IN204
Programmation Orientée Objet – Examen de mise en œuvre des notions de C++

Examen du 21 novembre 2023

B. Monsuez

NOM :	
PRENOM :	

Question n°1: Constructeurs

Nous nous intéressons à la classe suivante qui définit un intervalle :

```
class interval
{
private:
    int lowerBound;
    int upperBound;

public:
    interval()
    {...}
    interval(int theLowerBound, int theUpperBound)
    {...}
    interval(const interval& anotherInterval)
    {...}
};
```

Question n°1.1:

Pour chacun des constructeurs, explique la fonction du constructeur et indiquez si le constructeur est un constructeur que C++ génère automatique en son absence ou pas.

```
    interval()
```

```
interval(int theLowerBound, int theUpperBound)
```

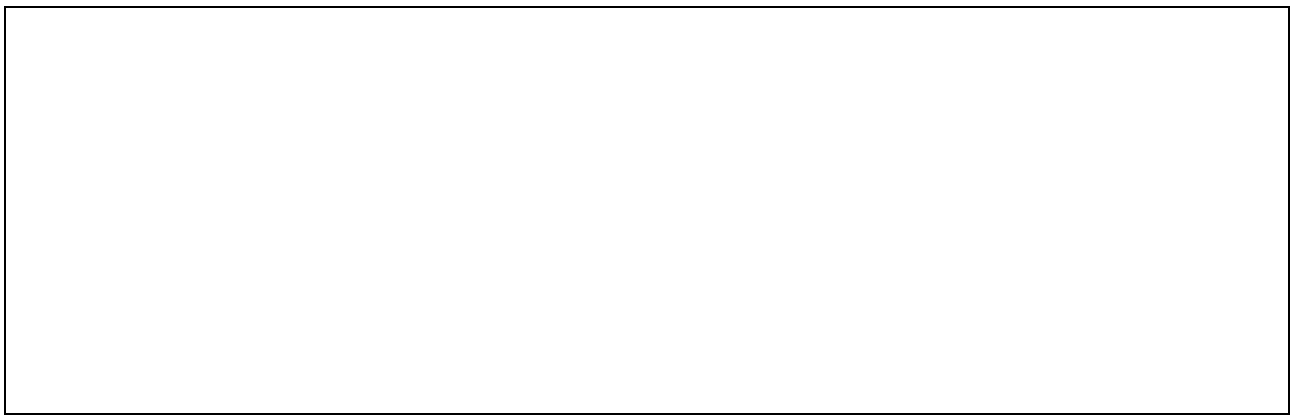
```
interval(const interval& anotherInterval)
```

Question n°1.2:

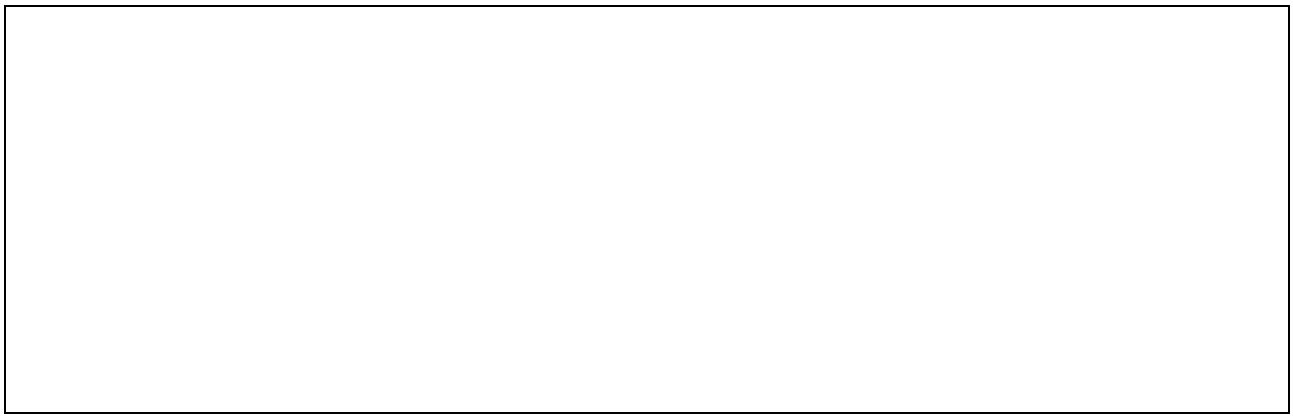
Écrivez pour chacun des constructeurs le code d'initialisation que le constructeur doit implanter.

```
interval()
```

```
interval(int theLowerBound, int theUpperBound)
```



```
interval(const interval& anotherInterval)
```



Question n° 2 : Accéder aux données internes stockées dans les champs

Question n° 2.1 : Champs privés

Est-il possible d'accéder aux champs `lowerBound` et `upperBound` ? Pourquoi donc ?

Question n° 2.2 :

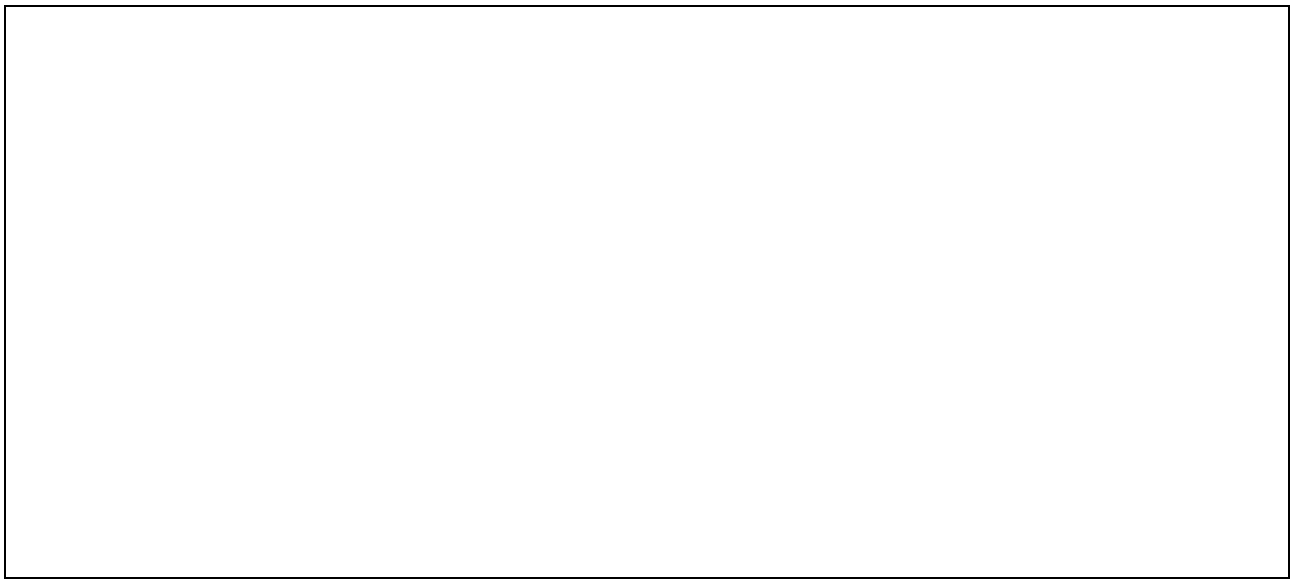
Proposez une méthode ou plusieurs méthodes pour accéder aux deux champs `lowerBound` et `upperBound` en lecture seulement.

Question n° 3 : Création des objects intervalles

Le constructeur `interval(int theLowerBound, int theUpperBound)` prend deux arguments, la borne minimale et la borne maximale. Nous aimerions pouvoir créer un intervalle en lui donnant la valeur de la bonne minimale ainsi que sa longeur.

Comment pourriez-vous pour implanter une telle fonction ? Proposer une solution.

> Pensez éventuellement à une méthode statique que l'on pourrait par exemple nommer `create`:
>
> ```
> static interval create(...) {}
> ```



Question n° 4 : Le problème de l'intervalle vide

Un intervalle vide est un intervalle qui ne contient aucun élément. Typiquement, on pourrait définir un intervalle vide comme étant un intervalle tel que :

```
upperBound - lowerBound < 0
```

Cependant, ceci n'est pas satisfaisant, parce que nous n'avons pas dès lors un seul intervalle vide, mais une infinité. Il est préférable de définir une seule représentation de l'intervalle libre en imposant que `upperBound` soit 0 et `lowerBound` soit 1.

Pour ce faire, nous devons modifier le code des constructeurs:

- pour que le constructeur par défaut retourne l'intervalle vide,
- pour que le constructeur qui prend deux arguments s'assure bien que `upperBound - lowerBound >= 0`, si ce n'est pas le cas, dans ce cas, il initialise l'intervalle comme étant l'intervalle vide.

Question n° 4.1 :

Procédez aux modifications du code pour les constructeurs suivants :

```
interval()
```

```
interval(int theLowerBound, int theUpperBound)
```

Faut-il modifier le code du constructeur :

```
interval(const interval& anotherInterval) ?
```

Question n° 4.2 :

Implantez une méthode qui teste si un intervalle représente l'intervalle vide :

```
bool is_empty() const
```

Expliquer pourquoi la méthode est déclarée `const` ?

Question n° 5 : Opérateurs de comparaison

Nous allons définir les opérations de comparaison. Nous commençons d'abord par l'opérateur d'égalité :

```
bool operator == (const interval& anotherInterval) const
```

Question n° 5.1

Expliquer la syntaxe de l'opérateur d'égalité puis donner son implantation.

Question n° 5.2

Nous nous intéressons aux opérateurs <= et >=. L'opérateur `a < b` teste si l'intervalle `a` est inclus dans ou égal à `b`. Ceci est le cas, - si `a` est l'intervalle vide - ou si `a.lowerBound >= b.lowerBound` et `a.upperBound <= b.upperBound`.

Proposer une implantation des opérateurs <= et >=.

Question n° 5.3

Terminer l'implantation des opérateurs !=, < et > en faisant appel aux opérateurs que vous avez déjà définis.

Question n° 6 : Opérateurs d'intersection & d'union

Nous nous proposons de définir & comme étant l'opérateur d'intersection et | comme étant l'opérateur d'union.

Question n° 6.1 :

Proposez une implantation des deux opérateurs qui prend deux intervalles et retourne un nouvel intervalle étant l'intersection, (resp. l'union) des deux intervalles arguments.

Question n° 7: Patrons

La classe `interval` est définie pour des intervalles de type entiers de type `int`. En fait, les intervalles peuvent être définis pour des entiers signés ou non signés pas aussi pour des nombres flottants et plus généralement, tout objet supportant un ordre.

Question n° 7.1

Proposer une transformation de la classe `interval` pour la paramétrer par un type représentant un nombre.

Question n° 7.2

En C++ 20, il est possible d'ajouter des contraintes sur les paramètres d'une fonction ou d'une classe. Déterminer les opérateurs ou les méthodes que le paramètre de la classe intervalle doit fournir pour que le code compile correctement.

Ensuite, proposer un ajout de contrainte pour vérifier dès la déclaration que le type paramètre fournit bien les opérateurs et méthodes attendues.

Nous rappellons que la bibliothèque standard de C++ fournit les concepts suivants:

```
template< class T >
concept integral; // Indique que le type T est un type entier (signé ou non s
igné).

template< class T >
concept floating_point; // Indique que le type T est un type de nombre à virg
ule flottant.

template< class T >
concept equality_comparable; // Indique que le type T supporte les opérateurs
== et !=.

template< class T >
concept totally_ordered; // Indique que le type T supporte les opérateurs ==
et !=, <, <=, >, >=
```

Question n° 8: Opérateurs surchargés & flux

Nous souhaitons écrire sur un flux de type. Pour ce faire, nous envisageons de définir un opérateur << qui a la signature suivante :

```
template<type T, typename charT, typename traits>
std::basic_ostream<charT, traits>& operator << (std::basic_ostream<charT, tra
its>& aStream, const interval<T>& theInterval);
```

Question n°8.1 :

Pourquoi cet opérateur ne peut pas être défini comme opérateur dans la classe `interval<T>` ?

Question n°8.2 :

Proposer une implantation qui pour l'objet `interval<float>(0.0, 1.5)` affiche [0.0, 1.5] et pour l'object `interval<int>()` affiche [].

Question n° 9: Dérivation

Nous souhaitons définir une classe dérivée de la classe d'intervalle qui ne représente que des singletons. Un singleton est un intervalle où `lowerBound` et `upperBound` sont égaux.

Question n° 9.1:

Construisez une classe :

```
template<typename T> class singleton ...
```

qui devra hériter de la classe `interval` et qui devra fournir les constructeurs suivants:

```
public:
    singleton(T aValue);
    singleton(const singleton<T>& anotherSingleton);
```

Question n° 9.2:

Nous nous proposons d'avoir une méthode publique qui `is_singleton` qui indique si l'objet que nous manipulons est un singleton. Typiquement, cette méthode sera définie dans la classe `interval` comme retournant toujours `false` et elle sera redéfinie dans la classe `singleton` comme retournant toujours `true`.

```
template<typename T>
class interval
{
    public:
        ...
        virtual bool is_singleton() const { return false; }
        ```
};

template<typename T>
class singleton: ...
{
    public:
        ...
        bool is_singleton() const { return true; }
        ...
};
```

Pourquoi devons-nous déclarer cette méthode comme virtuelle dans la classe `interval`?
```

Question n° 9.3:

Avec C++20, il est possible de rendre invisible des méthodes qui ont été déclarées dans la classe de base.

Pour rendre une méthode invisible dans la classe dérivée, il suffit de lui ajouter l'attribut `delete`.

```
struct A
{
    void method() {}
};

struct B: A
{
    void method() = delete;
};
```

La classe singleton même si elle ne dérive que de la classe `interval` ne contient qu'une seule valeur, les méthodes `getLowerBound()` et `setLowerBound()` vont retourner la valeur du singleton. Cependant, il serait plus judicieux de les remplacer par une méthode `getValue` qui retournerait `lowerBound` et de masques les deux méthodes `getLowerBound()` et `setLowerBound()`.

Modifier le code de la classe `singleton` afin de masquer les méthodes `getLowerBound()` et `setLowerBound()` et de définir une nouvelle méthode `getValue`.