Criterion Documentation

Release 1.1.0

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.
- TAP output format can be enabled with an option.
- Runs on Linux, FreeBSD, Mac OS X, and Windows (compiles only with Cygwin for the moment).
- xUnit framework structure

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, any GNU-C compatible compiler will suffice (GCC, Clang/LLVM, ICC, MinGW-GCC, ...).

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>
```

```c
// 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_SUITE`: occurs before a suite is initialized.
3. `PRE_INIT`: occurs before a test is initialized.
4. `PRE_TEST`: occurs after the test initialization, but before the test is run.
5. `ASSERT`: occurs when an assertion is hit
6. `TEST_CRASH`: occurs when a test crashes unexpectedly.
7. `POST_TEST`: occurs after a test ends, but before the test finalization.
8. `POST_FINI`: occurs after a test finalization.
9. `POST_SUITE`: occurs before a suite is finalized.
10. `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_set *` for `PRE_ALL`.
- `struct criterion_suite_set *` for `PRE_SUITE`.
- `struct criterion_test *` for `PRE_INIT` and `PRE_TEST`.
- `struct criterion_assert_stats *` for `ASSERT`.
- `struct criterion_test_stats *` for `POST_TEST`, `POST_FINI`, and `TEST_CRASH`.
- `struct criterion_suite_stats *` for `POST_SUITE`.
- `struct criterion_global_stats *` for `POST_ALL`.

CHAPTER 5

Environment and CLI

Tests built with Criterion expose by default various command line switchs and environment variables to alter their runtime behaviour.

5.1 Command line arguments

- `-h or --help`: Show a help message with the available switches.
- `-v or --version`: Prints the version of criterion that has been linked against.
- `-l or --list`: Print all the tests in a list.
- `-f or --fail-fast`: Exit after the first test failure.
- `--ascii`: Don't use fancy unicode symbols or colors in the output.
- `--pattern [PATTERN]`: Run tests whose string identifier matches the given shell wildcard pattern (see dedicated section below).
- `--no-early-exit`: The test workers shall not prematurely exit when done and will properly return from the main, cleaning up their process space. This is useful when tracking memory leaks with `valgrind --tool=memcheck`.
- `--always-succeed`: The process shall exit with a status of `0`.
- `--tap`: Enables the TAP (Test Anything Protocol) output format.
- `--verbose[=level]`: Makes the output verbose. When provided with an integer, sets the verbosity level to that integer.

5.2 Shell Wildcard Pattern

Patterns in criterion are matched against a test's string identifier with `fnmatch`.

Special characters used in shell-style wildcard patterns are:

Pattern	Meaning
`*`	matches everything
`?`	matches any character
`[seq]`	matches any character in *seq*
`[!seq]`	matches any character not in *seq*

A test string identifier is of the form `suite-name/test-name`, so a pattern of `simple/*` matches every tests in the `simple` suite, `*/passing` matches all tests named `passing` regardless of the suite, and `*` matches every possible test.

5.3 Environment Variables

Environment variables are alternatives to command line switches when set to 1.

- `CRITERION_ALWAYS_SUCCEED`: Same as `--always-succeed`.
- `CRITERION_NO_EARLY_EXIT`: Same as `--no-early-exit`.
- `CRITERION_ENABLE_TAP`: Same as `--tap`.
- `CRITERION_FAIL_FAST`: Same as `--fail-fast`.
- `CRITERION_USE_ASCII`: Same as `--ascii`.
- `CRITERION_VERBOSITY_LEVEL`: Same as `--verbose`. Sets the verbosity level to its value.
- `CRITERION_TEST_PATTERN`: Same as `--pattern`. Sets the test pattern to its value.

CHAPTER 6

F.A.Q

Q. When running the test suite in Windows' cmd.exe, the test executable prints weird characters, how do I fix that?

A. Windows' `cmd.exe` is not an unicode ANSI-compatible terminal emulator. There are plenty of ways to fix that behaviour:

- Pass `--ascii` to the test suite when executing.
- Define the `CRITERION_USE_ASCII` environment variable to `1`.
- Get a better terminal emulator, such as the one shipped with Git or Cygwin.

Q. I'm having an issue with the library, what can I do ?

A. Open a new issue on the github issue tracker, and describe the problem you are experiencing.