NOM :

PRÉNOM :

PLACE :

université PARIS-SACLAY

Correction examen 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,...).

Si vous avez besoin de place, il y a une page libre à la fin du document.

Le barème est indicatif et pourra changer à la correction.

Durée : 2h00.

▸ Exercice 1. (Questions de cours – 2 pt.)

Répondre en deux ou trois phrases aux questions ci-dessous :

1. Quel est la différence entre les mots clé `class` et `struct` ?

Dans une déclaration avec class, les déclaration sont par défaut privées, dans une déclaration avec struct les déclarations sont publiques.

.

2. Qu'est ce qu'un attribut privé ?

C'est un attribut auquel seules les méthodes de la classe peuvent accéder.

.

3. Quel est l'intérêt des attributs privés ?

Encapsulation, intégrité des données, invariants.

.

▸ Exercice 2. (Jeu de Quarto – 16 pt.)

Quarto est un jeu de société combinatoire créé par Blaise Muller. L'objectif du jeu est d'aligner quatre pièces ayant au moins un point commun entre elles. Mais chaque joueur ne joue pas ce qu'il veut, c'est son adversaire qui choisit pour lui.

Les seize pièces du jeu, toutes différentes, possèdent chacune quatre caractéristiques distinctes : Grande ou Petite, Ronde ou cArrée, Claire ou Foncée, pleiNe ou Trouée. Chaque joueur à son tour choisit et donne une pièce à l'adversaire, qui doit la jouer sur une case libre. Le gagnant est celui

PLACE :

qui, avec une pièce reçue, crée un alignement de quatre pièces ayant au moins une caractéristique commune.

Chaque combinaison de caractéristiques est représentée (par exemple : grand, rond, clair et troué) et chacune n'est représentée qu'une seule fois. Il y a donc $2^4 = 16$ pièces. Voici une photo montrant 5 des pièces :

La pièce à gauche à les caractéristiques Grand, Rond, Foncé, pleiNe. On la notera GRFN en abrégé. Les autres pièces de la photo sont GACT, PACT, PRFT, PACN (cArrée est représenté par A car C signifie Claire, de même pleiNe est représenté par N car P signifie Petite). Les pièces sont codées par un tableau de 4 booléens comme suit :

— Grand, Rond, Clair, pleiN sont codé par `true` (T en abrégé) ;

— Petit, cArré, Foncé, Troué sont codé par `false` (F en abrégé).

Ainsi les 5 pièces du dessin sont codées de gauche à droite par les tableaux abrégés TTFT, TFTF, FFTF, FTFF, FFTT.

On considère la classe `Piece` suivante qui code une pièce de jeu.

```
class Piece {
 public:
  Piece(std::array<bool, 4> c) : car_{c} {};

  bool caract(int i) const { return car_.at(i); }
 private:
  std::array<bool, 4> car_;
};
```

La classe contient un attribut privé `car_` pour le tableau de 4 booléens, un constructeur qui prend en paramètre un tel tableau, et une méthode `bool caract(int i)` qui retourne la i-ème caractéristique.

1. Coder la surcharge de l'opérateur `==` pour le type `Piece` (en dehors de la classe).

```
// le tableau est un attribut privé on ne peut pas faire l'égalité
// entre les deux tableau
bool operator==(Piece p1, Piece p2) {
  for (int i=0; i<4; i++) {
    if (p1.caract(i) != p2.caract(i)) return false;
  }
  return true;
}
```

2. En utilisant `doctest`, tester la surcharge de l'opérateur `==` sur des exemples pertinents.

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

PLACE :

3. Coder la surcharge de l'opérateur d'affichage pour afficher une pièce par sa chaîne de caractères abrégée (par exemple «GRFN») :

```
array<string, 4> stringofcar {{"PG", "RA", "CF", "NT"}};

std::ostream &operator<<(std::ostream &out, const Piece &p) {
  for (int i=0; i<4; i++) {
    out << stringofcar[i][p.caract(i)];
  }
  return out;
}
```

4. Déclarer une classe `Case` pour coder une case de jeu. Cette classe doit contenir deux attributs privés : un booléen `vide` et une Pièce `p`. La classe doit aussi contenir le constructeur par défault qui crée une case vide, et les quatre méthodes suivante :
   - `pose` : pose une pièce donnée sur la case
   - `enleve` : enlève la pièce de la case
   - `piece` : renvoie la pièce qui est dans la case
   - `estVide` : répond si la case est vide ou non.

   N'oubliez pas de mettre des `const` pour déclarer les méthodes constantes.
   Dans cette question, on veut juste les déclarations des méthodes et la définition en ligne du constructeur par défaut. Dans la suite on vous demandera de définir les méthodes.

```
class Case {

 public:

  Case() : vide_{true}, p_{{0,0,0,0}} {}

  Piece piece() const;
  bool estVide() const;

  void pose(Piece);
  void enleve();

 private:

  bool vide_;
  Piece p_;

};


//
```

5. Coder la méthode `pose`, qui pose une pièce si la case est vide, sinon lève une exception de type `runtime_error` avec le message "case occupee".

```
void Case::pose(Piece p) {
  if (not vide_) {
    throw runtime_error("Case deja occupee");
  }
  p_ = p;
  vide_ = false;
}
```

PLACE :

6. Coder la méthode `enleve`, qui enlève la pièce de la case si la case est occupée, sinon lève une exception de type `runtime_error` avec le message “case vide”.

```
void Case::enleve() {
  if (vide_) {
    throw runtime_error("Case vide");
  }
  vide_ = true;
}
```

7. Coder la méthode `piece`, qui renvoie la pièce contenue dans la case si la case est occupée, sinon lève une exception de type `runtime_error` avec le message “case vide”.

```
Piece Case::piece() const  {
  if (vide_) {
    throw runtime_error("Case vide");
  }
  return p_;
}
```

8. Coder la méthode `bool estVide()`.

```
bool Case::estVide() const { return vide_; }
```

PLACE :

On dispose d'un type `EnsPiece` qui permet de contenir un ensemble de pièces. Son interface est la suivante :

```
EnsPiece();                    // Constructeur par défaut: construit un ensemble vide
void ajoute(Piece o);           // Ajoute une Piece à l'ensemble
bool supprime(Piece o);         // Retire une Piece à l'ensemble
Piece element() const;          // Retourne une Piece de l'ensemble
bool estVide() const;           // Teste si l'ensemble est vide
int taille() const;             // retourne la taille de l'ensemble
bool contient(Piece o) const;   // Teste si une Piece est dans l'ensemble
```

On ne demande pas d'écrire les fonctions du type EnsPiece, mais seulement de les utiliser correctement.

9. Déclarer une classe `Grille` pour coder une Grille de jeu. Cette classe devra contenir un attribut privé `tab` pour représenter une grille 4x4, un constructeur et les méthodes suivantes :
   - `G.getCase(lig, col)` : renvoie la case de la grille `G` de coordonnée `lig`, `col`.
   - `G.setCase(lig, col, c)` : positionne dans `G` la case `c` aux coordonnées `lig, col` ;
   - `G.ligne(i)` : renvoie l'ensemble des pièces de la ième ligne de `G` ;
   - `G.colonne(i)` : renvoie l'ensemble des pièces de la ième colonne de `G` ;
   - `G.gagne()` : renvoie si la grille `G` est gagnante.

   Attention vous devez ajouter ou non des `const` pour chaque méthode.

```
// Une classe pour la Grille

class Grille {

  public:

    Grille() : tab_{} {};

    Case getCase(int lig, int col) const;
    void setCase(int lig, int col, const Case &);

    EnsPiece ligne(int i) const;
    EnsPiece colone(int i) const;
    bool gagne() const;

 private:

    array<array<Case, 4>, 4> tab_;

};
```

10. Coder le getter `getCase`, qui retourne une copie de la case de la ligne `lig` et la colonne `col`.

```
void checkLigCol(int lig, int col) {
  if (lig < 0 or lig >= 4 or col < 0 or col >= 4) {
    throw runtime_error("Coord invalide");
  }
}
Case Grille::getCase(int lig, int col) const {
  checkLigCol(lig, col);
  return tab_[lig][col];
}
```

11. Coder le setter `setCase`, qui met à jour la case `lig`, `col` de la grille avec la valeur `c`.

PLACE :

```
void Grille::setCase(int lig, int col, const Case &c) {
  checkLigCol(lig, col);
  tab_[lig][col] = c;
}
```

12. Coder la méthode `ligne`, qui ajoute toutes les cases non vides de la ligne `lig` dans un ensemble de pièces et retourne cet ensemble.

```
EnsPiece Grille::ligne(int lig) const {
  EnsPiece res;
  for (int i=0; i<4; i++) {
    Case c = getCase(lig, i);
    if (not c.estVide()){
      res.ajoute(c.piece());
    }
  }
  return res;
}
```

13. Coder la fonction `bool estGagnant(const EnsPiece &Q)`, qui prend en paramètre en ensemble de pièces `Q` et qui retourne `true` si cet ensemble contient exactement 4 pièces qui ont au moins un point commun entre elles.

```
bool estGagnant(const EnsPiece &Q) {
  if (Q.taille() != 4) return false;
  for (int c=0; c<4; c++) {
    bool gagne = true;
    EnsPiece E = Q;
    Piece p = E.element(); E.supprime(p);
    bool car = p.caract(c);
    while (not E.estVide()) {
      p = E.element(); E.supprime(p);
      if (car != p.caract(c)) gagne = false;
    }
    if (gagne) return true;
  }
  return false;
}
```

14. On suppose que l'on a aussi écrit une fonction `colonne` similaire à `ligne`.
    Coder la fonction `bool Grille::gagne()`, qui teste toutes les lignes et les colonnes, qui retourne `true` si une ligne ou une colonne est gagnante au sens de la précédente question (pour simplifier, on ignore les diagonales par rapport à la règle du jeu).

```
bool Grille::gagne() const {
  for (int i=0; i<4; i++) {
    EnsPiece e = ligne(i);
    if (estGagnant(e)) return true;
  }
  for (int i=0; i<4; i++) {
    EnsPiece e = colone(i);
    if (estGagnant(e)) return true;
  }
  return false;
}
```

PLACE :

▸ **Exercice 3. (Compilation – 2 pt.)**

Pour le jeu de Quarto de l'exercice précédent, on a besoin des classes suivantes :

— `Piece` : piece de jeu

— `Case` : case du jeu

— `EnsPiece` : ensemble de pièces

— `Grille` : grille de jeu

Chacune de ces classes devra être compilée dans un fichier à part. Pour chacune de ces classes on a écrit la déclaration dans un fichier portant le nom de la classe et l'extension `.hpp` et les définitions dans un fichier portant le nom de la classe et l'extension `.cpp`. Par exemple `Piece` est dans `Piece.hpp` et `Piece.cpp`

En outre, on a écrit le programme principal (`main`) dans le fichier `quarto.cpp`.

1. Quelles sont les directives d'inclusion (`include`) en tête du fichier `Grille.hpp` ? On ne demande pas d'écrire les inclusions de la bibliothèque standard (par exemple `iostream`, `vector`).

```
#include"Piece.hpp"
#include"Case.hpp"
#include"EnsPiece.hpp"
```

2. Écrire les deux lignes de `Makefile` qui concernent la compilation de la classe `Grille`. On vous demande d'écrire une version de base sans utiliser de variables. Si vous n'avez pas assez de place en largeur pour une ligne, vous pouvez couper une ligne en deux en terminant chaque ligne qui se prolonge sur la suivante par le caractère "\".

```
Grille.o: Grille.cpp Grille.hpp Piece.hpp Case.hpp EnsPiece.hpp
     g++ -std=c++11 -Wall Grille.cpp -c


--
```