Framework de tests

Programmation séquentielle en C, 2020-2021

Orestis Malaspinas (A401), ISC, HEPIA

2021-03-17

Inspirés des slides de F. Glück

Tests manuels (1/2)

Instructions conditionnelles

Utilisation *d'instruction conditionnelles* pour les tests unitaires:

```c
void test_add() { // test some add function
    if (add(1, -2) != -1) {
        printf("Error. Expected %d, Actual %d.\n",
            -1, add(1, -2));
        exit(-1);
    }
}
int main() {
    test_add();
}
```

```
$ ./tests
Error. Expected -1, Actual -2.
```

Erreurs détaillées mais *long à écrire* et *pas généralisable*.

Tests manuels (2/2)

Assertions

Utilisation des *assertions* pour les tests unitaires:

```
void test_add() { // test some add function
    assert(add(1, -2) == -1);
}

int main() {
    test_add();
}
```

Exécution:

```
$ ./tests
tests.c:6: your_function:
    Assertion `add(1, -2) == -1' failed.
```

Simple à écrire et *généralisable* mais **pas détaillé**.

Frameworks de tests

Grand nombre de frameworks:

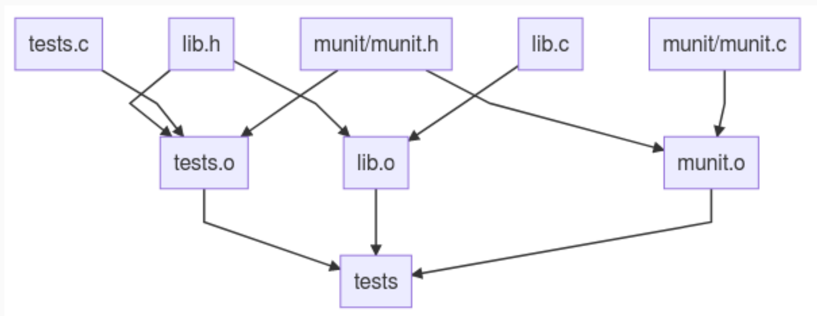
- CUnit, https://sourceforge.net/projects/cunit/
- Unity, https://github.com/ThrowTheSwitch/Unity
- Criterion, https://github.com/Snaipe/Criterion/

Ici on utilisera:

- μnit, https://nemequ.github.io/munit/, https://github.com/nemequ/munit
  - Simple à installer,
  - Simple à utiliser,
  - Erreurs détaillées et général.

Petit tuto μnit

- Pour plus d'informations voir ce lien.
- Structure possible de tests:



Exercice: Écrire un `Makefile` avec une cible `tests` qui compile et exécute les tests.

Utilisation simple

Fichier `tests.c`

```c
#include "munit/munit.h" // inclusion du header munit.h

int main() {
  // Le code vient ici
}
```

Mais que mettre dans le code?

Les tests les plus simples possibles

Une grande collection de macros

```c
#include "munit/munit.h" // inclusion du header munit.h

int main() {
      munit_assert_int(add(1, -2), ==, -1);
}
```

```
$ ./tests
ERROR> tests.c:5: assertion failed:
    add(1, -2) == -1 (-2 == -1)
```

Plusieurs saveurs de `munit_assert_TYPE()` (`char`, `float`, ...).

Structure des tests (1/3)

Fonction de test

- Chaque test doit être dans une fonction séparée.
- Chaque fonctionnalité doit avoir sa propre fonction de test.
- Le nom de la fonction doit ressembler à:

```
// - MunitResult: type de retour du test
//     - MUNIT_OK, MUNIT_SKIP, MUNIT_FAIL, MUNIT_ERROR
// - nom_du_test: l'identifiant de la fonction
// - params: les paramètres du test
// - user_data_or_fixture: permet plus de flexibilité
MunitResult nom_du_test(const MunitParameter params[],
    void* user_data_or_fixture);
```

- On s'intéresse pas à `params` et `user_data_or_fixture`.
- Le nombre de tests est arbitraire, maintenant voyons comment les exécuter.

Structure des tests (2/3)

Liste de tests

- Les tests sont mis dans une liste:

```
MunitTest mes_tests[] = {
    {
        "/mon_super_test", /* nom humain du test */
        nom_du_test, /* nom dela fonction de test */
        NULL, /* setup */
        NULL, /* tear_down */
        MUNIT_TEST_OPTION_NONE, /* options */
        NULL /* parameters */
    },
    // On peut rajouter plein d'autres tests
    { NULL, NULL, NULL, NULL, MUNIT_TEST_OPTION_NONE, NULL  }
    // La ligne ci-dessus indique qu'il n'y a pas d'autres tests
    // Elle est obligatoire
};
```

- `setup`, `tear_down`, `options`, `parameters` servent à paramétrer les tests on les laisse à `NULL` ici.

Structure de tests (3/3)

Suite de tests

- Quand on a notre liste de tests on peut les mettre dans une suite

```
const MunitSuite suite = {
    "/ma_suite", /* name */
    mes_tests, /* liste de tests */
    NULL, /* suites */
    1, /* iterations, pour les tests PRNG */
    MUNIT_SUITE_OPTION_NONE /* options */
};
```

On mélange et on secoue tout (1/2)

Fichier `tests.c`

```c
#include "munit/munit.h"
MunitResult test_init( // fonction de test
    const MunitParameter params[], void* user_data_or_fixture)
{
    stack init = stack_init();
    munit_assert_ptr_equal(init, NULL);
    return MUNIT_OK;
}
MunitTest test_suite_tests[] = { // liste de tests
    {
        "/test_init", /* name */
        test_init, /* test */
        NULL, /* setup */
        NULL, /* tear_down */
        MUNIT_TEST_OPTION_NONE, /* options */
        NULL /* parameters */
    },
    /* La fin de la liste de tests */
    { NULL, NULL, NULL, NULL, MUNIT_TEST_OPTION_NONE, NULL }
};
```

On mélange et on secoue tout (2/2)

Fichier `tests.c`

```c
/* La suite de tests qu'on va exécuter */
const MunitSuite test_suite = {
  /* Une chaîne de caractères mise avant chaque test de la suite. */
  (char*) "",
  /* La liste de tests. */
  test_suite_tests,
  /* Une autre suite, ici on a rien d'autre. */
  NULL,
  /* Le nombre de fois qu'on va exécuter les tests. Utile pour des tests
  random. */
  1,
  /* Des options qu'on utilise pas ici. */
  MUNIT_SUITE_OPTION_NONE
};

int main(int argc, char** argv) {
    /* On lance notre suite de tests: argc, argv pourraient être NULL. */
    return munit_suite_main(&test_suite, NULL, argc, argv);
}
```

Comment inclure `munit` dans le code?

Deux choix principaux:

Copier les fichiers depuis ce lien

- On a besoin que des fichiers `munit.h` et `munit.c`.

L'utilisation de sous-modules git

- Dans un repo git existant, ici `~/test`:

```
$ git submodule add https://github.com/nemequ/munit
Cloning into '~/test/munit'...
remote: Enumerating objects: 615, done.
remote: Total 615 (delta 0), reused 0 (delta 0), pack-reused 615
Receiving objects: 100% (615/615), 245.72 KiB | 2.61 MiB/s, done.
Resolving deltas: 100% (401/401), done.
$ ls -ltr
total 4
drwxr-xr-x 2 orestis orestis 4096 Mar 16 23:18 munit
```