EXPERIMENTS ON BUILDING HIGH LEVEL ABSTRACTIONS WITH C

PART 2: MODERN UNIT TESTING

A TALE OF PAIN AND WONDER

Franklin ”Snaipe” Mathieu – 2017
4/12/2015

EPITA – GCONFS

THE SORRY STATE OF C UNIT TESTING FRAMEWORKS

Unit testing is awesome!

UNIT TESTING IS AWESOME!

Java + JUnit:

```
@Test
public void testAThing() {
    Assert.assertTrue(true);
}
```

Python + unittest:

```
class TestAThing(unittest.TestCase):
  def test(self):
    self.assert(true)
```

Rust + cargo test:

```
#[test]
fn testAThing() {
    assert!(true);
}
```

WHAT ABOUT C?

Yay, what about C?

```c
void testAThing(void) {
    assert(1);
}
```

WHAT ABOUT C?

Yay, what about C?

test1.c:

```
void testAThing(void) {
    assert(1);
}
```

WHAT ABOUT C?

Yay, what about C?

test1.c:

```c
void testAThing(void) {
    assert(1);
}
```

test1.h:

```c
#ifndef TEST_1_H_
# define TEST_1_H_
void testAThing(void);
#endif // !TEST_1_H_
```

WHAT ABOUT C?

Yay, what about C?

test1.c:

```c
void testAThing(void) {
    assert(1);
}
```

test1.h:

```c
#ifndef TEST_1_H_
# define TEST_1_H_
void testAThing(void);
#endif // !TEST_1_H_
```

main.c:

```c
int main(void) {
    testAThing();
    return 0;
}
```

WHAT ABOUT C?

Yay, what about C?

test1.c:

```
void testAThing(void) { /* */ }
void testSomethingElse(void) { /* */ }
```

test1.h:

```
#ifndef TEST_1_H_
# define TEST_1_H_
void testAThing(void);
void testSomethingElse(void);
#endif // !TEST_1_H_
```

main.c:

```
int main(void) {
    testAThing();
    testSomethingElse();
    return 0;
```

Whoops.

Let's use a real framework!

LET’S USE CUNIT!

```
void test(void) { /* */ }

int main(void) {
    if (CUE_SUCCESS != CU_initialize_registry())
        return CU_get_error();

    CU_pSuite s = CU_add_suite(”suite”, NULL, NULL);
    if (NULL == s
      || (NULL == CU_add_test(s, ”test”, test)))
        goto cleanup;

    CU_basic_set_mode(CU_BRM_VERBOSE);
    CU_basic_run_tests();

cleanup:
    CU_cleanup_registry();
    return CU_get_error();
}
```

How about Check?

HOW ABOUT CHECK?

Check and plain C:

```
START_TEST (test_name)
{
  /* unit test code */
}
END_TEST
```

HOW ABOUT CHECK?

Check and plain C:

```
START_TEST (test_name)
{
  /* unit test code */
}
END_TEST
```

HOW ABOUT CHECK?

Check and plain C:

```
START_TEST (test_name)
{
  /* unit test code */
}
END_TEST
```

Check, m4, and C:

```
# suite The Suite
# tcase The Test Case

# test  the_test
  const char msg[] = ”Hello, world!\n”;

  int nc = printf(”%s”, msg);
  fail_unless(nc == (sizeof msg - 1));
```

HOW ABOUT CHECK?

Check and plain C:

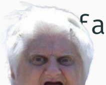
```
START_TEST (test_name)
{
  /* unit test code */
}
END_TEST
```

Check, m4, and C:

```
# suite The Suite
# tcase The Test Case

# test  the_test
  const char msg[] = ”Hello, world!\n”;

  int nc = printf(”%s”, msg);
  fail_unless(nc == (sizeof msg - 1));
```



HOW ABOUT CHECK?

Check and plain C:

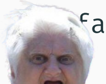
```
START_TEST (test_name)
{
  /* unit test code */
}
END_TEST
```

Check, m4, and C:

```
# suite The Suite
# tcase The Test Case

# test  the_test
  const char msg[] = ”Hello, world!\n”;

  int nc = printf(”%s”, msg);
  fail_unless(nc == (sizeof msg - 1));
```



HOW ABOUT CHECK?

Check and plain C:

```
START_TEST (test_name)
{
  /* unit test code */
}
END_TEST
```

Check, m4, and C:

```
# suite The Suite
# tcase The Test Case

# test  the_test
  const char msg[] = ”Hello, world!\n”;

  int nc = printf(”%s”, msg);
  fail_unless(nc == (sizeof msg - 1));
```

Let's step back.

WHAT DO WE WANT?

We want:

- Automatic test registration

WHAT DO WE WANT?

We want:

- Automatic test registration
- Industrial-grade quality

WHAT DO WE WANT?

We want:

- Automatic test registration
- Industrial-grade quality
- Let the user in full control

WHAT DO WE WANT?

We want:

- Automatic test registration
- Industrial-grade quality
- Let the user in full control
- Compatibility on all major platforms (Linux, Windows, Darwin, FreeBSD)

WHAT DO WE WANT?

We want:

- Automatic test registration
- Industrial-grade quality
- Let the user in full control
- Compatibility on all major platforms (Linux, Windows, Darwin, FreeBSD)
- And most importantly: KISS interface, easy and pleasant to use

IMPLEMENTING CRITERION

A SHORT LOOK AT EXECUTABLE FILE FORMATS

ELF (Linux), PE (Windows), Mach-O (Darwin):

```
/----------------------------------------------\
|                    HEADER                    |
|______________________________________________|
|                   SEGMENTS                   |
|       /--------------------------------\     |
|       |            SECTION             |     |
|       |________________________________|     |
|       |                                |     |
|       |            SECTION             |     |
|       |________________________________|     |
|       |                                |     |
|       |            SECTION             |     |
|       \________________________________/     |
|                                              |
\______________________________________________/
```

SECTIONS

- Sections may contain arbitrary data.

SECTIONS

- Sections may contain arbitrary data.
- Nonstandard sections are allowed and may hold user data.

SECTIONS

- Sections may contain arbitrary data.
- Nonstandard sections are allowed and may hold user data.
- The begining and end of a section are known at link time, and run time if the binary is not stripped.

SECTIONS

- Sections may contain arbitrary data.
- Nonstandard sections are allowed and may hold user data.
- The begining and end of a section are known at link time, and run time if the binary is not stripped.

We can use this to automatically discover our tests!

Put our test data in a section, then iterate over the section with pointer arithmetic.

DOING THINGS MANUALLY

User code:

```
struct test {
  const char *name;
  void (*fn)(void);
};

static void my_test_fn(void) { /* */ }

const static struct test my_test_data = {
        .name ”my_test”,
        .fn = my_test_fn
    };

__attribute__((section(”tests”)))
const static struct test *my_test_ptr = &my_test_data;
```

DOING THINGS MANUALLY

```
#define Test(Name)                              \
    static void Name ## _fn(void);              \
    const static struct test Name ## _data = {  \
            .name #Name,                        \
            .fn = Name ## _fn                   \
        };                                      \
    __attribute__((section(”tests”)))           \
    const static struct test *Name ## _ptr      \
        = &Name ## _data;                       \
    static void Name ## _fn(void)
```

DOING THINGS MANUALLY

```
#define Test(Name)                              \
    static void Name ## _fn(void);              \
    const static struct test Name ## _data = {  \
            .name #Name,                        \
            .fn = Name ## _fn                   \
        };                                      \
    __attribute__((section(”tests”)))           \
    const static struct test *Name ## _ptr      \
        = &Name ## _data;                       \
    static void Name ## _fn(void)

  Test(my_test) { /* */ }
```

TACKLING THE MISSING MAIN

Library code:

```
static struct test *__start_tests;
static struct test *__stop_tests;

int main(void) {
    for (struct test **t = &__start_tests;
         t < &__stop_tests;
         ++t) {

        if (*t == NULL)
            continue;

        printf(”Running test ’%s’.\n’”, (*t)->name);
        (*t)->fn();
    }
    return 0;
}
```

TACKLING THE MISSING MAIN

Where do we put the main?

TACKLING THE MISSING MAIN

Where do we put the main?

We could define a `CRITERION_DEFINE_MAIN` macro, but this is suboptimal.

Instead, let's remember that `main` is just a symbol, and should get resolved by the linker wherever it is.

TACKLING THE MISSING MAIN

Where do we put the main?

We could define a `CRITERION_DEFINE_MAIN` macro, but this is suboptimal.

Instead, let's remember that `main` is just a symbol, and should get resolved by the linker wherever it is.

We can then put our main in a separate library, and this "default" main will be picked up if the user does not define their own main.

LOOKING BACK AT WHAT WE DID

User code:

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

Test(foo) {}

Test(bar) {
    assert(0);
}
```

Usage:

```
$ cc -o test test.c -lcriterion
$ ./test
Running test ’foo’.
Running test ’bar’.
test: test.c:6: main: Assertion ‘0’ failed.
```

Wow.

ENHANCING THE LIBRARY

SANDBOXING FOR THE GREATER GOOD

A unit test should be self-contained and isolated from other parts of the system.

For the reporting to be relevant, I expect my test runner to be able to complete gracefully all the tests, even if they crash.

SANDBOXING FOR THE GREATER GOOD

A unit test should be self-contained and isolated from other parts of the system.

For the reporting to be relevant, I expect my test runner to be able to complete gracefully all the tests, even if they crash.

In other words, this shouldn't prevent the other tests from running:

```
Test(crash) {
    *((int *) NULL) = 42;
}
```

SANDBOXING FOR THE GREATER GOOD

A unit test should be self-contained and isolated from other parts of the system.

For the reporting to be relevant, I expect my test runner to be able to complete gracefully all the tests, even if they crash.

In other words, this shouldn’t prevent the other tests from running:

```c
Test(crash) {
    *((int *) NULL) = 42;
}
```

We need to sandbox each test.

But how could we recover from errors such as a segmentation fault?

SANDBOXING FOR THE GREATER GOOD

A unit test should be self-contained and isolated from other parts of the system.

For the reporting to be relevant, I expect my test runner to be able to complete gracefully all the tests, even if they crash.

In other words, this shouldn't prevent the other tests from running:

```c
Test(crash) {
    *((int *) NULL) = 42;
}
```

We need to sandbox each test.

But how could we recover from errors such as a segmentation fault?

Short answer: you can't. After a sigsegv hits, the best thing you can do is exit.

FORKING AROUND AT THE SPEED OF SOUND

Let’s isolate the test into its own forked process!

- The test runs in its own address space
- File handles/descriptors are bound to the process
- Threads are local to the process
- All of the above points are duplicated from the parent process
- For parent/child communication we use a pipe

With this, even in the event of major corruption or the end of the universe, we are guaranteed that the test runner will always be up and running until the very end.

FORKING AROUND AT THE SPEED OF SOUND

Let's isolate the test into its own forked process!

- The test runs in its own address space
- File handles/descriptors are bound to the process
- Threads are local to the process
- All of the above points are duplicated from the parent process
- For parent/child communication we use a pipe

With this, even in the event of major corruption or the end of the universe, we are guaranteed that the test runner will always be up and running until the very end.

Problem: there is no `fork()` or `pipe(int[2])` on Windows...

We would like to avoid using Cygwin.

woe32 strikes yet again

Welcome to hell.

How Cygwin implements fork() in a nutshell:

- Create a suspended process
- Copy all memory maps from the parent to the child
- Copy processor context, set the IP register
- Resume child and return

woe32 strikes yet again

Welcome to hell.

How Cygwin implements fork() in a nutshell:

- Create a suspended process
- Copy all memory maps from the parent to the child
- Copy processor context, set the IP register
- Resume child and return

However, we can’t do this, because we do not control any of the initialization normally provided by the cygwin toolchain.

It’s also horrible to debug.

woe32 strikes yet again

What I ended up doing:

- Create a suspended process
- Copy a custom heap from the parent to the child
- Copy test data into a named mapping
- Resume child, child opens the mapping and initialize itself.
- In parent, wait for the child to be initialized, then return its PID

woe32 strikes yet again

What I ended up doing:

- Create a suspended process
- Copy a custom heap from the parent to the child
- Copy test data into a named mapping
- Resume child, child opens the mapping and initialize itself.
- In parent, wait for the child to be initialized, then return its PID

This is easier to maintain and debug – however, we can’t no longer universally guarantee to have the test inheriting its parent’s address space.

WOE32 STRIKES YET AGAIN

What I ended up doing:

- Create a suspended process
- Copy a custom heap from the parent to the child
- Copy test data into a named mapping
- Resume child, child opens the mapping and initialize itself.
- In parent, wait for the child to be initialized, then return its PID

This is easier to maintain and debug – however, we can't no longer universally guarantee to have the test inheriting its parent's address space.

This is why we introduce a custom heap, and `cr_malloc`/`cr_free` to manipulate this heap on Windows, or simply call `malloc`/`free` on other systems.

ENHANCING THE API

```
Test(bar) {}

Test(foo) {
    assert(0);
}
```

ENHANCING THE API

```
Test(suite, bar) {}

Test(suite, foo) {
    assert(0);
}
```

ENHANCING THE API

```
Test(suite, crash, .signal = SIGSEGV) {
    *((int *) NULL) = 42;
}

Test(suite, foo, .description = ”Testing the foos”) {
    assert(0);
}
```

ENHANCING THE API

```
void setup(void) { /* */ }
void teardown(void) { /* */ }

Test(suite, crash,
      .signal = SIGSEGV,
      .init = setup,
      .fini = teardown) {
    *((int *) NULL) = 42;
}

Test(suite, foo,
      .description = ”Testing the foos”,
      .init = setup,
      .fini = teardown) {
    assert(0);
}
```

ENHANCING THE API

```
void setup(void) { /* */ }
void teardown(void) { /* */ }

TestSuite(suite, .init = setup, .fini = teardown);

Test(suite, crash, .signal = SIGSEGV) {
    *((int *) NULL) = 42;
}

Test(suite, foo, .description = ”Testing the foos”) {
    assert(0);
}
```

ENHANCING THE API

```
void setup(void) { /* */ }
void teardown(void) { /* */ }

TestSuite(suite, .init = setup, .fini = teardown);

Test(suite, crash, .signal = SIGSEGV) {
    *((int *) NULL) = 42;
}

Test(suite, foo, .description = ”Testing the foos”) {
    cr_assert(1);
    cr_assert(0, ”Assertion message”);
}
```

ENHANCING THE API

```
void setup(void) { /* */ }
void teardown(void) { /* */ }

TestSuite(suite, .init = setup, .fini = teardown);

Test(suite, crash, .signal = SIGSEGV) {
    *((int *) NULL) = 42;
}

Test(suite, foo, .description = ”Testing the foos”) {
    cr_assert(1);
    cr_expect_eq(1, 0); // fails but does not abort
    cr_assert(0, ”A %s format string”, ”cool”);
}
```

PROVIDING SANE DEFAULTS AND REPORTING TOOLS

CLI AND FRIENDS

Since we control the main, we can provide sane tooling that every programmer should expect of their unit test runner:

- Control the verbosity level (`--verbose[=N]`)
- Filtering the running tests with pattern matching (`--pattern PATTERN`)
- List the tests (`--list`)
- Fail fast: if one test fails, don't run the others (`--fail-fast`)
- And more...

REPORTING FORMATS

In addition to the normal CLI report, we can add multiple test report formats:

- TAP – Test Anything Protocol (Compatible with a lot of test drivers)
- JUnit XML (Compatible with CI services like Jenkins)
- Json (For custom webservices)

RAISING THE BAR ON THE TESTING TOOLS

MULTIPLE LANGUAGE COMPATIBILITY

Tests in C:

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

Test(sample, simple) {
    cr_assert(0, ”Hello, World.”);
}
```

MULTIPLE LANGUAGE COMPATIBILITY

Tests in C++:

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

Test(sample, simple) {
    cr_assert(0, ”Hello, World.”);
}
```

MULTIPLE LANGUAGE COMPATIBILITY

Tests in Objective-C:

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

Test(sample, simple) {
    cr_assert(0, ”Hello, World.”);
}
```

MULTIPLE LANGUAGE COMPATIBILITY

Tests in *:

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

Test(sample, simple) {
    cr_assert(0, ”Hello, World.”);
}
```

The interface is unified, but there are extensions for C++ (assertions on exceptions, catching unhandled exceptions, …)

PARAMETERIZED TESTS

Parameterized tests are tests that take a parameter from a finite dataset.

```
ParameterizedTestParameters(suite, test) {
    static int arr[] = { 1, 2, 3 };
    size_t len = sizeof (arr) / sizeof (int);
    return cr_make_param_array(int, arr, len);
}

ParameterizedTest(int *param, suite, test) {
    cr_assert_lt(*param, 4);
}
```

PARAMETERIZED TESTS

Parameterized tests are tests that take a parameter from a finite dataset.

```
ParameterizedTestParameters(suite, test) {
    static int arr[] = { 1, 2, 3 };
    size_t len = sizeof (arr) / sizeof (int);
    return cr_make_param_array(int, arr, len);
}

ParameterizedTest(int *param, suite, test) {
    cr_assert_lt(*param, 4);
}
```

Useful for testing and validating the **same logic** over a **finite set of data**.

THEORIES

Theories are another kind of test, designed for validating a result from a set of axioms.

THEORIES

Theories are another kind of test, designed for validating a result from a set of axioms.

Useful for testing properties that makes sense for **any kind of data**.

THEORIES

Theories are another kind of test, designed for validating a result from a set of axioms.

Useful for testing properties that makes sense for **any kind of data**.

Vanilla unit testing:

```
Test(algebra, mul) {
  cr_assert_eq(div(mul(2, 3), 3), 2);
}
```

But this test does not really make sense!

Should test for any kind of values, not just 2 and 3.

THEORIES

With a theory:

```
Theory((int lhs, int rhs), algebra, mul) {
    cr_assume_neq(rhs, 0);
    cr_assert_eq(div(mul(lhs, rhs) rhs), lhs);
}
```

THEORIES

With a theory:

```
Theory((int lhs, int rhs), algebra, mul) {
    cr_assume_neq(rhs, 0);
    cr_assert_eq(div(mul(lhs, rhs) rhs), lhs);
}
```

When writing the test, we don’t really care about `lhs` or `rhs`

THEORIES

With a theory:

```
Theory((int lhs, int rhs), algebra, mul) {
    cr_assume_neq(rhs, 0);
    cr_assert_eq(div(mul(lhs, rhs) rhs), lhs);
}
```

When writing the test, we don't really care about `lhs` or `rhs`

However, we **assume** that `rhs` is nonzero.

THEORIES

With a theory:

```
Theory((int lhs, int rhs), algebra, mul) {
    cr_assume_neq(rhs, 0);
    cr_assert_eq(div(mul(lhs, rhs) rhs), lhs);
}
```

When writing the test, we don’t really care about `lhs` or `rhs`

However, we **assume** that `rhs` is nonzero.

We pass interesting values to `lhs` and `rhs`.

```
TheoryDatapoints(algebra, mul) = {
    DataPoints(int, 0, -1, 1, INT_MAX, INT_MIN),
    DataPoints(int, 0, -1, 1, INT_MAX, INT_MIN),
};
```

What's next?

WHAT'S NEXT?

- Better float assertions (strategy choice between ULPs, Absolute epsilons, ...)
- Concolic testing
- Testing for embedded programming

(Logo designed by @pbouigue on twitter)

Any sensible contribution is welcome.

`https://snai.pe/git/criterion`

✉ `franklinmathieu+criterion@gmail.com`

`Snaipe` on `#criterion` or `#criterion-dev` @ `irc.freenode.net`

Questions?