
           i < sizeof(data)/sizeof(data[0]); ++i) {
             multiplication m(data[i].x, data[i].y);
             double result = m.times();
             if (verbose()) {
               std::cout << data[i].x
                         << " * " << data[i].y
                         << " got " << result
                         << " expected "
```

```
                        << data[i].expected
                        << std::endl;
            }
            fructose_assert(result == data[i].expected);
    }
```

However, another way which does not rely on the verbose flag is to use the loop assert macro family. These macros take the source line number of the test data and loop indexes as macro parameters. The code below shows the test modified to indicate the line of data and the loop index value. This makes the use of the verbose flag unnecessary.

```
    for (unsigned int i = 0;
         i < sizeof(data)/sizeof(data[0]); ++i) {
           multiplication m(data[i].x, data[i].y);
           double result = m.times();
           fructose_loop1_assert(
                  data[i].line_number, i,
                  result == data[i].expected);
    }
```

If the code code employed an array with two dimensions and thus had nested loops, one would use the macro `fructose_loop2_assert(lineNumber, i, j, assertion)`.

No other unit test framework that was examined provides loop asserts. These are very useful because they encourage the developer to to do systematic testing by covering more cases more conveniently.

The convenience of loop assert testing does not mean the developer can provide large volumes of test data just for the sake of appearing to do large amounts of tests. It is hoped that the loop asserts will lead to an increased use of a technique used in testing known as equivalence partitioning (ref [11]). This is a method that divides the input domain into classes of data from which test cases can be derived. All the data for the individual cases in all these data classes for a given FRUCTOSE test would be in static data table such as the one shown above. The classes would be grouped in the table with comments to show the grouping of classes of errors. An ideal test case uncovers a whole class of errors on its own that might otherwise require many cases to be executed. There is another technique called Boundary Value Analysis (also in Ref [11]) which loop asserts are well suited to.

4.4 Support for hamcrest matchers

At version 0.9.0, fructose was extended to provide (optional) support for hamcrest matchers.

Hamcrest (ref [17]) provides a library of matcher objects (also known as constraints or predicates) allowing 'match' rules to be defined declaratively, to be used in other frameworks. Typical scenarios include testing frameworks, mocking libraries and UI validation rules.

Hamcrest has been ported to Java, C++, Objective-C, Python, PHP and Erlang.

Note: Hamcrest it is not a testing library: it just happens that matchers are very useful for testing.

The java version of hamcrest is perhaps the most well-known. The C++ port is a little awkward to track down so at version 1.0.0 of FRUCTOSE it was included. It is not enabled by default though, in order to minimise dependencies. Hamcrest has to be built as a library. Therefore the macro `FRUCTOSE_USING_HAMCREST` is used by those parts of FRUCTOSE that work with hamcrest. Turn this macro on to enable those parts.

FRUCTOSE provides the macro fructose_assert_that which works with hamcrest matchers. Example 11, reproduced below, shows how this works.

```
// Copyright (c) 2011 Andrew Peter Marlow.
// All rights reserved.

// This example shows how fructose integrates with hamcrest matchers.
// By default all the test pass. Use the reverse mode of fructose
// to see the nice messages you get via hamcrest.

#define FRUCTOSE_USING_HAMCREST
#include "fructose/fructose.h"

using namespace hamcrest;

struct features_test : public fructose::test_base<features_test>
{
    void features(const std::string& test_name) {
      fructose_assert_that(4, equal_to(4));
      fructose_assert_that(10, is(equal_to(10)));
      fructose_assert_that(1, is_not(equal_to(2)));
      fructose_assert_that("good", any_of(equal_to("bad"),
                                          equal_to("mediocre"),
                                          equal_to("good"),
                                          equal_to("excellent")));
      fructose_assert_that("politicians", is_not(any_of(equal_to("decent"),
                                                        equal_to("honest"),
                                                        equal_to("truthful"))));
    }

};

int main(int argc, char* argv[])
{
    features_test tests;
    tests.add_test("features", &features_test::features);
    return tests.run(argc, argv);
}
```

4.5 Exception handling

FRUCTOSE only uses exceptions to deal with errors if the `-assert_fatal` flag is given on the command line. But FRUCTOSE realises that a class being tested may throw an exception which the developer did not expect to occur during the run of the unit test. This is treated as a fatal error. It is caught and reported and causes the unit test to terminate.

The developer may wish to test that certain exceptions are thrown when they are meant to be. The macro `fructose_assert_exception(X,E)` is provided for this. It asserts that during the evaluation of the condition `X`, an exception of type `E` is thrown. If exception of a different type is thrown, or no exception is thrown, then the assertion fails.

```
#include "fructose/test_base.h"
#include <stdexcept>
#include <vector>

struct timestest :
  public fructose::test_base<timestest> {
    void array_bounds(const std::string& test_name) {
      std::vector<int> v;
      v.push_back(1234);
      fructose_assert_exception(v.at(2),
                                std::out_of_range);
    };
};

int main(int argc, char* argv[]) {
  timestest tests;
  tests.add_test("array_bounds",
            &timestest::array_bounds);
  return tests.run(tests.run(argc, argv);
}
```

4.6 Setup and teardown

When FRUCTOSE was first developed it was felt that the setup and teardown machinery offered by other frameworks would not be required. However, it was later discovered that on relatively rare occasions it is useful. Most of the time if any setup and teardown procedure is required at all it is needed once at the start and finish of the program. In these cases it can be done in the constructor and destructor of the test class. But there will be cases where the setup and teardown need to be done for each test. Hence, `setup` and `teardown` functions are provided as virtual functions with an empty default implelementation in `test_root`. If the test class needs to override these then it can do so by providing its own `setup` and `teardown` functions. These get called by the `run` function of `test_base` before and after each test invocation.

4.7 Common functions

Certain functions are provided via the base class that are available for tests to use. These are as follows:

- `bool verbose() const`
  Returns true if -v or –verbose was given on the command line. This can be used by the test to provide more information.

- `std::vector<std::string> get_args() const`
  This is provided so that a test can pick up any command line arguments that are specific to the test. Example 15 shows this facility in use. Run the example with the option `--args <args>` after the test name. Where multiple arguments are required, enclose them in quotes (either single or double, provided they match). Two tests are provided in ex15 so that it can be demonstrated that different tests can be passed different arguments.

4.8 Code generation

FRUCTOSE comes with a python script that performs code generation. The script is called `fructose_gen.py`.

Many thanks to Brian Neal, who wrote and contributed this generator.

History

`fructose_gen.py` is used to generate function main, instantiate a test suite and make the calls to add_test. The approach is based on parsing a header file that contains the test class.

In early version of FRUCTOSE there was a C++ code generator. It was called 'generator' and was contributed by Chris Main. It used macros in the test code to spot the test class and associated test cases. `FRUCTOSE_CLASS(className)` was used to denote the name of the test class. `FRUCTOSE_TEST(testName)` was used to start the definition of a test function.

Later on, the python script was added. The python script knew more about C++ syntax and was able to perform the code generation without the test code having to use generator macros.

At version 1.2.0 of FRUCTOSE, the python script was extended to code with test source that used old style generator macros without having to be explicitly told (via the `--generator` option). The made the original code generator redundant and so it was removed.

This history explains why the python script has a `--generator` option and why FRUCTOSE comes with the macros `FRUCTOSE_CLASS` and `FRUCTOSE_TEST`.

An example of how to use the (python) generator is given in examples/ex06 in the FRUCTOSE distribution. It is an old-style example that uses the macros.

Here is the header file, `ex06.h`, with some comments removed for brevity.

```
#include <fructose/fructose.h>
#include <cmath>

FRUCTOSE_CLASS(simpletest)
{
public:
    FRUCTOSE_TEST(floating)
    {
      double mypi = 4.0 * std::atan(1.0);
      fructose_assert_double_eq(M_PI, mypi);
    }
};
```

To run generator on this, enter the command: `generator examples/ex6.h >examples/ex6.cpp`. The resultant ex6.cpp is shown below:

```
#include "ex6.h"

#include <stdlib.h>

int main(int argc, char* argv[])
{
    int retval = EXIT_SUCCESS;

    {
        simpletest simpletest_instance;
        simpletest_instance.add_test("floating", &simpletest::floating);
        const int r = simpletest_instance.run(argc, argv);
        retval = (retval == EXIT_SUCCESS) ? r : EXIT_FAILURE;
    }

    return retval;
}
```

Sample usage

```
$ python fructose_gen.py [options] test1.h test2.h ... testN.h > main.cpp
```

In this example, 'fructose_gen.py' will read the Fructose test files 'test1.h' through 'testN.h' and produce on standard output C++ code with a generated 'main()' function. This auto-generated code will instantiate all the test instances, register the appropriate tests with each instance, and then execute each test in turn.

To see usage information and a list of options:

```
$ python fructose_gen.py --help
```

Code Generation Styles

'fructose_gen.py' supports two styles of code generation, described below.

xUnit Style

The default style is xUnit style. In this form, 'fructose_gen.py' will scan C++ code looking for classes or structs that inherit from `fructose::test_base<>`. Inside those classes or structs, member functions that match the following pattern are assumed to be test functions:

```
void testXXXX(const std::string&)
```

Upon finding such a function, 'fructose_gen.py' will register that member function as a test with the name `testXXXX`.

Generator Style

The option `-g` (or --generator in long form) was provided to remain backward compatible with original the 'generator' program from Chris Main. In this style, 'fructose_gen.py' will scan files for the `FRUCTOSE_CLASS`, `FRUCTOSE_STRUCT` and `FRUCTOSE_TEST` macros.

However, this option is now redundant since the script now watches out for either form. The option is retained for backward compatibility.

Caveats

'fructose_gen.py' is not a true C++ code parser, and in fact is quite simple in how it operates. This is sufficient for most cases, but please be aware of the following limitations.

- Ensure your class (or struct) definition is all on one line:

  ```
  class my_unit_test : public fructose::test_base<my_unit_test>
  ```

  If you split the above across multiple lines 'fructose_gen.py' will not recognize your class and will not generate a test instance for it.

- ‘fructose_gen.py‘ does not understand C-style comments or the preprocessor. To comment out a test, you can either use C++ comments, or change the function name slightly to ensure it won’t be recognized.

  Examples:

  ```
  /*
   ** void test_is_sorted(const std::string& name)  // this won't work
   */

  #if 0
  void test_is_sorted(const std::string& name)   // this won't work
  #endif

  void not_a_test_is_sorted(const std::string& name) // this works
  // void test_is_sorted(const std::string& name)    // this works
  // FRUCTOSE_TEST(is_sorted)                        // this works
  ```

  The above also applies to commenting out test classes.

Examples

`ex12` shows how the generator works off C++ code directly without having to use macros.

`ex13` shows the `--generator` option, for the same tests as ex12 but using macros.

Both examples have rules in the Makefile to compare the generated code with the expected output and report any differences as an error.