Criterion Documentation

Release 2.3.3-161-gb67f3a2-dirty

Franklin "Snaipe" Mathieu

Jan 03, 2022

CONTENTS

CHAPTER

ONE

INTRODUCTION

Criterion is a dead-simple, non-intrusive unit testing framework for C and C++.

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

- C99 and C++11 compatible.
- Tests are automatically registered when declared.
- Implements a xUnit framework structure.
- 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.
- Unified interface between C and C++: include the criterion header and it *just* works.
- Supports parameterized tests and theories.
- 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, macOS, and Windows (Compiling with MinGW GCC and Visual Studio 2015+).

CHAPTER

TWO

SETUP

2.1 Prerequisites

The library is supported on Linux, macOS, FreeBSD, and Windows.

The following compilers are supported to compile both the library and the tests:

- GCC 4.9+ (Can be relaxed to GCC 4.6+ when not using C++)
- Clang 3.4+
- MSVC 14+ (Included in Visual Studio 2015 or later)

2.2 Building from source

First, install dependencies:

- C/C++ compiler
- Meson, Ninja
- CMake (for subprojects)
- pkg-config
- libffi (libffi-dev)
- libgit2 (libgit2-dev)

Other runtime dependencies will be bundled if they are not available on the system (BoxFort, debugbreak, klib, nanomsg, nanopb).

Clone this repository:

```
$ git clone --recursive https://github.com/Snaipe/Criterion
```

Then, run the following commands to build Criterion:

```
$ meson build
$ ninja -C build
```

2.3 Installing the library and language files (Linux, macOS, FreeBSD)

Run with an elevated shell:

```
$ ninja -C build install
```

2.4 Usage

To compile your tests with Criterion, you need to make sure to:

1. Add the include directory to the header search path
2. Install the library to your library search path
3. Link Criterion to your executable.

This should be all you need.

CHAPTER

THREE

GETTING STARTED

3.1 Adding tests

Adding tests is done using the `Test` macro:

Test(Suite, Name, ...)
Defines a new test.

Parameters

- **Suite** – The name of the test suite containing this test.
- **Name** – The name of the test.
- **...** – An optional sequence of designated initializer key/value pairs as described in the *criterion_test_extra_data* structure (see criterion/types.h).

  Example: .exit_code = 1

Example:

```
#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. See *Assertion reference* for a complete list.

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:

```
#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. If 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:

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

// This test will fail
```

(continues on next page)

(continued from previous page)

```
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:

struct **criterion_test_extra_data**
Contains all the options that can be set for a test, through the Test and TestSuite macros, or other means.

Public Members

void (***init**)(void)
The setup test fixture.

This function, if provided, will be executed during the initialization of the test.

void (***fini**)(void)
The teardown test fixture.

This function, if provided, will be executed during the finalization of the test.

int **signal**
The expected signal to be raised by the test.

If the test does not raise the specified signal, then the test is marked as failed.

A value of 0 means that is it not expected for the test to raise any signal.

int **exit_code**
The expected exit status to be returned by the test.

By default, criterion exits the test process with a value of 0. If it is expected for the test to exit with a non-zero status, this option can be used.

bool **disabled**
If `true`, skips the test.

The test will still appear in the test list, but will be marked as skipped and will not be executed.

const char ***description**
The long description of a test.

If a description is provided, it will be printed in test reports, and logged if the runner runs in verbose mode.

double **timeout**
The timeout for the test, in seconds.

If the realtime execution of a test takes longer than the specified value, then the test is immediately aborted and reported as timing out.

A value of `0` is equivalent to `+INFINITY` and means that the test does not timeout.

It is unspecified behaviour for the value of `timeout` to be negative or `NaN`.

void ***data**
Extra user data.

This field is currently unused.

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:

TestSuite(Name, ...)
Explicitely defines a test suite and its options.

Parameters

- **Name** – The name of the test suite.
- **...** – An optional sequence of designated initializer key/value pairs as described in the *criterion_test_extra_data* structure (see criterion/types.h). These options will provide the defaults for each test.

Example:

```
#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

FOUR

ASSERTION REFERENCE

This is an exhaustive list of all assertion macros that Criterion provides.

Note: This documents the new (but experimental) assertion API. To see the documentation of the old API, see: old-assertions-ref.

The new assertions API is centered around the use of criteria pseudo-functions to check various properties of the values being tested. Of note, there no longer are `cr_assert_<xyz>` macros, and instead all functionality has been merged into the `cr_assert` and `cr_expect` macros. For instance, instead of using `cr_assert_eq(1, 2)` one must use instead `cr_assert(eq(int, 1, 2))`, which involves the `eq` criterion and the `int` type tag.

To use the new assertion API, one must include the `<criterion/new/assert.h>` header.

All `assert` macros may take an optional `printf` format string and parameters.

4.1 Assertion API

group **AssertAPI**
Assertion API.

Defines

cr_fail(Format, ...)
Mark the test as failed.

The test is marked as a failure, printing the formatted string if provided, and the execution continues.

Parameters
- **Format** – **[in]** (optional) Printf-like format string
- **...** – **[in]** Format string parameters

cr_fatal(Format, ...)
Abort and mark the test as failed.

The test is marked as a failure, printing the formatted string if provided, and the execution of the test is aborted.

Parameters
- **Format** – **[in]** (optional) Printf-like format string
- **...** – **[in]** Format string parameters

cr_skip(Format, ...)
Abort and mark the test as skipped.

The test is marked as skipped, printing the formatted string if provided, and the execution of the test is aborted.

Parameters
- **Format** – **[in]** (optional) Printf-like format string
- **...** – **[in]** Format string parameters

cr_assert(Criterion, Format, ...)
Assert that a criterion is true and abort if it is not.

cr_assert evaluates the passed criterion and passes if it has a non-zero value.

The criterion may be any C expression of non-void type, in which case the assertion value will be `!!Criterion`. Alternatively, the criterion may be any of the valid pseudo-functions described in the *Criteria list*.

If the evaluated criterion is zero, then cr_fatal(Format, ...) is called.

Parameters
- **Criterion** – **[in]** The Criterion to evaluate
- **Format** – **[in]** (optional) Printf-like format string
- **...** – **[in]** Format string parameters

cr_expect(Criterion, Format, ...)
Expect that a criterion is true and fail if it is not.

cr_expect evaluates the passed criterion and passes if it has a non-zero value.

The criterion may be any C expression of non-void type, in which case the assertion value will be `!!Criterion`. Alternatively, the criterion may be any of the valid pseudo-functions described in the *Criteria list*.

If the evaluated criterion is zero, then cr_fail(Format, ...) is called.

Parameters
- **Criterion** – **[in]** The Criterion to evaluate
- **Format** – **[in]** (optional) Printf-like format string
- **...** – **[in]** Format string parameters

cr_assert_user(File, Line, FailFunc, Criterion, Format, ...)
cr_assert_user is an utility macro to help users implement their own assertions easily.

Users may pass file and line information. The function behaves like cr_assert and cr_expect, in that it evaluates the criterion to determine whether a test fails or not.

When the criterion evaluates to zero, a failed assertion event is raised back to the runner, and then FailFunc is called without parameters.

Parameters
- **File** – **[in]** The file in which the assertion has been called.
- **Line** – **[in]** The line number at which the assertion has been called.
- **FailFunc** – **[in]** The function to call on a failed assertion.
- **Criterion** – **[in]** The Criterion to evaluate.
- **Format** – **[in]** (optional) Printf-like format string.
- **...** – **[in]** Format string parameters.

4.2 Assertion Criteria

group **Criteria**

Criteria are pseudo-functions that evaluate to a boolean value.

Using criteria is recommended over standard C operator as they allow value pretty printing and other diagnostics on assertion failure.

Note: Criteria are neither symbols or macros, but pseudo-functions. They are only valid in the context of the assertion API when explicitely allowed and cannot be called alone.

Defines

not(Criterion)

Evaluates to `!Criterion`.

Parameters

- **Criterion** – **[in]** The criterion to negate

all(...)

Evaluates to true if all its arguments are true.

all() evaluates a sequence of criteria, and combines them into a single value with the logical and operator (&&).

Parameters

- **...** – **[in]** A sequence of criteria to evaluate

any(...)

Evaluates to true if any of its arguments is true.

any() evaluates a sequence of criteria, and combines them into a single value with the logical or operator (||).

Parameters

- **...** – **[in]** A sequence of criteria to evaluate

none(...)

Evaluates to true if none of its arguments is true.

none() evaluates a sequence of criteria, and combines their negation into a single value with the logical and operator (&&).

Parameters

- **...** – **[in]** A sequence of criteria to evaluate

group **TaggedCriteria**

Tagged Criteria are special criteria that take an optional type tag as their first argument.

Unless otherwise specified, all tagged criteria generally support any of the *supported tags*

Defines

eq(Tag, Actual, Expected)
Evaluates to `Actual == Expected`.

While this operator works for `flt`, `dbl`, and `ldbl`, the chance of having the values being exactly equal to each other is astronomically low due to round-off errors. Instead, please use as appropriate *ieee_ulp_eq* and *epsilon_eq*

Parameters
- **(optional)** (`Tag`) – **[in]** The type tag of the parameters
- **Actual** – **[in]** the actual value
- **Expected** – **[in]** the expected value

ne(Tag, Actual, Unexpected)
Evaluates to `Actual != Unexpected`.

While this operator works for `flt`, `dbl`, and `ldbl`, the chance of having the values being exactly equal to each other is astronomically low due to round-off errors. Instead, please use as appropriate *ieee_ulp_ne* and *epsilon_ne*

Parameters
- **(optional)** (`Tag`) – **[in]** The type tag of the parameters
- **Actual** – **[in]** the actual value
- **Unexpected** – **[in]** the unexpected value

lt(Tag, Actual, Reference)
Evaluates to `Actual < Reference`.

Parameters
- **(optional)** (`Tag`) – **[in]** The type tag of the parameters
- **Actual** – **[in]** the actual value
- **Reference** – **[in]** the reference value

le(Tag, Actual, Reference)
Evaluates to `Actual <= Reference`.

Parameters
- **(optional)** (`Tag`) – **[in]** The type tag of the parameters
- **Actual** – **[in]** the actual value
- **Reference** – **[in]** the reference value

gt(Tag, Actual, Reference)
Evaluates to `Actual > Reference`.

Parameters
- **(optional)** (`Tag`) – **[in]** The type tag of the parameters
- **Actual** – **[in]** the actual value
- **Reference** – **[in]** the reference value

ge(Tag, Actual, Reference)
Evaluates to `Actual >= Reference`.

Parameters
- **(optional)** (`Tag`) – **[in]** The type tag of the parameters
- **Actual** – **[in]** the actual value
- **Reference** – **[in]** the reference value

zero(Tag, Value)
Evaluates to true if `Value` is equal to the "zero value" of its type.

The zero value for primitive types and pointer types is the constant 0.

The zero value for c-strings (char *, wchar_t *) is the empty string, "" and L"" respectively.

User-defined types may be used, but what a zero value of these types mean depend on the language used.

In C, the function `bool cr_user_<type>_zero(const <type> *t)` must be defined, and will be invoked to check that `t` is a zero value.

In C++, the type corresponding to the passed tag, or the inferred type of Value if the tag is unspecified, must be default-constructible. The zero value of that type is the default construction of that type, and the value is compared against it with ==.

Parameters

- **(optional)** (Tag) – **[in]** The type tag of the parameter
- **Value** – **[in]** the value to compare for zeroness

ieee_ulp_eq(Tag, Actual, Expected, Ulp)

Evaluates to true if the IEEE 754 floating point numbers `Actual` and `Expected` are almost equal, by being within `Ulp` units from each other.

This method of comparison is more accurate when comparing two IEEE 754 floating point values when `Expected` is non-zero. When comparing against zero, please use *epsilon_eq* instead.

This tagged criterion only supports the `flt`, `dbl` and `ldbl` tags.

A good general-purpose value for Ulp is `4`.

Parameters

- **(optional)** (Tag) – **[in]** The type tag of the parameters
- **Actual** – **[in]** the actual value
- **Expected** – **[in]** the reference value
- **Ulp** – **[in]** the number of units in the last place used in the comparison

ieee_ulp_ne(Tag, Actual, Expected, Ulp)

Evaluates to true if the IEEE 754 floating point numbers `Actual` and `Expected` are different, by not being within `Ulp` units from each other.

This method of comparison is more accurate when comparing two IEEE 754 floating point values when `Expected` is non-zero. When comparing against zero, please use *epsilon_ne* instead.

This tagged criterion only supports the `flt`, `dbl` and `ldbl` tags.

A good general-purpose value for Ulp is `4`.

Parameters

- **(optional)** (Tag) – **[in]** The type tag of the parameters
- **Actual** – **[in]** the actual value
- **Expected** – **[in]** the reference value
- **Ulp** – **[in]** the number of units in the last place used in the comparison

epsilon_eq(Tag, Actual, Expected, Epsilon)

Evaluates to true if the floating point numbers `Actual` and `Expected` are almost equal, by being within an absolute `Epsilon` from each other (In other words, if `fabs(Actual + Expected) <= Epsilon`).

This method of comparison is more accurate when comparing two IEEE 754 floating point values that are near zero. When comparing against values that aren't near zero, please use *ieee_ulp_eq* instead.

This tagged criterion only supports the `flt`, `dbl` and `ldbl` tags.

It is recommended to have `Epsilon` be equal to a small multiple of the type epsilon (`FLT_EPSILON`, `DBL_EPSILON`, `LDBL_EPSILON`) and the input parameters.

Parameters

- **(optional)** (Tag) – **[in]** The type tag of the parameters
- **Actual** – **[in]** the actual value

- **Expected** – **[in]** the reference value
- **Epsilon** – **[in]** the epsilon used in the comparison

epsilon_ne(Tag, Actual, Expected, Epsilon)

Evaluates to true if the floating point numbers `Actual` and `Expected` are different, by not being within an absolute `Epsilon` from each other (In other words, if `fabs(Actual + Expected) > Epsilon`).

This method of comparison is more accurate when comparing two IEEE 754 floating point values that are near zero. When comparing against values that aren't near zero, please use *ieee_ulp_eq* instead.

This tagged criterion only supports the `flt`, `dbl` and `ldbl` tags.

It is recommended to have `Epsilon` be equal to a small multiple of the type epsilon (`FLT_EPSILON`, `DBL_EPSILON`, `LDBL_EPSILON`) and the input parameters.

Parameters

- **(optional)** (`Tag`) – **[in]** The type tag of the parameters
- **Actual** – **[in]** the actual value
- **Expected** – **[in]** the reference value
- **Epsilon** – **[in]** the epsilon used in the comparison

4.3 Tags

group **Tags**

Tags are special tokens representing a type, that allow Criterion to infer type information on parameters for better diagnostics on assertion failure.

Any tag may also use the square-bracket array subscript notation to denote an array type tag, like `i8[16]` or *`type(struct user)`*`[2]`, in which case the criterion will apply on each element of this array.

Note: A tag is a special, Criterion-specific language token. It it neither a symbol nor a macro, and cannot be used in any other context than when explicitely allowed.

Defines

i8

i16

i32

i64

u8

u16

u32

u64

sz

ptr

iptr

uptr

chr

int

uint

long

ulong

llong

ullong

flt

dbl

ldbl

cx_flt

cx_dbl

cx_ldbl

mem

str

wcs

tcs

type(UserType)

Represent an user-defined type.

The user type must be printable, and as such must implement a "to-string" operation:

```
(C only) char *cr_mem_<type>_tostr(const <type> *val);
(C++ only) std::ostream &operator<<(std::ostream &os, const <type> &
↪val);
```

Additionally, the user must implement the following functions to use various general-purpose criteria:

eq, *ne*, *le*, *ge*:

```
(C only) int cr_mem_<type>_eq(const <type> *lhs, const <type> *rhs);
(C++ only) bool operator==(const <type> &lhs, const <type> &rhs);
```

lt, *le*, *gt*, *ge*:

```
(C only) int cr_mem_<type>_lt(const <type> *lhs, const <type> *rhs);
(C++ only) bool operator<(const <type> &lhs, const <type> &rhs);
```

Due to implementation restrictions, UserType must either be a structure, an union, an enum, or a typedef.

For instance, these are fine:

```
type(foo)
type(struct foo)
```

and these are not:

```
type(foo *)
type(int (&foo)(void))
```

in these cases, use a typedef to alias those types to a single-word token.

CHAPTER

FIVE

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.

Note: Aborting the runner with any means (abort(), exit(), cr_assert(), ...) is unsupported. If you need to abort the runner, you need to iterate all subsequent tests and set their *disabled* field to 1.

5.1 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. `THEORY_FAIL`: occurs when a theory iteration fails.
7. `TEST_CRASH`: occurs when a test crashes unexpectedly.
8. `POST_TEST`: occurs after a test ends, but before the test finalization.
9. `POST_FINI`: occurs after a test finalization.

10. POST_SUITE: occurs before a suite is finalized.
11. POST_ALL: occurs after all the tests are done.

5.2 Hook Parameters

A report hook takes exactly one parameter. 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_theory_stats * for THEORY_FAIL.
- 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, this is a valid report hook declaration for the PRE_TEST phase:

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

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

CHAPTER

SIX

LOGGING MESSAGES

Sometimes, it might be useful to print some output from within a test or fixture – and while this can be done trivially with a `printf`, it doesn't integrate well with the current output, nor does it work *at all* when the process is testing a redirected stdout.

For these cases, Criterion exposes a logging facility:

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

Test(suite_name, test_name) {
    cr_log_info("This is an informational message. They are not displayed "
        "by default.");
    cr_log_warn("This is a warning. They indicate some possible malfunction "
        "or misconfiguration in the test.");
    cr_log_error("This is an error. They indicate serious problems and "
        "are usually shown before the test is aborted.");
}
```

`cr_log_info`, `cr_log_warn` and `cr_log_error` are all macros expanding to a call to the `cr_log` function. All of them take a mandatory format string, followed by optional parameters; for instance:

```
cr_log_info("%d + %d = %d", 1, 2, 3);
```

If using C++, the output stream objects `info`, `warn` and `error` are defined within the `criterion::logging` namespace, and can be used in conjunction with `operator<<`:

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

using criterion::logging::info;
using criterion::logging::warn;
using criterion::logging::error;

Test(suite_name, test_name) {
    info << "This is an informational message. "
         << "They are not displayed by default."
         << std::flush;
    warn << "This is a warning. "
         << "They indicate some possible malfunction "
         << "or misconfiguration in the test."
         << std::flush;
```

(continues on next page)

(continued from previous page)

```
    error << "This is an error. "
          << "They indicate serious problems and "
          << "are usually shown before the test is aborted."
          << std::flush;
}
```

Note that empty messages are ignored, and newlines in the log message splits the passed string into as many messages are there are lines.

CHAPTER

SEVEN

ENVIRONMENT AND CLI

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

7.1 Command line arguments

- `-h or --help`: Show a help message with the available switches.
- `-q or --quiet`: Disables all logging.
- `-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.
- `-jN or --jobs N`: Use `N` parallel jobs to run the tests. `0` picks a number of jobs ideal for your hardware configuration.
- `--filter [PATTERN]`: Run tests whose string identifier matches the given shell wildcard pattern (see dedicated section below). (*nix only)
- `--debug[=debugger]`: Run tests with a debugging server attached. `debugger` can be 'gdb', 'lldb', or 'windbg' (windows only).
- `--debug-transport [TRANSPORT]`: Make the debugging server use the specified remote transport. Only transports of the form `tcp:port` are currently supported. `tcp:1234` is the default.
- `--no-early-exit`: This flag is deprecated and no longer does anything.
- `-S or --short-filename`: The filenames are displayed in their short form.
- `--always-succeed`: The process shall exit with a status of `0`.
- `--tap[=FILE]`: Writes a TAP (Test Anything Protocol) report to FILE. No file or `"-"` means `stderr` and implies `--quiet`. This option is equivalent to `--output=tap:FILE`.
- `--xml[=FILE]`: Writes JUnit4 XML report to FILE. No file or `"-"` means `stderr` and implies `--quiet`. This option is equivalent to `--output=tap:FILE`.
- `--json[=FILE]`: Writes a JSON report to FILE. No file or `"-"` means `stderr` and implies `--quiet`. This option is equivalent to `--output=tap:FILE`.
- `--verbose[=level]`: Makes the output verbose. When provided with an integer, sets the verbosity level to that integer.

- `--full-stats`: Forces tests to fully report statistics. By default, tests do not report details for passing assertions, so this option forces them to do so. Activating this causes massive slowdowns for large number of assertions, but provides more accurate reports.
- `-OPROVIDER:FILE or --output=PROVIDER:FILE`: Write a test report to FILE using the output provider named by PROVIDER. If FILE is `"-"`, it implies `--quiet`, and the report shall be written to `stderr`.

7.2 Shell Wildcard Pattern

Extglob patterns in criterion are matched against a test's string identifier.

In the table below, a `pattern-list` is a list of patterns separated by `|`. Any extglob pattern can be constructed by combining any of the following sub-patterns:

Pattern	Meaning
*	matches everything
?	matches any character
[seq]	matches any character in *seq*
[!seq]	matches any character not in *seq*
?(pattern-list)	Matches zero or one occurrence of the given patterns
*(pattern-list)	Matches zero or more occurrences of the given patterns
+(pattern-list)	Matches one or more occurrences of the given patterns
@(pattern-list)	Matches one of the given patterns
!(pattern-list)	Matches anything except one of the given patterns

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.

7.3 Environment Variables

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

- `CRITERION_ALWAYS_SUCCEED`: Same as `--always-succeed`.
- `CRITERION_FAIL_FAST`: Same as `--fail-fast`.
- `CRITERION_USE_ASCII`: Same as `--ascii`.
- `CRITERION_JOBS`: Same as `--jobs`. Sets the number of jobs to its value.
- `CRITERION_SHORT_FILENAME`: Same as `--short-filename`.
- `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)
- `CRITERION_DISABLE_TIME_MEASUREMENTS`: Disables any time measurements on the tests.
- `CRITERION_OUTPUTS`: Can be set to a comma-separated list of `PROVIDER:FILE` entries. For instance, setting the variable to `tap:foo.tap,xml:bar.xml` has the same effect as specifying `--tap=foo.tap` and `--xml=bar.xml` at once.
- `CRITERION_ENABLE_TAP`: (Deprecated, use CRITERION_OUTPUTS) Same as `--tap`.

CHAPTER

EIGHT

WRITING TESTS REPORTS IN A CUSTOM FORMAT

Outputs providers are used to write tests reports in the format of your choice: for instance, TAP and XML reporting are implemented with output providers.

8.1 Adding a custom output provider

An output provider is a function with the following signature:

```
void func(FILE *out, struct criterion_global_stats *stats);
```

Once implemented, you then need to register it as an output provider:

```
criterion_register_output_provider("provider name", func);
```

This needs to be done before the test runner stops, so you may want to register it either in a self-provided main, or in a PRE_ALL or POST_ALL report hook.

8.2 Writing to a file with an output provider

To tell criterion to write a report to a specific file using the output provider of your choice, you can either pass `--output` as a command-line parameter:

```
./my_tests --output="provider name":/path/to/file
```

Or, you can do so directly by calling `criterion_add_output` before the runner stops:

```
criterion_add_output("provider name", "/path/to/file");
```

The path may be relative. If `"-"` is passed as a filename, the report will be written to `stderr`.

CHAPTER

NINE

USING PARAMETERIZED TESTS

Parameterized tests are useful to repeat a specific test logic over a finite set of parameters.

Due to limitations on how generated parameters are passed, parameterized tests can only accept one pointer parameter; however, this is not that much of a problem since you can just pass a structure containing the context you need.

9.1 Adding parameterized tests

Adding parameterized tests is done by defining the parameterized test function, and the parameter generator function:

group **ParameterizedBase**

Defines

ParameterizedTest(Type, Suite, Name, ...)
ParameterizedTest(Type *param, Suite, Name, [Options...]) { Function Body }.

Defines a new parameterized test.

A parameterized test only takes one parameter to pass multiple parameters, use a structure type.

Parameters

- **Type** – The type of the parameter.
- **Suite** – The name of the test suite containing this test.
- **Name** – The name of the test.
- **...** – An optional sequence of designated initializer key/value pairs as described in the *criterion_test_extra_data* structure (see criterion/types.h). Example: `.exit_code = 1`

ParameterizedTestParameters(Suite, Name)
Defines the parameter generator prototype for the associated parameterized test.

Parameters

- **Suite** – The name of the test suite containing the test.
- **Test** – The name of the test.

Returns A constructed instance of criterion::parameters, or the result of the cr_make_param_array macro.

cr_make_param_array(Type, Array, Len, Cleanup)
Constructs a parameter list used as a return value for a parameter generator.

This is only recommended for C sources. For C++, use `criterion::parameters` or `criterion_test_params`.

Parameters

- **Type** – The type of the array subscript.
- **Array** – The array of parameters.
- **Len** – The length of the array.
- **Cleanup** – The optional cleanup function for the array.

Returns The parameter list.

Typedefs

using **parameters** = std::vector<T, criterion::allocator<T>>

Represents a C++ dynamic parameter list for a parameter generator.

Param T The type of the parameter.

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

ParameterizedTestParameters(suite_name, test_name) {
    void *params;
    size_t nb_params;

    // generate parameter set
    return cr_make_param_array(Type, params, nb_params);
}

ParameterizedTest(Type *param, suite_name, test_name) {
    // contents of the test
}
```

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

`Type` is the compound type of the generated array. `params` and `nb_params` are the pointer and the length of the generated array, respectively.

Note: The parameter array must be reachable after the function returns – as such, local arrays must be declared with *static* or dynamically allocated.

9.2 Passing multiple parameters

As said earlier, parameterized tests only take one parameter, so passing multiple parameters is, in the strict sense, not possible. However, one can easily use a struct to hold the context as a workaround:

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

struct my_params {
    int param0;
    double param1;
    ...
};
```

(continues on next page)

(continued from previous page)

```
ParameterizedTestParameters(suite_name, test_name) {
    static struct my_params params[] = {
        // parameter set
    };

    size_t nb_params = sizeof (params) / sizeof (struct my_params);
    return cr_make_param_array(struct my_params, params, nb_params);
}

ParameterizedTest(struct my_params *param, suite_name, test_name) {
    // access param->param0, param->param1, ...
}
```

C++ users can also use a simpler syntax before returning an array of parameters:

```
ParameterizedTestParameters(suite_name, test_name) {
    static struct my_params params[] = {
        // parameter set
    };

    return criterion_test_params(params);
}
```

9.2.1 Dynamically allocating parameters

Any dynamic memory allocation done from a ParameterizedTestParameter function **must** be done with `cr_malloc`, `cr_calloc`, or `cr_realloc`.

Any pointer returned by those 3 functions must be passed to `cr_free` after you have no more use of it.

It is undefined behaviour to use any other allocation function (such as `malloc`) from the scope of a ParameterizedTestParameter function.

In C++, these methods should not be called explicitely – instead, you should use:

- `criterion::new_obj<Type>(params...)` to allocate an object of type `Type` and call its constructor taking `params...`. The function possess the exact same semantics as `new Type(params...)`.
- `criterion::delete_obj(obj)` to destroy an object previously allocated by `criterion::new_obj`. The function possess the exact same semantics as `delete obj`.
- `criterion::new_arr<Type>(size)` to allocate an array of objects of type `Type` and length `size`. `Type` is initialized by calling its default constructor. The function possess the exact same semantics as `new Type[size]`.
- `criterion::delete_arr(array)` to destroy an array previously allocated by `criterion::new_arr`. The function possess the exact same semantics as `delete[] array`.

Furthermore, the `criterion::allocator<T>` allocator can be used with STL containers to allocate memory with the functions above.

9.2.2 Freeing dynamically allocated parameter fields

One can pass an extra parameter to `cr_make_param_array` to specify the cleanup function that should be called on the generated parameter context:

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

struct my_params {
    int *some_int_ptr;
};

void cleanup_params(struct criterion_test_params *ctp) {
    cr_free(((struct my_params *) ctp->params)->some_int_ptr);
}

ParameterizedTestParameters(suite_name, test_name) {
    static my_params params[] = {{
        .some_int_ptr = cr_malloc(sizeof (int));
    }};
    param[0].some_int_ptr = 42;

    return cr_make_param_array(struct my_params, params, 1, cleanup_params);
}
```

C++ users can use a more convenient approach:

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

struct my_params {
    std::unique_ptr<int, decltype(criterion::free)> some_int_ptr;

    my_params(int *ptr) : some_int_ptr(ptr, criterion::free) {}
};

ParameterizedTestParameters(suite_name, test_name) {
    static criterion::parameters<my_params> params;
    params.push_back(my_params(criterion::new_obj<int>(42)));

    return params;
}
```

`criterion::parameters<T>` is typedef'd as `std::vector<T, criterion::allocator<T>>`.

9.3 Configuring parameterized tests

Parameterized tests can optionally recieve configuration parameters to alter their own behaviour, and are applied to each iteration of the parameterized test individually (this means that the initialization and finalization runs once per iteration). Those parameters are the same ones as the ones of the `Test` macro function (c.f. *Configuration reference*).

CHAPTER

TEN

USING THEORIES

Theories are a powerful tool for test-driven development, allowing you to test a specific behaviour against all permutations of a set of user-defined parameters known as "data points".

10.1 Adding theories

group **TheoryBase**

Defines

Theory(Params, Suite, Name, ...)
Defines a new theory test.

The parameters are selected from a cartesian product defined by a TheoryDataPoints macro.

Example:

```
Theory((int arg0, double arg1), suite, test) {
    // function body
};
```

Parameters
- **Params** – A list of function parameters.
- **Suite** – The name of the test suite containing this test.
- **Name** – The name of the test.
- **...** – An optional sequence of designated initializer key/value pairs as described in the *criterion_test_extra_data* structure (see criterion/types.h). Example: .exit_code = 1

TheoryDataPoints(Suite, Name)
Defines an array of data points.

The types of the specified data points *must* match the types of the associated theory.

Each entry in the array must be the result of the `DataPoints` macro.

Example:

```
TheoryDataPoints(suite, test) = {
    DataPoints(int, 1, 2, 3),              // first theory parameter
    DataPoints(double, 4.2, 0, -INFINITY),  // second theory parameter
};
```

Parameters
- **Suite** – The name of the test suite containing this test.
- **Name** – The name of the test.

DataPoints(Type, ...)
Defines a new set of data points.

Parameters
- **Type** – The type of each data point in the set.
- **...** – The data points in the set.

Adding theories is done by defining data points and a theory function:

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

TheoryDataPoints(suite_name, test_name) = {
    DataPoints(Type0, val0, val1, val2, ..., valN),
    DataPoints(Type1, val0, val1, val2, ..., valN),
    ...
    DataPoints(TypeN, val0, val1, val2, ..., valN),
}

Theory((Type0 arg0, Type1 arg1, ..., TypeN argN), suite_name, test_name) {
}
```

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

`Type0/arg0` through `TypeN/argN` are the parameter types and names of theory theory function and are available in the body of the function.

Datapoints are declared in the same number, type, and order than the parameters inside the `TheoryDataPoints` macro, with the `DataPoints` macro. Beware! It is undefined behaviour to not have a matching number and type of theory parameters and datatypes.

Each `DataPoints` must then specify the values that will be used for the theory parameter it is linked to (`val0` through `valN`).

10.2 Assertions and invariants

You can use any `cr_assert` or `cr_expect` macro functions inside the body of a theory function.

Theory invariants are enforced through the `cr_assume(Condition)` macro function: if `Condition` is false, then the current theory iteration aborts without making the test fail.

On top of those, more `assume` macro functions are available for common operations:

group **TheoryInvariants**

Defines

`cr_assume`(Condition)
Assumes `Condition` is true.

Evaluates `Condition` and continues execution if it is true. Otherwise the current theory iteration aborts without marking the test as failure.

Parameters
- **`Condition`** – **[in]** Condition to test

`cr_assume_not`(Condition)
Assumes `Condition` is false.

Evaluates `Condition` and continues execution if it is false. Otherwise the current theory iteration aborts without marking the test as failure.

Parameters
- **`Condition`** – **[in]** Condition to test

`cr_assume_eq`(Actual, Expected)
Assumes `Actual` is equal to `Expected`

Continues execution if `Actual` is equal to `Expected`. Otherwise the current theory iteration aborts without marking the test as failure.

Parameters
- **`Actual`** – **[in]** Value to test
- **`Expected`** – **[in]** Expected value

`cr_assume_neq`(Actual, Unexpected)
Assumes `Actual` is not equal to `Unexpected`

Continues execution if `Actual` is not equal to `Unexpected`. Otherwise the current theory iteration aborts without marking the test as failure.

Parameters
- **`Actual`** – **[in]** Value to test
- **`Unexpected`** – **[in]** Unexpected value

`cr_assume_gt`(Actual, Reference)
Assumes `Actual` is greater than `Reference`

Continues execution if `Actual` is greater than `Reference`. Otherwise the current theory iteration aborts without marking the test as failure.

Parameters
- **`Actual`** – **[in]** Value to test
- **`Reference`** – **[in]** Reference value

`cr_assume_geq`(Actual, Reference)
Assumes `Actual` is greater or equal to `Reference`

Continues execution if `Actual` is greater or equal to `Reference`. Otherwise the current theory iteration aborts without marking the test as failure.

Parameters
- **`Actual`** – **[in]** Value to test
- **`Reference`** – **[in]** Reference value

`cr_assume_lt`(Actual, Reference)
Assumes `Actual` is less than `Reference`

Continues execution if `Actual` is less than `Reference`. Otherwise the current theory iteration aborts without marking the test as failure.

Parameters

- **Actual** – **[in]** Value to test
- **Reference** – **[in]** Reference value

cr_assume_leq(Actual, Reference)
Assumes Actual is less or equal to Reference

Continues execution if Actual is less or equal to Reference. Otherwise the current theory iteration aborts without marking the test as failure.

Parameters
- **Actual** – **[in]** Value to test
- **Reference** – **[in]** Reference value

cr_assume_null(Value)
Assumes Value is NULL.

Continues execution if Value is NULL. Otherwise the current theory iteration aborts without marking the test as failure.

Parameters
- **Value** – **[in]** Value to test

cr_assume_not_null(Value)
Assumes Value is not NULL.

Continues execution if Value is not NULL. Otherwise the current theory iteration aborts without marking the test as failure.

Parameters
- **Value** – **[in]** Value to test

cr_assume_float_eq(Actual, Expected, Epsilon)
Assumes Actual is equal to Expected with a tolerance of Epsilon

Continues execution if Actual is equal to Expected with a tolerance of Epsilon. Otherwise the current theory iteration aborts without marking the test as failure.

Note: Use this to test equality between floats

Parameters
- **Actual** – **[in]** Value to test
- **Expected** – **[in]** Expected value
- **Epsilon** – **[in]** Tolerance between Actual and Expected

cr_assume_float_neq(Actual, Expected, Epsilon)
Assumes Actual is not equal to Expected with a tolerance of Epsilon

Continues execution if Actual is not equal to Expected with a tolerance of Epsilon. Otherwise the current theory iteration aborts without marking the test as failure.

Note: Use this to test equality between floats

Parameters
- **Actual** – **[in]** Value to test
- **Expected** – **[in]** Expected value
- **Epsilon** – **[in]** Tolerance between Actual and Expected

cr_assume_str_eq(Actual, Expected)
Assumes Actual is lexicographically equal to Expected

Continues execution if `Actual` is lexicographically equal to `Expected`. Otherwise the current theory iteration aborts without marking the test as failure.

Parameters

- **`Actual`** – **[in]** String to test
- **`Expected`** – **[in]** Expected string

`cr_assume_str_neq`(Actual, Unexpected)

Assumes `Actual` is not lexicographically equal to `Unexpected`

Continues execution if `Actual` is not lexicographically equal to `Unexpected`. Otherwise the current theory iteration aborts without marking the test as failure.

Parameters

- **`Actual`** – **[in]** String to test
- **`Unexpected`** – **[in]** Unexpected string

`cr_assume_str_lt`(Actual, Reference)

Assumes `Actual` is lexicographically less than `Reference`

Continues execution if `Actual` is lexicographically less than `Reference`. Otherwise the current theory iteration aborts without marking the test as failure.

Parameters

- **`Actual`** – **[in]** Value to test
- **`Reference`** – **[in]** Reference value

`cr_assume_str_leq`(Actual, Reference)

Assumes `Actual` is lexicographically less or equal to `Reference`

Continues execution if `Actual` is lexicographically less or equal to `Reference`. Otherwise the current theory iteration aborts without marking the test as failure.

Parameters

- **`Actual`** – **[in]** Value to test
- **`Reference`** – **[in]** Reference value

`cr_assume_str_gt`(Actual, Reference)

Assumes `Actual` is lexicographically greater than `Reference`

Continues execution if `Actual` is lexicographically greater than `Reference`. Otherwise the current theory iteration aborts without marking the test as failure.

Parameters

- **`Actual`** – **[in]** Value to test
- **`Reference`** – **[in]** Reference value

`cr_assume_str_geq`(Actual, Reference)

Assumes `Actual` is lexicographically greater or equal to `Reference`

Continues execution if `Actual` is lexicographically greater or equal to `Reference`. Otherwise the current theory iteration aborts without marking the test as failure.

Parameters

- **`Actual`** – **[in]** Value to test
- **`Reference`** – **[in]** Reference value

`cr_assume_arr_eq`(Actual, Expected, Size)

Assumes `Actual` is byte-to-byte equal to `Expected`

Continues execution if `Actual` is byte-to-byte equal to `Expected`. Otherwise the current theory iteration aborts without marking the test as failure.

> **Warning:** This should not be used on struct arrays

Parameters
- **Actual** – **[in]** Array to test
- **Expected** – **[in]** Expected array
- **Size** – **[in]** The size of both arrays

cr_assume_arr_neq(Actual, Unexpected, Size)
Assumes `Actual` is not byte-to-byte equal to `Unexpected`

Continues execution if `Actual` is not byte-to-byte equal to `Unexpected`. Otherwise the current theory iteration aborts without marking the test as failure.

> **Warning:** This should not be used on struct arrays

Parameters
- **Actual** – **[in]** Array to test
- **Unexpected** – **[in]** Unexpected array
- **Size** – **[in]** The size of both arrays

10.3 Configuring theories

Theories can optionally recieve configuration parameters to alter the behaviour of the underlying test; as such, those parameters are the same ones as the ones of the `Test` macro function (c.f. *Configuration reference*).

10.4 Full sample & purpose of theories

We will illustrate how useful theories are with a simple example using Criterion:

10.4.1 The basics of theories

Let us imagine that we want to test if the algebraic properties of integers, and specifically concerning multiplication, are respected by the C language:

```
int my_mul(int lhs, int rhs) {
    return lhs * rhs;
}
```

Now, we know that multiplication over integers is commutative, so we first test that:

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

Test(algebra, multiplication_is_commutative) {
    cr_assert_eq(my_mul(2, 3), my_mul(3, 2));
}
```

However, this test is imperfect, because there is not enough triangulation to insure that my_mul is indeed commutative. One might be tempted to add more assertions on other values, but this will never be good

enough: commutativity should work for *any* pair of integers, not just an arbitrary set, but, to be fair, you cannot just test this behaviour for every integer pair that exists.

Theories purposely bridge these two issues by introducing the concept of "data point" and by refactoring the repeating logic into a dedicated function:

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

TheoryDataPoints(algebra, multiplication_is_commutative) = {
    DataPoints(int, [...]),
    DataPoints(int, [...]),
};

Theory((int lhs, int rhs), algebra, multiplication_is_commutative) {
    cr_assert_eq(my_mul(lhs, rhs), my_mul(rhs, lhs));
}
```

As you can see, we refactored the assertion into a theory taking two unspecified integers.

We first define some data points in the same order and type the parameters have, from left to right: the first `DataPoints(int, ...)` will define the set of values passed to the `int lhs` parameter, and the second will define the one passed to `int rhs`.

Choosing the values of the data point is left to you, but we might as well use "interesting" values: `0`, `-1`, `1`, `-2`, `2`, `INT_MAX`, and `INT_MIN`:

```
#include <limits.h>

TheoryDataPoints(algebra, multiplication_is_commutative) = {
    DataPoints(int, 0, -1, 1, -2, 2, INT_MAX, INT_MIN),
    DataPoints(int, 0, -1, 1, -2, 2, INT_MAX, INT_MIN),
};
```

10.4.2 Using theory invariants

The second thing we can test on multiplication is that it is the inverse function of division. Then, given the division operation:

```
int my_div(int lhs, int rhs) {
    return lhs / rhs;
}
```

The associated theory is straight-forward:

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

TheoryDataPoints(algebra, multiplication_is_inverse_of_division) = {
    DataPoints(int, 0, -1, 1, -2, 2, INT_MAX, INT_MIN),
    DataPoints(int, 0, -1, 1, -2, 2, INT_MAX, INT_MIN),
};

Theory((int lhs, int rhs), algebra, multiplication_is_inverse_of_division) {
    cr_assert_eq(lhs, my_div(my_mul(lhs, rhs), rhs));
}
```

However, we do have a problem because you cannot have the theory function divide by 0. For this purpose, we can `assume` than `rhs` will never be 0:

```
Theory((int lhs, int rhs), algebra, multiplication_is_inverse_of_division) {
    cr_assume(rhs != 0);
    cr_assert_eq(lhs, my_div(my_mul(lhs, rhs), rhs));
}
```

`cr_assume` will abort the current theory iteration if the condition is not fulfiled.

Running the test at that point will raise a big problem with the current implementation of `my_mul` and `my_div`:

```
[----] theories.c:24: Assertion failed: (a) == (bad_div(bad_mul(a, b), b))
[----]   Theory algebra::multiplication_is_inverse_of_division failed with the␣
↪following parameters: (2147483647, 2)
[----] theories.c:24: Assertion failed: (a) == (bad_div(bad_mul(a, b), b))
[----]   Theory algebra::multiplication_is_inverse_of_division failed with the␣
↪following parameters: (-2147483648, 2)
[----] theories.c:24: Unexpected signal caught below this line!
[FAIL] algebra::multiplication_is_inverse_of_division: CRASH!
```

The theory shows that `my_div(my_mul(INT_MAX, 2), 2)` and `my_div(my_mul(INT_MIN, 2), 2)` does not respect the properties for multiplication: it happens that the behaviour of these two functions is undefined because the operation overflows.

Similarly, the test crashes at the end; debugging shows that the source of the crash is the divison of INT_MAX by -1, which is undefined.

Fixing this is as easy as changing the prototypes of `my_mul` and `my_div` to operate on `long long` rather than `int`.

10.5 What's the difference between theories and parameterized tests ?

While it may at first seem that theories and parameterized tests are the same, just because they happen to take multiple parameters does not mean that they logically behave in the same manner.

Parameterized tests are useful to test a specific logic against a fixed, *finite* set of examples that you need to work.

Theories are, well, just that: theories. They represent a test against an universal truth, regardless of the input data matching its predicates.

Implementation-wise, Criterion also marks the separation by the way that both are executed:

Each parameterized test iteration is run in its own test; this means that one parameterized test acts as a collection of many tests, and gets reported as such.

On the other hand, a theory act as one single test, since the size and contents of the generated data set is not relevant. It does not make sense to say that an universal truth is "partially true", so if one of the iteration fails, then the whole test fails.

CHAPTER

ELEVEN

CHANGING THE INTERNALS

11.1 Providing your own main

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

11.1.1 Configuring the test runner

First and foremost, you need to generate the test set; this is done by calling `criterion_initialize()`. The function returns a `struct criterion_test_set *`, that you need to pass to `criterion_run_all_tests` later on.

At the very end of your main, you also need to call `criterion_finalize` with the test set as parameter to free any ressources initialized by criterion earlier.

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
logger	struct criterion_logger *	The logger (see below)
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

if you want criterion to provide its own default CLI parameters and environment variables handling, you can also call `criterion_handle_args(int argc, char *argv[], bool handle_unknown_arg)` with the proper `argc/argv`. `handle_unknown_arg`, if set to true, is here to tell criterion to print its usage when an unknown CLI parameter is encountered. If you want to add your own parameters, you should set it to false.

The function returns 0 if the main should exit immediately, and 1 if it should continue.

11.1.2 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.

11.1.3 Example main

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

/* This is necessary on windows, as BoxFort needs the main to be exported
   in order to find it. */
#if defined (_WIN32) || defined (__CYGWIN__)
# if defined (_MSC_VER)
#  define DLLEXPORT __declspec(dllexport)
# elif defined (__GNUC__)
#  define DLLEXPORT __attribute__((dllexport))
# else
#  error No dllexport attribute
# endif
#else
# define DLLEXPORT
#endif

DLLEXPORT int main(int argc, char *argv[]) {
    struct criterion_test_set *tests = criterion_initialize();

    int result = 0;
    if (criterion_handle_args(argc, argv, true))
        result = !criterion_run_all_tests(tests);

    criterion_finalize(tests);
    return result;
}
```

11.2 Implementing your own logger

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

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 loggers in `src/log/`.

CHAPTER

TWELVE

DEBUGGING AND COVERAGE INFORMATION

Note: The following section assumes you have the relevant debugging server installed on your machine. For instance, if you're debugging with gdb, you'll need to have `gdbserver` installed and available in your PATH.

12.1 Debugging with GDB

In one terminal do:

```
$ ./test --debug=gdb
Process test created; pid = 20803
Listening on port 1234
```

Note: If no argument is passed to –debug, criterion will fall back to the appropriate debugging server for your compiler: gdbserver with gcc, lldb-server with clang, windbg with msvc.

In another terminal connect to this debug session:

```
$ gdb -q ./test
Reading symbols from ./test...done.
(gdb) target remote localhost:1234
Remote debugging using localhost:1234
0x00007ffff7dd9d90 in _start() from target:/lib64/ld-linux-x86-64.so.2
(gdb) continue
...
[Inferior 1 (process 25269) exited normally]
(gdb) q
```

After `continue` the first test is run:

```
Remote debugging from host 127.0.0.1
[RUN ] misc::failing
[----] /path/to/test.c:4: Assertion failed: The expression 0 is false.
[FAIL] misc::failing: (0,00s)

Child exited with status 0
```

And a new process is created for the next test:

```
Process /path/to/test created; pid = 26414
Listening on port 1234
```

Connect your remote debugger to this test with `target remote localhost:1234` and run the test with `continue`

To use a different port use `--debug --debug-transport=<protocol>:<port>`

12.2 Debugging with an unsupported debugger

If you want to use a debugger that criterion doesn't natively support, you may use the `idle` debugging mode: In this mode, the PID of the test will be printed, and the test itself will suspend all operations until a debugger resumes it.

```
$ ./test --debug=idle
<snip>
[----] misc::failing: Started test has PID 30587.
```

On unices, once attached, simply signal the process with SIGCONT to resume it.

```
$ sudo gdb ./test -p 30587
Attaching to process 30587
0x00007f9ea3780f3d in raise () from /usr/lib/libpthread.so.0
(gdb) signal SIGCONT
Continuing with signal SIGCONT.

Program received signal SIGCONT, Continued.
0x00007f9ea3780f5f in raise () from /usr/lib/libpthread.so.0
(gdb) c
...
(gdb) q
```

12.3 Coverage of Criterion tests

To use gcov, you have to compile your tests with the two GCC Options `-fprofile-arcs` and `-ftest-coverage`.

12.4 Using Valgrind with Criterion

Valgrind works out of the box. However, note that for all valgrind tools, you must pass `--trace-children=yes`, as criterion fork/execs test workers.

If you're using callgrind and `--callgrind-out-file`, make sure you specify `%p` in the filename, as it will get substituted by the worker PID. If you don't, all the test workers will overwrite the same file over and over, and you will only get the results for the last running test.

CHAPTER

THIRTEEN

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.

INDEX

M

N

P

S

T

U

W

Z