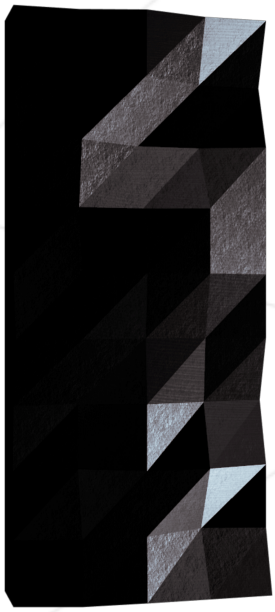


Rush - libunit

What the fork??

Coton coton@42.fr
Gaetan gaetan@42.us.org

Summary:
I will not put any untested code into production anymore.
I will not put any untested code into production anymore.
I will not put any untested code into production anymore.
I will not put any untested code into production anymore.
I will not put any untested code into production anymore.
I will not put any untested code into production anymore.
I will not put any untested code into production anymore.
I will not put any untested code into production anymore.
I will not put any untested code into production anymore.
...

Contents

Chapter I

Foreword

Howard Phillips Lovecraft, born August 20, 1890 in Providence, Rhode Island, United States and died on March 15, 1937 in the same city, is an American writer known for his horror stories, fantasy and science fiction.

His sources of inspiration, like his creations, are related to cosmic horror, to the idea that man can not understand life and that the universe is deeply foreign. Those who truly reason, like its protagonists, always jeopardize their mental health.

Lovecraft is often read for the myth he created, the myth of Cthulhu, to use August Derleth's expression: all the myths of Lovecraft's universe constituted for the author, a sort of "black pantheon", a "synthetic mythology" or a "synthetic mythology". a "synthetic folklore cycle". He essentially wanted to show that the cosmos is only not anthropocentric, that man, an insignificant form of life among others, is far away to hold a privileged place in the infinite hierarchy of life forms. Its work is deeply pessimistic and cynical and question the Enlightenment, the romanticism and Christian humanism. Lovecraft heroes generally experience feelings that are the opposite of gnosis and mysticism when, involuntarily, they get a glimpse of the horror of reality.

Although Lovecraft's readership was limited during his lifetime, his reputation evolved over time. and he is now considered one of the world's most respected horror writers. with Edgar Allan Poe, he "has a considerable influence on the next generations of horror writers.

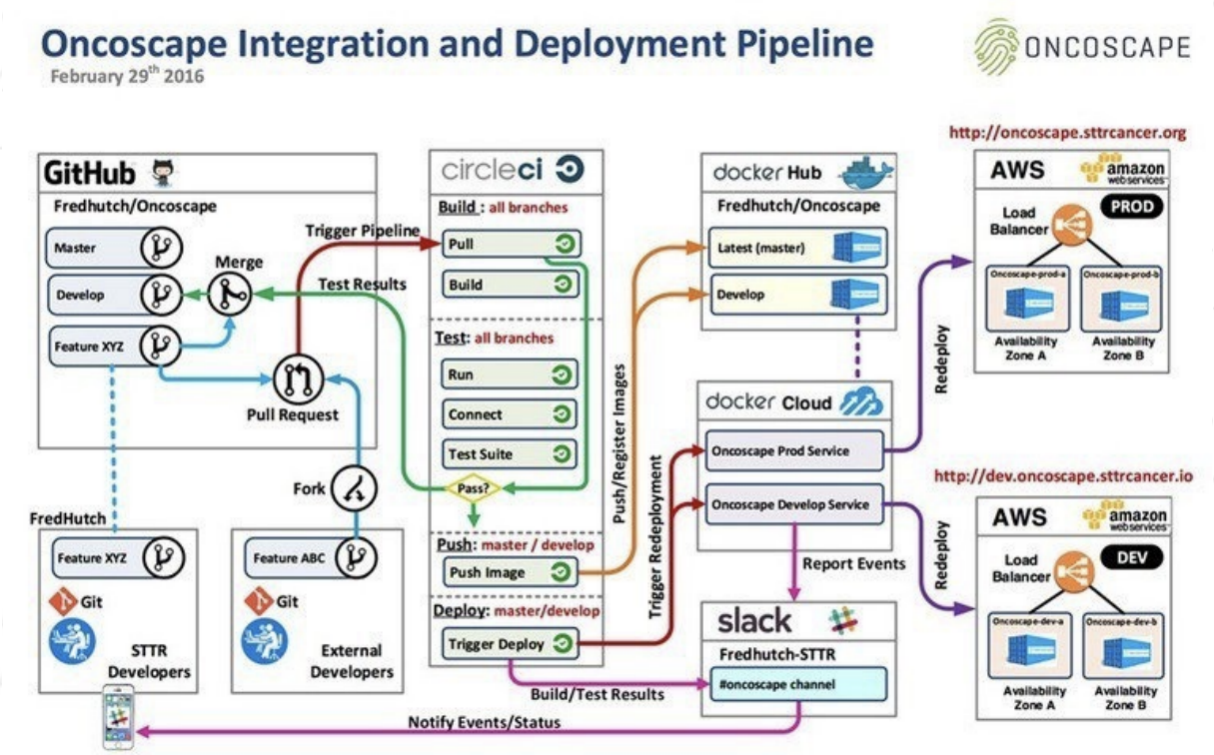
Stephen King called him "the greatest craftsman of the classic horror tale. twentieth century.

```
Ph'nglui mglw'nafh Cthulhu R'lyeh wgah'nagl fhtagn...
```

Chapter II

Introduction

Have you ever really wondered how a feature is put into production in a company?



This is the Oncoscape deployment pipeline. Beautiful, huh? What you can do It's just that there are a lot of arrows, so far nothing very witchcraft. However, what you don't see is that the code passes a series of tests before it goes out in production, like Moulinette.

The importance of this Moulinette is crucial in business, because it decides if the feature goes into production or not. Good news! This weekend, you're going to learn how to make your own moulinette! Yes you read it right: YOUR OWN MOULINETTE!

Chapter III

Objectives

As part of this rush, you will design a testing micro-framework in C, to torture in every way your functions in your C projects, with some subtleties as well. This Micro-framework will be created as a static C library that you will include in your future test functions.

The intrinsic objective of this rush is to give you a fun and useful way to organize your unit tests as much for your projects at 42 but also later for your internships and other work experience. Because the difference between a good developer and a excellent developer lies in the impartiality of his test functions.

This rush will also give you some notions to start the UNIX branch, if this is not already the case. At worst, you'll leave again in the wonderful world of processes. The e-learning video on ft_minishell will be a great help to you for this Rush.
You will also see that there are many unit testing solutions in the programming language frameworks or in development frameworks.

Ruby : Cucumber, Minitest
PHP : PHP-Unit
Javascript : Mocha, Supertest
C++ : CppUnit
C : Cgreen ...

But you might as well learn how to make your own micro-framework in C first!

This topic will propose you a simple and minimalist specification to design your Micro-framework but don't hesitate to look at the bonuses even after the rush. These are interesting ways to expand and consolidate your framework (and why don't you brag a little bit on GitHub).

Chapter IV

General instructions

This project will be corrected by humans. You are therefore free to organize and appoint your files as you wish, while nevertheless respecting the constraints listed here:

- The compiled library should be called `libunit.a`
- You have to submit a Makefile. This Makefile must compile the project, and must contain the usual rules. It should recompile the program only when necessary.
- The test function using your framework should not exit in an untimely manner. (Segmentation Fault, double free...). On the other hand, your function will manage the case of untimely exit of your unit tests without lamentably crashing. (See Mandatory Part)
- Your project must follow the Norme.
- You must submit, at the root of your repository, an author file containing your logins followed by a newline:

```
$>cat -e author
xlogin$
ylogin$
$>
```

- As part of your mandatory part, you have the right to use the following functions: `malloc`, `free exit`, `fork`, `wait`, `write`, the Macros in `<sys/wait.h>` and `<signal.h>`

Chapter V

Mandatory part

V.1 The Micro-framework

First, you will design a static library which will be the core of your Micro-Framework and that you will name `libunit`. This library will include at least one function to start a set of subfunctions without interruption and will have to analyze the return value or the interrupt signal (SegFault, Bus Error...), and a function that will load this set of sub-functions into the project.

Your Micro-framework must meet the following specifications:

- Your source files should be in a folder named `framework`
- It must be able to run a series of tests one after the other, without interruption (other than the normal end of your testing of course).
- Each test is loaded into a list/table/tree/whatever... with a specific name which will later have to be written on the standard output.
- Each test is performed in a separate process. This process ends at the end of the test and gives back the hand to the parent process.

```
man fork
```

- The parent process must be able to retrieve the test result or the type of interruption (at least SegFault and BusError)

```
man exit; man wait; man signal
```

- At the end of the test run, your program should display the function name tested, the name of each test with the result on the standard output with a nomenclature of this type :
  - **OK** : Test passed.
  - **KO** : Test failed.
  - **SEGV** : Fault segmentation detected.
  - **BUSE** : Error bus detected.
- A description of the successful tests on the total number of tests performed should be displayed at the end.
- If all tests pass, you will have to end your function by returning the value 0. If AT LEAST one test is in error (-1 or signal), the function will return -1.
- Only the result of each test is displayed on the standard output. See next part for more information.
- The arrangement of the elements on the standard output is free as long as it remains coherent.

V.2 The tests

To confirm the full power of your Micro-framework, you must be able to validate with a function with multiple tests. (Yes, repetition is desired). Your tests must follow these specifications:

- Each test function must be in the folder tests/<test_function>
- Each test is encapsulated in a function that MUST follow this prototype (return values included) :

```
int an_awesome_dummy_test_function( void )
{
    if (/* this test is successful */ )
        return (0);
    else /* this dumb test fails */
        return (-1);
}
```

- Tests must not use functions that write to the standard output.
- The test file tree should follow this mini-norme:
  - For each function tested, its tests are grouped in the same file, with a specific source file called `Launcher`.

- The `Launcher` is used to check and then run all the tests of a given function. You must design the Launcher so that you can jump one or more specific tests (either with an argument or by modifying source code easily, be creative).
- Only one test function must be present per file.
- Each test file starts with a number followed by an underscore, characterizing its order in the `Launcher` (example: 04_basic_test_four_a.c)
- The file starting with texttt00 is always considered as the textttLauncher.
- The test main must be placed at the root of this tree, and must call all the `Launchers`. You must design the test main so that that you can skip one or more specific Launchers.
- The Makefile associated with the program must contain an additional dependency called `test` which will compile your program with the test files and which will launch the executable created.
- The Norme on the number of lines per function as well as on the naming of the file does not apply to your test sources (main, launchers and tests).

V.3 Output example

Basic example of a test tree:

```
$> ls -R tests
main.c strlen
tests/strlen:
00_launcher.c 01_basic_test.c 02_null_test.c 03_bigger_str_test.c
$>
```

Test example:

```
$> cat strlen/01_basic_test.c
#include "libft.h"
#include <string.h>
int basic_test(void)
{
if (ft_strlen("Hello") == strlen("Hello"))
return(0);
else
return(-1);
}
$>
```

Example of launcher:

```
$> cat strlen/00_launcher.c
#include "101_basic_tests.h"
#include "libunit.h"
int strlen_launcher(void)
{
t_unit_test *testlist;
puts("STRLEN:");
load_test(&testlist, "Basic test", &basic_test);
load_test(&testlist, "NULL test", &null_test);
//load_test(&testlist, "Bigger string test", &bigger_str_test); /* This test won't be loaded */
return(launch_tests(&testlist));
}
$>
```

Example of a test function output:

```
$> make fclean & make test
[...]
********************************
** 42 - Unit Tests ****
********************************
STRLEN:
    > Basic test : [OK]
    > NULL test : [SEGV]

1/2 tests checked
$>
```

V.4 Your results

To validate your rush, you will have to submit:

- Test functions in the test folder with:
  - A TRUE test that returns OK
  - A TRUE test that returns KO
  - A TRUE test that returns Fault Segmentation
  - A TRUE test that returns Bus Error
- A function of at least 15 tests of your choice on an existing subject (Libft?) in the real-tests folder, with the project to test in addition. The choice of tests is decisive for grading.

You will need to create a Makefile for the framework to compile a static library Embedding your test engine. Similarly, you will need to create or modify a Makefile in tests and real_tests, such as so that the make test command run from one of the two folders executes the function in its entirety. Don't forget that your test sets must comply with the mini-standard of the test functions, in addition to Norme. Other than that, have fun and go go go!
For once you're being asked to give back the wrong code.

Chapter VI

Bonus part

Once your Micro-framework is complete, you are free to add the elements of your choice to make it even cooler.

You can:

- Add color code on test results.
- Added a timeout feature that kills the test process once a some time has passed (Beware of zombie processes)
- Add more signals to capture... as long as it is consistent. Don't go wrestling SIGUSRx signals if they are of no use to you in your test routines.
- Create a reporting file containing the test report, with the information that you deem necessary.

A HUGE PLUS is granted to the implementation of a Continuous Integration solution. The support is free, as long as it doesn't cost you anything. Besides, if you're planning to to implement this bonus, you can put your test functions as well as your framework on GitHub. You can use the following solutions:

- Jenkins
- Travis
- Circle CI ...

You will quickly see that the majority of existing CI solutions are not free. 42 and the editor of this PDF disclaim all responsibility for any reimbursement for the use of these services.

Chapter VII

Turn-in and peer-evaluation

Turn your work in using your `GiT` repository, as usual. Only work present on your repository will be graded in defense.

If you have opted for the bonus "Continuous Integration", the implementation of your solution will be peer-corrected. You will then have to explain your implementation to your corrector, and show him the joys of this magnificent thing that is the CI.