CxxTest User Guide

COLLABORATORS			
	TITLE : CxxTest User Guide		
ACTION	*NAME*	*DATE*	*SIGNATURE*
WRITTEN BY		December 23, 2011	

Contents

Abstract

CxxTest is a unit testing framework for C++ that is similar in spirit to JUnit, CppUnit, and xUnit. CxxTest is easy to use because it does not require precompiling a CxxTest testing library, it employs no advanced features of C++ (e.g. RTTI) and it supports a very flexible form of test discovery. This documentation describes CxxTest 4.0, which includes significant enhancements to the test discovery process, a modern test driver, and new documentation.

1 Overview

CxxTest is a unit testing framework for C++ that is similar in spirit to JUnit, CppUnit, and xUnit. CxxTest is designed to be as portable as possible; it does not require

- RTTI
- Member template functions
- Exception handling
- External libraries (including memory management, file/console I/O, graphics libraries)

In particular, the design of CxxTest was tailored for C++ compilers on embedded systems, for which many of these features are not supported. However, CxxTest can also leverage standard C++ features when they are supported by a compiler (e.g. catch unhandled exceptions).

Additionally, CxxTest supports *test discovery*. Tests are defined in C++ header files, which are parsed by CxxTest to automatically generate a test runner. Thus, CxxTest is somewhat easier to use than alternative C++ testing frameworks, since you do not need to *register* tests.

CxxTest is available under the GNU Lesser General Public license. The following are key online resources for CxxTest:

- CxxTest home page
- Release downloads
- HTML user guide
- PDF user guide

The CxxTest User Guide provides the following documentation:

- Getting Started: Some simple examples that illustrate how to use CxxTest
- Test Assertions: The test assertions supported by CxxTest
- The CxxTestGen Command: Documentation for the `cxxtestgen` command
- Test Runner Syntax: Discussion of command line options for test runners
- Advanced Testing Features: Advanced features of CxxTest
- Value Traits: Customizing data traits for error messages
- Testing with Mock Objects: How to test with mock global functions
- Installation: How to install CxxTest
- Status and Future Plans: Comments on the past, present and future of CxxTest

2 Getting Started

Testing is performed with CxxTest in a four-step process:

1. Tests are defined in C++ header files
2. The `cxxtestgen` command processes header files to generate files for the test runner.
3. Compile the test runner.

4. Execute the test runner to run all test suites.

CxxTest supports test automation, sharing of setup and shutdown code for tests, aggregation of tests into collections, and independence of the tests from the reporting framework. To achieve this, CxxTest supports some important concepts that are common to xUnit frameworks (e.g. JUnit, CppUnit, and xUnit):

test fixture
: A *test fixture* represents the preparation needed to perform one or more tests, and any associate cleanup actions. This may involve, for example, creating temporary or proxy databases, directories, or starting a server process.

test suite
: A *test suite* is a collection of test cases, which represent the smallest unit of testing. A test suite is defined by a class that inherits from the `CxxTest::TestSuite` class, and the tests in a test suite are executed together.

test
: A test is a public member function of a test suite whose name starts with `test`, e.g. `testDirectoryScanner()`, `test_cool_feature()` and `TestImportantBugFix()`.

test runner
: A *test runner* is a component which orchestrates the execution of tests across one or more test suites and provides the outcome to the user.

When building test fixtures using `TestSuite`, the `TestSuite.setUp` and `TestSuite.tearDown` methods can be overridden to provide initialization and cleanup for the fixture. The `TestSuite.setUp` method is run before each test is executed, and the `TestSuite.tearDown` method is run after each test is executed.

2.1 A First Example

The following is a simple example of a test suite with a single test, `testAddition`, which perform two test assertions:

```
// MyTestSuite1.h
#include <cxxtest/TestSuite.h>

class MyTestSuite1 : public CxxTest::TestSuite
{
public:
    void testAddition(void)
    {
        TS_ASSERT(1 + 1 > 1);
        TS_ASSERT_EQUALS(1 + 1, 2);
    }
};
```

You use the `cxxtestgen` script to generate a *test runner* for test suites in C++ header files:

```
cxxtestgen --error-printer -o runner.cpp MyTestSuite1.h
```

This command generates the file `runner.cpp`, which can be compiled.

```
cxxtestgen --error-printer -o runner.cpp MyTestSuite1.h
```

Note that additional compiler flags may be needed to include headers and libraries that are used during testing.

This runner can be executed to perform the specified tests:

```
./runner
```

which generates the following output:

```
Running 1 test.OK!
```

2.2 A Second Example

The following header file extends the previous example to include a test that generates an error:

```
// MyTestSuite2.h
#include <cxxtest/TestSuite.h>

class MyTestSuite2 : public CxxTest::TestSuite
{
public:
    void testAddition(void)
    {
        TS_ASSERT(1 + 1 > 1);
        TS_ASSERT_EQUALS(1 + 1, 2);
    }

    void testMultiplication(void)
    {
        TS_TRACE("Starting multiplication test");
        TS_ASSERT_EQUALS(2 * 2, 5);
        TS_TRACE("Finishing multiplication test");
    }
};
```

The test runner generated by `cxxtestgen` for this test suite generates the following output:

```
Running 2 tests.
In MyTestSuite::testMultiplication:
MyTestSuite2.h:15: Error: Expected (2 * 2 == 5), found (4 != 5)
Failed 1 of 2 tests
Success rate: 50%
```

2.3 Sample Problems

CxxTest comes with example test suites in the `cxxtest/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.

3 Test Assertions

The following table summarizes the test assertions supported by CxxTest. Appendix A provides examples that illustrate the use of these test assertions.

Macro	Description
TS_ASSERT(expr)	Verify `expr` is true
TS_ASSERT_DELTA(x,y,d)	Verify that `abs(x-y) < d`
TS_ASSERT_DIFFERS(x,y)	Verify that `x != y`
TS_ASSERT_EQUALS(x,y)	Verify that `x == y`
TS_ASSERT_LESS_THAN(x,y)	Verify that `x < y`
TS_ASSERT_LESS_THAN_EQUALS(x,y)	Verify that `x <= y`
TS_ASSERT_PREDICATE(P,x)	Verify `P(x)`
TS_ASSERT_RELATION(x,R,y)	Verify `x R y`
TS_ASSERT_SAME_DATA(x,y,size)	Verify two buffers are equal
TS_ASSERT_THROWS(expr,type)	Verify that `expr` throws the specified exception type
TS_ASSERT_THROWS_ANYTHING(expr)	Verify that `expr` throws an exception

Macro	Description
TS_ASSERT_THROWS_ASSERT(expr,arg,assertion)	Verify type and value of what `expr` throws
TS_ASSERT_THROWS_EQUALS(expr,arg,x,y)	Verify type and value of what `expr` throws
TS_ASSERT_THROWS_NOTHING(expr)	Verify that `expr` doesn't throw anything
TS_FAIL(message)	Fail unconditionally
TS_TRACE(message)	Print `message` as an informational message
TS_WARN(message)	Print `message` as a warning

The test assertions supported by CxxTest are defined as macros, which eliminates the need for certain templates within CxxTest and allows tests to catch exceptions. There are four categories of test assertions in CxxTest, which are distinguished by their prefixes:

TS_
: These test assertions perform a test. Catch exceptions generated during testing will cause the test to fail, except for tests that check for exceptions.

TSM_
: These test assertions perform the same tests as the corresponding `TS` assertions, but their first argument is a `const char*` message buffer that is printed when the test fails.

ETS_
: These test assertions perform the same tests as the corresponding `TS` assertions. However, these test assertions do not catch exceptions generated during testing.

ETSM_
: These test assertions perform the same tests as the corresponding `TS` assertions, but (1) their first argument is a `const char*` message buffer is printed when the test fails, and (2) these assertions do not catch exceptions generated during testing.

4 The CxxTestGen Command

The `cxxtestgen` command processes one or more C++ header files to generate a test runner. The `cxxtestgen` command performs test discovery by parsing the header files to find test classes, which inherit from the class `CxxTest::TestSuite`.

The `--help` option generates the following summary of the `cxxtestgen` command line options:

```
Usage: cxxtestgen [options] <filename> [<filename> ...]

Options:
  -h, --help            show this help message and exit
  --version             Write the CxxTest version.
  -o NAME, --output=NAME
                        Write output to file NAME.
  -w WORLD, --world=WORLD
                        The label of the tests, used to name the XML results.
  --include=HEADER      Include file HEADER in the test runner before other
                        headers.
  --abort-on-fail       Abort tests on failed asserts (like xUnit).
  --runner=CLASS        Create a test runner that processes test events using
                        the class CxxTest::CLASS.
  --gui=CLASS           Create a GUI test runner that processes test events
                        using the class CxxTest::CLASS. (deprecated)
  --error-printer       Create a test runner using the ErrorPrinter class, and
                        allow the use of the standard library.
  --xunit-printer       Create a test runner using the XUnitPrinter class.
  --xunit-file=XUNIT_FILE
                        The file to which the XML summary is written for test
                        runners using the XUnitPrinter class.  The default XML
```

```
                      filename is TEST-<world>.xml, where <world> is the
                      value of the --world option.  (default: cxxtest)
  --have-std          Use the standard library (even if not found in tests).
  --no-std            Do not use standard library (even if found in tests).
  --have-eh           Use exception handling (even if not found in tests).
  --no-eh             Do not use exception handling (even if found in
                      tests).
  --longlong=TYPE     Use TYPE as for long long integers.  (default: not
                      supported)
  --no-static-init    Do not rely on static initialization in the test
                      runner.
  --template=TEMPLATE Generate the test runner using file TEMPLATE to define
                      a template.
  --root              Write the main() function and global data for a test
                      runner.
  --part              Write the tester classes for a test runner.
  --factor            Declare the _CXXTEST_FACTOR macro.  (deprecated)
  -f, --fog-parser    Use new FOG C++ parser
```

The following section describe illustrate the use of these command line options.

4.1 General Options

The default behavior of `cxxtestgen` is to send the source for the test runner to the standard output stream. The `--output` (`-o`) option indicates a filename for the test runner.

The `--world` (`-w`) option specifies the value of the `CxxTest::RealWorldDescription::_worldName` variable. This option also customizes the filename used for XML output files (see below).

The `--include` option defines a filename that is included in the runner before all other headers.

The `--abort-on-fail` option forces an abort if a test fails, rather than continuing execution to the next test.

4.2 Test Listener Options

The test runner behavior is controlled by a *test listener* class that is used to define to the `main` function. The test listener class is a subclass of `TestListener` that receives notifications about the testing process, notably which assertions failed. The `--runner` option is used to specify the test listener that is used in the test runner. The following test listeners are defined in CxxTest:

`ErrorPrinter`
: This is the standard error printer, which formats its output to the standard output stream (`std::cout`).

`StdioPrinter`
: The same as `ErrorPrinter` except that it uses `printf` instead of `std::cout`.

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

`XmlPrinter`
: Print test results to an XML file.

`XUnitPrinter`
: This test listener generates output using both `ErrorPrinter` and `XmlPrinter`.

4.2.1 ErrorPrinter

The `--error-printer` option creates a runner using the `ErrorPrinter` test listener, and it indicates that the standard library is used in the test runner. The `ErrorPrinter` test listener prints dots to summarize test execution, along with a summary of the test results. For example, the command

```
cxxtestgen --error-printer -o runner.cpp MyTestSuite2.h
```

generates the following output:

```
Running 2 tests.
In MyTestSuite::testMultiplication:
MyTestSuite2.h:15: Error: Expected (2 * 2 == 5), found (4 != 5)
Failed 1 of 2 tests
Success rate: 50%
```

4.2.2 StdioPrinter

If your compiler does not support `std::cout`, then the `ErrorPrinter` test listener cannot be used. In this case, the `StdioPrinter` test listener can be used; it provides the same output as `ErrorPrinter` but it uses the `printf` function. For example, the command line:

```
cxxtestgen --runner=StdioPrinter -o runner.cpp MyTestSuite2.h
```

generates the following output:

```
Running 2 tests.
In MyTestSuite::testMultiplication:
MyTestSuite2.h:15: Error: Expected (2 * 2 == 5), found (4 != 5)
Failed 1 of 2 tests
Success rate: 50%
```

4.2.3 ParenPrinter

The `--runner=ParenPrinter` option creates a similar test runner:

```
cxxtestgen --runner=ParenPrinter -o runner.cpp MyTestSuite2.h
```

This test runner generates output that is similar to the `ErrorPrinter` test listener:

```
Running 2 tests.
In MyTestSuite::testMultiplication:
MyTestSuite2.h(15): Error: Expected (2 * 2 == 5), found (4 != 5)
Failed 1 of 2 tests
Success rate: 50%
```

The only difference is the parentheses used in the output. This test listener provides a format that can be recognized by Visual Studio.

4.2.4 XmlPrinter

The `--runner=XmlPrinter` option creates a test runner whose output is an XML summary of the test results. For example, the command:

```
cxxtestgen --runner=XmlPrinter -o runner.cpp MyTestSuite2.h
```

generates the following output:

```
<?xml version="1.0" encoding="UTF-8" ?>
<testsuite name="cxxtest"  tests="2" errors="0" failures="1" time="0" >
    <testcase classname="MyTestSuite" name="testAddition" line="7" />
    <testcase classname="MyTestSuite" name="testMultiplication" line="13">
        <failure file="MyTestSuite2.h" line="15" type="failedAssertEquals" >Error: Expected ↩
            (2 * 2 == 5), found (4 != 5)</failure>
    </testcase>
</testsuite>
```

This XML format is conforms to the XML standard used by other xUnit tools. Thus, this output can be used as input in other tools, like Jenkins, to generate test summaries.

4.2.5 XUnitPrinter

The `XUnitPrinter` test listener generates output using both the ErrorPrinter+ and `XmlPrinter` test listeners. This allows the user to interactively view a simple test summary, while simultaneously generating an XML summary of the test results. The `--xunit-printer` option specifies the use of `XUnitPrinter`:

```
cxxtestgen --xunit-printer -o runner.cpp MyTestSuite2.h
```

This test runner generates the following output:

```
Running 2 tests.
In MyTestSuite::testMultiplication:
MyTestSuite2.h:15: Error: Expected (2 * 2 == 5), found (4 != 5)
Failed 1 of 2 tests
Success rate: 50%
```

The default filename for the XML results is `TEST-cxxtest.xml`. The `--xunit-file` option can be used to specify an alternative filename. Additionally, the value of the `--world` option can be used to specify the filename `TEST-<world>.xml`.

4.3 Language Options

When `cxxtestgen` performs test discovery, it also performs checks to detect whether (1) the standard library is used and (2) exceptions are used. These checks configure CxxTest to *not* assume that these C++ language features are used when generating the test driver. Thus, CxxTest can naturally be used with compilers that do not support these features.

The `cxxtestgen` command includes several options that override these checks and define features of C++ that are used by the test runner. The `--have-std` option indicates that the test runner should use the standard library, and the `--no-std` option indicates that the test runner should not use the standard library. The --have-eh+ options indicates that the test runner should use exception handling, and the `--no-eh` indicates that the test runner should not not use exception handling.

The `--longlong` option specifies the type used for long long integers. The default is for *no* long long integer type to be specified, which is consistent with the current C++ standard.

CxxTest test runners depend quite heavily on static initialization of objects that are used to define and execute tests. The --no-static-init+ option can be used to avoid static initialization for compilers or linkers that have trouble compiling the default test runner.

4.4 Creating Test Runners from Parts

The default behavior of `cxxtestgen` is to generate a test runner that directly integrates classes that define the tests along with a `main()` function that executes all test suites. It is often useful to allow test suites to be processes separately and then linked together. The `--root` and `--part` options support this logic. For example, suppose that we wish to define a test runner for tests in the headers MyTestSuite1.h+ and `MyTestSuite2.h`. We execute `cxxtestgen` with the `--part` option to generate source files for each of the test suites:

```
cxxtestgen --part --error-printer -o MyTestSuite1.cpp MyTestSuite1.h
cxxtestgen --part --error-printer -o MyTestSuite2.cpp MyTestSuite2.h
```

Similarly, we execute `cxxtestgen` with the `--root` opiton to generate the `main()` routine:

```
cxxtestgen --root --error-printer -o runner.cpp
```

Finally, the test runner is built by compiling all of these source files together:

```
g++ -o runner -I$CXXTEST runner.cpp MyTestSuite1.cpp MyTestSuite2.cpp
```

4.5 Template Files

CxxTest supports the use of *template files* to provide a custom main()+ function. This may be useful when using a custom test listener, or when using an existing CxxTest test listener in a nonstandard manner. A template file is an ordinary source files with the embedded declaration `<CxxTest world>`, which tells cxxtestgen+ to insert the world definition at that point.

The `--template` option is used to specify the use of a template file:

```
cxxtestgen -o runner.cpp --template runner10.tpl MyTestSuite2.h
```

For example, consider the following template file:

```
#define CXXTEST_HAVE_EH
#define CXXTEST_ABORT_TEST_ON_FAIL
#include <cxxtest/ErrorPrinter.h>

int main()
{
    std::cout << "Starting test runner" << std::endl;
    int status = CxxTest::ErrorPrinter().run();
    std::cout << "Stopping test runner" << std::endl;
    return status;
}

// The CxxTest "world"
<CxxTest world>
```

This file specifies macros that customize the test runner, and output is generated before and after the tests are run.

Note that 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 behavior is not what you need, use the directive `<CxxTest preamble>` to specify where this preamble is inserted.

4.6 Test Discovery Options

The `cxxtestgen` command performs test discovery by searching C++ header files for CxxTest test classes. The default process for test discovery is a simple process that analyzes each line in a header file sequentially, looking for a sequence of lines that represent class definitions and test method definitions.

There are many limitations to this simple process for test discovery, and in CxxTest 4.0 a new test discovery mechanism was added based on the a parser for the Flexible Object Generator (FOG) language, which is a superset of C+. The grammar for the FOG language was adapted to parse C+ header files to identify class definitions and class inheritance relationships, class and namespace nesting of declarations, and class methods. This allows `cxxtestgen` to identify test classes that are defined with complex inheritance relationships.

The `--fog` option is used to specify the use of the FOG parser for test discovery. Although the FOG parser is more powerful, the simpler `cxxtestgen` test discover process is the default because the FOG parser is slower execute. Additionally, the FOG

parser requires the installation of `ply` and, for Python version 2.6, ordereddict+. If these packages are not available, then the `--fog` option is automatically disabled.

The following sections illustrate differences between these two test discovery mechanisms, along with general limitations of the test discovery process.

4.6.1 Unexpected Test Suite Format

The default test discovery mechanism does a very simple analysis of the input files, which can easily fail when test classes are not formated in a standard manner. For example, consider the following test suite:

```
// MyTestSuite4.h
#include <cxxtest/TestSuite.h>

class MyTestSuite4
    :
public CxxTest::TestSuite
{
public:
    void testAddition(void)
    {
        TS_ASSERT(1 + 1 > 1);
        TS_ASSERT_EQUALS(1 + 1, 2);
    }
};
```

This test suite is not recognized by the default test discovery mechanism, but the FOG parser correctly parsers this file and recognizes the test suite. A variety of similar discovery failures arise due to the simple process used by the test discovery mechanism.

4.6.2 Commenting Out Tests

Adding and disabling tests are two common steps in test development. The process of test discovery makes adding tests very easy. However, disabling tests is somewhat more complicated. Consider the following header file, which defines four tests (three of which are disabled):

```
// MyTestSuite3.h
#include <cxxtest/TestSuite.h>

class MyTestSuite3 : public CxxTest::TestSuite
{
public:
    void testAddition(void)
    {
        TS_ASSERT(1 + 1 > 1);
        TS_ASSERT_EQUALS(1 + 1, 2);
    }

//   void testMultiplication( void )
//   {
//      TS_ASSERT( 1 * 1 < 2 );
//      TS_ASSERT_EQUALS( 1 * 1, 2 );
//   }

/*
     void testSubtraction( void )
     {
        TS_ASSERT( 1 - 1 < 1 );
        TS_ASSERT_EQUALS( 1 - 1, 0 );
     }
```

```
*/

    void XtestDivision(void)
    {
        TS_ASSERT(1 / 1 < 2);
        TS_ASSERT_EQUALS(1 / 1, 1);
    }
};
```

The first is commented out with C++-style comments, the second test is commented out with C-style comments, and the third test is named in a manner that is not recognized through test discovery (i.e., it does not start with `test`).

The default test discovery mechanism only works with the first and third methods for disabling tests, but the FOG parser works with all three. The FOG parser performs a complex, multi-line parse of the source file, so it can identify multi-line C-style comments.

Note, however, that the use of C macros will not work:

```
// BadTestSuite1.h
#include <cxxtest/TestSuite.h>

class BadTestSuite1 : public CxxTest::TestSuite
{
public:
    void testAddition(void)
    {
        TS_ASSERT(1 + 1 > 1);
        TS_ASSERT_EQUALS(1 + 1, 2);
    }
#if 0
    void testSubtraction(void)
    {
        TS_ASSERT(1 - 1 < 1);
        TS_ASSERT_EQUALS(1 - 1, 0);
    }
#endif
};
```

The `cxxtestgen` discovery mechanisms do not perform a C preprocessing step, since that would generally require using externally defined preprocessing variable definitions. Additionally, preprocessor macros that act like functions will cause the FOG parser to fail unless they are followed by a semicolon.

5 Test Runner Syntax

The default behavior of the CxxTest test runner is to execute all tests in all of the test suites that are linked into the runner. However, CxxTest test runners process command line options that allow individual tests and test suites to be selected.

For example, consider a test runner defined as follows:

```
cxxtestgen -f --error-printer -o runner.cpp MyTestSuite1.h MyTestSuite2.h MyTestSuite4.h
```

The `--help` (`-h`) option can be used to print the command line options for a test runner. The command

```
./runner --help
```

generates the following output:

```
./runner <suitename>
./runner <suitename> <testname>
./runner -h
```

```
./runner --help
./runner --help-tests
./runner -v             Enable tracing output.
```

The `--help-tests` option is used to list all test suites that are defined in a test runner. The command

```
./runner --help-tests
```

generates the following output:

```
Suite/Test Names
--------------------------------------------------------------------------
MyTestSuite1 testAddition
MyTestSuite2 testAddition
MyTestSuite2 testMultiplication
MyTestSuite4 testAddition
```

The first column is the test suite name, and the second column is the test name.

All tests in a test suite can be executed by simply specifying the test suite name. For example

```
./runner MyTestSuite2
```

executes the tests in test suite `MyTestSuite2`:

```
Running 2 tests.
In MyTestSuite2::testMultiplication:
MyTestSuite2.h:16: Error: Expected (2 * 2 == 5), found (4 != 5)
Failed 1 of 2 tests
Success rate: 50%
```

Similarly, a single test can be executed by specifying the test suite followed by the test name. For example

```
./runner MyTestSuite2 testMultiplication
```

executes the `testMultiplication` test in test suite `MyTestSuite2`:

```
Running 1 test
In MyTestSuite2::testMultiplication:
MyTestSuite2.h:16: Error: Expected (2 * 2 == 5), found (4 != 5)
Failed 1 of 1 test
Success rate: 0%
```

The `-v` option enables the printing of trace information generated by the `TS_TRACE` function. For example, the `testMultiplicatio` test contains trace declarations before and after the multiplication test. Thus, the command

```
./runner -v MyTestSuite2 testMultiplication
```

generates this trace output before and after the test:

```
Running 1 test
In MyTestSuite2::testMultiplication:
MyTestSuite2.h:15: Trace: Starting multiplication test
MyTestSuite2.h:16: Error: Expected (2 * 2 == 5), found (4 != 5)
MyTestSuite2.h:17: Trace: Finishing multiplication test
Failed 1 of 1 test
Success rate: 0%
```

6 Advanced Testing Features

6.1 Preprocessor Macros

CxxTest recognizes a variety of preprocessor macros that can be used to modify the behavior of a test runner. Many of these mimic the options of the `cxxtestgen` command.

Preprocessor Macro	Description
`CXXTEST_HAVE_STD`	Use the standard library.
`CXXTEST_HAVE_EH`	Use exception handling.
`CXXTEST_ABORT_TEST_ON_FAIL`	Abort tests on failed asserts.
`CXXTEST_USER_VALUE_TRAITS`	Enable user-defined value traits. The default traits dump up to 8 bytes of the data as hex values.
`CXXTEST_OLD_TEMPLATE_SYNTAX`	Use old template syntax that is used by some compilers (e.g. Borland C++ 5).
`CXXTEST_OLD_STD`	Use old syntax for libraries where `std::` is not recognized.
`CXXTEST_MAX_DUMP_SIZE`	The value of this macro defines the maximum number of bytes to dump if `TS_ASSERT_SAME_DATA()` fails. The default is 0, which indicates no limit.
`CXXTEST_DEFAULT_ABORT`	The value of this macro is the default value of the dynamic *abort on fail* flag.
`CXXTEST_LONGLONG`	The value of this macro is used to define long long integers.

These preprocessor macros must be defined before the CxxTest header files are included in the test runner. For example, the following template file defines `CXXTEST_HAVE_EH` and `CXXTEST_ABORT_TEST_ON_FAIL` before other headers are included:

```
#define CXXTEST_HAVE_EH
#define CXXTEST_ABORT_TEST_ON_FAIL
#include <cxxtest/ErrorPrinter.h>

int main()
{
    std::cout << "Starting test runner" << std::endl;
    int status = CxxTest::ErrorPrinter().run();
    std::cout << "Stopping test runner" << std::endl;
    return status;
}

// The CxxTest "world"
<CxxTest world>
```

Several of these macros concern whether modern C++ conventions are supported by the compiler. If tests need to be ported to multiple compilers, then one important convention is whether the namespace `std::` is supported. For example, switching between `cout` and `std::cout` typically needs to be done throughout a code. CxxTest supports this with the `CXXTEST_STD()` macro. For example, `CXXTEST_STD(cout)` can be used within a test suite, and CxxTest handles the mapping of this to `cout` or `std::cout` depending on options provided to `cxxtestgen`.

6.2 Customizing Test Fixtures

6.2.1 Setup and Teardown

CxxTest test fixtures can be customized in several ways to manage the environment for test suites and individual tests. A common feature of test suites is that they share a common logic for setting up data used in the tests. Thus, there may be duplicate code

for creating objects, files, inputs, etc. Similarly, the tests may share common logic for cleaning up after the test is finished (e.g. deleting temporary objects).

You can put this shared code in a common place by overriding the virtual functions `TestSuite::setUp()` and `TestSuite::tear`
The `setUp()` function is called before each test, and `tearDown()` is called after each test.

For example, the following test suite employs `setUp()` and `tearDown()` methods to allocate and deallocate memory for a string buffer:

```
// MyTestSuite5.h
#include <cxxtest/TestSuite.h>
#include <string.h>

class MyTestSuite5 : public CxxTest::TestSuite
{
    char *_buffer;

public:

    void setUp()
    {
        _buffer = new char[1024];
    }

    void tearDown()
    {
        delete [] _buffer;
    }

    void test_strcpy()
    {
        strcpy(_buffer, "Hello, world!");
        TS_ASSERT_EQUALS(_buffer[0], 'H');
        TS_ASSERT_EQUALS(_buffer[1], 'e');
    }

    void test_memcpy()
    {
        memcpy(_buffer, "Hello, world!", sizeof(char));
        TS_ASSERT_EQUALS(_buffer[0], 'H');
        TS_ASSERT_EQUALS(_buffer[1], 'e');
    }
};
```

6.2.2 Dynamically Created Test Suites

CxxTest test fixtures can also be customized during the construction and deconstruction of test suites. By default, CxxTest test suites are instantiated statically in the test runner. However, dynamically created test suites can be used to perform suite-level setup and teardown operations, verify the environment needed to execute a test suite, and construct test suites that require a nontrivial constructor.

CxxTest instantiates a test suite dynamically if the `createSuite()` or `destroySuite()` methods are defined. For example, the following test suite checks to see if it is being compiled with Microsoft Visual Studio. If not, the `createSuite()` returns a null pointer, indicating that the test suite was not created.

```
// MyTestSuite6.h
#include <cxxtest/TestSuite.h>

class MyTestSuite6 : public CxxTest::TestSuite
{
public:
```

```
    static MyTestSuite6* createSuite()
    {
    #ifdef _MSC_VER
    return new MyTestSuite6();
    #else
    return 0;
    #endif
    }

    static void destroySuite( MyTestSuite6* suite )
    { delete suite; }

    void test_nothing()
    {
        TS_FAIL( "Nothing to test" );
    }
};
```

6.2.3 Global and World Fixtures

CxxTest supports two related mechanisms for performing *global* setup and teardown operations. *Global fixtures* are classes that inherit from `CxxTest::GlobalFixture`, and they define `setUp` and `tearDown` methods. The `setUp` method for all global fixtures is called before each test is executed, and the `tearDown` method for all global fixtures is called after each test is completed. Thus, this mechanism provides a convenient way of defining setup and teardown operations that apply to all test suites.

For example, consider the following test suite:

```
// MyTestSuite8.h
#include <cxxtest/TestSuite.h>
#include <cxxtest/GlobalFixture.h>

//
// Fixture1 counts its setUp()s and tearDown()s
//
class Fixture1 : public CxxTest::GlobalFixture
{
public:
    unsigned setUpCount;
    unsigned tearDownCount;

    Fixture1() { setUpCount = tearDownCount = 0; }

    bool setUp() { ++ setUpCount; return true; }
    bool tearDown() { ++ tearDownCount; return true; }

    bool setUpWorld() { printf( "Starting a test suite\n" ); return true;}
    bool tearDownWorld() { printf( "Finishing a test suite\n" ); return true;}
};
static Fixture1 fixture1;


//
// Fixture2 counts its setUp()s and tearDown()s and makes sure
// its setUp() is called after Fixture1 and its tearDown() before.
//
class Fixture2 : public Fixture1
{
public:
    bool setUp()
```

```
    {
        TS_ASSERT_EQUALS(setUpCount, fixture1.setUpCount - 1);
        TS_ASSERT_EQUALS(tearDownCount, fixture1.tearDownCount);
        return Fixture1::setUp();
    }

    bool tearDown()
    {
        TS_ASSERT_EQUALS(setUpCount, fixture1.setUpCount);
        TS_ASSERT_EQUALS(tearDownCount, fixture1.tearDownCount);
        return Fixture1::tearDown();
    }
};
static Fixture2 fixture2;


//
// Verify the counts for the global fixtures
//
class MyTestSuite8 : public CxxTest::TestSuite
{
public:
    void testCountsFirstTime()
    {
        TS_ASSERT_EQUALS(fixture1.setUpCount,    1);
        TS_ASSERT_EQUALS(fixture1.tearDownCount, 0);
        TS_ASSERT_EQUALS(fixture2.setUpCount,    1);
        TS_ASSERT_EQUALS(fixture2.tearDownCount, 0);
    }

    void testCountsSecondTime()
    {
        TS_ASSERT_EQUALS(fixture1.setUpCount,    2);
        TS_ASSERT_EQUALS(fixture1.tearDownCount, 1);
        TS_ASSERT_EQUALS(fixture2.setUpCount,    2);
        TS_ASSERT_EQUALS(fixture2.tearDownCount, 1);
    }
};
```

This test suite defines a runner that generates the following output:

```
Running 2 testsStarting a test suite
Starting a test suite
..Finishing a test suite
Finishing a test suite
OK!
```

Note that the global fixtures are instantiated with static global values. This ensures that these fixtures are created before the runner is initialized. Also, note that the `setUp` methods are called in the same sequence that the global fixtures are instantiated, and the `tearDown` methods are called in the reverse sequence. Finally, note that the `setUp` and `tearDown` methods in global fixtures return a boolean value, which indicates success or failure of that operation.

This example also illustrates the use of *world fixtures*, which perform setup and teardown operations that are executed once each when beginning and finishing tests in each test suite. World fixtures are defined with the `setUpWorld` and `tearDownWorld` methods in a global fixture.

6.2.4 Runtime Test Customization

CxxTest defines several functions that can be called in a test suite to modify the default behavior of CxxTest.

Test Suite Method	Description
`setAbortTestOnFail(bool)`	This function specifies whether tests abort after a failure. The default value of the flag is `false`. This function only has an effect if exception handling is enabled.
`setMaxDumpSize(unsigned)`	This function sets the maximum number of bytes that are dumped when `TS_ASSERT_SAME_DATA()` fails. The default is 0, which indicates no limit.

Note that the the configuration parameters are reset to their default values after each test is executed (more precisely, after `tearDown()` is called). Consequently, calling these functions in the `setUp()` function has the effect of setting that value for the entire test suite.

7 Value Traits

CxxTest's test assertions like TS_ASSERT_EQUALS work for built-in types, but they will not likely work for user-defined data types. This is because CxxTest needs a way to compare objects and to convert them to strings when printing test failure summaries. Thus, user-defined data types need to have the `operator=` method defined to ensure that test assertions can be applied.

For example, the following code

```
// MyTestSuite7.h
#include <cxxtest/TestSuite.h>
#include <iostream>

class MyTestSuite7 : public CxxTest::TestSuite
{
public:

   struct Data
   {
      char data[3];
      bool operator==(Data o) {
          return (memcmp(this, &o, sizeof(o)) == 0);
      }
   };

   struct Data2
   {
      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 );

      Data2 z, w;
      memset( z.data, 0x12, sizeof(x.data) );
      memset( w.data, 0xF6, sizeof(y.data) );
      TS_ASSERT_SAME_DATA( &z, &w, sizeof(z) )
   }
};
```

defines a test runner that generates the following output

```
Running 1 test
In TestMyData::testCompareData:
MyTestSuite7.h:27: Error: Expected (x == y), found ({ 12 12 12  } != { F6 F6 F6  })
MyTestSuite7.h:32: Error: Expected sizeof(z) (3) bytes to be equal at (&z) and (&w), found:
   { 12 12 12 }
     differs from
   { F6 F6 F6 }
Failed 1 of 1 test
Success rate: 0%
```

The `operator=` method is required to apply TS_ASSERT_EQUALS to `Data` objects. However, the TS_ASSERT_SAME_DATA assertion can be applied to `Data2` objects that do not have `operator=` defined.

Since CxxTest does not rely on any external library, 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:

```
int i = 10;
CxxTest::ValueTraits<int> converter(i);
const char* string = converter.asString();
```

The CxxTest header file `cxxtest/ValueTraits.h` defines value traits for standard types like `int`, `char`, `double`, etc. The default `ValueTraits` class for unknown types dumps up to 8 bytes of the value in hex format.

If the macro `CXXTEST_USER_VALUE_TRAITS` is defined, then CxxTest will omit the default definitions for `ValueTraits`. This allows a user to define their own trait specifications to customize the display of trait information.

7.1 Enumeration Traits

CxxTest provides a simple way to define value traits for enumeration types. The `CXXTEST_ENUM_TRAITS` macro is used to define value traits for all members of an enumeration set.

For example, the following code

```
// MyTestSuite9.h
#include <cxxtest/TestSuite.h>

enum Answer {
    Yes,
    No,
    Maybe,
    DontKnow,
    DontCare
};

// Declare value traits for the Answer enumeration
CXXTEST_ENUM_TRAITS( Answer,
                     CXXTEST_ENUM_MEMBER( Yes )
                     CXXTEST_ENUM_MEMBER( No )
                     CXXTEST_ENUM_MEMBER( Maybe )
                     CXXTEST_ENUM_MEMBER( DontKnow )
                     CXXTEST_ENUM_MEMBER( DontCare ) );

// Test the trait values
class EnumTraits : public CxxTest::TestSuite
{
public:
    void test_Enum_traits()
    {
        TS_FAIL( Yes );
        TS_FAIL( No );
        TS_FAIL( Maybe );
```

```
        TS_FAIL( DontKnow );
        TS_FAIL( DontCare );
        TS_FAIL( (Answer)1000 );
    }
};
```

defines a test runner that generates the following output

```
Running 1 test
In EnumTraits::test_Enum_traits:
MyTestSuite9.h:26: Error: Test failed: Yes
MyTestSuite9.h:27: Error: Test failed: No
MyTestSuite9.h:28: Error: Test failed: Maybe
MyTestSuite9.h:29: Error: Test failed: DontKnow
MyTestSuite9.h:30: Error: Test failed: DontCare
MyTestSuite9.h:31: Error: Test failed: (Answer)1000
Failed 1 of 1 test
Success rate: 0%
```

The enumeration value traits print strings that represent the elements of the enumeration, except where a numeric value is provided.

Note that the `CXXTEST_ENUM_TRAITS` macros has two arguments; the list of `CXXTEST_ENUM_MEMBER` macros is not separated by commas!

7.2 Defining New Value Traits

Defining value traits for a new class is done by providing a class specialization of `ValueTraits` that converts an object of the new class to a string. For example, consider the definition of the `MyClass` class:

```
// MyClass.h

class MyClass
{
public:

   int value;

   MyClass(int value_) : value(value_) {}

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

   // This is required 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
// This declaration is only activated when building a CxxTest test suite
#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
```

This class includes definitions of `operator==` and `operator<` that support comparisons with TS_ASSERT_EQUALS and TS_ASSERT_LESS_THAN. Additionally, this header contains a specialization of `ValueTraits` (in the `CxxTest` namespace) that generates a string description of a `MyClass` instance.

The following test suite illustrates how these definitions can be used to define a test runner:

```
// MyTestSuite10.h
#include <cxxtest/TestSuite.h>
#include <MyClass.h>

class MyTestSuite10 : public CxxTest::TestSuite
{
public:
   void test_le()
   {
      MyClass x(1), y(2);
      TS_ASSERT_LESS_THAN( x, y );
   }

   void test_eq()
   {
      MyClass x(1), y(2);
      TS_ASSERT_EQUALS( x, y );
   }
};
```

This runner for this test suite generates the following output:

```
Running 2 tests.
In MyTestSuite10::test_eq:
MyTestSuite10.h:17: Error: Expected (x == y), found (MyClass( 1 ) != MyClass( 2 ))
Failed 1 of 2 tests
Success rate: 50%
```

The test failure print logic uses the specialization of `ValueTraits` to create the string description of `MyClass` that appears in the output.

7.3 Defining Value Traits for Template Classes

A simple modification to the above example illustrates how a trait can be defined for a template class:

```
// MyTestSuite11.h
#include <cxxtest/TestSuite.h>
#include <TMyClass.h>

class MyTestSuite11 : public CxxTest::TestSuite
{
public:
   void test_le()
   {
      TMyClass<int> x(1), y(2);
      TS_ASSERT_LESS_THAN( x, y );
   }
```

```
   void test_eq()
   {
      TMyClass<int> x(1), y(2);
      TS_ASSERT_EQUALS( x, y );
   }
};
```

Unfortunately, this example employs partial template specialization, which is not supported by all C++ compilers.

8 Testing with Mock Objects

Mock Objects are a very useful concept for testing complex software. The key idea is to pass special objects to tested code that facilitates the testing process. For instance, a class that implements a protocol over TCP might rely on an abstract `ISocket` interface. Then a mock testing strategy could pass a `MockSocket` object that does anything that is useful for testing (e.g., keep a log of all data "sent" to verify later).

However, when a challenge for C/C++ developers is that you may need to call *global* functions which you cannot override. Consider any code that uses `fopen()`, `fwrite()` and `fclose()`. It is not very elegant to have this code actually create files while being tested. Even more importantly, you need to test how the code behaves when "bad" things happen (e.g., when `fopen()` fails). Handling these types of exceptional conditions is often a very challenging issue for software testing.

CxxTest addresses this challenge by providing a generic mechanism for defining mock global functions. The next section illustrates this mechanism for a single global function. The following section provides more detail about specific features of CxxTest's support for mock testing.

8.1 Example: A Mock `time()` Function

Suppose that we want to perform mock testing using the well known standard library function `time()`. Setting up a test suite with a mock global function for `time()` can be broken down into the following steps.

8.1.1 Declare Mock Functions

The `CXXTEST_MOCK_GLOBAL` macro is used to declare mock global functions. It is often convenient to include these declarations in a header file, which is used in both the test suite as well as the code that is being tested:

```
// time_mock.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        */ );
```

8.1.2 Mock Functions in Tested Code

The tested code uses mock global functions, rather than using the global functions directly. You access mock functions in the `T` (for *Test*) namespace, so the tested code calls `T::time()` instead of `time()`. This is the equivalent of using abstract interfaces instead of concrete classes.

```
// rand_example.cpp
#include <time_mock.h>

int generateRandomNumber()
{
```

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

8.1.3 Mock Source Files

A source file needs to be defined that implements `T::time()` by calling the real global function. This definition is performed automatically by defining `CXXTEST_MOCK_REAL_SOURCE_FILE` before the header file is defined:

```
// time_real.cpp
#define CXXTEST_MOCK_REAL_SOURCE_FILE
#include <time_mock.h>
```

This source file is not used for testing, but instead it supports normal use of the tested code.

Similarly, a source file needs to be defined that implements `T::time()` by calling the mock global function. This definition is performed automatically by defining `CXXTEST_MOCK_TEST_SOURCE_FILE` before the header file is defined:

```
// time_mock.cpp
#define CXXTEST_MOCK_TEST_SOURCE_FILE
#include <time_mock.h>
```

8.1.4 Test Suites using Mock Functions

A mock object for the `time()` function is created using the `T::Base_time` class, which is automatically created by CxxTest. This class includes a `time()` method whose API is the same as the global `time()` function. Thus, this method can be defined to have whatever behavior is desired during testing. For example, the following example defines a mock object that increments a counter to define an incremental value for `time()`.

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

int generateRandomNumber();


class MockObject : public T::Base_time
{
public:
    MockObject(int initial) : counter(initial) {}
    int counter;
    time_t time( time_t * ) { return counter++; }
};

class TestRandom : public CxxTest::TestSuite
{
public:
    void test_generateRandomNumber()
    {
        MockObject t(1);
        TS_ASSERT_EQUALS( generateRandomNumber(), 3 );
        TS_ASSERT_EQUALS( generateRandomNumber(), 6 );
        TS_ASSERT_EQUALS( generateRandomNumber(), 9 );
    }
};
```

Note that CxxTest uses global data to associate calls made with `T::time()` to calls to `MockObject::time()`. The `MockObject` class simply needs to be instantiated prior to the call to `T::time()`.

8.1.5 Building the Test Runner

The `cxxtestgen` command is used to create a test runner with mock functions in a normal manner:

```
cxxtestgen --error-printer -o runner.cpp MockTestSuite.h
```

The test runner source file, `runner.cpp`, needs to be compiled an linked to the mock function definition, `time_mock.cpp`, as well as the code being tested, `rand_example.cpp`:

```
g++ -o runner -I. -I$CXXTEST runner.cpp time_mock.cpp rand_example.cpp
```

This generates a test runner that generates the following output:

```
Running 1 test.OK!
```

8.2 Advanced Topics

8.2.1 Void Functions

The `CXXTEST_MOCK_VOID_GLOBAL` is used to define mock global functions that return `void`. This is identical to `CXXTEST_MOCK_` except that it does not specify the return type. Take a look in `sample/mock/T/stdlib.h` for a demonstation.

8.2.2 Calling the Real Functions While Testing

During testing it is sometimes necessary to call the real global function instead of the mock global function. CxxTest allows a user to do this by creating a special mock object. For a global mock function of `time()`, the object `T::Real_time` represents the real function. If this class is created, then `T::time()` will be redirected to the real function.

8.2.3 Mocking Nonexistent Functions

Sometimes the tested code calls functions that are not available when testing. For example, this can happen when testing driver code that calls kernel functions that are not available to a user-mode test runner. CxxTest can provide mock global function definitions for the test code while using the original functions in the tested code.

The `CXXTEST_SUPPLY_GLOBAL` and `CXXTEST_SUPPLY_VOID_GLOBAL` macros are used to provide mock global function definitions. For example, the following declaration creates a mock global function for 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 driver code calls `IoCallDriver()` normally; there is no need for the `T::` syntax. The test suite is defined using the `T::Base_IoCallDriver` as with normal mock objects.

CxxTest also provides the macros `CXXTEST_SUPPLY_GLOBAL_C` and `CXXTEST_SUPPLY_GLOBAL_VOID_C` that declare the functions with `C` linkage (i.e., using `extern "C"`). These macros are used to declare function prototypes, since you may not be able to include the header files in the test suite that are associated with the mock global function.

8.2.4 Functions in Namespaces

The `CXXTEST_MOCK` macro is used to declare a mock global function that is associated with a function in a namespace, including static class member functions. For example, consider the function `bool Files::FileExists( const String &name );` the namespace `Files` contains the function `FileExists`. The mock class will be called `T::Base_Files_FileExist` and the function to implemented would be `fileExists`. The `CXXTEST_MOCK` macro declares this mock global function as follows:

```
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        */ );
```

Similarly, the `CXXTEST_MOCK_VOID` macro is used to declare a mock global function that returns `void`.

The `CXXTEST_SUPPLY` and `CXXTEST_SUPPLY_VOID` macros are used to provide mock global function definitions for nonexistent functions. For example:

```
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         */ );
```

Similarly, the `CXXTEST_SUPPLY_C` and `CXXTEST_SUPPLY_VOID_C` macros declare the functions with `C` linkage.

8.2.5 Overloaded Functions

The `CXXTEST_MOCK` and `CXXTEST_MOCK_VOID` macros have a flexible interface that can provide mock global function definitions for overloaded functions. The arguments simply need to specify different mock class names, mock member names and different prototype definitions. These different mock declarations will generate different mock objects that can be explicitly referenced in a test suite.

8.2.6 The Mock Namespace

The default namespace for mock functions is `T::`. This namespace can be changed by defining the `CXXTEST_MOCK_NAMESPACE` macro.

9 Installation

A key feature of CxxTest is that it does has virtually no installation process. The `cxxtestgen` script can be added be directly executed from the `cxxtest/bin` directory, so it may be convenient to add this directory to a `PATH` environment. Beyond that, the build process for CxxTest simply needs to reference the `cxxtest` root directory to enable proper includes while compiling test runners.

The `cxxtestgen` script script should work with versions Python 2.6 or newer. The FOG parser requires two Python packages:

- `ply`
- `ordereddict` (This is only needed when running Python 2.6)

If these packages are not available, then `cxxtestgen` will generate an error when the FOG parser option is selected. If you have setuptools or distribute installed, then you can install these packages from PyPI by executing

```
easy_install ply
easy_install ordereddict
```

10 Status and Future Plans

The CxxTest 4.0 release reflects major changes in the management and focus of CxxTest. The 4.0 release is the first release of CxxTest in over seven years, and virtually all of the initial developers have moved on to other projects. CxxTest is heavily used at Sandia National Laboratories, and Sandia's ongoing use of CxxTest is a major driver for the 4.0 release.

Note
TODO: Say more here about webresources for hosting, mailing lists, testinging and project management

Similarly, major changes in CxxTest reflect the focus of the developer team:

- Perl is no longer used to support CxxTest scripts. Python is now the only scripting language used by CxxTest.
- The testing scripts have been rewritten using the PyUnit framework.
- The installation process for CxxTest now leverages and integrates with the system Python installation.
- A more comprehensive C++ parser is now available, which supports testing of templates.
- The CxxTest GUI is no longer supported, and the TS_WARN is deprecated.
- CxxTest runners now have a command-line interface that facilitates interative use of the test runner.
- A new user guide is now available in PDF, HTML and Ebook formats.
- Updated the `cxxtestgen` script to work with Python 2.6 through 3.2

Additionally, CxxTest is now validated with continuous integration tests. Yes, the CxxTest developers eat their own dog food!

Although the GUI option for `cxxtestgen` appears to work fine, this GUI is rather primitive. It simply provides a visual summary of the test results, and not the interactive test execution that a user would expect. This capability is deprecated since none of the current developers use this feature. CxxTest users should consider using CxxTest with Jenkins. The `XUnitPrinter` test listener generates XML files that can be easily integrated by Jenkins, which creates a visual summary of test results with links to drill-down into test outputs.

This documentation has highlighted the commonly used test listeners. There are a variety of other test listeners provided by CxxTest that support advanced Cxxtest applications. For example, the `YesNoRunner` is perhaps the simplest test listener; it simply returns the number of test failures. The `StdioFilePrinter` is used by `StdioPrinter`, but it does not assume that `stdio` is the default output stream. This test listener can be used in contexts where a custom output stream must be specified.

11 Acknowledgements

CxxTest was originally developed by Erez Volk. The following developers actively contributed to the CxxTest 4.0 release:

- Gašper Ažman
- Kevin Fitch
- William Hart
- John Siirola

A major advancement in CxxTest's capability is the new test discovery mechanism that is based on a parser of the Flexible Object Language (FOG). FOG generalizes the C++ syntax, which enables CxxTest to extract high-level class structure for test discovery. FOG was developed by Edward Willink:

- Edward D. Willink. *Meta-Compilation for C++*, PhD Thesis, Computer Science Research Group, University of Surrey, January 2000.

The FOG parser in CxxTest critically relies on the excellent LALR parser provided by Dave Beazley's `ply` Python package. The scalable performance of `ply` is critical for CxxTest.

The development of CxxTest has been partially supported by Sandia National Laboratories. Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

A Test Assertion Examples

TS_ASSERT
This is the most basic test assertion, which simply verifies that the `expr` argument is true:

```
void test_assert(void)
{
    TS_ASSERT(1 + 1 > 1);
}
```

TS_ASSERT_DELTA
This test assertion verifies two floating point values are within a specified absolute difference:

```
void test_assert_delta(void)
{
    TS_ASSERT_DELTA(sqrt(4.0), 2.0, 1e-7);
}
```

TS_ASSERT_DIFFERS
This test assertion verifies that the two arguments are not equal:

```
void test_assert_differs(void)
{
    TS_ASSERT_DIFFERS(1, 2);
}
```

TS_ASSERT_EQUALS
This test assertion verifies that the two arguments are equal:

```
void test_assert_equals(void)
{
    TS_ASSERT_EQUALS(21 % 5, 1);
}
```

Note that this test is performed using the C++ == operator, whose behavior may be redefined for the two argument types.

TS_ASSERT_LESS_THAN
This test assertion verifies that the first argument is strictly less than the second argument:

```
void test_assert_less_than(void)
{
    TS_ASSERT_LESS_THAN(0, 1);
}
```

TS_ASSERT_LESS_THAN_EQUALS
This test assertion verifies that the first argument is less than or equal to the second argument:

```
void test_assert_less_than_equals(void)
{
    TS_ASSERT_LESS_THAN_EQUALS(0, 0);
}
```

TS_ASSERT_PREDICATE

This test assertion takes as an argument the name of a class, similar to a STL `unary_function`, and evaluates the `operator()` method:

```
class IsOdd
{
public:
    bool operator()(int x) const { return x % 2 == 1; }
};

void test_assert_predicate(void)
{
    TS_ASSERT_PREDICATE(IsOdd, 29);
}
```

This test assertion can be seen as a generalization of TS_ASSERT, but it allows the tester to see the failed value.

TS_ASSERT_RELATION

It takes as an argument the name of a class, similar to a STL `binary_function`, and evaluates the `operator()` method:

```
void test_assert_relation(void)
{
    TS_ASSERT_RELATION(std::greater<double>, 1e6, 1000.0);
}
```

This test assertion can be seen as a generalization of TS_ASSERT_EQUALS, TS_ASSERT_DIFFERS, TS_ASSERT_LESS_THAN and TS_ASSERT_LESS_THAN_EQUALS. This can be used to assert comparisons which are not covered by the builtin test assertions.

TS_ASSERT_SAME_DATA

This test assertion is similar to TS_ASSERT_EQUALS, except that it compares the contents of two buffers in memory:

```
void test_assert_same_data(void)
{
    char input = "The quick brown fox ran over the lazy dog";
    char output[26];
    memcopy(output, input, 26);
    TS_ASSERT_SAME_DATA(input, output, 26);
}
```

The standard runner dumps the contents of both buffers as hex values when this test fails.

TS_ASSERT_THROWS

This test assertion verifies that the specified exception is thrown when the first argument is executed:

```
void throws_runtime_error(void)
{
    raise std::runtime_error, "This method simply generates an exception";
}
```

```
void test_assert_throws(void)
{
    TS_ASSERT_THROWS(self.throws_runtime_error(), std::runtime_error);
}
```

TS_ASSERT_THROWS_ANYTHING
This test assertion verifies that *some* exception is thrown when the first argument is executed:

```
void test_assert_throws_anything(void)
{
    TS_ASSERT_THROWS_ANYTHING(self.throws_runtime_error());
}
```

TS_ASSERT_THROWS_ASSERT
This test assertion verifies that an exception is thrown when executing the first argument. The second argument specifies a variable declaration for the exception, and the third argument is executed to test that exception value:

```
void throws_value(void)
{
    raise 1;
}

void test_assert_throws_assert(void)
{
    TS_ASSERT_THROWS_ASSERT(self.throws_value(), const Error & e, TS_ASSERT_EQUALS(e,  ↩
        1));
}
```

Note that this can be viewed as a generalization of TS_ASSERT_THROWS_EQUALS.

TS_ASSERT_THROWS_EQUALS
This test assertion verifies that an exception is thrown when executing the first argument. The second argument specifies a variable declaration for the exception, and the third and fourth arguments are values that are asserted equal after the exception is thrown:

```
void test_assert_throws_equals(void)
{
    TS_ASSERT_THROWS_EQUALS(self.throws_value(), const Error & e, e.what(), 1);
}
```

TS_ASSERT_THROWS_NOTHING
This test assertion verifies that an exception is *not* thrown when executing the first argument:

```
void throws_nothing(void)
{ }

void test_assert_throws_nothing(void)
{
    TS_ASSERT_THROWS_ASSERT(self.throws_nothing());
}
```

TS_FAIL
This function triggers a test failure with an associated message:

```
    void test_fail(void)
    {
        TS_FAIL("This test has failed.");
    }
```

TS_TRACE
This function prints an informational message:

```
    void test_trace(void)
    {
        TS_TRACE("This is a test tracing message.");
    }
```

TS_WARN
This function prints a message as a warning:

```
    void test_warn(void)
    {
        TS_WARN("This is a warning message.");
    }
```

B Integrating with Your Build Environment

CxxTest can be integrated into a variety of build environments to automate the generation, compilation and execution of test runners. Here is a rough breakdown of this process:

- 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.
- Configure the build tool to run the tests automatically.

Unfortunately, different build tools and IDEs need to setup this process in different ways. The following sections provide rough guidance for doing this for some come use cases.

Note
These examples are not actively maintained and tested. Please send suggestions to the CxxTest developers for updating this documentation.

B.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 $^
```

B.2 Using Cons

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

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

B.3 Using Microsoft Visual Studio

See `cxxtest/sample/msvc` in the distribution to see a reasonable integration of CxxTest with Microsoft Visual Studio's IDE. 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. The script `sample/msvc/FixFiles.bat` can automate some of this process.

B.4 Using Microsoft Windows DDK

To use CxxTest with the `build` utility for device drivers, you add the generated tests file as an extra dependency using the `NTBUILDTARGET0` macro and the `Makefile.inc` file. An example of how to do this is in the CxxTest distribution under `sample/winddk`.

C Testing CxxTest

In the `cxxtest/test` directory, you can execute

```
python test_cxxtest.py
```

to launch all tests. By default, this script executes test suites for a variety of compilers if they are found on the user's path: `g++`, `clang++`, `cl` (the Microsoft Visual Studio compiler). Additionally, this test script includes separate test suites for the default test discovery mechanism as well as test discovery using the new FOG parser.

You can execute a specific test suite by giving its name as an argument to this test script. For example, the command

```
python test_cxxtest.py TestGpp
```

executes the `TestGpp` test suite, which tests CxxTest with the `g++` compiler. Similarly, the command

```
python test_cxxtest.py TestGppFOG
```

executes the test suite that tests CxxTest using the `g++` compiler and the FOG parser.

The `test_cxxtest.py` script should work with versions Python 2.7 or newer. If you are running Python 2.6, you will need to install the `unittest2` package. If you have setuptools or distribute installed, then you can install this package from PyPI by executing

```
easy_install unittest2
```

The FOG parser requires two Python packages:

- `ply`
- `ordereddict` (This is only needed when running Python 2.6)

If these packages are not available, then `test_cxxtest.py` will skip the FOG tests.

A CxxTest Releases

A.1 Version 4.0 (TODO)

- Perl is no longer used to support CxxTest scripts. Python is now the only scripting language used by CxxTest.
- The testing scripts have been rewritten using the PyUnit framework.
- The installation process for CxxTest now leverages and integrates with the system Python installation.
- A more comprehensive C++ parser is now available, which supports testing of templates.
- The CxxTest GUI is no longer supported, and the TS_WARN is deprecated.
- CxxTest runners now have a command-line interface that facilitates interative use of the test runner.
- A new user guide is now available in PDF, HTML and Ebook formats.

A.2 Version 3.10.1 (2004-12-01)

- Improved support for VC7
- Fixed clash with some versions of STL

A.3 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

A.4 Version 3.9.1 (2004-01-19)

- Fixed small bug with runner exit code
- Embedded test suites are now deprecated

A.5 Version 3.9.0 (2004-01-17)

- Added TS_TRACE
- Added --no-static-init
- CxxTest::setAbortTestOnFail() works even without --abort-on-fail

A.6 Version 3.8.5 (2004-01-08)

- Added --no-eh
- Added CxxTest::setAbortTestOnFail() and CXXTEST_DEFAULT_ABORT
- Added CxxTest::setMaxDumpSize()
- Added StdioFilePrinter

A.7 Version 3.8.4 (2003-12-31)

- Split distribution into cxxtest and cxxtest-selftest
- Added ‘sample/msvc/FixFiles.bat’

A.8 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

A.9 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.

A.10 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

A.11 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)

A.12 Version 3.7.0 (2003-09-20)

- Added TS_ASSERT_LESS_THAN_EQUALS
- Minor cleanups

A.13 Version 3.6.1 (2003-09-15)

- Improved QT GUI
- Improved portability some more

A.14 Version 3.6.0 (2003-09-04)

- Added --longlong
- Some portability improvements

A.15 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

A.16 Version 3.1.1 (2003-08-27)

- Fixed small bug in TS_ASSERT_THROWS_*()

A.17 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

A.18 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

A.19 Version 2.8.4 (2003-07-21)

- Now supports g++-3.3
- Added --have-eh
- Fixed bug in numberToString()

A.20 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!

A.21 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

A.22 Version 2.8.1 (2003-01-16)

- Fixed charToString() for negative chars

A.23 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

A.24 Version 2.7.0 (2002-09-29)

- Added embedded test suites
- Major internal improvements