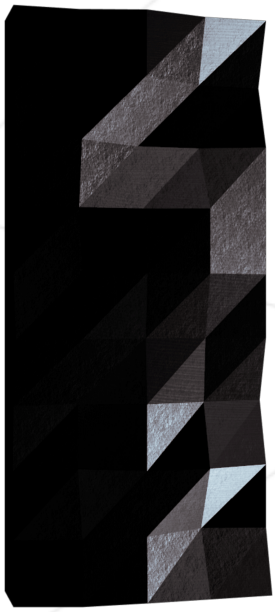


Rush - libunit

What the fork ? ?

Coton coton@42.fr
42 Staff pedago@staff.42.fr

Résumé:
Je ne mettrai plus en prod du code non testé
Je ne mettrai plus en prod du code non testé
Je ne mettrai plus en prod du code non testé
Je ne mettrai plus en prod du code non testé
Je ne mettrai plus en prod du code non testé
Je ne mettrai plus en prod du code non testé
Je ne mettrai plus en prod du code non testé
Je ne mettrai plus en prod du code non testé
Je ne mettrai plus en prod du code non testé
Je ne mettrai plus en prod du code non testé
...

Table des matières

Chapitre I

Préambule

Howard Phillips Lovecraft, né le 20 août 1890 à Providence, Rhode Island, états-Unis et mort le 15 mars 1937 dans la même ville, est un écrivain américain connu pour ses récits d'horreur, fantastique et de science-fiction.

Ses sources d'inspiration, tout comme ses créations, sont relatives à l'horreur cosmique, à l'idée selon laquelle l'homme ne peut pas comprendre la vie et que l'univers lui est profondément étranger. Ceux qui raisonnent véritablement, comme ses protagonistes, mettent toujours en péril leur santé mentale.

On lit souvent Lovecraft pour le mythe qu'il a cré, le mythe de Cthulhu, pour employer l'expression d'August Derleth : l'ensemble des mythes de l'univers de Lovecraft constituaient pour l'auteur une sorte de « panthéon noir », une « mythologie synthétique » ou un « cycle de folklore synthétique ». Il voulait montrer essentiellement que le cosmos n'est pas anthropocentrique, que l'homme, forme de vie insignifiante parmi d'autres, est loin de tenir une place privilégiée dans la hiérarchie infinie des formes de vie. Ses travaux sont profondément pessimistes et cyniques et remettent en question le Siècle des Lumières, le romantisme ainsi que l'humanisme chrétien. Les héros de Lovecraft éprouvent en général des sentiments qui sont à l'opposé de la gnose et du mysticisme au moment où, involontairement, ils ont un aper cu de l'horreur de la réalité.

Bien que le lectorat de Lovecraft fût limité de son vivant, sa réputation évolue au fil des décennies et il est à présent considéré comme l'un des écrivains d'horreur les plus influents du xxe siècle ; avec Edgar Allan Poe, il a « une influence considérable sur les générations suivantes d'écrivains d'horreur ».

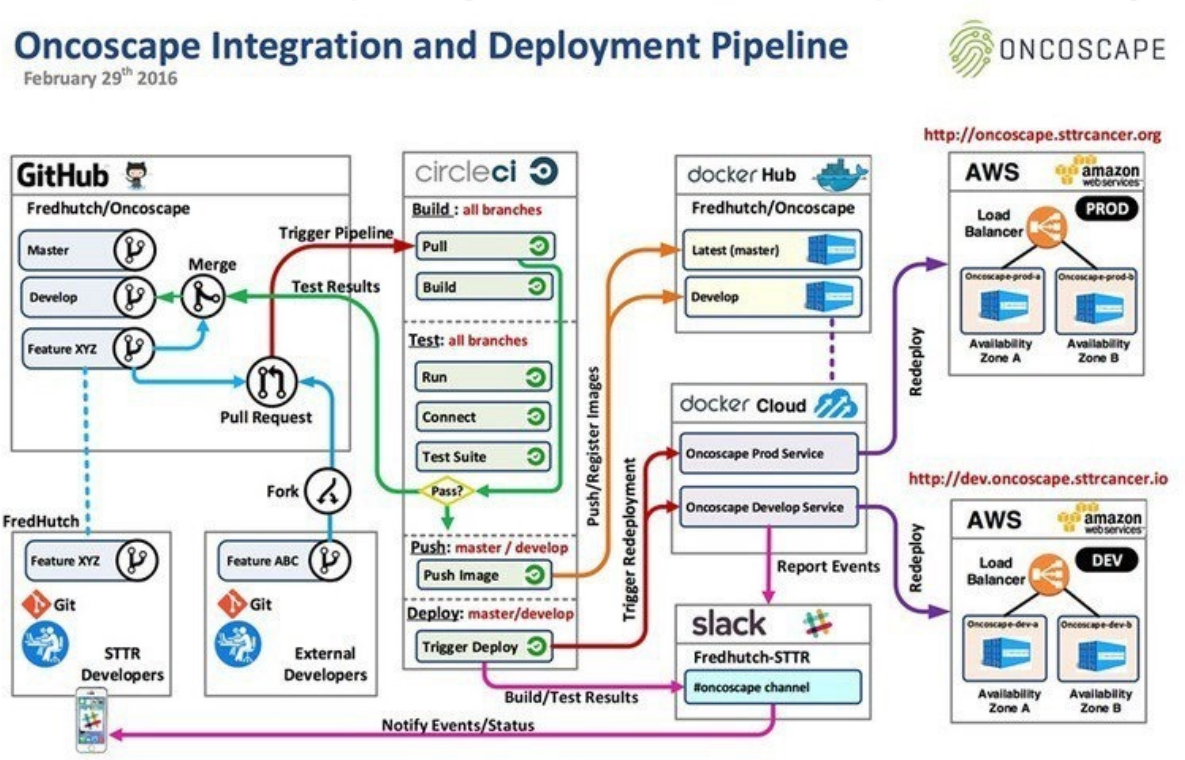
Stephen King a dit de lui qu'il était « le plus grand artisan du récit classique d'horreur du vingtième siècle »

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

Chapitre II

Introduction

Vous vous êtes déjà vraiment demandé comment une feature était mise en production dans une entreprise ?



Voila le pipeline de déploiement de `Oncoscape`. Magnifique hein ? Ce que vous pouvez constater, c'est qu'il y a beaucoup de flèches, jusqu'ici rien de bien sorcier. Toutefois, ce que vous ne voyez pas, c'est que le code passe une série de tests avant de partir en production, une `moulinette` quoi.
L'importance de cette moulinette est capitale en entreprise, car elle décide si la feature passe en production ou non.
Bonne nouvelle ! Ce weekend, vous allez apprendre à faire votre propre moulinette ! Oui vous avez bien lu : **VOTRE PROPRE MOULINETTE** !

Chapitre III

Objectifs

Dans le cadre de ce rush, vous allez concevoir un `Micro-framework` de test en langage C, pour torturer dans tous les sens vos fonctions sur vos projets en C, *avec quelques subtilités en plus.*
Ce `Micro-framework` sera crée sous la forme d'une librairie statique C que vous incluerez dans vos futures routines de tests.

L'objectif intrinsèque de ce rush est de vous donner un moyen ludique et utile d'organiser vos tests unitaires autant pour vos projets à `42` mais aussi plus tard pour vos stages et autres expériences professionelles. Parce que la difference entre un bon developpeur et un excellent developpeur réside dans l'impartialité de ses routines de tests.

Vous verrez aussi qu'il existe beaucoup de solutions de tests unitaires dans les langages de programmation plus récents ou dans les frameworks de developpement :

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

Mais autant apprendre a faire son propre `Micro-framework` en C d'abord !

`Ce rush vous donnera aussi quelques notions pour commencer la branche UNIX, si ce n'est pas déjà le cas. Au pire, vous repartirez dans le monde merveilleux des process. La video de e-learning sur ft_minishell vous sera d'une grande aide pour ce Rush. Elle est disponible sur l'intra.`

Ce sujet vous proposera un cahier des charges simple et minimaliste pour concevoir votre `Micro-framework` mais n'hésitez pas à vous pencher sur les bonus même après le rush. Ce sont des pistes interessantes pour étoffer et consolider votre framework *(et pourquoi pas vous la péter un peu sur GitHub).*

Chapitre IV

Consignes générales

- Ce projet sera corrigé par des humains. Vous êtes donc libres d'organiser et nommer vos fichiers comme vous le désirez, en respectant néanmoins les contraintes listées ici.

- La librairie compilée devra s'appeler `libunit.a`

- Vous devez rendre un Makefile. Ce Makefile devra compiler le projet, et doit contenir les règles habituelles. Il ne doit recompiler le programme qu'en cas de nécessité.

- La routine de test utilisant votre framework ne devra pas quitter de fa con intempestive (Segmentation Fault, double free...). En revanche, votre routine gèrera les cas de sorties intempestives de vos tests unitaires sans planter lamentablement (Voir Partie Obligatoire).

- Votre projet doit être à la Norme.

- Vous devez rendre, à la racine de votre dépôt de rendu, un fichier auteur contenant vos logins suivi d'un '`\n`' :

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

- Dans le cadre de votre partie obligatoire, vous avez le droit d'utiliser les fonctions suivantes :

  - `malloc` et `free`
  - `exit`, `fork` et `wait`
  - `write`
  - `les macros de la librairie <sys/wait.h>`
  - `les macros de la librairie <signal.h>`

Chapitre V

Partie obligatoire

V.1 Le Micro-framework

Dans un premier temps, vous allez concevoir une librairie statique qui sera le moteur de votre `Micro-Framework` et que vous nommerez `libunit`. Cette librairie embarquera au moins une fonction pour lancer un ensemble de sous-fonctions sans interruption et devra analyser la valeur de retour ou le signal d'interruption (SegFault, Bus Error...), ainsi qu'une fonction qui se chargera de charger cet ensemble de sous-fonctions dans le programme.

Votre `Micro-framework` doit répondre au cahier des charges suivant :

- Vos fichiers sources devront se trouver dans un dossier `framework`
- Il doit pouvoir exécuter une serie de tests les uns à la suite des autres, sans interruption (autre que la fin normale de votre routine de tests bien entendu).
- Chaque test est chargé dans une liste/tableau/arbre/whatever... avec un nom spécifique qui devra plus tard être écrit sur la sortie standard.
- Chaque test est exécuté dans un processus à part. Ce processus se termine à la fin du test et redonne la main au processus parent.

```
man fork
```

- Le processus parent doit pouvoir récuperer le resultat du test ou le type d'interruption (au minimum SegFault et BusError)

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

- A la fin de l'execution des tests, votre programme doit afficher le nom de la fonction testée, le nom de chaque test avec le resultat sur la sortie standard avec une nomenclature de ce type :

  OK : Test concluant.

  KO : Test non concluant.

  SEGV : Segmentation Fault détecté.

  BUSE : Bus Error détecté.

- Un descriptif comptabilisant les tests réussis sur le nombre total de tests effectués devra être affiché a la fin.

- Si tous les tests passent, vous devrez quitter votre routine en retournant la valeur 0. Si **AU MOINS** un test est en erreur (-1 ou signal), la routine renverra -1.

- Seul le résultat de chaque test est affiché sur la sortie standard. Voir partie suivante pour plus d'informations.

- La disposition des éléments sur la sortie standard est libre tant qu'elle reste cohérente.

V.2 Les tests

Pour confirmer la toute puissance de votre `Micro-framework`, vous devez pouvoir le faire valider avec une routine de test... de test (Oui, la repetition est voulue). Mais avant, vos tests doivent suivre ce cahier des charges :

- Chaque routine de test doit se trouver dans le dossier `tests/<fonction_a_tester>`
- Chaque test est encapsulé dans une fonction qui doit suivre **OBLIGATOIREMENT** ce prototype (valeurs de retours incluses) :

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

- Les tests ne doivent pas utiliser des fonctions qui écrivent sur la sortie standard (SPOILER ALERT : Ceci est réservé pour un bonus)
- L'arborescence des fichiers de tests doit suivre cette mini-norme :
  - Pour chaque fonction testée, ses tests sont regroupés dans un même dossier, avec un fichier source spécifique appellé `Launcher`.
  - Le `Launcher` sert à ch arger puis lancer l'ensembre des tests d'une fonction donnée. Vous devez concevoir le `Launcher` de telle sorte que vous pouvez sauter un ou plusieurs tests spécifique (que ce soit avec un argument ou en modifiant le code source facilement, soyez créatifs).
  - Une seule fonction de test doit être présent par fichier.
  - Chaque fichier de test commence par un numero suivi d'un underscore, caracterisant son ordre de passage dans le `Launcher` (exemple : 04_basic_test_four_a.c)
  - Le fichier commen cant par 00 est toujours considéré comme le `Launcher`.
  - Le main de test doit être placé à la racine de cette arborescence, et doit appeller tous les `Launchers`. Vous devez concevoir le main de test de telle sorte que vous pouvez sauter un ou plusieurs `Launchers` spécifiques (voir règle plus haut).
  - Le `Makefile` associé au programme doit contenir une dépendance supplémentaire appellée `test` qui compilera votre programme avec les fichiers de tests et qui lancera l'executable crée.
  - La norme sur le nombre de lignes par fonction ainsi que sur le nommage du fichier ne s'applique pas sur vos sources de tests (main, launchers et tests).

V.3 Exemple de sortie

Exemple basique d'arborescence de test :

```
$> ls -R tests
main.c  strlen

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

Exemple de test :

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

Exemple de 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));
}
$>
```

Exemple de sortie d'une routine de test :

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

1/2 tests checked
$>
```

V.4 Votre rendu

Pour valider votre rush, vous devrez rendre :

- Les sources de votre `Micro-framework` dans le dossier `framework`
- Une routine de tests dans le dossier `tests` avec :
  - Un **VRAI** test qui renvoie OK
  - Un **VRAI** test qui renvoie KO
  - Un **VRAI** test qui renvoie Segmentation Fault
  - Un **VRAI** test qui renvoie Bus Error

- Une routine d'au moins 15 tests de votre choix sur un sujet existant (Libft ?) dans le dossier `real-tests`, avec le projet a tester en plus. Le choix des tests est décisif pour la notation.

Vous devrez créer un Makefile pour le framework pour compiler une librairie statique embarquant votre moteur de test.
De même, vous devrez créer ou modifier un Makefile dans `tests` et `real_tests`, de telle sorte que la commande `make test` lancée depuis l'un des deux dossiers exécute la routine en entier.
N'oubliez pas que vos sets de tests doivent respecter obligatoirement la mini-norme des routines de tests, en plus de la Norme elle même. Mis à part cela, profitez-en et lâchez-vous !
Pour une fois qu'on vous demande de rendre du code foireux.

Chapitre VI

Partie bonus

Une fois votre `Micro-framework` créé, libre à vous d'ajouter les éléments de votre choix pour qu'il soit encore plus swag.

Vous pouvez :

- Ajouter un code couleur sur les résultats de tests
- Ajouter le support des fonctions écrivant dans la sortie standard. (Attention : La fonction testée ne doit toujours pas écrire sur la sortie standard.)
- Ajouter une fonctionnalité de timeout qui tue le processus du test une fois qu'un certain temps soit écoulé (Attention aux process zombies)
- Ajouter d'autres signaux à capturer... tant que c'est cohérent. N'allez pas catcher les signaux SIGUSRx si cela ne vous sert à rien dans vos routines de tests.
- Créer un fichier de reporting contenant le rapport de test, avec les informations que vous jugez nécessaires.
- Etc...

Un **ENORME PLUS** est accordé à la mise en place d'une solution d'Integration Continue. Le support vous est libre, tant qu'il ne vous coute rien. D'ailleurs, si vous comptez réaliser ce bonus, vous pouvez mettre votre routine de test ainsi que votre framework sur GitHub. Vous pouvez utiliser les solutions suivantes :

- Jenkins
- Travis
- Circle CI
- ...

`Vous verrez rapidement que la majorité des solutions de CI existantes ont un cout dans certains cas d'utilisation. 42 et le rédacteur de ce PDF se dégagent de toute responsabilité quant a un quelconque remboursement d'utilisation de ces services.`

Chapitre VII

Rendu et peer-évaluation

Rendez-votre travail sur votre dépot `GiT` comme d'habitude. Seul le travail présent sur votre dépot sera évalué en soutenance.

Si vous avez opté pour le bonus "Intégration Continue", la mise en place de votre solution sera testée en peer-correction. Vous devrez alors expliquer votre mise en place à votre correcteur, et lui montrer les joyeusetés de cette magnifique chose qu'est le `CI`.