université PARIS-SACLAY

Examen final 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. (Programmes faux) – sur 4 points –

Les deux programmes suivants sont faux. Une erreur est signalée par le compilateur aux lignes 14 marquées d'un commentaire. Pouvez-vous en quelques phrases expliquer l'erreur et indiquer quelles modifications il faut faire pour avoir un programme correct. La correction ne doit pas changer profondément le sens du code. Pour la donner, il n'est pas utile de recopier tout le programme, mais par exemple de dire «remplacer la ligne no 5 par la ligne suivante» ou encore «ajouter les lignes suivantes après la ligne no 8».

```
1  #include<iostream>
2  using namespace std;
3
4  class C {
5    int i_;
6   public:
7    C() : i_{0} {}
8    int get_i() const { return i_; }
9    void set_i(int i) { i_ = i; }
10 };
11
12 ostream &operator<<(ostream &out,
13                      const C &t) {
14   out << t.i_; // <===== Erreur
15   return out;
16 }
17
18 int main() {
19   C t;
20   t.set_i(4);
21   cout << t << endl;
22 }
```

```
1  #include<iostream>
2  using namespace std;
3
4  class C {
5    int i_;
6   public:
7    C(int i) : i_{i} {}
8    void set_i(int i) { i_ = i; }
9    bool est_positif() const { return i_ >= 0; }
10 };
11
12
13 int main() {
14   C t; // <========= Erreur
15   t.set_i(4);
16   cout << t.est_positif() << endl;
17 }
```

Problème : Réseau de Bus

Le Réseau Aléatoire des Trajets Poussifs souhaite créer une application permettant aux usagers de chercher leur trajets et horaires dans un réseau de bus. On commence par modéliser les horaires de passage.

▸ **Exercice 2. (Horaires)** – sur 6 points –

Dans cet exercice, on souhaite créer une classe pour modéliser les horaires de passage des bus aux arrêts. Les horaires sont à la minute près (par exemple «`09:04`»). Les heures sont comprises entre 0 et 23 inclus et les minutes entre 0 et 59 inclus. Pour permettre une manipulation sûre des heures, la classe `Horaire` doit permettre de construire uniquement des horaires corrects, et il ne doit pas être possible de modifier un horaire.

Voici la liste des méthodes permettant de manipuler les objets de la classe :

- un constructeur à partir de deux entiers `h` et `m` ;
- un opérateur d'égalité ;
- un opérateur `<` qui permet de savoir si un horaire est avant un autre dans une même journée ;
- une méthode `imprime` qui prend en paramètre un flux de type `ostream &` et qui imprime l'horaire courant dans le flux en respectant le format `09:24` ou bien `12:00`.

On souhaite également surcharger l'opérateur d'affichage pour les `Horaire`.

Note : en dehors de l'affichage et des comparaisons, aucune autre opération ne sera effectuée sur les horaires. On n'a donc pas besoin de méthode d'accès (getter) pour les heures et les minutes d'un horaire.

1. Écrire le fichier `horaire.hpp` qui permet de déclarer la classe `Horaire` avec les quatre méthodes précédentes et l'opérateur d'affichage. On demande d'écrire l'intégralité du fichier avec les gardes d'inclusions multiples. On n'oubliera pas d'inclure convenablement le fichier `iostream` de manière à pouvoir utiliser le type `ostream`. Enfin, le fichier ne devra pas contenir de définition «en ligne».
2. Écrire la définition du constructeur de la classe `Horaire`. Dans le cas où les heures ou les minutes ne sont pas dans les intervalles ci-dessus, on lèvera une exception `invalid_argument`.
3. Dans quel fichier doit-on écrire cette définition ?
4. Écrire la définition de l'opérateur de comparaison `<`.
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`.

   Écrire la définition de la méthode `imprime`.
6. En utilisant la méthode `imprime`, surcharger l'opérateur d'affichage pour les `Horaire`.
7. Donner les deux lignes d'un fichier `Makefile` permettant de compiler un fichier `horaire.o` ; la première ligne devra donner toutes les dépendances et la deuxième ligne la commande de compilation.

▸ **Exercice 3. (Lignes de bus)** – 8 points –

On veux maintenant modéliser une ligne de bus. Tous les bus d'une ligne s'arrêtent à tous les arrêts de cette ligne. Le dernier arrêt d'une ligne est un terminus qui ne permet pas de rejoindre le début de la ligne (pas de ligne circulaire). Un même nom d'arrêt de bus peut apparaître dans plusieurs lignes pour modéliser les correspondances entre lignes. Pour simplifier, on considère que tout bus parti un jour donné arrive ce même jour (aucun bus n'est en cours de circulation à minuit) et que les horaires de passage sont les mêmes chaque jour. Enfin, pour un arrêt donné, les passages sont triés dans l'ordre croissant.

Voici l'exemple des horaires de la ligne de bus 19.60a :

Ligne : Bus 19.60a

Arret	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Andsay	08:05	08:30	09:10	10:30	11:10	11:40	12:10	13:22	13:37	14:10	15:30	16:20	17:02	17:20	18:00	18:17	19:35
Le Pichet	08:10	08:35	09:15	10:35	11:15	11:45	12:15	13:27	13:42	14:15	15:35	16:25	17:07	17:25	18:05	18:22	19:40
Castel	08:12	08:37	09:17	10:37	11:17	11:47	12:17	13:29	13:44	14:17	15:37	16:27	17:09	17:27	18:07	18:24	19:42
Ris	08:17	08:42	09:22	10:42	11:22	11:52	12:22	13:34	13:49	14:22	15:42	16:32	17:14	17:32	18:12	18:29	19:47
Laiseau	08:20	08:45	09:25	10:45	11:25	11:55	12:25	13:37	13:52	14:25	15:45	16:35	17:17	17:35	18:15	18:32	19:50
Looseur	08:27	08:52	09:32	10:52	11:32	12:02	12:32	13:44	13:59	14:32	15:52	16:42	17:24	17:42	18:22	18:39	19:57
Bebelvedaire	08:32	08:57	09:37	10:57	11:37	12:07	12:37	13:49	14:04	14:37	15:57	16:47	17:29	17:47	18:27	18:44	20:02
Boulon	08:43	09:08	09:48	11:08	11:48	12:18	12:48	14:00	14:15	14:48	16:08	16:58	17:40	17:58	18:38	18:55	20:13

Cette ligne comporte 8 arrêts, et 17 bus (numérotés de 0 à 16) circulent dans une journée. Pour ne pas confondre le numéro de la ligne (19.60a), on appelle le numéro des 17 bus le **rang** du bus. Ainsi le bus 19.60a de rang 4 passe à `11:15` à l'arrêt «Le Pichet».

Pour modéliser une ligne, on va utiliser deux classes dont voici un extrait des déclarations :

```
class Arret {
  string nom;
  // horaires de passage à cet arrêt,
  // triés de façon croissante
  vector<Horaire> passages;

 public:
  Arret(string n, vector<Horaire> s);
  // ...
};
```

```
class Ligne {
  string nom;
  // liste des arrêts de la ligne,
  // dans l'ordre de parcours.
  vector<Arret> arrets;

 public:
  Ligne(string s, vector<Arret> a);
  // ...
};
```

1. Les deux classes sont déclarées et définies dans des fichiers séparés portant le même nom que la classe. Avec la classe `Horaire` précédente, on a donc trois fichiers `.hpp` et trois fichiers `.cpp`. Pour chacun de ces 6 fichiers, donner les directives d'inclusion des autres fichiers que nous avons définis (on ne demande pas les inclusions de la librairie standard du C++ comme `vector`).
2. Pour chacune des deux classes `Arret` et `Ligne`, écrire la ligne du `Makefile` qui donne la liste des dépendances du fichier `.o` associé.
3. Déclarer et définir en une seule ligne le getter `get_nom` qui retourne le nom d'un `Arret`. Dans quel fichier et à quel endroit placez-vous cette déclaration ?
4. Donner la déclaration d'une méthode `prochain_passage`, qui prend un `Horaire` et qui retourne un entier donnant le *rang* du passage qui suit l'horaire donné. Par exemple, à l'arrêt nommé «Laiseau» à `11:05` le prochain passage est le passage de rang 4 à `11:25`.
5. Donner la définition de la méthode `prochain_passage` précédente.
6. Donner la déclaration et la définition d'une méthode `ajoute_passage` qui prend un `Horaire` et qui ajoute un nouveau passage à cet horaire à la fin de la liste des passages de l'arrêt. La méthode doit vérifier que l'ajout est valide (la liste des passages reste triée) et signaler une exception si ce n'est pas le cas.

Pour être valide une ligne doit vérifier les invariants suivants : une ligne est composée de plusieurs arrêts (au moins 2). Comme les bus marquent tous les arrêts, les arrêts qui composent une ligne ont tous le même nombre de passages. De plus, pour un rang de passage donné, les horaires des différents arrêts doivent aller en ordre croissant, sinon le bus devrait remonter le temps pour faire sa tournée.

7. Écrire le constructeur de la classe `Ligne`. Le constructeur doit vérifier les trois conditions ci-dessus et signaler une exception avec un message décrivant le problème si ce n'est pas le cas.
8. Donner la déclaration et la définition d'une méthode `cherche_arret` qui prend une chaîne de caractères et qui retourne la position de l'arrêt dont c'est le nom dans la ligne, ou -1 s'il n'y a pas d'arrêt portant ce nom.
9. Définir une méthode `voyage` qui permet de chercher un voyage entre deux arrêts de la même ligne à un horaire donné. La méthode prend trois paramètres : un nom d'arrêt de départ, un horaire à partir duquel on veut partir, et un nom d'arrêt d'arrivée. Elle retourne un tableau de taille 2 décrivant le voyage, le premier élément du tableau est l'horaire de passage du bus à l'arrêt de départ et le deuxième l'horaire de passage du bus à l'arrêt d'arrivée.

▸ **Exercice 4. (Réseau de Bus)** – 2 points –

On considère maintenant des trajets avec des correspondances grâce aux arrêts qui appartiennent à plusieurs lignes du réseau. Un trajet est décrit comme un horaire de départ et une succession d'«étapes» dont chacune mentionne une ligne et deux arrêts de cette ligne. On se donne les types ci-dessous :

```
struct Etape {
  string nomLigne;     /* nom de la ligne a utiliser */
  string nomDepart;    /* arret de depart de cette etape */
  string nomArrivee;   /* arret d'arrivee de cette etape */
};

struct Trajet {
  Horaire hDepart;       /* heure initiale de depart */
  vector<Etape> etapes;  /* sequence d'etapes a utiliser */
};

class Reseau {
  vector<Ligne> lignes;          // un réseau est un ensemble de lignes

 public:
  Reseau(vector<Ligne> l);
  int cherche_ligne(string nom) const; // le numéro de la ligne portant le nom
  Horaire horaire_arrivee(const Trajet &t) const;
};
```

Pour qu'un trajet soit cohérent il faut que l'arrêt d'arrivée d'une étape ait le même nom que le point de départ de l'étape suivante, que chaque étape corresponde à un trajet possible sur cette ligne et enfin qu'à chaque étape il existe bien encore un horaire de départ en fonction de l'heure à laquelle on va arriver à cet endroit.

1. Écrire une méthode de la classe `Reseau` qui calcule l'horaire d'arrivée d'un `Trajet` tout en vérifiant si le trajet est bien cohérent. On pourra utiliser sans la définir la méthode `cherche_ligne` déclarée ci-dessus.