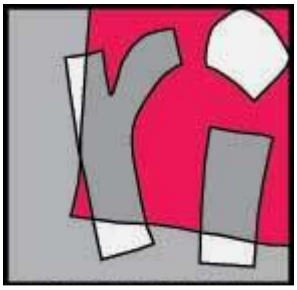
ISIMA

Institut **S**upérieur
d'**I**nformatique de **M**odélisation
et de leurs **A**pplications
Campus des Cézeaux
24 avenue des Landais
BP 10125
63173 AUBIERE cedex France



Laboratoire de **R**echerche
en **I**nformatique

Bât 650
Université Paris Sud
91405 ORSAY Cedex France

Rapport d'ingénieur

Projet de 2ème année

Filière Génie Logiciel et Systèmes Informatiques

Génération et Rapport de Tests de Méta-programmes

Présenté par : **Samuel Morel et Romain Valette**

Responsable entreprise : **Joël Falcou**

18 Mars 2013

Remerciements

Merci à Joël Falcou de nous avoir guidés dans ce projet, et d'avoir été présent, malgré la distance entre les deux établissements.

Table des figures

Listings

Résumé

Le sujet de ce projet était la génération de tests, ainsi que leurs rapports, de meta-programmes écrit en C++. C'est-à-dire, la génération de tests sur des programmes C++ pouvant constituer une API (*Application Programming Interface*). Ce projet a pour but de tester une API en vérifiant qu'elle est bien conforme aux attentes, et qu'un code réalisant des actions interdites par l'API est bien reporté comme erroné. Pour cela, nous devions utiliser un langage de script comme CMake pour générer des tests devant réussir si, et seulement si, des erreurs connues sont levées. Il nous a donc fallu créer une macro CMake qui essaie de compiler le code pour en tirer les erreurs de compilation et vérifier, avec un test généré, que les erreurs attendues sont présentes à l'aide de CTest. Si c'est le cas, le test est validé. Si des erreurs non attendues sont levées, le test échoue. Il a donc été nécessaire de rédiger un modèle de test couvrant toutes les possibilités. De même, ce script doit être portable et doit pouvoir s'exécuter sur Windows, Linux/Unix et Mac. Nous avons donc dû utiliser des fonctions de base, non spécifiques.

Mots-clés : CMake, CTest, tests unitaires, API, C++, multiplateforme, compilation

Abstract

The subject of this project was the generation of C++ meta-program tests and their repports. That is to say the generation of test in C++ that could be part of an API (*Application Programming Interface*). This project aim to test an API verifying that it's congruent to the objectif, and that a code which make API forbidden actions is well reported as bad. For that, we had to use a script language, as CMake, to generate tests which had to succeed if, and only if, known errors are thrown. We had to create a CMake macro which try to compile the code and then catch compilation errors and check, with a generated test, that expected errors are found thanks to CTest. If it's the case, the test is validated. If unexpected errors are found, the test fail. We had to code basics tests model covering all possibilities. Similary, this script has to be portative and has to be executable on Windows, Linux/Unix and Mac, so we had to use basic functions, not specific ones.

Keywords : CMake, CTest, Unit testing, API, C++, cross-platform, compilation

Table des matières

Introduction

Une API (*Application Programming Interface*) est ce qu'on appelle une interface de programmation. C'est ce qui permet à des programmes de communiquer. C'est, d'un point de vue purement technique, un ensemble de fonctions, méthodes, classes, etc. fourni par un programme, une bibliothèque logicielle, un service ou bien plus globalement, le système d'exploitation. Notre projet visait à permettre la validation d'une API, c'est-à-dire dans notre cas, vérifier qu'une API ne permet pas ce qu'elle ne doit pas permettre. Ceci s'effectue via un script CMake générant des tests qui ne passent que si les erreurs attendues sont levées, c'est-à-dire si un code qui utilise l'API de manière non exacte reçoit bien des messages d'erreurs. Mais ces messages d'erreurs ne sont pas n'importe lesquels, ils doivent être ceux que nous attendons. Nous devions donc tout d'abord comprendre le fonctionnement d'une API ainsi que le fonctionnement de CMake. C'est pourquoi, une grande partie de ce projet a été consacrée à la découverte de CMake. Nous n'avions pas d'exemple concret, nous avons donc dû en créer des pertinents pour pouvoir répondre à la problématique. Il fallait ensuite, avant de faire un script CMake spécialisé dans la validation, réaliser un script simple, permettant de compiler et de tester de manière simple nos exemples à l'aide de CTest. Après cela, nous nous sommes lancés dans la création d'un script spécifique, répondant au mieux à la problématique.

Nous allons maintenant exposer ce projet en présentant dans un premier temps CMake et CTest, puis nous montrerons notre solution pour finir avec un guide d'utilisation indiquant comment se servir de notre outil.

1 Présentation de la suite CMake

1.1 CMake

CMake est un outil de construction logicielle (ou moteur de production) multi-plateforme. Son développement a commencé en 1999 comme une partie du projet *Insight Toolkit* (*ITK*, *www.itk.org*). *ITK* est un gros projet logiciel qui fonctionne sur plusieurs plateformes et qui interagit avec plusieurs autres packages (pouvant être différents suivant l'OS). Pour gérer toutes ces dépendances en faisant abstraction du système hôte, l'équipe de l'*US National Library of Medicine* a débuté la création de *CMake*. Depuis, *CMake* est maintenu par *Kitware* et d'autres projets open-source comme *K Desktop Environment* (*KDE*).

CMake est disponible sur *www.cmake.org*, il est gratuit sous licence *BSD*.

Figure 1 : Logo CMake

Il existe d'autres outils fournissant, à peu de choses près, les mêmes fonctionnalités que *CMake*. *QMake* est le plus similaire à *CMake*. Sa syntaxe se rapproche de celle d'un *Makefile* classique. *ANT* utilise *XML*, *SCons* du *Python* et *JAM* a créé son propre langage. De plus, ces outils requièrent l'installation de packages tiers comme *Java* et *Python* ; une tâche qui peut se révéler pénible sous certains OS…

CMake permet d'automatiser la construction de projet. Pour cela, il va créer des *Makefiles*, des projets *Visual Studio*, *Xcode* ou *Eclipse* (et bien d'autres). Il s'agit en fait d'une sorte de « meta-*make* ». Il peut être lancé sur une grande variété de plateforme : *Windows*, *Mac OS X*, les *Unix-like*, *IRIX*, *AIX*,… Il est aussi capable de définir les compilateurs disponibles sur la plateforme et connaît la plupart d'entre eux (*gcc*, *Intel C*, *Borland*, *Visual Studio*, *Sun CC*,…). Pour créer un fichier projet, *CMake* va utiliser un générateur.

La configuration d'un projet à l'aide de *CMake* se fait dans un fichier texte *CMakeLists.txt*. Il peut y avoir plusieurs de ces fichiers pour décrire un projet.

CMake utilise une syntaxe qui lui est propre. Celle-ci est exclusivement constituée de commandes. Ces commandes peuvent prendre plusieurs paramètres.

Ci-après, un exemple de *CMakeLists.txt* simpliste avec des appels à quelques commandes fondamentales :

Listing 1 : exemple de CMakeLists.txt

```
cmake_minimum_required(VERSION 2.6)

project(myProj)

set(EXECUTABLE_OUTPUT_PATH test/bin/)

add_executable(
    myExe
    src/main.cpp
)
```

Dans ce fichier, on peut voir l'appel à quatre commandes : *cmake_minimum_required* qui définit la version minimale de *CMake* à utiliser (au moment où ce rapport est rédigé, la version est la *2.8*), *project* qui sert à nommer le projet, *set* qui permet d'affecter des valeurs aux variables et *add_executable* qui ajoute un exécutable.

Pour configurer ce projet, il faut lancer le programme *cmake* en lui indiquant dans quel répertoire est situé le fichier *CMakeLists.txt*. Par exemple :

```
$ cmake ..
```

Voici la sortie générée par *CMake* sur un *Mac OS X* :

Listing 2 : Sortie console cmake

```
-- The C compiler identification is GNU 4.2.1
-- The CXX compiler identification is Clang 4.0.0
-- Checking whether C compiler has -isysroot
-- Checking whether C compiler has -isysroot - yes
-- Checking whether C compiler supports OSX deployment target flag
-- Checking whether C compiler supports OSX deployment target flag - yes
-- Check for working C compiler: /usr/bin/gcc
-- Check for working C compiler: /usr/bin/gcc -- works
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Check for working CXX compiler: /usr/bin/c++
-- Check for working CXX compiler: /usr/bin/c++ -- works
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Configuring done
-- Generating done
-- Build files have been written to:
/Users/rovalette/Documents/work/Project_ZZ2/build
```

Ici, le projet a été généré pour être construit avec *make*. Il suffit donc de lancer la commande *make* pour compiler notre programme. On voit apparaître les contrôles réalisés durant l'inspection du système par *cmake*. Par exemple, le compilateur C trouvé sur la machine est *gcc*. Le compilateur C++ est *clang*. On peut tout de même modifier ces compilateurs en l'indiquant en paramètre à *cmake* :

```
$ cmake .. -DCMAKE_CXX_COMPILER=g++
```

De cette manière, *cmake* inspectera le système à la recherche du compilateur *g++*. Nous avons utilisé cette méthode pour vérifier que les résultats étaient corrects sur les compilateurs les plus en vogue. Il est aussi possible de spécifier le générateur à utiliser avec l'option *–G*.

Voici un schéma résumant le fonctionnement de *CMake* :

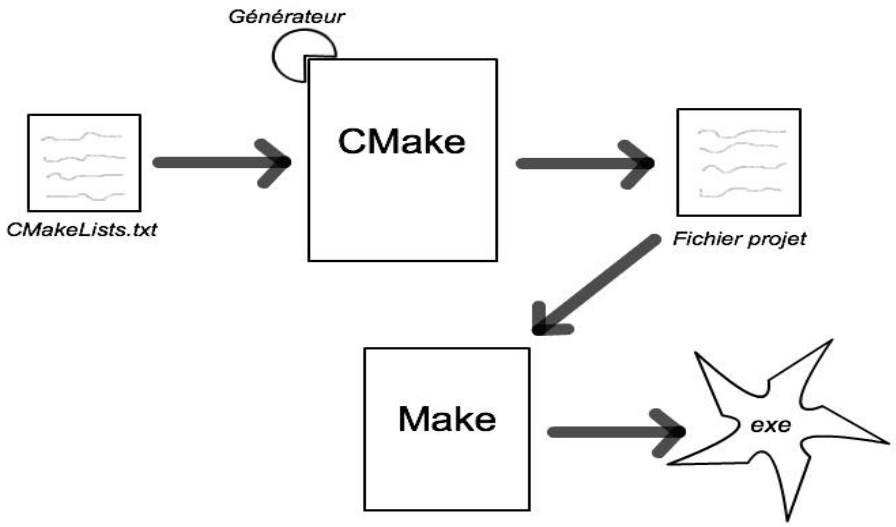


Figure 2 : Schéma de fonctionnement de CMake

1.2 CTest

CMake est une suite d'outils comprenant *CTest* et *CPack*. *CTest* est le composant permettant d'automatiser les tests logiciels. Il s'interface facilement avec *CDash*, un outil d'intégration continue. Quant à *CPack*, c'est un dispositif de création de programme d'installation. Nous nous intéressons ici à *CTest*.

Quand la taille d'un projet devient plus importante, il peut être intéressant d'utiliser des tests unitaires pour s'assurer que toutes les parties du logiciel répondent correctement aux attentes. Il y a aussi les tests de non régression qui viennent s'ajouter à ces tests pour vérifier qu'une nouvelle fonctionnalité (dans le logiciel ou même dans l'environnement) n'engendre pas un résultat imprévu dans une partie non modifiée du programme. Les tests d'intégration sont quant à eux utiles lorsqu'on a beaucoup de développeurs sur un même logiciel. Viennent ensuite les tests de recette qui vont valider l'analyse métier. Cet ensemble de tests a évolué vers ce qu'on appelle l'intégration continue et a initié une pratique plus propre et efficace des méthodes agiles. Avec ces dernières, il devient intéressant d'adopter une technique de développement piloté par les tests (*TDD*).

CMake et *CTest* sont tout à fait adaptés à ces pratiques, notamment grâce à l'outil *CDash*. Il s'agit d'un tableau de bord (*dashboard*) qui permet de suivre les résultats de

compilation et des tests d'un projet. L'exemple de dashboard suivant est une partie des résultats du projet *CMake* lui-même (*http://open.cdash.org/index.php?project=CMake*).

Nightly

Site	Build Name	Update	Configure		Build		Test			Build Time
		Files	Error	Warn	Error	Warn	Not Run	Fail	Pass	
TheGibson.kitware	CentOS5-i686-pgi113	1	0	0	0	11	0	16	261	3 hours ago
TheGibson.kitware	CentOS5-x86_64-pathscale4010	1	0	0	0	0	0	2	278	3 hours ago
TheGibson.kitware	CentOS5-x86_64-pgi113	1	0	0	0	14	0	1	276	3 hours ago
murron.hobbs-hancock	Fedora-17-x86_64	1	0	0	0	1	0	1	278	5 hours ago
bubbles.hooperlab	Fedora-17-x86_64	1	0	0	0	1	0	1	284	5 hours ago
voyager.sf-tec.de	Linux-Gentoo-HPPA32-4.6.3	1	0	0	0	1	0	0	280	7 hours ago
uran.tmux.org	OpenBSD-5.3-gcc-4.2.1	1	0	0	0	0	0	0	269	7 hours ago
stillaguamish.tecplot	SunOS5.9-CC	1	0	0	0	0	0	0	274	8 hours ago

Coverage

Site	Build Name	Percentage	LOC Tested	LOC Untested	Date
dash23.kitware	Linux-g++4.1	75.37%	35082	11465	8 hours ago
dash5ubuntu.kitware	Linux-g++4.6.1	74.77%	34203	11540	6 hours ago
hythloth.kitware	Linux64-bullseye-cov	84.58%	3720	678	7 hours ago
dash22.kitware	Win32-vs9-Release-Coverage	72.56%	3704	1401	8 hours ago

Dynamic Analysis

Site	Build Name	Checker	Defect Count	Date
dash23.kitware	Linux-g++4.1	Valgrind	0	8 hours ago
FarAway.kitware	Linux-valgrind2	Valgrind	0	5 hours ago

Figure 3 : Exemple du dashboard CMake

L'intérêt d'un tel outil est qu'il permet de voir facilement les erreurs et warnings levés en fonction de la plateforme de compilation. On a aussi accès à des tests de couverture (par exemple *gcov*) ainsi qu'à des données de profilage (par exemple *valgrind*).

CTest permet de créer, à partir de plusieurs exécutables, une suite de tests. Ces tests sont ensuite lancés à l'aide de la commande *ctest*.

Les tests sont créés dans le *CMakeLists.txt*. Il faut utiliser la commande *add_test*. Voici un exemple d'utilisation :

Listing 3 : exemple d'utilisation de add_test

```
cmake_minimum_required(VERSION 2.6)

project(projTest)

add_executable(
	testExe
	test/src/mainTest.cpp
)

enable_testing()

add_test(
	myTest
	testExe
)
```

La commande *enable_testing* (sans paramètre) est obligatoire lorsqu'on veut utiliser *ctest*. Ensuite, la commande *add_test* prend comme paramètres le nom du test et l'exécutable associé. Les tests ne sont pas obligatoirement des exécutables compilés. Il peut s'agir d'une cible définie par *CMake*, comme une tâche générant la documentation avec *Doxygen*.

Un exemple de sortie de *ctest* :

Listing 4 : Sortie console ctest

```
Test project /Users/rovalette/Documents/work/Project_ZZ2/build
    Start 1: FirstTest
1/2 Test #1: FirstTest .......................   Passed    0.07 sec
    Start 2: SecondTest
2/2 Test #2: SecondTest ......................   Passed    0.02 sec

100% tests passed, 0 tests failed out of 2

Total Test time (real) =   0.10 sec
```

Si jamais les exécutables des tests écrivent sur la sortie standard de la console, il faut lancer *ctest* avec l'option *–V* comme suit :

```
$ ctest –V
```

2 Notre Solution

2.1 L'exemple créé

Nous devions, avant de pouvoir créer le script, écrire des exemples qui permettraient de développer une solution. Ces courts exemples devaient être à la fois simples, et bien choisis. Nous avons décidé de faire comme si nous testions, de manière très basique, l'API C++, en faisant des erreurs typiques. Nous devions avoir quatre types de programme d'exemple. Un ne faisant pas d'erreur, un faisant une erreur inattendue, un faisant ce que l'on attend c'est-à-dire levant une erreur attendue, et un dernier levant une erreur attendue et une erreur inattendue.

C'est ainsi que nous avons créé quatre exemples appelés *main1, main2, main3* et *main4.*

Voici ces exemples :

Listing 5 : main1.cpp

```
1 | #include <iostream>
2 |
3 | int main() {
4 |
5 |    return 1
6 | }
```

Ce programme lève une erreur, car le point-virgule a été oublié à la ligne 5. Il fera partie des erreurs attendues, levées par l'API lors de la compilation.

Listing 6 : main2.cpp

```
1 | #include <headernotfound>
2 |
3 | int main()
4 | {
5 |     return 0;
6 | }
```

Ce programme lève une erreur, car le header, à la ligne 1 n'existe pas. Cette erreur doit être levée par l'API et fera donc partie des erreurs attendues. Voici donc les deux programmes levant des erreurs attendues.

Listing 7 : main3.cpp

```
1 | #include <iostream>
2 |
3 | int main()
4 | {
5 |     std::cout << "hello world !" << std::endl;
6 |     return 0;
7 | }
```

Ce programme ne génère aucune erreur et doit donc générer un test qui réussit. Le fait qu'il ne permette pas de tester l'API n'est pas grave selon nous, le programme, n'ayant pas d'erreur, doit être validé par l'API, et donc générer un test qui passe.

Listing 8 : main4.cpp

```
1 | #include <list>
2 |
3 | int main()
4 | {
5 |     std::list<int> i = a;
6 |
7 |     for (auto& j : i);
8 |
9 | }
```

Ce programme génère deux erreurs. Une première car '*a*' n'est pas déclaré ligne 5. Cette erreur est connue et doit être levée par l'API. Il y a une seconde erreur, ligne 7 : il est impossible de créer des boucles *for* basées sur des intervalles (*range-based for loop*) en C++98 ainsi que l'utilisation du mot-clé *auto*. Cette erreur ne sera pas reconnue, car non placée dans le dictionnaire des erreurs. De plus, cette erreur n'est pas toujours levée, cela dépend du compilateur. Par exemple, *clang++* (versions supérieures à 4.0) ne relèvera que la première erreur et créera deux *warnings* pour la boucle alors que *g++* lancera beaucoup plus d'erreurs et notamment les versions inférieures ou égales à *4.2* (car le support C++0x ou C++11 est arrivé avec la *4.3*). Et ces erreurs sont parfois difficilement exploitables car le compilateur y voit des erreurs de syntaxe plutôt que des erreurs de norme.

2.2 Le script CMake

Nous avons réalisé un script CMake permettant de compiler nos programmes dans un premier temps, puis de générer des tests adaptés dans un second temps que nous verrons dans la sous-partie suivante. Nous allons donc vous présenter la partie CMake de notre solution.

Nous avons créé un module comprenant les différentes macros, cela permet d'appeler notre solution grâce à un *include*, ce qui évite de copier-coller le code dans un fichier CMake existant.

Comme nous l'avons vu dans la première partie, CMake fonctionne avec un fichier *CMakeList.txt*, c'est dans ce fichier qu'il faudra inclure notre module.

La solution que nous avons créée est composée d'un module principal, *CompileTest.cmake* et d'un module secondaire *errors_dico.cmake*. Nous allons commencer par vous présenter le module secondaire.

Ce module est très simple, et sert à la définition des erreurs. Souhaitant une solution portable, nous avons testé notre programme sur différentes plateformes, et nous nous sommes rendus compte que selon le compilateur, les erreurs peuvent varier, et cela a pour conséquence que les erreurs attendues sont différentes. C'est pourquoi nous avons opté

pour un fichier à part contenant les différentes erreurs des différents compilateurs. Ce fichier sera à enrichir avec les erreurs qu'on souhaite tester. Voici à quoi ressemble notre dictionnaire d'erreurs :

Listing 9 : errors_dico.cmake

```
1 | set(NO_ERROR "::_N_O__E_R_R_O_R_::")
2 | set(ANY_ERROR "@_@")
3 | set(FILE_NOT_FOUND "file not found@No such file or directory")
4 | set(UNDECLARED_IDENTIFIER "use of undeclared identifier@was not
declared in this scope")
5 | set(EXPECTED_SEMICOLON "expected ';'@expected ';'")
6 | set(EXPECTED_PRIMARY_EXPRESSION "expected primary-expression")
```

Ce fichier est vraiment très succin et est ici défini pour tester notre solution, il devra être rempli plus tard pour être plus complet et définir plus d'erreurs, car nous n'avons mis que les erreurs correspondant à nos exemples.

Une erreur est définit avec la forme suivante :

```
set(NomErreur "erreur compilateur1@erreur compilateur2@...")
```

Avec cette syntaxe, nous pouvons définir des erreurs multi-plateforme, seule la première utilisation sur un compilateur nécessite un passage par ce fichier. Les erreurs *NO_ERROR* et *ANY_ERROR* sont à garder. En effet, *NO_ERROR* permet de compiler le code sans erreur, et *ANY_ERROR* avec n'importe quelle erreur.

Nous allons maintenant vous présenter le module principal, *CompileTest.cmake*. Ce module est composé de deux macros que nous allons détailler. Avant ces modules, nous avons inséré une ligne indiquant simplement que nous allons générer des tests. Cette ligne est `enable_testing()`, suivie de l'inclusion du dictionnaire. A ce moment, le script connait les erreurs grâce à l'inclusion. Nous avons ensuite une macro permettant d'essayer la compilation des fichiers sources.

Voici le code de cette macro :

Listing 10 : macro compile_test

```
1  | macro(compile_test _src_file _result _error_log)
2  |
3  |     try_compile(
4  |         result
5  |         ${CMAKE_CURRENT_BINARY_DIR}/bin
6  |         ${_src_file}
7  |         OUTPUT_VARIABLE error_log
8  |     )
9  |
10 |     set(${_result} ${result})
11 |
12 |     if(NOT result)
13 |         set(${_error_log} ${error_log})
14 |     endif()
15 |
16 | endmacro(compile_test)
```

Cette macro s'appelle *compile_test* et prend en paramètres le fichier source à compiler, ainsi que deux variables vide, qui contiendront le résultat de la compilation (c'est-à-dire si le programme a compilé ou non), ainsi que le log, c'est-à-dire les éventuelles erreurs que peut avoir généré ce programme. Pour compiler ce programme, nous nous appuyons sur la fonction *try_compile* proposée par CMake visant à effectuer une compilation au sein de CMake. Cette fonction s'utilise avec les paramètres suivants : premièrement, la variable de retour, c'est elle qui contiendra *TRUE* si la compilation a réussie et *FALSE* sinon. Il prend en deuxième paramètre le dossier où les fichiers binaires générés par la compilation devront être stockés. Le troisième paramètre est le fichier source qu'on veut compiler, et le dernier paramètre est la variable dans laquelle sera écrit le log, c'est-à-dire la sortie de compilation contenant les erreurs.

A cette étape, notre programme est compilé (ou a échoué), il nous faut donc générer un test par programme, qui réponde à la problématique de validation d'API.

Pour cela, nous avons créé une autre macro CMake appelée *create_compile_test* qui appellera la précédente macro et générera un test par exécutable suivant le modèle C++ donné en paramètre.

Listing 11 : macro add_compile_test

```
1  | macro(add_compile_test _src_file _test_name _expected_result
_expected_error)
2  |
3  |       ############### ARG CONTROL ####################
4  |
5  |       # check if the expected error is defined
6  |       if (NOT ${_expected_error})
7  |           message(FATAL_ERROR "${_expected_error} is not defined")
8  |       endif()
9  |       set(__expected_error ${_expected_error})
10|       set(_expected_error ${_expected_error})
11|       set(expected_error ${${_expected_error }})
12|       set(_expected_result ${_expected_result})
13|
14|       # arguments control
15|       if(_expected_result MATCHES "^EXPECT$")
16|
17|           if(_expected_error MATCHES NO_ERROR)
18|               set(__expected_result true)
19|               set(_allow_unexpected false)
20|           else()
21|               set(__expected_result false)
22|               set(_allow_unexpected true)
23|           endif()
24|
25|       elseif(_expected_result MATCHES "^EXPECT_ONLY$")
26|
27|           if(_expected_error MATCHES ANY_ERROR)
28|               message(FATAL_ERROR "EXPECT_ONLY argument cannot be used
with ANY_ERROR argument")
29|           elseif(_expected_error MATCHES NO_ERROR)
30|               set(__expected_result true)
31|               set(_allow_unexpected false)
```

```
32|         else()
33|             set(__expected_result false)
34|             set(_allow_unexpected false)
35|         endif()
36|
37|     else()
38|         message(FATAL_ERROR "invalid argument ${_expected_result} in
add_compile_test")
39|     endif()
40|
41|     ######################## END ARG CONTROL ############################
42|
43|     # launch the try_compile task
44|     compile_test(${_src_file} result log)
45|     string(TOLOWER ${result} result)
46|
47|     if(${result} MATCHES true)
48|         if(${__expected_result} MATCHES true)
49|             set(_no_error_text "TEST SUCCEEDED")
50|         else()
51|             set(_no_error_text "TEST FAILED (${_expected_error} error
was expected)")
52|         endif()
53|     endif()
54|
55|     # put the log into a file
56|     set(_error_file
${CMAKE_CURRENT_BINARY_DIR}/test/data/${_test_name}_errors.txt)
57|     configure_file(
58|         modules/template_errors.txt.in
59|         ${_error_file}
60|     )
61|
62|     # configure a cpp file with the result of try_compile
63|     configure_file(
64|         modules/template_test.cpp.in
65|         ${CMAKE_CURRENT_BINARY_DIR}/test/src/${_test_name}.cpp
66|     )
67|
68|     # create an executable for the test
69|     add_executable(
70|         ${_test_name}
71|         ${CMAKE_CURRENT_BINARY_DIR}/test/src/${_test_name}.cpp
72|     )
73|
74|     # add the executable to the ctest suite
75|     add_test(
76|         ${_test_name}
77|         test/bin/${_test_name}
78|     )
79| endmacro(add_compile_test)
```

Cette macro prend en paramètres les fichiers sources et le nom que portera le test créé, puis un des mots clés *EXPECT* ou *EXPECT_ONLY* suivi du nom de l'erreur que l'on veut catcher. C'est cette macro qu'il faut appeler, l'autre sera appelée automatiquement.

En ce qui concerne le fonctionnement de cette macro, elle procède à un contrôle d'argument, c'est-à-dire qu'elle vérifie s'il y a bien le mot clé *EXPECT* ou *EXPECT_ONLY* et

donne des valeurs à des variables en conséquence, par exemple si plusieurs erreurs sont tolérées, si le résultat attendu est la compilation ou la non compilation etc. Si un mauvais mot clé est donné en paramètre, le programme quitte avec une erreur fatale.

Une fois cette vérification effectuée, cette macro appelle la macro précédente, pour compiler le code et obtenir le log contenant les erreurs de compilation. Ensuite, elle affecte une variable locale qui contient le nom du test. Après cela, nous regardons le résultat renvoyé par la macro *compile_test* qui nous informe si le code a bien compilé ou non. S'il a bien compilé, on écrit le log dans un fichier qui sera lu par un code C++.

Nous procédons ensuite à un bref affichage, puis nous configurons la variable *_error_file* comme un fichier sur le disque ayant pour nom le nom du test suivi de *errors*. Puis nous inscrivons le log dans ce fichier, et nous créons le test grâce à un code C++ qui sera détaillé dans la sous-partie suivante. Une fois cela fait, nous le rendons exécutable, et ajoutons ce test à *CTest*.

2.3 La génération de test

Nous allons maintenant expliquer le fonctionnement du code C++ des tests. Ce code est appelé, comme dit précédemment, dans la macro *compile_test*. Il est générique et s'adapte grâce aux variables *CMake* affectées dans la macro.

Listing 12 : template_test.cpp.in

```
1  | #include <iostream>
2  | #include <sstream>
3  | #include <fstream>
4  | #include <list>
5  |
6  | int main(int argc, char ** argv)
7  | {
8  |     int return_value = @__expected_result@?1:0;
9  |
10 |     if (! @result@)
11 |     {
12 |         std::ifstream errorFile("@_error_file@");
13 |
14 |         std::istringstream expected("@expected_error@");
15 |         std::list<std::string> listError;
16 |
17 |         std::string str;
18 |
19 |         while (std::getline(expected, str, '@'))
20 |         {
21 |             listError.push_front(str);
22 |         }
23 |
24 |         std::list<std::string>::iterator it;
25 |
26 |         while (std::getline(errorFile, str))
27 |         {
28 |             if (str.find("error:") != std::string::npos /*||
str.find("warning:") != std::string::npos*/)
```

```
29|             {
30|                 it = listError.begin();
31|
32|                 //std::cout << str << std::endl;
33|
34|                 while(it != listError.end() && str.find(*it) ==
std::string::npos)
35|                 {
36|                     ++it;
37|                 }
38|
39|                 // if not expected error
40|                 if (it == listError.end())
41|                 {
42|                     std::cout << "UNEXPECTED ERROR FOUND :" <<
std::endl;
43|                     if (! @_allow_unexpected@)
44|                     {
45|                         return_value = 1;
46|                     }
47|                 }
48|                 else
49|                 {
50|                     std::cout << "*** EXPECTED ERROR FOUND
(@__expected_error@) :" << std::endl;
51|                 }
52|
53|                 int idx = str.find("error:");
54|                 std::string errorText = str.substr(idx);
55|                 std::cout << '\t' << str.erase(idx) << std::endl;
56|                 std::cout << '\t' << errorText << std::endl
57|             }
58|         }
59|
60|         if ( (! @_allow_unexpected@) && (return_value == 1))
61|         {
62|             std::cout << std::endl << "*** TEST FAILED : unexpected
errors were not allowed" << std::endl << std::endl;
63|         }
64|
65|         return return_value;
66|     }
67|
68|     std::cout << "*** NO ERROR FOUND : @_no_error_text@" << std::endl;
69|
70|     return !return_value;
71| }
```

Lors de l'appel à *configure_file* dans la macro *add_compile_test* présentée précédemment, *CMake* remplace les valeurs entre '@' par leurs valeurs dans le contexte d'exécution. Ce programme est utilisé pour la création des tests, et a accès aux variables de *CMake.* Si le résultat de compilation est *TRUE*, le programme ne rentre pas dans la conditionnelle *if* et renvoie directement le résultat. Sinon, on ouvre le fichier de log et on met les erreurs attendues dans une liste. Il y a plusieurs erreurs car notre programme cherche les erreurs de plusieurs compilateurs. Ensuite on essaye de trouver l'erreur attendue dans le log. Si on la trouve, c'est bon. Sinon on change la valeur de retour pour

l'adapter. Cette valeur est adaptée selon si on accepte d'autres types d'erreur que celle attendue, seulement l'erreur donnée en paramètre, ou aucune erreur. Ce code permet, grâce aux valeurs de CMake d'être adaptable en fonction de ce qui est donné en paramètre dans le *CMakeLists.txt*. Lors de l'exécution des tests, il est judicieux d'utiliser *CTest* avec l'option *–V* pour qu'il nous donne bien toutes les informations, sinon, nous savons juste si le test est passé ou non mais pas pourquoi ni ce qui a été inscrit dans le log. En effet, nous réalisons un affichage succin permettant de savoir quelle erreur a été trouvée, dans quel cadre, et la valeur de retour du test.

3 Utiliser notre solution

3.1 Appeler notre module

Pour se servir de notre module, il faut formuler un appel dans un *CMakeLists.txt*. Au préalable, il faut avoir entré les erreurs qu'on souhaitera voir apparaitre, pour que notre programme les connaisse. Pour cela, il faut les entrer dans le fichier *errors_dico.cmake*. La syntaxe d'écriture a été explicitée plus tôt, il s'agit d'affecter une variable dont on choisit le nom, et de rentrer, entre guillemets le texte de l'erreur, avec les différentes variantes selon le compilateur utilisé. Par exemple, lors de l'utilisation d'un header inconnu, dans le test *main2.cpp, clang* renvoie *« file not found »* alors que g++ renvoie l'erreur *« No such file or directory »*. C'est pour cela que nous avons fait le choix d'utiliser plusieurs types d'erreur, mais lors de l'appel, cette erreur sera identifiée par *« FILE_NOT_FOUND »* dans *CMake*. Cette variable factorisera l'ensemble des variantes que pourront lancées les compilateurs.

Dans le fichier *CMake* existant ou à créer, il suffit de faire un *include* de notre module comme suit : `include(modules/CompileTest.cmake)`. Grâce à cette commande, nous pouvons utiliser notre macro principale. Voici l'exemple de *CMakeLists.txt* que nous avons utilisé :

Listing 13 : CMakeLists.txt pour notre exemple

```
1 | cmake_minimum_required(VERSION 2.6)
2 |
3 | project(myProj)
4 |
5 | set(EXECUTABLE_OUTPUT_PATH test/bin/)
6 |
7 | include(modules/CompileTest.cmake)
8 |
9 | add_compile_test(${CMAKE_CURRENT_SOURCE_DIR}/src/main1.cpp
"test_main1_compile_pas" EXPECT_ONLY EXPECTED_SEMICOLON)
10| add_compile_test(${CMAKE_CURRENT_SOURCE_DIR}/src/main2.cpp
"test_main2_compile_pas" EXPECT FILE_NOT_FOUND)
11| add_compile_test(${CMAKE_CURRENT_SOURCE_DIR}/src/main3.cpp
"test_main3_compile" EXPECT NO_ERROR)
12| add_compile_test(${CMAKE_CURRENT_SOURCE_DIR}/src/main4.cpp
"test_main4_compile_pas" EXPECT_ONLY UNDECLARED_IDENTIFIER)
```

Nous pouvons donc voir que l'appel à la macro se fait grâce à son nom, *add_compile_test* avec en paramètre le fichier source à compiler, le nom du test, un des deux mots clés (*EXPECT* ou *EXPECT_ONLY*) suivi de l'erreur attendue.

Si on veut que le test ne réussisse que dans le cas où il n'y ait eu qu'une seule erreur connue levée, nous pouvons utiliser *EXPECT_ONLY* avec le nom de l'erreur attendue. Comme cela, notre macro cherche dans le log une occurrence de l'erreur attendue. Si une autre erreur est présente, le test échoue, sinon, il passe.

Dans le cas où on veut que le programme passé en paramètre compile sans erreur, il suffit d'appeler la macro avec *EXPECT NO_ERROR* ou *EXPECT_ONLY NO_ERROR.* Comme ça, la macro va vérifier que la compilation a lieu correctement, sans aucune erreur.

Dans le cas où on souhaiterait absolument, malgré les erreurs, que le test réussisse, il faut le lancer avec *EXPECT ANY_ERROR,* la macro va alors générer un test qui réussit s'il y a des erreurs. La combinaison des mots clés *EXPECT_ONLY* et *ANY_ERROR* est interdite.

Une fois la rédaction du *CMakeLists.txt* effectuée, il faut lancer CMake dessus. Pour cela, il faut se placer à l'endroit où est situé le *CMakeLists.txt* et créer un dossier *build* soit dans un terminal grâce à la commande mkdir sous Unix, Linux, Mac et Windows, soit directement via l'interface de l'explorateur de fichier du système. Une fois cette opération effectuée, il faut ouvrir un terminal (cmd, powershell sous Windows) et se placer dans le dossier *build* grâce à la commande *cd* ou *dir*, puis lancer CMake avec le type de Makefile compatible avec le compilateur du système. Pour notre exemple, sous windows, il faut lancer CMake avec la commande :

```
$ cmake -G "MinGW Makefiles" ..
```

En effet, notre compilateur est fourni par *MinGW*, c'est donc un Makefile *MinGW* qu'il faut produire. En cas de doute, on peut laisser faire *CMake* qui trouvera le premier compilateur disponible. Les « .. » sont très important, en effet, ils permettent d'informer *CMake* que le *CMakeFiles.txt* se situe dans le dossier parent.

De même sous Linux et Mac nous pouvons utiliser :

```
$ cmake -G "Unix Makefiles" ..
```

Une fois *CMake* lancé, si la configuration s'est bien déroulée, nous pouvons lancer *make*, qui exécutera le *Makefile* créé. Pour lancer *make,* cela dépend encore du système, sous Windows, il faut utiliser *mingw32-make* pour appeler *make,* sur les systèmes Unix-like, *make* existe.

Nous sommes maintenant prêt à lancer les tests avec *ctest*, c'est ce que nous allons voir dans la partie suivante.

3.2 Les résultats

Maintenant que les tests sont compilés, nous pouvons les lancer et étudier les résultats. Nous lancerons les tests avec la commande suivante :

```
$ ctest -V
```

Nous utiliserons les fichiers *main* définis précédemment.

- Cas n°1 : on attend une erreur bien particulière. Utilisons le fichier source *main1.cpp* et le test suivant :

```
add_compile_test(${CMAKE_CURRENT_SOURCE_DIR}/src/main1.cpp
"test_main1_compile_pas" EXPECT_ONLY EXPECTED_SEMICOLON)
```

Voici la sortie (abrégée) :

Listing 14 : Sortie cas n°1

```
1: Test command: test/bin/test_main1_compile_pas
1: Test timeout computed to be: 9.99988e+06
1: *** EXPECTED ERROR FOUND (EXPECTED_SEMICOLON) :
1:    /Users/rovalette/Documents/work/Project_ZZ2/src/main1.cpp:5:10:
1:    error: expected ';' after return statement
1/4 Test #1: test_main1_compile_pas ...........   Passed    0.02 sec
```

L'erreur attendue (oubli d'un point-virgule) a bien été rencontrée. Aucune autre erreur n'a été levée. Le test réussit comme espéré.

- Cas n°2 : on souhaite que la compilation se déroule sans erreur. On utilise le fichier *main3.cpp* et le test suivant :

```
add_compile_test(${CMAKE_CURRENT_SOURCE_DIR}/src/main3.cpp
"test_main3_compile" EXPECT NO_ERROR)
```

Le résultat :

Listing 15 : Sortie cas n°2

```
3: Test command: test/bin/test_main3_compile
3: Test timeout computed to be: 9.99988e+06
3: *** NO ERROR FOUND : TEST SUCCEEDED
3/4 Test #3: test_main3_compile ...............   Passed    0.00 sec
```

Ici, le code ne contenait pas d'erreur (*hello world* classique). Le test passe donc sans problème en indiquant qu'aucune erreur n'a été trouvée.

- Cas n°3 : cette fois, on attendra une seule erreur mais d'autre seront levées (erreurs inattendues). Ce test se déroule avec le fichier *main4.cpp* et le test ci-dessous :

```
add_compile_test(${CMAKE_CURRENT_SOURCE_DIR}/src/main4.cpp
"test_main4_compile_pas" EXPECT_ONLY UNDECLARED_IDENTIFIER)
```

Normalement, ce test doit échouer car d'autres erreurs vont être levées et on utilise le mot-clef *EXPECT_ONLY*. Mais ici, le résultat va dépendre du compilateur. Nous utilisons ici *g++ 4.2* (qui est un compilateur encore *trop* répandu) :

Listing 16 : Sortie cas n°3

```
4: Test command: test/bin/test_main4_compile_pas
4: Test timeout computed to be: 9.99988e+06
4: *** EXPECTED ERROR FOUND (UNDECLARED_IDENTIFIER) :
4:    /Users/rovalette/Documents/work/Project_ZZ2/src/main4.cpp:5:
4:    error: ‘a’ was not declared in this scope
4: UNEXPECTED ERROR FOUND :
4:    /Users/rovalette/Documents/work/Project_ZZ2/src/main4.cpp:9:
4:    error: a function-definition is not allowed here before ‘:’ token
4: UNEXPECTED ERROR FOUND :
4:    /Users/rovalette/Documents/work/Project_ZZ2/src/main4.cpp:11:
4:    error: expected primary-expression before ‘}’ token
4: UNEXPECTED ERROR FOUND :
4:    /Users/rovalette/Documents/work/Project_ZZ2/src/main4.cpp:11:
4:    error: expected `;' before ‘}’ token
4: UNEXPECTED ERROR FOUND :
4:    /Users/rovalette/Documents/work/Project_ZZ2/src/main4.cpp:11:
4:    error: expected primary-expression before ‘}’ token
4: UNEXPECTED ERROR FOUND :
4:    /Users/rovalette/Documents/work/Project_ZZ2/src/main4.cpp:11:
4:    error: expected `)' before ‘}’ token
4: UNEXPECTED ERROR FOUND :
4:    /Users/rovalette/Documents/work/Project_ZZ2/src/main4.cpp:11:
4:    error: expected primary-expression before ‘}’ token
4: UNEXPECTED ERROR FOUND :
4:    /Users/rovalette/Documents/work/Project_ZZ2/src/main4.cpp:11:
4:    error: expected `;' before ‘}’ token
4:
4: *** TEST FAILED : unexpected errors were not allowed
4:
4/4 Test #4: test_main4_compile_pas ...........***Failed    0.00 sec
```

Le test a échoué, car même s'il a rencontré l'erreur attendue, des erreurs inattendues ont été trouvées. Ce test aurait réussi si on avait utilisé *EXPECT* au lieu de *EXPECT_ONLY*.

3.3 Mise en évidence des différences entre compilateurs

Un des principaux problèmes qui ait pu se poser est la différence qu'il existe entre les nombreux compilateurs et leurs différentes versions. Ne serait-ce même qu'en ne prenant qu'un seul compilateur, par exemple *g++*, toutes ses versions renverront des erreurs différentes pour un même code. Pour poursuivre sur le même exemple, nous avons compilé le fichier *main4.cpp* (présenté précédemment) avec trois compilateurs différents : *g++ 4.2*, *g++ 4.7* et *clang++ 4.0*. Voilà les résultats que nous avons obtenus :

Premièrement, avec *g++ 4.2* :

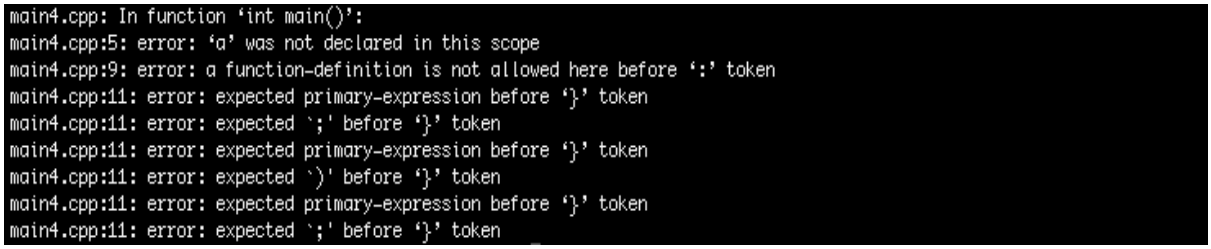


Figure 4 : Génération d'erreur avec g++ 4.2

On voit que dans cette version, *g++* repère huit erreurs. La première est justifiée car ‘*a*’ n’est pas déclaré. Les sept autres sont dues à la boucle *for range-based* (qui est une nouveauté de la norme C++11). On voit bien que ces erreurs ne sont pas explicites. En effet, la release officielle de ce compilateur date du 13 mai 2007, époque à laquelle la norme C++0x n’avait pas encore été unanimement adoptée. Elle le fut en mars 2008 avec l’arrivée de *g++ 4.3*. La norme C++11 arrive avec *g++ 4.7* en mars 2012 (sources : *http://gcc.gnu.org/releases.html*).

Voyons maintenant avec *g++ 4.7* :

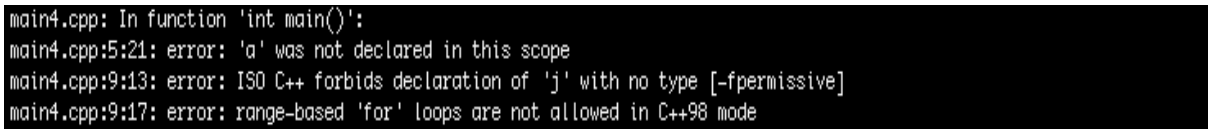


Figure 5 : Génération d'erreur avec g++ 4.7

Ici, seulement trois erreurs sont levées. La première ne diffère pas du compilateur précédent. Par contre, les deux erreurs suivantes sur l’utilisation de la norme C++11 sont un peu plus explicites. La première indique que ‘*j*’ ne peut pas être déclaré sans type. En fait, le compilateur ne reconnaît pas le mot-clé *auto*. Il explique ensuite correctement que la boucle *for* basée sur un intervalle n’est pas autorisée en C++98.

Finalement, avec *clang 4.0* :

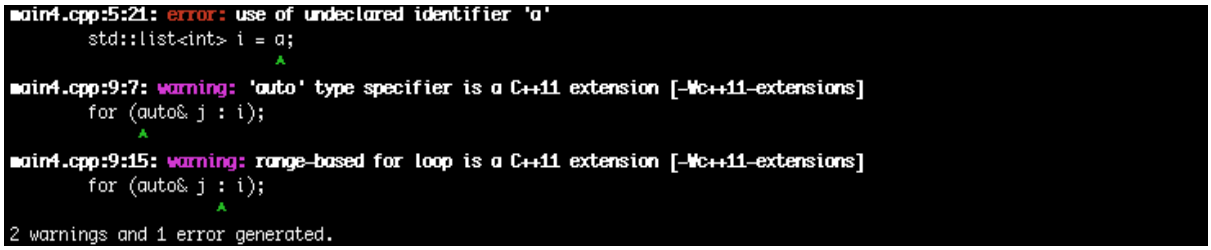


Figure 6 : Génération d'erreur avec clang++ 4.0

Comme *g++*, *clang* repère correctement la première erreur. La différence, mis à part la coloration et l’indication des erreurs, est qu’il accepterait de compiler le programme s’il n’y avait pas la première erreur. En effet, seuls des warnings sont remontés, et ils sont très explicites. Il faut noter que dans le cas n°3 des résultats précédemment présentés, le test serait passé.

Mais ici, la différence qui nous intéresse réellement, c'est surtout le message associé à la première erreur. *g++* retourne « *'a' was not declared in this scope* » alors que *clang* renvoie « *use of undeclared identifier 'a'* ». C'est cette différence qui est gênante, car à cause de celle-ci, il faut tenir à jour le fichier *errors_dico.cmake*.

Conclusion

Nous avons donc réalisé un script *CMake* permettant de tester une API en vérifiant que les opérations interdites par l'API le sont bien. Le script réalisé est portable, car il n'utilise que CMake qui est par définition portable (Cross-platform Make). Notre outil peut donc être utilisé tant sur Windows, que sur des systèmes Unix-like tels que Mac ou Linux. Nous avons fait le choix de ne traiter qu'une erreur attendue à la fois, car dans des tests unitaires, nous ne sommes censés tester qu'une seule fonctionnalité à la fois. Cependant, ce pourrait être intéressant de reconnaître plusieurs erreurs à la fois, pour vérifier par exemple qu'une erreur, lors de sa levée, entraîne bien d'autres erreurs également attendues.

Nous avons eu beaucoup de mal à cerner le sujet. En effet, nous n'avions pas à disposition de code à tester, et le fait de devoir le concevoir du début à la fin nous a retardés. Cependant, une fois ces exemples créés, nous avons pu nous lancer dans l'écriture d'un script CMake. Mais la documentation sur CMake sur internet étant très rare, il a été compliqué de trouver comment utiliser des fonctions spécifiques avec des paramètres complexes et peu courants. Grâce au livre Mastering CMake (Martin & Hoffman, 2010) [2], nous avons pu mieux comprendre le fonctionnement de CMake et réaliser ce projet.

Ce script CMake génère des tests pouvant sans problème être couplé à un tableau de bord comme CDash, pour réaliser de l'intégration continue. Le fait de traiter plusieurs erreurs pourrait être une perspective d'amélioration de ce projet, ainsi que le fait de nous attarder sur la gestion des warnings. En effet, l'API peut lever des warnings qui peuvent être très importants.

Références bibliographiques

[1] Amy Brown & GregWilson, « The Architecture of Open Source Applications Volume 1»

[2] Ken Martin & Bill Hoffman, « Mastering CMake », 2010, Kitwar Inc

[3] Nicolai M. Josuttis, « The C++ Standard Library, second edition », 2012, Addison-Wesley, Pearson Education

[4] Bjarne Stroustrup, « The C++ Programming Language, Special Edition », 2000, Addison-Wesley, AT&T

Webographie

[5] http://cmake.org/, (date de consultation 2013)

[6] http://public.kitware.com/pipermail/cmake-developers/, (date de consultation 2013)

[7] http://www.mail-archive.com/cmake@cmake.org/msg19538.html, 03 mars 2009 (date de consultation 2013)