# Clark’s Testing Harness

## 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 “how do I use this thing”, please check out the “Sample” project bundled with CTH. I go through all the features there, along with how to get it set up.

## Tests

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

TEST(MyCoolTestName)

{

// MY AWESOME TEST CODE HERE.

}

## 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.