

Ch. 13 - Tests unitaires avec Catch

Langage C++

R. Absil

Haute École Bruxelles-Brabant
École supérieure d'Informatique



23 juin 2017

Table des matières

- 1 Introduction
- 2 Configuration
- 3 Tests simples
- 4 Scénarios de test

Introduction

Contexte

- Le standard C++ ne fournit pas de framework de tests
 - Pas d'équivalent Java « JUnit »
- Il existe de nombreux frameworks permettant de faire des tests unitaires
- Certains ont des dépendances vers des librairies tierces
 - P. ex., QTest
- Volonté :
 - 1 Écrire des test *unitaires*
 - 2 Écrire des tests sans dépendances externes
 - 3 Écrire des tests « parlants »

Frameworks de test

- Exemples de frameworks de test :
 - ATF : développé à l'origine pour NetBSD
 - Boost Test Library : inclus à la librairie Boost
 - QTest : inclus à la librairie Qt
 - Catch : header unique, inclut style BDD
 - CPPOCL/test : header unique, sans style BDD
 - etc.
- Source : « List of unit testing frameworks » (Wikipedia, 23/06/2017)

Le framework Catch

- Dans le cadre de ce cours, le framework Catch est présenté



- Ce framework
 - n'a pas de dépendance externe
 - est composé d'un unique header fournissant la syntaxe
 - dispose d'une syntaxe simple
 - permet de faire des tests en style BDD
 - Behaviour-driven development

Configuration

Installation

- Pour utiliser Catch, il suffit d'importer le header dans votre projet
 - Copier / coller
 - Chemin d'accès, etc.
- Ce header fournit
 - 1 la syntaxe des tests
 - 2 une méthode `main` de tests

Remarque

- Ce header est *gros*, le recompiler peut prendre du temps

Décider de lancer ses tests

- À l'exécution, il faut décider de lancer ou non les tests

Question

- Si vous n'avez pas la possibilité de définir plusieurs `main`, que faire ?
 - Compilation en ligne de commande, sans `makefile`
 - L'IDE ne permet pas de le faire
- Il faut indiquer à l'éditeur des liens que faire

Configuration *ex nihilo*

■ Utiliser le préprocesseur et la directive

```
#define CATCH_CONFIG_RUNNER
```

■ Fichier `main.cpp`

```
1  #define CATCH_CONFIG_RUNNER
2  #include "catch.hpp"
3  #define RUN_TEST 1 //change this value
4
5  #if RUN_TEST
6  int main(int argc, char* const argv [])
7  {
8      Catch::Session().run(argc, argv);
9  }
10 #else
11 #include <iostream>
12 using namespace std;
13
14 int main()
15 {
16     cout << "Regular_main" << endl;
17 }
18 #endif
```

Configuration avec QtCreator

- Utiliser le template « subdirs project »
 - File > New project > Other projects > subdirs project
- Ce template permet de définir un projet comme un ensemble de sous-projets
 - Partie métier, partie contrôleurs, partie graphique, partie tests, etc.
- Chaque sous-projet peut avoir son propre `main`
- Souvent, on aura donc
 - 1 un `main` pour lancer l'application
 - 2 un `main` pour lancer les tests
- On lance le sous-projet des tests pour lancer les tests
- On lance le sous-projet « main » pour lancer l'application

Exemple

- **Fichiers** ObserverDemo/ObserverDemo.pro,
ObserverDemo/ObserverDemo.pro,
ObserverDemo/core/core.pro, etc.

```
1  TEMPLATE = subdirs
2
3  SUBDIRS += \
4      core \
5      tests \
6      controllers \
7      console \
8
9  OTHER_FILES += \
10     defaults.pri
```

```
1  INCLUDEPATH += $$PWD/core
2
3  SRC_DIR = $$PWD
4
5  CONFIG -= app_bundle
6  CONFIG -= qt
7  CONFIG += c++14
```

Exemple

- **Fichiers** ObserverDemo/ObserverDemo.pro,
ObserverDemo/ObserverDemo.pro,
ObserverDemo/core/core.pro, etc.

```
1 include (../defaults.pri)
2
3 TEMPLATE = lib
4 TARGET = libcore
5 DESTDIR = ../lib
6
7 HEADERS += \
8     array.hpp \
9     bubblesort.hpp \
10    heapsort.hpp \
11    insertion sort.hpp \
12    integerarraygenerator.h \
13    permutationsort.hpp \
14    sortalgorithm.hpp
15
16 SOURCES += \
17     integerarraygenerator.cpp
```

Tests simples

Exemple

■ Fichier `simple.cpp`

```
1  TEST_CASE("Testing_array_instanciation")
2  {
3      SECTION("Testing_list_instanciation")
4      {
5          Array<int> a = {1,2,3,4,5};
6          REQUIRE(a.size() == 5);
7
8          for(int i = 0; i < a.size(); i++)
9              REQUIRE(a[i] == i + 1);
10     }
11
12     SECTION("Testing_copy_cstr")
13     {
14         Array<int> a = {1,2,3,4,5};
15         Array<int> b(a);
16         REQUIRE(a.size() == b.size());
17
18         for(int i = 0; i < a.size(); i++)
19             REQUIRE(a[i] == b[i]);
20     }
21 }
```

Exceptions

- Il est possible de tester si une exception est lancée
- Possibilité de tester le type de l'exception

Syntaxe

- `REQUIRE_NO_THROW (expression) ;` teste qu'aucune exception ne soit lancée
- `REQUIRE_THROWS (expression) ;` teste qu'une exception soit lancée
- `REQUIRE_THROWS_AS (expression, excpt-type) ;` teste qu'une exception d'un type donné soit lancée

Exemple

■ Fichier `excpt.cpp`

```
1  TEST_CASE( "Array_access" )
2  {
3      SECTION( "Acces_within_bounds" )
4      {
5          Array<int> a = {1,2,3,4,5};
6
7          for( int i = 0; i < a.size(); i++)
8          {
9              REQUIRE_NOTHROW(a[i]);
10             REQUIRE(a[i] == i + 1);
11         }
12     }
13
14     SECTION( "Acces_out_of_bounds" )
15     {
16         Array<int> a = {1,2,3,4,5};
17
18         REQUIRE_THROWS(a[-1]);
19         REQUIRE_THROWS_AS(a[-1], std::out_of_range);
20
21         REQUIRE_THROWS(a[5]);
22         REQUIRE_THROWS_AS(a[5], std::out_of_range);
23     }
24 }
```

Instructions multiples

- Les assertions relatives au lancement d'exception attendent une unique expression en paramètre
 - Pas une instruction
 - Pas une suite d'instructions
- Si l'on veut tester le lancement sur une série d'instructions, il faut utiliser une lambda

```
1 REQUIRE_NO_THROW([&]()  
2 {  
3     int i = 1;  
4     int j = 2;  
5     auto k = i + j;  
6     if (k == 3)  
7         throw 1;  
8 }());
```

Scénarios de test

Contexte

- Parfois, la structure des tests « simples » est peu parlante
 - Tests, longs, cas complexes de tests, etc.
- Il n'existe pas de documentation standard pour les tests
- Si l'on veut relire / adapter des tests, c'est parfois difficile

Solution

- Il faudrait une syntaxe de tests permettant de les comprendre « facilement »

Scénarios

- Possibilité de définir des scénarios

Idée

- Possibilité de dire
 - « étant donné ces données »
 - « quand une condition est remplie »
 - « action une action est réalisée »
- Avantages : tests plus lisibles
- Inconvénients : tests très verbeux

Exemple

■ Fichier scenario.cpp

```
1 SCENARIO("You_cannot_access_an_array_out_of_its_bounds")
2 {
3     GIVEN("An_array_with_5_items")
4     {
5         Array<int> a = {1,2,3,4,5};
6         WHEN("The_array_is_accessed_within_bounds")
7         {
8             THEN("The_element_is_accessed")
9             {
10                 for(int i = 0; i < a.size(); i++)
11                 {
12                     REQUIRE_NO_THROW(a[i]); REQUIRE(a[i] == i + 1);
13                 }
14             }
15         }
16         WHEN("The_array_is_accessed_out_of_its_bounds")
17         {
18             THEN("A_std::out_of_bound_exception_is_thrown")
19             {
20                 REQUIRE_THROWS(a[-1]); REQUIRE_THROWS_AS(a[-1], std::out_of_range);
21                 REQUIRE_THROWS(a[5]); REQUIRE_THROWS_AS(a[5], std::out_of_range);
22             }
23         }
24     }
25 }
```