Criterion Documentation

Release 2.1.0

Franklin "Snaipe" Mathieu

November 25, 2015

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

The library is supported on Linux, OS X, FreeBSD, and Windows.

The following compilers are supported to compile both the library and the tests: * GCC 4.9+ * Clang 3.4+ * MSVC 14+ (Included in Visual Studio 2015 or later)

2.2 Building from source

```
$ mkdir build
$ cd build
$ cmake ..
$ cmake --build .
```

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

From the build directory created above, run with an elevated shell:

```
$ make 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 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. See *Assertion reference* for a complete list.

3.3 Configuring tests

Tests may receive optional configuration parameters to alter their behaviour or provide additional meta-data.

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.
.exit_code	int	Expect the test to exit with the specified status.

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.

Assertion reference

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

As each `assert` macros have an `expect` counterpart with the exact same number of parameters and name suffix, there is no benefit in adding `expect` macros to this list. Hence only `assert` macros are represented here.

4.1 Common Assertions

Macro	Passes if and only if	Notes
cr_assert(Condition, [Message, [Args...]])	`Condition` is true.	
cr_assert_not(Condition, [Message, [Args...]])	`Condition` is false.	
cr_assert_null(Value, [Message, [Args...]])	`Value` is `NULL`.	
cr_assert_not_null(Value, [Message, [Args...]])	`Value` is not `NULL`.	
cr_assert_eq(Actual, Expected, [Message, [Args...]])	`Actual` is equal to `Expected`.	Compatible with C++ operator overloading
cr_assert_neq(Actual, Unexpected, [Message, [Args...]])	`Actual` is not equal to `Unexpected`.	Compatible with C++ operator overloading
cr_assert_lt(Actual, Reference, [Message, [Args...]])	`Actual` is less than `Reference`.	Compatible with C++ operator overloading
cr_assert_leq(Actual, Reference, [Message, [Args...]])	`Actual` is less or equal to `Reference`.	Compatible with C++ operator overloading
cr_assert_gt(Actual, Reference, [Message, [Args...]])	`Actual` is greater than `Reference`.	Compatible with C++ operator overloading
cr_assert_geq(Actual, Reference, [Message, [Args...]])	`Actual` is greater or equal to `Reference`.	Compatible with C++ operator overloading
cr_assert_float_eq(Actual, Expected, Epsilon, [Message, [Args...]])	`Actual` is equal to `Expected` with a tolerance of `Epsilon`.	Use this to test equality between floats
cr_assert_float_neq(Actual, Unexpected, Epsilon, [Message, [Args...]])	`Actual` is not equal to `Unexpected` with a tolerance of `Epsilon`.	Use this to test inequality between floats

4.2 String Assertions

Note: these macros are meant to deal with *native* strings, i.e. char arrays. Most of them won't work on `std::string` in C++, with some exceptions – for `std::string`, you should use regular comparison assersions, as listed above.

Macro	Passes if and only if	Notes
cr_assert_str_empty(Value, [Message, [Args...]])	`Value` is an empty string.	Also works on std::string
cr_assert_str_not_empty(Value, [Message, [Args...]])	`Value` is not an empty string.	Also works on std::string
cr_assert_str_eq(Actual, Expected, [Message, [Args...]])	`Actual` is lexicographically equal to `Expected`.	
cr_assert_str_neq(Actual, Unexpected, [Message, [Args...]])	`Actual` is not lexicographically equal to `Unexpected`.	
cr_assert_str_lt(Actual, Reference, [Message, [Args...]])	`Actual` is lexicographically less than `Reference`.	
cr_assert_str_leq(Actual, Reference, [Message, [Args...]])	`Actual` is lexicographically less or equal to `Reference`.	
cr_assert_str_gt(Actual, Reference, [Message, [Args...]])	`Actual` is lexicographically greater than `Reference`.	
cr_assert_str_geq(Actual, Reference, [Message, [Args...]])	`Actual` is lexicographically greater or equal to `Reference`.	

4.3 Array Assertions

Macro	Passes if and only if	Notes
cr_assert_arr_eq(Actual, Expected, [Message, [Args...]])	`Actual` is byte-to-byte equal to `Expected`.	This should not be used on struct arrays, consider using `cr_assert_arr_eq_cmp` instead.
cr_assert_arr_neq(Actual, Unexpected, [Message, [Args...]])	`Actual` is not byte-to-byte equal to `Unexpected`.	This should not be used on struct arrays, consider using `cr_assert_arr_neq_cmp` instead.
cr_assert_arr_eq_cmp(Actual, Expected, Size, Cmp, [Message, [Args...]])	`Actual` is comparatively equal to `Expected`	Only available in C++ and GNU C99
cr_assert_arr_neq_cmp(Actual, Unexpected, Size, Cmp, [Message, [Args...]])	`Actual` is not comparatively equal to `Expected`	Only available in C++ and GNU C99
cr_assert_arr_lt_cmp(Actual, Reference, Size, Cmp, [Message, [Args...]])	`Actual` is comparatively less than `Reference`	Only available in C++ and GNU C99
cr_assert_arr_leq_cmp(Actual, Reference, Size, Cmp, [Message, [Args...]])	`Actual` is comparatively less or equal to `Reference`	Only available in C++ and GNU C99
cr_assert_arr_gt_cmp(Actual, Reference, Size, Cmp, [Message, [Args...]])	`Actual` is comparatively greater than `Reference`	Only available in C++ and GNU C99
cr_assert_arr_geq_cmp(Actual, Reference, Size, Cmp, [Message, [Args...]])	`Actual` is comparatively greater or equal to `Reference`	Only available in C++ and GNU C99

4.4 Exception Assertions

The following assertion macros are only defined for C++.

Macro	Passes if and only if	Notes
cr_assert_throw(Statement, Exception, [Message, [Args...]])	`Statement` throws an instance of `Exception`.	
cr_assert_no_throw(Statement, Exception, [Message, [Args...]])	`Statement` does not throws an instance of `Exception`.	
cr_assert_any_throw(Statement, [Message, [Args...]])	`Statement` throws any kind of exception.	
cr_assert_none_throw(Statement, [Message, [Args...]])	`Statement` does not throw any exception.	

4.5 File Assertions

Macro	Passes if and only if	Notes
cr_assert_file_contents_eq_str(File, ExpectedContents, [Message, [Args...]])	The contents of `File` are equal to the string `ExpectedContents`.	
cr_assert_file_contents_neq_str(File, ExpectedContents, [Message, [Args...]])	The contents of `File` are not equal to the string `ExpectedContents`.	
cr_assert_stdout_eq_str(ExpectedContents, [Message, [Args...]])	The contents of `stdout` are equal to the string `ExpectedContents`.	
cr_assert_stdout_neq_str(ExpectedContents, [Message, [Args...]])	The contents of `stdout` are not equal to the string `ExpectedContents`.	
cr_assert_stderr_eq_str(ExpectedContents, [Message, [Args...]])	The contents of `stderr` are equal to the string `ExpectedContents`.	
cr_assert_stderr_neq_str(ExpectedContents, [Message, [Args...]])	The contents of `stderr` are not equal to the string `ExpectedContents`.	
cr_assert_file_contents_eq(File, RefFile, [Message, [Args...]])	The contents of `File` are equal to the contents of `RefFile`.	
cr_assert_file_contents_neq(File, RefFile, [Message, [Args...]])	The contents of `File` are not equal to the contents of `RefFile`.	
cr_assert_stdout_eq(RefFile, [Message, [Args...]])	The contents of `stdout` are equal to the contents of `RefFile`.	
cr_assert_stdout_neq(RefFile, [Message, [Args...]])	The contents of `stdout` are not equal to the contents of `RefFile`.	
cr_assert_stderr_eq(RefFile, [Message, [Args...]])	The contents of `stderr` are equal to the contents of `RefFile`.	
cr_assert_stderr_neq(RefFile, [Message, [Args...]])	The contents of `stderr` are not equal to the contents of `RefFile`.	

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.

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:

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

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

CHAPTER 6

Environment and CLI

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

6.1 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`.
- `-S or --short-filename`: The filenames are displayed in their short form.
- `--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.

6.2 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.

6.3 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_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)

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.

7.1 Adding parameterized tests

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

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

ParameterizedTestParameter(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.

7.2 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;
    ...
};

ParameterizedTestParameter(suite_name, test_name) = {
    size_t nb_params = 32;
    struct my_params *params = cr_malloc(sizeof (struct my_params) * nb_params);

    // generate parameter set

    params[0] = ...
    params[1] = ...

    ...

    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, ...
}
```

7.2.1 Dynamically allocating fields

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`.

7.2.2 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);
}

ParameterizedTestParameter(suite_name, test_name) = {
    static my_params param = {
        .some_int_ptr = cr_malloc(sizeof (int));
    };
    *param.some_int_ptr = 42;

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

7.3 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 8

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".

8.1 Adding theories

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`).

8.2 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:

Macro	Description
cr_assume_not(Condition)	Assumes Condition is false.
cr_assume_null(Ptr)	Assumes Ptr is NULL.
cr_assume_not_null(Ptr)	Assumes Ptr is not NULL.
cr_assume_eq(Actual, Expected)	Assumes Actual == Expected.
cr_assume_neq(Actual, Unexpected)	Assumes Actual != Expected.
cr_assume_lt(Actual, Expected)	Assumes Actual < Expected.
cr_assume_leq(Actual, Expected)	Assumes Actual <= Expected.
cr_assume_gt(Actual, Expected)	Assumes Actual > Expected.
cr_assume_geq(Actual, Expected)	Assumes Actual >= Expected.
cr_assume_float_eq(Actual, Expected, Epsilon)	Assumes Actual == Expected with an error of Epsilon.
cr_assume_float_neq(Actual, Unexpected, Epsilon)	Assumes Actual != Expected with an error of Epsilon.
cr_assume_strings_eq(Actual, Expected)	Assumes Actual and Expected are the same string.
cr_assume_strings_neq(Actual, Unexpected)	Assumes Actual and Expected are not the same string.
cr_assume_strings_lt(Actual, Expected)	Assumes Actual is less than Expected lexicographically.
cr_assume_strings_leq(Actual, Expected)	Assumes Actual is less or equal to Expected lexicographically.
cr_assume_strings_gt(Actual, Expected)	Assumes Actual is greater than Expected lexicographically.
cr_assume_strings_geq(Actual, Expected)	Assumes Actual is greater or equal to Expected lexicographically.
cr_assume_arrays_eq(Actual, Expected, Size)	Assumes all elements of Actual (from 0 to Size - 1) are equals to those of Expected.
cr_assume_arrays_neq(Actual, Unexpected, Size)	Assumes one or more elements of Actual (from 0 to Size - 1) differs from their counterpart in Expected.

8.3 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*).

8.4 Full sample & purpose of theories

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

8.4.1 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),
};
```

8.4.2 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:

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

The associated theory is straight-forward:

```c
#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:

```c
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 paramet
[----] theories.c:24: Assertion failed: (a) == (bad_div(bad_mul(a, b), b))
[----]   Theory algebra::multiplication_is_inverse_of_division failed with the following paramet
[----] 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`.

8.5 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 9

Changing the internals

9.1 Providing your own main

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

9.1.1 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
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

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.

9.1.2 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.

9.1.3 Example main

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

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

    if (!criterion_handle_args(argc, argv, true))
        return 0;

    int result = !criterion_run_all_tests(set);

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

9.2 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 10

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.