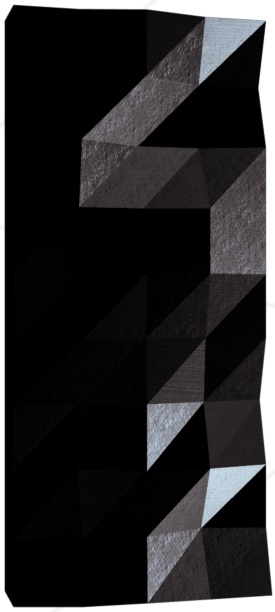


Rush - libunit

What the fork ??

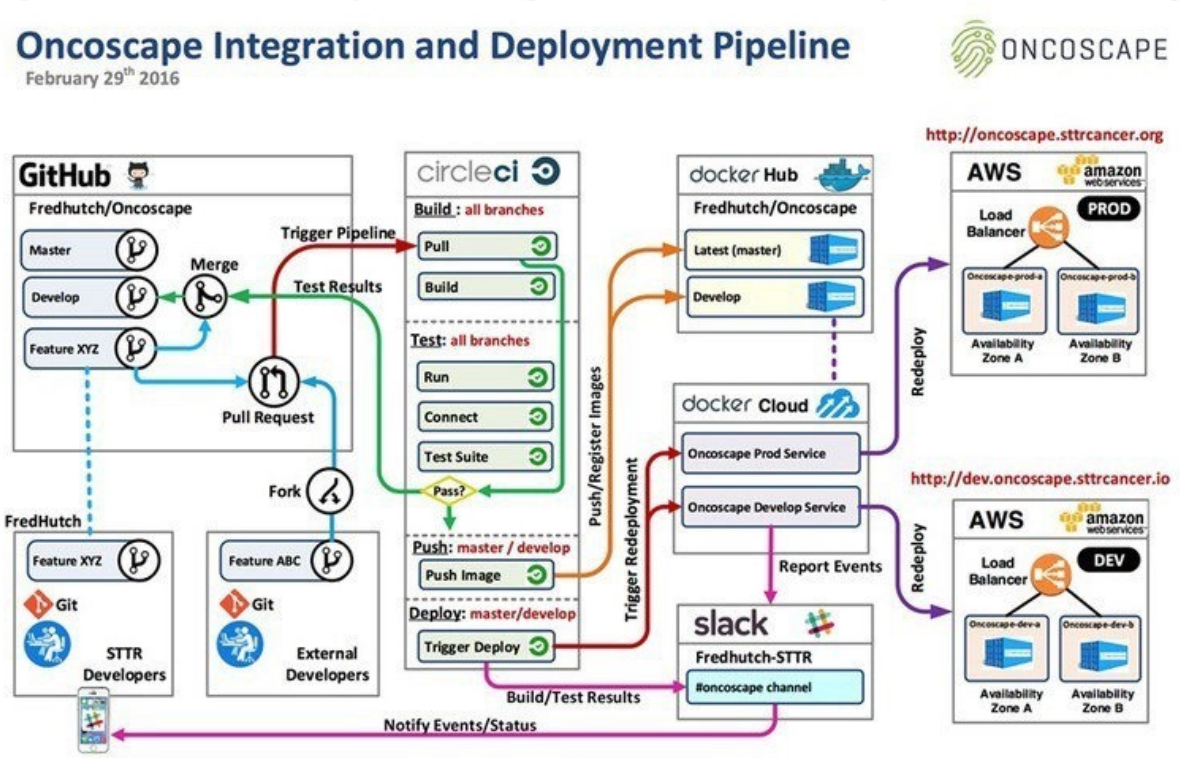
Summary:
I will NEVER deploy untested code
I will NEVER deploy untested code
I will NEVER deploy untested code
I will NEVER deploy untested code
I will NEVER deploy untested code
I will NEVER deploy untested code
I will NEVER deploy untested code
I will NEVER deploy untested code
I will NEVER deploy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

Contents

Chapter I

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 surprising. 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 II

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 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.`

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

Common Instructions

- Your project must be written in accordance with the Norm. If you have bonus files/functions, they are included in the norm check and you will receive a `0` if there is a norm error inside.
- Your functions should not quit unexpectedly (segmentation fault, bus error, double free, etc) apart from undefined behaviors. If this happens, your project will be considered non functional and will receive a `0` during the evaluation.
- All heap allocated memory space must be properly freed when necessary. No leaks will be tolerated.
- If the subject requires it, you must submit a `Makefile` which will compile your source files to the required output with the flags `-Wall`, `-Wextra` and `-Werror`, use cc, and your Makefile must not relink.
- Your `Makefile` must at least contain the rules `$(NAME)`, `all`, `clean`, `fclean` and `re`.
- To turn in bonuses to your project, you must include a rule `bonus` to your Makefile, which will add all the various headers, librairies or functions that are forbidden on the main part of the project. Bonuses must be in a different file `_bonus.{c/h}`. Mandatory and bonus part evaluation is done separately.
- If your project allows you to use your `libft`, you must copy its sources and its associated `Makefile` in a `libft` folder with its associated Makefile. Your project's `Makefile` must compile the library by using its `Makefile`, then compile the project.
- We encourage you to create test programs for your project even though this work **won't have to be submitted and won't be graded**. It will give you a chance to easily test your work and your peers' work. You will find those tests especially useful during your defence. Indeed, during defence, you are free to use your tests and/or the tests of the peer you are evaluating.
- Submit your work to your assigned git repository. Only the work in the git repository will be graded. If Deepthought is assigned to grade your work, it will be done after your peer-evaluations. If an error happens in any section of your work during Deepthought's grading, the evaluation will stop.

Chapter V

Mandatory Part

Program name	`libunit.a`
Turn in files	`*.c, *.h, Makefile`
Makefile	`Yes`
External functs.	`malloc, free, exit, fork, wait, write`
Libft authorized	`Yes`
Description	`Write a micro-framework in C`

V.1 The Micro-framework

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

- The `framework` must be able to execute a series 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 standard 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 execution of the test, 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.

  SIGSEGV : Segmentation Fault detected.

  SIGBUS : Bus Error detected.

- Your output must be formated like this:

```
[test_function]:[test_name]:[status]
```

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

- 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.

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 tests to the chosen function. You must design it to be able to choose to silent one or few test chosen. (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 the test file must begin with a number followed by an underscore which defines 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 with 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;

  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
STRLEN: Basic test : [OK]
STRLEN: 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 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 functional, you can 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 functionality which kills the test process after x time (Watch out for zombie processes)
- Catch more signals.
  - SIGABRT
  - SIGFPE
  - SIGPIPE
  - SIGILL
- Create a log file reporting useful information about the tests. The file must be named [function_name].log

All feature added must be accompagned by a routine of test wich test the feature.