IN204

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

Examen du 22 novembre 2022

B. Monsuez

|  |  |
| --- | --- |
| NOM : | Nunes Trofino |
| PRENOM : | Guilherme |

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] 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);

1e Déclaration:

Création d’un objet de class *view* avec la référence d’un autre objet de class *view* mais n’initialisé pas les valeurs d’int. Le mot clé **explicit** ne permit pas de conversions de type.

2e Déclaration:

Créatition d’un objet de class *view* avec la référence d’un autre objet de class *view* en initialisant les valeurs d’int.

### 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.

explicit view(std::vector<int>& vector);  
 view(std::vector<int>& vector, int first\_index, int last\_index);

1e Déclaration:

Explicit view( std::vector )

### 

### XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

### 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.

Il manque le constructeur vide qu’est normalement ajouté automatiquement donc il n’y a pas besoin de compléter la liste des constructeurs.

### 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 ?

Non, ce n’est pas possible directement à cause du modifier d’accès **private** utilisé pour les variables de la classe.

### 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.

    std::vector<int>& getContainer(void) {

        return m\_container;

    };

    int getFirstIndex(void) {

        return m\_first\_index;

    };

    int getLastIndex(void) {

        return m\_last\_index;

    };

## 3. Opérateurs de comparaison

Nous souhaitons définir un opérateur qui détermine si deux objets **view** désigne 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;  
...  
};

    bool operator == (const view& rView) const

    {

        view lView = \*this;

        std::vector<int>& rVector = rView.m\_container;

        int rEnd = rView.m\_last\_index;

        int rStart = rView.m\_first\_index;

        std::vector<int>& lVector = lView.m\_container;

        int lStart = lView.m\_first\_index;

        int lEnd = lView.m\_last\_index;

        if ((lEnd - lStart) != (rEnd - rStart))

            return false;

        bool equal = true;

        for(int i = lStart; i < lEnd; i++) {

            if (lVector[i] != rVector[i])

                equal = false;

        }

        return equal;

    };

## 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; }  
};

### #include<vector>

### template<typename T = std::vector<int>>  // default template value

### class view

### {

### public:

### using container = T;

### using containerInt = typename container::int;

### private:

### container& 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:

### using container& = std::vector<int>&;

### explicit view(container& vector) {

### m\_container = vector;

### m\_first\_index = 0;

### m\_last\_index = -1;

### };

### view(container& vector, int first\_index, int last\_index) {

### m\_container = vector;

### m\_first\_index = first\_index;

### m\_last\_index = last\_index;

### };

### container& getContainer(void) const {

### return this->m\_container;

### };

### int getFirstIndex(void) const {

### return this->m\_first\_index;

### };

### int getLastIndex(void) const{

### return this->m\_last\_index;

### };

### bool operator == (const view& rView) const

### {

### view lView = \*this;

### 

### container& rVector = rView.m\_container;

### int rEnd = rView.m\_last\_index;

### int rStart = rView.m\_first\_index;

### container& lVector = lView.m\_container;

### int lStart = lView.m\_first\_index;

### int lEnd = lView.m\_last\_index;

### if ((lEnd - lStart) != (rEnd - rStart))

### return false;

### bool equal = true;

### for(int i = lStart; i < lEnd; i++) {

### if (lVector[i] != rVector[i])

### equal = false;

### }

### return equal;

### };

### bool operator != (const view& rView) const

### {

### view lView = \*this;

### 

### container& rVector = rView.m\_container;

### int rEnd = rView.m\_last\_index;

### int rStart = rView.m\_first\_index;

### container& lVector = lView.m\_container;

### int lStart = lView.m\_first\_index;

### int lEnd = lView.m\_last\_index;

### if ((lEnd - lStart) != (rEnd - rStart))

### return false;

### bool equal = false;

### for(int i = lStart; i < lEnd; i++) {

### if (lVector[i] != rVector[i])

### equal = true;

### }

### return equal;

### };

### void empty(void) {

### int end = this->m\_last\_index;

### int start = this->m\_first\_index;

### for(int i = start; i < end; i++) {

### this->m\_container[i] = 0;

### };

### };

### int size(void) {

### return this->m\_last\_index - this->m\_first\_index;

### };

### int begin(void) {

### return this->m\_first\_index;

### };

### int end(void) {

### return this->m\_last\_index;

### };

### };

### 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.

using containerInt = typename container::int;

### 

### 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.)

    void empty(void) {

        int end = this->m\_last\_index;

        int start = this->m\_first\_index;

        for(int i = start; i < end; i++) {

            this->m\_container[i] = 0;

        };

    };

    int size(void) {

        return this->m\_last\_index - this->m\_first\_index;

    };

    int begin(void) {

        return this->m\_first\_index;

    };

    int end(void) {

        return this->m\_last\_index;

    };

## 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 accéder 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);  
 ...  
};

    template<typename T = std::vector<int>> // default template valueclass view

{

public:

using container = T;

using containerInt = typename container::int;

private:

container& m\_container; // Référence au container stockant les valeurs.

…

### 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\*> |  |

D’ici je n’avais pas beaucoup de temps pour bien réfléchir pour chaque question donc juste une idée générale sera donne.

1e correcte

2e correcte

3e je n’ai jamais utilisé cette commande. Je pense que marchera.

4e pas correcte, il n’arrive pas a faire des itérations pour une int

5e pas correcte, il n’arrive pas a faire des itérations pour une pointeur

### 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 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\_indexsont 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.

**T operator[](int index) const {**

**return this->m\_container[index];**

**};**