IN204
Programmation Orientée Objet – Les Itérateurs

Séance de Travaux Dirigés du 8 octobre 2019

B. Monsuez

Partie I – Manipulations d'itérateurs

Question n°1

Nous nous intéressons typiquement à la fonction suivante définie sur un tableau dynamique :

```
#include <vector>       // std::vector

template <class T>
int upper(std::vector<T> aVector, const T& theValue)
{
        for (int i = 0; i < aVector.size(); i++)
        {
                if (aVector[i] > theValue)
                        return i;
        }
        return -1;
}
```

Cette fonction retourne l'indice du premier élément du tableau qui est plus grand que theValue.

Question n°1.1

Modifier cette fonction pour qu'elle fonctionne désormais avec des itérateurs et non plus un tableau.

Question n°1.2

Tester cette fonction avec le code suivant :

```
int testUpper()
{
        int myints[] = { 10,20,30,30,20,10,10,20 };
        std::vector<int> v(myints, myints + 8);
        std::vector<int>::iterator up = upper(v.begin(), v.end(), 20);
        std::cout << "first value greater than 20  at position "
```

```
            << (up - v.begin()) << '\n';
        return 0;
}
```

Question n°2

Dans la STL, de nombreux algorithmes sont fournis dont notamment des algorithmes de tri. Ces algorithmes sont présents dans le fichier d'entête :

```
#include <algorithm>
```

Nous trouvons par exemple l'algorithme de tri :

```
template <class randomAccessIterator>
void sort(randomAccessIterator first, randomAccessIterator last);
```

ainsi que :

```
template <class randomAccessIterator>
void sort_heap(randomAccessIterator first, randomAccessIterator last);
```

qui fournisse un tri rapide et tri sur le tas.

Question n°2.1

Nous souhaitons déterminer la position du premier élément dans notre tableau qui est plus grand que 20 avec d'ordonner le tableau. Ceci correspond au code de la fonction définie à la question 1.2, et nous souhaitons ensuite déterminer la position du premier élément qui est plus grand que 20 après avoir ordonner le tableau.

Modifier le code de la fonction définie à la question 1.2 pour ajouter cette nouvelle opération en appelant soit la fonction `sort` soit la fonction `sort_heap` pour effectuer le tri de votre tableau.

Partie II – Création d'itérateurs

Nous considérons un ensemble de nombres entiers correspondant à un interval fermé entre [minValue, maxValue] défini par la classe suivante :

```
class interval
{
private:
        int minValue;
        int maxValue;

public:
        typedef int value_type;
        typedef size_t size_type;
        typedef ptrdiff_t difference_type;

        interval(int theMinValue, int theMaxValue) :
                minValue(theMinValue), maxValue(theMaxValue)
        {}
        interval(const interval& anotherInterval):
                minValue(anotherInterval.minValue),
                maxValue(anotherInterval.maxValue)
        {}
        interval& operator = (const interval& anotherInterval)
        {
                minValue = anotherInterval.minValue;
                maxValue = anotherInterval.maxValue;
                return *this;
        }
        size_type size() const
        {
                return (size_type)(maxValue - minValue);
        }
        int operator[](size_type anIndex) const
        {
                if (anIndex > size())
                        throw std::out_of_range("Index out of range");
                return minValue + (int)anIndex;
        }
        bool operator == (const interval& anotherInterval) const
        {
                return anotherInterval.maxValue == maxValue &&
                        anotherInterval.minValue == minValue;
        }
        bool operator != (const interval& anotherInterval) const
        {
                return anotherInterval.maxValue == maxValue &&
                        anotherInterval.minValue == minValue;
        }
};
```

Nous souhaitons pouvoir définir des itérateurs pour pouvoir itérer sur cet interval. Pour ce faire, nous devons ajouter au moins les méthodes suivantes et champs suivants :

```
        friend class interval_iterator;
        typedef interval_iterator const_iterator;
        const_iterator begin() const noexcept;
        const_iterator end() const noexcept;
```

En supposant que la classe itérateur s'appelle `interval_iterator`.

Il ne reste plus qu'à créer une classe `interval_iterator` dont le squelette serait le suivant :

```
class interval_iterator
        : public std::iterator <std::forward_iterator_tag, int>
{
private:
        friend class interval;
        const interval* mInterval; // Référence à l'interval.
        int mPosition; // -1 if denotes no more elements.

        interval_iterator(const interval& anInterval, int aPosition) :
                mInterval(&anInterval), mPosition(aPosition) {}

public:
        interval_iterator(const interval_iterator& anotherIterator):
                mInterval(anotherIterator.mInterval),
                mPosition(anotherIterator.mPosition) {}

        interval_iterator& operator = (interval_iterator& anotherIterator)
        {
                std::swap(mInterval, anotherIterator.mInterval);
                std::swap(mPosition, anotherIterator.mPosition);
                return *this;
        }

        const reference operator*() const;
        const pointer operator->() const;
        interval_iterator& operator++();
        interval_iterator& operator++(int);
        bool operator ==(const interval_iterator&) const;
        bool operator !=(const interval_iterator&) const;

};
```

Question 1.0 : complétez les classes `interval` et `interval_iterator` afin de pouvoir supporter les itérateurs.

Question 2.0 : Tester le bon fonctionnement de classe avec le code suivant :