Correction examen de Programmation Modulaire

—Licence MI/IM/MNSI - 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. (Compilation séparée)** – sur 5 points –

Une équipe de développeurs travaille sur des systèmes de calculs. Ils doivent développer une application de calculette capable de manipuler des fractions dont les numérateurs et dénominateurs peuvent être très grands ; Pour ceci, il vont écrire les classes suivantes :

- `Entier` : une classe pour les entiers en précision arbitraire ;
- `Rationel` : une classe pour les fractions dont les numérateurs et dénominateurs sont des `Entier` ;
- `Memoire` : une classe pour gérer la mémoire de la calculatrice ; cette classe contient un tableau de `Rationel` ;

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 `Entier` est dans `Entier.hpp` et `Entier.cpp`

En outre, ils ont écrit le programme principal (`main`) dans le fichier `calc.cpp`.

1. Pour chacun des 7 fichiers, on demande d'écrire les directives d'inclusion (`include`) des fichiers qui ont été développés par l'équipe. On ne demande pas d'écrire les inclusions de la bibliothèque standard (par exemple `iostream`, `vector`).

   ✂ ----------

   Sur 1.5pt.

   - 0.5 pour la syntaxe, 0.25 si `#include <Entier.hpp>` (avec <>).
   - -0.25 par erreur ou fichier manquant
   - -0.5 si l'on inclus un .cpp

```
1 Entier.hpp:
2
3 Entier.cpp:
4   #include "Entier.hpp"
```

```
5
6  Rationel.hpp:
7    #include "Entier.hpp"
8
9  Rationel.cpp:
10   #include "Entier.hpp" // Pas obligatoire, déjà dans le .hpp
11   #include "Rationel.hpp"
12
13 Mémoire.hpp
14   #include "Entier.hpp" // Pas obligatoire, déjà inclus dans Rationel.hpp
15   #include "Rationel.hpp"
16
17 Memoire.cpp
18   #include "Entier.hpp" // Pas obligatoire, déjà dans le .hpp
19   #include "Rationel.hpp" // Pas obligatoire, déjà dans le .hpp
20   #include "Memoire.hpp"
21
22 calc.cpp
23   #include "Entier.hpp" // Pas obligatoire
24   #include "Rationel.hpp" // Pas obligatoire
25   #include "Memoire.hpp"
```

---

2. Quelle directive d'inclusion doit-on mettre pour que l'on puisse manipuler un tableau dans la classe `Mémoire` ?

Sur 0.5pt,

```
1        #include <array>   ou     #include <vector>
```

- 0 si `#include "array"` (avec ").
- 0.25 si `#include <array.hpp>` (avec `hpp`)

---

3. Quelle est la commande qui permet de compiler la classe `Entier`. Quel est le nom du fichier produit ?

Sur 1pt,

```
1        g++ -std=c++11 -Wall -c Entier.cpp
2  ou
3        g++ -std=c++11 -Wall Entier.cpp -c
4  produit un fichier Entier.o
```

- 0.5 pour `g++ -c Entier.hpp`.
- -0.25 si `-o ...`.
- 0.25 pour l'enssemble deux options `-std=c++11 -Wall` (avec `hpp`), aucun point s'il en manque une.
- 0.25 pour `Entier.o`.

---

4. Quelle est la commande qui permet de produire l'exécutable pour la calculatrice ?

✂ ----------

Sur 0.75pt,

```
1         g++ Entier.o Rationel.o Memoire.o calc.o -o calc
2       ou
3         g++ -o calc Entier.o Rationel.o Memoire.o calc.o
```

- 0.25 pour `g++ -o calc`.
- 0.25 si `calc.o` et pas `calc.(ch)pp|`.
- 0.25 pour la liste complète des `.o`

---------- ✂

5. Écrire les deux lignes de `Makefile` qui concernent la compilation de la classe `Memoire`.

✂ ----------

Sur 1.25pt,

```
1  Memoire.o : Memoire.cpp Memoire.hpp Rationel.hpp Entier.hpp
2          g++ -std=c++11 -Wall Memoire.cpp -c
```

- 0.5 pour la syntaxe avec `cible : dep`.
- 0.25 pour dépendance envers `memoire.cpp`
- 0.25 pour dépendance envers `memoire.hpp`
- 0.25 pour les autres dépendances.

---------- ✂

Problème : Réservation de chambres d'étudiants

Le but de ce problème est de participer à la construction d'un système de réservation de chambre dans une résidence pour étudiants. La résidence dispose d'un nombre fixé `nb_chambres` de chambres.

▸ **Exercice 2. (Classe pour représenter les d'étudiants)** – sur 4 points –

Un étudiant est modélisé par la classe `Etudiant`. Quand on crée un étudiant, on connaît son nom (une chaîne de caractères) et son numéro d'étudiant (un entier). Ils ne seront jamais modifiés. En revanche, il n'a pas encore de chambre affectée. On pourra ensuite lui affecter une chambre (le numéro de chambre est un entier).

1. Écrire la déclaration du constructeur de la classe `Etudiant`.

✂ ----------

Sur 0.5pt. Ne pas mettre de point s'il y a `Etudiant::` devant

- -0.25 si l'on passe le numéro de chambre en paramètre.

```
9   Etudiant(string nm, int no);
```

---------- ✂

2. Écrire les déclarations des méthodes qui permettent d'accéder aux nom, numéro et chambre d'un étudiant, ainsi que celles des méthodes qui permettent de les modifier (uniquement pour les attibuts où cela est utile).

✂ ----------

Sur 1.5pt.

- 0.5 sur les `const` aux bons endroits
- 0.5 pour les 3 getter
- 0.5 pour le setter
- -0.5 si l'on a un autre setter
- -0.5 s'il y a un `Etudiant::`

```
13  string get_nom() const;
14  int get_numero() const;
15  int get_chambre() const;
16  void set_chambre(int ch);
```

---------- ✂

3. Dans quel fichier les déclarations des constructeurs et des méthodes d'accès et de modification doivent-elles figurer ? A quel endroit du programme très précisément ?

✂ ----------

Sur 0.5pt.

- 0.25 pour le nom du fichier
- 0.25 pour la partie publique de la classe

Fichier `Etudiant.hpp`, dans la partie publique de la déclaration de la classe Etudiant.

---------- ✂

4. Quand un étudiant n'a pas de chambre, on lui donne le numéro de chambre 0. Déclarer une méthode `est_loge` qui renvoie si l'étudiant à une chambre ou non.

✂ ----------

- 0.25 pour l'entête
- 0.25 pour le const

Sur 0.5pt.

```
19  bool est_loge() const;
```

---------- ✂

5. Définir la méthode `est_loge`.

Sur 0.5pt.

- 0.25 pour l'entête, y compris le const. Je le note deux fois car il doit être dans la déclaration et la définition
- 0.25 pour la logique

```
35 bool Etudiant::est_loge() const { return chambre != 0; }
```

6. À quel endroit et dans quel fichier doit-on placer cette définition ?

Sur 0.5pt.

- 0.25 pour le nom du fichier
- 0.25 pour n'importe où

À n'importe quel endroit du fichier `Etudiant.cpp`.

▸ **Exercice 3. (Représentation des Chambres)** – sur 1 points –

On s'intéresse maintenant à la structure pour représenter les chambres. La résidence comporte 100 chambres réparties sur deux étages. Les chambres de l'étage 1 sont numérotée de 100 à 149. Les chambres de l'étage 2 sont numérotées de 200 à 249. Pour stocker les informations des chambres dans des tableaux, on a besoin d'associer à chaque chambre un *indice* entre 0 et 99. Ainsi les chambres de l'étage 1 correspondent aux indices de 0 à 49 et les chambres de l'étage 2 aux indices de 50 à 99. Par exemple, la chambre 127 a pour indice 27 et la chambre 233 a pour indice 83.

7. Écrire une fonction `indice_chambre` qui renvoie l'indice d'une chambre connaissant son numéro. Si le numéro de la chambre n'est pas valide, on signalera une exception.

Sur 1pt.

- 0.25 pour l'entête
- 0.5 pour la logique
- 0.25 pour l'exception

```
39 int indice_chambre(int chambre) {
40   if (100 <= chambre and chambre <= 149) return chambre-100;
41   if (200 <= chambre and chambre <= 249) return chambre-150;
42   throw invalid_argument("Mauvais numero de chambre");
43 }
```

Dans la suite on suppose écrite une fonction similaire `numéro_chambre` qui renvoie le numéro d'une chambre connaissant son indice.

Voici la partie publique de la déclaration de la classe `Residence` :

```
// Constructeur
Residence();
// Renvoie true si la chambre du numéro donné est libre
bool est_libre(int numero) const;
// Réserve une chambre pour l'étudiant
void reserve(Etudiant &etu);
// Libère la chambre de l'étudiant
void libere(Etudiant &etu);
```

▸ **Exercice 4. (Implémentation simple à un seul tableau)** – sur 5.75 points –

Dans une première partie on représente les chambres par un attribut privé de la classe `Residence`, nommé `chambres`, qui est un tableau d'entiers. Dans la case `i` du tableau on notera le numéro de l'étudiant qui occupe la chambre, ou 0 si la chambre n'est pas occupée.

8. Écrire la déclaration de l'attribut `chambres`. Dire très précisément où cette déclaration doit être écrite.

✂ ----------

Sur 0.75pt.

- 0.25 pour la déclaration
- 0.5 pour la partie privée de la classe

```
72   array <int, 100> chambres;
```

Dans la partie privée de la déclaration de la classe résidence.

---------- ✂

9. Écrire la définition du constructeur de la classe `Residence`.

✂ ----------

Sur 0.75pt.

- 0.25 pour l'entête
- 0.5 pour la logique et l'accès à l'attribut

```
77  Residence::Residence() {
78    for (int i=0; i<100; i++) chambres[i] = 0;
79  }
```

---------- ✂

10. Écrire la définition de la méthode `est_libre`.

✂ ----------

Sur 1.25pt.

- 0.5 pour l'entête
- 0.25 pour l'appel à `indice_chambre`
- 0.5 pour la valeur de retour correcte

```
82 bool Residence::est_libre(int numero) const {
83   return chambres[indice_chambre(numero)] == 0;
84 }
```

---------- ✂

11. Écrire la définition de la méthode `reserve`. L'étudiant ne doit pas avoir déjà une chambre, sinon on lèvera une exception. D'autre part, s'il n'y a pas de chambre libre, on lèvera également une exception. L'étudiant est passé par référence à la méthode car on doit modifier l'étudiant pour lui donner sa chambre.

✂ ----------

Sur 2pt.

- 0.5 pour la recherche correcte
- 0.5 pour les deux cas exceptionnels
- 0.5 pour la modification de `chambre[i]`
- 0.5 pour l'appel à la méthode `set_chambre`
- -0.25 si l'entête n'est pas correct

```
87 void Residence::reserve(Etudiant &etu) {
88   if (etu.get_chambre() != 0)
89     throw invalid_argument("L'étudiant a déjà une chambre");
90   for (int i=0; i<100; i++) {
91     if (chambres[i] == 0) {
92       chambres[i] = etu.get_numero();
93       etu.set_chambre(numero_chambre(i));
94       return;
95     }
96   }
97   throw invalid_argument("Aucune chambre libre");
98 }
```

---------- ✂

12. Écrire la définition de la méthode `libere`. Si l'étudiant n'a pas de chambre, on lèvera une exception. L'étudiant est passé par référence à la méthode car on doit modifier l'étudiant pour lui enlever sa chambre.

✂ ----------

Sur 1pt.

- 0.25 pour l'appel à la méthode `get_chambre`
- 0.25 pour le cas exceptionnel

- 0.25 pour la modification de `chambre[i]`
- 0.25 pour l'appel à la méthode `set_chambre`
- -0.25 si l'entête n'est pas correct

```
101 void Residence::libere(Etudiant &etu) {
102   int ch = etu.get_chambre();
103   if (ch == 0)
104     throw invalid_argument("L'étudiant n'a pas de chambre");
105   chambres[indice_chambre(ch)] = 0;
106   etu.set_chambre(0);
107 }
```

▸ **Exercice 5. (Implémentation avec un tableau et un vecteur)** – sur 4.25 points –

L'université de Marie-sa-Clé souhaite utiliser cette application. Malheureusement, elle ne possède pas 100 chambres mais plutôt 4000. Si les développeurs on pu facilement adapter les fonctions `indice_chambre` et `numéro_chambre`, l'implémentation précédente se révèle trop lente à l'usage. Les développeurs décident de faire une deuxième implémentation qui utilise en plus du tableau `chambres`, un vecteur `libres` qui contient les indices des chambres libres, dans un ordre quelconque. Quand on libère une case on rajoute son indice à la fin du vecteur. Quand on réserve une case, on réserve celle dont l'indice est à la fin du vecteur. On n'a donc jamais besoin de décaler les éléments du vecteur.

Avec cette nouvelle solution, on demande de ré-écrire les définitions ci-dessous. Bien sûr, l'interface de ces méthodes ne doit pas changer. De plus, pour que le programme aille le plus vite possible, on s'efforcera de ne pas utiliser de boucle.

13. la définition du constructeur de la classe `Residence`.

Sur 0.75pt.

- 0.5 pour les `push_back`
- 0.25 l'init de chambre

```
78 Residence::Residence() {
79   for (int i=0; i<4000; i++) {
80     chambres[i] = 0;
81     libres.push_back(i);
82   }
83 }
```

---

14. la définition de la méthode `reserve`.

✂ ----------

Sur 2pt.

- 0.5 pour le cas exceptionnel `libre.size()`. On ne renote pas l'autre.
- 0.5 pour l'appel à back (ou size-1)
- 0.5 pour le `pop_back`
- 0.5 pour le reste

```
91  void Residence::reserve(Etudiant &etu) {
92    if (etu.get_chambre() != 0)
93      throw invalid_argument("L'étudiant a déjà une chambre");
94    if (libres.size() == 0)
95      throw invalid_argument("Aucune chambre libre");
96    int i = libres.back();
97    libres.pop_back();
98    chambres[i] = etu.get_numero();
99    etu.set_chambre(numero_chambre(i));
100 }
```

---------- ✂

15. la définition de la méthode `libere`.

✂ ----------

Sur 1.5pt.

- 0.75 pour le `push_back` du bon indice
- 0.75 pour le reste

```
103 void Residence::libere(Etudiant &etu) {
104   int ch = etu.get_chambre();
105   if (ch == 0)
106     throw invalid_argument("L'étudiant n'a pas de chambre");
107   int i = indice_chambre(ch);
108   chambres[indice_chambre(ch)] = 0;
109   etu.set_chambre(0);
110   libres.push_back(i);
111 }
```

---------- ✂