FRUCTOSE – a C++ unit test framework

Andrew Peter Marlow
44 Stanhope Road, North Finchley
London N12 9DT, England
public@marlowa.plus.com

January 14, 2007

Contents

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. The purpose of this article is to describe the C++ techniques that were used to develop FRUCTOSE, why and how FRUCTOSE is different from other C++ unit test frameworks and the advantages this confers for Test Driven Development (TDD).

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.

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.

There are certain kinds of assertion tests that are useful for any unit test framework to offer. This article discusses those and how they are offered by FRUCTOSE. In particular, FRUCTOSE offers loop assertion macros. These are designed to be used in tests that use static tables for their test data. An assertion failure from test data needs to report not only the line of code that contains the assertion but also the line from the test data table.

FRUCTOSE has been developed as free software (Ref [15]). This meant chosing an appropriate license for it. The intent is to have a license that is non-

product specific with an element of copyleft but that still allows FRUCTOSE to be used in proprietary software. The license chosen was 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. In the search for a suitable license the LGPL was the winner by default. It is the only license listed on the Free Software Foundation's web site that is non-product specific and a GPL-compatible free software license that has some copyleft provision.

Chapter 2

Why another C++ unit test framework?

When I first started to look seriously at C++ unit testing I looked at CppUnit. Conventional wisdom said to build on the work of others rather than reinvent a framework, and here was an established one which was itself built on the work of JUnit. The feature list is impressive and it looked like it would be mature enough to give a trouble-free build and be usable right away. Unfortunately, this proved not to be the case. A quick search of sourceforge reveals that there are a few products for C++ unit testing. These also turn out to have issues of their own (discussed later). In October's issue of Overload there is an article by Peter Sommerlad on a C++ unit testing framework called CUTE (Ref [13]). This article also mentions that there are issues with using CppUnit and that some developers want something else that is smaller and simpler. However, at the time I started this article CUTE was not available online and Peter was too busy to work on a collaboration. This provided the motivation for me to write something which is driven by the same need expressed in the CUTE article (smaller and simpler than CppUnit). There are some important differences between FRUCTOSE and CUTE. FRUCTOSE avoids two significant dependencies; one on Boost and the other on platform-specfic RTTI.

Chapter 3

The FRUCTOSE approach

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 behavour 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.

3.1 The Curiously Recurring Template Pattern (CRTP)

Having said that FRUCTOSE is smaller and simpler than other frameworks, I have to confess that when one writes a test class that is to be used with FRUCTOSE, the the test class needs to inherit from a FRUCTOSE base class. Not all test frameworks require inheritance to be used but FRUCTOSE does. Also the style of inheritance employs a pattern that some people may not have seen before. It is known as the Curiously Recurring Template Pattern (CRTP) (ref [14]). CRTP is where one inherits from a template base class whose template parameter is the derived class. This section explains why.

There is a need for machinery that can add tests to a test suite and run the tests. This is all done by the test class inheriting 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. CRTP is used so that the base class can specify a function signature that uses the derived class.

Suppose our test class is called `simpletest`. Its declaration would start like this:

```
struct simpletest :
  public fructose::test_base<simpletest> {
     :
     :
```

Let's see how this works: `test_base` names its template parameter `test_container` (it is the class that contains the tests). `test_base` declares the typedef `test_case` expressed in terms of `test_container`:

```
typedef void (test_container::*test_case)
                        (const std::string&);
```

This enables it to declare `add_test` to take a parameter of type `test_case`. `test_base` maintains a map of test case function pointers, keyed by test name. The declaration for this map is:

```
std::map<std::string, test_case> m_tests;
```

The declaration of the `add_test` fuction is:

```
void add_test(const std::string& name,
              test_case the_test);
```

Here is an example of a complete test harness, showing how the use of CRTP means the test cases are defined in the test class and registered using `add_test`:

```
#include "fructose/test_base.h"
const int neighbour_of_the_beast = 668;
struct simpletest :
  public fructose::test_base<simpletest> {
    void beast(const std::string& test_name) {
      fructose_assert(neighbour_of_the_beast == 668)
    }
};

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

3.2 Named tests and command line options

The example in the previous section shows that tests are named when they are registered but the example does not actually make any practical use of this. Using the `run()` function above, all registered 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. This is done by using the overloaded function `int run(int argc, char* argv[]);`.

The Open Source library TCLAP is used to parse the command line (Ref [9]). The command line is considered to consist of optional parameters followed by an optional list of test names. During parsing, TCLAP 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.

3.3 To inherit or not to inherit

The authors of some test frameworks feel that the requirement for a test class to have to inherit from anything is unreasonable. It it said that such a requirement makes the test framework hard to use and causes undesirable coupling between the test class and the framework. FRUCTOSE has several things to say in response to this:

- Some test frameworks that employ inheritance may be hard to use (e.g. CppUnit) but it does not follow that inheritance is the cause. Some frameworks have just become large and complex during their evolution, and offer the developer and bewildering number of choices for the design of their test classes.

- CRTP can seem slightly daunting to those that have not seen it before. Howver, the use of CRTP by FRUCTOSE is quite simple and is there simply to enforce that the code that does the tests is in functions of the test class. If anyone knows of any other way to enforce this I would be most interested to hear from them.

- FRUCTOSE applications are intended to be run as command line programs with the ability to use various command line options that come as part of FRUCTOSE. The test class gets this capability by inheritance. If anyone knows of a better way to do this I would be most interested to hear from them.

- FRUCTOSE only makes a handful of functions available. There is `add_test` to register the tests and `run` to run them. The assertion testing macros (discussed later) rely on a function in the base class but that is an implementation detail that is of no concern to the programmer. Other FRUCTOSE functionality such as the command line options accessors is op-

tional. Hence, the amount of coupling between the test class and the FRUCTOSE base class is actually quite low.

- The FRUCTOSE assert macros include the test name in any assertion failure message because the function `get_test_name()` is available to any class that inherits from `test_base`. This is part of what enables FRUCTOSE to have named tests. Without using this technique it is hard to see how user-friendly test names can be registered and used by the framework. This was a difficulty mentioned in the CUTE article. CUTE overcame the problem by using a compiler-specific demangle routine. Other frameworks just don't allow the user to name the tests at all.

- Some of the test frameworks I have examined avoid the need for the test class to inherit from a base class by their assert macros expanding to a large volume of code. The sort of code these macros expand to is the kind that FRUCTOSE places in the base class. FRUCTOSE takes the view that in general macros should be avoided in C++. However, FRUCTOSE does use macros for its asserts. It does this for two reasons: first it needs to get the assertion expression as a complete string much as the C assert facility does; and second, it uses the `__FILE__` and `__LINE__` macros to report the filename and line number at which the assert occurs. The macro definitions are small.

- Because the test class inherits from a base class to get all the functionality required, 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 actually has two classes, `test_base` and `test_root`. The user sees the former since it must be inherited from but does not see the latter (it is an implementation detail). `test_base` inherits from `test_root`. This is a division of labour; `test_base` contains code that uses the template argument. `test_root` contains code that is not required to be in a template class. The reason for this is largely historical; during the early stages of FRUCTOSE design it came as a library that had to be built, then linked with. `test_root` was in the library whilst the template code was all in the headers and instantied at compile time. When FRUCTOSE changed to be implemented entirely in header files (inspired by the same approach in TCLAP), the separation of classes was retained. It still provides a distinction between code that is required to be template code and that which does not.

`test_root` contains the following:

- private data members and associated accessors for the flags read from the command line.

- A private data member and associated accessor for the name of the current test.

- A count of the number of assertion errors, plus an associated accessor and mutator.
- Functions that implement most of the assertion code.
- Default `setup` and `teardown` functions.

3.4 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. 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`.
2. 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.

The `fructose_assert` macro expands to a call to a function, `test_assert` which takes the boolean result of the assertion expression, the test name (obtained via the function call `get_test_name()`), the assertion expression as a string, the source filename and line number. Here is the definition:

```
#define fructose_assert(X) \
    { fructose::test_root::test_assert((X), \
      get_test_name(), #X, __FILE__, __LINE__);}
```

The `test_assert` function 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.

Chapter 4

The FRUCTOSE 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.

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

```
USAGE:

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


Where:

   -h,  --help
     -h[elp]

   -r,  --reverse
     -r[everse]

   -a,  --assert_fatal
     -a[ssert_fatal]

   -v,  --verbose
     -v[erbose]

   --,  --ignore_rest
     Ignores the rest of the labeled
        arguments following this flag.

   <testNameString>  (accepted multiple times)
     test names

   -verbose turns on extra trace for those
         tests that have made use of it.

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

   -help produces this help.

   -reverse reverses the sense of
        test assertions.

   It is only used to test the test framework
       itself.

   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
```

Chapter 5

How FRUCTOSE works

Consider the `simpletest` example in the previous section. The `simpletest` class is referred to as the test class. This provides all the functions that do the testing. It must inherit from `test_base` using the curiously recurring template pattern (CRTP). It does this for several reasons.

1. FRUCTOSE maintains a map of function pointers for when it has to invoke those functions. The function pointer type needs to be declared which means establishing a calling convention. FRUCTOSE takes the view that the function should be a member function of the test class whose return type is void and that takes the test name as a `const std::string` reference. This is enforced at compile time by use of CRTP, which allows `test_base` to declare the function pointer type using the name of the test class.

2. `test_base` makes the function `bool verbose() const` available, which returns `true` if the verbose flag was given on the command line.

3. FRUCTOSE avoids users having to know about several classes by providing the test registration function `add_test` and the test runner function `run`, via the base class. It also provides an overloaded `run` function that parses the command line and runs the tests specified. These functions are not only convenient, they ensure that the user of the framework does not have to worry about the existence of any classes other than the one he is testing and the test class he is testing it with.

It is felt by some that when a framework forces the test class to inherit from anything this is an unreasonable requirement.

The argument says that because inheritance is very strong coupling between classes this arrangement couples the test case too tightly to the framework. The trouble is, there has to be some coupling between the test class and the machinery that invokes its functions. If nothing else, the invoking machinery must establish a call convention for the functions that it calls.

FRUCTOSE uses a convention of the test function having the void return type and taking a const `std::string` reference to the test name, which must be called `test_name` (this is required by the FRUCTOSE test macros). As another example, CUTE requires that the test function have the void return type and have no function arguments in order that a boost functor may be implicitly created when a test function is passed to the CUTE macro. The CUTE approach does eliminate the coupling by inheritance but does so at the cost of not being able to explicitly name the test. Also, there is no particular advantage in allowing the test functions to be unrelated. In fact one could argue that they should be all grouped in the same test class on the principle of grouping related things together.

FRUCTOSE has the idea that tests are registered by name. The name has to be explicitly given in the `add_test` function. It is not deduced from the names or functions or the use of RTTI. This allows the test to be referred to from the command line. The `test_case` class provides the registration and command line parsing functions.

Chapter 6

FRUCTOSE coding techniques

FRUCTOSE is designed to work in commercial environments as well as open source environments. Commercial environments place some constraints on the C++ coding techniques that can be used. In the commercial world ancient compilers are still very much alive and well (for various reasons). Also, multiple operating systems have to be supported (Microsoft Windows and Solaris are probably the most important). This means that the lowest common denominator approach has to be taken for the dialect of C++ chosen. Platform-dependent RTTI, use of complex template meta-programming, and other advanced C++ techniques were avoided. So were dependencies upon packages that use such techniques.

FRUCTOSE achieves what it needs by simple inheritance. True, it uses CRTP, but the main reason for this is so the function pointers it maintains have to be functions that belong to the test class.

The STL is also used. Usage is kept very simple. Strings are always of type `std::string`. The function pointers to run are held in a `std::map`, keyed by test name string. The list of named tests to run is held in a `std::vector`. The exception handling macros also use the standard exception header `stdexcept` for things such as catching exceptions by the standard base class,
`std::exception`. The headers required for these uses of the STL and standard exceptions are automatically included by FRUCTOSE so there is no need for the test harness to include them again. All the functions are inlined, so there is no library to link against; just use the header files.

Chapter 7

Features

7.1 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_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.

7.2 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 `double`s.

7.3 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.

7.4 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);
}
```

7.5 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.

Chapter 8

The problems with CppUnit

When I first looked into providing a unit test environment for a commercial project I was working on, I was advised to look at Cppunit, so I did. I found that it would not build on the Solaris development environment we had (Forte 6.0). A port was in progress at the time but we needed something immediately. It was too much work to do a port ourselves when we were supposed to be using something off the shelf. We were also using a non-standard version of the STL and had to ensure that everything we built would build with that STL. The build procedure for CppUnit made this awkward. These problems are very specific to the development environment I was in. I made the recommendation that CppUnit not be used. Some of the reasons I gave were these environment-specific reasons but I also quoted the following reasons, which it seems have been the experience of others:

- Other people also find it hard to build. There is even a manual on the Wiki pages explaining how to build for various platforms. It should not be that complicated!
- CppUnit is very large for a unit test framework. The tarball is over 3MB once uncompressed. Admittedly, quite a bit of this is documentation and examples but no other C++ unit test framework I looked at was anywhere near this size. It takes quite a while to build it too (over four minutes on my dedicated Linux machine).
- It is hard to use. That's why there are lots of tutorials, lots of documentation and even a cookbook, discussion forums and an FAQ.
- CppUnit is too large and complex for many people's needs: I am sure that the reason for this volume of documentation is that CppUnit has alot of functionality to offer. However, in my opinion it does so at the cost of frightening off the developer that only needs something simple. The

number of C++ unit test frameworks that have sprung up, all with the goal of providing something smaller, cutdown and simpler are a testimony to the size and complexity of CppUnit.

- Even in the simple cases, CppUnit places too many complex requirements on the developer, particularly regarding new classes to be written and which base classes they are supposed to inherit from. In most of the alternatives to CppUnit there is only one class to inherit from (in the case of CUTE there is no need even for that). In CppUnit there is a choice of inheriting from `TestCase` or `TestFixture`. There are also `TestRunner`s, `TestCaller`s and `TestSuite`s to worry about.

Chapter 9

The problems with other frameworks

My search of sourceforge revealed several C++ unit test frameworks. However, the problem was usually that it was either for a specific environment or it did not allow the test selection and verbose flag setting via the command line that are so useful when doing TDD. The packages considered include the following:

- unit—- (ref[1]). The Unit test aid for C++. Judging from the examples and in the opinion of this author, unit—- is too cut-down, not providing much of the basic functionality provided by the other packages.
- csUnit (Ref [2]). It is for managed C++.
- Symbian OS C++ Unit Testing Framework (Ref [3]). It is only for the Symbian operating system,
- RapidoTest (Ref [4]). It is for Unix only with particular emphasis on Linux.
- Mock Objects for C++ (Ref [5]). It is a framework that builds on a framework; it provides mock objects by building on either CppUnit or cxxunit.
- UnitTest++ (Ref [6]). This actually comes quite close. It is much smaller and simpler than CppUnit and I did not encounter any build problems. It is multi-platform. However, there is no built-in control over which tests to run, neither can the test be identified during the run. This is fine for overnight regression testing but is not so good for TDD.
- QuickTest (Ref [7]). This was discovered after FRUCTOSE was released. QuickTest only consists of one very small header file with no documentation and no examples. It's approach has alot in common but FRUCTOSE is slightly richer in functionality, particularly with the test assertions than

can be made and the command line and named test features. Again, this makes it more suitable for TDD. FRUCTOSE also comes with documentation and a couple of examples.

- CppTest (Ref [8]). A slight wrinkle was found in the build procedure – doxygen is mandory otherwise the configure script will not produce a Makefile. Apart from that this package looked to be quite good, more mature and than QuickTest, also with better documentation. However, in common with QuickTest and UnitTest++ there does not seem to be a mechanism for selecting which tests to run, controlling verbosity and so on.

- Boost test. This was avoided because of the direct dependency on Boost. The test library has to be built in a similar way to Boost, which is well known for having a complex Unix-centric build procedure. Like other frameworks, Boost test has separate classes for tests and the ability to runs the tests. It lacks the ability to name the tests and run them selectively.

Chapter 10

The advantages of FRUCTOSE

The main strength of FRUCTOSE compared to the packages above, is its emphasis on its use during code development. This is via its features for reporting in detail the test that failed, the ability to supplement this trace with diagnostics controlled by the verbose flag on the command line, and the ability to select tests by name and optionally fail at the first test failure. None of the other packages provide this; their focus seems to be on running batches of testing in an overnight run to detect regressions. FRUCTOSE will also do that but provides the command line flexibility as well.

Another strength of FRUCTOSE is that it only has one external package dependency. It uses TCLAP (Ref [9]) for command line argument handling. This is a very small dependency, since TCLAP is quite small and is implemented entirely in header files. Some other frameworks have a larger set of dependency requirements and some of these dependencies are non-trivial. For example, some depend on Boost. Whilst Boost is recognised to be a fine set of high-quality libraries, some projects, particularly those in some commercial environments, do not want to depend on it. This is for several reasons:

1. Boost does not yet build out of the box in some commercial environments. This is the fault of the compilers, not the fault of Boost. But sometimes a commercial project has little choice of which compiler to use. This can be dictated by company policy, use of other closed-source third-party C++ libraries, and/or customer support obligations where the customer has an old compiler environment.

2. Boost is huge and complex. This turns off many projects/companies from looking at it and using it, even though there are many benefits.

3. If a project is already using Boost (and many are) then a unit test framework that also uses it is not a problem. But given the buildability issues with Boost and its size and complexity, some projects/companies would

be reluctant to be forced to use it just because the unit test framework requires it.

Chapter 11

Possible future work

FRUCTOSE deliberately does not provide any machinery for producing HTML test summaries, reports, or ways of running multiple test suites. Yet anything but the smallest projects will probably want this facility as part of the overnight build and test regime. One way to do this would be for FRUCTOSE to provide scripts, say in perl or python, that used some convention for naming and grouping the test harnesses so they can be run and the results organised into groups.

FRUCTOSE may provide some facility in the future to augment the command line options with additional options that a developer needs for their particular needs. For example, a harness that uses a database may wish to pass in the database parameters (database name, machine name, username, password). At the moment a FRUCTOSE harness would have to find some other way to receive these parameters.

FRUCTOSE does not provide any means to assess code coverage in its tests. There are separate tools to do this. For example, a FRUCTOSE test harness could be run with PureCoverage to assess how much code was exercised. Such tools do not often provide output or reports in way that lend themselves to brief reporting via such things as an overnight build and test run. One possible enhancement of FRUCTOSE would be to develop scripts that work with tools like PureCoverage to give a brief summary of which functions were called and which were not, and what the percentage code coverage was of the functions that were called.

Chapter 12

Conclusion

It is hoped that FRUCTOSE provides enough unit test machinery to enable projects to develop unit tests that can be used both in overnight regression tests and to help TDD. I welcome any feedback on the usefulness(or otherwise) of FRUCTOSE in other projects.

FRUCTOSE's simple implementation and cutdown approach mean that providing anything more complex will probably be permanently outside of the project's scope. However, this does not mean that FRUCTOSE is not open to changes. Any suggestions on how FRUCTOSE can be further simplified without reducing functionality will be gratefully received. It may be downloaded from sourceforge (Ref [12]).

Bibliography

[1] Unit—- at https://sourceforge.net/projects/unitmm

[2] csUnit at https://sourceforge.net/projects/csunit

[3] Symbian OS C++ Unit Testing Framework at https://sourceforge.net/projects/symbianosunit

[4] Rapido test at https://sourceforge.net/projects/rapidotest

[5] Mock Objects for C++ at https://sourceforge.net/projects/mockpp

[6] UnitTest++ at https://sourceforge.net/projects/unittest-cpp

[7] QuickTest at https://sourceforge.net/projects/quicktest

[8] CppTest at https://sourceforge.net/projects/cpptest

[9] TCLAP Templatised C++ Command Line Parser Library at http://tclap.sourceforge.net.

[10] The Art of Computer Programming, Volume 2, by Professor Don Knuth.

[11] Software Engineering; a practioners approach, by Roger Pressman (4th Edition). Section 16.6.2.

[12] FRUCTOSE at https://sourceforge.net/projects/fructose.

[13] Overload October 2006, CUTE.

[14] C++ Templates, The Complete Guide, section 16.3, by Vandevoorde and Josuttis.

[15] The Free Software Foundation, `http://www.fsf.org`