université PARIS-SACLAY

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 –

Répondre en une ou deux phrases aux questions suivantes :

1. Comment le compilateur fait-il la différence entre la **déclaration** d'une fonction usuelle et celle d'une fonction membre (aussi appelée méthode) ?
2. Même question pour la **définition**.
3. Qu'est-ce qu'une **conversion implicite** ? Donner deux exemples.
4. Donner un exemple où l'on a besoin de faire une **conversion explicite**.

▸ Exercice 2. (Représentation des nombres en virgule fixe) – sur 10 points –

Dans certaines applications, on a besoin de contrôler très précisément les chiffres après la virgule des nombres. Par exemple, les banques doivent considérer des montants ayant exactement deux chiffres après la virgule. Dans cet exercice, le nombre de chiffres après la virgule sera une constante nommée `NB_CHIFFRES`, égale à 6 dans les exemples ci-dessous. On va utiliser les deux définitions de constantes suivantes :

```
const int NB_CHIFFRES = 6;
const int P10NB = pow(10, NB_CHIFFRES);  // L'entier 10 à la puissance NB_CHIFFRES
```

Les nombres seront représentés par une structure contenant deux entiers où l'on mettra dans `avant` les chiffres avant la virgule et dans `après` les autres (exactement `NB_CHIFFRES`). Le nombre représenté par la variable `v` sera

$$x = \texttt{v.avant} + \texttt{v.apres}/10^{\texttt{NB\_CHIFFRES}} = \texttt{v.avant} + \texttt{v.apres}/\texttt{P10NB}\,.$$

Pour respecter cette relation, si x est négatif, les **deux champs** `avant` et `après` devront être négatifs tous les deux. On respectera donc les invariants suivants :

- $-10^{\texttt{NB\_CHIFFRES}} < \texttt{apres} < 10^{\texttt{NB\_CHIFFRES}}$
- `avant` et `apres` ont toujours le même signe.

Par exemple :

— 3,141562 sera représenté par `avant` $= 3$ et `apres` $= 141592$.

— $12{,}0054 = 12{,}005400$ sera représenté par `avant` $= 12$ et `apres` $= 5400$.

— $-35{,}202314$ sera représenté par `avant` $= -35$ et `apres` $= -202314$.

1. Écrire la déclaration de la structure `Nombre` décrite précédemment.
2. Écrire une fonction `estCorrect` qui prend en paramètre un nombre et qui renvoie `true` si le nombre vérifie bien les invariants ci-dessus et `false` sinon.
3. Proposer, en utilisant l'infrastructure `doctest`, un cas de test comportant plusieurs tests de la fonction `estCorrect` ci-dessus. On fera attention à bien tester tous les comportements.
4. Surcharger l'opérateur d'égalité pour les `Nombre`.
5. On rappelle que

   ```
   cout << setw(5) << setfill('0') << n;
   ```

   permet d'afficher le nombre `n` sur 5 caractères en remplissant avec des 0 si besoin. Ainsi si $n = 12$, l'affichage sera `00012`. Surcharger l'opérateur d'affichage pour le type `Nombre`.
6. Écrire une fonction `abs` qui renvoie la valeur absolue d'un `Nombre`. Le résultat renvoyé sera de type `Nombre`.
7. Surcharger l'opérateur d'addition pour le type `Nombre`. Le résultat renvoyé sera de type `Nombre`.

On veut maintenant que `Nombre` soit une classe, qui contienne

— un constructeur à partir de deux entiers représentant les chiffres avant et après la virgule ;
— un constructeur par défaut construisant le nombre 0 ;
— une méthode `abs` (correspondant à la fonction `abs` précédente).

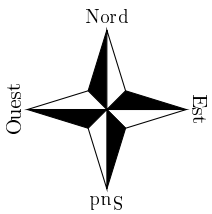
8. Écrire la déclaration de la classe.
9. Écrire la définition des deux constructeurs. Pour le constructeur à partir de deux entiers, si les invariants ne sont pas vérifiés, on lèvera une exception `invalid_argument`.
10. Écrire la définition de la méthode `abs`.

▸ **Exercice 3. (Jeu de Karkassohn)** – sur 7 points –

Le jeu de Karkassohn (qui ressemble à un autre jeu que vous connaissez peut-être par ailleurs) est une sorte de puzzle où l'on pose des pièces carrées sur une grille. Les quatre bords de la pièce sont orientés chacun selon une direction Nord, Est, Sud ou Ouest. Il peuvent être occupés soit par un champ, soit par une route, soit par une forêt. De plus, le milieu de la pièce peut être occupé par une ville. On a donc déclaré les types suivants :

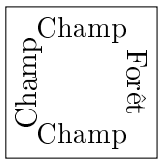
```
enum class Bord { Champ, Route, Foret };
enum class Dir { Nord, Est, Sud, Ouest };

struct Piece {
  array<Bord, 4> bords;
  bool ville;
};
```

Voici quelques exemples de pièces :

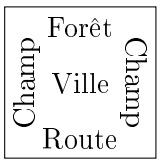


pièce 1

pièce 2



pièce 3



pièce 4

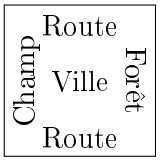
La pièce 1 a par exemple une forêt au nord, une route à l'est, un champ au sud et une route à l'ouest, mais pas de ville. La pièce 2 a les mêmes bord que la pièce 1 et une ville.

Dans le tableau `bords` d'une pièce, on rangera les bords dans l'ordre indiqué par l'énumération `Dir`. Par exemple, la pièce 1 sera codée par le tableau

0	1	2	3
Forêt	Route	Champ	Route

1. Déclarer et initialiser en une seule instruction une variable nommée `piece1` de type `Piece` pour représenter la pièce 1 ci-dessus.
2. Écrire une fonction `opposee` qui prend une `Dir` et qui renvoie la direction opposée. Par exemple, la direction opposée de Sud est Nord. On demande d'écrire cette fonction en utilisant un `switch`.
3. Écrire une fonction `bordPiece` qui prend une `Piece` et une `Dir` et qui renvoie le bord de la pièce dans la direction. On demande d'utiliser le codage des types énumérés par un entier sans écrire ni condition ni `switch`.
4. Écrire une fonction `tourne90` qui prend une pièce et qui renvoie la pièce tournée d'un quart de tour dans le sens des aiguilles d'une montre. On utilisera une boucle pour les 4 directions en s'interdisant d'écrire 4 fois un code similaire. Voici un exemple :

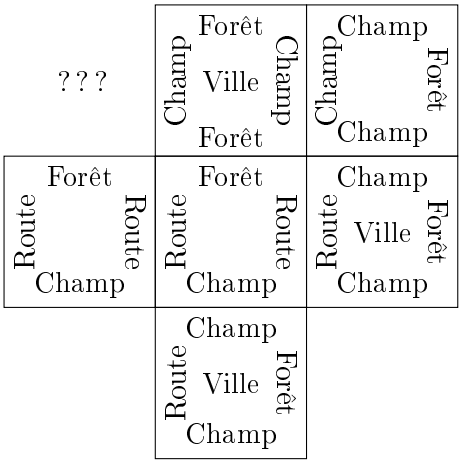
pièce 2

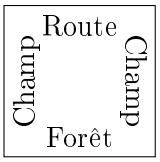


pièce 2 tounée

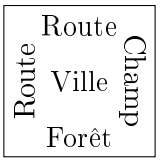
Règle de placement des pièces

La règle du jeu indique que l'on peut placer deux pièces côte à côte uniquement si les bords adjacents sont identiques. De plus, il est interdit de placer deux villes côte à côte. Voici un exemple de placement autorisé :

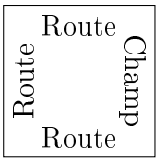




A : oui



B : non



C : non

On peut, de plus, placer la pièce A dans l'emplacement marqué « ? ? ?», mais ni la pièce B (car on aurait deux villes côte à côte) ni la pièce C (car dans la direction Sud, on aurait une Route en face d'une Forêt).

5. Écrire une fonction `bool estCompatible(Piece p1, Piece p2, Dir d)` qui retourne `true` si l'on peut placer la pièce `p2` dans la direction `d` de la pièce `p1` (sans la tourner) en respectant les règles, et `false` sinon.

Modélisation du plateau de jeu

Le plateau de jeu est une grille carrée dont la longueur du bord est donnée par la constante ci-dessous :

```
const int TAILLEGRILLE = 10;
```

On représente une case de la grille par le type `Case` suivant

```
struct Case {
  int joueur;
  Piece p;
};
```

où `joueur` contient le numéro du joueur qui a placé la pièce dans la case. La valeur -1 signifie que la case est vide.

6. Déclarer un type `Grille` pour représenter le plateau de jeu.
7. Écrire une fonction `bool okGrille(const Grille &gr)` qui teste si la grille respecte les règles du jeu. Indication : si la pièce en position $(3, 4)$ est compatible avec la pièce en position $(3, 3)$ (dans la direction Sud), alors automatiquement la pièce en position $(3, 3)$ est compatible avec la pièce en position $(3, 4)$ dans la direction Nord. Il n'est pas utile de tester les deux compatibilités. La même chose est vraie dans le sens Est Ouest.