# 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 $\left[a,b\right]$ 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$<=$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. Containeur

Nous souhaitons que la classe interval soit un containeur. En effet, un object 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 containeur :

| Type | Description |
| --- | --- |
| value\_type | Type des valeurs stockées dans le containeur (T) |
| reference | Type référence des valeurs stockées dans le containeur (T&) |
| const\_reference | Type référence non modifiable des valeurs stockées dans le containeur (const T&) |
| iterator | Itérateur référençant les valeurs stockées dans le containeur et autorisant la modification de celles-ci |
| const\_iterator | Itérateur référençant les valeurs stockées dans le containeur mais ne permettant pas de modifier le contenu du containeur. |
| size\_type | Type permettant d’exprimer le nombre d’éléments stockés dans le containeur (unsigned long) |

| Expression | Type de retour | Description |
| --- | --- | --- |
| c.begin() | (const\_)iterator | Itérateur référençant le premier élément stocké dans le containeur |
| c.end() | (const\_)iterator | Itérateur référençant l’élément dénotant la fin de la séquence |
| c.empty() | bool | Aucun élément dans le containeur |
| c.size() | size\_type | Nombre d’éléments dans le containeur. |

### Question 5.1

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

**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 containeur 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 |
| --- | --- |
| value\_type | Type des valeurs stockées dans le containeur (T) |
| reference | Type référence des valeurs stockées dans le containeur (T&) |
| const\_reference | Type référence non modifiable des valeurs stockées dans le containeur (const T&) |
| iterator | Itérateur référençant les valeurs stockées dans le containeur et autorisant la modification de celles-ci |
| const\_iterator | Itérateur référençant les valeurs stockées dans le containeur mais ne permettant pas de modifier le contenu du containeur. |
| size\_type | Type permettant d’exprimer le nombre d’éléments stockés dans le containeur (unsigned long) |

**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 |
| --- | --- | --- |
| c.empty() | bool | Aucun élément dans le containeur |
| c.size() | size\_type | Nombre d’éléments dans le containeur. |

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 implante 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 refence=...;
 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 |
| --- | --- |
| interval<int> |  |
| interval<long unsigned> |  |
| interval<double> |  |
| interval<std::string> |  |

### Question 6.3

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

Sachang que le prédicat: [std::is\_integral](https://en.cppreference.com/w/cpp/types/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’interval créé n’est pas valide. Nous souhaitons que lors de la construction de l’interval, nous vérifions que lowerBound n’est pas plus grand que upperBound. Si lowerBound est plus grand que upperBound, la construction de l’interval 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.