Google Test

Le framework **Google Test** est un outil puissant pour écrire et exécuter des tests pour le code C++.

Exemple MACRO **EXPECT_EQ:**

Considérons d'abord un exemple simple de test d'une fonction qui calcule la factorielle d'un nombre donné. Nous pouvons écrire un cas de test pour cette fonction en utilisant la macro **TEST** et la macro d'assertion **EXPECT_EQ**. La macro TEST prend deux arguments : le nom du cas de test et le nom de la fonction de test. La fonction de test doit appeler la fonction factorielle et utiliser la macro **EXPECT_EQ** pour vérifier que le résultat est égal à la sortie attendue. Par exemple:

```
TEST(FactorialTest, Zero) {
   int result = Factorial(0);
   EXPECT_EQ(1, result);
}
```

Dans cet exemple, nous testons la fonction factorielle avec une entrée de 0 et attendons une sortie de 1. En exécutant ce cas de test, nous pouvons nous assurer que la fonction factorielle fonctionne correctement pour cette entrée spécifique.
La sortie de la console pour le scénario de test FactorialTest qui a été écrit ci-dessus ressemblerait à ceci :

```
[==========] Running 1 test from 1 test case.
[----------] Global test environment set-up.
[----------] 1 test from FactorialTest
[ RUN      ] FactorialTest.Zero
[       OK ] FactorialTest.Zero (0 ms)
```

```
[----------] 1 test from FactorialTest (0 ms total)

[----------] Global test environment tear-down
[==========] 1 test from 1 test case ran. (0 ms total)
[  PASSED  ] 1 test.
```

La sortie commence par un résumé du nombre de tests en cours d'exécution et à partir de quels cas de test. Il indique ensuite que la configuration de l'environnement de test global a été effectuée. Ensuite, il affiche le nom du cas de test FactorialTest et sa fonction de test Zero et il est en cours d'exécution. Après cela, il indique que le test a réussi (indiqué par [OK]) et le temps qu'il a fallu pour exécuter le test en millisecondes. Ensuite, il affiche le cas de test et le résumé de la suite de tests avec le nombre de tests exécutés, le temps total qu'il a fallu et le nombre de tests réussis.
Il convient également de noter que si un cas de test échoue, la sortie indiquera quel cas de test a échoué et la raison pour laquelle il a échoué, par exemple, si la valeur attendue ne correspond pas à la valeur réelle.
La sortie de la console pour le cas de test FactorialTest qui a été écrit ci-dessus, si le cas de test échoue, ressemblerait à ce qui suit :

```
[==========] Running 1 test from 1 test case.
[----------] Global test environment set-up.
[----------] 1 test from FactorialTest
[ RUN      ] FactorialTest.Zero
[  FAILED  ] FactorialTest.Zero (0 ms)
          Value of: factorial(0)
          Expected: 1
          Actual  : 11
[----------] 1 test from FactorialTest (0 ms total)

[----------] Global test environment tear-down
[==========] 1 test from 1 test case ran. (0 ms total)
[  PASSED  ] 0 tests.
[  FAILED  ] 1 test, listed below:
[  FAILED  ] FactorialTest.Zero
```

Il indique que le test a échoué (indiqué par **[FAILED]**) et le temps qu'il a fallu pour exécuter le test en millisecondes. Ensuite, il affiche les valeurs attendues et réelles et la raison pour laquelle il a échoué.
Enfin, il affiche le cas de test et le résumé de la suite de tests avec le nombre de tests exécutés, le temps total qu'il a fallu, le nombre de tests réussis et le nombre de tests échoués. Il répertorie également les tests échoués pour plus de détails.
Il est important de noter qu'en cas d'échec d'un cas de test, il est crucial d'étudier la cause de l'échec et de corriger le problème avant de relancer le test.

Exemple MACRO **EXPECT_EQ, EXCEPT_THROW :**

Ensuite, considérons un exemple plus complexe de test d'une classe. Dans cet exemple, nous utiliserons une simple implémentation de pile comme classe à tester. Nous pouvons écrire des cas de test pour les méthodes de la classe en utilisant la macro **TEST** et les macros d'assertion **EXPECT_EQ**, **EXPECT_THROW** et **ASSERT_TRUE**. Par exemple:

```
TEST(StackTest, Push) {
   Stack s;
   s.push(1);
   s.push(2);
   EXPECT_EQ(2, s.size());
}

TEST(StackTest, Pop) {
   Stack s;
   s.push(1);
   s.push(2);
   EXPECT_EQ(2, s.pop());
   EXPECT_EQ(1, s.pop());
   EXPECT_THROW(s.pop(), std::out_of_range);
}
```

Dans ces exemples, nous testons les méthodes push et pop de la classe Stack. Nous vérifions que la taille de la pile est correcte après avoir poussé les éléments et que les éléments sont disséminés dans le bon ordre. Nous testons également que la méthode pop lève une exception lorsque la pile est vide. En exécutant ces scénarios de test, nous pouvons nous assurer que la classe fonctionne correctement pour ces scénarios spécifiques.

Un autre aspect important des tests est le test des exceptions. Dans cet exemple, nous allons tester une fonction qui divise deux nombres et pourrait lever une exception si le diviseur est zéro. Nous pouvons écrire un cas de test pour cette fonction en utilisant la macro TEST et la macro d'assertion EXPECT_THROW. Par exemple:

```
TEST(DivideTest, ZeroDivisor) {
   EXPECT_THROW(divide(10, 0), std::invalid_argument);
}
```

Dans cet exemple, nous testons la fonction de division avec une entrée de 10 et 0, et nous nous attendons à ce qu'une exception de std::invalid_argument soit levée. En exécutant ce scénario de test, nous pouvons nous assurer que la fonction de division gère correctement ce scénario spécifique.

Les objectifs des tests

Il est important de noter que ces exemples ne sont qu'un petit échantillon des types de cas de test qui peuvent être écrits à l'aide de Google Test. L'essentiel à retenir est qu'en écrivant des cas de test approfondis et bien conçus, nous pouvons accroître notre confiance dans le bon fonctionnement de notre code et faciliter l'identification et la correction des bogues qui pourraient survenir.
Lors de la rédaction de scénarios de test, il est important de garder à l'esprit les meilleures pratiques telles que :

- Couvrir une variété de scénarios, y compris les apports attendus et inattendus
- Utiliser des noms descriptifs pour les cas de test et les fonctions de test, et inclure des commentaires et de la documentation dans le code de test
- Organiser et regrouper des cas de test, tels que l'utilisation de dispositifs de test et de suites de tests

Il est également important de noter que si les tests peuvent améliorer la qualité et la robustesse de notre code, ils doivent être utilisés en conjonction avec d'autres techniques de développement logiciel telles que les revues de code et l'analyse statique de code.

Exemple de tests organisés avec Google Test

Un exemple d'organisation et de regroupement de cas de test à l'aide d'appareils de test et de suites de tests pourrait être le test d'une classe qui représente un compte bancaire. La classe a plusieurs méthodes telles que le dépôt, le retrait et le transfert.
Au lieu d'écrire des cas de test individuels pour chaque méthode, nous pouvons les regrouper dans un montage de test appelé "AccountTest". La classe peut contenir des fonctions de test pour chaque méthode, ainsi que tout code de configuration ou de nettoyage commun nécessaire.

```
class AccountTest : public ::testing::Test {
 protected:
  Account account;
  virtual void SetUp() {
    account = Account(1000);
  }
};

TEST_F(AccountTest, Deposit) {
  account.deposit(500);
  EXPECT_EQ(1500, account.getBalance());
}

TEST_F(AccountTest, Withdraw) {
  account.withdraw(500);
```

```
  EXPECT_EQ(500, account.getBalance());
}

TEST_F(AccountTest, Transfer) {
  Account account2(2000);
  account.transfer(500, &account2);
  EXPECT_EQ(500, account.getBalance());
  EXPECT_EQ(2500, account2.getBalance());
}
```

Création du main le plus simpliste avec Google Test

```
int main(int argc, char **argv) {
  ::testing::InitGoogleTest(&argc, argv);
  return RUN_ALL_TESTS();
}
```

L'importance de couvrir tous les cas possibles

En utilisant des appareils de test et des suites de tests, nous pouvons facilement organiser et regrouper les cas de test associés, ce qui facilite la compréhension de la suite de tests globale et facilite l'exécution des tests.
Un exemple de couverture d'une variété de scénarios, y compris des entrées attendues et inattendues, pourrait être le test d'une fonction qui calcule le carré d'un nombre donné.
Voici un exemple de la façon d'écrire des cas de test qui couvrent une variété de scénarios :

```
TEST(SquareTest, PositiveNumbers) {
    int result = square(5);
    EXPECT_EQ(25, result);
    result = square(10);
    EXPECT_EQ(100, result);
}

TEST(SquareTest, NegativeNumbers) {
    int result = square(-5);
    EXPECT_EQ(25, result);
    result = square(-10);
    EXPECT_EQ(100, result);
}

TEST(SquareTest, Zero) {
    int result = square(0);
```

```
    EXPECT_EQ(0, result);
}

TEST(SquareTest, LargeNumbers) {
    int result = square(INT_MAX);
    EXPECT_EQ(INT_MAX*INT_MAX, result);
}
```

Dans cet exemple, nous testons la fonction carrée avec différents types d'entrées, notamment des nombres positifs, des nombres négatifs, des zéros et des grands nombres. Nous testons également le comportement de la fonction pour les entrées attendues et inattendues. Par exemple, nous testons le comportement de la fonction lorsqu'elle reçoit des nombres positifs, des nombres négatifs et également lorsqu'elle reçoit de grands nombres, ce qui pourrait être une entrée inattendue.
Nous testons également que la fonction se comporte correctement lorsqu'elle reçoit un zéro en entrée. Il s'agit d'un exemple de cas marginal, qui est une entrée spécifique qui se situe soit aux limites de la plage d'entrées attendue, soit une entrée inhabituelle qui pourrait entraîner un comportement différent du programme.
En écrivant des cas de test qui couvrent une variété de scénarios, nous pouvons augmenter notre confiance dans le fait que la fonction fonctionnera correctement pour une large gamme d'entrées et qu'elle peut gérer des entrées inattendues.

Organiser des classes tests avec des paramètres Macro **TEST_P**

Une façon d'organiser les classes Google Test consiste à utiliser la macro **TEST_P** qui permet des tests paramétrés. La macro TEST_P est similaire à la macro TEST, mais elle vous permet de passer un ensemble de paramètres à la fonction de test, ce qui vous permet d'écrire une seule fonction de test qui peut tester plusieurs cas.
Voici un exemple d'utilisation de la macro TEST_P pour organiser des cas de test pour une fonction simple qui calcule l'aire d'un rectangle :

```
class RectangleTest : public ::testing::TestWithParam<std::tuple<int,
int, int>> {};

TEST_P(RectangleTest, AreaCalculation) {
    int width = std::get<0>(GetParam());
    int height = std::get<1>(GetParam());
    int expected_area = std::get<2>(GetParam());
    int calculated_area = calculateArea(width, height);
    EXPECT_EQ(expected_area, calculated_area);
}
```

```
INSTANTIATE_TEST_SUITE_P(
    RectangleTests, RectangleTest,
    ::testing::Values(
        std::make_tuple(5, 3, 15),
        std::make_tuple(10, 2, 20),
        std::make_tuple(2, 8, 16)
    )
);
```

Dans cet exemple, nous utilisons la macro `TEST_P` pour tester la fonction calculateArea. Nous passons un ensemble de paramètres à la fonction de test sous la forme d'un tuple contenant la largeur, la hauteur et l'aire attendue d'un rectangle. Nous utilisons ensuite la fonction std :: get<> pour extraire les valeurs du tuple et les utiliser dans notre fonction de test.
Nous utilisons également la macro **INSTANTIATE_TEST_SUITE_P** pour créer des instances de la fonction de test pour chaque ensemble de paramètres.
Cette approche nous permet d'organiser nos cas de test plus efficacement. Nous pouvons écrire une seule fonction de test qui peut tester plusieurs cas, plutôt que d'avoir à écrire une fonction de test distincte pour chaque cas. De plus, cela facilite l'ajout de nouveaux cas de test en ajoutant simplement de nouveaux tuples à la liste des valeurs.

Les méthodes SetUp() and TearDown()

Les fonctions SetUp () et TearDown () servent à effectuer toutes les actions de configuration ou de nettoyage qui doivent être effectuées avant et après chaque cas de test.
Voici un exemple d'utilisation des fonctions SetUp() et TearDown() conjointement avec la macro TEST_P :

```
class RectangleTest : public ::testing::TestWithParam<std::tuple<int,
int, int>> {

 protected:
  Rectangle* rect;

  virtual void SetUp() {
    int width = std::get<0>(GetParam());
    int height = std::get<1>(GetParam());
    rect = new Rectangle(width, height);
  }

  virtual void TearDown() {
    delete rect;
    rect = nullptr;
```

```
  }

};

TEST_P(RectangleTest, AreaCalculation) {
    int expected_area = std::get<2>(GetParam());
    int calculated_area = rect->calculateArea();
    EXPECT_EQ(expected_area, calculated_area);
}

INSTANTIATE_TEST_SUITE_P(
    RectangleTests, RectangleTest,
    ::testing::Values(
        std::make_tuple(5, 3, 15),
        std::make_tuple(10, 2, 20),
        std::make_tuple(2, 8, 16)
    )
);
```

Dans cet exemple, nous utilisons la fonction **SetUp()** pour créer une nouvelle instance de la classe rectangle avant chaque cas de test, et transmettons la largeur et la hauteur du tuple au constructeur. Nous utilisons également la fonction **TearDown()** pour supprimer l'objet rectangle après chaque cas de test, de cette façon nous n'avons pas de fuites de mémoire ou d'autres effets indésirables d'un cas de test à l'autre.
L'utilisation de SetUp() et TearDown() peut aider à améliorer l'efficacité de notre suite de tests en réduisant la quantité de code en double qui doit être écrit pour chaque cas de test. De plus, cela peut également aider à garantir que chaque cas de test commence par un état connu et cohérent.

Il est important de noter que la fonction SetUp() est appelée avant chaque cas de test et que la fonction TearDown() est appelée après chaque cas de test, que le cas de test ait réussi ou échoué.