université PARIS-SACLAY

Correction TP de programmation n°4

Cours de programmation modulaire

—Licence MI/IM - Info 3—

Les tests avec `doctest`

Dans cette séance de travaux pratique, nous nous familiarisons avec l'infrastructure de test `doctest`. Cette infrastructure sera utilisée systématiquement dans les séances suivantes. Il est donc important de ne pas se contenter de répondre aux questions mais aussi d'explorer ce concept et son utilisation pour voir ce qui est faisable et ce qui ne l'est pas.

Nous en profiterons pour revoir les `structs` et la surcharge d'opérateurs.

▸ Exercice 1. (On fait des tests avec `doctest`)

1. Ouvrir le fichier `puissance.cpp` et coder la fonction `int puissance(int nombre, int exposant)` qui calcule la puissance de `nombre` par la valeur de `exposant`.

✂ ----------

```
1      int res=1;
2      for (int i=1; i<=exposant; i++) {
3          res = res*nombre;
4      }
5      return res;
```

---------- ✂

2. Pour pouvoir faire des tests avec `doctest`, il suffit que le fichier `doctest.h` soit dans le même répertoire que le fichier `puissance.cpp`.
   — Remarquer les deux lignes du fichier `puissance.cpp` qui permettent d'inclure `doctest` :

```
1  #define DOCTEST_CONFIG_IMPLEMENT
2  #include "doctest.h"
```

   — Remarquer aussi les 4 lignes à partir du `main` qui permettent de lancer tous les tests `doctest` :

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

   — Enfin, remarquer les deux lignes qui permettent de définir l'opération de test ainsi qu'une proposition de test :

```
1  TEST_CASE("Test de la fonction puissance") {
2     CHECK(puissance(10, 0) == 1);
```

3. Proposer d'autres tests pertinents avec `doctest` pour la fonction `int puissance(int nombre, int exposant)`.

```
    CHECK(puissance(1, 1) == 1);
    CHECK(puissance(2, 1) == 2);
    CHECK(puissance(3, 2) == 9);
    CHECK(puissance(10, 5) == 100000);
```

4. Compiler et exécuter votre programme. Regarder et analyser ce que votre programme affiche.

```
[doctest] doctest version is "2.4.0"
[doctest] run with "--help" for options
===============================================================================
[doctest] test cases:      1 |      1 passed |      0 failed |      0 skipped
[doctest] assertions:      5 |      5 passed |      0 failed |
[doctest] Status: SUCCESS!
1
2
4
8
16
32
64
128
256
512
1024
2048
4096
8192
16384
32768
65536
131072
262144
524288
1048576
```

5. Ajouter un test volontairement faux, compiler et lancer les tests. Vérifier que l'erreur est bien reportée.

```
[doctest] doctest version is "2.4.0"
[doctest] run with "--help" for options
===============================================================================
puissance.cpp:25:
TEST CASE:  Test de la fonction puissance

puissance.cpp:28: ERROR: CHECK( puissance(1, 1) == 5 ) is NOT correct!
  values: CHECK( 1 == 5 )

===============================================================================
```

```
[doctest] test cases:      1 |      0 passed |      1 failed |      0 skipped
[doctest] assertions:      5 |      4 passed |      1 failed |
[doctest] Status: FAILURE!
1
2
4
8
16
32
64
128
256
512
1024
2048
4096
8192
16384
32768
65536
131072
262144
524288
1048576
```

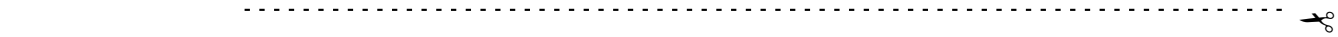
---

6. Remplacer le `CHECK` du test faux par un `CHECK_FALSE`, recompiler et vérifier que l'erreur n'apparaît plus. Note : ce test avec `CHECK_FALSE` est très probablement inutile ici. On va néanmoins le conserver pour l'exemple.
7. Exécuter votre programme avec la commande `./puissance -h`. Cela va afficher l'aide (help) de toutes les options que vous pouvez essayer.
8. Essayer en particulier ce que font les appels `./puissance -s` et `./puissance -d`.

`./puissance -s` affiche aussi le détail pour les tests réussis (comme ci-dessous), et `./puissance -d` affiche le temps passé sur chaque test.

```
[doctest] doctest version is "2.4.0"
[doctest] run with "--help" for options
===============================================================================
puissance.cpp:25:
TEST CASE:  Test de la fonction puissance

puissance.cpp:26: SUCCESS: CHECK( puissance(10, 0) == 1 ) is correct!
  values: CHECK( 1 == 1 )

puissance.cpp:28: ERROR: CHECK( puissance(1, 1) == 5 ) is NOT correct!
  values: CHECK( 1 == 5 )

puissance.cpp:29: SUCCESS: CHECK( puissance(2, 1) == 2 ) is correct!
  values: CHECK( 2 == 2 )

puissance.cpp:30: SUCCESS: CHECK( puissance(3, 2) == 9 ) is correct!
  values: CHECK( 9 == 9 )

puissance.cpp:31: SUCCESS: CHECK( puissance(10, 5) == 100000 ) is correct!
```

```
  values: CHECK( 100000 == 100000 )

===============================================================================
[doctest] test cases:      1 |      0 passed |      1 failed |      0 skipped
[doctest] assertions:      5 |      4 passed |      1 failed |
[doctest] Status: FAILURE!
1
2
4
8
16
32
64
128
256
512
1024
2048
4096
8192
16384
32768
65536
131072
262144
524288
1048576
```



▸ **Exercice 2. (Surcharge et tests `doctest` de fonctions pour les dates)** Vous avez déjà codé plusieurs fonctions sur le sujet des dates dans la première séance de TP. On va donc se baser sur ce que vous aviez codé pour effectuer des surcharges d'opérateurs pour la manipulation des dates et utiliser ensuite `doctest` pour réaliser des tests sur un certain nombre de fonctions.

On vous demande d'ouvrir le fichier `date-doctest.cpp` et de :

1. Coder la surcharge de l'opérateur d'affichage `<<` pour afficher une `Date` sous le format jj/mm/aaaa.

```
17
18 /** Surcharge de << pour afficher une Date sous le format jj/mm/aaaa
19 * @param[in] d : Date
20 **/
21 std::ostream& operator<< (std::ostream &out, Date d) {
22   //
23   out << setfill('0') << setw(2) << d.jour << "/"
24       << setfill('0') << setw(2) << d.mois << "/" << d.annee;
25   return out;
26   //
27 }
```

2. Le `main` fourni contient des affichages de dates. Compiler et exécuter votre programme pour vérifier votre opérateur d'affichage. Comme `doctest` lance tous les tests y compris ceux des fonctions que vous n'avez pas encore écrites, `doctest` signale qu'un certains nombre de tests ont échoués, mais vous devez voir ensuite s'afficher deux dates.

Les fonctions suivantes seront testées directement avec `doctest` au fur et à mesure en vérifiant que les tests correspondants passent.

3. Coder la surcharge de l'opérateur `==` pour vérifier que deux dates `d1` et `d2` sont égales.

```
32
33 /** Surcharge de == pour verifier que deux dates d1 et d2 sont egales
34 * @param[in] d1 : Date
35 * @param[in] d2 : Date
36 * @return le booléen correspondant au test
37 **/
38 bool operator==(Date d1, Date d2) {
39   //
40   return d1.jour == d2.jour and d1.mois == d2.mois and d1.annee == d2.annee;
41   //
42 }
```

4. Coder la surcharge de l'opérateur `!=` pour vérifier que deux dates `d1` et `d2` sont différentes.

```
46
47 /** Surcharge de != pour verifier que deux dates d1 et d2 sont differentes
48 * @param[in] d1 : Date
49 * @param[in] d2 : Date
50 * @return le booléen correspondant au test
51 **/
52 bool operator!=(Date d1, Date d2) {
53   //
54   return not (d1 == d2);
55   //
56 }
```

---

5. On vous a donné un exemple de test `doctest` pour ces deux dernières surcharges, proposer d'autres tests pertinents.

```
60 TEST_CASE("surcharge ==  et !=") {
61    CHECK(Date{1, 1, 2000} == Date{1, 1, 2000});
62    //
63    CHECK(Date{14, 7, 2021} == Date{14, 7, 2021});
64    CHECK(Date{14, 2, 1994} == Date{14, 2, 1994});
65    CHECK_FALSE(Date{14, 2, 1994} == Date{15, 2, 1994});
66    CHECK(Date{1, 1, 2000} != Date{10, 1, 2000});
67    CHECK(Date{1, 1, 2000} != Date{1, 1, 2020});
68    CHECK_FALSE(Date{1, 1, 2000} != Date{1, 1, 2000});
69    //
70 }
```

---

6. Proposer des tests pertinents pour la fonction fournie `bool estBissextile(int annee)`.

```
83 TEST_CASE("fonction estBissextile") {
84    CHECK(estBissextile(2000)); // année multiple de 400
85    //
86    CHECK(estBissextile(2020)); // année multiple de 4
87    CHECK(estBissextile(40)); // année multiple de 4
88    CHECK_FALSE(estBissextile(2021)); // année non multiple de 4
89    CHECK_FALSE(estBissextile(1900)); // année multiple de 100
90    //
91 }
```

---

7. Proposer des tests pertinents pour la fonction fournie `int nbJourMois(int mois, int annee)`. Parmi les tests, il faut aussi vérifier que si le mois est invalide (-1, 0 où 13 par exemple), une exception est bien levée. Pour ceci, on utilise `CHECK_THROWS_AS`.

```
114 TEST_CASE("fonction nbJourMois") {
115    CHECK(nbJourMois(2, 2000) == 29);
116    //
117    CHECK(nbJourMois(6, 2020) == 30);
118    CHECK(nbJourMois(12, 2021) == 31);
119    CHECK(nbJourMois(2, 1900) == 28);
120    CHECK_THROWS_AS(nbJourMois(-4, 1900), range_error);
121    CHECK_THROWS_AS(nbJourMois(0, 1900), range_error);
122    CHECK_THROWS_AS(nbJourMois(13, 1900), range_error);
123    //
124 }
```

8. Coder la surcharge de l'opérateur de lecture `>>` pour lire une Date (jj mm aaaa). Il faut bien évidemment vérifier que la date lue est correcte en utilisant la fonction fournie `bool estCorrecteDate(d)` qui vérifie si la date `d` est correcte. Si ce n'est pas le cas, on lèvera une exception.

```
138
139 /** Surcharge de >> pour lire une Date
140 **/
141 std::istream& operator>> (std::istream &in, Date &d) {
142   //
143   in >> d.jour >> d.mois >> d.annee;
144   if (not estCorrecteDate(d)) {
145       throw runtime_error("Date incorrecte !");
146   }
147   return in;
148   //
149 }
```

9. Dé-commenter dans le main les lignes qui font la saisie de la date `aujourdhui` et vérifier que tout marche bien.
10. Coder la surcharge de l'opérateur `<` qui dit si la date `d1` est avant la date `d2`. On demande de n'utiliser ni boucle ni la fonction lendemain.

```
170
171 /** Surcharge de < pour verifier si une date d1 est avant une date d2
172 * @param[in] d1 : Date
173 * @param[in] d2 : Date
174 * @return le booléen correspondant au test
175 **/
176 bool operator<(Date d1, Date d2) {
177   //
178   if (d1.annee < d2.annee) return true;
179   if (d1.annee == d2.annee) {
180     if (d1.mois < d2.mois) return true;
```

```
181      if (d1.mois == d2.mois) {
182        if (d1.jour < d2.jour) return true;
183      }
184    }
185    return false;
186    //
187  }
```

---

11. Proposer des tests pertinents pour la fonction de surcharge précedente `<`.

```
191  TEST_CASE("surcharge <") {
192     CHECK_FALSE(Date{1, 1, 2000} < Date{1, 1, 1999});
193     //
194     CHECK(Date{14, 7, 2021} < Date{15, 7, 2021});
195     CHECK_FALSE(Date{14, 2, 1994} < Date{14, 1, 1994});
196     CHECK(Date{1, 1, 2000} < Date{10, 2, 2000});
197     //
198  }
```

---

12. Coder la surcharge de l'opérateur `+` qui ajoute un nombre de jours `n` à une date `d`. On supposera que `n` est positif et lèvera une exception si ce n'est pas le cas. On pourrait faire une version simple en utilisant la fonction lendemain, mais elle ne serait pas très efficace. Il est mieux de faire une version sans appel à lendemain (et sans passer par tous les jours de `d` à `d+n`).

```
203  Date operator+(Date d, int n){
204    //
205    if (n < 0) throw range_error("ajout d'un nombre négatif à une date");
206    d.jour = d.jour + n;
207    while (d.jour > nbJourMois(d.mois, d.annee)) {
208      d.jour = d.jour - nbJourMois(d.mois, d.annee);
209      d.mois++;
210      if (d.mois == 13) {
211        d.mois = 1;
212        d.annee++;
213      }
214    }
215    return d;
216    //
217  }
```

---

13. Proposer des tests pertinents pour l'opérateur `+`.

```
TEST_CASE("Operateur + ") {
   CHECK((Date{1, 1, 2000} + 8) == Date{9, 1, 2000});
   //
   CHECK((Date{31, 12, 1999} + 5) == Date{5, 1, 2000});
   CHECK((Date{28, 2, 2020} + 2) == Date{1, 3, 2020});
   CHECK((Date{28, 2, 2020} + 0) == Date{28, 2, 2020});
   CHECK_THROWS_AS((Date{1, 1, 2000} + (-1)), range_error);
   //
}
```

14. Coder la surcharge de l'opérateur - qui calcule le nombre de jours écoulés entre les dates `d2` et `d1`. On supposera que `d1` est après `d2` et levera une exception si ce n'est pas le cas.

```
int operator-(Date d1, Date d2){
    //
    if (d1 < d2) throw range_error("difference de date invalide");
    int nbDeJourDeDifference = 0;
    while (d2 < d1) {
      d2 = lendemain(d2);
      nbDeJourDeDifference++;
    }
    return nbDeJourDeDifference;
    //
}
```

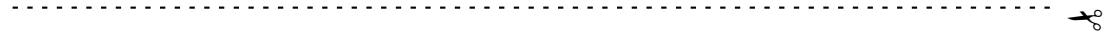
15. Proposer des tests pertinents pour l'opérateur -.

```
TEST_CASE("Operateur - ") {
   CHECK((Date{1, 1, 2001} - Date{1, 1, 2000}) == 366);
   //
   CHECK((Date{5, 1, 2000} - Date{1, 1, 2000}) == 4);
   CHECK((Date{19, 9, 2013} - Date{19, 9, 2013}) == 0);
   CHECK_THROWS_AS((Date{1, 1, 2000} - Date{5, 1, 2000}), range_error);
   //
}
```

16. Proposer des tests pertinents pour la fonction `int jourDate(Date d)` qui renvoie le jour de la semaine d'une date (renvoie 0 pour lundi, 1 pour mardi, ...).

```
TEST_CASE("fonction jourDate") {
   CHECK(jourDate({1, 1, 2000}) == 5 );
   //
   CHECK(jourDate(Date{8, 1, 2000}) == 5 );
```

```
277     CHECK(jourDate(Date{24, 1, 2000}) == 0 );
278     CHECK(jourDate(Date{17, 2, 2021}) == 2 );   //jour de TD le mercredi
279     CHECK(jourDate(Date{18, 2, 2021}) == 3 );   //jour de TD le jeudi
280     //
281 }
```



▸ **Exercice 3.** S'il vous reste du temps, reprendre le jeu d'échec du TP 3, le compléter et ajouter des tests avec `doctest` (voir exercices 2 et 5 du TP3, à compléter aussi selon vos propres idées).

Pour installer `doctest` dans le répertoire du TP 3 il suffit d'y copier le fichier `doctest.h`. Eventuellement l'ajouter aux fichiers pris en compte par Git et les scripts Travo avec

```
git add doctest.h
```

De même, vous pouvez reprendre la calcul de racine carrée du TP 3, le compléter et ajouter des tests avec `doctest` (voir exercices 3 et 4 du TP3).