Cpputest

CppUTest unit testing and mocking framework for C/C++

CppUTest coverage 100%

- Core Manual
- CppUMock Manual
- Plugin Manual
- Platforms stories
- View on GitHub

Download Release 3.8 as .zip Download Release 3.8 as .tar.gz

CppUTest is a C /C++ based unit xUnit test framework for unit testing and for test-driving your code. It is written in C++ but is used in C and C++ projects and frequently used in embedded systems.

CppUTest's core design principles

- Simple to use and small
- Portable to old and new platforms
- Build with Test-driven Development in mind

Table of Content

Getting Started

Your first test

To write your first test, all you need is a new cpp file with a TEST_GROUP and a TEST, like:

```
#include "CppUTest/TestHarness.h"

TEST_GROUP(FirstTestGroup)
{
};

TEST(FirstTestGroup, FirstTest)
{
   FAIL("Fail me!");
}
```

This test will fail. For adding new test_groups, this will be all you need to do (and make sure its compiled). If you want to add another test, all you need to do it:

```
TEST(FirstTestGroup, SecondTest)
{
    STRCMP_EQUAL("hello", "world");
}
```

One of the key design goals in CppUTest is to make it *very easy* to add and remove tests as this is something you'll be doing a lot when test-driving your code.

Writing your main

Of course, in order to get it to run, you'll need to create a main. Most of the mains in CppUTest are very similar. They typically are in an AllTests.cpp file and look like this:

```
#include "CppUTest/CommandLineTestRunner.h"

int main(int ac, char** av)
{
    return CommandLineTestRunner::RunAllTests(ac, av);
}
```

CppUTest will automatically find your tests (as long as you don't like them in a library).

Makefile changes

To get the above to work, you'll need a Makefile or change your existing one. The needed changed are:

CppUTest path

If you have a system installed version (e.g. via apt-get) then you probably don't need to change the path. Otherwise you'll need to add CppUTest include directories to your Makefile. Usually this is done by defininig a CppUTest path either as system variable or in the Makefile, such as:

```
CPPUTEST_HOME = /Users/vodde/workspace/cpputest
```

Compiler options

For the compiler you have to add the include path and optional (but recommended) the CppUTest pre-include header which enables debug information for the memory leak detector *and* offers memory leak detection in C. Lets start with the include path, you'll need to add:

```
CPPFLAGS += -I$(CPPUTEST_HOME)/include
```

(CPPFLAGS works for both .c and .cpp files!)

Then for the memory leak detection, you'll need to add:

```
CXXFLAGS += -include $(CPPUTEST_HOME)/include/CppUTest/MemoryLeakDetectorNewMacros.h
CFLAGS += -include $(CPPUTEST_HOME)/include/CppUTest/MemoryLeakDetectorMallocMacros.h
```

These flags need to be added to *both* test code *and* production code. They will replace the malloc and new with a debug variant.

Linker options

You need to add CppUTest library to the linker flags, for example, like:

```
LD_LIBRARIES = -L$(CPPUTEST_HOME)/lib -lCppUTest -lCppUTestExt
```

(The last flag is only needed when you want to use extensions such as mocking)

Most commonly used Test Macros

- TEST(group, name) - define a test
- IGNORE_TEST(group, name) - turn off the execution of a test
- TEST_GROUP(group) - Declare a test group to which certain tests belong. This will also create the link needed from another library.
- TEST_GROUP_BASE(group, base) - Same as TEST_GROUP, just use a different base class than Utest
- IMPORT_TEST_GROUP(group) - Export the name of a test group so it can be linked in from a library (also see Advanced Stuff)

Set up and tear down support

- Each TEST_GROUP may contain setup or teardown methods
- Setup is called prior to each TEST body and Teardown is called after the test body

Assertions

The failure of one of these macros causes the current test to immediately exit:

- CHECK(boolean condition) - checks any boolean result.
- CHECK_TEXT(boolean condition, text) - checks any boolean result and prints text on failure.
- CHECK_FALSE(condition) - checks any boolean result
- CHECK_EQUAL(expected, actual) - checks for equality between entities using ==. So if you have a class that supports operator==() you can use this macro to compare two instances. You will also need to add a StringFrom() function like those found in SimpleString. This is for printing the objects when the check failed.
- CHECK_COMPARE(first, relop, second) - checks thats a relational operator holds between two entities. On failure, prints what both operands evaluate to.
- CHECK_THROWS(expected_exception, expression) - checks if expression throws expected_exception (e.g. std::exception). CHECK_THROWS is only available if CppUTest is built with the Standard C++ Library (default).
- STRCMP_EQUAL(expected, actual) - checks const char* strings for equality using strcmp().
- STRNCMP_EQUAL(expected, actual, length) - checks const char* strings for equality using strncmp().
- STRCMP_NOCASE_EQUAL(expected, actual) - checks const char* strings for equality, not considering case.
- STRCMP_CONTAINS(expected, actual) - checks whether const char* actual contains const char* expected.
- LONGS_EQUAL(expected, actual) - compares two numbers.
- UNSIGNED_LONGS_EQUAL(expected, actual) - compares two positive numbers.
- BYTES_EQUAL(expected, actual) - compares two numbers, eight bits wide.
- POINTERS_EQUAL(expected, actual) - compares two pointers.
- DOUBLES_EQUAL(expected, actual, tolerance) - compares two floating point numbers within some tolerance
- FUNCTIONPOINTERS_EQUAL(expected, actual) - compares two void (*)() function pointers
- MEMCMP_EQUAL(expected, actual, size) - compares two areas of memory
- BITS_EQUAL(expected, actual, mask) - compares expected to actual bit by bit, applying mask
- FAIL(text) - always fails

NOTE Most macros have _TEXT() equivalents like CHECK_TEXT(), which are not explicitly listed here.

CHECK_EQUAL Warning:

CHECK_EQUAL(expected, actual) can produce misleading error reports as it will evaluate expected and actual more than once. This especially leads to confusions when used with mocks. This happens if the mock function expects to be called exactly once, since the macro needs to evaluate the actual expression more than once. The problem does not occur with type specific checks (e.g. LONGS_EQUAL()), so it is recommended to use them if possible. Instead of:

```
CHECK_EQUAL(10, mock_returning_11())
```

which reports: Mock Failure: Unexpected additional call, rather use

```
LONGS_EQUAL(10, mock_returning_11()) // reports actual different from expected
```

This issue could only be avoided with advanced language features like C++ templates, which would violate the CppUTest design goal portability to old environments.

Setup and Teardown

Every test group can have a setup and a teardown method. The setup method is called *before* each test and the teardown method is called *after* each test.

You can define setup and teardown like this:

```
TEST_GROUP(FooTestGroup)
{
   void setup()
   {
      // Init stuff
   }

   void teardown()
   {
      // Uninit stuff
   }
};

TEST(FooTestGroup, Foo)
{
   // Test FOO
}

TEST(FooTestGroup, MoreFoo)
{
   // Test more FOO
}

TEST_GROUP(BarTestGroup)
{
   void setup()
```

```
    {
        // Init Bar
    }
};

TEST(BarTestGroup, Bar)
{
    // Test Bar
}
```

The test execution of this will *likely* (no guarantee of order in CppUTest) be:

- setup BarTestGroup
- Bar
- setup FooTestGroup
- MoreFoo
- teardown FooTestGroup
- setup FooTestGroup
- Foo
- teardown FooTestGroup

Command line Switches

- *-c* colorize output, print green if OK, or red if failed
- *-g group* only run test whose group contains the substring *group*
- *-k* package name, Add a package name in JUnit output (for classification in CI systems)
- *-lg* print a list of group names, separated by spaces
- *-ln* print a list of test names in the form of *group.name*, separated by spaces
- *-n name* only run test whose name contains the substring *name*
- *-ojunit* output to JUnit ant plugin style xml files (for CI systems)
- *-oteamcity* output to xml files (as the name suggests, for TeamCity)
- *-p* run tests in a separate process.
- *-r#* repeat the tests some number (#) of times, or twice if # is not specified. This is handy if you are experiencing memory leaks. A second run that has no leaks indicates that someone is allocating statics and not releasing them.
- *-sg group* only run test whose group exactly matches the string *group*
- *-sn name* only run test whose name exactly matches the string *name*
- *-v* verbose, print each test name as it runs
- *-xg group* exclude tests whose group contains the substring *group* (v3.8)
- *-xn name* exclude tests whose name contains the substring *name* (v3.8)
- *"TEST(group, name)"* only run test whose group and name matches the strings group and name. This can be used to copy-paste output from the -v option on the command line.

You can specify multiple -s|sg, -s|sn and "TEST(group, name)" parameters:

Specifying only test groups with multiple -s|sg parameters will run all tests in those groups, since no test name matches all test names.

Specifying only test names with multiple -s|sn parameters will run all tests whose names match, since no test group matches all test groups.

Mixing multiple -s|sg and -s|sn parameters (or using "TEST(group, name)" will only run tests whose groups match as well as their names.

Combining one -xg parameter with one -xn parameter will run only those tests that satisfy both criteria.

Combining -s|sg with -xn, or -s|sn with -xg will run only those tests that satisfy both criteria.

Specifying several -xg or -xn with each other or in other combinations has no effect.

NOTE Be careful with *-p*:

- Some systems do not support this feature, in which case tests will fail with a suitable message.
- Using *-p* to run tests in a separate process can have unexpected side effects.
- While running tests in a separate process can help to get more information about an unexpected crash, when an expected crash is part of the test scenario, the *-p* command line option should not be used, but running in a separate process should be enabled on a per-test basis like this:

```
TestRegistry::getCurrentRegistry()->setRunTestsInSeperateProcess();
```

Examples for this can be found in CppUTests's own tests.

Memory Leak Detection

CppUTest has memory leak detection support on a per-test level. This means that it automatically checks whether the memory at the end of a test is the same as at the beginning of the test.

Explained another way:

1. Pre-setup -> Record the amount of memory used
2. Do setup
3. Run test
4. Do teardown
5. Post-teardown -> Check whether the amount of memory is the same

The memory leak detector consists of three parts:

- Memory leak detector base (including linker symbols for operator new)
- Macros overloading operator new for additional file and line info
- Macros overloading malloc/free for memory leak detection in C

All of these are on by default. For the macro support, you'll need to add to your Makefile:

```
CXXFLAGS += -include $(CPPUTEST_HOME)/include/CppUTest/MemoryLeakDetectorNewMacros.h
CFLAGS += -include $(CPPUTEST_HOME)/include/CppUTest/MemoryLeakDetectorMallocMacros.h
```

These are added by default when you use the CppUTest Makefile helpers.

Turning memory leak detection off and on

If you want to disable the memory leak detection (because you have too many memory leaks?) then you can do so in several ways. However, it is strongly recommended to keep the memory leak detector on and fix your memory leaks (and your static initialization issues) as this tends to lead to higher quality code.

You can turn the memory leak detection completely off by adding this to your main:

```
int main(int argc, char** argv)
{
    MemoryLeakWarningPlugin::turnOffNewDeleteOverloads();
    return CommandLineTestRunner::RunAllTests(argc, argv);
}
```

You can do the same by turning it off on a test by test basis, by adding this to the test group:

```
void setup()
{
    MemoryLeakWarningPlugin::turnOffNewDeleteOverloads();
}

void teardown()
{
    MemoryLeakWarningPlugin::turnOnNewDeleteOverloads();
}
```

(Do not forget to turn it on in the teardown again!)

If you want to completely disable memory leak detection then you can do so by building CppUTest with "configure –disable-memory-leak-detection" or passing -DCPPUTEST_MEM_LEAK_DETECTION_DISABLED to the compiler when compiling CppUTest.

Conflicts with operator new macros (STL!)

It is common for the memory leak detection macros to conflict with an overloaded operator new or with STL. This is because the macro replaces the call to operator new to a call to operator new with **FILE**, and **LINE**. If you overload operator new, it will replace your overloaded definition resulting in a compiler error. This is common when using the Standard C++ library (STL).

Resolving conflicts with STL

The easiest way is to not pass the –include MemoryLeakDetectionNewMacros.h to the compiler, but this would lose all your file and line information. So this is not recommended. An alternative is to create your own NewMacros.h file which will include the STL file *before* the new macro is defined. For example, the following NewMacros file can be used for a program that uses std::list:

```
#include "list"
#include "CppUTest/MemoryLeakDetectorNewMacros.h"
```

Now the call to the compiler needs to be -include MyOwnNewMacros.h and this will ensure that the operator new overload is *before* the define and thus all compiler errors are resolved.

Conflicts with my own overload!

This one is harder (and luckily less common). You can solve this the same way as the conflict with the STL, but it's probably better to use a finer grained control. So, instead you can temporary disable the new macros, overload operator new, enable the new macro again. This can be done with the following code:

```
class NewDummyClass
{
public:
#if CPPUTEST_USE_NEW_MACROS
   #undef new
#endif
   void* operator new (size_t size, int additional)
#if CPPUTEST_USE_NEW_MACROS
   #include "CppUTest/MemoryLeakDetectorNewMacros.h"
#endif
   {
      // Do your thing!
   }
};
```

Yes, its ugly. But usually people don't overload operator new everywhere. If you do, consider turning off the new macro completely.

Conflicts with MFC

Tbd

Test Plugins

Test plugins let you add a pre-action and a post-action to each test case. Plugin examples:

- Memory leak detector (provided)
- Pointer restore mechanism (provided) - helpful when tests overwrite a pointer that must be restored to its original value after the test. This is especially helpful when a pointer to a function is modified for test purposes.
- IEEE754 Floating point exceptions (provided; v3.8) - automatically checks whether any floating point exception flags are set at the end of every test and if so, fails the test.
- All Mutex's released - you could write a plugin that checks that any Mutexs or other shared resource is released before the test exits.

Complete Documentation for provided plugins can be found on the Plugin Manual page.

Scripts

There are some scripts that are helpful in creating your initial header, source, and Test files. These scripts save a lot of typing. See scripts/README.TXT from the CppUTest distribution.

Advanced

Customize CHECK_EQUAL to work with your types that support operator==()

Create the function

```
SimpleString StringFrom (const yourType&)
```

The Extensions directory has a few of these.

Building default checks with TestPlugin

- CppUTest can support extra checking functionality by inserting TestPlugins
- TestPlugin is derived from the TestPlugin class and can be inserted in the TestRegistry via the installPlugin method.
- All TestPlugins are called before and after running all tests and before and after running a single test (like Setup and Teardown). TestPlugins are typically inserted in the main.
- TestPlugins can be used for, for example, system stability and resource handling like files, memory or network connection clean-up.
- In CppUTest, the memory leak detection is done via a default enabled TestPlugin

How to run tests when they are linked in a library

In larger projects, it is often useful if you can link the tests in "libraries of tests" and then link them to the library of a component or link them all together to be able to run all the unit tests. Putting the tests in a library however causes an interesting problem because the lack of reference to the tests (due to the auto-registration of tests) causes the linker to discard the tests and it won't run any of them. There are two different work-arounds for this:

- You can use the IMPORT_TEST_GROUP macro to create a reference. This is typically done in the main.cpp or the main.h. You'll need to do this for every single TEST_GROUP (and the tests groups shouldn't be distributed over multiple files)
- When you use gnu linker (on linux, but not MacOSX) then you can use an additional linker option that will make sure the whole library is linked. You do this by adding the library to be linked between the "-Wl,-whole-archive" and the -Wl,-no-whole-archive" options. For example:

```
gcc -o test_executable production_library.a -Wl,-whole-archive test_library.a -Wl,-no-whole-archive $(OTHER_LIBRARIES)
```

C Interface

Sometimes, a C header will not compile under C++. For such cases, there are macros that allow you to specify test cases in a .c source, without involving C++ at all. There are also macro wrappers that pull these test cases into a .cpp source for CppUTest to work with. You will find all C macro definitions in TestHarness_c.h.

Here is a small example of how this is done.

First, the header of the function we want to test, PureCTests.h:

```
/** Legal C code that would not compile under C++ */
int private (int new);
```

Next, the C file that defines our tests, PureCTests.c:

```
#include "PureCTests_c.h" /** the offending C header */
#include "CppUTest/TestHarness_c.h"
#include "CppUtestExt/MockSupport_c.h"

/** Mock for function internal() */
int internal(int new)
{
    mock_c()->actualCall("internal")
            ->withIntParameters("new", new);
    return mock_c()->returnValue().value.intValue;
}

/** Implementation of function to test */
int private (int new)
{
    return internal(new);
}

/** Setup and Teardown per test group (optional) */
TEST_GROUP_C_SETUP(mygroup)
{
}
TEST_GROUP_C_TEARDOWN(mygroup)
{
    mock_c()->checkExpectations();
    mock_c()->clear();
}

/** The actual tests for this test group */
TEST_C(mygroup, test_success)
{
    mock_c()->expectOneCall("internal")->withIntParameters("new", 5)->andReturnIntValue(5);
    int actual = private(5);
    CHECK_EQUAL_C_INT(5, actual);
}
TEST_C(mygroup, test_mockfailure)
{
    mock_c()->expectOneCall("internal")->withIntParameters("new", 2)->andReturnIntValue(5);
    int actual = private(5);
    CHECK_EQUAL_C_INT(5, actual);
}
TEST_C(mygroup, test_equalfailure)
{
    mock_c()->expectOneCall("internal")->withIntParameters("new", 5)->andReturnIntValue(2);
    int actual = private(5);
    CHECK_EQUAL_C_INT(5, actual);
}
```

Finally, the .cpp file that wraps it all up for CppUTest, PureCTests.cpp:

```
#include "CppUTest/CommandLineTestRunner.h"
#include "CppUTest/TestHarness_c.h"

/** For each C test group */
TEST_GROUP_C_WRAPPER(mygroup)
{
    TEST_GROUP_C_SETUP_WRAPPER(mygroup); /** optional */
    TEST_GROUP_C_TEARDOWN_WRAPPER(mygroup); /** optional */
};
```

```
/** For each C test */
TEST_C_WRAPPER(mygroup, test_success);
TEST_C_WRAPPER(mygroup, test_mockfailure);
TEST_C_WRAPPER(mygroup, test_equalfailure);

/** Test main as usual */
int main(int ac, char** av)
{
    return RUN_ALL_TESTS(ac, av);
}
```

You can leave out TEST_GROUP_C_SETUP() / TEST_GROUP_C_TEARDOWN() and TEST_GROUP_C_SETUP_WRAPPER() / TEST_GROUP_C_TEARDOWN_WRAPPER(), if you don't need them.

The following assertion macros are supported in the pure C interface:

```
CHECK_EQUAL_C_INT(expected,actual);
CHECK_EQUAL_C_REAL(expected,actual,threshold);
CHECK_EQUAL_C_CHAR(expected,actual);
CHECK_EQUAL_C_STRING(expected,actual);
CHECK_EQUAL_C_POINTER(expected,actual); /* v3.8 */
CHECK_EQUAL_C_BITS(expected, actual, mask); /* v3.8, pending */
FAIL_TEXT_C(text);
FAIL_C();
CHECK_C(condition);
```

These macros ensure tests get terminated in a way appropriate for pure C code.

Using Google Mock

You can use Google Mock directly in CppUTest. In order to do this, you'll need to build with the real google mock. You do that like this:

```
$ GMOCK_HOME = /location/of/gmock
$ configure --enable-gmock
$ make
$ make install
```

Then in your tests, you can #include "CppUTestExt/GMock.h". Do remember to set the CPPUTEST_USE_REAL_GMOCK define (pass -DCPPUTEST_USE_REAL_GMOCK to the compiler). Also, do not forget to link the CppUTestExt library.

This way you can use GMock directly in your code. For example:

```
class MyMock : public ProductionInterface
{
public:
        MOCK_METHOD0(methodName, int());
};

TEST(TestUsingGMock, UsingMyMock)
{
        NiceMock<MyMock> mock;
        EXPECT_CALL(mock, methodName()).Times(2).WillRepeatedly(Return(1));

        productionCodeUsing(mock);
}
```

The above will probably leak to the memory leak detector complaining about memory leaks (in google mock). These aren't really memory leaks, but they are static data that gtest (unfortunately) allocates on the first run. There are a couple of ways to get around that. First, you turn of the memory leak detector (see Memory Leak Detection). A better solutions is to use the GTestConvertor.

You can do that by adding the following code to your main:

```
#include "CppUTestExt/GTestConvertor.h"

int main(int argc, char** argv)
{
    GTestConvertor convertor;
    return CommandLineTestRunner::RunAllTests(argc, argv);
}
```

The most important line to add is the GTestConvertor. Make sure you define the CPPUTEST_USE_REAL_GTEST to signal the gtest dependency. (by adding -DCPPUTEST_USE_REAL_GTEST to the compiler)

Running Google Tests in CppUTest

People feel wonderfully religious about unit testing tools. Of course, we feel strongly that CppUTest beats other tools when you actually test-drive your software. But unfortunately, people still use tools like GoogleTest (which is actually not as bad as e.g. CppUnit). It is unlikely that we're going to

convince people to use CppUTest instead, so therefore we've written some integration code where you can actually link google test and CppUTest tests together in one binary (with the CppUTest test runner). This also gives you some additional benefits:

- You get memory leak detection over your google tests…
- You don't get the verbose gtest output
- You can use both CppUMock and GMock in one project

The way to do this is really quite simple. First, you'll need to compile CppUtest with the GTest support enabled (by default this is off to prevent the dependency with GTest). You do that this way (assuming you want to use GMock too):

```
$ GMOCK_HOME = /location/of/gmock
$ configure --enable-gmock
$ make
$ make install
```

Or, if you don't want to use GMock and only GTest then:

```
$ GMOCK_HOME = /location/of/gtest
$ configure --enable-real-gtest
$ make
$ make install
```

To let CppUTest know there are gtest being linked, you'll need to add the following to the main:

```
#include "./include/CppUTestExt/GTestConvertor.h"

int main(int argc, char** argv)
{
    GTestConvertor convertor;
    convertor.addAllGTestToTestRegistry();
    return CommandLineTestRunner::RunAllTests(argc, argv);
}
```

(of course, you'll need make sure you link also gtest and also add it to the include path.)