CSPEC

Table of Contents

1. QuickStart . . . 1
2. Fucntions . . . 2
   2.1. Contexts . . . 2
      2.1.1. Context . . . 2
      2.1.2. Describe . . . 3
   2.2. Tests and Assertions . . . 3
      2.2.1. Tests (it) . . . 3
      2.2.2. Assertions (should) . . . 3
   2.3. Hooks - before y after . . . 4
      2.3.1. before . . . 4
      2.3.2. after . . . 5
3. License . . . 5
   Thanks! . . . 5

> Small behavior driven development (BDD) framework for C.

1. QuickStart

This is just an small example of how to use CSPEC

```
#include <stdio.h>
#include <stdbool.h>
#include <cspecs/cspec.h>

context (example) {

    describe("Hello world") {

        it("true should be true") {
            should_bool(true) be equal to(true);
        } end

        it("true shouldn't be false") {
            should_bool(true) not be equal to(false);
        } end

        it("this test will fail because 10 is not equal to 11") {
            should_int(10) be equal to(11);
        } end

        skip("this test will fail because \"Hello\" is not \"Bye\"") {
            should_string("Hello") be equal to("Bye");
        } end

    } end

}
```

2. Fucntions

2.1. Contexts

2.1.1. Context

Each behaviour to test must be declared within a `context`. The syntax to define a `context` is shown below:

```
context(<identifier>) {
    /* You're inside the context */
}
```

Inside a `context`, you can write functions and call them in your tests, you can also include files (.h), define macros and write scenarios using `describe`.

You should always have at the top of your test file, as the first level of nesting, either a `context` or a `describe`. The former is prefered.

2.1.2. Describe

Each scenario is written inside a `describe`, declared in this way:

```
describe("Brief description of the scenario") {
    /* Here goes the code */
} end
```

We can have multiple scenarios on each context. Again, inside a `describe` you can write functions and call them in your tests, include files (.h), define macros and write the tests using `it`.

2.2. Tests and Assertions

2.2.1. Tests (it)

Each `it` represents a test.

```
it("Brief description of the test") {
    /* Here goes the test code, along with the assertions */
} end
```

Inside it, you have to write the assertions about the behaviour you want to test. In order to do that cspec has a set of basic operations to do that, the `should` statements.

2.2.2. Assertions (should)

Each `should` is an assertion, that expects 2 values. The first is the actual value and the second, the expected one.

Currently we have a maximum threshold for assertions on each tests, defined in MAX_SHOULDS_PER_IT. Current threshold: 64.

```
should_bool(<actual_boolean>) be equal to(<expected_boolean>);
should_bool(<actual_boolean>) not be equal to(<unexpected_boolean>);

should_char(<caracter_actual>) be equal to(<caracter_esperado>);
should_char(<caracter_actual>) not be equal to(<caracter_no_esperado>);

should_short(<actual_number>) be equal to(<expected_number>);
should_short(<actual_number>) not be equal to(<unexpected_number>);

should_int(<actual_number>) be equal to(<expected_number>);
should_int(<actual_number>) not be equal to(<unexpected_number>);

should_long(<actual_number>) be equal to(<expected_number>);
should_long(<actual_number>) not be equal to(<unexpected_number>);

should_float(<actual_float>) be equal to(<expected_float>);
should_float(<actual_float>) not be equal to(<unexpected_float>);

should_double(<decimal_actual>) be equal to(<decimal_esperado>);
should_double(<decimal_actual>) not be equal to(<decimal_no_esperado>);

should_ptr(<actual_pointer>) be equal to(<expected_pointer>);
should_ptr(<actual_pointer>) not be equal to(<unexpected_pointer>);

should_string(<actual_word>) be equal to(<expected_word>);
should_string(<actual_word>) not be equal to(<unexpected_word>);
```

Also, cspec offers syntactic sugar for some of the assertions, like the following examples:

```
should_bool(<actual_boolean>) be truthy;
should_bool(<actual_boolean>) not be truthy;

should_bool(<actual_boolean>) be falsey;
should_bool(<actual_boolean>) not be falsey;

should_ptr(<actual_pointer>) be null;
should_ptr(<actual_pointer>) not be null;
```

2.3. Hooks - before y after

Sometimes the scenarios, initial configurations, or deallocation of the variables get repeated between tests. In order to handle that, inside each `describe`, you can add a block code to execute `before` and `after` each test (`it`).

2.3.1. before

```
before {
    /* Code to execute before each test */
} end
```

2.3.2. after

```
after {
    /* Code to execute after each test */
} end
```

As stated before, the context and the describe are executed secuentially, that's why it's *very important* to remember that the `before` and `after` must be declared in the beggining of the `describe` scenario, even before the first test.

Currently we have a maximum threshold for hooks on each test file, defined in MAX_CHAINS_HOOKS. Current threshold: 64.

3. License

This framework uses the GPLv3 as license. Fork it and contribute with the project!

Thanks!