Criterion Documentation

Release 0.1.0

Franklin "Snaipe" Mathieu

November 25, 2015

Contents

CHAPTER 1

Introduction

Criterion is a dead-simple, non-intrusive testing framework for the C programming language.

1.1 Philosophy

Most test frameworks for C require a lot of boilerplate code to set up tests and test suites – you need to create a main, then register new test suites, then register the tests within these suits, and finally call the right functions.

This gives the user great control, at the unfortunate cost of simplicity.

Criterion follows the KISS principle, while keeping the control the user would have with other frameworks.

1.2 Features

- Tests are automatically registered when declared.
- A default entry point is provided, no need to declare a main unless you want to do special handling.
- Test are isolated in their own process, crashes and signals can be reported and tested.
- Progress and statistics can be followed in real time with report hooks.

CHAPTER 2

Setup

2.1 Prerequisites

Currently, this library only works under *nix systems.

To compile the static library and its dependencies, GCC 4.9+ is needed.

To use the static library, GCC or Clang are needed.

2.2 Installation

```
$ git clone https://github.com/Snaipe/Criterion.git
$ cd Criterion
$ ./autogen.sh && ./configure && make && sudo make install
```

2.3 Usage

Given a test file named test.c, compile it with *-lcriterion*:

```
$ gcc -o test test.c -lcriterion
```

CHAPTER 3

Getting started

3.1 Adding tests

Adding tests is done using the `Test` macro:

```
#include <criterion/criterion.h>

Test(suite_name, test_name) {
    // test contents
}
```

`suite_name` and `test_name` are the identifiers of the test suite and the test, respectively. These identifiers must follow the language identifier format.

Tests are automatically sorted by suite, then by name using the alphabetical order.

3.2 Asserting things

Assertions come in two kinds:

- `assert*` are assertions that are fatal to the current test if failed; in other words, if the condition evaluates to `false`, the test is marked as a failure and the execution of the function is aborted.
- `expect*` are, in the other hand, assertions that are not fatal to the test. Execution will continue even if the condition evaluates to `false`, but the test will be marked as a failure.

`assert()` and `expect()` are the most simple kinds of assertions criterion has to offer. They both take a mandatory condition as a first parameter, and an optional failure message:

```
#include <string.h>
#include <criterion/criterion.h>

Test(sample, test) {
    expect(strlen("Test") == 4, "Expected \"Test\" to have a length of 4.");
    expect(strlen("Hello") == 4, "This will always fail, why did I add this?");
    assert(strlen("") == 0);
}
```

On top of those, more assertions are available for common operations:

- `{assert,expect}_not(Actual, Expected, [Message])`
- `{assert,expect}_eq(Actual, Expected, [Message])`

- `{assert,expect}_neq(Actual, Unexpected, [Message])`
- `{assert,expect}_lt(Actual, Expected, [Message])`
- `{assert,expect}_leq(Actual, Expected, [Message])`
- `{assert,expect}_gt(Actual, Expected, [Message])`
- `{assert,expect}_geq(Actual, Expected, [Message])`
- `{assert,expect}_float_eq(Actual, Expected, Epsilon, [Message])`
- `{assert,expect}_float_neq(Actual, Unexpected, Epsilon, [Message])`
- `{assert,expect}_strings_eq(Actual, Expected, [Message])`
- `{assert,expect}_strings_neq(Actual, Unexpected, [Message])`
- `{assert,expect}_strings_lt(Actual, Expected, [Message])`
- `{assert,expect}_strings_leq(Actual, Expected, [Message])`
- `{assert,expect}_strings_gt(Actual, Expected, [Message])`
- `{assert,expect}_strings_geq(Actual, Expected, [Message])`
- `{assert,expect}_arrays_eq(Actual, Expected, Size, [Message])`
- `{assert,expect}_arrays_neq(Actual, Unexpected, Size, [Message])`

3.3 Fixtures

Tests that need some setup and teardown can register functions that will run before and after the test function:

```c
#include <stdio.h>
#include <criterion/criterion.h>

void setup(void) {
    puts("Runs before the test");
}

void teardown(void) {
    puts("Runs after the test");
}

Test(suite_name, test_name, .init = setup, .fini = teardown) {
    // test contents
}
```

3.4 Testing signals

If a test receives a signal, it will by default be marked as a failure. You can, however, expect a test to only pass if a special kind of signal is received:

```c
#include <stddef.h>
#include <signal.h>
#include <criterion/criterion.h>
```

```
    // This test will fail
    Test(sample, failing) {
        int *ptr = NULL;
        *ptr = 42;
    }

    // This test will pass
    Test(sample, passing, .signal = SIGSEGV) {
        int *ptr = NULL;
        *ptr = 42;
    }
```

CHAPTER 4

Report Hooks

Report hooks are functions that are called at key moments during the testing process. These are useful to report statistics gathered during the execution.

A report hook can be declared using the `ReportHook` macro:

```
#include <criterion/criterion.h>
#include <criterion/hooks.h>

ReportHook(Phase)() {
}
```

The macro takes a Phase parameter that indicates the phase at which the function shall be run. Valid phases are described below.

4.1 Testing Phases

The flow of the test process goes as follows:

1. `PRE_ALL`: occurs before running the tests.
2. `PRE_INIT`: occurs before a test is initialized.
3. `PRE_TEST`: occurs after the test initialization, but before the test is run.
4. `ASSERT`: occurs when an assertion is hit
5. `TEST_CRASH`: occurs when a test crashes unexpectedly.
6. `POST_TEST`: occurs after a test ends, but before the test finalization.
7. `POST_FINI`: occurs after a test finalization.
8. `POST_ALL`: occurs after all the tests are done.

4.2 Hook Parameters

A report hook may take zero or one parameter. If a parameter is given, it is undefined behaviour if it is not a pointer type and not of the proper pointed type for that phase.

Valid types for each phases are:

- `struct criterion_test *` for `PRE_INIT` and `PRE_TEST`.

- `struct criterion_test_stats *` for `POST_TEST`, `POST_FINI`, and `TEST_CRASH`.
- `struct criterion_assert_stats *` for `ASSERT`.
- `struct criterion_global_stats *` for `POST_ALL`.

`PRE_ALL` does not take any parameter.

CHAPTER 5

Environment and CLI

Tests built with Criterion support environment variables to alter their runtime behaviour.

5.1 Environment Variables

- *CRITERION_ALWAYS_SUCCEED*: when set to *1*, the exit status of the test process will be 0, regardless if the tests failed or not.
- *CRITERION_NO_EARLY_EXIT*: when set to *1*, the test workers shall not call *_exit* when done and will properly return from the main and clean up their process space. This is useful when tracking memory leaks with *valgrind –tool=memcheck*.