IN204

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

Examen du 22 novembre 2022

B. Monsuez

NOM :	
PRENOM :	

Nous nous intéressons dans le cadre de ce sujet à la définition d'une classe **view** qui permet d'accéder à un sous-ensemble des éléments contenus dans un containeur, ce containeur doit supporter un accès par indexation.

D'une certaine manière, l'idée est d'offrir en C++ une fonctionnalité équivalente à celle de a[1:4] en Python qui retourne la séquence d'éléments débutants avec l'élément se situant en position 1 et se terminant avec l'élément se situant en position 4 et ce de manière générique pour un containeur supportant un accès par index.

Partie n°1: Définition d'une classe view pour un vecteur d'entier

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

```
#include<vector>

class view
{
private:
    std::vector<int>& m_container;  // Référence au container stockant les
valeurs.
    int m_first_index;              // Index de la première valeur de la vue.
    int m_last_index;               // Index de la dernière valeur de la vue.
```

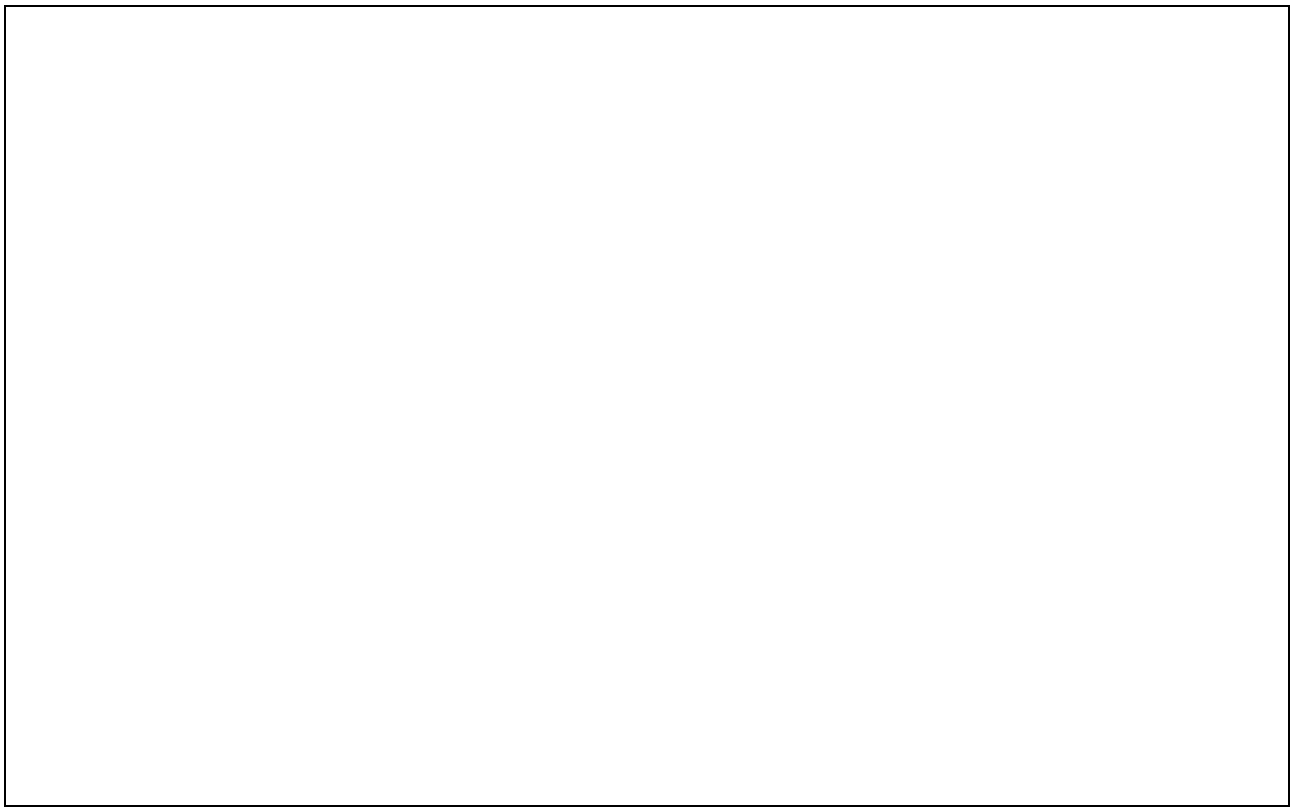
```
public:
    explicit view(std::vector<int>& vector);
    view(std::vector<int>& vector, int first_index, int last_index);
};
```

1. Les constructeurs

Question 1.1

Expliquer à quoi correspondent les déclarations suivantes :

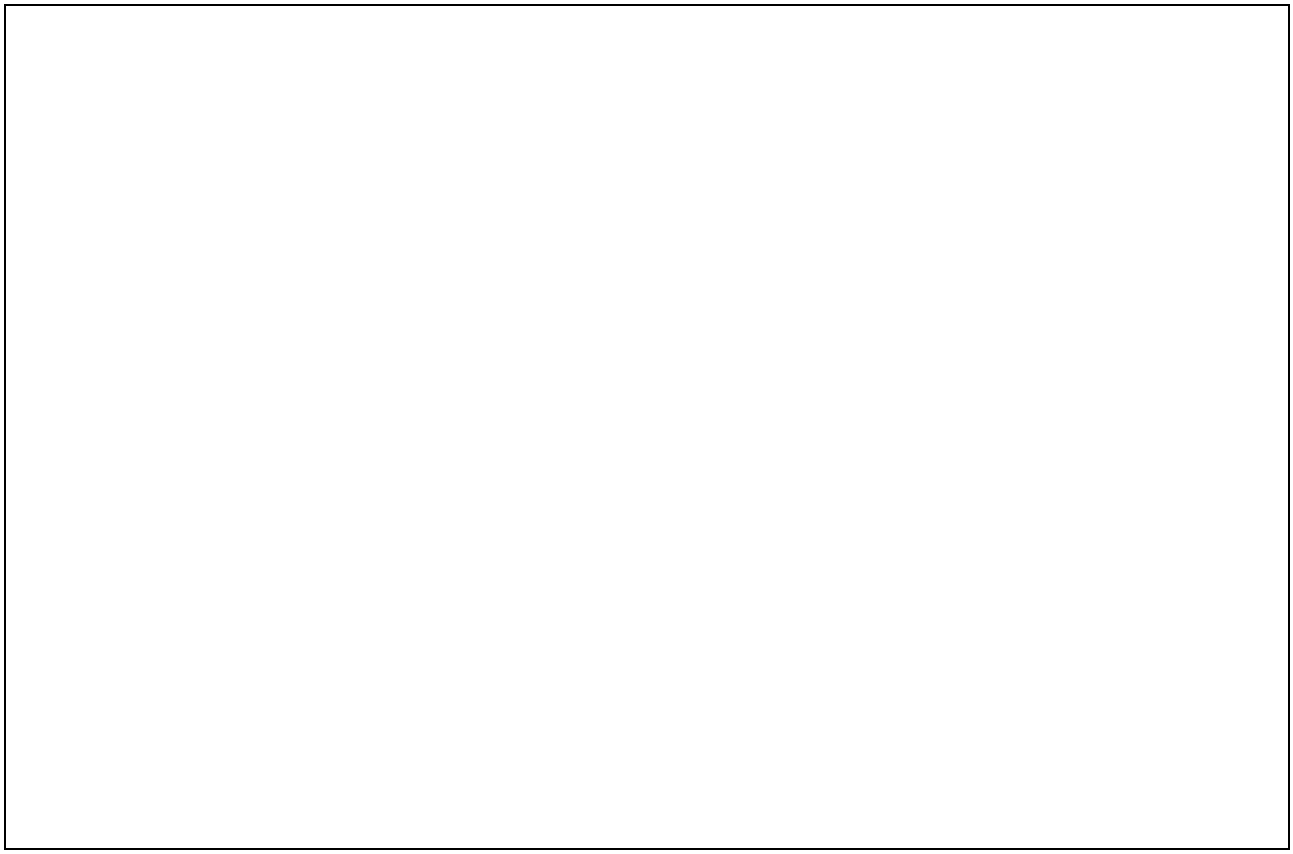
```
    explicit view(std::vector<int>& vector);
    view(std::vector<int>& vector, int first_index, int last_index);
```



Question 1.2

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

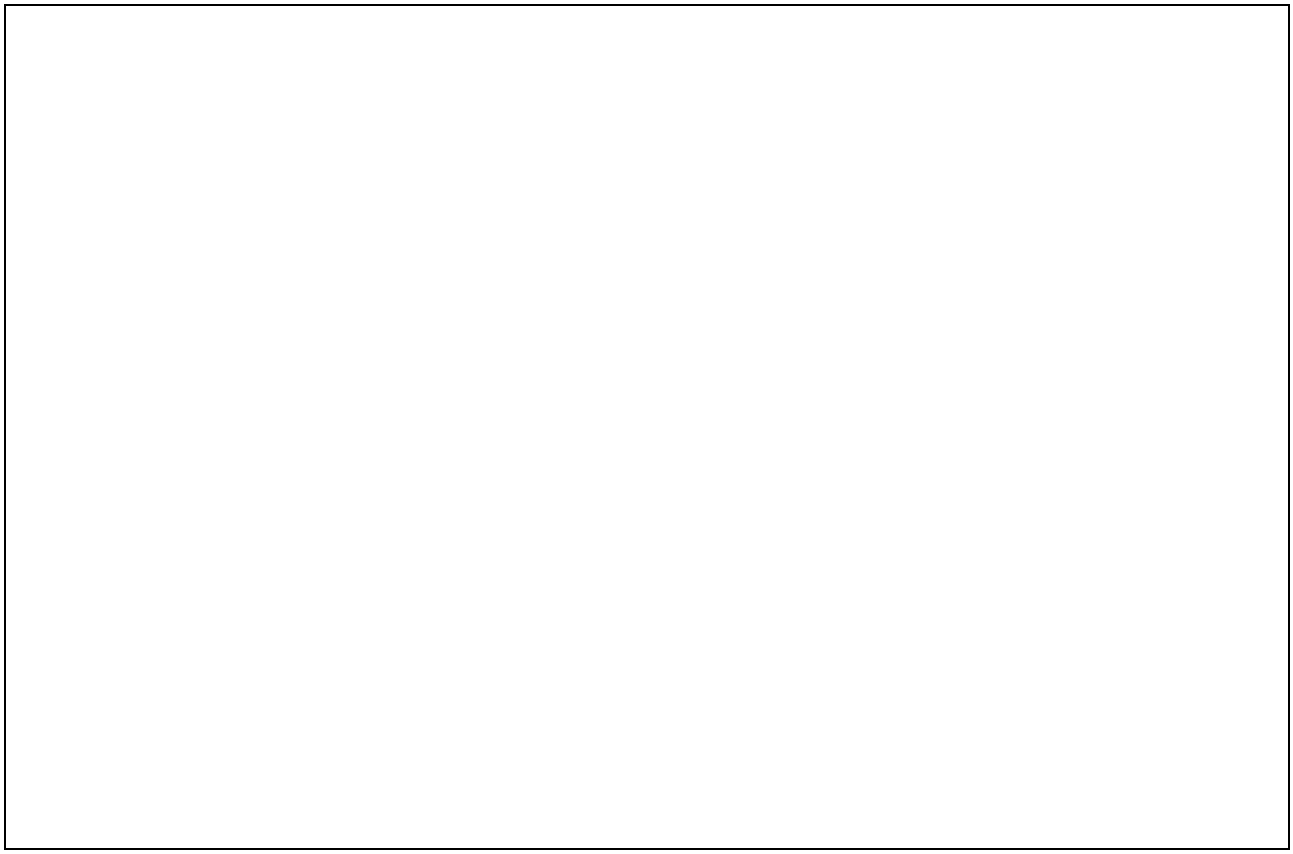
Remarque : Le code est minimaliste, on ne demande pas de vérifier si les paramètres `first_index` et `last_index` désignent des index valides vecteur.



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.



2. Les données stockées au sein de la classe

Question 2.1

Est-il possible d'accéder aux champs `m_container`, `m_first_index` et `m_last_index` en dehors de la classe view. Expliquer pourquoi ?

Question 2.2

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.

3. Opérateurs de comparaison

Nous souhaitons définir un opérateur qui détermine si deux objets **view** désignent la même séquence d'un même vecteur d'entiers.

Question 3.1

Proposer une implantation des deux opérateurs suivants :

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

4. Containeur

Nous souhaitons que la classe `view` soit un containeur. 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 4.1

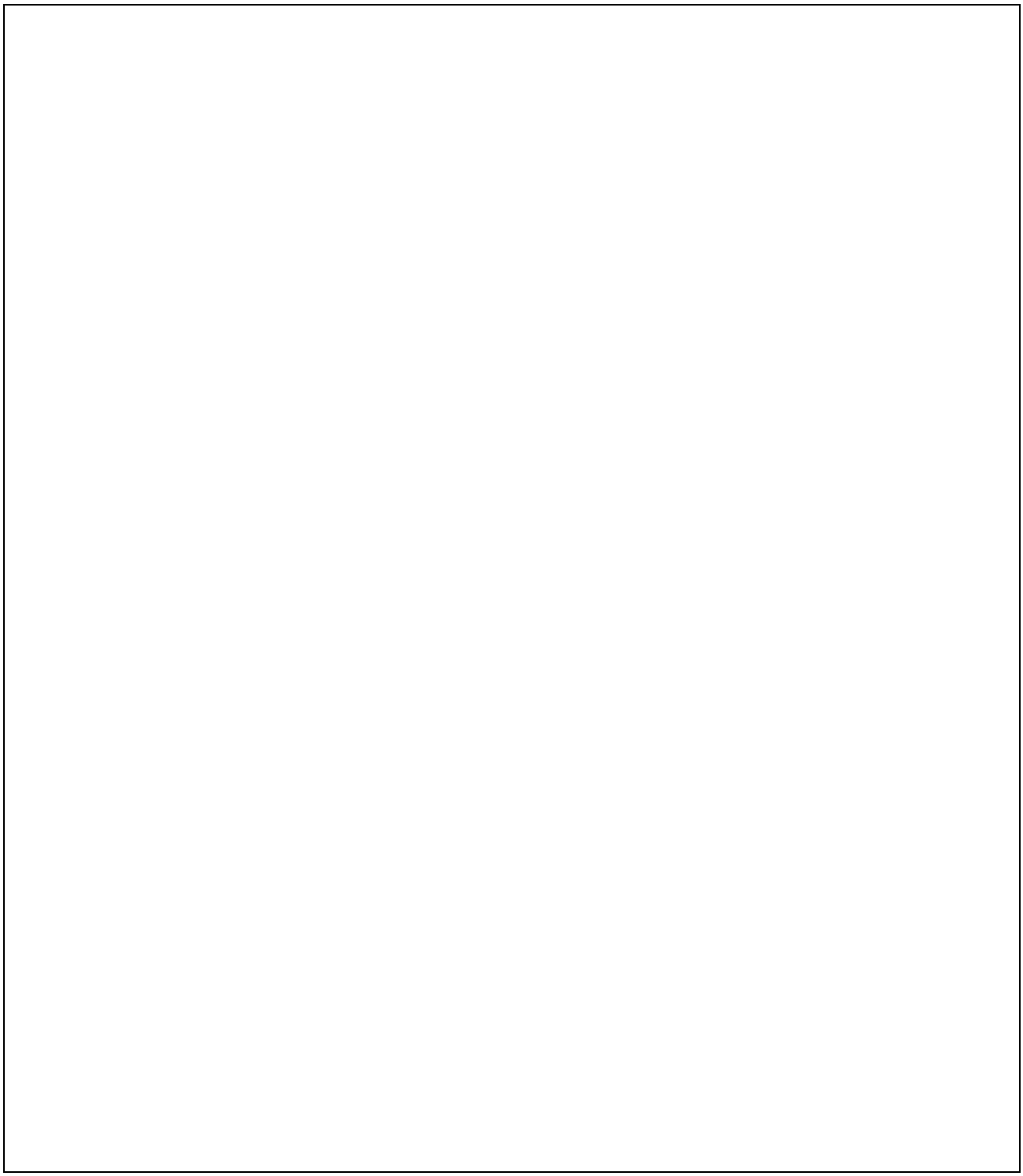
Nous souhaitons simplifier l'écriture et ne pas avoir à systématiquement recopier notre containeur d'origine qui est `std::vector<int>`, surtout que nous allons prochainement généraliser la classe à d'autres containeurs. Pour ce faire, nous souhaitons que la classe `view` expose un alias de type public qui se dénome : `container`. Définissez ce type dans la classe `view` afin qu'il désigne le type `std::vector<int>`.

Remplacer ensuite toutes les références à `std::vector<int>` par une référence à l'alias de type `container`.

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; }
};
```



Question 4.2

Les types des containeurs sont déjà définis dans le containeur `std::vector<int>`. Ainsi, il est possible de définir le type `value_type` en faisant référence au type `value_type` défini dans `std::vector<int>`.

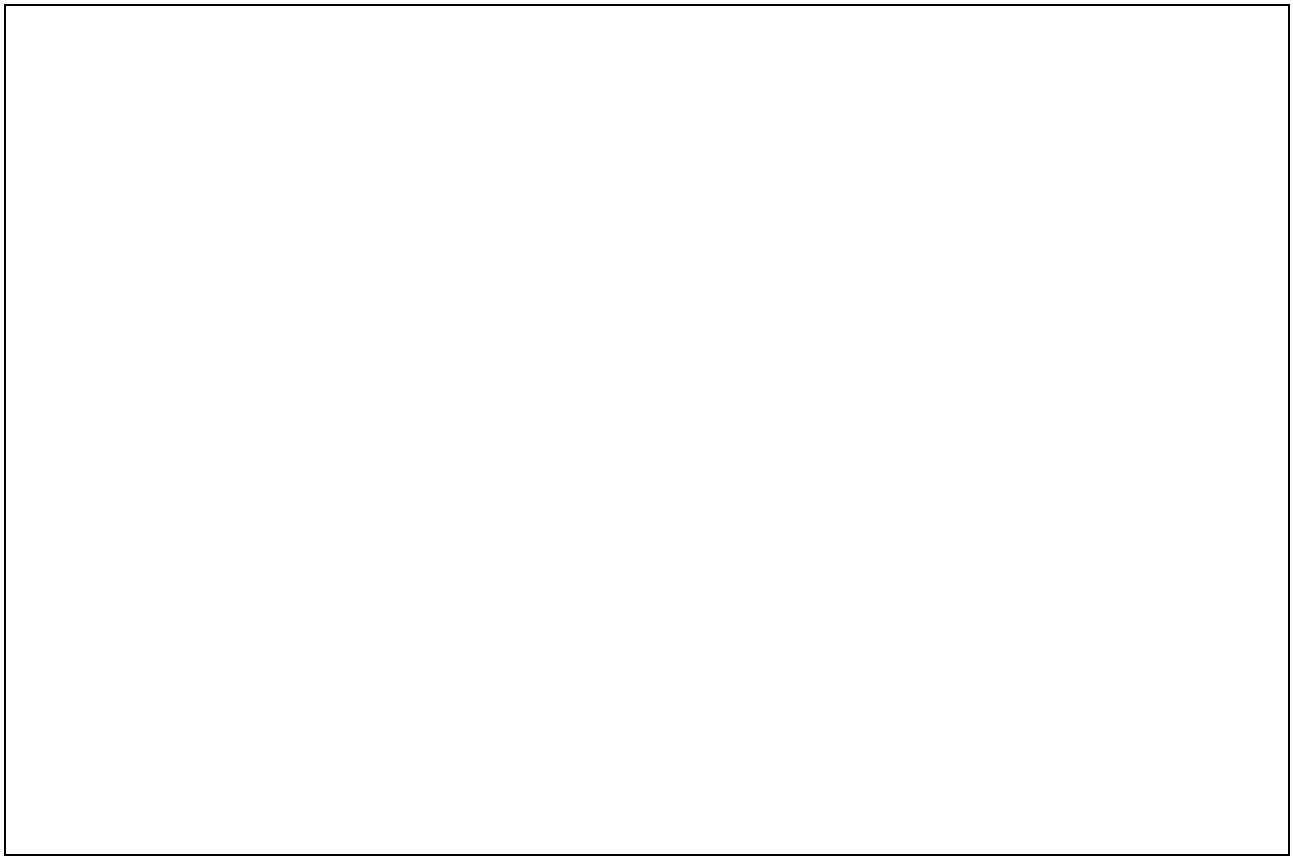
```
class view
{
public:
    ...
    using value_type = typename container::value_type;

private:
    container& m_container;
    int m_first_index;
    int m_last_index;
    ...
public:
    ...
};
```

Introduisez l'ensemble des types nécessaires en n'hésitant pas à faire référence aux types présents dans le `std::vector<int>`. Utilisez l'alias `container` en lieu et place de `std::vector<int>`, en effet cela évitera d'avoir à réécrire le code quand on généralisera le type à d'autres containeurs.

Question 4.3

Maintenant que les alias de types sont définis nous pouvons générer les méthodes que doit implanter un containeur tel que définit précédemment. Commencer par définir les méthodes `empty()` et `size()`.

Ensuite proposer une écriture des méthodes `begin()` et `end()`. (Ne pas oublier que le containeur peut-être accessible en lecture ou en lecture et en écriture.)



5. Patrons

La classe `view` est définie pour un containeur de type `std::vector<int>`. Cependant, cette classe peut aussi fonctionner avec un containeur de type `std::vector<float>` ou même un `std::array<std::string>`.

Question 5.1

Transformer la classe `view` en la paramètrant par le type du containeur qui stocke les éléments qui sont accédés par la classe `view`.

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

```
class view
{
class view
{
public:
    [ Définition du type container]
    ...
    using value_type = container::value_type;
    [ Définition des types iterator, const_iterator, size_type]
    ...
private:
    container& m_container;
    int m_first_index;
    int m_last_index;

public:
    explicit view(container& vector);
    view(container& vector, int first_index, int last_index);
    ...
};
```

Question 5.2

Dites parmi les définitions suivantes :

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

Instantiation	Compile
`view<std::vector<int>>`	
`view<std::array<int>>`	
`view<std::set<std::string>>`	
`view<int>`	
`view<int*>`	

Question 5.3

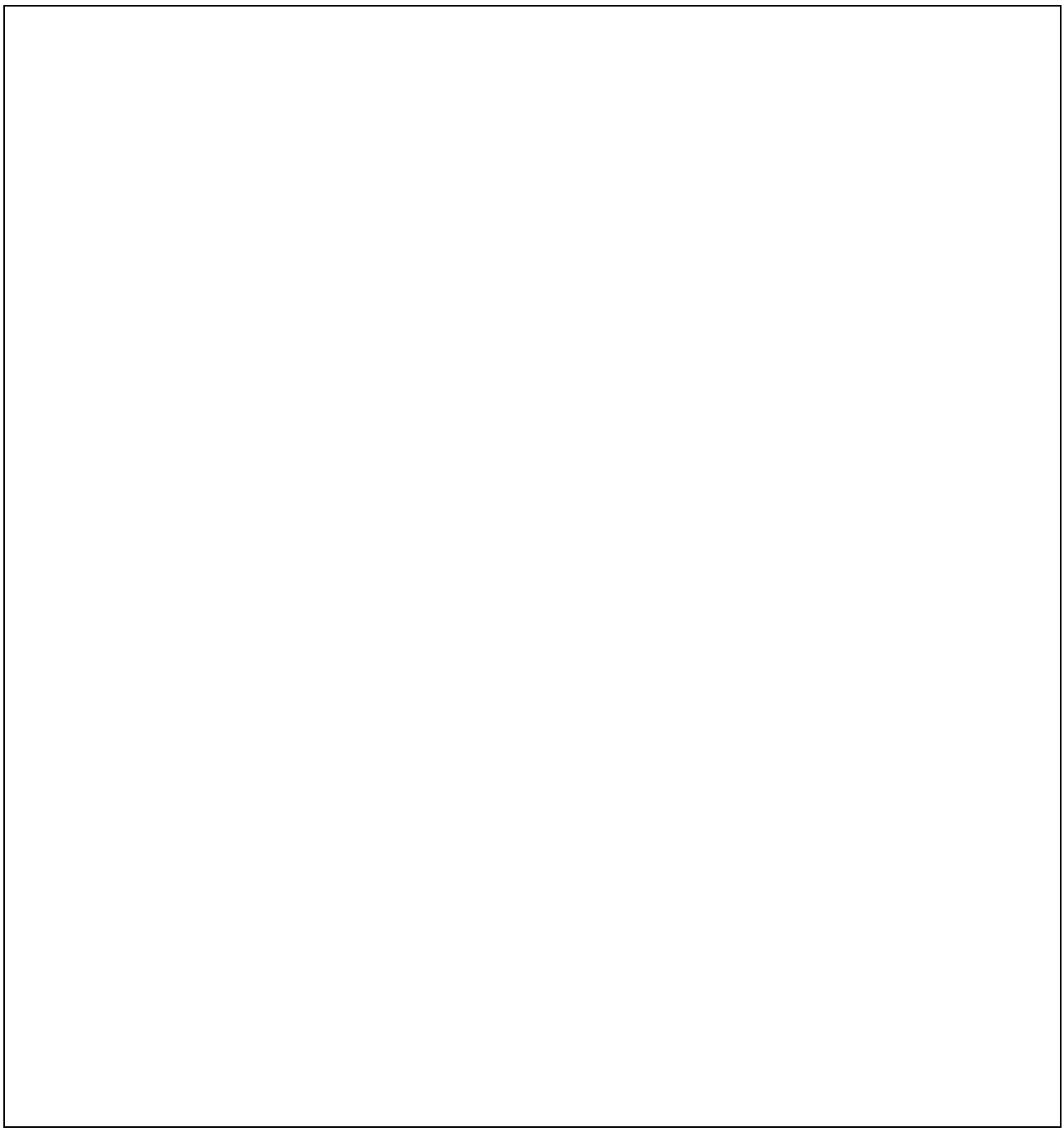
Nous souhaitons définir en C++20 des contraintes sur le type containeur. Définissez la liste des types ainsi que la liste des fonctions que le containeur doit exposer pour que l'instanciation se déroule correctement.

Définissez à partir de cela un `concept` dénommé `view_container` qui s'assure que les fonctions sont bien présentes.

Pour vous aider, nous rappelons les fonctions dont vous avez besoin. Le containeur doit fournir les types suivants : `value_type`, `reference`, `const_reference`, `iterator`, `const_iterator`, `size_type`. Ce même containeur doit fournir les fonction `begin` et `end`, ces fonctions doivent retourner `iterator`, le concept associé au random access itérateur est `std::random_access_iterator`. Donc `std::random_access_iterator<iterator>` doit-être vrai.

Nous rappelons la syntaxe des `concept` dans l'exemple suivant qui expose les méthodes pour une valeur entière qui supporte l'addition.

```
template<typename T>
concept addable = requires(T a, T b)
    // Indique qu'il est possible de créer deux valeurs a & b
    // ayant comme type T qui seront utilisées dans les clauses
    // suivantes.
{
    typename T::value_type; // Indique que
        // le type value_type doit être présent dans
        // la définition de T.
    { a + b } -> std::same_as<value_type>; // Indique que
        // l'expression a + b doit compiler et que la valeur
        // résultat a pour type value_type.
    { a == b } -> std::convertible_to<bool>; // Indique que
        // l'expression a == b doit compiler et que la valeur
        // résultat doit être convertible vers une valeur booléene.
    requires std::is_integral<T::value_type>;
        // indique que le type doit vérifier la contrainte std::is_integral
        // qui indique que le type est un entier.
};
```



6. Exceptions

Le constructeur de la classe `view`:

```
view(container& container, int first_index, int second_index);
```

ne génère aucune erreur si jamais `first_index` est inférieur à zéro ou plus grand que l'indice du dernier élément dans le containeur. L'erreur se produira quand l'on essayera d'accéder aux itérateurs. Il en va de même pour `last_index` qui peut accepter une valeur inférieure à zéro, plus grande que l'indice du dernier élément dans le containeur ou plus petite que le premier indice `first_index`. Nous souhaitons détecter ces cas d'erreur et générer une exception `std::out_of_range` pour signaler l'erreur d'initialisation de la classe.

Question 6.1

Modifier le constructeur pour qu'il vérifie que les indices `first_index` et `last_index` sont valides et si ce n'est le cas, génère une exception de type `std::out_of_range`.

7. Opérateur d'indexation

L'opérateur d'indexation est défini dans une classe comme:

```
template<typename T>
class indexed
{
    T& operator[](int index) { ... }
    T operator[](int index) const { ... }
};
```

Il existe deux versions de l'opérateur, l'une définie comme opération constante retournant la valeur associée à l'index, l'autre définie comme opération non constante permettant de modifier la valeur en renvoyant une référence sur la zone mémoire servant à stocker la valeur.

Question 7.1

Proposer un opérateur `operator []` pour la classe `view`.