université PARIS-SACLAY

Correction TP de programmation n°7

Cours de programmation modulaire

—Licence MI/IM/MNSI - Info 3—

Classes et Constructeurs

Dans cette séance de travaux pratiques, nous allons apprendre à contrôler la construction d'un objet.

▸ **Exercice 1. (Nombres entiers en précision arbitraire)** Il existe plusieurs formats de stockage des entiers en `C++`, notamment `int` et `long int` (mais aussi `short`, `unsigned short`, `unsigned int` et `unsigned long int`). Avec `int`, celui que nous avons le plus utilisé, on ne peut écrire des nombres que jusqu'à $2^{31} - 1 = 2147483647$. Si l'on incrémente ce nombre, il se produit un dépassement de capacité, le résultat est alors le nombre *négatif* -2147483648. Vous pouvez le tester avec le code du fichier `depcap.cpp` fourni :

```
int n = 2147483647;
cout << n+1 << endl;
```

Ce comportement peut nous bloquer lors d'opérations avec de très grands nombres. Ce problème peut être mitigé avec un `long int`, par exemple, qui a une taille en mémoire plus grande qu'un `int`, mais les `long int` sont aussi limités à $2^{63} - 1 = 9223372036854775807$.

En `C++`, on utilise habituellement la bibliothèque `GnuMultiPrecision` pour résoudre ce problème (voir `https://gmplib.org/` et `https://gmplib.org/manual/C_002b_002b-Class-Interface` pour les classes `C++`).

Dans cet exercice, nous allons écrire le code par nous mêmes en créant une classe pour modéliser et stocker des nombres entiers naturels sans limitation de précision. On rappelle que l'écriture 2375 en base dix signifie

$$2375 = 5 + 7 \times 10 + 3 \times 100 + 2 \times 1000 = 5 \times 10^0 + 7 \times 10^1 + 3 \times 10^2 + 2 \times 10^3 \,.$$

Le chiffre correspondant à la puissance i de 10 est appelé *chiffre de poids* i. Par exemple dans 2375, le chiffre de poids 0 est 5 et celui de poids 2 est 3.

Le format de stockage va être le suivant :

— les chiffres du nombre sont stockés dans un vecteur de `int` nommé `chiffres`.

— le chiffre de poids i est stocké dans la case i du vecteur. Ainsi pour l'affichage du nombre on commence par la **fin** du vecteur.

— Invariant : le chiffre de plus grand poids doit toujours être différent de zéro.

Ainsi, 456 est codé par le vecteur `{6,5,4}`, et 789065 est codé `{5,6,0,9,8,7}`.

Attention : une conséquence du choix précédent est que le nombre 0 est codé par le vecteur vide et non par le vecteur `{0}`.

Pour cet exercice, vous modifierez le fichier `entier.cpp`.

1. Déclarez et définissez un constructeur à partir d'un vecteur d'entiers donné (pour la classe `Naturel` fournie). Si l'une des entrées du vecteur ne contient pas un chiffre valide (entre 0 et 9), on signalera une exception `invalid_argument`. Pour le moment, on ne s'occupe pas des 0 qu'il pourrait y avoir à la fin du vecteur, ce sera le sujet de la question suivante.

```
13 // Le chiffre de poids i est en position i
14 // Le chiffre de poids le plus fort est différent de zéro
15 struct Naturel {
16   vector<int> chiffres;
17
18   // Constructeurs
19   //
20   Naturel(const vector<int>& ch);
21   Naturel() : chiffres{} {}; // Ajouté en question 4.
22   Naturel(int n); // Ajouté en question 8.
23   //
49 };
56 // Constructeur
57 Naturel::Naturel(const std::vector<int>& ch) : chiffres{ch} {
58   //
59   for (int i : chiffres) {
60     if (i < 0 or i > 9)
61       throw invalid_argument("Chiffre invalide " + to_string(i));
62   }
63   normalise();  // Ajouté à la fin de la question 2.
64   //
65 }
```

2. Déclarez et définissez une méthode `normalise` qui supprime les 0 à la fin du vecteur de chiffres du `Naturel`, de manière à respecter l'invariant. Attention ! Il ne faut supprimer que les 0 à la fin. On pourra utiliser la méthode `pop_back` des vecteurs. Par exemple, 0420 s'écrira `{0,2,4,0}` et sera normalisé en `{0,2,4}`.

```
69 void Naturel::normalise() {
70   while (chiffres.size() != 0 and chiffres.back() == 0) {
71     chiffres.pop_back();
72   }
73 }
```

3. Appeler cette méthode à la fin du constructeur, pour que le `Naturel` construit soit normalisé.
4. Complétez les tests fournis afin de tester soigneusement le constructeur, en vérifiant bien tous les cas d'erreur et la suppression des 0.

```
 86 TEST_CASE("Constructeur") {
 87   //
 88   CHECK(Naturel{{3, 2, 1}}.chiffres == vector<int>{{3, 2, 1}});
 89   CHECK(Naturel{{2, 2, 0, 0}}.chiffres == vector<int>{{2, 2}});
 90   CHECK(Naturel{{0, 2, 2, 0, 0}}.chiffres == vector<int>{{0, 2, 2}});
 91   CHECK(Naturel{{0, 0, 0}}.chiffres == vector<int>{});
 92
 93   CHECK_THROWS_AS((Naturel{{0, 12, 0}}), invalid_argument);
 94   CHECK_THROWS_AS((Naturel{{0, -1, 2}}), invalid_argument);
 95
 96   CHECK(Naturel{0}.chiffres == vector<int>{});
 97   CHECK(Naturel{123}.chiffres == vector<int>{{3, 2, 1}});
 98   CHECK(Naturel{1230}.chiffres == vector<int>{{0, 3, 2, 1}});
 99   //
100 }
```

5. Écrire le constructeur par défaut qui construit le nombre 0.
6. Écrire une méthode `nb_chiffres` qui retourne le nombre de chiffres. Tester cette méthode.
7. Écrire une méthode `ieme_chiffre` qui retourne le chiffre de poids i. Note : le chiffre de poids 5 de 421 est 0. Tester cette méthode.

```
 28   int nb_chiffres() const { return chiffres.size(); }
 29   int ieme_chiffre(int i) const {
 30     if (i < nb_chiffres()) return chiffres[i]; else return 0;
 31   }
104 TEST_CASE("Méthode nb_chiffres") {
105   //
106   CHECK(Naturel{{3, 2, 1}}.nb_chiffres() == 3);
107   CHECK(Naturel{{2, 2}}.nb_chiffres() == 2);
108   CHECK(Naturel{{0, 2, 2}}.nb_chiffres() == 3);
109   CHECK(Naturel{}.nb_chiffres() == 0);
110   //
111 }
112
113 TEST_CASE("Méthode ieme_chiffres") {
114   //
115   CHECK(Naturel{{3, 2, 1}}.ieme_chiffre(0) == 3);
116   CHECK(Naturel{{2, 2}}.ieme_chiffre(1) == 2);
117   CHECK(Naturel{{0, 2, 2}}.ieme_chiffre(2) == 2);
118   CHECK(Naturel{{0, 2, 2}}.ieme_chiffre(3) == 0);
119   CHECK(Naturel{}.ieme_chiffre(1) == 0);
120   //
121 }
```

8. Surchargez l'opérateur << pour la classe `Naturel`.

```
125 ostream& operator<<(ostream& out, const Naturel &n) {
126   //
127   if (n.chiffres.size() == 0) out << "0";
128   for (int i = n.chiffres.size() - 1; i >= 0; i--)
129     out << n.chiffres[i];
130   return out;
131   //
132 }
```

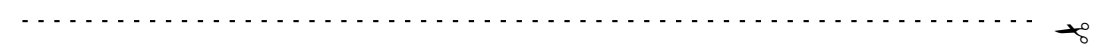
9. Écrivez un constructeur qui prend un `int` habituel. Testez.

```
77 Naturel::Naturel(int n) : chiffres{} {
78   while (n != 0) {
79     chiffres.push_back(n % 10);
80     n = n / 10;
81   }
82 }
```

10. Surchargez les opérateurs == et != pour comparer deux entiers. Testez ces fonctions.

```
179 //
180 bool Naturel::operator== (const Naturel& b) const {
181   return chiffres == b.chiffres;
182 }
183
184
185 bool Naturel::operator!= (const Naturel& b) const {
186   return not (*this == b);
187 }
188
189 TEST_CASE("operator ==") {
190   CHECK(N0 == N0);
191   CHECK(N1 == N1);
192   CHECK(N2 == N2);
193   CHECK(N1000 == N1000);
194   CHECK(a == a);
195   CHECK(b == b);
196   CHECK(N0 != N1);
197   CHECK_FALSE(N0 != N0);
198   CHECK_FALSE(N0 == N1000);
199   CHECK_FALSE(N2 == a);
```

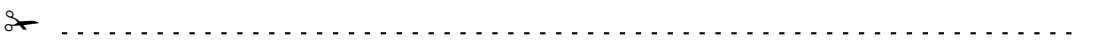
```
200   CHECK_FALSE(a == b);
201 }
202 //
```



▸ Exercice 2. (Nombres entiers en précision arbitraire (suite))

Cet exercice est prévu pour ceux qui vont vite. S'il vous reste moins d'une heure, passez à l'exercice suivant.

1. Surchargez les opérateurs <, >, <= et >= pour comparer deux Naturels. Vous pourrez vous aider de fonctions annexes, `compare` pour comparer deux int (qui pourra vous retourner différentes valeurs selon que vos deux arguments sont supérieurs l'un à l'autre ou égaux), que vous pourrez surcharger pour comparer deux Naturels. Testez ces fonctions.



```
206
207 //
208 int compare_int(int a, int b) {
209   if (a == b) return 0;
210   if (a > b) return +1;
211   return -1;
212 }
213 int Naturel::compare(const Naturel& b) const {
214   int res = compare_int(chiffres.size(), b.chiffres.size());
215   if (res != 0) return res;
216   for (int i = chiffres.size() - 1; i >= 0; i--) {
217     res = compare_int(chiffres[i], b.chiffres[i]);
218     if (res != 0) return res;
219   }
220   return 0;
221 }
222
223
224 bool Naturel::operator < (const Naturel& b) const {
225 //
226   return compare(b) < 0;
227 //
228 }
229 bool Naturel::operator > (const Naturel& b) const {
230 //
231   return b < *this;
232 //
233 }
234
235 TEST_CASE("operator <"){
236 //
237   vector<Naturel> testsample = {N0, N1, N2, N1000 , a, b};
238   for (int i = 0; i < int(testsample.size()); i++)
```

```
239     for (int j = 0; j < int(testsample.size()); j++) {
240       CHECK((testsample[i] < testsample[j]) == (i < j));
241       CHECK((testsample[i] > testsample[j]) == (i > j));
242     }
243 //
244 }
245
246 bool Naturel::operator <= (const Naturel& b) const {
247 //
248   return compare(b) <= 0;
249 //
250 }
251 bool Naturel::operator >= (const Naturel& b) const {
252 //
253   return b <= *this;
254 //
255 }
256 TEST_CASE("operator <="){
257 //
258   vector<Naturel> testsample = {N0, N1, N2, N1000, a, b};
259   for (int i = 0; i < int(testsample.size()); i++)
260     for (int j = 0; j < int(testsample.size()); j++) {
261       CHECK((testsample[i] <= testsample[j]) == (i <= j));
262       CHECK((testsample[i] >= testsample[j]) == (i >= j));
263     }
264 }
265 //
```

---

2. Surchargez l'opérateur +. Testez cette fonction.

```
270 Naturel Naturel::operator+ (const Naturel &b) const {
271 //
272   if (chiffres.size() > b.chiffres.size()) {
273     return b + *this;
274   }
275   Naturel res{};
276   bool retenue = false;
277   int i = 0;
278   while (i < int(chiffres.size())) {
279     int ch = chiffres[i] + b.chiffres[i] + retenue;
280     retenue = (ch >= 10);
281     if (retenue) ch -= 10;
282     res.chiffres.push_back(ch);
283     i++;
284   }
285   while (i < int(b.chiffres.size())) {
286     int ch = b.chiffres[i] + retenue;
287     retenue = (ch >= 10);
288     if (retenue) ch -= 10;
289     res.chiffres.push_back(ch);
```

```
290       i++;
291     }
292     if (retenue)
293       res.chiffres.push_back(retenue);
294     return res;
295   //
296   }
297
298   TEST_CASE("operator +"){
299   //
300     vector<Naturel> testsample = {N0, N1, N2, N1000, a, b};
301     CHECK(N0 + N0 == N0);
302     for (Naturel n : testsample) {
303       CHECK(N0 + n == n);
304       CHECK(n + N0 == n);
305     }
306     for (Naturel n : testsample)
307       for (Naturel m : testsample)
308         CHECK(n + m == m + n);
309     for (Naturel n : testsample)
310       for (Naturel m : testsample)
311         for (Naturel p : testsample)
312           CHECK((n + m) + p == n + (m + p));
313     CHECK(a + b == Naturel{{0, 0, 6, 3, 0, 3, 1, 5}});
314   //
315   }
```

3. Surchargez l'opérateur `-=`. Testez cette fonction.

```
320   void Naturel::operator-=(const Naturel &b) {
321     if (chiffres.size() < b.chiffres.size())
322       throw runtime_error("Soustraction impossible");
323     bool retenue = false;
324     int i = 0;
325     while (i < int(b.chiffres.size())) {
326       chiffres[i] -= b.chiffres[i] + retenue;
327       retenue = (chiffres[i] < 0);
328       if (retenue) chiffres[i] += 10;
329       i++;
330     }
331     while (i < int(chiffres.size()) and retenue) {
332       chiffres[i] -= retenue;
333       retenue = (chiffres[i] < 0);
334       if (retenue) chiffres[i] += 10;
335       i++;
336     }
337     if (retenue)
338       throw runtime_error("Soustraction impossible");
339     normalise();
340   }
```

4. Surchargez l'opérateur -. Testez cette fonction.

```
Naturel Naturel::operator-(const Naturel& b) const {
  Naturel res{*this};
  res -= b;
  return res;
}

TEST_CASE("operator -"){
  vector<Naturel> testsample = {N0, N1, N2, N1000, a, b};
  for (Naturel n : testsample) {
    CHECK(n - N0 == n);
    CHECK(n - n == N0);
  }
  for (int i = 0; i < int(testsample.size()); i++)
    for (int j = 0; j < int(testsample.size()); j++) {
      Naturel a = testsample[i];
      Naturel b = testsample[j];
      if (i < j) {
        CHECK_THROWS_AS(a - b, runtime_error);
      } else {
        CHECK((a - b) + b == a);
      }
    }
}
```

5. Pour vérifier "l'efficacité" de notre méthode, nous allons regarder quelle est la plus grande valeur qu'un long int peut prendre en comparaison avec nos objets de la classe Naturel. En effet, les long int ont une taille limitée en mémoire, et ne peuvent pas avoir une valeur trop grande. Écrivez une boucle qui en commençant à 1, à chaque étape, double le long int et le naturel et les affiche. Observez le résultat.

```
int main(int argc, const char** argv) {
  doctest::Context context(argc, argv);
  int test_result = context.run();
  if (context.shouldExit()) return test_result;
        //
  long int ni = 1;
  Naturel n{{1}};
  for (int i=1; i<100; i++) {
    n = n + n;
    ni = ni + ni;
    cout << i << " : " << ni << " " << n << endl;
  }
  //
  return 0;
}
```

La bibliothèque GMP fonctionne selon le même principe, sauf que c'est un gros gaspillage de place mémoire et de temps que de calculer en base 10 en ne stockant qu'un seul chiffre dans chaque case du vecteur. On pourrait faire mieux en stockant par exemples 3 chiffres, c'est à dire en calculant en base 1000. Pour optimiser au maximum, GMP calcule en base 2^{64}.

▸ Exercice 3. (Les tours de Hanoi)

Le jeu des tours de Hanoi (voir `https://fr.wikipedia.org/wiki/Tours_de_Hano%C3%AF`) est un puzzle de réflexion dans lequel on déplace des disques de diamètres différents sur trois piquets (les tours). On a une tour de départ sur laquelle tous nos disques seront installés dans l'ordre décroissant de bas en haut (les plus grands en bas les plus petits en haut). On a une tour d'arrivée sur laquelle on veut reproduire la pyramide de disques de la tour de départ.

Pour bouger les disques, il y a deux règles :
— On ne peut déplacer qu'un disque à la fois ;
— On ne peut placer un disque que sur un disque plus grand ou un emplacement vide.
Dans cet exercice, nous allons modéliser les tours de Hanoi, mais nous n'allons pas modéliser leur résolution.

Vous modifierez le fichier `hanoi.cpp`. On vous y a fourni une surcharge de l'opérateur << pour les tableaux et pour les vecteurs.

1. Écrivez une classe Hanoi. Elle aura pour attributs :
   — le nombre de disques
   — un vecteur de la position des disques (la case d'indice i contient l'indice du piquet sur lequel se trouve le disque i) ; Par exemple `{0,1,1,2}` signifie que le disque zéro est sur le piquet 0, les disques 1 et 2 sur le piquet 1 et le disque 3 sur le piquet 2).
   — un vecteur de la position "objectif" des disques (exemple : si on veut que les quatre disques soient sur le piquet 1 : `{1,1,1,1}`)
   — le nombre de mouvements depuis le début.

✂ ----------

```
26 struct Hanoi {
27   int nb_disques, nb_mvmts;
28   //
29   vector<int> statut;
30   vector<int> objectif;
31   // constructeur
32   Hanoi(int nb_disques);
33   //
34
35   array<vector<int>, 3> piquets_statut() const;
36   //
37   /** reinitialiser le plateau **/
38   void reinitialise();
39   void bouge_disque(int deb, int fin);
```

```
40    bool gagne() const;
41    //
42  };
```

2. Écrivez le constructeur de la classe Hanoi qui prend en paramètre le nombre de disques avec la syntaxe `Hanoi::Hanoi(int nb)`, et qui construit l'état initial du jeu.

```
50  //
51  Hanoi::Hanoi(int nb) :
52      nb_disques{nb},
53      nb_mvmts{0},
54      statut(nb, 0),
55      objectif(nb, 1) {
56  }
57  //
```

3. Écrivez une méthode `reinitialise` permettant de réinitialiser le "plateau" (un élément de la classe Hanoi).

```
61  //
62  void Hanoi::reinitialise() {
63    nb_mvmts = 0;
64    statut = vector<int>(nb_disques, 0);
65  }
66  //
```

4. Écrivez une méthode `piquets_statut` permettant d'obtenir le tableau de vecteurs trié des disques sur chaque piquet (par exemple pour la situation : piquet 0 : 5 4 3, piquet 1 : 2, piquet 2 : 1 0, la méthode retournera le tableaux de vecteur `{{3,4,5},{2},{0,1}}`).

```
70  //
71  // get the statut of the disks on the peg
72  array<vector<int>, 3> Hanoi::piquets_statut() const {
73    //
74    array<vector<int>, 3> res;
75    for (int j = 0; j < nb_disques; j++) {
76      res[statut[j]].push_back(j);
77    }
78    return res;
79  //
80  }
```

5. Écrivez une méthode `void Hanoi::bouge_disque(int deb, int fin)` pour bouger un disque. Cette méthode prendra en paramètres le piquet de départ et le piquet d'arrivée. Bien sûr, on signalera une exception si le mouvement n'est pas valide.

```
110 //
111 void Hanoi::bouge_disque(int deb, int fin) {
112   //
113   if (deb < 0 or deb >= 3)
114     throw invalid_argument("Pas de piquet no " + to_string(deb));
115   if (fin < 0 or fin >= 3)
116     throw invalid_argument("Pas de piquet no " + to_string(fin));
117   array<vector<int>, 3> piquet_sorted = piquets_statut();
118   if (piquet_sorted[deb].size() == 0) {
119     throw invalid_argument("Pas de disque en position " + to_string(deb));
120   } else {
121     if (piquet_sorted[fin].size() != 0 and
122         piquet_sorted[fin][0] < piquet_sorted[deb][0]) {
123       throw invalid_argument("Mouvement interdit !");
124     } else {
125       nb_mvmts += 1;
126       statut.at(piquet_sorted[deb][0]) = fin;
127     }
128   }
129 }
130
131 TEST_CASE("move disk and reinitialiser") {
132   Hanoi a(5);
133   Hanoi b = a;
134   b.bouge_disque(0, 1);
135   array<vector<int>, 3> piquet_a = a.piquets_statut();
136   array<vector<int>, 3> piquet_b = b.piquets_statut();
137   array<vector<int>, 3> tab_a = {{{{0, 1, 2, 3, 4}}, {}, {}}};
138   array<vector<int>, 3> tab_b = {{{{1, 2, 3, 4}}, {{0}}, {}}};
139   CHECK(piquet_a != piquet_b);
140   CHECK(piquet_a == tab_a);
141   CHECK(piquet_b == tab_b);
142   b.reinitialise();
143   piquet_b = b.piquets_statut();
144   CHECK(piquet_a == piquet_b);
145 }
146 //
```

6. Testez cette fonction ainsi que la fonction `reinitialise`.
7. Écrivez une méthode `gagne` permettant de savoir si l'état du plateau est l'état "objectif", donc si on a bien gagné.

```
151 //
152 // est ce que le statut est egal a l'objectif ?
153 bool gagne(Hanoi h) { return h.statut == h.objectif; }
154
155 TEST_CASE("gagne") {
156   Hanoi a(2);
157   Hanoi b = a;
158   b.bouge_disque(0, 2);
159   b.bouge_disque(0, 1);
160   b.bouge_disque(2, 1);
161   CHECK(gagne(a) == false);
162   CHECK(gagne(b) == true);
163 }
164 //
```

8. Écrivez le programme principal avec une boucle qui affiche l'état du jeu et demande à l'utilisateur deux entiers décrivant les piquets de début et fin du mouvement qu'il souhaite effectuer.

```
168 int main(int argc, const char **argv) {
169   doctest::Context context(argc, argv);
170   int test_result = context.run();
171   if (context.shouldExit()) return test_result;
172
173   //
174   // test d'affichage
175   int val;
176   cout << "Entrez un nombre de disques : ";
177   cin >> val;
178   Hanoi h(val);
179   int dep;
180   int fin;
181   while (gagne(h) == false) {
182     cout << h << endl;
183     cout << "Entrez les piquet de départ et d'arrivée : ";
184     cin >> dep >> fin;
185     try {
186       h.bouge_disque(dep, fin);
187     } catch (invalid_argument &e) {
188       cout << "Mouvement impossible : " << e.what() << endl;
189     }
190   }
191   cout << h << endl;
192   cout << "Gagné en " << h.nb_mvmts << " mouvements !" << endl;
193   //
194   return 0;
195 }
```

9. Vous pouvez maintenant essayer de résoudre le casse-tête !