Criterion Documentation

Release 1.3.0

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 (Compiling with MinGW GCC).
- 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 4.6+ 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
$ LOCAL_INSTALL=/usr .ci/install-libcsptr.sh
$ mkdir build && cd $_ && cmake -DCMAKE_INSTALL_PATH=/usr ..
$ 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:

- `cr_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.
- `cr_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.

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

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

- `cr_assert_null(Ptr, [Message])`: passes if Ptr is NULL.
- `cr_assert_eq(Actual, Expected, [Message])`: passes if Actual == Expected.

- `cr_assert_lt(Actual, Expected, [Message])`: passes if Actual < Expected.
- `cr_assert_leq(Actual, Expected, [Message])`: passes if Actual <= Expected.
- `cr_assert_gt(Actual, Expected, [Message])`: passes if Actual > Expected.
- `cr_assert_geq(Actual, Expected, [Message])`: passes if Actual >= Expected.
- `cr_assert_float_eq(Actual, Expected, Epsilon, [Message])`: passes if Actual == Expected with an error of Epsilon.
- `cr_assert_arrays_eq(Actual, Expected, Size, [Message])`: passes if all elements of Actual (from 0 to Size - 1) are equals to those of Expected.
- `cr_assert_arrays_eq_cmp(Actual, Expected, Size, Cmp, [Message])`: Same as `arrays_eq` but equality is defined by the result of the binary Cmp function.

Equality and lexical comparison assertions are also available for strings:

- `cr_assert_strings_eq(Actual, Expected, [Message])`
- `cr_assert_strings_lt(Actual, Expected, [Message])`
- `cr_assert_strings_leq(Actual, Expected, [Message])`
- `cr_assert_strings_gt(Actual, Expected, [Message])`
- `cr_assert_strings_geq(Actual, Expected, [Message])`

And some assertions have a logical negative counterpart:

- `cr_assert_not(Condition, [Message])`
- `cr_assert_not_null(Ptr, [Message])`
- `cr_assert_neq(Actual, Unexpected, [Message])`
- `cr_assert_float_neq(Actual, Unexpected, Epsilon, [Message])`
- `cr_assert_strings_neq(Actual, Unexpected, [Message])`
- `cr_assert_arrays_neq(Actual, Unexpected, Size, [Message])`
- `cr_assert_arrays_neq_cmp(Actual, Unexpected, Size, Cmp, [Message])`

Of course, every `assert` has an `expect` counterpart.

Please note that `arrays_(n)eq` assertions should not be used on padded structures – please use `arrays_(n)eq_cmp` instead.

3.3 Configuring tests

Tests may receive optional configuration parameters to alter their behaviour or provide additional metadata.

3.3.1 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
}
```

If a setup crashes, you will get a warning message, and the test will be aborted and marked as a failure. Is a teardown crashes, you will get a warning message, and the test will keep its result.

3.3.2 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;
}
```

This feature will also work (to some extent) on Windows for the following signals on some exceptions:

Signal	Triggered by
SIGSEGV	STATUS_ACCESS_VIOLATION, STATUS_DATATYPE_MISALIGNMENT, STATUS_ARRAY_BOUNDS_EXCEEDED, STATUS_GUARD_PAGE_VIOLATION, STATUS_IN_PAGE_ERROR, STATUS_NO_MEMORY, STATUS_INVALID_DISPOSITION, STATUS_STACK_OVERFLOW
SIGILL	STATUS_ILLEGAL_INSTRUCTION, STATUS_PRIVILEGED_INSTRUCTION, STATUS_NONCONTINUABLE_EXCEPTION
SIGINT	STATUS_CONTROL_C_EXIT
SIGFPE	STATUS_FLOAT_DENORMAL_OPERAND, STATUS_FLOAT_DIVIDE_BY_ZERO, STATUS_FLOAT_INEXACT_RESULT, STATUS_FLOAT_INVALID_OPERATION, STATUS_FLOAT_OVERFLOW, STATUS_FLOAT_STACK_CHECK, STATUS_FLOAT_UNDERFLOW, STATUS_INTEGER_DIVIDE_BY_ZERO, STATUS_INTEGER_OVERFLOW
SIGALRM	STATUS_TIMEOUT

See the windows exception reference for more details on each exception.

3.3.3 Configuration reference

Here is an exhaustive list of all possible configuration parameters you can pass:

Parameter	Type	Description
.description	const char *	Adds a description. Cannot be `NULL`.
.init	void (*)(void)	Adds a setup function the be executed before the test.
.fini	void (*)(void)	Adds a teardown function the be executed after the test.
.disabled	bool	Disables the test.
.signal	int	Expect the test to raise the specified signal.

3.4 Setting up suite-wise configuration

Tests under the same suite can have a suite-wise configuration – this is done using the `TestSuite` macro:

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

TestSuite(suite_name, [params...]);

Test(suite_name, test_1) {
}

Test(suite_name, test_2) {
}
```

Configuration parameters are the same as above, but applied to the suite itself.

Suite fixtures are run *along with* test fixtures.

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.

Note: there are no guarantees regarding the order of execution of report hooks on the same phase. In other words, all report hooks of a specific phase could be executed in any order.

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`.

For instance, these are valid report hook declarations for the `PRE_TEST` phase:

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

ReportHook(PRE_TEST)() {
    // not using the parameter
}

ReportHook(PRE_TEST)(struct criterion_test *test) {
    // using the parameter
}
```

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). (*nix only)
- `--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`. This feature is only available on *nix systems where `fnmatch` is provided.

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. (*nix only)

CHAPTER 6

Changing the internals

6.1 Providing your own main

If you are not satisfied with the default CLI or environment variables, you can define your own main function.

6.1.1 Configuring the test runner

You'd usually want to configure the test runner before calling it. Configuration is done by setting fields in a global variable named `criterion_options` (include criterion/options.h).

Here is an exhaustive list of these fields:

Field	Type	Description
logging_threshold	enum criterion_logging_level	The logging level
output_provider	struct criterion_output_provider *	The output provider (see below)
no_early_exit	bool	True iff the test worker should exit early
always_succeed	bool	True iff criterion_run_all_tests should always returns 1
use_ascii	bool	True iff the outputs should use the ASCII charset
fail_fast	bool	True iff the test runner should abort after the first failure
pattern	const char *	The pattern of the tests that should be executed

6.1.2 Starting the test runner

The test runner can be called with `criterion_run_all_tests`. The function returns 0 if one test or more failed, 1 otherwise.

6.2 Implementing your own output provider

In case you are not satisfied by the default output provider, you can implement yours. To do so, simply set the `output_provider` option to your custom output provider.

Each function contained in the structure is called during one of the standard phase of the criterion runner.

For more insight on how to implement this, see other existing output providers in `src/log/`.

CHAPTER 7

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, along with the platform you are running criterion on.