# Clark’s Testing Harness

<http://github.com/wowus/CTH>

## Introduction

Hello, world. I made this little testing harness for you.

I’ve spent way too much time in my life searching for the “best” C++ testing harness. During the course of my search, I’ve come up with a few criteria that would be my ideal.

1. **It must be simple.** I like writing code a lot more than documentation, so I should do my best to remove as much need for documentation as possible. Therefore, keep the whole damn thing as simple at all costs.
2. **It must be scalable.** I don’t want to be using a testing framework only to figure out it chokes when you have more than one thousand tests. A good harness will do its best to utilize the hardware to its full potential, and have as little overhead as possible.
3. **It must be easily extensible.** No tool will ever be “just right” for you. If you want perfect, you have to do it yourself. Therefore, the harness should be as easy to modify as possible without breaking anything, and without having to think too much.
4. **Parallelism is really cool.** And because it’s cool, this framework will exploit it as much as possible. More on the internals of this later.

These four points compose the philosophy I will follow with this project. If I ever violate any of these principles, submit a bug report!

## Getting Started

For a quick overview, please check out the “Sample” project bundled with CTH. I go through most of the features there, along with set up instructions and tips. I also highly recommend reading through Test.h. It has a lot of goodies in it, and is only about 200 lines long.

## Setting Up Your Build Environment

So you want to start testing, eh? Well, here’s a step-by-step guide on getting your tests set up. An example of this is in the /Sample directory.

1. Compile the “Test” project. That is, the thing you got from the git repository. You might want to change the project settings and configuration to suit your needs. I have it set exclusively to x64 for my personal needs, because my Visual Studio 2010 is bugged (x86 doesn’t work). If you got the code pre-built, this step is very skippable.
2. Create a new project in your solution next to the one you will be writing tests for. Call it “Test” or something descriptive like that. It should be a console application.
3. Copy the build output into a place easily accessible to the project you will be using for testing. Ensure the .dll is in a directory accessible by the test executable, and the .lib is in a directory accessible by the linker.
4. Copy “Test.h” into a place where it can be easily included.
5. Use the Sample project as a reference for setting up the files, and the directory structure with the /Tests/ folder is highly recommended, along with PCH usage.
6. Finally, as a post-build step, use “$(TargetPath)”, including the quotes. This will run the tests as a part of your builds. So when they fail, your build fails. Also, test output should get redirected to your “errors” windows in visual studio.

## Tests

First of all, there is only one, simple way to define tests:

TEST(MyCoolTestName)

{

// MY AWESOME TEST CODE HERE.

}

## Suites

Tests should be grouped into suites. You do this by having multiple .cpp files with tests in them. Tests in the same suite (file) may not have the same name as other tests within the same suite, but they CAN have identical names between suites. Say you have FileA.cpp and FileB.cpp. Test “foo” can be in both FileA AND FileB, and the framework will execute and handle each test correctly.

When you organize your suites, make it easy for people to see all the tests regarding “X”, where “X” is a single unit of your code. The level of abstraction of “X” is up to you; just try to make sense to the reader. The way I usually do it is that for every .cpp file in the main program, there is one .cpp to test it.

## Checks and Assertations

Once that is set up, you can start doing assertions. These come in two categories – CHECK\_\* and ASSERT\_\* (\* denotes wildcard) macros. When an ASSERT\_\* macro fails, it terminates the current test, and the test runner will move on to the next one. In contrast, CHECK\_\* macros will NOT terminate the current test if they fail, and flow will continue as usual, with the error being logged.

Now, you may be thinking to yourself: “But what goes in that wildcard spot?” Below is a comprehensive list of all the CHECK/ASSERT macros.

|  |  |
| --- | --- |
| ASSERT\_TRUE(condition) | CHECK\_TRUE(condition) |
| ASSERT\_FALSE(condition) | CHECK\_FALSE(condition) |
| ASSERT\_EQUAL(expected, actual) | CHECK\_EQUAL(expected, actual) |
| ASSERT\_CLOSE(expected, actual, tolerance) | CHECK\_CLOSE(expected, actual, tolerance) |
| ASSERT\_NULL(pointer) | CHECK\_NULL(pointer) |
| ASSERT\_VALID(pointer) | CHECK\_VALID(pointer) |

As you can see, every ASSERT has a corresponding CHECK, and vice versa. Some of these macros are self-explanatory (\*\_TRUE, \*\_FALSE, \*\_EQUAL, \*\_NULL), but the other two need a wee bit of explanation.

\*\_CLOSE is used to compare floating point values. Unfortunately, we can’t perfectly represent floating point numbers (think about how you’d represent 1/3 in decimal). We have to store approximations. This means that floating point comparison is almost impossible to get right. Therefore, with the \*\_CLOSE macro, you supply the expected and actual value, and then you tell it exactly how close they can be to each other. So if you want them to be within 0.00001 of each other, then that will be your tolerance parameter.

\*\_VALID is just the inverse of \*\_NULL. Therefore, instead of checking if the pointer is null, it check if it’s not null. VALID just sounds and looks better than NOT\_NULL.

As a short little side note, \*\_EQUAL works perfectly for C strings, and for all other types calls operator==, so have no fear about that.

## Failing a Test

If you want to manually fail a test because of a custom condition (such as an exception being caught), you can call TERMINATE\_TEST(message) with an informative message about why the test was terminated. If a TERMINATE\_TEST is ever hit, it will stop the current test, and output an error (in the form of “message”).

## Performance

And, this is where CTH shines. This testing framework will scale beautifully. This is ensured by the controlled parallelism of the test runs. If you have 100,000 tests, split across 1000 files, your tests will run across all your cores, and with extremely low overhead per test case. It takes a negligible amount of time to get a test started, and to finish a test. There are a few things you need to consider as a tester for this to happen nicely, though.

The way your tests are parallelized is as follows.

1. Your tests are separated into test lists. There is one test list per test suite.
2. Each test in a test list is executed IN PARALLEL with the rest of the test lists.

This has the implication that if your tests are in the same file, they will be run sequentially with the other tests in that file, and if they are in different files, they will be run in parallel. This offers you fine grain control over what gets to be sequential and what gets to be parallel. Be very careful about shared state being tested with this framework, as race conditions that only appear in your tests can be quite confusing.

## Modifying CTH

I tried to make this harness as extensible as possible, and any suggestions on improving that are best sent to the mailing list mentioned in the “helping out” section. So far, I have a couple hooks in place for customization. These hooks and any others I may add to the code can be found in Config.cpp. I try my best to document as thoroughly as possible, but if confusion ever sets in, ask the mailing list!

Not only are there hooks in place, but I strive to make the source as succinct as possible. It shouldn’t take more than an hour of your time to get a good grasp of what I do and how I do it. In fact, after reading through, you should be able to implement a testing framework of your own!

## Getting Help

If you have any questions, feel free to ask on <http://groups.google.com/group/cth-discussion>. If you find a bug, or have a feature request, use the issue tracker on github.

## Helping Out

If you want to help with development, or have patches that need submitting, etc. Feel free to join the CTH-dev group, over at <http://groups.google.com/group/cth-dev>. This project, like any other, will never be finished. Instead, it will evolve.

## Thanks!

Wow, you actually read through all of that. Congratulations. Hope you enjoy this little project of mine, and maybe even contribute some things! Just remember the core principles, and everything should be fine and dandy. I’m so sorry for using visual studio 2010, but it just has a wonderful parallel\_for\_each that makes everything so elegant. The main .dll must be built with VS2010, but the header should work in any C++ compiler. I did my best to keep it as simple as possible for that sole purpose. Therefore, if you have a compiler that runs on Windows and a pre-build Test.dll, you should be fine.