Les modèles de bibliothèques standard

1. Problématique

- Une application est constituée de :
  - Noms de variables ayant un type donné « `T` » : ex. « `int` », « `double` », « `char` », etc.,
  - Conteneurs gérant les données décrites dans cette application : ex. « `vecteur` », « `liste` », etc.,
  - Et, finalement, des algorithmes appliqués sur ces données : ex. « `sort` », « `find` », etc.,

- Supposons que l'on a « `i` » types, « `j` » conteneurs et « `k` » algorithmes, pour concevoir l'application, on devrait écrire:

```
i*j*k versions de code!!!
```

- Avec l'apport des fonctions (ou classes) génériques (template), « `i = 1` », car un seul type « `T` » est traité, ce nombre est réduit à:

```
j*k
```

- Avec des algorithmes capables de travailler sur n'importe quel type de conteneurs: ex. « `sort` » pour trier un vecteur, liste, etc. ce nombre tombe à :

```
j+k
```

- Le but des STL est d'uniformiser le traitement, afin de réduire la prolifération de versions de code.

2. Définitions

- STL est un ensemble de classes qui, tout en collaborant, permettent de réaliser dans une catégorie de logiciels des conceptions réutilisables.

- Les STL :
  - utilisent les templates,
  - n'utilisent pas l'héritage et les fonctions virtuelles pour des raisons d'efficacité.

3. Composants

- Les STL contiennent 6 catégories principales:

3.1. Algorithmes

- Ils sont utilisés pour traiter les éléments d'un ensemble de données.
- Ils définissent une procédure informatique, ex: « `find` », « `sort` », « `copy` », etc.

3.2. Conteneurs

- Ils sont utilisés pour gérer une collection d'objets d'un type donné.
- Ils gèrent donc les structures de données ou bien ce sont des objets pour stocker d'autres objets.
- Les conteneurs peuvent être implantés comme des tableaux, listes chaînées, ou bien comme une clé spéciale pour chacun des éléments.

3.3. Itérateurs

- Ils fournissent aux algorithmes un moyen pour parcourir un conteneur (généralisation de la notion de pointeurs).

3.4. Fonctions objets

- Des classes qui redéfinissent l'opérateur d'appel de fonction ().

3.5. Adaptateurs

- C'est l'encapsulation d'un autre composant pour fournir une autre interface. Par exemple: d'un tableau peut dériver une pile, mais avec une interface réduite.

3.6. Allocateurs

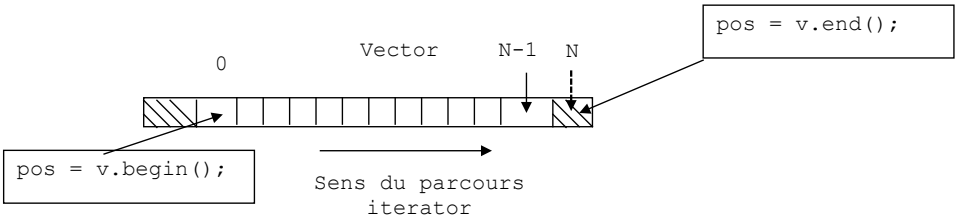
- Ils encapsulent le modèle de gestion de la mémoire. C'est une sorte de surdéfinition des opérateurs « `new` » et « `delete` ».

4. Itérateurs

- Les conteneurs ont des fonctions membres « `begin()` » et « `end()` » qui retournent un itérateur.
- Un itérateur est une sorte de pointeur permettant de parcourir un conteneur.

```
vector<int> v; // taille N

vector<int>::iterator pos;
```

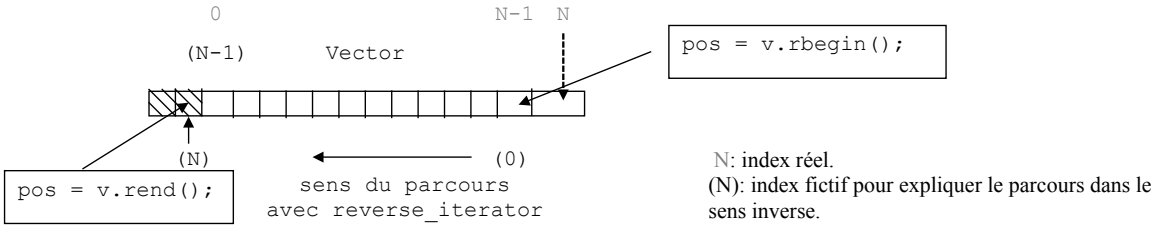
- « `pos` » est pointeur qui va parcourir un vecteur d'entiers.



- On peut parcourir aussi un conteneur à l'envers:

```
vector<int>::reverse_iterator inverse;
```

- Il est associé dans ce cas les fonctions permettant d'obtenir les positions dans le cas d'un parcours à l'envers : « `rbegin()` » et « `rend()` »



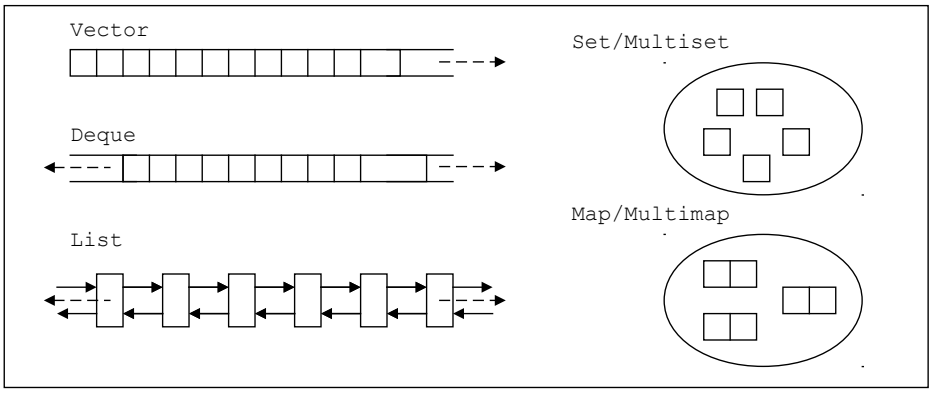
N: index réel.
(N): index fictif pour expliquer le parcours dans le sens inverse.

Particularités générales des itérateurs

- Si « `it` » est un itérateur, on peut toujours faire :
  - « `it++` » pour pointer l'élément suivant,
  - Déréférencer un itérateur, « `(*it)` » est l'élément pointé,
  - Comparer les itérateurs par « `==` » ou « `!=` ».
- Certains conteneurs permettent aussi à des itérateurs de faire :
  - « `it--` » pour pointer l'élément précédent,
  - « `it+n` » accès direct,
  - « `> <` » des tests de comparaison,
  - si « `ita` » et « `itb` » sont deux itérateurs sur le même conteneur, et on peut atteindre « `itb` » de « `ita` » alors l'expression « `ita, itb` » désigne l'intervalle `[ita,itb)`.

5. Conteneurs

- Il y a trois catégories de conteneurs :
  - Séquentiels : l'accès est linéaire, c'est l'ordre qui compte.
  - Associatifs : l'accès par clés.
  - Adaptateurs : l'accès est réglementé.



5.1. Conteneurs séquentiels

vector

- Un vecteur gère ses éléments dans un tableau dynamique.
- Il permet d'accéder directement à un élément à partir d'un index.
- Ajouter ou enlever des éléments à partir de la fin est une opération très rapide à exécuter.
- Par contre, insérer un élément au début ou bien au milieu du vecteur, prend un certain temps, car les éléments doivent être déplacés un par un pour créer une place au nouvel arrivant.

```
#include <iostream>
#include <vector>

using namespace std;

// Fonction générique pour afficher le contenu
// d'un vecteur.

template<class T>
void affiche(vector<T> v) {
    for(unsigned int i=0; i<v.size(); i++)
      cout<<v[i]<<" ";
    cout << endl;
}
```



```
int main() {

    vector<int> v1; // conteneur vide.
    vector<double> v2(6); // conteneur de 6 éléments égaux à zéro.
    vector<int> v3(10,999); // 10 éléments égaux à 999.
    int tab[4] = { 1, 2, 3, 4};

    // nouveau conteneur vector par recopie des éléments du tableau tab.
    vector<int> v4(tab,tab+4); // 1 2 3 4
    cout << "contenu du vecteur v4\n";
    affiche(v4);  // 1 2 3 4

    // la taille du conteneur v4.
    cout << "taille de v4 = " << v4.size() << endl; // 4

    // échanger les éléments: v4 devient v3, et vice versa.
    cout << "echange des elements: v4 vs. v3, avant\n";
    cout << "v3: ";
    affiche(v3); // v3 : 999 999 999 999 999 999 999 999 999 999
    cout << "v4: ";
    affiche(v4); // v4 : 1 2 3 4

    v4.swap(v3); // v4: 999 999 999 999 999 999 999 999 999 999
                 // v3: 1 2 3 4
    cout << "echange des elements: v4 vs. v3, apres\n";
    cout << "v3: ";
    affiche(v3); // v3: 1 2 3 4
    cout << "v4: ";
    affiche(v4); // v4: 999 999 999 999 999 999 999 999 999 999

    // destructeur de v4 est appelé. Maintenant size de v4 = 0.
    cout << "on detruit le vecteur\n";
    v4.clear ();

    cout << "taille de v4 = " << v4.size() << endl; // 0
```

```
        // conversion illégale: v2 (double) alors que v3 (int).
        // v2 = v3;

        // fixe une nouvelle taille du vecteur et initialise son contenu
        // avec une valeur donnée.
        cout << "avant la modification de la valeur d'un element donne\n";
        affiche(v3); // v3: 1 2 3 4
        v3.assign(2,5);
        cout << "apres\n";
        affiche(v3); // v3: 5 5

        // on insère au début, deux éléments identiques!
        // Opération à éviter, car prend "trop" de temps!
        cout << "avant l'insertion\n";
        affiche(v2); // 0 0 0 0 0 0
        v2.insert(v2.begin(),2,0.89); // 0.89 0.89 0 0 0 0 0 0
        cout << "apres l'insertion\n";
        affiche(v2); // 0.89 0.89 0 0 0 0 0 0

        // on insère à partir de la 3e position du début,
        // le contenu de du tableau tab des éléments de
        // la position 1 à 3, donc que 2 éléments.
        // Opération à éviter, car prend "trop" de temps!
        cout << "avant l'insertion\n";
        affiche(v2); // 0.89 0.89 0 0 0 0 0 0
        v2.insert(v2.begin()+2,tab+1,tab+3);
        cout << "apres l'insertion\n";
        affiche(v2);  // 0.89 0.89 2 3 0 0 0 0 0 0

        // on insère à partir de la fin, opération idéale à faire!
        cout << "avant l'insertion\n";
        affiche(v2); // 0.89 0.89 2 3 0 0 0 0 0 0
        v2.insert(v2.end(),2,28.99);
        cout << "apres l'insertion\n";
        affiche(v2); // 0.89 0.89 2 3 0 0 0 0 0 0 28.99 28.99
```



```
        // une autre façon pour insérer un élément, à la manière des piles!
        cout << "avant l'insertion avec push\n";
        affiche(v2); // 0.89 0.89 2 3 0 0 0 0 0 0 28.99 28.99
        v2.push_back(55.23);
        cout << "apres l'insertion avec push\n";
        affiche(v2); // 0.89 0.89 2 3 0 0 0 0 0 0 28.99 28.99 55.23

        // on retire le 2e élément du vecteur, le premier étant indexé par 0.
        cout << "avant le retrait\n";
        affiche(v2); // 0.89 0.89 2 3 0 0 0 0 0 0 28.99 28.99 55.23
        v2.erase(v2.begin()+1);
        cout << "apres le retrait\n";
        affiche(v2); // 0.89 2 3 0 0 0 0 0 0 28.99 28.99 55.23

        // on retire le dernier élément dans v4, un pop à la pile
        cout << "avant le retrait\n";
        affiche(v2); // 0.89 2 3 0 0 0 0 0 0 28.99 28.99 55.23
        v2.pop_back();
        cout << "apres le retrait\n";
        affiche(v2); // 0.89 2 3 0 0 0 0 0 0 28.99 28.99

        return 0;

    }
```

deque (DoubleEndedQUEue)

- C'est un tableau dynamique qui peut grossir dans les deux directions, une file bilatérale.

- Insérer des éléments au début ou à la fin sont deux opérations rapides à exécuter, par contre insérer un élément au milieu va prendre un certain temps, car il faudra déplacer les éléments.

```
// Il faudra inclure le fichier d'en tête <deque>
#include <deque>

using namespace std;

int main() {

     // conteneur deque contenant 4 éléments identiques,
     // valant 8.
     deque<int> v1(4,8); // 8 8 8 8

     return 0;
}
```

- Les fonctions disponibles pour la classe « `vector` » se retrouvent en majeure partie dans la classe « `deque` ».
- Par contre, la classe « `deque` » est enrichie de deux fonctions, « `push_front()` » et « `pop_front()` », vu que c'est une file bilatérale.
- Ces deux fonctions se comportent de la même manière que « `push_back()` » et « `pop_back()` » expliquées dans l'exemple du conteneur « `vector` ».

list

- C'est une liste doublement chaînée. Chaque élément de la liste a son propre segment de mémoire, et pointe son prédécesseur et son successeur.

- Les listes ne permettent pas un accès aléatoire, c.-à-d. à un index donné, comme le font les conteneurs « `vector` » et « `deque` ». Pour accéder au « `Nième` » élément de la liste, il faut passer par les « `N-1` » prédécesseurs.

- L'accès à un élément donné prend un temps linéaire nettement supérieur que si cet élément était dans un « `vector` » ou « `deque` ».

- L'avantage des listes est que l'insertion ou le retrait d'un élément se fait rapidement. Il suffit de modifier les adresses des pointeurs des prédécesseurs et successeurs.

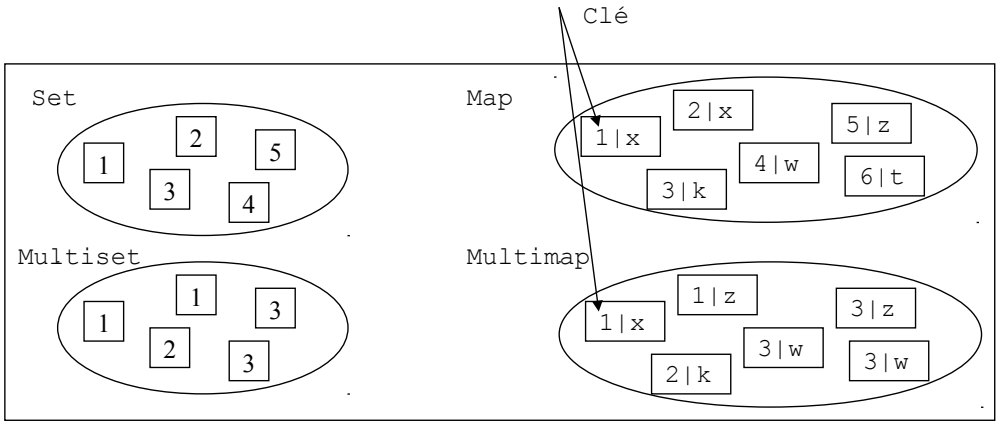
```
#include <iterator>
#include <iostream>
#include <list>

using namespace std;

template<class T>
void affiche(list<T> L) {
    // ici on copie le contenu de la liste de la position begin à end,
    // et envoie le résultat vers cout, en séparant les éléments
    // par un espace blanc.
    copy(L.begin(), L.end(),ostream_iterator<T>(cout," "));
    cout << endl;
}
```

```
int main() {

    // conteneur list contenant 4 éléments identiques du type int, et = 8.
    list<int> l1(4,8);
    affiche(l1); // 8 8 8 8
    l1.push_front(20);
    affiche(l1); // 20 8 8 8 8
    l1.push_back(56);
    affiche(l1); // 20 8 8 8 8 56
    l1.reverse();
    affiche(l1); // 56 8 8 8 8 20
    l1.erase(l1.begin());
    affiche(l1); // 8 8 8 8 20

    return 0;
}
```

5.2. Conteneurs associatifs

- Les éléments sont triés suivant un certain critère.
- Ce critère est une sorte de fonction qui compare suivant la valeur (le cas de « `set/multiset` ») ou bien la clé associée à la valeur (le cas de « `map/multipmap` »).
- Par défaut, cette fonction utilise l'opérateur de comparaison « < ». Cependant, une autre fonction de comparaison qui se base sur d'autres critères, peut-être définie pour réaliser cette opération.
- Nous avons donc deux catégories de conteneurs :

- `set/multiset` : ne contiennent que la valeur, ex: {jobs, gates, ellison}.
- `map/multimap` : une paire contenant une clé et une valeur associée à cette clé, ex: {(jobs,apple),(gates, microsoft),(ellison,oracle)}



- La différence entre « `set` » et « `multiset` » est que la valeur peut être dupliquée.
- La différence entre « `map` » et « `multipmap` » est que la clé peut être dupliquée.

set

```
// Il faudra inclure le fichier d'en-tête <set>
#include <set>

#include <iostream>
#include <iterator>


using namespace std;

int main() {
     // Définit un conteneur set avec ordre ascendant de ses
     // éléments (configuration par défaut).
     typedef set<int> IntSet;

     // Définit un type de conteneur avec ordre descendant de ses
     // éléments. Ici on a utilisé un autre critère de tri, on
     // pouvait définir aussi notre propre fonction et l'inclure à la place
     // de greater<int>.
     typedef set<int,greater<int> > IntGreatSet;

     IntSet coll1;
     // On insert des éléments dans coll1.
     coll1.insert(4);
     coll1.insert(3);
     coll1.insert(5);
     coll1.insert(1);
     coll1.insert(6);
     coll1.insert(2);
     // Pas grave si on duplique, la valeur ne sera pas prise en compte.
     coll1.insert(5);

     // Depuis C++11, il est permis d'écrire
     coll1.insert( {9,10,23,55});
```



```
     IntSet::iterator pos;

     // Affichage des éléments de coll1 // 1 2 3 4 5 6 9 10 23 55
     for (pos=coll1.begin(); pos != coll1.end(); ++pos) {
         cout << *pos << ' ';
     }
     cout << endl;

     // Création par recopie de coll2 à partir de coll1,
     // avec tri descendant.
     IntGreatSet coll2(coll1.begin(),coll1.end());

     // Copie vers le flux de sortie le contenu de coll2
     // 55 23 10 9 6 5 4 3 2 1
     copy (coll2.begin(),coll2.end(),ostream_iterator<int>(cout," "));
     cout << endl;

     // Enlève tous les éléments ayant la valeur 5.
     int num;
     num = coll2.erase(5);
     cout << "element(s) retire(s) du conteneur: " << num << endl; // 1

     // Efface tous les éléments jusqu'à l'élément ayant une valeur = 3
     coll2.erase (coll2.begin(),coll2.find(3)); // 3 2 1

     // Copie vers le flux de sortie le contenu de coll2 // 3 2 1
     copy (coll2.begin(),coll2.end(),ostream_iterator<int>(cout," "));
     cout << endl;
     return 0;
}
```

- On peut tester aussi si un élément se trouve déjà dans le conteneur.
- On utilise pour cela la notion de pair définie dans le fichier d'en-tête « `utility` »

multiset

- Ce qui va différer avec l'exemple précédent c'est la déclaration de « `multiset` » au lieu de « `set` » et la sortie du programme.

```
Il faudra inclure le fichier d'en-tête <set>
#include <set>


Dans l'exemple utilisant les « set », vous remplacez
les lignes suivantes:

typedef set<int> IntSet;
typedef set<int,greater<int> > IntGreatSet;

par

typedef multiset<int> IntSet;
typedef multiset<int,greater<int> > IntGreatSet;

Sortie:
1 2 3 4 5 5 6 9 10 23 55
55 23 10 9 6 5 5 4 3 2 1
élément(s) retiré(s) du conteneur: 2
3 2 1
```


map

```
// Il faudra inclure le fichier d'en-tête <map>
#include <map>

#include <iostream>
#include <string>

using namespace std;

int main() {

    /*
    crée un map / tableau associatif
    la clé est une string, représentant le nom du compositeur
    la valeur une string, représentant une de ses œuvres
    */

    typedef map<string,string> StringStringMap;

    StringStringMap compositeurs;

    // insérer quelques éléments.

    compositeurs["Vivaldi"] = "Les 4 saisons";
    compositeurs["Mozart"] = "La flute enchantee";
    compositeurs["Beethoven"] = "9";
    compositeurs["Bach"] = "La passion selon saint Matthieu"; // J.S.B
    compositeurs["Puccini"] = "Madama Butterfly";
```

```
        StringStringMap::iterator pos;

        // on peut accéder à chaque composante d'une paire clé/valeur par:
        // first: pour avoir la clé ; second: pour avoir la valeur associée.
        for (pos = compositeurs.begin(); pos!= compositeurs.end(); ++pos) {
            cout << "nom du compositeur: " << pos->first << "\t" << \
                                    "un echantillon: " << pos->second << endl;
        }
        cout << endl;

        // on copie directement les éléments de Bach vers "lui-même"
        // (JS: Jean Sébastien)
        compositeurs["Bach_js"] = compositeurs["Bach"];

        // on peut enlever un élément
        compositeurs.erase("Bach");

        for (pos = compositeurs.begin(); pos!= compositeurs.end(); ++pos) {
            cout << "nom du compositeur: " << pos->first << "\t" << \
                                    "un echantillon: " << pos->second << endl;
        }
        // on va chercher si un compositeur est dans la liste
        pos = compositeurs.find("Berlioz");

        // si on arrive à la fin sans l'avoir, cela veut dire qu'il n'a pas
        // été inclus dans la liste.
        if (pos==compositeurs.end()){
            cout << "Berlioz n'est pas dans la liste\n";
        } else {
            cout << "nom du compositeur: Berlioz\tun echantillon: " \
                                        << compositeurs["Berlioz"]  << endl;
        }
        return 0;
    }
```



```
>./06_maptst
nom du compositeur: Bach     un echantillon: La passion selon saint Matthieu
nom du compositeur: Beethoven un echantillon: 9
nom du compositeur: Mozart   un echantillon: La flute enchantee
nom du compositeur: Puccini  un echantillon: Madama Butterfly
nom du compositeur: Vivaldi  un echantillon: Les 4 saisons

nom du compositeur: Bach_js  un echantillon: La passion selon saint Matthieu
nom du compositeur: Beethoven un echantillon: 9
nom du compositeur: Mozart   un echantillon: La flute enchantee
nom du compositeur: Puccini  un echantillon: Madama Butterfly
nom du compositeur: Vivaldi  un echantillon: Les 4 saisons
Berlioz n'est pas dans la liste
>Exit code: 0
```

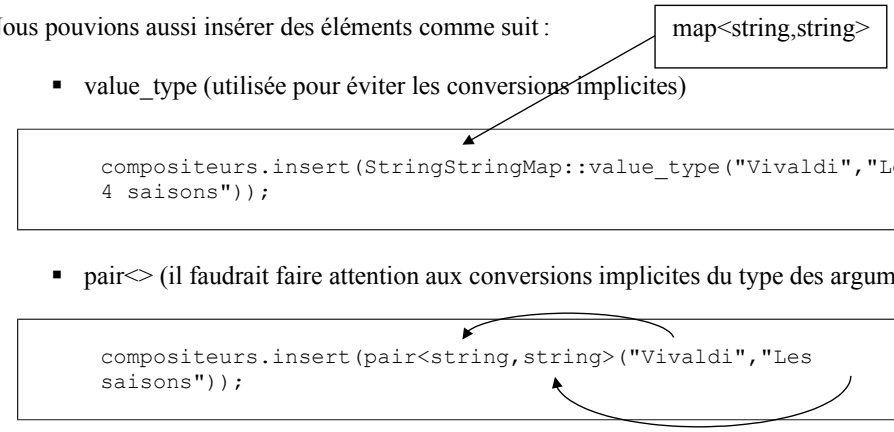
multimap

- Dans le cas des « multimap », la clé peut-être dupliquée.
- Ainsi pour la clé « mozart », avec les « multimap », on peut associer à cette clé d'autres noms de compositions.

- Il y a plusieurs manières pour insérer des éléments. Nous avons montré dans l'exemple précédent une insertion du style tableau associatif :

```
compositeurs["Vivaldi"] = "Les 4 saisons";
```

- Nous pouvions aussi insérer des éléments comme suit :



  - value_type (utilisée pour éviter les conversions implicites)

```
compositeurs.insert(StringStringMap::value_type("Vivaldi","Les
4 saisons"));
```

  - pair<> (il faudrait faire attention aux conversions implicites du type des arguments)

```
compositeurs.insert(pair<string,string>("Vivaldi","Les        4
saisons"));
```


  - make_pair() la plus simple à écrire, cependant faire attention aux conversions de type.

```
compositeurs.insert(make_pair("Vivaldi","Les 4 saisons"));
```

```
#include <iostream>

// Il faudra inclure le fichier d'en-tête <map>
#include <map>

#include <iostream>
#include <string>

using namespace std;

int main() {

    /*
    Crée un multimap
    La clé est une string, représentant le nom du compositeur.
    La valeur est une string, représentant une de ses oeuvres.
    */

    typedef multimap<string,string> StringStringMultimap;

    StringStringMultimap compositeurs;

    // Insérer quelques éléments.

    compositeurs.insert(make_pair("Vivaldi","Les 4 saisons"));
    compositeurs.insert(make_pair("Mozart","La Flute enchantee"));
    compositeurs.insert(make_pair("Beethoven","9"));
```

```
        compositeurs.insert(make_pair("Bach","La passion selon saint \
                                                    Matthieu" ));
        compositeurs.insert(make_pair("Puccini","Madama Butterfly"));
        compositeurs.insert(make_pair("Mozart","Don Giovanni"));
        compositeurs.insert(make_pair("Beethoven","Pour Elise"));
        compositeurs.insert(make_pair("Puccini","La Boheme"));

        // On déclare un itérateur.
        StringStringMultimap::iterator pos;

        // On peut accéder à chaque composante d'une paire, clé/valeur par:
        // first: pour avoir la clé ; second: pour avoir la valeur associée.

        for (pos = compositeurs.begin(); pos!= compositeurs.end(); ++pos) {
            cout << "nom du compositeur: " << pos->first << "\t" << \
                                "un echantillon: " << pos->second << endl;
        }
        cout << endl;

        // Afficher toutes les valeurs associées à la clé Beethoven

        string composer("Beethoven");
        cout << composer << ": " << endl;

        // lower_bound: retourne la première position où un élément avec
        // la clé composer doit être inséré.
        // upper_bound: retourne la dernière position où un élément avec
        // la clé composer doit être inséré.

        for (pos = compositeurs.lower_bound(composer);
                        pos != compositeurs.upper_bound(composer); ++pos) {
            cout << "\t" << pos->second << endl;
        }

        // Afficher toutes les clés associées à la composition no 9
        composer = ("9");
        cout << composer << ": " << endl;
```



```
        // On cherche dans toutes les valeurs de la position begin à end
        // toutes les clés associées à cette valeur.
        for (pos = compositeurs.begin(); pos != compositeurs.end(); ++pos) {
            if (pos->second == composer) {
                cout << "\t" << pos->first << endl;
            }
        }
        return 0;
    }
```

```
>./07_multimaptst
nom du compositeur: Bach     un  echantillon:  La  passion  selon
saint Matthieu
nom du compositeur: Beethoven     un echantillon: 9
nom du compositeur: Beethoven     un echantillon: Pour Elise
nom du compositeur: Mozart   un echantillon: La Flute enchantee
nom du compositeur: Mozart   un echantillon: Don Giovanni
nom du compositeur: Puccini  un echantillon: Madama Butterfly
nom du compositeur: Puccini  un echantillon: La Boheme
nom du compositeur: Vivaldi  un echantillon: Les 4 saisons

Beethoven:
     9
     Pour Elise
9:
     Beethoven
>Exit code: 0
```

5.3. Adaptateurs de conteneurs séquentiels

- Ces conteneurs adaptent les conteneurs séquentiels afin de remplir des missions précises.
- Parmi ces adaptateurs, nous citerons : pile (stack), file (queues) et les files prioritaires (priority queues).

- Nous allons donner un exemple uniquement dans le cas d'une pile.

- La classe stack contient une pile du type LIFO (LastInFirstOut: dernier arrivé, premier servi) :

  - push() insère un élément dans la pile,
  - pop() retire un élément de la pile,
  - top() retourne le prochain élément dans la pile.

```
// Il faudra inclure le fichier d'en-tête
#include <stack>

#include <iostream>

using namespace std;


int main() {
    // une pile contenant des entiers.
    stack<int> st;

    // on insère 3 éléments dans la pile.
    st.push(1);
    st.push(2);
    st.push(3);
```



```
    // on affiche l'élément se trouvant au sommet de la pile:
    cout << st.top() << ' ' ; // 3

    // on retire l'élément se trouvant au sommet de la pile.
    st.pop(); // 3

    // on affiche l'élément se trouvant au sommet de la pile.
    cout << st.top() << ' ' ; // 2

    // on retire l'élément se trouvant au sommet de la pile.
    st.pop(); // 2

    // on insère au sommet de la pile la valeur 77 on écrase
    // ainsi la valeur 1.
    st.top() = 77;

    // on insère deux éléments dans la pile.
    st.push(4);
    st.push(5);

    // on retire l'élément se trouvant au sommet de la pile: 5.
    st.pop(); // 5

    // tant que la pile n'est pas vide, on affiche son contenu.
    while (!st.empty()) {
        cout << st.top() << ' ' ;
        st.pop();
    }

    cout << endl;
    return 0;
}
```

```
>08_stacktst
    3 2 4 77
>Exit code: 0
```

- On peut aussi créer une pile à partir d'un conteneur, par exemple:

```
vector<int> v;

stack<int,v> st;
```

- On crée par recopie (élément par élément) la pile st à partir du conteneur vecteur v.

6. Algorithmes

- Les algorithmes peuvent être classés en fonction de la tâche à réaliser.

- On distingue principalement les 3 divisions suivantes :
  - Algorithmes non mutants: ne modifient pas les données (ordre ou valeur), par exemple: « `find` », « `find_if` », « `count` », etc.
  - Algorithmes mutants: modifient les données, par exemple « `reverse` », « `swap` », etc.
  - Algorithmes de tris: « `sort` », « `heap` », etc.

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <ctime>

using namespace std;

// Pour afficher les éléments.
void dump(int i){
    cout << i << endl;
}

// Pour détecter les nombres impairs.
bool odd(int i) {
    return i%2 != 0;
}
// Pour comparer deux éléments.
bool comp(const int& i1, const int& i2) {
    return i1>i2;
}
```

```
int main() {

    // On initialise le générateur pseudo-aléatoire, pour éviter que
    // rand() ne retourne les mêmes valeurs après le même
    // appel au programme.
    srand(time(NULL));

    // Un vecteur de 10 éléments = 0
    vector<int> v(10);

    cout << "------------\n";

    // On génère 10 valeurs aléatoires et on les stocke dans v.
    generate(v.begin(),v.end(),rand);

    // On affiche le contenu du vecteur
    for_each(v.begin(),v.end(),dump);

    cout << "------------\n";

    // On remplace tous les éléments impairs par 0.
    replace_if(v.begin(),v.end(),odd,0);

    // On trie les éléments par ordre décroissant.
    sort(v.begin(),v.end(),comp);

    // On affiche le contenu du vecteur.
    for_each(v.begin(),v.end(),dump);

    cout << "------------\n";

    return 0;
}
```



```
>./09_algotst
     ------------
     555477088
     1866868338
     1467343723
     1941032508
     1990781343
     1090602013
     698483628
     1377735820
     1812556201
     1100249352
     ------------
     1941032508
     1866868338
     1377735820
     1100249352
     698483628
     555477088
     0
     0
     0
     0
     ------------
>Exit code: 0
```

7. Fonctions objets

- Une fonction objet est une instance d'une classe qui définit l'opérateur parenthèse « `()` ».
- Chaque fois qu'une fonction objet est utilisée comme une fonction parenthèse, c'est son opérateur « `()` » qui est appelé.
- Généralement, un functor est plus « rapide » qu'un appel vers une fonction classique.

```
#include <iostream>
#include <vector>
#include <algorithm>

// Il faudra inclure le fichier d'en-tête
#include <functional>

using namespace std;

class PlusGrandQue {
    int valeur;
public:
    // par défaut la variable valeur = 5
    PlusGrandQue(int x=5):valeur(x) {}

    bool operator()(int val) const {return val > valeur;}
};
```



```
int main() {

    int tab[10] = {1,2,2,4,5,8,7,9,4,10};
    vector<int> v(tab,tab+10);

    vector<int>::iterator premier;
    premier = find_if(v.begin(),v.end(),PlusGrandQue());

    // 8 est la 1ère valeur > 5 (valeur par défaut)
    cout << *premier << endl;

    // not1 : l'inverse du critère.
    premier = find_if(v.begin(),v.end(),not1(PlusGrandQue()));

    cout << *premier << endl; // 1 est la première valeur <= 5
    return 0;
}
```

```
Sortie: 8
        1
```

- Fonctions prédéfinies : le langage fournit quelques fonctions prédéfinies comme « less<T> », « greater<T> », negate<T>, multiplies<T>, etc.

8. Trucs et astuces

8.1. Utiliser un opérateur pré incrément au lieu d'un opérateur post incrément

- Dans le chapitre 05, nous avons introduit la surcharge des deux opérateurs post et pré incrémentation.

```
// ++Compte
Compte Compte::operator++() {


    solde++;
    return (*this);
}
```

```
// Compte++
Compte Compte::operator++(int n) {

   Compte temp = *this;
   solde++;
   return temp;
}
```

- La différence entre les deux méthodes c'est la mémorisation de la valeur avant de l'avoir modifiée.
- Dans les exemples des boucles « for » utilisées, le fait d'avoir préservé cette valeur ne nous sert à rien, car cette valeur, non modifiée, ne sera pas utilisée.
- De ce fait, en utilisant « ++index » au lieu de « index++ » nous fera gagner du temps puisqu'on évite ainsi de créer une instance locale qui de toute façon elle ne sera pas utilisée.

8.2. Fixer la taille de votre conteneur

- Si vous avez une bonne connaissance de votre espace de travail, il est conseillé de fixer par vous même la taille de votre conteneur.
- Par exemple, « vector<string> vec(900); » va réserver « 900 » éléments du type « string ».

8.3. Pointeurs sur des objets

- Dans un tableau, il est préférable de stocker des pointeurs au lieu d'objets.

```
vector<UnObjet> Vecteur;
UnObjet obj = Vecteur[10];
```

- Appel de l'opérateur d'affectation de « UnObjet » pour faire la recopie de l'élément retourné par « Vecteur[10] ».
- Une perte de temps pour rien. Si nous avions stocké des pointeurs dans ce vecteur, il suffisait de recopier l'adresse du pointeur.

8.4. Initialiser les éléments

- Il est conseillé d'initialiser les éléments de votre conteneur pour éviter d'avoir des surprises.

```
string pardefaut = "";
vector<string> vecteur(120,pardefaut);
```

8.5. Vide ou pas vide ?

- Il est conseillé d'utiliser la méthode « empty » pour vérifier si un conteneur est vide ou pas. Cette méthode est à utiliser à la place du test classique si « `size==0` ».

9. Fonctions de hachage

- On aimerait stocker « N » éléments dans une table.
- Comment doit-on organiser ces éléments dans la table afin que d'une part de les trouver rapidement sans faire une recherche linéaire (parcourir un à un les éléments de la table) et d'autre part de limiter au maximum l'espace mémoire utilisé pour préserver les éléments de la table (éviter donc une table de taille « infinie »).

- Pour trouver un élément dans une table, il faudra donc rechercher son index si nous voulons éviter une recherche linéaire.
- Son index est déterminé en utilisant pour cela une fonction donnée.
- Par exemple, dans le cas des caractères, cette fonction peut retourner la valeur « Unicode » du caractère c.-à-d. « (int) c », qui représente l'index de l'élément dans la table.
- Pour une chaîne de caractères de longueur « n », c'est une fonction de tous les caractères de la chaîne.
- Par généralisation, pour un objet donné « x », l'appel « x.hashcode() » doit retourner un entier qui est l'index de l'objet dans la table.

- En conclusion pour localiser l'élément dans sa table, il faut donc savoir comment calculer une fonction de l'élément qui donne un entier qui sera l'indice dans le tableau.

- Il faudra avoir une bonne fonction qui minimise les indices identiques pour des objets différents.

- Si la fonction de hachage retourne l'index de la première lettre d'un mot, et si tous les mots de la table commencent par la même lettre, alors la recherche d'un élément dans la table se transforme à une recherche linéaire, l'utilisation dans ce cas du mécanisme de hachage n'a pas de sens.

- Soit « T » la taille du tableau.

- Tout élément dont l'index dépasse la valeur « T » sera calculé modulo cette taille.
- Ainsi nous aurons une meilleure distribution des éléments dans la table au lieu qu'ils soient complètement dispersés.

- Soit l'ensemble des éléments entiers strictement positifs « Z = {11, 59, 32, 44, 31, 26, 19} » et un tableau de taille « 10 ».
- Nous allons donc ranger les éléments de « Z » dans un tableau dont l'index varie entre « 0 et 9 (<10) ».

- La fonction de hachage pour l'élément « z_i » de l'ensemble « Z » est déterminée par :

$h(z_i) = z_i$ modulo [10].

z_i	$h(z_i)$
11	1
59	9
32	2
44	4
31	1
26	6
19	9

- On constate dans cet exemple une duplication d'indices : 1 pour 11 et 31 ; 9 pour 59 et 19.

- Plusieurs solutions sont possibles pour des éléments ayant le même indice : soit, utiliser un indice libre parmi les indices disponibles dans la table moyennant certaines règles à définir ; sinon utiliser des listes (chaînées) d'objets (« buckets ») associées à cet indice.

- Dans le cas de l'utilisation de « buckets », l'algorithme de recherche d'un élément dans la table se résume ainsi :

  - Obtenir l'index « i » dans la table
    - Calculer le hash code.
    - Réduire le hash code modulo la taille de la table pour obtenir l'index « i ».
  - Itérer dans le bucket à l'adresse « i ».
  - Pour chaque élément dans le bucket vérifier s'il est égal à « x », l'élément recherché.
  - Si on trouve un élément égal à « x » alors « x » est dans l'ensemble.
  - Autrement, « x » n'est pas dans l'ensemble.

i	**0**	***1***	**2**	**3**	**4**	**5**	**6**	**7**	**8**	***9***
t[i]		*11*	32		44		26			*59*
		31								*19*

- h(11) = h(31) = 1.

- En utilisant la technique de la première case vide à droite de l'élément, nous obtenons la représentation suivante :

i	**0**	***1***	**2**	**3**	**4**	**5**	**6**	**7**	**8**	***9***
T[i]	**19**	*11*	32	**31**	44		26			*59*

- Dans ce cas il faudra définir une fonction h qui tient compte de cette particularité.

10. Conteneurs associatifs et techniques de hachage

- Les conteneurs « set », « multiset », « map » et « multimap » stockent leurs éléments dans un ordre lexicographique. Les éléments sont stockés dans des arbres binaires.
- Auparavant, le standard C++ ne définissait pas de conteneurs permettant de stocker les éléments en utilisant les techniques de hachage.
- Il fallait utiliser pour cela des librairies externes définies par « SGI ».
- Plusieurs compilateurs « C++ » incluent dans leurs paquetages ces librairies externes.
- Elles sont définies dans le répertoire « ext » pour « externe ».

- Depuis le standard C++11, le langage a intégré des conteneurs dont les éléments sont rangés en utilisant les techniques de hachage.

Ordre lexicographique	SGI	C++11
set	hash_set	unordered_set
multiset	hash_multiset	unordered_multiset
map	hash_map	unordered_map
multimap	hash_multimap	unordered_multimap

- S'il est important que les éléments soient ordonnés dans un ordre particulier ; la version appropriée est celle qui ordonne ces éléments dans un ordre lexicographique.

unordered_set<Cle, FonctHash, CleEgal, Alloc>

```
Cle: le type de la clé et la valeur.
FonctHash : La fonction de hachage à utiliser (sinon celle par
défaut).
CleEgal : Elle détermine si deux clés sont égales.
Alloc : Pour la gestion interne de la mémoire.
```

```
##include <iostream>
#include  <unordered_set>
#include <string.h>

using namespace std;

struct egalitestr{
  bool operator()(const char* s1, const char* s2) const{
    return strcmp(s1, s2) == 0;
  }
};
```

```
void Cherche(const unordered_set<const char*, hash<const char*>,\
                 egalitestr>& Set,const char* mot){

  unordered_set<const char*, hash<const char*>,\
                egalitestr>::const_iterator it = Set.find(mot);

  cout << mot << ": "\
       << (it != Set.end() ? "present" : "n'est pas present")\
       << endl;
}

int main(){
  unordered_set<const char*, hash<const char*>, egalitestr> Set;
  Set.insert("kiwi");
  Set.insert("prune");
  Set.insert("pomme");
  Set.insert("mangue");
  Set.insert("abricot");
  Set.insert("banane");

  Cherche(Set, "mangue");
  Cherche(Set, "pomme");
  Cherche(Set, "salade");

  return 0;
}
```

- La même approche est à utiliser pour les autres conteneurs associatifs de cette catégorie.

11. Pointeur intelligent

11.1. Généralités

- L'utilisation du pointeur intelligent (« smart pointer ») permet de résoudre les problèmes qu'on rencontre fréquemment lors du désign ou le codage d'un programme en « C++ ».

```
 void f(){

     int* ptr = new int[100];
     /* …… plusieurs lignes de code …… */
     delete [] ptr ;
 }
```

- Quand on a affaire à 3 lignes de code, oublier un appel à « delete » est une chose rare.
- Mais quand on a affaire à un code complexe sur plusieurs lignes, un oubli est vite arrivé !
- Une façon de corriger ce problème est l'utilisation du mécanisme d'exceptions.
- Mais cette façon de procéder est trop complexe et demande une certaine doigtée. Il faudra capturer l'erreur associée à une fuite de la mémoire. La traiter. Puis relancer le processus avec un second « throw » pour permettre au programme de poursuivre « normalement ».

- Une autre solution est d'utiliser un pointeur « intelligent ». Ce dernier, associé à un élément donné, permet de garder une trace en mémoire de cet élément, afin de le libérer au moment opportun.
- Cette approche est simple à utiliser et en plus elle va permettre d'éviter les fuites de mémoire.

- Le langage « C++ » a fourni un premier pointeur intelligent dans le standard « C++98 ». Il s'agit de « auto_ptr ».

- Malheureusement, la classe définissant un tel pointeur n'avait pas défini un constructeur de recopie et un opérateur d'affectation. Son utilisation dans les conteneurs, par exemple « vector », était chaotique.

- Le standard « C++11 » a apporté une nouvelle approche dans la définition et l'utilisation d'un pointeur intelligent.

11.2. Description des pointeurs intelligents

- Le standard C++11 a défini plusieurs variantes de pointeurs intelligents :

    - « unique_ptr » : un élément donné est associé à un seul pointeur du type « unique_ptr ». La propriété est unique.

    - « shared_ptr » : plusieurs pointeurs du type « shared_ptr » peuvent se partager un seul élément. La propriété dans ce cas est partagée.

    - « weak_ptr » : c'est une référence faible à un « shared_ptr ». Il permet de corriger les problèmes cycliques susceptibles d'être provoqués par les « shared_ptr ».

    - L'utilisation de ces pointeurs nécessite l'inclusion du fichier d'en-tête « <memory> ».

- « unique_ptr »

```
void f(){

    unique_ptr<T> pt(new T);

    /* …… plusieurs lignes de code …… */

}
```

- Le pointeur « unique_ptr » est défini dans l'espace de nom « std ». Il faudra donc l'inclure, sinon faire un appel explicite.

- L'appel explicite à « delete » dans le programme (ou la fonction) n'est plus nécessaire.

- Le pointeur va se charger, peu importe comment le programme prendra fin (sortie normale, ou bien un plantage), de libérer l'espace mémoire alloué.

- Par rapport aux deux autres pointeurs « shared_ptr » et « weak_ptr », « unique_ptr » permet de gérer un tableau de pointeurs intelligents.

11.3. Utilisation de « unique_ptr »

- Un pointeur « unique_ptr » lorsqu'il gère un élément, il devient propriétaire unique.
- Cet élément n'est plus géré par un ou plusieurs autres pointeurs.

```
unique_ptr<int> px(new int); // allocation
*px = 10; // affectation

// Ok, p2 à la propriété du pointeur
unique_ptr<int> p2 = std::move(px);

p2.reset(); //efface la memoire
px.reset(); //aucun effet
unique_ptr<int> p3(new int(10)); // allocation

// p4 accède au même pointeur que p3
// attention p3 n'a pas lâché le pointeur
int* p4 = p3.get();

// p5 accède au même pointeur que p3
// attention p3 a lâché le pointeur
int* p5 = p3.release();
p3.reset(); //aucun effet
delete p5; // attention, ne pas oublier de le faire

// On peut utiliser ce pointeur pour un tableau
unique_ptr<int[]> py(new int[10]);
py[0] = 20;
```


11.4. Éviter les dégâts

- « unique_ptr » a un constructeur explicite qu'il faudra appeler.

```
unique_ptr<int> px(new int); // Ok
std::unique_ptr<int> p1 = px; // Erreur

unique_ptr<int> pt(new int(10)); // Ok
unique_ptr<int> pt = new int(2); // Erreur.

unique_ptr<int> py(new int(10)); //Ok
unique_ptr<int> pt2(py); // Erreur
```

- Il n'y a pas d'arithmétique sur les pointeurs « unique_ptr »

```
unique_ptr<int> pt(new int(10));
*pt = 20; // OK
++pt; // Erreur
```

- Comme nous l'avons mentionné, le constructeur de recopie n'a pas été défini pour « unique_ptr ». On a accès qu'à ses fonctionnalités : « get », « release » ou « move ».

- Voir l'exemple associé à l'utilisation de « unique_ptr » dans le cas des instances de classes.

11.5. Ajout d'un « deleter »

- On peut définir avec « unique_ptr » une méthode spécifique à appeler pour libérer l'espace mémoire alloué par « unique_ptr ».

```
class un_deleter {
public:
     template <class T>
     void operator()(T* p) {
          cout << "T" << endl;
          delete p;
     }
};
int main () {

     std::unique_ptr<int> p;

     std::unique_ptr<int,un_deleter> alpha (new int);
     std::unique_ptr<int,un_deleter>
                         beta (new int,alpha.get_deleter());

     p.reset(new int);
     alpha.reset(new int);
     beta.reset(new int);

     return 0;
}
```


11.6. Utilisation de « shared_ptr »

- Un « shared_ptr » partage la propriété d'un objet avec d'autres pointeurs.
- L'objet n'est détruit que lorsque toutes les références qui lui sont associées sont détruites.

```
// spt5 est pointeur partagé
shared_ptr<int> spt5(new int);
*spt5 = 88;

// spt6 partage le même objet avec spt5

shared_ptr<int> spt6 = spt5;

// use_count affiche combien de références sont associées à
// l'objet pointé
// On affiche 2 (spt5 et spt6)

cout << "compteur_6: " << spt6.use_count() << endl;

// Un nouveau pointeur partagé initialisé avec spt6

shared_ptr<int> spt7;
spt7 = spt6;

// On affiche 3 (spt5, spt6 et spt7)
cout << "compteur_7: " << spt7.use_count() << endl;
```

```
// Pointent-ils le même objet?

cout << "spt5: " << spt5 << " ; spt6: " << spt6 \
     << " ; spt7: " << spt7 << endl;


// Contiennent-ils la même valeur? (88)

cout << "spt5: " << *spt5 << " ; spt6: " << *spt6 \
     << " ; spt7: " << *spt7 << endl;
```

- L'utilisation d'un pointeur partagé peut poser un problème de références cycliques.
- Dans l'exemple qui suit, la classe A contient un pointeur partagé sur B et la classe B contient un pointeur partagé sur A.
- Dans la fonction « main », nous avons déclaré deux pointeurs partagés et nous avons initialisé les pointeurs partagés internes avec ces pointeurs.
- Quand l'objet est associé au pointeur partagé, une première référence est créée. Quand ce pointeur est affecté au pointeur partagé interne, une seconde référence est créée.
- Quand la fonction « main » prend fin, les variables « sptrA » et « sptrB » cessent d'exister. Les premières références sont ainsi donc détruites. Mais quand est-il des secondes références? Fuite mémoire en perspective.


```
class B;
class A {
public:
     A() : m_sptrB(nullptr) { };
     ~A() {
          cout<<" A est détruite"<<endl;
     }
     shared_ptr<B> m_sptrB;
};
class B {
public:
     B() : m_sptrA(nullptr) { };
     ~B(){
          cout<<" B est détruite"<<endl;
     }
     shared_ptr<A> m_sptrA;
};

int main( ) {
     shared_ptr<B> sptrB( new B ); // 1e reference shared
     shared_ptr<A> sptrA( new A ); // 1e reference shared
     sptrB->m_sptrA = sptrA; // 2e référence shared
     sptrA->m_sptrB = sptrB; // 2e référence shared
     return 0;
}
```

- Pour corriger le problème, nous utilisons un pointeur faible « weak_ptr ».
- Sa tâche est de garder une trace sur l'objet pointé par « shared_ptr » sans augmenter la valeur de la référence associée.
- On reprend le même programme et on remplace les déclarations des pointeurs « m_sptrA » et « m_sptrB » comme suit :

```
weak_ptr<B> m_sptrB; // Dans la classe A

weak_ptr<A> m_sptrA; // Dans la classe B
```

- Comme le constructeur de « weak_ptr » est sans argument, il faut enlever les constructeurs des classes « A » et « B ». Nous aurons ainsi :

```
sptrB->m_sptrA = sptrA; // 1 référence shared + 1 référence weak

sptrA->m_sptrB = sptrB; // 1 référence shared + 1 référence weak
```

- Quand le pointeur partagé est détruit, le compteur référence « ptr_shared » est égal à 0, celui de « ptr_weak » reste à 1.
- Or dès qu'une référence associée à « shared_ptr » est égale 0, elle met de facto le compteur de références associé à un « weak_ptr » à 0.
- Ainsi donc, la fuite de mémoire est colmatée.