

TD-TP : ALGORITHMES GENERIQUES

find, find_if, for_each, partition

Objectif : Manipuler des fonctions de la bibliothèque standard C++ sur des conteneurs C++.

Pour se centrer sur l'objectif ci-dessus, vous ne programmerez qu'un seul main.cpp, dans lequel vous coderez les procédures exempleDeFind(), exempleFindIf() et exempleForEachEtPartition(). Dans ce fichier main.cpp vous intégrerez au niveau global les quatre définitions ci-dessous.

```
// Définition de 4 types de données : Libelle, Reference, Prix et Poids
typedef string Libelle;
typedef string Reference;
typedef int Prix;
typedef int Poids;
```

1. LA METHODE map::find DANS map

La méthode `map::iterator find (clefRecherchée)` de la classe `map` cherche dans la section `[begin, end[` d'un `map` donné, une paire (key, value) telle que `key` égale `clefRecherchée`.

Cette méthode retourne un `iterator` (c.a.d un *pointeur*) sur une paire du `map` qui a la `clefRecherchée` comme `clef`, ou bien retourne la valeur `end` sinon.

Travail à faire : Définition d'un map dans lequel on va rechercher

Il s'agit d'écrire une procédure `exempleDeFind()`. Au fur et à mesure de sa rédaction, tester son code en l'appelant depuis le `main()`. Dans cette procédure `exempleDeFind()`. Vous devez :

- 1.1. Créer une classe `MapDeProduits` sur la base d'un `map`, telle que la `clef` est une référence du type `string` et la `valeur` est une pair composée d'un libellé et d'un prix.
- 1.2. Déclarer `unMapDeProduits` instance de cette classe.
- 1.3. Alimenter `unMapDeProduits` avec 3 produits différents.
- 1.4. Quelle serait la représentation UML de votre code ?

Travail à faire : Recherche dans le map avec la méthode find()

- 1.5. Lancer deux recherches sur `unMapDeProduits`. L'une sera fructueuse et on affiche le libellé et le prix du produit, l'autre sera non fructueuse et affiche le message « échec de recherche ». Pour cela :
 - Déclarer une `referenceRecherchee`
 - Déclarer un `iterator` existe pour récupérer le résultat de la recherche `find()`
 - Initier une recherche
 - Selon la valeur du résultat récupéré dans `existe`, afficher le message demandé ci-dessus.

2. LA FONCTION GÉNÉRIQUE `find_if` DANS `#include<algorithm>`

La fonction générique `iterator find_if (first, last, predicat)` retourne un `iterator` (cad pointeur) sur le 1^{er} élément de l'intervalle `[first, last[` d'un conteneur, pour lequel la propriété booléenne `bool predicat(*iterator)` vaut `true`. Si aucun élément ne vérifie cette propriété la fonction `find_if` retourne `last`. La fonction générique `find_if` est définie comme ci-dessous :

```
template <class Iterator, class Predicate>
Function  find_if (Iterator first, Iterator last, Predicate predicat) {
    while (first != last) {
        if (predicat (*first))    // Si *first satisfait la propriété
            return first;        // on retourne first
        ++first;                 // Sinon on progresse dans l'intervalle
    }
    return last; // Tout est parcouru, aucun élément n'a satisfait la propriété
}
```

Travail à faire : Création d'une liste de produits

- 2.1. Créer une classe `Produits`, définissant les attributs publics `référence`, `libellé`, `prix`, plus un constructeur et une méthode `toString()` pour ses instances.

Il s'agit maintenant d'écrire une procédure globale `exempleFindIf()`. Au fur et à mesure de sa rédaction, tester son code en l'appelant depuis le `main()`. Dans cette procédure vous devez :

- 2.2. Dans une fonction `exempleFindIf()`, créer `uneListeDeProduits` et l'alimenter avec 3 produits différents.

Travail à faire : Recherche dans la liste avec la méthode `find_if()`

Dans cette procédure `exempleFindIf()`, lancer deux recherches sur `uneListeDeProduits`. Pour cela :

- 2.3. Déclarer une variable globale (cad en dehors des fonctions) `referenceCherchee` du type `Reference`.
- 2.4. Ecrire une fonction booléenne `memeReference(Produit)` qui retourne `true` si l'attribut `laReference` du `Produit` passé en paramètre est égal à la valeur de la `referenceCherchee` ; et retourne `false` sinon.
- 2.5. Compléter la fonction `exempleFindIf()` en lançant deux recherches dans `uneListeDeProduits`. L'une sera fructueuse et on affiche le libellé et le prix du produit, l'autre sera non fructueuse et affiche un message d'erreur. Ces deux recherches sont lancées en utilisant la fonction générique `find_if()`

3. FONCTIONS GENERIQUES `for_each` ET `partition`

Travail à faire : Préparation de ressources logicielles

- 3.1. Créer une classe `Piece`, définissant les attributs publics référence, libellé, prix, poids plus un constructeur et une méthode `toString()` pour ses instances.
- 3.2. Ecrire une procédure globale `void afficher(Piece* unePiece)` qui affiche la pièce considérée.
- 3.3. Ecrire une procédure `void surPoids(Piece* unePiece)` qui augmente la prix de 10% uniquement si la pièce concernée pèse plus de 80 kilos.
- 3.4. Ecrire une fonction globale `bool tropCestTrop(Piece* unePiece)` qui retourne `true` si le prix de la pièce concernée dépasse 100 euros.

Travail à faire : Tester `for_each` et `partition`

Il s'agit maintenant d'écrire une procédure globale `exempleForEachEtPartition()`. Au fur et à mesure de sa rédaction, tester son code en l'appelant depuis le `main()`. Dans cette procédure vous devez :

- 3.5. Créer quatre instances différentes de la classe `Piece`, puis...
- 3.6. ... injecter l'adresse de chacune des pièces dans une `listeDePiece`
- 3.7. Etant donnée l'**Annexe** ci-après, utiliser la fonction générique `for_each` pour afficher toutes les pièces de la liste.
- 3.8. Utilise la fonction générique `for_each` pour appliquer `surPoids` à toutes les pièces de la liste.
- 3.9. Etant donnée l'**Annexe** ci-après, utiliser la fonction générique `partition` pour réorganiser la liste et placer en tête de liste les pièces dont le prix dépasse 100 euros.
- 3.10. Utiliser la fonction `for_each` pour afficher les pièces ayant un prix supérieur à 100 euros.
- 3.11. Utiliser la fonction `for_each` pour afficher les autres pièces.
- 3.12. Selon vous, pourquoi est-ce que la liste est composée de `Piece*` ?

Annexe

En plus des **classes génériques** dédiées aux conteneurs (list, map...), la bibliothèque standard de C++ propose des **algorithmes génériques** qui appliquent des traitements à des éléments stockés dans un conteneur, éléments qui sont accessibles par des itérateurs.

Présentation de la fonction générique `for_each`

L'algorithme générique `for_each` (first, last, fct) applique la fonction fct à chacun des éléments d'un conteneur qui sont entre les itérateurs [first, last[. La fonction fct doit nécessairement admettre un seul paramètre. Ce paramètre est obligatoirement du même type que les éléments contenus par le conteneur (i.e. pointés par la section [first, last[). Cet algorithme générique `for_each` est défini comme ci-dessous dans la bibliothèque standard C++ :

```
template <class Iterator, class Function>
```

```
Function for_each (Iterator first, Iterator last, Function fct) {
```

```
    Iterator it;
```

```
    for (it = first ; it != last; ++it ) // Parcours complet des éléments avec it
```

```
        fct (*it); // Applique la fonction fct à l'élément pointé par it
```

```
    return fct;
```

```
}
```

Exemple d'utilisation de la fonction générique `for_each`

L'usage d'une telle fonction générique suppose d'inclusion de `algorithm`. D'autre part, la fonction, nommée en troisième paramètre, doit être définie par le programmeur. Cette fonction admet un seul paramètre dont le type est le même que le type de celui des éléments contenus par le conteneur. Exemple de code :

```
#include <iostream>
```

```
#include <algorithm>
```

```
#include <list>
```

```
using namespace std;
```

```
// Fonction invoquée depuis la fonction for_each
```

```
void maFonction (int i) {
```

```
    cout << " " << i; // Manipulation - ici affichage - de l'élément
```

```
}
```

```
// Fonction principale
```

```
int main () {
```

```
    list<int> maListeInt; // Création d'un conteneur (cf. list de int)
```

```
    maListeInt.push_back(10); // Ajout trois éléments du type int dans le conteneur
```

```
    maListeInt.push_back(20);
```

```
    maListeInt.push_back(30);
```

```
    cout << "maListeInt contient : ";
```

```
    // Invocation de maFonction pour chacun des éléments de maListeInt (cf. de begin() à end() )
```

```
    for_each (maListeInt.begin(), maListeInt.end(), maFonction);
```

```
    // Ainsi, les éléments de maListeInt sont tous affichés
```

```
    return 0;
```

```
}
```

Présentation de la fonction générique partition

L'algorithme générique **partition** (first, last, bFct) réarrange les éléments localisés dans un conteneur entre [first, last[, de telle sorte que les éléments pour lesquels la fonction bFct retourne true précèdent tous les éléments pour lesquels la fonction retourne faux. La valeur retournée est l'itérateur qui pointe sur le premier élément du second groupe.

Le comportement de l'algorithme générique partition est défini comme ci-dessous en C++ :

```
template <class Iterator, class BooleanFunction>
Iterator partition (Iterator first, Iterator last, BooleanFunction bFct) {
    Itérateur it;
    // Réorganise les éléments en 2 partitions : ceux du début satisfont bFct, ceux de la fin « NON »
    while (true) {
        while (first != last && bFct(*first)) // Les éléments du début qui satisfont bFct ...
            first++;                          // ... restent à leur place
        if (first == last--)                 // Si tout est parcouru...
            break;                           // ... on a fini de réorganiser
        while (first != last && !bFct(*last)) // Les éléments de la fin qui ne satisfont pas bFct...
            last--;                          // ... restent à leur place
        if (first == last)                  // Si tout est parcouru...
            break;                           // ... on a fini de réorganiser
        swap (*first++, *last);              // Echanger les éléments et continuer la réorganisation
    }
    return first;                            // Retourne l'adresse du premier de la 2ème partition
}
```