

IN204 : Examen à mi-parcours

24 novembre 2024

Bruno Monsuez

NOM :	
PRENOM :	

Nous nous intéressons dans le cadre de ce sujet à la définition d'une classe **interval** qui définit un interval $[a, b]$ où a et b sont des valeurs entières.

L'idée est d'offrir en C++ un nouveau type de donnée permettant de définir un ensemble de valeur consécutive, comprise entre une valeur initiale et une valeur maximale.

Partie n°1: Définition d'une classe interval d'entiers

Nous considérons dans un premier temps le squelette de classe suivant :

```
#include<vector>
```

```
class interval
{
private:
    int mLowerBound;           // Index de la première valeur de la vue.
    int mUpperBound;           // Index de la dernière valeur de la vue.

public:
    interval();
    interval(int lowerBound, int upperBound);
};
```

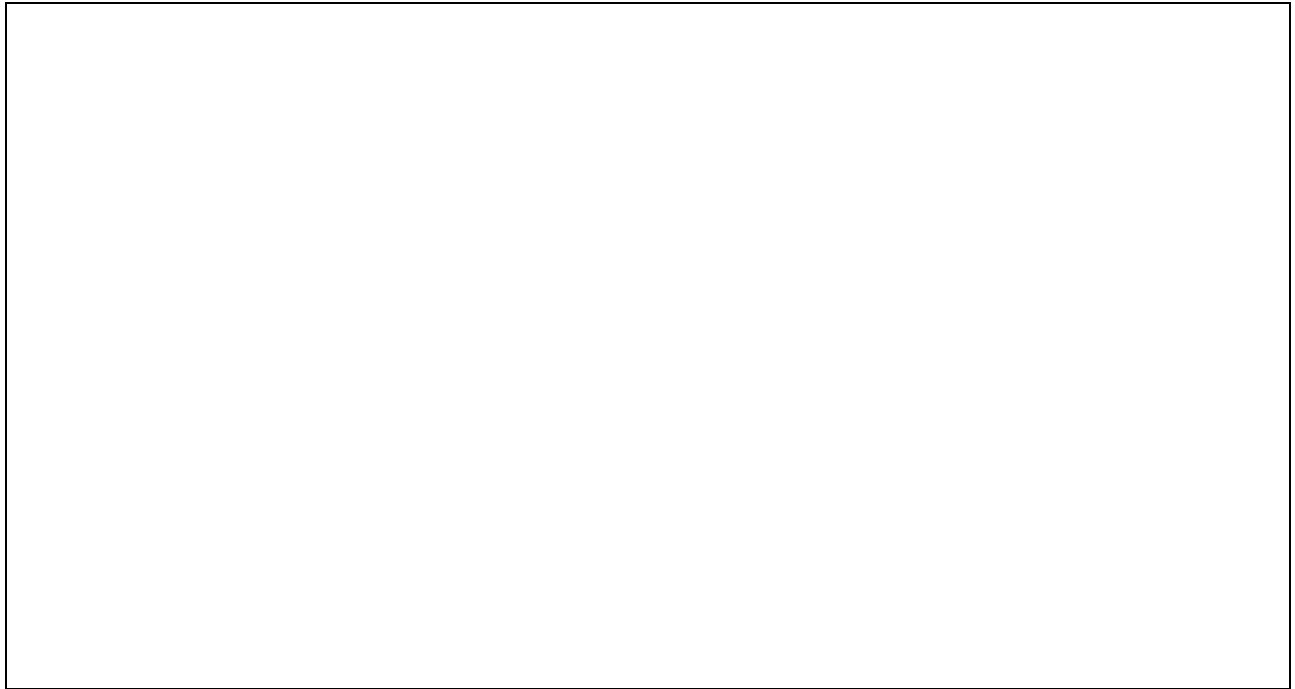
Nous considérons que si l'interval n'est pas vide, la propriété: $mLowerBound \leq mUpperBound$ doit toujours être vérifiée. Si l'interval est vide, alors $mLowerBound = 0$ et $mUpperBound = -1$.

1. Les constructeurs

Question 1.1

Expliquer à quoi correspondent les déclarations suivantes :

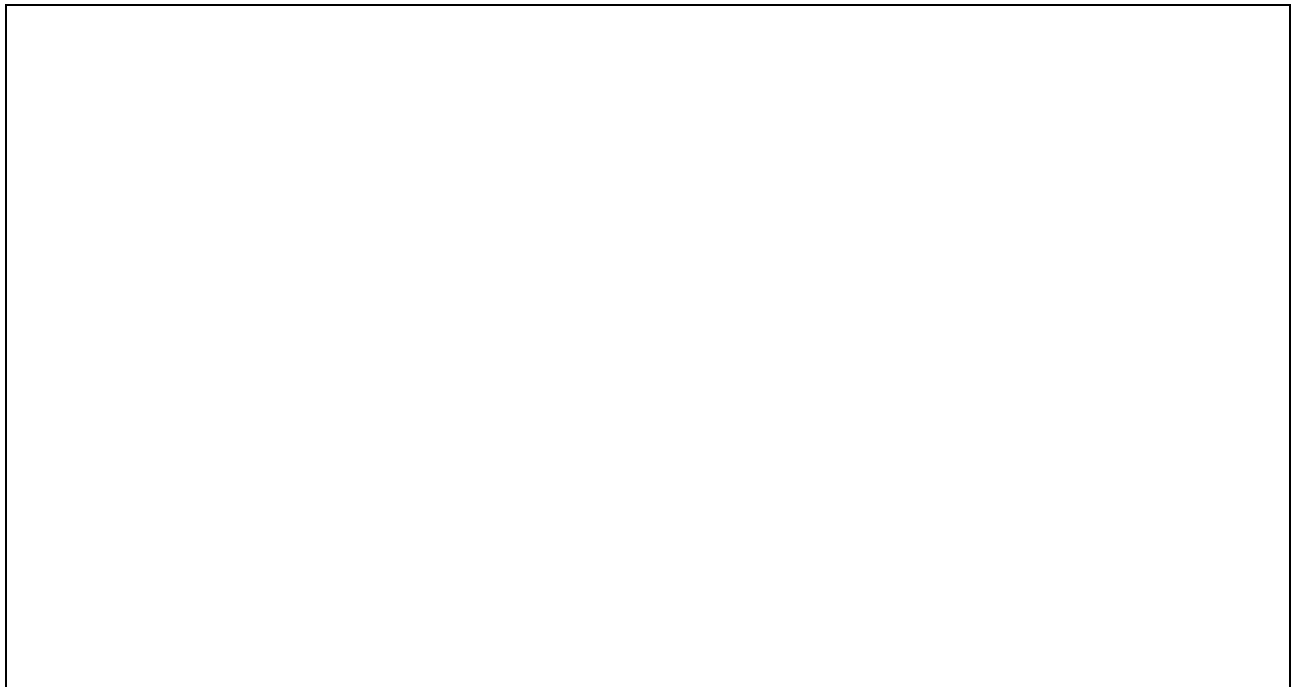
```
interval();
interval(int lowerBound, int upperBound);
```



Question 1.2

Pour chacun des constructeurs précédents, compléter le code des constructeurs.

Remarque : Le code est minimaliste, on ne demande pas de vérifier si les paramètres `lowerBound` et `upperBound` désignent des bornes valides.



Question 1.3

Y aurait-il besoin de compléter la liste des constructeurs. Expliquer pourquoi c'est nécessaire ou ce n'est pas nécessaire.

Si vous ajoutez un ou plusieurs constructeurs, écrivez le constructeur et son code.

Remarque : Penser à un interval qui ne représente qu'un seul élément par exemple.

2. Les intervals “vides”

Nous avons un souci avec les intervals “vides”. Un interval est dit vide si quand on passe les paramètres `lowerBound` et `upperBound`, `upperBound` est plus petit que `lowerBound`.

Question 2.1 Les intervals “vides”

Proposer une fonction membre `bool empty() const` qui teste si l’interval est vide ou n’est pas vide.

Question 2.2 Les intervals “vides”

Pourquoi la fonction le mot clé `const` suit la définition de la fonction `bool empty()` ?

3. Les données stockées au sein de la classe

Question 3.1

Est-il possible d'accéder aux champs `mLowerBound`, `mUpperBound` en dehors de la classe `interval` ? Expliquer pourquoi ?

Question 3.2

Proposer un moyen pour pouvoir accéder en lecture aux données stockées dans ces champs, mais surtout pas en écriture.

Conseils : Penser aux méthodes d'accès.

4. Opérateurs de comparaison

Nous souhaitons définir un opérateur qui détermine si deux objets **interval** désignent le même interval.

Question 4.1

Proposer une implantation des deux opérateurs suivants :

```
class interval
{
    ...
public:
    ...
    bool operator == (const interval&) const;
    bool operator != (const interval&) const;
    ...
};
```

5. Conteneur

Nous souhaitons que la classe `interval` soit un conteneur. En effet, un objet `interval` est en mesure de lister l'ensemble des valeurs comprises entre `mLowerBound` et `mUpperBound`. Nous rappelons rapidement les types et comportements que doit définir un conteneur :

Type	Description	
<code>value_type</code>	Type des valeurs stockées dans le conteneur (T)	
<code>reference</code>	Type référence des valeurs stockées dans le conteneur (T&)	
<code>const_reference</code>	Type référence non modifiable des valeurs stockées dans le conteneur (const T&)	
<code>iterator</code>	Itérateur référençant les valeurs stockées dans le conteneur et autorisant la modification de celles-ci	
<code>const_iterator</code>	Itérateur référençant les valeurs stockées dans le conteneur mais ne permettant pas de modifier le contenu du conteneur.	
<code>size_type</code>	Type permettant d'exprimer le nombre d'éléments stockés dans le conteneur (unsigned long)	

Expression	Type de retour	Description
<code>c.begin()</code>	<code>(const_)iterator</code>	Itérateur référençant le premier élément stocké dans le conteneur
<code>c.end()</code>	<code>(const_)iterator</code>	Itérateur référençant l'élément dénotant la fin de la séquence
<code>c.empty()</code>	<code>bool</code>	Aucun élément dans le conteneur
<code>c.size()</code>	<code>size_type</code>	Nombre d'éléments dans le conteneur.

Question 5.1

Est-il possible de modifier le contenu de l'interval en utilisant un itérateur les valeurs dans le conteneur ?

Remarque : Seule la borne inférieure et la borne supérieure de l'intervalle sont définies, les autres valeurs sont calculées à partir de ces deux valeurs.

Question 5.2

Si le conteneur n'est pas modifiable, expliquer pourquoi

- il n'y a qu'une seule définition des fonctions membres `begin()` et `end()`,
- ```
class interval
{
...
public:
...
 const_iterator begin() const;
 const_iterator end() const;
...
};
```
- `reference`, `const_reference` désignent le même type,
- `reference`, `const_reference` sont le plus souvent des alias de `value_type`.

### Question 5.3

Nous vous proposons le squelette de la classe `const_iterator` suivant:

```
class interval
{
 ...
public:
 struct const_iterator:
 std::iterator<std::bidirectional_iterator_tag, // iterator_category
 int, // value_type
 int, // difference_type
 const int*, // pointer
 int>
 {
 private:
 const interval* mInterval;
 int mCurrent;
 public:
 const_iterator(const interval& anInterval, int aValue):
 mInterval(&anInterval)
 {}
 reference operator*() const
 {
 return mCurrent;
 }
 bool operator == (const const_iterator& anotherIterator)
 {
 return mInterval == anotherIterator.mInterval
 && mCurrent == anotherIterator.mCurrent;
 }
 };
};
```

```

 }
 bool operator != (const const_iterator& anotherIterator)
 {
 return mInterval != anotherIterator.mInterval
 || mCurrent != anotherIterator.mCurrent;
 }
 const_iterator& operator ++()
 {
 if(mCurrent <= mInterval->mUpperBound)
 mCurrent ++;
 return *this;
 }
 const_iterator operator ++(int)
 {
 const_iterator iterator = *this;
 if(mCurrent <= mInterval->mUpperBound)
 mCurrent ++;
 return iterator;
 }
 const_iterator& operator --()
 {
 if(mCurrent <= mInterval->mUpperBound)
 mCurrent ++;
 return *this;
 }
 const_iterator operator --(int)
 {
 const_iterator iterator = *this;
 if(mCurrent <= mInterval->mUpperBound)
 mCurrent ++;
 return iterator;
 }
};

...
bool operator == (const interval&) const {...}
bool operator != (const interval&) const {...}

const_iterator begin() const;
const_iterator end() const;

bool empty() const {...}

interval() {...}
interval(int lowerBound, int upperBound) {...}
explicit interval(int lowerAndUpperBound) {...}
...
};

```

Expliquer quel est le type de l'itérateur (input, output, forward, bidirectional ou random access) ?

#### Question 5.4

Proposer une implantation des fonctions:

```
const_iterator begin() const;
const_iterator end() const;
```

de la classe interval.

Nous avons implanté le type `const_iterator` ainsi que les fonction `const_iterator begin()` `const` et `const_iterator end()` `const`. Nous devons désormais implanter les autres types:

| Type                         | Description                                                                                                              |
|------------------------------|--------------------------------------------------------------------------------------------------------------------------|
| <code>value_type</code>      | Type des valeurs stockées dans le conteneur (T)                                                                          |
| <code>reference</code>       | Type référence des valeurs stockées dans le conteneur (T&)                                                               |
| <code>const_reference</code> | Type référence non modifiable des valeurs stockées dans le conteneur ( <code>const T&amp;</code> )                       |
| <code>iterator</code>        | Itérateur référençant les valeurs stockées dans le conteneur et autorisant la modification de celles-ci                  |
| <code>const_iterator</code>  | Itérateur référençant les valeurs stockées dans le conteneur mais ne permettant pas de modifier le contenu du conteneur. |
| <code>size_type</code>       | Type permettant d'exprimer le nombre d'éléments stockés dans le conteneur ( <code>unsigned long</code> )                 |

**Rappel:** Pour définir un alias de type dans une classe, par exemple dans une classe `number` un type `float_type` qui est égal à `double` comme suit :

```
class number:
{
public:
 using float_type = double;

 float_type zero() const { return 0.0; }
};
```

Implanter les types qui n'ont pas été encore défini en utilisant des alias de types.

### Question 5.6

| Expression             | Type de retour         | Description                          |
|------------------------|------------------------|--------------------------------------|
| <code>c.empty()</code> | <code>bool</code>      | Aucun élément dans le conteneur      |
| <code>c.size()</code>  | <code>size_type</code> | Nombre d'éléments dans le conteneur. |

Il reste ces deux fonctions. La fonction `empty()` a déjà été implantée, il ne reste plus qu'à implanter la fonction: `size()`. Proposer le code qui plante la fonction `size()`.

## 6. Patrons

La classe `interval` est définie pour des entiers `int`. Cependant, il existe d'autres types d'entiers que les entiers de type `int`. En effet, on pourrait définir un interval pour des entiers non signés: `unsigned`, des petits entiers `short`, des grands entiers `long`, voir même des trs grand entiers `long long`.

Transformer la classe `interval` en la paramétrant par le type d'entier.

Pour rappel, le squelette de la classe est le suivant :

```
class interval
{
 ...
public:

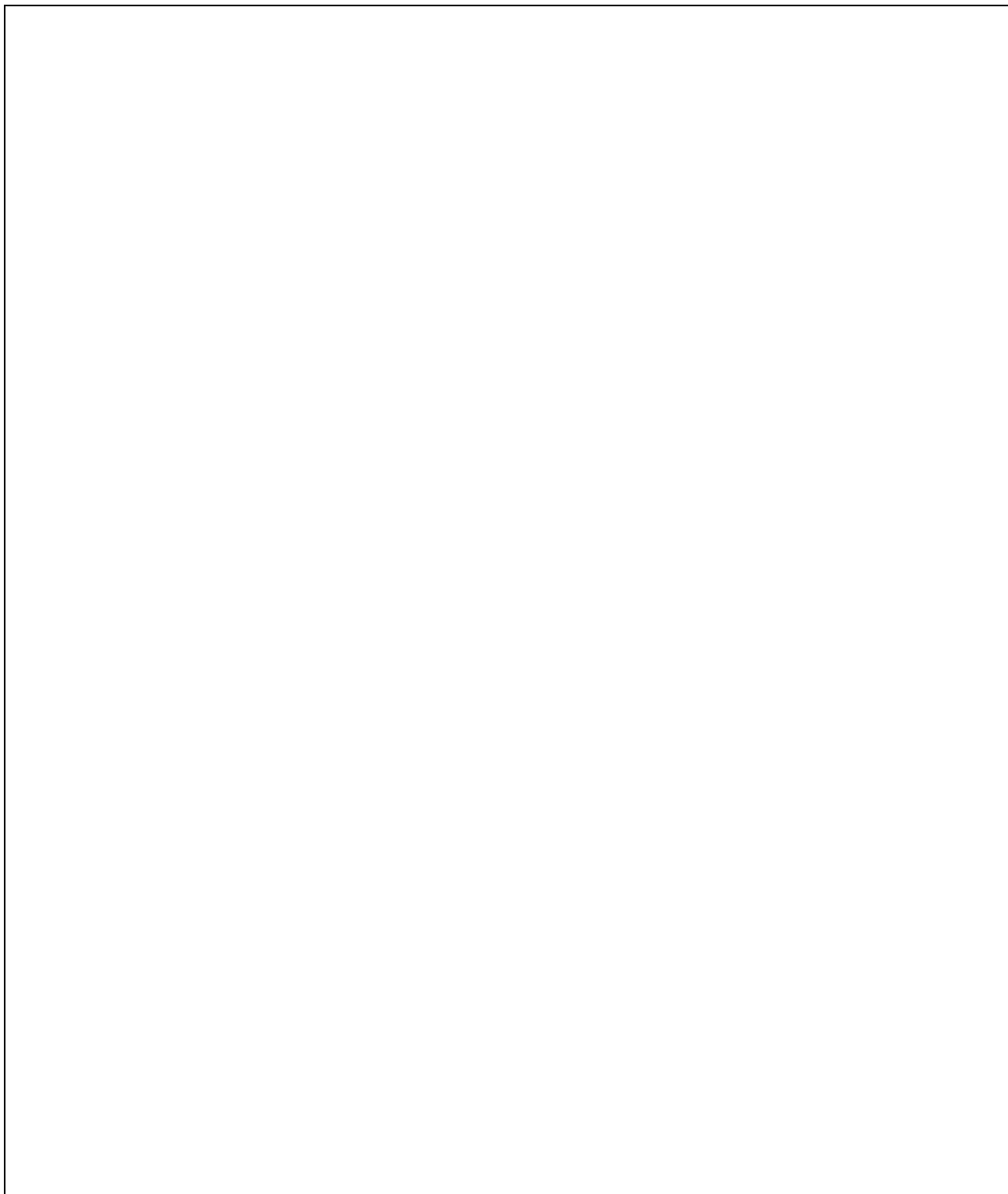
 using value_type=...;
 using reference=...;
 using const_reference=...;
 using iterator=...;
 using size_type=...;

 struct const_iterator:
 std::iterator<std::bidirectional_iterator_tag, // iterator_category
 int, // value_type
 int, // difference_type
 const int*, // pointer
 int>
 {
 [Implantation de la class const_iterator]
 };
 ...
 bool operator == (const interval&) const {...}
 bool operator != (const interval&) const {...}

 const_iterator begin() const;
 const_iterator end() const;

 bool empty() const {...}
 size_type size() const const {...}

 interval() {...}
 interval(int lowerBound, int upperBound) {...}
 explicit interval(int lowerAndUpperBound) {...}
 ...
};
```



### Question 6.2

Dites parmi les définitions suivantes :

- Celles qui sont correctes,
- Celles qui ne compilent pas.

| Instanciation                              | Compile |
|--------------------------------------------|---------|
| <code>interval&lt;int&gt;</code>           |         |
| <code>interval&lt;long unsigned&gt;</code> |         |
| <code>interval&lt;double&gt;</code>        |         |
| <code>interval&lt;std::string&gt;</code>   |         |

### Question 6.3

Nous souhaitons définir en C++20 des contraintes sur le type paramètre de la classe T.

Sachant que le prédicat: `std::is_integral` qui est défini dans `<type_traits>` est vrai si le type est un entier, comment pouvons-nous utiliser ce prédicat pour restreindre les paramètres de type aux seuls entiers ?

## 7. Exceptions

Le constructeur

```
interval(int lowerBound, int upperBound);
```

n'implante pas de vérification. Cependant, si `lowerBound` ne peut pas être plus grand que `upperBound`. Si c'est le cas, l'intervall créé n'est pas valide. Nous souhaitons que lors de la construction de l'intervall, nous vérifions que `lowerBound` n'est pas plus grand que `upperBound`. Si `lowerBound` est plus grand que `upperBound`, la construction de l'intervall doit échouer et nous devons lever une exception `std::out_of_range`.

### Question 7.1

Modifier le code du constructeur pour vérifier la condition et lever l'exception si la condition n'est pas validé.

### Question 7.2

Modifier l'entête de la fonction pour indiquer qu'une exception peut-être levée.