université PARIS-SACLAY

Correction partiel de Programmation Modulaire

—Licence MI/IM - Info 3—

Aucun document n'est autorisé à part la fiche résumé de $C++$, où vous pouviez consigner des notes manuscrites personnelles au verso. Tous les exercices sont indépendants. Même si l'on ne sait pas répondre à une question, on peut utiliser la réponse dans la suite de l'exercice. Une grande importance sera accordée à la qualité de la rédaction (lisibilité, indentation, ...).
Le barème est indicatif et pourra changer à la correction.
Durée : 2h00.

▸ Exercice 1. (Question de cours) – sur 3 points –

1. Que peut-on faire comme opérations sur une variable de type `struct` si l'on n'a surchargé aucun opérateur ?

```
- l'affectation,
- l'accès à un champs par notation pointée,
- le calcul de l'adresse (réponse non exigée).
```

2. Déclarer un type énuméré pour représenter les saisons (printemps été automne hiver).

```
enum class Saison {printemps, ete, automne, hiver};
// Alternative
// enum struct Saison {printemps, ete, automne, hiver};
```

3. On veut écrire une fonction `suivante` qui reçoit en paramètre une saison et qui retourne la saison suivante. On demande d'écrire *deux* fonctions, l'une utilisant l'instruction `switch`, l'autre le codage des types énumérés avec les entiers.

```
Saison suivante(Saison s) {
  return Saison((int(s) + 1) % 4);
}

Saison suivanteSwitch(Saison s) {
  switch(s) {
    case Saison::printemps : return Saison::ete;
    case Saison::ete : return Saison::automne;
    case Saison::automne : return Saison::hiver;
    case Saison::hiver : return Saison::printemps;
  }
}
```

▸ **Exercice 2. (Nombres quadratiques)** – sur 11 points –

On veut représenter de manière exacte et sans approximation les nombres de la forme $a + b\sqrt{2}$ où a et b sont des entiers relatifs. L'ensemble ce ces nombres est noté $\mathbb{Z}[\sqrt{2}]$. Par exemple, les quatre nombres suivant -4, $2\sqrt{2}$, $-5\sqrt{2}$, $2-4\sqrt{2}$ appartiennent à $\mathbb{Z}[\sqrt{2}]$, mais pas $1+\frac{\sqrt{2}}{2}$, car les coefficients ne sont pas des entiers, ni $2+\sqrt{3}$ car on a pris la racine carré de 3 au lieu de 2. Comme $\sqrt{2}$ est un nombre irrationnel, on peut montrer que l'écriture $a+b\sqrt{2}$ est unique, c'est-à-dire que les deux nombres $a+b\sqrt{2}$ et $a'+b'\sqrt{2}$ sont égaux si et seulement si $a=a'$ et $b=b'$.

Note : les nombres dans $\mathbb{Z}[\sqrt{2}]$ se comporte de manière très similaires à des nombres complexes si l'on remplace $\sqrt{2}$ par i.

Dans ce qui suit :

— On ignore les dépassements de capacité sur les entiers en faisant comme si l'on pouvait stocker des entiers arbitrairement grands dans un `int`.

— Sauf mention explicite dans la question, on s'interdit d'utiliser les types `float` où `double`, car ils ne permettent de manipuler que des approximations.

1. Déclarer une structure `NombreQ` contenant deux champs `a` et `b` permettant de représenter les nombres appartenant à $\mathbb{Z}[\sqrt{2}]$.

```
struct NombreQ {
    int a, b;
};
```

2. Surcharger l'opérateur d'égalité pour les `NombreQ`.

```
bool operator==(NombreQ x, NombreQ y) {
    return x.a == y.a and x.b == y.b;
}
```

3. Surcharger l'opérateur de multiplication pour les `NombreQ`.

```
NombreQ operator*(NombreQ x, NombreQ y) {
    return {
        x.a * y.a + 2 * x.b * y.b,
        x.a * y.b + x.b * y.a };
}
```

4. En utilisant l'infrastructure `doctest` écrire un cas de test pour l'opérateur de comparaison. Le cas de test devra comporter au moins 4 vérifications et tester le mieux possible la fonction.

```
TEST_CASE("operator==") {
    CHECK(NombreQ {0, 1} == NombreQ {0, 1});
    CHECK(NombreQ {1, 1} == NombreQ {1, 1});
    CHECK_FALSE(NombreQ {0, 1} == NombreQ {1, 1});
    CHECK_FALSE(NombreQ {1, 1} == NombreQ {1, 0});
}
```

5. En utilisant l'infrastructure `doctest` écrire un cas de test pour l'opérateur de multiplication.

```
TEST_CASE("operator*") {
    CHECK(NombreQ {3, 2} * NombreQ {0, 0} == NombreQ {0, 0});
    CHECK(NombreQ {3, 2} * NombreQ {1, 0} == NombreQ {3, 2});
    CHECK(NombreQ {1, 0} * NombreQ {3, 2} == NombreQ {3, 2});
    CHECK(NombreQ {3, 2} * NombreQ {1, 1} == NombreQ {7, 5});
}
```

6. Écrire une procédure `mulsq2` qui prends en paramètre une variable de type `NombreQ` et qui change le contenu de cette variable en multipliant le nombre correspondant par $\sqrt{2}$. Par exemple, si la variable `x` contient le nombre $3 + 2\sqrt{2}$, après l'exécution de `mulsq2` sur `x`, la nouvelle valeur sera $4 + 3\sqrt{2}$.

```
void mulsq2(NombreQ &x) {
    int tmp = x.a;
    x.a = 2 * x.b;
    x.b = tmp;
}
```

7. La *norme quadratique* d'un tel nombre est l'entier défini par la formule $N(a+b\sqrt{2}) := a^2 - 2b^2$. Écrire une fonction `norme` qui renvoie la norme d'un `NombreQ`.

```
int norme(NombreQ x) {
    return x.a * x.a - 2 * x.b * x.b;
}
```

On veux maintenant remplacer la structure `NombreQ` par une classe avec trois constructeurs :

— un constructeur par défaut qui construit le nombre 0 ;
— un constructeur qui convertit un entier en `NombreQ` ;
— un constructeur qui construit un `NombreQ` à partir de ses deux parties.

La classe devra également contenir deux méthodes équivalentes aux fonctions `mulsq2` et `norme`.

8. Écrire la *déclaration complète* de la classe. Seule la définition du constructeur par défaut sera mise en ligne.

```
struct NombreQ {
    int a, b;

    NombreQ() : a{0}, b{0} {}
    NombreQ(int n);
    NombreQ(int _a, int _b);

    void mulsq2();
    int norme() const;
};
```

9. Écrire la *définition* du constructeur à partir d'un seul entier.

```
NombreQ::NombreQ(int n) : a{n}, b{0} {};
```

10. Écrire les *définitions* des méthode `norme` et `mulsq2`.

```
int NombreQ::norme() const {
    return a * a - 2 * b * b;
}
void NombreQ::mulsq2() {
    int tmp = a;
    a = 2 * b;
    b = tmp;
}
```

11. Écrire un test pour chacune de ces deux méthodes. C'est juste pour vérifier que vous savez les appeler correctement, il est inutile de mettre les `TEST_CASE` autour.

```
    CHECK((NombreQ {4, 3}).norme() == -2);
    NombreQ x = {3, 2};
    x.mulsq2();
    CHECK(x == NombreQ {4, 3});
```

▸ **Exercice 3. (Gestion d'équipe sportive)** – 6 points

On considère la gestion d'équipes de joueurs d'un sport collectif pour laquelle on mémorise différentes informations. Pour un joueur, on connaît son nom, son numéro de licence (un entier strictement positif, unique sur l'ensemble des joueurs), son équipe et son numéro de maillot (un entier compris entre 1 et 25) dans l'équipe. Chaque équipe a un nom et un effectif (avec donc au plus 25 joueurs). Au sein de l'équipe, chaque joueur est désigné par son numéro de licence.

1. Complétez les définitions suivantes en précisant les types des champs :

```
1  struct Joueur {
2    ...       nomJ;      // le nom du joueur
3    ...       licence;   // son numéro de licence
4    ...       equipe;    // l'indice de son équipe d'appartenance
5    ...       maillot;   // son numéro de maillot dans son équipe
6  };
7
8  struct Equipe {
9    ...       nomE;      // le nom de l'équipe
10   ...       licences;  // les numéros de licence des joueurs de l'équipe
11 };
12
13 struct Championnat {
14    vector<Joueur> joueurs;  // tous les joueurs licenciés
15    vector<Equipe> equipes;  // toutes les équipes enregistrées
16    int dernier;             // dernier numéro de licence attribué (on
17                             // suppose qu'on met 0 à l'initialisation)
18 };
```

2. Écrire une fonction qui prend en entrée une référence sur un championnat et une chaîne de caractères et qui recherche s'il existe dans le championnat une équipe qui porte ce nom. La fonction renvoie alors l'indice de l'équipe dans le vecteur des équipes du championnat. S'il n'existe aucune telle équipe la fonction doit renvoyer -1.

```
int chercheEquipe(const Championnat &c, string nomE) {
```

```
  for (int i = 0; i < c.equipes.size(); i++) {
    if (c.equipes[i].nomE == nomE) { return i; }
  }
  return -1;
}
```

Dans la suite on pourra supposer qu'on dispose d'une fonction de recherche d'un joueur à partir d'un numéro de licence, qui renvoie l'indice du joueur dans le vecteur de tous les joueurs et -1 s'il n'existe aucun tel joueur. On ne demande pas d'écrire la fonction.

```
int chercheJoueur(const Championnat &c, int licence);
```

3. Écrire une fonction qui prend en entrée une référence sur un championnat et un vecteur de noms d'équipes à admettre. Votre fonction doit créer et ajouter au championnat des équipes avec ces noms, en rejetant la création des équipes dont le nom serait déjà pris. La fonction renvoie le nombre d'équipes effectivement ajoutées.

```
int ajouter(Championnat &c, vector<string> lesNoms) {
```

```
  int ajoutes = 0;
  for (int i = 0; i < lesNoms.size(); i++) {
    string n = lesNoms[i];
    if (chercheEquipe(c, n) == -1) {
      ajoutes ++;
      c.equipes.push_back({ n, { } });
    } // si le nom est deja pris => on ignore l'equipe.
  }
  return ajoutes;
}
```

4. Écrire une fonction pour inscrire un nouveau joueur dans le championnat en l'ajoutant à l'équipe dont on fournit le nom, avec le numéro de maillot indiqué. L'équipe doit exister et le numéro de maillot doit être disponible dans cette équipe. On attribue automatiquement un numéro de licence au joueur. La fonction doit lever une exception `invalid argument` s'il n'a pas été possible d'enregistrer le joueur dans l'équipe ; la raison peut être l'une des suivantes :
   — l'équipe n'existe pas,
   — le numéro de maillot n'est pas valide,
   — l'équipe est déjà complète
   — ou le maillot est déjà pris par un autre joueur.

```
void inscrire (Championnat &c, string nomJ, string nomE, int num) {
```

```
  int indE = chercheEquipe(c, nomE);
  if (indE == -1 or num < 1 or num > 25) {
      throw invalid_argument("Numéro invalide");
  }
  Equipe e = c.equipes[indE];  // ATTENTION : Copie !
  if (e.licences.size() == 25) {
      throw invalid_argument("Équipe complète");
  }
  // verifier le numero de maillot
  for (int j = 0; j < e.licences.size(); j = j+1) {
    if (maillotJoueur(c, e.licences[j]) == num) {
      throw invalid_argument("Maillot déjà pris");
    }
  }
  Joueur j = { nomJ, c.dernier+1, num, indE };
  c.dernier ++;
  c.joueurs.push_back(j);
  e.licences.push_back(c.dernier);
  c.equipes[indE] = e;  // Range la copie dans le tableau des équipes
  // On pourrait éviter avec une référence
}
```