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,...).

Répondre dans les cadres fournis. Si vous avez besoin de place supplémentaire, il y a une page libre à la fin du document. Dans ce cas, précisez-le toujours dans le cadre prévu pour la réponse.

Le barème est indicatif et pourra changer à la correction.

Durée : 2h00.

▸ **Exercice 1. (Programme faux – 2 points)** On a écrit une classe `C` qui sert à stocker un entier donné à la construction. On souhaite construire un objet de cette classe et afficher l'entier stocké. On vous donne ci-dessous le code de la classe ainsi qu'un programme **faux** essayant de résoudre le problème. Expliquez ci-dessous les **deux raisons** qui font que le programme ne marche pas.

- La déclaration C var; va essayer de construire un objet sans paramètre, alors que l'unique constructeur de la classe C nécessite la donnée d'un entier.
- On cherche à accéder à l'entier stocké pour le modifier / l'afficher alors que c'est un champ privé.

.

Donnez sur la droite une fonction `main` qui fait le travail demandé **sans modifier le code de la classe** `C` (autrement dit, ré-écrire le `main` en corrigeant ses erreurs).

```
class C {
public:
    C(int n): i{n} {}
    int get_i() const { return i;}
private:
        int i;
};

int main() {
    C var;
    var.i = 2;
    cout << "Valeur : ";
    cout << var.i << endl;
    return 0;
}
```

```
int main() {
    C var(2);
    cout << "Valeur : ";
    cout << var.get_i() << endl;
    return 0;
}

//
```

PLACE :

Problème : Hotel en bord de mer

Les exercices suivants traitent tous d'un même problème : la gestion des chambres dans un hôtel. Cependant, les exercices et les questions sont **indépendantes** les unes des autres, dans la mesure où on peut répondre à n'importe quelle question sans avoir su faire les précédentes. Néanmoins pour une bonne partie des questions, il est nécessaire d'avoir **lu en détail l'énoncé** des questions précédentes (voire des exercices précédents) pour pouvoir répondre correctement à la question.

Description du problème

On souhaite écrire un logiciel pour gérer la réservation des chambres dans un hôtel. Pour cela, on créera trois classes :

— la classe `Client` pour représenter un client de l'hôtel
— la classe `Chambre` pour représenter une chambre de l'hôtel
— la classe `Hotel` pour représenter l'hôtel et gérer les réservations de chambres par les clients

▸ **Exercice 2. (Organisation multi-fichiers et compilation séparée – 2 points)** On souhaite que chaque classe soit écrite dans son propre fichier. En plus de cela, on aura aussi un fichier qui contiendra la fonction `main` pour lancer le logiciel et un fichier pour créer un exécutable de test.

1. Donnez la liste de tous les fichiers `.cpp` et `.hpp` que nous devons créer.

```
* client.hpp et client.cpp
* chambre.hpp et chambre.cpp
* hotel.hpp et hotel.cpp
* hotel-main.cpp (ou autre nom)
* hotel-test.cpp (ou autre nom)
  .
.
```

2. Sachant que la classe `Hotel` utilise les classes `Client` et `Chambre`, quelles sont les dépendances de `hotel.o` dans le `Makefile` ? (On ne demande pas la ligne de compilation, seulement la liste des fichiers qui, s'ils sont modifiés, doivent entrainer une nouvelle compilation de `hotel.o`).

```
hotel.cpp, hotel.hpp, client.hpp, chambre.hpp
.
```

PLACE :

▸ **Exercice 3. (La classe Client – 4 points)**

Un client doit toujours être créé avec un id, un nom et un prénom (dans cet ordre) comme ci-dessous, un id étant un numéro qui permet d'identifier le client.

```
Client c{2, "Lovelace", "Ada"};
```

Ces trois valeurs ne sont jamais modifiées. On stocke aussi un possible identifiant de chambre : à la création, il n'y a pas de chambre (on met `-1` par défaut), et on veut des méthodes pour : assigner un identifiant de chambre, le récupérer, le supprimer, et tester si une chambre a été assignée.

1. Dans quel fichier doit se trouver la **déclaration** de la classe `Client` ?

```
Le fichier client.hpp
.
```

2. Donner la **déclaration** de la classe `Client` (on n'inclura pas la définition des méthodes) de sorte qu'elle respecte ce qui est demandé au début de l'exercice, et que l'interface soit compatible avec l'exemple ci-dessous où `client` est un objet de type `Client` et `chId` un entier pré-défini. On prendra bien soin de séparer la partie `public` et `private` et de spécifier les méthodes constantes.

```
cout << "Id, nom et prénom du client : ";
cout << client.getId() << " " << client.getNom() << " " << client.getPrenom();
if(client.aChambre()) {
    cout << "Son identifiant de chambre est : ";
    cout << client.getChambreId() << endl;
} else {
    client.setChambreId(chId);
    cout << "On lui assigne la chambre " << chId <<endl;
}
// ... plus loin dans le code:
client.supprimeChambre();
```

```
class Client {
public:
    Client(int id, string nom, string prenom);
    int getId() const;
    string getNom() const;
    string getPrenom() const;

    bool aChambre() const;
    int getChambreId() const;
    void supprimeChambre();
    void setChambreId(int ch);

private:
    int id;
    string nom;
    string prenom;
    int chambreId;
};









//
```

PLACE :

3. Donner la définition du constructeur de la classe `Client`. On rappelle qu'un client n'a pas de chambre à la construction. Par ailleurs, son id doit être supérieur ou égal à 0, dans le cas contraire, on lèvera une exception `invalid_argument`.

```
Client::Client(int id, string nom, string prenom): id{id}, nom{nom}, prenom{prenom}, chambreId{-1} {
    if(id < 0) {
        throw invalid_argument("Id négatif");
    }
}









//
```

4. Écrire des tests pour les méthodes de gestion du champ identifiant la chambre : tester qu'un client n'a pas de chambre à la création, qu'il a bien une chambre si on lui en assigne une et qu'on récupère le bon identifiant, tester la suppression de la chambre et enfin, tester qu'une exception `invalid_argument` est levée si on lui assigne un identifiant de chambre négatif.

```
    Client c{2, "Lovelace", "Ada"};
    CHECK_FALSE(c.aChambre());
    c.setChambreId(3);
    CHECK(c.getChambreId() == 3);
    CHECK(c.aChambre());
    c.setChambreId(5);
    CHECK(c.getChambreId() == 5);
    c.supprimeChambre();
    CHECK_FALSE(c.aChambre());
    CHECK_THROWS_AS(c.setChambreId(-1), invalid_argument);
```

PLACE :

▸ **Exercice 4. (La Classe Chambre – 3 points)** L'hôtel possède 200 chambres au total, et chaque chambre correspond à un identifiant allant de 0 à 199. Voici une version simplifiée de la déclaration de la classe `Chambre` avec seulement son constructeur et son identifiant.

```
class Chambre {
public:
    Chambre(int id);
    int getId() const;

private:
    int id;
};
```

Cependant, l'hôtel est organisé en deux bâtiments, de 5 étages chacun avec 20 chambres par étages. Si on connaît le numéro de bâtiment (1 ou 2), l'étage (de 1 à 5) et de le numéro de porte (de 1 à 20), on peut calculer l'identifiant de la chambre avec la formule suivante

$$id = (bat - 1) \times 100 + (etage - 1) \times 20 + porte - 1.$$

De même, on peut retrouver le bâtiment, l'étage et la porte à partir de l'identifiant. **Seul l'identifiant est stocké dans la classe**.

1. On déclare le constructeur suivant pour la classe `Chambre` :

   ```
   Chambre(int bat, int etage, int porte);
   ```

   Donner la **définition** de ce constructeur qui **doit** utiliser le constructeur précédent qui prend seulement un identifiant et qu'on suppose déjà écrit. Par ailleurs, on lèvera une exception `invalid_argument` si le numéro de bâtiment, d'étage ou de porte sont incorrects.

```
Chambre::Chambre(int bat, int etage, int porte): Chambre{(bat -1)*100 + (etage-1)*20 + porte-1} {
    if(bat <= 0 or bat > 2 or etage <= 0 or etage > 5 or porte <= 0 or porte > 20) {
        throw invalid_argument("Chambre invalide");
    }
}









//
```

2. Le bâtiment, l'étage et la porte peuvent être calculés à partir de l'identifiant, on ajoute donc les 3 méthodes suivantes à la classe `Chambre`

   ```
   int getBatiment() const;
   int getEtage() const;
   int getPorte() const;
   ```

   Indiquer précisément où ces méthodes doivent être déclarées.

```
Dans la classe Chambre, dans la partie public



.
```

PLACE :

3. On suppose les 3 méthodes précédentes écrites. Une chambre est face à la mer si elle est dans le bâtiment 1 avec un numéro de porte pair. Donner la définition de la méthode `faceMer` ci-dessous qu'on a ajoutée à la classe `Chambre`

```
bool faceMer() const;
```

```
bool Chambre::faceMer() const {
    return (getBatiment() == 1 and getPorte()%2 == 0);
}

//
```

4. Donner la définition de la surcharge de l'opérateur d'affichage `<<` qui permette d'afficher les informations de la chambre de la façon suivante

```
Chambre identifiant 49 : batiment 1, étage 3, porte 10
```

```
ostream& operator<<(ostream &out, const Chambre &chambre) {
    out << "Chambre identifiant " << chambre.getId();
    out << " : batiment " << chambre.getBatiment();
    out << ", étage " << chambre.getEtage();
    out << ", porte " << chambre.getPorte();
    return out;
}

//
```

▸ **Exercice 5. (L'hôtel)**

La classe `Hotel` s'occupe de la gestion des chambres : quelles chambres sont libres, lesquelles sont réservées par quels clients, etc. Voici la partie publique de la déclaration de la classe :

PLACE :

```
class Hotel {
public:
    /** Crée un hotel vide **/
    Hotel();

    /** Teste si une chambre donnée est libre
     * @param ch, une chambre
     * @return true si la chambre est libre, false sinon
     **/
    bool estLibre(const Chambre &ch) const;

    /** Renvoie l'id client d'une chambre
     * @param ch, une chambre
     * @return un identifiant de client
     **/
    int idClient(const Chambre &ch) const;

    /** Réserve une chambre libre quelconque pour un client
     * @param client, le client à qui on attribue une chambre
     **/
    void reserve(Client &client);

    /** Réserve une chambre libre pour un client avec faceMer fixé
     * @param client, le client à qui on attribue une chambre
     * @param faceMer, la propriété de la chambre (face à la mer ou non)
     **/
    void reserve(Client &client, bool faceMer);

    /** Libère la chambre occupée par un client donné
     * @param client, le client qui occupe une chambre
     **/
    void libere(Client &client);

};
```

1. Suivant cette interface (ainsi que celles des exercices précédents) et en supposant que toutes les méthodes sont définies, écrire les instructions (lignes de code) qui créent un hôtel vide, créent un client, réservent une chambre pour le client puis testent que : le client a bien une chambre, que cette chambre est bien notée comme occupée pour l'hôtel, que le numéro de client associé à la chambre dans l'hôtel est bien le bon.

```
Hotel h;
Client c{1,"Lovelace", "Ada"};
h.reserve(c);
CHECK(c.aChambre());
Chambre ch{c.getChambreId()};
CHECK_FALSE(h.estLibre(ch));
CHECK(h.idClient(ch) == c.getId());









//
```

PLACE :

2. On propose une première implantation concrète de la classe `Hotel` avec simplement un tableau d'entiers dont les indices sont les identifiants des chambres. Si une chambre est libre, elle contient -1, sinon, elle contient l'id du client qui l'occupe. Voilà la déclaration du tableau :

   ```
   array<int, MAXCHAMBRES> chambres;
   ```

   Où précisément doit être ajoutée cette déclaration ?

```
Dans la classe Hotel avec le mot clé private


.
```

3. Donner la définition du constructeur de la classe `Hotel` (où toutes les chambres sont libres par défaut).

```
Hotel::Hotel() {
    for(int i=0; i < MAXCHAMBRES; i++) chambres[i] = -1;
}









//
```

4. Donner la définition de la méthode `estLibre(const Chambre &chambre)`.

```
bool Hotel::estLibre(const Chambre &ch) const {
    return chambres[ch.getId()] == -1;
}









//
```

PLACE :

5. Donner la définition de la méthode `reserve(Client &client, bool faceMer)`. Cette méthode doit modifier l'hôtel ET le client pour associer une chambre libre au client passé en paramètre. La chambre assignée doit respecter la condition donnée par `faceMer`. Si l'hôtel n'a plus de chambre de ce type disponible, on lèvera une exception `runtime_error`. Si le client a déjà une chambre, on lèvera une exception `invalid_argument`. Voici un exemple d'appel à cette méthode dans un test :

```
Hotel h;
Client c1{1,"Lovelace", "Ada"};
Client c2{2,"Hopper", "Grace"};
h.reserve(c1, true);
CHECK(Chambre{c1.getChambreId()}.faceMer());
h.reserve(c2, false);
CHECK_FALSE(Chambre{c2.getChambreId()}.faceMer());
```

```
void Hotel::reserve(Client &client, bool faceMer) {
    if(client.aChambre()) {
        throw invalid_argument("Le client a déjà une chambre");
    }
    for(int i = 0; i < MAXCHAMBRES; i++) {
        if(chambres[i] == -1 and Chambre{i}.faceMer() == faceMer) {
            chambres[i] = client.getId();
            client.setChambreId(i);
            return;
        }
    }
    throw runtime_error("Plus de chambres selon les critères");
}

//
```

PLACE :

6. Donner la définition de la méthode `libere(Client &client)`. La méthode doit modifier l'hôtel ET le client pour supprimer la chambre réservée. Si le client n'a pas de chambre, on lèvera une exception `invalid_argument`. Si la chambre du client n'était pas marquée comme occupée dans l'hôtel, on lèvera une exception `runtime_error`.

```
void Hotel::libere(Client &client) {
    if(not client.aChambre()) {
        throw invalid_argument("Le client n'a pas de chambre");
    }
    int ch = client.getChambreId();
    if(chambres[ch] == -1) {
        throw runtime_error("La chambre est déjà libre");
    }
    chambres[ch] = -1;
    client.supprimeChambre();
}









//
```

7. On souhaite utiliser une structure de donnée plus efficace pour la réservation des chambres. On ajoute pour cela deux `vector` d'entiers qui stockent les identifiants chambres libres face à la mer ou non (dans un ordre quelconque). Les champs de la classe hôtel sont maintenant les suivants :

```
array<int, MAXCHAMBRES> chambres;
vector<int> libresFaceMer;
vector<int> libresNonFaceMer;
```

Quelles sont les méthodes dont la définition doit être mise à jour pour prendre en compte cette nouvelle structure de donnée ?

```
* le constructeur Hotel
* les deux méthodes de réservation
* la méthode libere









.
```

PLACE :

8. Donner la nouvelle définition du constructeur de la classe `Hotel`

```
Hotel::Hotel() {
    for(int i=0; i < MAXCHAMBRES; i++) {
        chambres[i] = -1;
        Chambre ch{i};
        if(ch.faceMer()) {
            libresFaceMer.push_back(i);
        } else {
            libresNonFaceMer.push_back(i);
        }
    }
}









//
```

9. Donner la nouvelle définition de la méthode `libere(Client &client)`

```
void Hotel::libere(Client &client) {
    if(not client.aChambre()) {
        throw invalid_argument("Le client n'a pas de chambre");
    }
    int chid = client.getChambreId();
    Chambre ch{chid};
    if(chambres[chid] == -1) {
        throw runtime_error("La chambre est déjà libre");
    }
    chambres[chid] = -1;
    client.supprimeChambre();
    if(ch.faceMer()) {
        libresFaceMer.push_back(chid);
    } else {
        libresNonFaceMer.push_back(chid);
    }
}









//
```

PLACE :

Cette page peut être utilisée si vous n'avez pas eu la place dans les cadres prévus pour les réponses. **Attention !** Nous corrigeons en ne voyant QUE le cadre, si vous écrivez dans les espaces supplémentaires, NOTEZ LE au niveau du cadre prévu pour la réponse.

.