

# Programmation Modulaire

## Exemple d'infrastructure de tests

**Florent Hivert**

Mél : `Florent.Hivert@lri.fr`

Adresse universelle : `http://www.lri.fr/~hivert`

**1** Pourquoi a-t-on besoin d'infrastructure pour les tests ?

**2** L'infrastructure doctest

**3** Écriture des tests

- Les tests unitaires
- Les tests de comportement
- Tester les affichages
- Quelques conseils pour écrire de bons tests

**4** Exécution des tests

# Plan

- 1** Pourquoi a-t-on besoin d'infrastructure pour les tests ?
- 2 L'infrastructure doctest
- 3 Écriture des tests
- 4 Exécution des tests

## Pourquoi les tests ?

Dans un monde parfait, tester serait une perte de temps :

Besoins → Imaginer → Écrire → Compiler → Exécuter.

Dans le monde réel, tester est un gain de temps !

- **Errare humanum est** – l'erreur est humaine : Un bon programmeur qui écrit du code fait en moyenne une erreur toutes les 20 lignes. La difficulté est de les trouver !!!
- **perseverare diabolicum est** – Si l'erreur est humaine, il est diabolique de persévérer (dans l'erreur) : les tests permettent de s'assurer que les erreurs passées ne seront pas reproduites.

## Rappel : l'importance des tests

### Retenir

*Les tests permettent de*

- *contrôler la qualité d'un logiciel*
- *vérifier qu'une correction ne casse pas une autre fonctionnalité (tests de **non régression**)*
- *documenter l'utilisation d'un logiciel par des exemples fonctionnels*

**Le test est un outil de contrôle du processus de fabrication du logiciel.**

## Infrastructure de test

### Problème

#### La macro CHECK

```
#define CHECK(test) if (!(test)) \  
    cout << "Test failed in file " << __FILE__ \  
        << " line " << __LINE__ << ": " #test << endl
```

que vous avez utilisée jusqu'ici rend bien des services, mais elle ne **passé pas à l'échelle** dès que l'on a plus que quelques fonctions.

## Test : passage à l'échelle

Quelques problèmes avec les tests tels que vous les avez pratiqués :

- Pas de message d'erreur : on croit qu'il n'y a pas de problèmes, alors que l'on a en fait **oublié de lancer certains tests** ;

⇒ On voudrait que le compilateur **trouve automatiquement** les tests à lancer ;

- **Arrêt du programme** en cas d'ASSERT qui ne passe pas dans un test.

⇒ On voudrait que le **programme continue de s'exécuter**.

- Quand un test rate, on voit la ligne, mais on **ne voit pas les valeurs des variables** ;

⇒ En cas d'échec, on voudrait avoir un **rapport le plus précis possible** sur le problème ;

- On lance systématiquement **tous les tests**, c'est parfois long ;

⇒ On voudrait pouvoir sélectionner les tests à lancer sans avoir à **recompiler le code** ;

## Test : passage à l'échelle

Quelques problèmes avec les tests tels que vous les avez pratiqués :

- Pas de message d'erreur : on croit qu'il n'y a pas de problèmes, alors que l'on a en fait **oublié de lancer certains tests** ;

⇒ On voudrait que le compilateur **trouve automatiquement** les tests à lancer ;

- **Arrêt du programme** en cas d'ASSERT qui ne passe pas dans un test.

⇒ On voudrait que le **programme continue de s'exécuter**.

- Quand un test rate, on voit la ligne, mais on **ne voit pas les valeurs des variables** :

⇒ En cas d'échec, on voudrait avoir un **rapport le plus précis possible** sur le problème ;

- On lance systématiquement **tous les tests**, c'est parfois long ;

⇒ On voudrait pouvoir sélectionner les tests à lancer sans avoir à **recompiler le code** ;

## Test : passage à l'échelle

Quelques problèmes avec les tests tels que vous les avez pratiqués :

- Pas de message d'erreur : on croit qu'il n'y a pas de problèmes, alors que l'on a en fait **oublié de lancer certains tests** ;

⇒ On voudrait que le compilateur **trouve automatiquement** les tests à lancer ;

- **Arrêt du programme** en cas d'ASSERT qui ne passe pas dans un test.

⇒ On voudrait que le **programme continue de s'exécuter**.

- Quand un test rate, on voit la ligne, mais on **ne voit pas les valeurs des variables** :

⇒ En cas d'échec, on voudrait avoir un **rapport le plus précis possible** sur le problème ;

- On lance systématiquement **tous les tests**, c'est parfois long ;

⇒ On voudrait pouvoir sélectionner les tests à lancer sans avoir à recompiler le code ;

## Test : passage à l'échelle

Quelques problèmes avec les tests tels que vous les avez pratiqués :

- Pas de message d'erreur : on croit qu'il n'y a pas de problèmes, alors que l'on a en fait **oublié de lancer certains tests** ;

⇒ On voudrait que le compilateur **trouve automatiquement** les tests à lancer ;

- **Arrêt du programme** en cas d'ASSERT qui ne passe pas dans un test.

⇒ On voudrait que le **programme continue de s'exécuter**.

- Quand un test rate, on voit la ligne, mais on **ne voit pas les valeurs des variables** :

⇒ En cas d'échec, on voudrait avoir un **rapport le plus précis possible** sur le problème ;

- On lance systématiquement **tous les tests**, c'est parfois long ;

⇒ On voudrait pouvoir sélectionner les tests à lancer sans avoir à recompiler le code ;

## Test : passage à l'échelle

Quelques problèmes avec les tests tels que vous les avez pratiqués :

- Pas de message d'erreur : on croit qu'il n'y a pas de problèmes, alors que l'on a en fait **oublié de lancer certains tests** ;

⇒ On voudrait que le compilateur **trouve automatiquement** les tests à lancer ;

- **Arrêt du programme** en cas d'ASSERT qui ne passe pas dans un test.

⇒ On voudrait que le **programme continue de s'exécuter**.

- Quand un test rate, on voit la ligne, mais on **ne voit pas les valeurs des variables** :

⇒ En cas d'échec, on voudrait avoir un **rapport le plus précis possible** sur le problème ;

- On lance systématiquement **tous les tests**, c'est parfois long ;

⇒ On voudrait pouvoir **sélectionner les tests** à lancer sans avoir à recompiler le code ;

## Test : passage à l'échelle

Quelques problèmes avec les tests tels que vous les avez pratiqués :

- Pas de message d'erreur : on croit qu'il n'y a pas de problèmes, alors que l'on a en fait **oublié de lancer certains tests** ;

⇒ On voudrait que le compilateur **trouve automatiquement** les tests à lancer ;

- **Arrêt du programme** en cas d'ASSERT qui ne passe pas dans un test.

⇒ On voudrait que le **programme continue de s'exécuter**.

- Quand un test rate, on voit la ligne, mais on **ne voit pas les valeurs des variables** :

⇒ En cas d'échec, on voudrait avoir un **rapport le plus précis possible** sur le problème ;

- On lance systématiquement **tous les tests**, c'est parfois long ;

⇒ On voudrait pouvoir **sélectionner les tests** à lancer sans avoir à recompiler le code ;

## Test : passage à l'échelle

Quelques problèmes avec les tests tels que vous les avez pratiqués :

- Pas de message d'erreur : on croit qu'il n'y a pas de problèmes, alors que l'on a en fait **oublié de lancer certains tests** ;

⇒ On voudrait que le compilateur **trouve automatiquement** les tests à lancer ;

- **Arrêt du programme** en cas d'ASSERT qui ne passe pas dans un test.

⇒ On voudrait que le **programme continue de s'exécuter**.

- Quand un test rate, on voit la ligne, mais on **ne voit pas les valeurs des variables** :

⇒ En cas d'échec, on voudrait avoir un **rapport le plus précis possible** sur le problème ;

- On lance systématiquement **tous les tests**, c'est parfois long ;

⇒ On voudrait pouvoir **sélectionner les tests** à lancer sans avoir à recompiler le code ;

## Test : passage à l'échelle

Quelques problèmes avec les tests tels que vous les avez pratiqués :

- Pas de message d'erreur : on croit qu'il n'y a pas de problèmes, alors que l'on a en fait **oublié de lancer certains tests** ;

⇒ On voudrait que le compilateur **trouve automatiquement** les tests à lancer ;

- **Arrêt du programme** en cas d'ASSERT qui ne passe pas dans un test.

⇒ On voudrait que le **programme continue de s'exécuter**.

- Quand un test rate, on voit la ligne, mais on **ne voit pas les valeurs des variables** :

⇒ En cas d'échec, on voudrait avoir un **rapport le plus précis possible** sur le problème ;

- On lance systématiquement **tous les tests**, c'est parfois long ;

⇒ On voudrait pouvoir **sélectionner les tests** à lancer sans avoir à recompiler le code ;

## Test : passage à l'échelle (2)

Exemple de rapport d'erreur dont vous avez l'habitude :

```
Test failed in file rational.cpp line 45: pgcd(213, 42) == 23
```

Exemple de rapport de tests réussis :

```
[doctest] doctest version is "2.4.0"
[doctest] run with "--help" for options
=====
[doctest] test cases:      16 |    16 passed |      0 failed |      0 skipped
[doctest] assertions:    89 |    89 passed |      0 failed |
[doctest] Status: SUCCESS!
```

Exemple de rapport de tests avec un rapport d'erreur détaillé :

```
[doctest] doctest version is "2.4.0"
[doctest] run with "--help" for options
=====
rational-test.cpp:29:
TEST CASE: fonction pgcd

rational-test.cpp:36: ERROR: CHECK( pgcd(213, 42) == 23 ) is NOT correct!
  values: CHECK( 3 == 23 )

=====
[doctest] test cases:      16 |    15 passed |      1 failed |      0 skipped
[doctest] assertions:    90 |    89 passed |      1 failed |
[doctest] Status: FAILURE!
```

## De nombreuses infrastructures de tests

### Attention

Le standard du C++ ne fournit pas d'infrastructure de tests.

Mais de très nombreux projets :

- Google Test <https://github.com/google/googletest>
- Boost Test [https://www.boost.org/doc/libs/1\\_75\\_0/libs/test/doc/html/index.html](https://www.boost.org/doc/libs/1_75_0/libs/test/doc/html/index.html)
- Dans Visual Studio© <https://visualstudio.microsoft.com/>
- CppUnit <https://sourceforge.net/projects/cppunit/>
- Catch <https://github.com/catchorg/Catch2>
- Cute <https://cute-test.com/>
- Doctest <https://github.com/onqtam/doctest>
- et plus d'une cinquantaine d'autres recensés sur Wikipedia.

# Plan

- 1 Pourquoi a-t-on besoin d'infrastructure pour les tests ?
- 2 L'infrastructure doctest**
- 3 Écriture des tests
- 4 Exécution des tests

## Une infrastructure simple : doctest

### Retenir

*Pour ce cours j'ai choisi doctest.*

Relativement récente et pas l'infrastructure la plus utilisée, mais

- facile à installer (un seul fichier à copier)
- pas de commande de compilation particulière

```
g++ -std=c++11 -Wall rational-test.cpp -o rational-test
```

- compilation rapide
- pas besoin de technique de programmation avancée (objet)
- possibilité de mélanger le code et les tests dans le même fichier

## Premier exemple

factorial-test.cpp

```
1 // Configure doctest pour qu'il fournisse le main
2 #define DOCTEST_CONFIG_IMPLEMENT_WITH_MAIN
3 #include "doctest.h"
4
5 int factorial(const int n) {
6     if (n <= 1) return 1;
7     else return factorial(n - 1) * n;
8 }
9
10 TEST_CASE("testing the factorial function") {
11     CHECK(factorial(0) == 1);
12     CHECK(factorial(1) == 1);
13     CHECK(factorial(2) == 2);
14     CHECK(factorial(3) == 6);
15     CHECK(factorial(10) == 3628800);
16 }
```

## Premier exemple avec main

Si l'on veut écrire soi-même son main, il faut appeler doctest comme suit

factorial-test-main.cpp

```
6 // Configure doctest pour être appelé depuis le main
7 #define DOCTEST_CONFIG_IMPLEMENT
8 #include "doctest.h"
9
10 int main(int argc, char **argv) {
11     doctest::Context context(argc, argv);
12     int res = context.run(); // run doctest
13     context.setAsDefaultForAssertsOutOfTestCases();
14     if (context.shouldExit()) return res;
15 // Puis écrire ce qu'on veut dans le main
16     cout << "Hello, World!" << endl;
17 }
```

# Plan

- 1 Pourquoi a-t-on besoin d'infrastructure pour les tests ?
- 2 L'infrastructure doctest
- 3 Écriture des tests**
- 4 Exécution des tests

## Test unitaire, test de comportement

### Retenir

*On distingue en général deux sortes de tests :*

- Les **tests unitaires** vérifient une **fonctionnalité élémentaire** du logiciel (une fonction, une valeur ...)
- Les **tests de comportement** vérifient le bon comportement d'un logiciel au cours d'un **scénario d'utilisation**.

- On exécute les tests unitaires dès que la fonctionnalité est implémentée.
- On exécute les tests de comportement quand le logiciel est en phase de complétion.

⇒ Écrire les tests le plus tôt possible, voire même avant le logiciel.

# Plan

- 1 Pourquoi a-t-on besoin d'infrastructure pour les tests ?
- 2 L'infrastructure doctest
- 3 Écriture des tests**
  - Les tests unitaires
  - Les tests de comportement
  - Tester les affichages
  - Quelques conseils pour écrire de bons tests
- 4 Exécution des tests

## Suites de tests, cas de test, assertions

### Retenir

- *un «**cas de test**» unitaire teste une fonctionnalité élémentaire unique*  
*Ex : test de la fonction factorielle*
- *il est composé de plusieurs **vérifications/assertions***  
*Ex : test que  $0! = 1$ ,  $1! = 1$ ,  $2! = 2$ ,  $3! = 6$  et  $4! = 24$*
- *on regroupe les tests unitaires reliés en «**suites de tests**»*  
*Ex : test des fonctions de combinatoire (factorielle, nombre d'arrangements, nombre de combinaisons. . .)*

## Exemple

Exemple d'un test unique d'une fonction

rational-test.cpp

```
36 TEST_CASE("fonction pgcd") {  
37     CHECK(pgcd(15, 12) == 3);  
38     CHECK(pgcd(15, 0) == 15);  
39     CHECK(pgcd(0, 12) == 12);  
40     CHECK(pgcd(-15, 12) == 3);  
41     CHECK(pgcd(42, 1) == 1);  
42     CHECK(pgcd(1, 42) == 1);  
43 }
```

# Écriture des assertions

## Retenir

*Pour écrire les assertions on écrit*

```
CHECK(propriété qui doit être vraie);
```

```
CHECK_FALSE(propriété qui doit être fausse);
```

Par exemple (pour les nombres rationnels) :

```
CHECK_FALSE(rat0 < rat0);  
CHECK(rat0 < rat1);
```

rational-test.cpp

## Assertion : exception

### Retenir

*On peut tester qu'un code déclenche une erreur (exception) :*

```
CHECK_THROWS_AS(calcul, exception attendue);
```

On teste ci-dessous que l'inverse du rationnel 0 fait une erreur :

```
CHECK_THROWS_AS(inverse(rat0), runtime_error);
```

rational-test.cpp

## Exemple d'une suite de tests

On regroupe ensemble les tests de la structure `Rat` (qu'on avait définie pour représenter les rationnels comme quotient de 2 entiers) :

```
                                rational-test.cpp    }
TEST_SUITE_BEGIN("Structure Rat");
TEST_CASE("Opérateur==") {
    CHECK(Rat{5, 4} == Rat{5, 4});
    CHECK(Rat{-5, 4} == Rat{-5, 4});
    CHECK(Rat{0, 1} == Rat{0, 1});
    CHECK_FALSE(Rat{0, 1} == Rat{1, 1});
    CHECK_FALSE(Rat{4, 5} == Rat{4, 3});
    CHECK_FALSE(Rat{5, 4} == Rat{-5, 4});
}
TEST_CASE("Opérateur!=") {
    CHECK_FALSE(Rat{5, 4} != Rat{5, 4});
    CHECK_FALSE(Rat{-5, 4} != Rat{-5, 4});
    CHECK_FALSE(Rat{0, 1} != Rat{0, 1});
    CHECK(Rat{0, 1} != Rat{1, 1});
    CHECK(Rat{4, 5} != Rat{4, 3});
    CHECK(Rat{5, 4} != Rat{-5, 4});
}
                                }
TEST_CASE("Fonction abs") {
    CHECK(abs(Rat{5, 4}) == Rat{5, 4});
    CHECK(abs(Rat{-5, 4}) == Rat{5, 4});
    CHECK(abs(Rat{2, 1}) == Rat{2, 1});
    CHECK(abs(Rat{-2, 1}) == Rat{2, 1});
    CHECK(abs(Rat{0, 1}) == Rat{0, 1});
}
TEST_CASE("Opérateur+") {
    CHECK(Rat{5, 4} + rat0 == Rat{5, 4});
    CHECK(rat0 + Rat{5, 4} == Rat{5, 4});
    CHECK(Rat{5, 4} + Rat{5, 4} == Rat{5, 2});
    CHECK(Rat{1, 2} + Rat{1, 3} == Rat{5, 6});
    CHECK(Rat{5, 4} + Rat{-5, 4} == rat0);
}
TEST_SUITE_END();
```

## Sous cas de test

### Retenir

*Il est possible de faire des sous-cas de tests :*

- *pour rassembler des tests reliés*
- *pour avoir des tests qui ont une initialisation commune*

## Sous cas de test : tests reliés

rational-test.cpp

```

121 TEST_CASE("Constructeur Ratio") {
122     // Deux surcharges de la fonction Ratio
123     SUBCASE("à partir de 2 entiers") {
124         CHECK(Ratio(15, 12) == Rat{5, 4});
125         CHECK(Ratio(15, -12) == Rat{-5, 4});
126         CHECK(isRatCorrect(Ratio(15, 12)));
127         CHECK(isRatCorrect(Ratio(15, -12)));
128         CHECK_THROWS_AS(isRatCorrect(Ratio(1, 0)), runtime_error);
129         CHECK_THROWS_AS(isRatCorrect(Ratio(5, 0)), runtime_error);
130     }
131     SUBCASE("à partir d'un entier") {
132         CHECK(Ratio(15) == Rat{15, 1});
133         CHECK(Ratio(-15) == Rat{-15, 1});
134         CHECK(isRatCorrect(Ratio(15)));
135         CHECK(isRatCorrect(Ratio(-15)));
136     }
137 }

```

## Sous cas de test : tests avec initialisation commune

rational-test.cpp

```
345 TEST_CASE("Modification") {
346     // Les deux sous-cas ci-dessous utilisent
347     // la même initialisation:
348     Rat a {3, 2};
349     SUBCASE("Ajout de 1") {
350         // a est initialisé comme ci-dessus
351         a = a + rat1;
352         CHECK(a == Rat {5, 2});
353     }
354     SUBCASE("Ajout de 2") {
355         // la variable a est initialisé mais n'est
356         // pas la même que pour le cas 1.
357         a = a + rat2;
358         CHECK(a == Rat {7, 2});
359     }
360 }
```

# Plan

- 1 Pourquoi a-t-on besoin d'infrastructure pour les tests ?
- 2 L'infrastructure doctest
- 3 Écriture des tests**
  - Les tests unitaires
  - Les tests de comportement**
  - Tester les affichages
  - Quelques conseils pour écrire de bons tests
- 4 Exécution des tests

## Scénario de test

### Compléments

Dans les tests de comportement, on raconte une histoire

- on fixe un état de départ
- on fait quelques actions
- on vérifie l'état d'arrivée

rational-test.cpp

```
SCENARIO("Exemple de calcul") {  
  GIVEN("Un nombre rationnel quelconque") {  
    Rat a {3, 2};  
    WHEN("on lui ajoute un") {  
      Rat b = a + rat1;  
      THEN("on obtient un rationnel plus grand") {  
        CHECK(b >= a);  
        CHECK(a <= b);  
      }  
    }  
  }  
}
```

# Plan

- 1 Pourquoi a-t-on besoin d'infrastructure pour les tests ?
- 2 L'infrastructure doctest
- 3 Écriture des tests**
  - Les tests unitaires
  - Les tests de comportement
  - Tester les affichages**
  - Quelques conseils pour écrire de bons tests
- 4 Exécution des tests

## Tester un affichage

### Retenir

*Pour tester un affichage, on va afficher dans un `ostringstream`.  
C'est un flux de sortie qui écrit dans une chaîne de caractères.*

rational-test.cpp

```
TEST_CASE("Opérateur <<") {  
    ostringstream ch; // Une sorte de chaîne où l'on peut afficher.  
    // ch.str() retourne la chaîne.  
    // ch.str(s) remplace la chaîne par s.  
    ch << Ratio(5, 4);  
    CHECK(ch.str() == "5/4");  
    ch.str(""); // remet ch à zéro (chaîne vide)  
    ch << Ratio(5);  
    CHECK(ch.str() == "5");  
}
```

## Tester un affichage (2)

L'utilisation de sous-cas évite d'avoir à réinitialiser la chaîne.

rational-test.cpp

```
TEST_CASE("Opérateur << avec SUBCASE") {
    ostringstream ch;
    SUBCASE("Affiche 5/4") {
        ch << Ratio(5, 4);
        CHECK(ch.str() == "5/4");
    }
    // Pas besoin de remettre ch à zero avec SUBCASE
    SUBCASE("Affiche 5") {
        ch << Ratio(5);
        CHECK(ch.str() == "5");
    }
}
```

# Plan

- 1 Pourquoi a-t-on besoin d'infrastructure pour les tests ?
- 2 L'infrastructure doctest
- 3 Écriture des tests**
  - Les tests unitaires
  - Les tests de comportement
  - Tester les affichages
  - Quelques conseils pour écrire de bons tests
- 4 Exécution des tests

## Quelques conseils pour écrire de bons tests

### Retenir

*Ne pas oublier de tester les cas triviaux ou particulier !*

- *addition de 0*
- *multiplication par 0 et 1*
- *recherche dans une chaîne de caractères / un tableau vide*
- *cas d'erreur*

```
1 CHECK(Rat{5, 4} + rat0 == Rat{5, 4});
2 CHECK(-rat0 == rat0);
3 CHECK(Rat{5, 4} * rat0 == rat0);
4 CHECK(Rat{5, 4} * rat1 == Rat{5, 4});
5 CHECK(Rat{-5, 4} / rat1 == Rat{-5, 4});
6 CHECK_THROWS_AS(isRatCorrect(Ratio(5, 0)), runtime_error);
```

## Quand utiliser CHECK\_FALSE ?

### Retenir

*Pour vérifier qu'une fonction booléenne renvoie bien **false** quand elle le doit :*

- *fonction de recherche, de test ou vérification ...*
- *opérateurs d'égalité, de comparaison.*

```

rational-test.cpp
TEST_CASE("Fonction de test isRatCorrect") {
    CHECK(isRatCorrect(rat0));
    CHECK(isRatCorrect(rat1));
    CHECK(isRatCorrect(rat2));
    CHECK(isRatCorrect(rat12));
    CHECK(isRatCorrect(ratn1));
    CHECK_FALSE(isRatCorrect({1, 0}));
    CHECK_FALSE(isRatCorrect({2, -1}));
    CHECK_FALSE(isRatCorrect({4, 6}));
}

TEST_CASE("Operateur==") {
    CHECK(Rat{-5, 4} == Rat{-5, 4});
    CHECK(Rat{0, 1} == Rat{0, 1});
    CHECK_FALSE(Rat{0, 1} == Rat{1, 1});
    CHECK_FALSE(Rat{4, 5} == Rat{4, 3});
    CHECK_FALSE(Rat{5, 4} == Rat{-5, 4});
}

TEST_CASE("Operateur!=") {
    CHECK_FALSE(Rat{5, 4} != Rat{5, 4});
    CHECK_FALSE(Rat{-5, 4} != Rat{-5, 4});
    CHECK_FALSE(Rat{0, 1} != Rat{0, 1});
    CHECK(Rat{0, 1} != Rat{1, 1});
    CHECK(Rat{4, 5} != Rat{4, 3});
    CHECK(Rat{5, 4} != Rat{-5, 4});
}

```

## Quand ne pas utiliser CHECK\_FALSE ?

### Attention

Dans les autres cas, il faut presque jamais utiliser CHECK\_FALSE. Il est en effet inutile de re-tester les opérateurs booléens que l'on a déjà testé par ailleurs.

Par exemple, dans le test de la valeur absolue

```
CHECK(abs(Rat{-5, 4}) == Rat{5, 4}); // OUI
```

```
CHECK_FALSE(abs(Rat{-5, 4}) == Rat{3, 2}); // NON
```

le deuxième test est **inutile**, puisque l'on a **déjà testé** l'égalité. Il est même nuisible car il donne l'impression que l'on fait un autre test de la fonction abs, alors qu'il teste une conséquence de la première assertion.

# Plan

- 1 Pourquoi a-t-on besoin d'infrastructure pour les tests ?
- 2 L'infrastructure doctest
- 3 Écriture des tests
- 4 Exécution des tests**

## Paramètres en ligne de commande

### Retenir

*Quand on lance un programme depuis le terminal, on peut lui passer des paramètres.*

Par exemple, dans la ligne

```
g++ -Wall -std=c++11 rational-test.cpp -o rational-test
```

le programme est `g++`, les paramètres sont les 5 chaînes de caractères : `"-Wall"`, `"-std=c++11"`, `"rational-test.cpp"`, `"-o"` et `"rational-test"`.

# Paramètres en ligne de commande

## Retenir

*En C++, pour récupérer les paramètres, on modifie l'entête de la fonction main en :*

```
int main(int npar, const char* params[])
```

*où*

- *npar est le nombre d'arguments de la ligne de commande (y compris le programme lui-même);*
- *params est un tableau de chaînes de caractères bas niveau C.*

## Exemple de récupération de la ligne de commande

cmdline.cpp

```
1 #include <iostream>
2 #include <string>
3 using namespace std;
4
5 int main(int nbpar, char *params[]) {
6     cout << "Nom du programme : " << params[0] << endl;
7
8     for (int i=1; i < nbpar; ++i) {
9         string s = params[i];
10        s = "\"" + s + "\"";
11        cout << "Parametre no " << i << " : " << s << endl;
12    }
13    return 0;
14 }
```

## Doctest et ligne de commande

Le début du main

```
int main(int argc, const char** argv) {  
    doctest::Context context(argc, argv);  
    int test_result = context.run();  
    context.setAsDefaultForAssertsOutOfTestCases();  
    if (context.shouldExit()) return test_result;
```

récupère la ligne de commande et la transmet à doctest.

### Retenir

*On peut donc configurer l'exécution de doctest avec la ligne de commande d'exécution de notre programme.*

## Doctest et ligne de commande

Le début du main

```
int main(int argc, const char** argv) {  
    doctest::Context context(argc, argv);  
    int test_result = context.run();  
    context.setAsDefaultForAssertsOutOfTestCases();  
    if (context.shouldExit()) return test_result;
```

récupère la ligne de commande et la transmet à doctest.

### Retenir

*On peut donc configurer l'exécution de doctest avec la ligne de commande d'exécution de notre programme.*

## Quelques options de ligne de commande de doctest

- `-h` : affiche l'aide
- `-ltc` ou `-list-test-cases` : affiche les cas de tests
- `-lts` ou `-list-test-suites` : affiche les suites de tests
- `-test-case=<filters>` : n'exécute que les cas qui passent le filtre.
- `-s` : affiche également les tests réussis
- `-d` : affiche le temps passé sur chaque test
- et plein d'autre options ...

**Bonne écriture de tests !!!**

La qualité des tests sera prise en compte dans la notation de vos TP et surtout du projet. . .