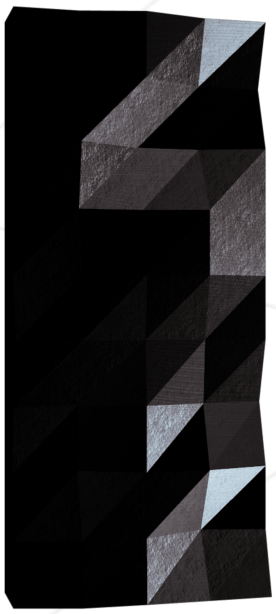


Rush - libunit

What the fork ??

42 Staff pedago@staff.42.fr

Summary:
I will NEVER deploy untested code
I will NEVER deploy untested code
I will NEVER deploy untested code
I will NEVER deploy untested code
I will NEVER deploy untested code
I will NEVER deploy untested code
I will NEVER deploy untested code
I will NEVER deploy untested code
I will NEVER deploy untested code
I will NEVER deploy untested code
...

Contents

Chapter I

Foreword

Howard Phillips Lovecraft (August 20, 1890 – March 15, 1937) was an American author of horror, fantasy, and science fiction, especially the subgenre known as weird fiction.

Lovecraft's guiding literary principle was what he termed "cosmicism" or "cosmic horror", the idea that life is incomprehensible to human minds and that the universe is fundamentally alien. Those who genuinely reason, like his protagonists, gamble with sanity. As early as the 1940s, Lovecraft had developed a cult following for his Cthulhu Mythos, a series of loosely interconnected fiction featuring a pantheon of humanity-nullifying entities, as well as the Necronomicon, a fictional grimoire of magical rites and forbidden lore. His works were deeply pessimistic and cynical, challenging the values of the Enlightenment, Romanticism, and Christian humanism. Lovecraft's protagonists usually achieve the mirror-opposite of traditional gnosis and mysticism by momentarily glimpsing the horror of ultimate reality and the abyss.

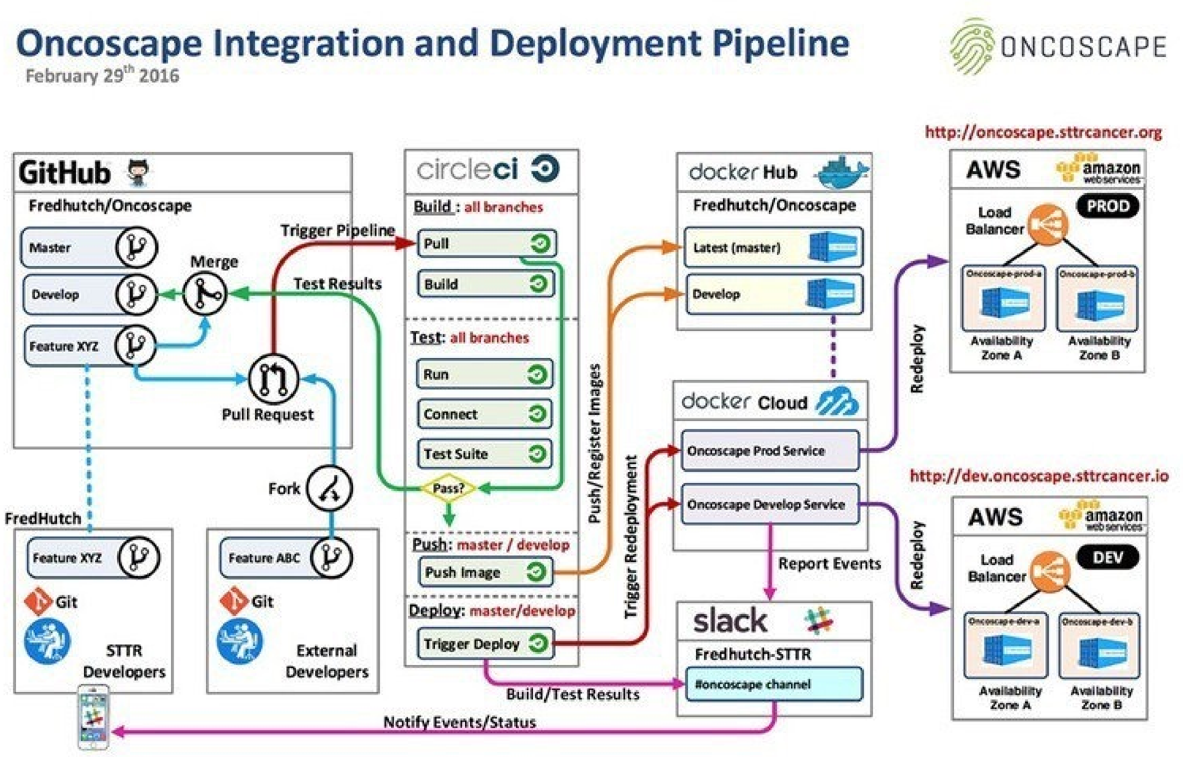
Although Lovecraft's readership was limited during his life, his reputation has grown over the decades, and he is now regarded as one of the most influential horror writers of the 20th century. According to Joyce Carol Oates, Lovecraft — as with Edgar Allan Poe in the 19th century — has exerted "an incalculable influence on succeeding generations of writers of horror fiction". Stephen King called Lovecraft "the twentieth century's greatest practitioner of the classic horror tale."

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

Chapter II

Introduction

Have you ever wondered how a feature is deployed in a company?



This is the deployment pipeline for `Oncoscape`. Wonderful isn't it? You can see a lot of arrows, so far nothing surprizing. However, what you do not see is that the code passes a series of tests before being deployed, a `moulinette` in fact.
The importance of this moulinette (grinder) is crucial in companies, as it decides whether the feature goes into production or not.
Good news! This weekend, you will learn how to make your own moulinette! Yes you read correctly: **YOUR OWN MOULINETTE**!

Chapter III

Objectives

During this rush, you will design a `Micro-framework` in C language dedicated to testing, in order to challenge in every possible way the functions of your projects in C, *with some additional subtleties.*
This `Micro-framework` will be created as a C static library that you will include in your future test routines.

The intrinsic objective of this rush is to give you a fun and useful way to organize your unit tests for your projects here at `42` but also later for your internships and other professional experiences. Because the difference between a good developer and an excellent developer lies in the impartiality of his/her test routines.

You will also see that there are many unit testing solutions using the most recent programming languages or development frameworks:

Ruby : RSpec

PHP : PHP-Unit

Javascript : Mocha, Supertest

C++ : CppUnit

etc...

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

`This rush will also give you some useful notions before you start the UNIX branch, if it's not already the case. At least, you will return to the wonderful world of processes. The e-learning video for ft_minishell will be very helpful for this Rush. It is available on the intra.`

This small project will give you simple and minimalist specifications to design your `Micro-framework` but do not hesitate to look at the bonuses even after the rush. They are interesting tracks in order to beef up and consolidate your framework *(and why not shine a little bit on GitHub).*

Chapter IV

General Instructions

- This project will only be corrected by actual human beings. You are therefore free to organize and name your files as you wish, although you need to respect some requirements listed below.

- The compiled library must be named `libunit.a`

- You must submit a `Makefile`. That `Makefile` needs to compile the project and must contain the usual rules. It can only recompile the program if necessary.

- In no way can the test routine using your framework quit in an unexpected manner (Segmentation fault, double free, etc). On the other hand, your routine must handle unexpected interruption of your unit tests in a sensitive manner (see Mandatory Part).

- Your project must be written in `C` in accordance with the Norme.

- You'll have to submit at the root of your folder, a file called `author` containing your usernames followed by a '`\n`' :

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

- Within your mandatory part you are allowed to use the following functions:
  - `malloc` et `free`
  - `exit`, `fork` and `wait`
  - `write`
  - `macros (#define) from the library <sys/wait.h>`
  - `macros (#define) from the library <signal.h>`

Chapter V

Mandatory Part

V.1 The Micro-framework

Your `Micro-framework` must meet the specifics below:

- Your source files must be placed in a folder `framework`
- The `framework` must be able to execute a serie of tests one after the other without interruption.
- Each test should be stored in a list/array/tree/whatever... with a specific name which should be written on the standart output.
- Each test is executed in a separate process. This process will be closed at the end of the test and it will give the hand back to the parent process.

```
man fork
```

- The parent process must be able to catch the result of the child process or the type of interruption if it crashes (at least SegFault and BusError)

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

- At the end of the tests execution, your program should write the name of the tested functions and the name of each test with the corresponding result on the standard output according to the following syntax:

  OK : Test succeeded.

  KO : Test failed.

  SEGV : Segmentation Fault detected.

  BUSE : Bus Error detected.

- At the end, the total number of tests and the count of succeeded tests must be displayed.

- In case of complete success, the routine exits returning 0. If `at least` one of the tests failed the routine returns -1.

- Only the result of each test should be written on the standard output. See the part below for more informations.

- You are free to choose the output format as long as everything remains consistent.

V.2 The tests

To confirm the great power of your `Micro-framework`, you must be able to run your test on... a routine of tests (Yes, testing some tests, you get it !). More specifications:

- Each routine must be placed in a folder `tests/<function_to_test`
- Each test is encapsulated in a function which `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);
}
```

- The tests are not meant for functions writing on the standard output
- The file tree for your test files must follow this mini-norme:
    - For each function, the corresponding tests are grouped in the same folder, with a specific source file called `Launcher`.
    - This `Launcher` is used to load and run all the test to the choosen function. You must design it to be able to choose to silent one or few test choosen. (with a flag or by modifying a line in the source code, your call)
    - You must write only one function per file
    - Each name of test file must begin by a number followed by an underscore which define the run order(example: 04_basic_test_four_a.c)
    - The file with a name starting by 00_xxx will always be considered as the Launcher.
    - The main containing the tests must be located in the root folder. it must call all the `Launchers`. You must design it in order to be able to choose to skip one or several Launchers. (with a flag or by modifying a line in the source code, your call)
    - The Makefile associated to the program must contain an additional dependency called `test` which will compile your program with the test files and then run the binary file.
    - The restricted number of lines in a function (set by the Norme) does not apply to the Launchers and the main containing your tests.

V.3 Output example

Basic example of file 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
$>
```

Launcher example:

```
$> 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));
}
$>
```

Output example of a test routine:

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

1/2 tests checked
$>
```

V.4 Submission

To succeed your rush, you must turn in:

- The source code of your `Micro-framework` in the folder `framework`.
- A routine of tests in the folder `tests` including:
  - A **REAL** test which returns OK
  - A **REAL** test which returns KO
  - A **REAL** test which returns Segmentation Fault
  - A **REAL** test which returns Bus Error
- A routine with at least 15 tests of your choice on a project you did before (example: Libft) in the folder `real-tests`. The tests you choose will be decisive when grading your work.

Each routine must have its own Makefile including the `test` dependency. Don't forget that your files must respect the mini-norme listed above and of course the Norme.

Chapter VI

Bonus part

Once your `Micro-framework` is fully fonctionnal, you have the possibility to add some new features to make it even more swag.

You can:

- Add a color code for the results of the tests.
- Add support for functions writing on the standard output. (Be careful : the tested function must still not write on the standard output)
- Add a timeout functionnality which kills the test process after x time (Watchout for zombie processes)
- Catch more signals... as long as it remains consistent. For example, don't catch SIGUSRx signals if it's not useful in your test routine.
- Create a log file reporting useful information about the tests.
- Etc...

A **BIG PLUS** will be awarded for the implementation of a solution with Continuous integration. You can choose any support you like, as long as it's free. By the way, if you achieve this bonus, you can add your test routine and your framework on Github. You can use the following solutions:

- Jenkins
- Travis
- Circle CI
- ...

```
You will easily see that most existing CI solutions have a cost.
This is particularly true for some use case.  42 and the creator of
this PDF decline all responsibility and won't reimburse any payment
for these services.
```

Chapter VII

Submission and peer correction

Submit your work on your `GiT` repository as usual. Only the work on your repository will be graded.

If you have opted for the "Continuous Integration" bonus, the implementation of your solution will be tested by your peers. You will then have to explain your implementation to the corrector, and show him/her the wonderfulness of this magnificent thing that is `CI`.