CUnit: librairie de tests

CUnit est une librairie de tests unitaires en C. Cette librairie vous sera utile lors de développement de projets en C.

Installation

CUnit n'est pas installé par défaut sur les machines des salles. Vous devez donc l'installer par vous-même. Le reste de cette section a pour but de vous aider dans l'installation de celle-ci.

La première étape consiste à récupérer les sources de CUnit sur **https://sourceforge.net/projects/cunit/files/**. Les sources se trouvent dans une archive `CUnit-*-src.tar.bz2` et la dernière version devrait se nommer `CUnit-2.1-3-src.tar.bz2`. Une fois l'archive téléchargée, ouvrez un terminal et placez-vous dans le dossier où se trouve celle-ci. Exécutez:

```
$ tar xjvf CUnit-2.1.-3.tar.bz2
$ cd CUnit-2.1-3
$ ./bootstrap
$ ./configure --prefix=$HOME/local
$ make
$ make install
```

Une fois ces commandes exécutées, la librairie ainsi que ses fichiers d'entête sont installés dans le dossier `$HOME/local` (`$HOME` est en fait une variable bash qui définit votre répertoire principal). Comme vous n'avez pas les droits administrateur, vous ne pouvez pas installer d'application ni de librairie dans les chemins classiques (c.-à-d., par exemple dans `/usr/lib`, `/usr/include`, `/usr/bin`). C'est pour cela que nous installons la librairie dans un dossier local.

Compilation, édition des liens et exécution

Comme la librairie n'est pas installée dans les chemins classiques, il faut pouvoir dire à gcc où se trouvent les fichiers d'entête ainsi que la librairie afin d'éviter les erreurs de compilation. Pour cela, il faut spécifier à la compilation l'argument `-I${HOME}/local/include` afin de lui dire qu'il doit également aller chercher des fichiers d'entête dans le dossier `$HOME/local/include` en plus des chemins classiques tels que `/usr/include` et `/usr/local/include`.

Lors de l'édition des liens avec le linker, il faut spécifier où se trouve la librairie dynamique afin de résoudre les symboles. Pour cela, il faut passer l'argument `-lcunit` pour effectuer la liaison avec la librairie CUnit ainsi que lui spécifier `-L${HOME}/local/lib` afin qu'il cherche également des librairies dans le dossier `$HOME/local/lib`.

Lors de l'exécution, il faut également spécifier où se trouvent les librairies. Par exemple pour un binaire `test` qui utilise la librairie CUnit, on peut exécuter:

```
$ export LD_LIBRARY_PATH=$HOME/local/lib:$LD_LIBRARY_PATH
$ ./test
```

Utilisation

nécessaires

Dans CUnit, on retrouve toutes les fonctions nécessaire pour gérer un ensemble de suites de tests. Cet ensemble forme un catalogue; il est composé d'une ou plusieurs suite(s) de tests; chaque suite est composée d'un ou plusieurs tests.

Pour pouvoir concrètement exécuter un ensemble de tests, il est nécessaire de réaliser les différentes étapes suivantes:

1. Programmer les tests
2. Initialiser le catalogue
3. Ajouter les suites de tests dans le catalogue
4. Ajouter les tests dans les suites de tests
5. Exécuter les tests
6. Terminer proprement l'exécution des tests

La suite de la section détaille chacune de ces étapes.

Tout d'abord, il est nécessaire d'écrire les tests. Aucune libraire ne peut les écrire pour vous. Toutefois, CUnit vient avec un certain nombre de macros permettant de vérifier les propriétés qui nous intéressent. Pour pouvoir utiliser ces macros, il est nécessaire d'importer `CUnit.h`. La table suivante récapitule les principales macros. Il est important d'appeler ces macros lorsque l'on rédige les tests, ce sont ces appels qui détermineront si oui ou non, le test est fructueux.

Assertion	Définition
`CU_ASSERT(int expression)`	Vérifie que la valeur est non-nulle (`true`).
`CU_ASSERT_TRUE(value)`	Vérifie que la valeur est non-nulle (`true`).
`CU_ASSERT_FALSE(value)`	Vérifie que la valeur est nulle (`false`).
`CU_ASSERT_EQUAL(actual, expected)`	Vérifie que `actual` est égal à `expected`.
`CU_ASSERT_NOT_EQUAL(actual, expected))`	Vérifie que `actual` n'est pas égal à `expected`.
`CU_ASSERT_PTR_EQUAL(actual, expected)`	Vérifie que le pointeur `actual` est égal au pointeur `expected`.

Assertion	Définition
`CU_ASSERT_PTR_NOT_EQUAL(actual, expected)`	Vérifie que le pointeur `actual` est différent du pointeur `expected`.
`CU_ASSERT_PTR_NULL(value)`	Vérifie que le pointeur est `NULL`.
`CU_ASSERT_PTR_NOT_NULL(value)`	Vérifie que le pointeur n'est pas `NULL`.
`CU_ASSERT_STRING_EQUAL(actual, expected)`	Vérifie que la chaîne de caractère `actual` est égale à la chaîne de caractère `expected`.
`CU_ASSERT_STRING_NOT_EQUAL(actual, expected)`	Vérifie que la chaîne de caractère `actual` n'est pas égale à la chaîne de caractère `expected`.
`CU_ASSERT_NSTRING_EQUAL(actual, expected, count)`	Vérifie que les `count` premiers caractères de la chaîne `actual` sont égaux aux `count` premiers caractères de la chaîne `expected`.
`CU_ASSERT_NSTRING_NOT_EQUAL(actual, expected, count)`	Vérifie que les `count` premiers caractères de la chaîne `actual` ne sont pas égaux aux `count` premiers caractères de la chaîne `expected`.
`CU_ASSERT_DOUBLE_EQUAL(actual, expected, granularity)`	Vérifie que `actual` et `expected` ne diffèrent pas plus que `granularity` ($\|actual - expected\| \leq \|granularity\|$)
`CU_ASSERT_DOUBLE_NOT_EQUAL(actual, expected, granularity)`	Vérifie que `actual` et `expected` diffèrent de plus que `granularity` ($\|actual - expected\| > \|granularity\|$)
`CU_PASS(message)`	Ne vérifie rien mais notifie que le test est réussis
`CU_FAIL(message)`	Ne vérifie rien mais notifie que le test est raté

Par exemple, les méthodes ci-dessous vérifie chacune certaines propriétés.

```
void test_assert_true(void)
{
  CU_ASSERT(true);
}

void test_assert_2_not_equal_minus_1(void)
{
  CU_ASSERT_NOT_EQUAL(2, -1);
}

void test_string_equals(void)
{
  CU_ASSERT_STRING_EQUAL("string #1", "string #1");
}

void test_failure(void)
{
  CU_ASSERT(false);
}

void test_string_equals_failure(void)
{
  CU_ASSERT_STRING_EQUAL("string #1", "string #2");
}
```

Une fois les tests écrits, il faut initialiser le catalogue (et donc l'infrastructure de tests) en appelant la méthode `CU_initialize_registry()`. Cette méthode retourne un code d'erreur qu'il est impératif de vérifier pour s'assurer du bon fonctionnement de la vérification des tests. Par exemple,

```
if (CUE_SUCCESS != CU_initialize_registry())
  return CU_get_error();
```

Pour ajouter les suites de tests au catalogue, il faut faire appel à la méthode `CU_add_suite(const char* strName, CU_InitializeFunc pInit, CU_CleanupFunc pClean)`. Comme on peut le voir, cette méthode demande un nom (qui doit être unique pour un catalogue) ainsi que deux pointeurs de fonction. Ces pointeurs de fonction permettent d'exécuter du code avant (typiquement appelé *setup*) ou après (typiquement *teardown*) l'exécution des tests de la suite. Ces méthodes sont utiles pour initialiser un environnement d'exécution pour des tests le nécessitants (par exemple, s'assurer de la présence de fichier, initialiser certaines variables, etc.). Ces méthodes sont bien sûr optionnelles, si aucune n'est nécessaire, il suffit alors de passer `NULL` en paramètre. Par ailleurs, notons que ces méthodes doivent retourner 0 si tout c'est bien passé, un chiffre positif dans le cas contraire. Comme pour l'initialisation du catalogue, il est bien entendu nécessaire de vérifier le code retourné par la méthode. La table suivante décrit les codes d'erreurs.

Code d'erreur	Définition
`CUE_SUCCESS`	Aucune erreur
`CUE_NOREGISTRY`	Erreur d'initialisation
`CUE_NO_SUITENAME`	Nom manquant
`CUE_DUP_SUITE`	Nom non unique
`CUE_NOMEMORY`	Pas de mémoire disponible

Par exemple, le code suivant crée une nouvelle suite de test nommée *ma_suite*, avec une fonction d'initialisation et une fonction de terminaison.

```
int setup(void)  { return 0; }
int teardown(void) { return 0; }
// ...
CU_pSuite pSuite = NULL;
// ...
pSuite = CU_add_suite("ma_suite", setup, teardown);
if (NULL == pSuite) {
  CU_cleanup_registry();
  return CU_get_error();
}
```

Les tests peuvent ensuite être ajoutés à la suite de test. Pour cela, il faut faire appel à la méthode `CU_add_test(CU_pSuite pSuite, const char* strName, CU_TestFunc pTestFunc)`. Comme pour une suite de tests, il est nécessaire de préciser un nom. Ce nom doit être unique pour la suite de test. Le second paramètre est un pointeur vers la fonction de test. A nouveau, il est important de vérifier la valeur de retour de la méthode.

Code d'erreur	Définition
`CUE_SUCCESS`	Aucune erreur
`CUE_NOSUITE`	Suite de tests `NULL`
`CUE_NO_TESTNAME`	Nom manquant
`CUE_DUP_TEST`	Nom non unique
`CUE_NO_TEST`	Pointeur de fonction `NULL` ou invalide
`CUE_NOMEMORY`	Pas de mémoire disponible

Le code suivant ajoute les tests décrits ci-dessus à la suite de test que nous avons créé juste avant.

```
if ((NULL == CU_add_test(pSuite, "Test assert true", test_assert_true)) ||
    (NULL == CU_add_test(pSuite, "Test assert 2 not equal -1", test_assert_2_not_equal_minus_1)) ||
    (NULL == CU_add_test(pSuite, "Test string equals", test_string_equals)) ||
    (NULL == CU_add_test(pSuite, "Test failure", test_failure)) ||
    (NULL == CU_add_test(pSuite, "Test string equals failure", test_string_equals_failure)))
{
  CU_cleanup_registry();
  return CU_get_error();
}
```

Maintenant que le catalogue est initialisé, qu'il contient des suites de tests et que les tests ont été ajoutés à ces suites, il nous est possible d'exécuter ces tests. Il existe plusieurs moyens d'exécuter les tests CUnit, nous présentons uniquement le mode de base, non interactif. Pour les autres modes, referez-vous à la **documentation**. Pour faire tourner les tests, il suffit d'appeler la méthode `CU_basic_run_tests()` qui appellera tous les tests dans toutes les suites des catalogues référencés. Ensuite, on peut afficher le rapport à l'aide de `CU_basic_show_failures(CU_pFailureRecord pFailure)` et `CU_get_failure_list()`.

```
CU_basic_run_tests();
CU_basic_show_failures(CU_get_failure_list());
```

Avec le programme illustré ci-dessous, la console nous affiche les messages suivants :

```
     CUnit - A unit testing framework for C - Version 2.1-2
     http://cunit.sourceforge.net/


Suite ma_suite, Test Test failure had failures:
    1. cunit.c:24  - false
Suite ma_suite, Test Test string equals failure had failures:
    1. cunit.c:29  - CU_ASSERT_STRING_EQUAL("string #1","string #2")

Run Summary:    Type  Total    Ran Passed Failed Inactive
              suites      1      1    n/a      0        0
               tests      5      5      3      2        0
             asserts      5      5      3      2      n/a

Elapsed time =    0.000 seconds


  1. cunit.c:24  - false
  2. cunit.c:29  - CU_ASSERT_STRING_EQUAL("string #1","string #2")
```

Enfin, il est nécessaire de libérer les ressources en appelant `CU_cleanup_registry()`.