ECOLE CENTRALE DE NANTES

SMP - TP10

L'objectif de ce TP est d'expérimenter les tests unitaires et d'avoir une première expérience avec **Google Test (GTest)**. Vous allez aussi utiliser Doxygen pour générer des fichiers de documentation.

1 INSTALLATION DE DOXYGEN

Doxygen est multiplateforme et disponible à l'adresse suivante `https://www.doxygen.nl/index.html`. Suivez les instructions d'installation selon votre système d'exploitation `https://www.doxygen.nl/manual/index.html`.
Remarque : sous mac il est possible de l'installer avec homebrew avec la commande :
`brew install doxygen`

Vous pourrez ensuite créer un fichier de configuration doxygen automatiquement avec la commande (en remplaçant `<config-file>` par le nom que votre fichier de configuration) :
`doxygen -g <config-file>`

puis pour générer la documentation d'un projet replit il suffira de taper la commande :
`doxygen <config-file>`

Vous pouvez aussi utiliser l'interface graphique si vous l'avez installée.
Tester avec des fichiers d'un tp précédent que vous aurez complétés avec des commentaires doxygen `https://www.doxygen.nl/manual/docblocks.html`.

2 INSTALLATION DE CMAKE

CMake est multiplateforme et disponible à l'adresse `https://cmake.org`. Téléchargez le binaire qui correspond à votre distribution à l'adresse suivante `https://cmake.org/download/`.

3 INSTALLATION DE GTEST POUR VOTRE PROJET

Dans votre projet commencez par cloner GTest :

```
git clone https://github.com/google/googletest.git
```

Remarque : Si vous votre système ne supporte pas c++14, cloner bonne branche v1.12.x qui fonctionne avec c++11
`git clone --branch v1.12.x https://github.com/google/googletest.git`

Nous allons compiler GoogleTest avec l'outil Cmake. Il faut le déclarer dans le fichier `replit.nix` en ajoutant la ligne :

Puis compilez google test avec CMake :
`cd googletest/`
`cmake -Dgtest_build_samples=ON -DBUILD_GMOCK=OFF` (pour compiler aussi des exemples) - pour compiler sans exemple `cmake -DBUILD_GMOCK=OFF`
Effectuez ensuite la compilation proprement dite :
`cmake --build .`
A la fin de la compilation, aller dans le sous-dossier `googletest`
`cd googletest`
et taper :
`make`
Cela crée (entre autres) les librairies statiques dans le dossier `GoogleTest/lib` et compile les exemples. Vous avez les commandes de compilation permettant de faire le lien à ces librairies dans le Makefile fourni (codeTp1).
Tester les fichiers exécutables exemples compilés dans `./googletest/googletest` (les sources sont dans `./googletest/googletest/sample`).

4 EXÉCUTION DE TESTS AVEC GOOGLE TEST

Le code fourni vous donne des premiers exemples de tests. Exécutez les.

5 PREMIER TEST AVEC GOOGLE TEST

Créez un fichier de test pour tester les fichiers de `codeTp2`.
Testez de nouveau la fonction factorielle à l'aide de
`TEST(TestCaseName , TestName)`

et des macros `ASSERT_EQ(expected,actual)` et `EXPECT_EQ(expected,actual)`.
Puis désactivez ce test en plaçant un
`DISABLED_`
dans l'une des entrées de TEST, i.e.,
`TEST(DISABLED_TestCaseName, TestName)`
ou
`TEST(TestCaseName, DISABLED_TestName)`
et observez les résultats.

6 Expérimenter Google Test et le Test Driven Development (TDD)

Le TDD est une méthode agile de code qui est guidée par les tests. Cette dernière est cyclique et comporte les phases suivantes :

1. Écrire un seul test qui décrit une partie du problème à résoudre
2. Vérifier que le test échoue, c'est-à-dire que le code se rapportant à ce test n'existe pas
3. Écrire juste assez de code pour que le test réussisse
4. Vérifier que le test ainsi que ceux pré-existants passent
5. Remanier le code, i.e., améliorer sans altérer le comportement, e.g., enlever les doublons

Tout lintérêt du TDD consiste à suivre le cycle précédent dans lordre. Donc, toute déviation par rapport à lordre des questions dans cette feuille réduit lintérêt de ce travail.

6.1 Test 1 - `EXPECT_DOUBLE`

Créez un test à l'aide de la macro
`TEST(TestCaseName , TestName)`
pour une fonction nommée `div` (dans la classe `Calculator`) qui retourne 2 (type *double*) lorsqu'elle prend entrée les valeurs 4 et 2 (type *double* également).
Attention les macros à utiliser pour tester les *float* et *double* sont différentes. Par exemple, il est ici nécessaire d'utiliser
`EXPECT_DOUBLE_EQ (expected, actual)`
Créez une fonction vide pour que le code puisse compiler et vérifiez que le test ne passe pas. L'objectif est de créer le code pas à pas, en utilisant à chaque étape une méthode simple, rapide, et faisant passer les tests existants. Écrivez donc la fonction la plus simple possible permettant de faire passer ce test.

6.2 Test 2 - `EXPECT_NEAR`

Pour le second test, réalisez une fonction `div` qui vérifie que le résultat de la fonction `div` lorsqu'elle prend en entrée 7 et 3 retourne 2.333 ± 0.001. La macro correspondante à utiliser est la suivante :

`EXPECT_NEAR (expected, actual, absolute_range)`
Vérifiez que le test ne passe pas.
Écrivez la fonction `div` (pour division, bien sûr) la plus simple possible permettant de vérifier le test ci-dessus.
Testez de nouveau la fonction "div" avec la macro
`EXPECT_NEAR`
pour savoir si le résultat est à 2.333 ± 0.0001 et observez le résultat du test.

6.3 Test 3 - ASSERT_EXIT

Pour le troisième et dernier test, utilisez
`ASSERT_EXIT(statement, predicate, expected_message)`
afin de tester si le programme quitte bien avec le prédicat
`::testing::ExitedWithCode(255)`
et le message
`Error: Division by 0 not possible`
lorsque la fonction `div` prend en entrée 7 et 0. Vérifiez que le test ne passe pas.
Modifiez votre fonction afin de réussir le test précédent.

6.4 Test 4 - A vous de jouer

Bien que le programme soit simple, d'autres fonctionnalités restent à tester. Identifiez celles primordiales au bon fonctionnement du programme. Modifiez le programme pour les rendre testables. Créez un code pour tester son fonctionnement. Enfin, modifier le programme afin qu'il passe ce test.