CxxTest User's Guide

Table of Contents

1 Introduction

CxxTest is a JUnit/CppUnit/xUnit-like framework for C++.

Its advantages over existing alternatives are that it:

- Doesn't require RTTI
- Doesn't require member template functions
- Doesn't require exception handling
- Doesn't require any external libraries (including memory management, file/console I/O, graphics libraries)

In other words, CxxTest is designed to be as portable as possible. Its only requirements are a reasonably modern C++ compiler and either Perl or Python. However, when advanced features are supported in your environment, CxxTest can use them, e.g. catch unhandled exceptions and even display a GUI.

In addition, CxxTest is slightly easier to use than the C++ alternatives, since you don't need to "register" your tests. It also features some extras like a richer set of assertions and even support for a "to do" list (see `TS_WARN()`).

CxxTest is available under the GNU Lesser General Public License.

1.1 About this guide

This guide is not intended as an introduction to Extreme Progamming and/or unit testing. It describes the design and usage of CxxTest.

2 Getting started

2.1 Getting CxxTest

The homepage for CxxTest is http://cxxtest.tigris.org. You can always get the latest release from the documents and files page, here. The latest version of this guide is available online at http://cxxtest.tigris.org/guide.html. A PDF version is also available at http://cxxtest.tigris.org/guide.pdf.

2.2 Your first test!

Here's a simple step-by-step guide:

1. Tests are organized into "Test Suites". Test suites are written in header files.

   A test suite is a class that inherits from `CxxTest::TestSuite`. A test is a public `void (void)` member function of that class whose name starts with `test`, e.g. `testDirectoryScanner()`, `test_cool_feature()` and even `TestImportantBugFix()`.

   ```
   // MyTestSuite.h
   #include <cxxtest/TestSuite.h>

   class MyTestSuite : public CxxTest::TestSuite
   {
   public:
      void testAddition( void )
      {
         TS_ASSERT( 1 + 1 > 1 );
         TS_ASSERT_EQUALS( 1 + 1, 2 );
      }
   };
   ```

2. After you have your test suites, you use CxxTest to generate a "test runner" source file:

   ```
   # cxxtestgen --error-printer -o runner.cpp MyTestSuite.h
   ```

   or, for those less fortunate:

   ```
   C:\tmp> python cxxtestgen --error-printer -o runner.cpp MyTestSuite.h
   ```

3. You would then simply compile the resulting file:

   ```
   # g++ -o runner runner.cpp
   ```

   or perhaps

   ```
   C:\tmp> cl -GX -o runner.exe runner.cpp
   ```

   or maybe even

   ```
   C:\tmp> bcc32 -erunner.exe runner.cpp
   ```

4. Finally, you run the tests and enjoy a well tested piece of software:

   ```
   # ./runner
   Running 1 test.OK!
   ```

2.3 Your second test

Now let's see what failed tests look like. We will add a failing test to the previous example:

```
// MyTestSuite.h
#include <cxxtest/TestSuite.h>

class MyTestSuite : public CxxTest::TestSuite
```

```
{
public:
   void testAddition( void )
   {
      TS_ASSERT( 1 + 1 > 1 );
      TS_ASSERT_EQUALS( 1 + 1, 2 );
   }

   void testMultiplication( void )
   {
      TS_ASSERT_EQUALS( 2 * 2, 5 );
   }
};
```

Generate, compile and run the test runner, and you will get this:

```
# ./runner
Running 2 tests.
MyTestSuite.h:15: Expected (2 * 2 == 5), found (4 != 5)
Failed 1 of 2 tests
Success rate: 50%
```

Fixing the bug is left as an excercise to the reader.

2.4 Graphical user interface

(v3.0.1) CxxTest can also display a simple GUI. The way to do this is depends on your compiler, OS and environment, but try the following pointers:

- Under Windows with Visual C++, run `python cxxtestgen -o runner.cpp --gui=Win32Gui MyTestSuite.h`.
- Under X-Windows, try `./cxxtestgen -o runner.cpp --gui=X11Gui MyTestSuite`. You may need to tell the compiler where to find X, usually something like `g++ -o runner -L/usr/X11R6/lib runner.cpp -lX11`.
- If you have Qt installed, try running `cxxtestgen` with the option `--gui=QtGui`. As always, compile and link the the Qt headers and libraries.

See Graphical user interface and Running the samples for more information.

3 *Really* using CxxTest

There is much more to CxxTest than seeing if two times two is four. You should probably take a look at the samples in the CxxTest distribution. Other than that, here are some more in-depth explanations.

3.1 What can you test

Here are the different "assertions" you can use in your tests:

Macro	Description	Example
`TS_FAIL(`*message*`)`	Fail unconditionally	`TS_FAIL("Test not implemented");`
`TS_ASSERT(`*expr*`)`	Verify *(expr)* is true	`TS_ASSERT(messageReceived());`
`TS_ASSERT_EQUALS(`*x, y*`)`	Verify *(x==y)*	`TS_ASSERT_EQUALS(nodeCount(), 14);`
`TS_ASSERT_SAME_DATA(`*x, y, size*`)`	Verify two buffers are equal	`TS_ASSERT_SAME_DATA(input, output,, size);`
`TS_ASSERT_DELTA(`*x, y, d*`)`	Verify *(x==y)* up to *d*	`TS_ASSERT_DELTA(sqrt(4.0), 2.0, 0.0001);`
`TS_ASSERT_DIFFERS(`*x, y*`)`	Verify *!(x==y)*	`TS_ASSERT_DIFFERS(exam.numTook(), exam.numPassed());`
`TS_ASSERT_LESS_THAN(`*x, y*`)`	Verify *(x<y)*	`TS_ASSERT_LESS_THAN(ship.speed(), SPEED_OF_LIGHT);`
`TS_ASSERT_LESS_THAN_EQUALS(`*x, y*`)`	Verify *(x<=y)*	`TS_ASSERT_LESS_THAN_EQUALS(requests, items);`
`TS_ASSERT_PREDICATE(`*R, x*`)`	Verify *P(x)*	`TS_ASSERT_PREDICATE(SeemsReasonable, salary);`
`TS_ASSERT_RELATION(`*R, x, y*`)`	Verify *x R y*	`TS_ASSERT_RELATION(std::greater, salary, average);`
`TS_ASSERT_THROWS(`*expr, type*`)`	Verify that *(expr)* throws a specific type of exception	`TS_ASSERT_THROWS(parse(file), Parser::ReadError);`
`TS_ASSERT_THROWS_EQUALS(`*expr, arg, x, y*`)`	Verify type and value of what *(expr)* throws	(See text)
`TS_ASSERT_THROWS_ASSERT(`*expr, arg, assertion*`)`	Verify type and value of what *(expr)* throws	(See text)
`TS_ASSERT_THROWS_ANYTHING(`*expr*`)`	Verify that *(expr)* throws an exception	`TS_ASSERT_THROWS_ANYTHING(buggy());`
`TS_ASSERT_THROWS_NOTHING(`*expr*`)`	Verify that *(expr)* doesn't throw anything	`TS_ASSERT_THROWS_NOTHING(robust());`

`TS_WARN(`*message*`)`	Print *message* as a warning	`TS_WARN("TODO: Check invalid parameters");`
`TS_TRACE(`*message*`)`	Print *message* as an informational message	`TS_TRACE(errno);`

3.1.1 TS_FAIL

`TS_FAIL` just fails the test. It is like an `assert(false)` with an error message. For example:

```
void testSomething( void )
{
   TS_FAIL( "I don't know how to test this!" );
}
```

3.1.2 TS_ASSERT

`TS_ASSERT` is the basic all-around tester. It works just like the well-respected `assert()` macro (which I sincerely hope you know and use!) An example:

```
void testSquare( void )
{
   MyFileLibrary::createEmptyFile( "test.bin" );
   TS_ASSERT( access( "test.bin", 0 ) == 0 );
}
```

3.1.3 TS_ASSERT_EQUALS

This is the second most useful tester. As the name hints, it is used to test if two values are equal.

```
void testSquare( void )
{
   TS_ASSERT_EQUALS( square(-5), 25 );
}
```

3.1.4 TS_ASSERT_SAME_DATA

(v3.5.1) This assertion is similar to `TS_ASSERT_EQUALS()`, except that it compares the contents of two buffers in memory. If the comparison fails, the standard runner dumps the contents of both buffers as hex values.

```
void testCopyMemory( void )
{
   char input[77], output[77];
   myCopyMemory( output, input, 77 );
   TS_ASSERT_SAME_DATA( input, output, 77 );
}
```

3.1.5 TS_ASSERT_DELTA

Similar to `TS_ASSERT_EQUALS()`, this macro verifies two values are equal up to a delta. This is basically used for floating-point values.

```
void testSquareRoot( void )
{
   TS_ASSERT_DELTA( squareRoot(4.0), 2.0, 0.00001 );
}
```

3.1.6 TS_ASSERT_DIFFERS

The opposite of `TS_ASSERT_EQUALS()`, this macro is used to assert that two values are not equal.

```
void testNumberGenerator( void )
{
   int first = generateNumber();
   int second = generateNumber();
   TS_ASSERT_DIFFERS( first, second );
}
```

3.1.7 TS_ASSERT_LESS_THAN

This macro asserts that the first operand is less than the second.

```
void testFindLargerNumber( void )
{
   TS_ASSERT_LESS_THAN( 23, findLargerNumber(23) );
}
```

3.1.8 TS_ASSERT_LESS_THAN_EQUALS

(v3.7.0) Not surprisingly, this macro asserts that the first operand is less than or equals the second.

```
void testBufferSize( void )
{
   TS_ASSERT_LESS_THAN_EQUALS( bufferSize(), MAX_BUFFER_SIZE );
}
```

3.1.9 TS_ASSERT_PREDICATE

(v3.8.2) This macro can be seen as a generalization of TS_ASSERT(). It takes as an argument the name of a class, similar to an STL `unary_function`, and evaluates `operator()`. The advantage this has over TS_ASSERT() is that you can see the failed value.

```
class IsPrime
{
public:
   bool operator()( unsigned ) const;
};

// ...

void testPrimeGenerator( void )
{
   TS_ASSERT_PREDICATE( IsPrime, generatePrime() );
}
```

3.1.10 TS_ASSERT_RELATION

(v3.8.0) Closely related to TS_ASSERT_PREDICATE(), this macro can be seen as a generalization of TS_ASSERT_EQUALS(), TS_ASSERT_DIFFERS(), TS_ASSERT_LESS_THAN() and TS_ASSERT_LESS_THAN_EQUALS(). It takes as an argument the name of a class, similar to an STL `binary_function`, and evaluates `operator()`. This can be used to very simply assert comparisons which are not covered by the builtin macros.

```
void testGreater( void )
{
   TS_ASSERT_RELATION( std::greater<int>, ticketsSold(), 1000 );
}
```

3.1.11 TS_ASSERT_THROWS and friends

These assertions are used to test whether an expression throws an exception. TS_ASSERT_THROWS is used when you want to verify the type of exception thrown, and TS_ASSERT_THROWS_ANYTHING is used to just make sure something is thrown. As you might have guessed, TS_ASSERT_THROWS_NOTHING asserts that nothing is thrown.

(v3.10.0) TS_ASSERT_THROWS_EQUALS checks the type of the exception as in TS_ASSERT_THROWS then allows you to compare two value (one of which will presumably be the caught object). TS_ASSERT_THROWS_ASSERT is the general case, and allows you to make any assertion about the

thrown value. These macros may seem a little complicated, but they can be very useful; see below for an example.

```
void testFunctionsWhichThrowExceptions( void )
{
   TS_ASSERT_THROWS_NOTHING( checkInput(1) );
   TS_ASSERT_THROWS( checkInput(-11), std::runtime_error );
   TS_ASSERT_THROWS_ANYTHING( thirdPartyFunction() );

   TS_ASSERT_THROWS_EQUALS( validate(), const std::exception &e,
                            e.what(), "Invalid value" );
   TS_ASSERT_THROWS_ASSERT( validate(), const Error &e,
                            TS_ASSERT_DIFFERS( e.code(), SUCCESS ) );
}
```

3.1.12 TS_TRACE and TS_WARN

(v3.0.1) TS_WARN just prints out a message, like the #warning preprocessor directive. I find it very useful for "to do" items. For example:

```
void testToDoList( void )
{
   TS_WARN( "TODO: Write some tests!" );
   TS_WARN( "TODO: Make $$$ fast!" );
}
```

In the GUI, TS_WARN sets the bar color to yellow (unless it was already red).

(v3.9.0) TS_TRACE is the same, except that it doesn't change the color of the progress bar.

3.1.13 The ETS_ macros

The TS_ macros mentioned above will catch exceptions thrown from tested code and fail the test, as if you called TS_FAIL(). Sometimes, however, you may want to catch the exception yourself; when you do, you can use the ETS_ versions of the macros.

```
void testInterestingThrower()
{
   // Normal way: if an exception is caught we can't examine it
   TS_ASSERT_EQUALS( foo(2), 4 );

   // More elaborate way:
   try { ETS_ASSERT_EQUALS( foo(2), 4 ); }
   catch( const BadFoo &e ) { TS_FAIL( e.bar() ); }
}
```

3.1.14 The TSM_ macros

Sometimes the default output generated by the ErrorPrinter doesn't give you enough information. This often happens when you move common test functionality to helper functions inside the test suite; when an assertion fails, you do not know its origin.

In the example below (which is the file 'sample/MessageTest.h' from the CxxTest distribution), we need the message feature to know which invocation of checkValue() failed:

```
class MessageTest : public CxxTest::TestSuite
{
public:
   void testValues()
   {
      checkValue( 0, "My hovercraft" );
      checkValue( 1, "is full" );
      checkValue( 2, "of eels" );
```

```
    }

    void checkValue( unsigned value, const char *message )
    {
        TSM_ASSERT( message, value );
        TSM_ASSERT_EQUALS( message, value, value * value );
    }
};
```

3.1.14.1 The ETSM_ macros

Note: As with normal asserts, all `TSM_` macros have their non-exception-safe counterparts, the `ETSM_` macros.

3.2 Running the samples

CxxTest comes with some samples in the '`sample/`' subdirectory of the distribution. If you look in that directory, you will see three Makefiles: '`Makefile.unix`', '`Makefile.msvc`' and '`Makefile.bcc32`' which are for Linux/Unix, MS Visual C++ and Borland C++, repectively. These files are provided as a starting point, and some options may need to be tweaked in them for your system.

If you are running under Windows, a good guess would be to run `nmake -fMakefile.msvc run_win32` (you may need to run `VCVARS32.BAT` first). Under Linux, `make -fMakefile.unix run_x11` should probably work.

3.3 Test fixtures

When you have several test cases for the same module, you often find that all of them start with more or less the same code—creating objects, files, inputs, etc. They may all have a common ending, too—cleaning up the mess you left.

You can (and should) put all this code in a common place by overriding the virtual functions `TestSuite::setUp()` and `TestSuite::tearDown()`. `setUp()` will then be called before each test, and `tearDown()` after each test.

```
class TestFileOps : public CxxTest::TestSuite
{
public:
    void setUp() { mkdir( "playground" ); }
    void tearDown() { system( "rm -Rf playground"); }

    void testCreateFile()
    {
        FileCreator fc( "playground" );
        fc.createFile( "test.bin" );
        TS_ASSERT_EQUALS( access( "playground/test.bin", 0 ), 0 );
    }
};
```

Note new users: This is probably the single most important feature to use when your tests become non-trivial.

3.3.1 Test suite level fixtures

`setUp()`/`tearDown()` are executed around each test case. If you need a fixture on the test suite level, i.e. something that gets constructed once before all the tests in the test suite are run, see Dynamically creating test suites below.

3.4 Integrating with your build environment

It's very hard to maintain your tests if you have to generate, compile and run the test runner manually all the time. Fortunately, that's why we have build tools!

3.4.1 Overview

Let's assume you're developing an application. What I usually do is the following:

- Split the application into a library and a main module that just calls the library classes. This way, the test runner will be able to access all your classes through the library.
- Create another application (or target, or project, or whatever) for the test runner. Make the build tool generate it automatically.
- For extra points, make the build tool run the tests automatically.

3.4.2 Actually doing it

Unfortunately, there are way too many different build tools and IDE's for me to give ways to use CxxTest with all of them.

I will try to outline the usage for some cases.

3.4.2.1 Using Makefiles

Generating the tests with a makefile is pretty straightforward. Simply add rules to generate, compile and run the test runner.

```
all: lib run_tests app

# Rules to build your targets
lib: ...

app: ...

# A rule that runs the unit tests
run_tests: runner
	./runner

# How to build the test runner
runner: runner.cpp lib
	g++ -o $@ $^

# How to generate the test runner
runner.cpp: SimpleTest.h ComplicatedTest.h
	 cxxtestgen -o $@ --error-printer $^
```

3.4.2.2 Using Cons

Cons is a powerful and versatile make replacement which uses Perl scripts instead of Makefiles.

See '`sample/Construct`' in the CxxTest distribution for an example of building CxxTest test runners with Cons.

3.4.2.3 Using Microsoft Visual Studio

I have tried several ways to integrate CxxTest with visual studio, none of which is perfect. Take a look at '`sample/msvc`' in the distribution to see the best solution I'm aware of. Basically, the workspace has three projects:

- The project `CxxTest_3_Generate` runs `cxxtestgen`.
- The project `CxxTest_2_Build` compiles the generated file.
- The project `CxxTest_1_Run` runs the tests.

This method certainly works, and the test results are conveniently displayed as compilation errors and warnings (for `TS_WARN()`). However, there are still a few things missing; to integrate this approach with your own project, you usually need to work a little bit and tweak some makefiles and project options. I have provided a small script in '`sample/msvc/FixFiles.bat`' to automate some of the process.

3.4.2.4 Using Microsoft Windows DDK

Unit testing for device drivers?! Why not? And besides, the '`build`' utility can also be used to build user-mode application.

To use CxxTest with the '`build`' utility, you add the generated tests file as an extra dependency using the `NTBUILDTARGET0` macro and the '`Makefile.inc`' file.

You can see an example of how to do this in the CxxTest distribution under '`sample/winddk`'.

3.5 Graphical user interface

There are currently three GUIs implemented: native Win32, native X11 and Qt. To use this feature, just specify '`--gui=X11Gui`', '`--gui=Win32Gui`' or '`--gui=QtGui`' as a parameter for '`cxxtestgen`' (instead of e.g. '`--error-printer`'). A progress bar is displayed, but the results are still written to standard output, where they can be processed by your IDE (e.g. Emacs or Visual Studio). The default behavior of the GUI is to close the window after the last test.

Note that whatevr GUI you use, you can combine it with the '`--runner`' option to control the formatting of the text output, e.g. Visual Studio likes it better if you use '`--runner=ParenPrinter`'.

3.5.1 Starting the GUI minimized

If you run the generated Win32 or Qt GUIs with the command line '`-minimized`', the test window will start minimized (iconified) and only pop up if there is an error (the bar turns red). This is useful if you find the progress bar distracting and only want to check it if something happens.

3.5.2 Leaving the GUI open

The Win32 GUI accepts the '`-keep`' which instructs it to leave the window open after the tests are done. This allows you to see how many tests failed and how much time it took.

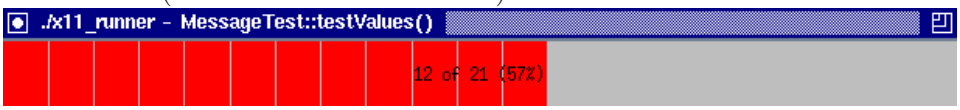
3.5.3 Screenshots!

As with any self-respecting GUI application, here are some screenshots for you to enjoy:

- Using the Qt GUI on Linux (with the WindowMaker window manager):

- Using the Win32 GUI on Windows 98:

- Using the X11 GUI (with the venerable TWM):



- And of course, no GUI is complete without the ability to mess around with its appearance:

Ahhh. Nothing like a beautiful user interface.

4 Advanced topics

Topics in this section are more technical, and you probably won't find them interesting unless you need them.

4.1 Aborting tests after failures

Usually, when a `TS_ASSERT_*` macro fails, CxxTest moves on to the next line. In many cases, however, this is not the desired behavior. Consider the following code:

```
void test_memset()
{
   char *buffer = new char[1024];
   TS_ASSERT( buffer );
   memset( buffer, 0, 1024 ); // But what if buffer == 0?
}
```

If you have exception handling enabled, you can make CxxTest exit each test as soon as a failure occurs. To do this, you need to define `CXXTEST_ABORT_TEST_ON_FAIL` before including the CxxTest headers. This can be done using the '`--abort-on-fail`' command-line option or in a template file; see '`sample/aborter.tpl`' in the distribution. Note that if CxxTest doesn't find evidence of exception handling when scanning your files, this feature will not work. To overcome this, use the '`--have-eh`' command-line option.

4.1.1 Controlling this behavior at runtime

(v3.8.5) In some scenarios, you may want some tests to abort on failed assertions and others to continue. To do this you use the '`--abort-on-fail`' option and call the function `CxxTest::setAbortTestOnFail( bool )` to change the runtime behavior. This flag is reset (normally, to `true`) after each test, but you can set it in your test suite's `setUp()` function to modify the behavior for all tests in a suite.

(v3.9.0) Note that this behavior is available whenever you have exception handling ('`--have-eh`' or `CXXTEST_HAVE_EH`); all '`--abort-on-fail`' does is set the default to `true`.

4.2 Commenting out tests

CxxTest does a very simple analysis of the input files, which is sufficient in most cases. This means, for example, that you can't indent you test code in "weird" ways.

A slight inconvenience arises, however, when you want to comment out tests. Commenting out the tests using C-style comments or the preprocessor will not work:

```
class MyTest : public CxxTest::TestSuite
{
public:
/*
   void testCommentedOutStillGetsCalled()
   {
   }
*/

#if 0
   void testMarkedOutStillGetsCalled()
   {
   }
#endif
};
```

(v3.10.0) If you need to comment out tests, use C++-style comments. Also, if you just don't want CxxTest to run a specific test function, you can temporarily change its name, e.g. by prefixing it with `x`:

```
class MyTest : public CxxTest::TestSuite
{
public:
// void testFutureStuff()
// {
// }

   void xtestFutureStuff()
   {
   }
};
```

4.3 Comparing equality for your own types

You may have noticed that `TS_ASSERT_EQUALS()` only works for built-in types. This is because CxxTest needs a way to compare object and to convert them to strings, in order to print them should the test fail.

If you do want to use `TS_ASSERT_EQUALS()` on your own data types, this is how you do it.

4.3.1 The equality operator

First of all, don't forget to implement the equality operator (`operator==()`) on your data types!

4.3.2 Value traits

Since CxxTest tries not to rely on any external library (including the standard library, which is not always available), conversion from arbitrary data types to strings is done using value traits.

For example, to convert an integer to a string, CxxTest does the following actions:

- `int i = value to convert;`
- `CxxTest::ValueTraits<int> converter(i);`
- `string = converter.asString();`

CxxTest comes with predefined `ValueTraits` for `int`, `char`, `dobule` etc. in '`cxxtest/ValueTraits.h`' in the '`cxxtest-selftest`' archive.

4.3.3 Unknown types

Obviously, CxxTest doesn't "know" about all possible types. The default ValueTraits class for unknown types dumps up to 8 bytes of the value in hex format.

For example, the following code

```
#include <cxxtest/TestSuite.h>

class TestMyData : public CxxTest::TestSuite
{
public:
   struct Data
   {
      char data[3];
   };

   void testCompareData()
```

```
   {
      Data x, y;
      memset( x.data, 0x12, sizeof(x.data) );
      memset( y.data, 0xF6, sizeof(y.data) );
      TS_ASSERT_EQUALS( x, y );
   }
};
```

would output

```
Running 1 test.
TestMyData.h:16: Expected (x == y), found ({ 12 12 12 } != { F6 F6 F6 })
Failed 1 of 1 test
Success rate: 0%
```

4.3.4 Enumeration traits

(v3.10.0) CxxTest provides a simple way to define value traits for your enumeration types, which is very handy for things like status codes. To do this, simply use `CXXTEST_VALUE_TRAITS` as in the following example:

```
enum Status { STATUS_IDLE, STATUS_BUSY, STATUS_ERROR };

CXXTEST_ENUM_TRAITS( Status,
                     CXXTEST_ENUM_MEMBER( STATUS_IDLE )
                     CXXTEST_ENUM_MEMBER( STATUS_BUSY )
                     CXXTEST_ENUM_MEMBER( STATUS_ERROR ) );
```

See '`sample/EnumTraits.h`' for a working sample.

4.3.5 Defining new value traits

Defining value traits for new (non-enumeration) types is easy. All you need is to define a way to convert an object of your class to a string. You can use this example as a possible skeleton:

```
class MyClass
{
   int _value;

public:
   MyClass( int value ) : _value( value ) {}
   int value() const { return _value; }

   // CxxTest requires a copy constructor
   MyClass( const MyClass &other ) : _value( other._value ) {}

   // If you want to use TS_ASSERT_EQUALS
   bool operator== ( const MyClass &other ) const { return _value == other._value; }

   // If you want to use TS_ASSERT_LESS_THAN
   bool operator== ( const MyClass &other ) const { return _value < other._value; }
};

#ifdef CXXTEST_RUNNING
#include <cxxtest/ValueTraits.h>
#include <stdio.h>

namespace CxxTest
{
   CXXTEST_TEMPLATE_INSTANTIATION
   class ValueTraits<MyClass>
   {
      char _s[256];

   public:
```

```
      ValueTraits( const MyClass &m ) { sprintf( _s, "MyClass( %i )", m.value() ); }
      const char *asString() const { return _s; }
   };
};
#endif // CXXTEST_RUNNING
```

4.3.5.1 Defining value traits for template classes

A simple modification to the above scheme allows you to define value traits for your template classes. Unfortunately, this syntax (partial template specialization) is not supported by some popular C++ compilers. Here is an example:

```
template<class T>
class TMyClass
{
   T _value;

public:
   TMyClass( const T &value ) : _value( value );
   const T &value() const { return _value; }

   // CxxTest requires a copy constructor
   TMyClass( const TMyClass<T> &other ) : _value( other._value ) {}

   // If you want to use TS_ASSERT_EQUALS
   bool operator== ( const TMyClass<T> &other ) const { return _value == other._value; }
};

#ifdef CXXTEST_RUNNING
#include <cxxtest/ValueTraits.h>
#include <typeinfo>
#include <sstream>

namespace CxxTest
{
   template<class T>
   class ValueTraits< TMyClass<T> >
   {
      std::ostringstream _s;

   public:
      ValueTraits( const TMyClass<T> &t )
         { _s << typeid(t).name() << "( " << t.value() << " )"; }
      const char *asString() const { return _s.str().c_str(); }
   };
};
#endif // CXXTEST_RUNNING
```

4.3.6 Overriding the default value traits

(v2.8.2) If you don't like the way CxxTest defines the default `ValueTrait`s, you can override them by `#define`-ing `CXXTEST_USER_VALUE_TRAITS`; this causes CxxTest to omit the default definitions, and from there on you are free to implement them as you like.

You can see a sample of this technique in '`test/UserTraits.tpl`' in the '`cxxtest-selftest`' archive.

4.4 Global Fixtures

(v3.5.1) The `setUp()` and `tearDown()` functions allow to to have code executed before and after each test. What if you want some code to be executed before all tests in *all* test suites? Rather

than duplicate that code, you can use *global fixtures*. These are basically classes that inherit from `CxxTest::GlobalFixture`. All objects of such classes are automatically notified before and after each test case. It is best to create them as static objects so they get called right from the start. Look at '`test/GlobalFixtures.h`' in the '`cxxtest-selftest`' archive.

Note: Unlike `setUp()` and `tearDown()` in `TestSuite`, global fixtures should return a `bool` value to indicate success/failure.

4.4.1 World fixtures

(v3.8.1) CxxTest also allows you to specify code which is executed once at the start of the testing process (and the corresponding cleanup code). To do this, create (one or more) global fixture objects and implement `setUpWorld()`/`tearDownWorld()`. For an example, see '`test/WorldFixtures.h`' in the '`cxxtest-selftest`' archive.

4.5 Mock Objects

(v3.10.0) Mock Objects are a very useful testing tool, which consists (in a nutshell) of passing special objects to tested code. For instance, to test a class that implements some protocol over TCP, you might have it use an abstract `ISocket` interface and in the tests pass it a `MockSocket` object. This `MockSocket` object can then do anything your tests find useful, e.g. keep a log of all data "sent" to verify later.

So far, so good. But the problem when developing in C/C++ is that your code probably needs to call *global* functions which you cannot override. Just consider any code which uses `fopen()`, `fwrite()` and `fclose()`. It is not very elegant to have this code actually create files while being tested. Even more importantly, you (should) want to test how the code behaves when "bad" things happen, say when `fopen()` fails. Although for some cases you can cause the effects to happen in the test code, this quickly becomes "hairy" and unmaintainable.

CxxTest solves this problem by allowing you to override any global function while testing. Here is an outline of how it works, before we see an actual example:

- For each function you want to override, you use the macro `CXXTEST_MOCK_GLOBAL` to "prepare" the function (all is explained below in excruciating detail).
- In the tested code you do not call the global functions directly; rather, you access them in the `T` (for *Test*) namespace. For instance, your code needs to call `T::fopen()` instead of `fopen()`. This is the equivalent of using abstract interfaces instead of concrete classes.
- You link the "real" binary with a source file that implements `T::fopen()` by simply calling the original `fopen()`.
- You link the test binary with a source file that implements `T::fopen()` by calling a mock object.
- To test, you should create a class that inherits `T::Base_fopen` and implement its `fopen()` function. Simply by creating an object of this class, calls made to `T::fopen()` will be redirected to it.

This may seem daunting at first, so let us work our way through a simple example. Say we want to override the well known standard library function `time()`.

- Prepare a header file to be used by both the real and test code.

```
// T/time.h
#include <time.h>
#include <cxxtest/Mock.h>

CXXTEST_MOCK_GLOBAL( time_t,        /* Return type          */
                     time,          /* Name of the function */
```

```
                                ( time_t *t ), /* Prototype            */
                                ( t )          /* Argument list        */ );
```

- In our tested code, we now include the special header instead of the system-supplied one, and call `T::time()` instead of `time()`.

```
// code.cpp
#include <T/time.h>

int generateRandomNumber()
{
    return T::time( NULL ) * 3;
}
```

- We also need to create a source file that implements `T::time()` by calling the real function. This is extremely easy: just define `CXXTEST_MOCK_REAL_SOURCE_FILE` before you include the header file:

```
// real_time.cpp
#define CXXTEST_MOCK_REAL_SOURCE_FILE
#include <T/time.h>
```

- Before we can start testing, we need a different implementation of `T::time()` for our tests. This is just as easy as the previous one:

```
// mock_time.cpp
#define CXXTEST_MOCK_TEST_SOURCE_FILE
#include <T/time.h>
```

- Now comes the fun part. In our test code, all we need to do is create a mock, and the tested code will magically call it:

```
// TestRandom.h
#include <cxxtest/TestSuite.h>
#include <T/time.h>

class TheTimeIsOne : public T::Base_time
{
public:
    time_t time( time_t * ) { return 1; }
};

class TestRandom : public CxxTest::TestSuite
{
public:
    void test_Random()
    {
        TheTimeIsOne t;
        TS_ASSERT_EQUALS( generateRandomNumber(), 3 );
    }
};
```

4.5.1 Actually doing it

I know that this might seem a bit heavy at first glance, but once you start using mock objects you will never go back. The hardest part may be getting this to work with your build system, which is why I have written a simple example much like this one in '`sample/mock`', which uses GNU Make and G++.

4.5.2 Advanced topic with mock functions

4.5.2.1 Void functions

Void function are a little different, and you use `CXXTEST_MOCK_VOID_GLOBAL` to override them. This is identical to `CXXTEST_MOCK_GLOBAL` except that it doesn't specify the return type. Take a look in '`sample/mock/T/stdlib.h`' for a demonstation.

4.5.2.2 Calling the real functions while testing

From time to time, you might want to let the tested code call the real functions (while being tested). To do this, you create a special mock object called e.g. `T::Real_time`. While an object of this class is present, calls to `T::time()` will be redirected to the real function.

4.5.2.3 When there is no real function

Sometimes your code needs to call functions which are not available when testing. This happens for example when you test driver code using a user-mode test runner, and you need to call kernel functions. You can use CxxTest's mock framework to provide testable implementations for the test code, while maintaing the original functions for the real code. This you do with `CXXTEST_SUPPLY_GLOBAL` (and `CXXTEST_SUPPLY_VOID_GLOBAL`). For example, say you want to supply your code with the Win32 kernel function `IoCallDriver`:

```
CXXTEST_SUPPLY_GLOBAL( NTSTATUS,                /* Return type */
                       IoCallDriver,            /* Name        */
                       ( PDEVICE_OBJECT Device, /* Prototype   */
                         PIRP Irp ),
                       ( Device, Irp )          /* How to call */ );
```

The tested code (your driver) can now call `IoCallDriver()` normally (no need for `T::`), and the test code uses `T::Base_IoCallDriver` as with normal mock objects.

Note: Since these macros can also be used to actually declare the function prototypes (e.g. in the above example you might not be able to include the real `<ntddk.h>` from test code), they also have an `extern "C"` version which declares the functions with `C` linkage. These are `CXXTEST_SUPPLY_GLOBAL_C` and `CXXTEST_SUPPLY_GLOBAL_VOID_C`.

4.5.2.4 Functions in namespaces

Sometimes the functions you want to override are not in the global namespace like `time()`: they may be global functions in other namespaces or even static class member functions. The default mock implementation isn't suitable for these. For them, you can use the generic `CXXTEST_MOCK`, which is best explained by example. Say you have a namespace `Files`, and you want to override the function `bool Files::FileExists( const String &name )`, so that the mock class will be called `T::Base_Files_FileExists` and the function to implement would be `fileExists`. You would define it thus (of course, you would normally want the mock class name and member function to be the same as the real function):

```
CXXTEST_MOCK( Files_FileExists,       /* Suffix of mock class  */
              bool,                   /* Return type           */
              fileExists,             /* Name of mock member   */
              ( const String &name ), /* Prototype             */
              Files::FileExists,      /* Name of real function */
              ( name )                /* Parameter list        */ );
```

Needless to say, there is also `CXXTEST_MOCK_VOID` for void functions.

There is also an equivalent version for `CXXTEST_SUPPLY_GLOBAL`, as demonstrated by another function from the Win32 DDK:

```
CXXTEST_SUPPLY( AllocateIrp,         /* => T::Base_AllocateIrp */
                PIRP,                /* Return type            */
                allocateIrp,         /* Name of mock member    */
                ( CCHAR StackSize ), /* Prototype              */
                IoAllocateIrp,       /* Name of real function  */
                ( StackSize )        /* Parameter list         */ );
```

And, with this macro you have `CXXTEST_SUPPLY_VOID` and of course `CXXTEST_SUPPLY_C` and `CXXTEST_SUPPLY_VOID_C`.

4.5.2.5 Overloaded functions

If you have two or more global functions which have the same name, you cannot create two mock classes with the same name. The solution is to use the general `CXXTEST_MOCK`/`CXXTEST_MOCK_VOID` as above: just give the two mock classes different names.

4.5.2.6 Changing the mock namespace

Finally, if you don't like or for some reason can't use the `T::` namespace for mock functions, you can change it by defining `CXXTEST_MOCK_NAMESPACE`. Have fun.

4.6 Test Listeners and Test Runners

A `TestListener` is a class that receives notifications about the testing process, notably which assertions failed. CxxTest defines a standard test listener class, `ErrorPrinter`, which is responsible for printing the dots and messages seen above. When the test runners generated in the examples run, they create an `ErrorPrinter` and pass it to `TestRunner::runAllTests()`. As you might have guessed, this functions runs all the test you've defined and reports to the `TestListener` it was passed.

4.6.1 Other test listeners

If you don't like or can't use the `ErrorPrinter`, you can use any other test listener. To do this you have to omit the '`--error-printer`', '`--runner=`' or '`--gui=`' switch when generating the tests file. It is then up to you to write the `main()` function, using the test listener of your fancy.

4.6.1.1 The `stdio` printer

If the `ErrorPrinter`'s usage of `std::cout` clashes with your environment or is unsupported by your compiler, don't dispair! You may still be able to use the `StdioPrinter`, which does the exact same thing but uses good old `printf()`.

To use it, invoke '`cxxtestgen`' with the '`--runner=StdioPrinter`' option.

(v3.8.5) **Note:** '`cxxtest/StdioPrinter`' makes reference to `stdout` as the default output stream. In some environments you may have `<stdio.h>` but not `stdout`, which will cause compiler errors. To overcome this problem, use '`--runner=StdioFilePrinter`', which is exactly the same as '`--runner=StdioPrinter`', but with no default output stream.

4.6.1.2 The Yes/No runner

As an example, CxxTest also provides the simplest possible test listener, one that just reports if there were any failures. You can see an example of using this listener in '`sample/yes_no_runner.cpp`'.

4.6.1.3 Template files

To use you own test runner, or to use the supplied ones in different ways, you can use CxxTest *template files*. These are ordinary source files with the embedded "command" `<CxxTest world>` which tells '`cxxtestgen`' to insert the world definition at that point. You then specify the template file using the '`--template`' option.

See '`samples/file_printer.tpl`' for an example.

Note: CxxTest needs to insert certain definitions and `#include` directives in the runner file. It normally does that before the first `#include <cxxtest/*.h>` found in the template file. If this behvaior is not what you need, use the "command" `<CxxTest preamble>`. See '`test/preamble.tpl`' in the '`cxxtest-selftest`' archive for an example of this.

4.7 Dynamically creating test suites

Usually, your test suites are instantiated statically in the tests file, i.e. say you defined `class MyTest : public CxxTest::TestSuite`, the generated file will contain something like `static MyTest g_MyTest;`.

If, however, your test suite must be created dynamically (it may need a constructor, for instance), CxxTest doesn't know how to create it unless you tell it how. You do this by writing two static functions, `createSuite()` and `destroySuite()`.

See '`sample/CreatedTest.h`' for a demonstration.

4.8 Static initialization

(v3.9.0) The generated runner source file depends quite heavily on static initialization of the various "description" object used to run your tests. If your compiler/linker has a problem with this approach, use the '`--no-static-init`' option.

Appendix A Command line options

Here are the different command line options for `cxxtestgen`:

A.1 '`--version`'

(v3.7.1) Specify '`--version`' or '`-v`' to see the version of CxxTest you are using.

A.2 '`--output`'

Specify '`--output=FILE`' or '`-o FILE`' to determine the output file name.

A.3 '`--error-printer`'

This option creates a test runner which uses the standard error printer class.

A.4 '`--runner`'

Specify '`--runner=CLASS`' to generate a test runner that `#include`s `<cxxtest/CLASS.h>` and uses `CxxTest::CLASS` as the test runner.

The currently available runners are:

'`--runner=ErrorPrinter`'
: This is the standard error printer, which formats its output to `std::cout`.

'`--runner=ParenPrinter`'
: Identical to `ErrorPrinter` except that it prints line numbers in parantheses. This is the way Visual Studio expects it.

'`--runner=StdioPrinter`'
: The same as `ErrorPrinter` except that it uses `printf` instead of `cout`.

'`--runner=YesNoRunner`'
: This runner doesn't produce any output, merely returns a true/false result.

A.5 '`--gui`'

Specify '`--gui=CLASS`' to generate a test runner that `#include`s `<cxxtest/CLASS.h>` and uses `CxxTest::CLASS` to display a graphical user interface. This option can be combined with the '`--runner`' option to determine the text-mode output format. The default is the standard error printer.

There are three different GUIs:

'`--gui=Win32Gui`'
: A native Win32 GUI. It has been tested on Windows 98, 2000 and XP and should work unmodified on other 32-bit versions of Windows.

'`--gui=X11Gui`'
: A native XLib GUI. This GUI is very spartan and should work on any X server.

'`--gui=QtGui`'
: A GUI that uses the Qt library from Troll. It has been tested with Qt versiond 2.2.1 and 3.0.1.

A.6 '--include'

(v3.5.1) If you specify '`--include=FILE`', `cxxtestgen` will add `#include "FILE"` to the runner before including any other header. This allows you to define things that modify the behavior of CxxTest, e.g. your own ValueTraits.

Note: If you want the runner to `#inculde <FILE>`, specify it on the command line, e.g. '`--include=<FILE>`'. You will most likely need to use shell escapes, e.g. '`"--include=<FILE>"`' or '`--include=\<FILE\>`'.

Examples: '`--include=TestDefs.h`' or '`--include=\<GlobalDefs.h\>`'.

A.7 '--template'

Specify '`--template=FILE`' to use '`FILE`' as a template file. This is for cases for which '`--runner`' and/or '`--include`' are not enough. One example is the Windows DDK; see '`sample/winddk`' in the distribution.

A.8 '--have-eh'

(v2.8.4) `cxxtestgen` will scan its input files for uses of exception handling; if found, the `TS_` macros will catch exceptions, allowing the testing to continue. Use '`--have-eh`' to tell `cxxtestgen` to enable that functionality even if exceptions are not used in the input files.

A.9 '--no-eh'

(v3.8.5) If you want `cxxtestgen` to ignore what may look as uses of exception handling in your test files, specify '`--no-eh`'.

A.10 '--have-std'

(v3.10.0) Same as '`--have-eh`' but for the standard library; basically, if you use this flag, CxxTest will print the values of `std::string`.

Note: If you reference the standard library anywhere in your test files, CxxTest will (usually) recognize it and automatically define this.

A.11 '--no-std'

(v3.10.0) The counterpart to '`--have-std`', this tells CxxTest to ignore any evidence it finds for the `std::` namespace in your code. Use it if your environment does not support `std::` but `cxxtestgen` thinks it does.

A.12 '--longlong'

(v3.6.0) Specify '`--longlong=TYPE`' to have CxxTest recognize `TYPE` as "long long" (e.g. '`--longlong=__int64`'). If you specify just '`--longlong=`' (no type), CxxTest will use the default type name of `long long`.

A.13 '--abort-on-fail'

(v2.8.2) This useful option tells CxxTest to abort the current test when any `TS_ASSERT` macro has failed.

A.14 '--part'

(v3.5.1) This option tells CxxTest not to write the CxxTest globals in the output file. Use this to link together more than one generated file.

A.15 '--root'

(v3.5.1) This is the counterpart of '`--part`'; it makes sure that the Cxxtest globals are written to the output file. If you specify this option, you can use `cxxtestgen` without any input files to create a file that hold only the "root" runner.

A.16 '--no-static-init'

(v3.9.0) Use this option if you encounter problems with the static initializations in the test runner.

Appendix B Controlling the behavior of CxxTest

Here are various `#define`s you can use to modify how CxxTest works. You will need to `#define` them *before* including any of the CxxTest headers, so use them in a template file or with the '`--include`' option.

B.1 CXXTEST_HAVE_STD

This is equivalent to the '`--have-std`' option.

B.2 CXXTEST_HAVE_EH

This is equivalent to the '`--have-eh`' option.

B.3 CXXTEST_ABORT_TEST_ON_FAIL

(v2.8.0) This is equivalent to the '`--abort-on-fail`' option.

B.4 CXXTEST_USER_VALUE_TRAITS

This tells CxxTest you wish to define you own ValueTraits. It will only declare the default traits, which dump up to 8 bytes of the data as hex values.

B.5 CXXTEST_OLD_TEMPLATE_SYNTAX

Some compilers (e.g. Borland C++ 5) don't support the standard way of instantiating template classes. Use this define to overcome the problem.

B.6 CXXTEST_OLD_STD

Again, this is used to support pre-`std::` standard libraries.

B.7 CXXTEST_MAX_DUMP_SIZE

This sets the standard maximum number of bytes to dump if `TS_ASSERT_SAME_DATA()` fails. The default is 0, meaning no limit.

B.8 CXXTEST_DEFAULT_ABORT

This sets the default value of the dynamic "abort on fail" flag. Of course, this flag is only used when "abort on fail" is enabled.

B.9 CXXTEST_LONGLONG

This is equivalent to '`--longlong`'.

Appendix C Runtime options

The following functions can be called during runtime (i.e. from your tests) to control the behavior of CxxTest. They are reset to their default values after each test is executed (more precisely, after `tearDown()` is called). Consequently, if you set them in the `setUp()` function, they will be valid for the entire test suite.

C.1 `setAbortTestOnFail( bool )`

This only works when you have exception handling. It can be used to tell CxxTest to temporarily change its behavior. The default value of the flag is `false`, `true` if you set '`--abort-on-fail`', or `CXXTEST_DEFAULT_ABORT` if you `#define` it.

C.2 `setMaxDumpSize( unsigned )`

This temporarily sets the maximum number of bytes to dump if `TS_ASSERT_SAME_DATA()` fails. The default is 0, meaning no limit, or `CXXTEST_MAX_DUMP_SIZE` if you `#define` it.

Appendix D Version history

- **Version 3.10.1 (2004-12-01)**
  - Improved support for VC7
  - Fixed clash with some versions of STL
- **Version 3.10.0 (2004-11-20)**
  - Added mock framework for global functions
  - Added `TS_ASSERT_THROWS_ASSERT` and `TS_ASSERT_THROWS_EQUALS`
  - Added `CXXTEST_ENUM_TRAITS`
  - Improved support for STL classes (vector, map etc.)
  - Added support for Digital Mars compiler
  - Reduced root/part compilation time and binary size
  - Support C++-style commenting of tests
- **Version 3.9.1 (2004-01-19)**
  - Fixed small bug with runner exit code
  - Embedded test suites are now deprecated
- **Version 3.9.0 (2004-01-17)**
  - Added `TS_TRACE`
  - Added '`--no-static-init`'
  - CxxTest::`setAbortTestOnFail()` works even without '`--abort-on-fail`'
- **Version 3.8.5 (2004-01-08)**
  - Added '`--no-eh`'
  - Added `CxxTest::setAbortTestOnFail()` and `CXXTEST_DEFAULT_ABORT`
  - Added `CxxTest::setMaxDumpSize()`
  - Added StdioFilePrinter
- **Version 3.8.4 (2003-12-31)**
  - Split distribution into cxxtest and cxxtest-selftest
  - Added '`sample/msvc/FixFiles.bat`'
- **Version 3.8.3 (2003-12-24)**
  - Added `TS_ASSERT_PREDICATE`
  - Template files can now specify where to insert the preamble
  - Added a sample Visual Studio workspace in '`sample/msvc`'
  - Can compile in MSVC with warning level 4
  - Changed output format slightly
- **Version 3.8.1 (2003-12-21)**
  - Fixed small bug when using multiple '`--part`' files.
  - Fixed X11 GUI crash when there's no X server.
  - Added `GlobalFixture::setUpWorld()`/`tearDownWorld()`
  - Added `leaveOnly()`, `activateAllTests()` and '`sample/only.tpl`'
  - Should now run without warnings on Sun compiler.
- **Version 3.8.0 (2003-12-13)**
  - Fixed bug where '`Root.cpp`' needed exception handling
  - Added `TS_ASSERT_RELATION`

  - `TSM_` macros now also tell you what went wrong
  - Renamed `Win32Gui::free()` to avoid clashes
  - Now compatible with more versions of Borland compiler
  - Improved the documentation
- **Version 3.7.1 (2003-09-29)**
  - Added '`--version`'
  - Compiles with even more exotic g++ warnings
  - Win32 Gui compiles with UNICODE
  - Should compile on some more platforms (Sun Forte, HP aCC)
- **Version 3.7.0 (2003-09-20)**
  - Added `TS_ASSERT_LESS_THAN_EQUALS`
  - Minor cleanups
- **Version 3.6.1 (2003-09-15)**
  - Improved QT GUI
  - Improved portability some more
- **Version 3.6.0 (2003-09-04)**
  - Added '`--longlong`'
  - Some portability improvements
- **Version 3.5.1 (2003-09-03)**
  - Major internal rewrite of macros
  - Added `TS_ASSERT_SAME_DATA`
  - Added '`--include`' option
  - Added '`--part`' and '`--root`' to enable splitting the test runner
  - Added global fixtures
  - Enhanced Win32 GUI with timers, '`-keep`' and '`-title`'
  - Now compiles with strict warnings
- **Version 3.1.1 (2003-08-27)**
  - Fixed small bug in `TS_ASSERT_THROWS_*()`
- **Version 3.1.0 (2003-08-23)**
  - Default ValueTraits now dumps value as hex bytes
  - Fixed double invocation bug (e.g. `TS_FAIL(functionWithSideEffects())`)
  - `TS_ASSERT_THROWS*()` are now "abort on fail"-friendly
  - Win32 GUI now supports Windows 98 and doesn't need comctl32.lib
- **Version 3.0.1 (2003-08-07)**
  - Added simple GUI for X11, Win32 and Qt
  - Added `TS_WARN()` macro
  - Removed '`--exit-code`'
  - Improved samples
  - Improved support for older (pre-std::) compilers
  - Made a PDF version of the User's Guide
- **Version 2.8.4 (2003-07-21)**
  - Now supports g++-3.3
  - Added '`--have-eh`'

  - Fixed bug in `numberToString()`
- **Version 2.8.3 (2003-06-30)**
  - Fixed bugs in cxxtestgen.pl
  - Fixed warning for some compilers in ErrorPrinter/StdioPrinter
  - Thanks Martin Jost for pointing out these problems!
- **Version 2.8.2 (2003-06-10)**
  - Fixed bug when using `CXXTEST_ABORT_TEST_ON_FAIL` without standard library
  - Added `CXXTEST_USER_TRAITS`
  - Added '`--abort-on-fail`'
- **Version 2.8.1 (2003-01-16)**
  - Fixed `charToString()` for negative chars
- **Version 2.8.0 (2003-01-13)**
  - Added `CXXTEST_ABORT_TEST_ON_FAIL` for xUnit-like behaviour
  - Added '`sample/winddk`'
  - Improved ValueTraits
  - Improved output formatter
  - Started version history
- **Version 2.7.0 (2002-09-29)**
  - Added embedded test suites
  - Major internal improvements