Programmation C++ : Les conteneurs STL

La bibliothèque standard C++ **2**

Notion de conteneurs . . . 2
Notion de complexité . . . 3
Le tableau dynamique (`vector`) . . . 3
Notion d'itérateurs (`iterator`) . . . 5
La liste (`list`) . . . 6
La table associative (`map`) . . . 7
Une paire d'éléments (`pair`) . . . 9
Un ensemble d'éléments (`set`) . . . 11
La pile (`stack`) . . . 12
La file (`queue`) . . . 14
Table de hachage . . . 16
Autres (*boost*) . . . 20
Tri d'un conteneur (`sort`) . . . 20
Suppresion des éléments d'un conteneur (`clear`, `erase` et `remove`) . . . 26
Recherche d'éléments (`find`) . . . 30
Choix d'un conteneur . . . 31
Comparaison des complexités . . . 32
Exemple détaillé : utilisation d'un vector . . . 33

Liste des exemples **36**

Programmation C++

Les objectifs de ce cours sont de découvrir les conteneurs de la STL en C++.

La bibliothèque standard C++

Le C++ possède une **bibliothèque standard** (SL pour *Standard Library*) qui est composée, entre autre, d'une bibliothèque de flux, de la bibliothèque standard du C, de la gestion des exceptions, ..., et de la STL (*Standard Template Library* : **bibliothèque de modèles standard**). La STL implémente de nombreux types de données de manière efficace et générique.

En fait, STL est une appellation historique communément acceptée et comprise. Dans la norme, on ne parle que de SL.

Dans un programme C++, on privilégiera toujours l'utilisation de la STL par rapport à une implémentation manuelle : gain en efficacité, robustesse, facilité, lisibilité car standard.

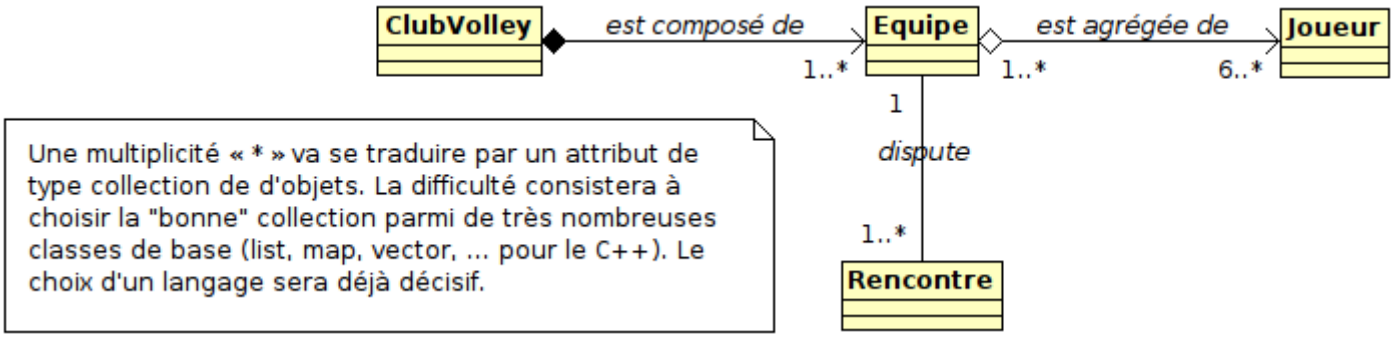
L'objectif de ce document n'est pas de faire un inventaire exhaustif des possibilités offertes par la STL, mais de donner quelques exemples courants d'utilisation. On pourra trouver un aperçu très détaillé des classes de la STL ici : www.sgi.com/tech/stl/ et www.cplusplus.com/reference/stl/.

Notion de conteneurs

La STL fournit un certain nombre de **conteneurs** pour gérer des **collections d'objets** : les **tableaux** (***vector***), les **listes** (*list*), les **ensembles** (*set*), les **piles** (*stack*), et beaucoup d'autres ...

Un **conteneur** (*container*) est un **objet qui contient d'autres objets**. Il fournit un moyen de gérer les objets contenus (au minimum ajout, suppression, parfois insertion, tri, recherche, ...) ainsi qu'un accès à ces objets qui dans le cas de la STL consiste très souvent en un **itérateur**.

Exemples : une liste d'articles à commander, une flotte de bateaux, les équipes d'un club, ...



Bien qu'il soit possible de créer des tableaux d'objets, ce ne sera pas forcément la bonne solution. On préfère recourir à des collections parmi lesquelles les plus utilisées sont :

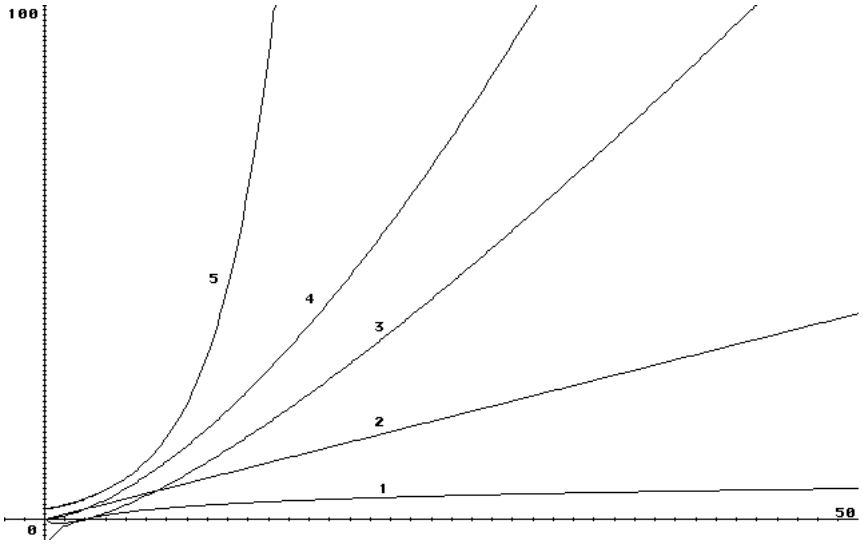
- Java : `ArrayList`, `HashMap`, ...
- C# : `ArrayList`, `SortedList`, `HashTable` ...
- C++ (STL) : `vector`, `list`, `map`, `set` ...
- Qt/C++ : `QVector`, `QList`, `QMap` ...

Notion de complexité

Il est particulièrement important de choisir une classe fournie par la STL **cohérente avec son besoin**. Certains conteneurs ont des algorithmes plus ou moins efficaces pour accéder à un élément, pour l'insérer, ... Pour quantifier l'efficacité, on s'intéresse à son contraire, la **complexité** (notation O, «grand O»).

Si vous possédez un tableau de *n* cases et, que pour insérer un nouvel élément, il vous faut décaler les *n* éléments déjà présents, la complexité de cette insertion sera alors O(n).

Il est au préalable nécessaire d'avoir quelques notions de **complexité**. Soit n la taille d'un conteneur : un algorithme est dit linéaire en O(n) si son temps de calcul est proportionnel à n. De la même façon, un algorithme peut être instantané (O(1)), logarithmique (O(log(n))), linéaire (O(n)), polynomial ($O(n^k)$), exponentiel (O(e(n))), etc... dans l'ordre croissant des proportions de temps de calcul. L'enjeu de cette partie consiste à présenter les avantages et les inconvénients de chacun des conteneurs.



1. complexité logarithmique en $O(ln(n))$
2. complexité linéaire en $O(n)$
3. complexité quasi-linéaire en $O(n*ln(n))$
4. complexité polynômiale en $O(n^k)$
5. complexité exponentielle en $O(a^n)$

Dans ce document, on s'intéressera principalement à la complexité pour l'accès (recherche) à une donnée stockée dans un conteneur et pour insérer une donnée.

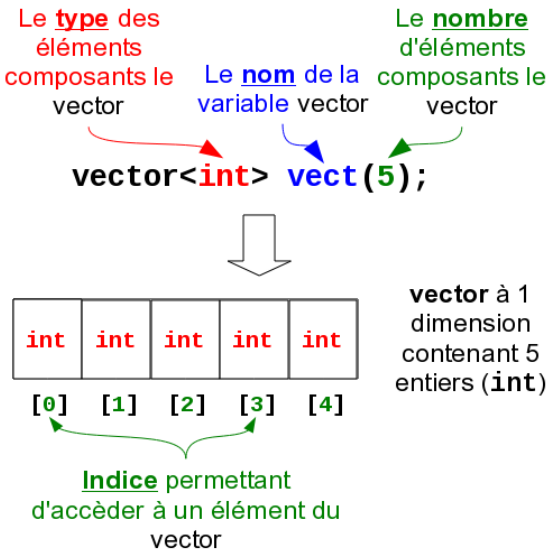
Le tableau dynamique (`vector`)

Un **vector** est un **conteneur séquentiel** qui encapsule les **tableaux de taille dynamique**. Les éléments sont stockés de façon contigüe, ce qui signifie que les éléments sont accessibles non seulement via les itérateurs, mais aussi à partir des pointeurs classiques sur un élément.

Lien : http://www.cplusplus.com/reference/vector/vector/

Il est particulièrement aisé d'**accéder directement** aux divers éléments par un **index**, et d'en ajouter ou en retirer à la fin.

A la manière des tableaux de type C, l'espace mémoire alloué pour un objet de type *vector* est toujours continu, ce qui permet des algorithmes rapides d'accès aux divers éléments.




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

using namespace std;

int main()
{
  vector<int> vect(5); // un vecteur de 5 entiers

  vect[0] = 1; // accès direct à l'indice 0 pour affecter la valeur 1
  vect[1] = 2; // accès direct à l'indice 1 pour affecter la valeur 2

  // augmente et diminue la taille du vector
  vect.push_back( 6 ); // ajoute l'entier 6 à la fin
  vect.push_back( 7 ); // ajoute l'entier 7 à la fin
  vect.push_back( 8 ); // ajoute l'entier 8 à la fin
  vect.pop_back(); // enleve le dernier élément et supprime l'entier 8

  cout << "Le vecteur vect contient " << vect.size() << " entiers : \n";

  // utilisation d'un indice pour parcourir le vecteur vect
  for(int i=0;i<vect.size();i++)
      cout << "vect[" << i << "] = " << vect[i] << '\n';
  cout << '\n';

  return 0;
}
```

Exemple d'utilisation d'un `vector`

On obtient à l'exécution :

```
Le vecteur vect contient 7 entiers :
vect[0] = 1
vect[1] = 2
vect[2] = 0
vect[3] = 0
vect[4] = 0
```

```
vect[5] = 6
vect[6] = 7
```

Une "case" n'est accessible par l'opérateur [] que si elle a été allouée au préalable (sinon une erreur de segmentation est déclenchée).

Il ne faut pas perdre de vue qu'une réallocation mémoire est coûteuse en terme de performances. Ainsi si la taille d'un `vector` est connue par avance, il faut autant que possible le créer directement à cette taille (voir méthodes `resize` et `reserve`).

Complexité

- Accès : O(1)
- Insertion : O(n) en début de vector (`pop_back`), O(1) en fin de `vector` (`push_back`). Dans les deux cas des réallocations peuvent survenir.

Il existe aussi un conteneur `deque` (*Double Ended QUEue*) qui s'utilise de la même façon que `vector` à deux différences près :

- `deque` est optimisé pour que les éléments soient ajoutés ou retirés à la fin (`push_back` et `pop_back`) ou **au début** (`push_front` et `pop_front`)
- `deque` ne stockent pas (forcément) les éléments de façon contigüe, il est donc plus efficace en terme de réallocation mémoire importante

La STL possède des adaptateurs de conteneurs (réduisant les possibilités d'un `vector` à quelques fonctionnalités) : `stack` (pile), `queue` (file) et `priority_queue` (file à priorités).

Notion d'itérateurs (`iterator`)

Les **itérateurs** (*iterator*) sont une généralisation des **pointeurs** : ce sont des **objets qui pointent sur d'autres objets**.

Lien : http://www.cplusplus.com/reference/iterator/

Comme son nom l'indique, les itérateurs sont utilisés pour **parcourir une série d'objets** de telle façon que si on incrémente l'itérateur, il désignera l'objet suivant de la série.

```cpp
#include <iostream>
#include <vector>
using namespace std;

int main()
{
  vector<int> v2(4, 100); // un vecteur de 4 entiers initialisés avec la valeur 100
  cout << "Le vecteur v2 contient " << v2.size() << " entiers : ";
  // utilisation d'un itérateur pour parcourir le vecteur v2
  for (vector<int>::iterator it = v2.begin(); it != v2.end(); ++it)
    cout << ' ' << *it;
  cout << '\n';
  return 0;
}
```

Exemple d'utilisation d'un `iterator`

On obtient à l'exécution :

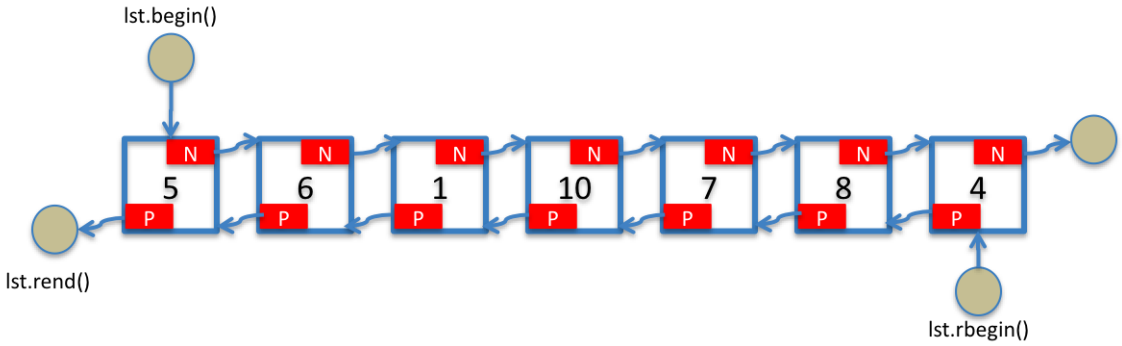
```
Le vecteur v2 contient 4 entiers : 100 100 100 100
```

Il existe aussi un itérateur inversé : `reverse_iterator` (http://www.cplusplus.com/reference/iterator/reverse_iterator/).

La liste (`list`)

La classe `list` fournit une structure générique de **listes doublement chaînées** (c'est-à-dire que l'on peut parcourir dans les deux sens) pouvant éventuellement contenir des doublons.

Lien : http://www.cplusplus.com/reference/list/list/




```cpp
#include <iostream>
#include <list>

using namespace std;

int main()
{
  list<int> lst; // une liste vide

  lst.push_back( 5 );
  lst.push_back( 6 );
  lst.push_back( 1 );
  lst.push_back( 10 );
  lst.push_back( 7 );
  lst.push_back( 8 );
  lst.push_back( 4 );
  lst.push_back( 5 );
  lst.pop_back(); // enleve le dernier élément et supprime l'entier 5

  cout << "La liste lst contient " << lst.size() << " entiers : \n";

  // utilisation d'un itérateur pour parcourir la liste lst
  for (list<int>::iterator it = lst.begin(); it != lst.end(); ++it)
    cout << ' ' << *it;
  cout << '\n';

  // afficher le premier élément
  cout << "Premier element : " << lst.front() << '\n';
```

```
  // afficher le dernier élément
  cout << "Dernier element : " << lst.back() << '\n';

  // parcours avec un itérateur en inverse
  for ( list<int>::reverse_iterator rit = lst.rbegin(); rit != lst.rend(); ++rit )
  {
      cout << ' ' << *rit;
  }
  cout << '\n';

  return 0;
}
```

Exemple d'utilisation d'une `list`

On obtient à l'exécution :

```
La liste lst contient 7 entiers :
 5 6 1 10 7 8 4
Premier element : 5
Dernier element : 4
 4 8 7 10 1 6 5
```

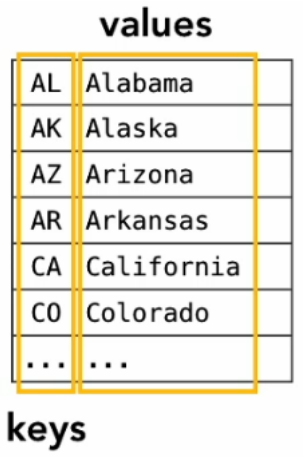
Complexité

- Insertion (en début ou fin de liste) : O(1)
- Recherche : O(n) en général, O(1) pour le premier et le dernier maillon

La table associative (map)

Une **table associative** map permet d'**associer une clé à une donnée**.

Lien : http://www.cplusplus.com/reference/map/map/



Le tableau associatif (aussi appelé dictionnaire ou table d'association) peut être vu comme une généralisation du tableau : alors que le tableau traditionnel associe des entiers consécutifs à des valeurs d'un certain type, le tableau associatif associe des valeurs d'un type arbitraire à des valeurs d'un autre type.

La map prend donc au moins deux paramètres :
- le **type de la clé** (dans l'exemple ci-dessous, une chaîne de caractères string)
- le **type de la donnée** (dans l'exemple ci-dessous, un entier non signé unsigned int)

```
#include <iostream>
#include <iomanip>
#include <map>
#include <string>

using namespace std;

int main()
{
  map<string,unsigned int> nbJoursMois;

  nbJoursMois["janvier"] = 31;
  nbJoursMois["février"] = 28;
  nbJoursMois["mars"] = 31;
  nbJoursMois["avril"] = 30;
  //...

  cout << "La map contient " << nbJoursMois.size() << " elements : \n";
  for (map<string,unsigned int>::iterator it=nbJoursMois.begin(); it!=nbJoursMois.end(); ++
      it)
  {
    cout << it->first << " -> \t" << it->second << endl;
  }

  cout << "Nombre de jours du mois de janvier : " << nbJoursMois.find("janvier")->second <<
      '\n';

  // affichage du mois de janvier
  cout << "Janvier : \n" ;
  for (int i=1; i <= nbJoursMois["janvier"]; i++)
  {
      cout << setw(3) << i;
      if(i%7 == 0)
         cout << endl;
  }
  cout << endl;

  return 0;
}
```

Exemple d'utilisation d'une map

On obtient à l'exécution :

```
La map contient 4 elements :
avril ->        30
février ->      28
janvier ->      31
mars ->         31
Nombre de jours du mois de janvier : 31
Janvier :
```

```
 1  2  3  4  5  6  7
 8  9 10 11 12 13 14
15 16 17 18 19 20 21
22 23 24 25 26 27 28
29 30 31
```

Complexité

- **Insertion :** O(log(n))
- **Recherche :** O(log(n))

Le fait d'accéder à une clé via l'opérateur [] insère cette clé dans la map (avec le constructeur par défaut pour la donnée). Ainsi l'opérateur [] n'est pas adapté pour vérifier si une clé est présente dans la map, il faut utiliser la méthode find.

Une paire d'éléments (pair)

Une **paire** est une **structure contenant deux éléments éventuellement de types différents**.

Lien : http://www.cplusplus.com/reference/utility/pair/

Certains algorithmes de la STL (find par exemple) retournent des paires (position de l'élément trouvé et un booléen indiquant s'il a été trouvé).

En pratique, il faut voir les classes conteneurs de la STL comme des "legos" (briques logicielles) pouvant être imbriqués les uns dans les autres. Ainsi, on pourra tout à fait manipuler un vector de pair, etc ...

```cpp
#include <iostream>
#include <utility>
#include <vector>
#include <list>
#include <map>
#include <set>

using namespace std;

int main()
{
  pair<char,int> c1 = make_pair('B', 2); // coordonnées type "bataille navale"
  pair<char,int> c2 = make_pair('J', 1);

  cout << "Un coup en " << c1.first << '.' << c1.second << endl;

  pair<int,int> p; // position de type "morpion"

  p.first = 1;
  p.second = 2;

  cout << "Un coup en " << p.first << ',' << p.second << endl;
  cout << endl;
```

```
  vector<pair<char,int> > tableauCoups(2); // ou par exemple : list<pair<char,int> >
     listeCoups;

  tableauCoups[0] = c1;
  tableauCoups[1] = c2;
  cout << "Le vector contient " << tableauCoups.size() << " coups : \n";
  for (vector<pair<char,int> >::iterator it=tableauCoups.begin(); it!=tableauCoups.end(); ++
     it)
  {
    cout << (*it).first << "." << (*it).second << endl;
  }
  cout << endl;

  map<pair<char,int>,bool> mapCoups;

  mapCoups[c1] = true; // ce coup a fait mouche
  mapCoups[c2] = false; // ce coup a fait plouf

  cout << "La map contient " << mapCoups.size() << " coups : \n";
  for (map<pair<char,int>,bool>::iterator it=mapCoups.begin(); it!=mapCoups.end(); ++it)
  {
    cout << it->first.first << "." << it->first.second << " -> \t" << it->second << endl;
  }
  cout << endl;

  set<pair<char,int> > ensembleCoups;

  ensembleCoups.insert(c1);
  ensembleCoups.insert(c2);

  cout << "L'ensemble set contient " << ensembleCoups.size() << " coups : \n";
  for (set<pair<char,int> >::iterator it=ensembleCoups.begin(); it!=ensembleCoups.end(); ++
     it)
  {
    cout << (*it).first << "." << (*it).second << endl;
  }
  cout << endl;

  return 0;
}
```

Exemple d'utilisation d'un `pair`

On obtient à l'exécution :

```
Un coup en B.2
Un coup en 1,2

Le vector contient 2 coups :
B.2
J.1

La map contient 2 coups :
B.2 -> 1
```

```
J.1 -> 0

L'ensemble set contient 2 coups :
B.2
J.1
```

Complexité

- **Insertion :** O(1)
- **Recherche :** O(1)

Un ensemble d'éléments (set)

Dans l'exemple ci-dessus, on utilise un autre conteneur qui se nomme **set**. La classe **set** est un conteneur qui stocke des éléments uniques suivants un ordre spécifique (c'est-à-dire un ensemble ordonné et sans doublons d'éléments). La complexité est O(log(n)) pour la recherche et l'insertion.

Lien : http://www.cplusplus.com/reference/set/set/

Les ensembles set sont généralement mis en oeuvre dans les arbres binaires de recherche.

```cpp
#include <iostream>
#include <set>
using namespace std;

int main()
{
  int desEntiers1[] = {75, 23, 65, 42, 13, 100}; // non ordonné
  int desEntiers2[] = {75, 23, 75, 23, 13}; // non ordonné avec des doublons
  set<int> ensemble1 (desEntiers1, desEntiers1+6); // the range (first,last)
  set<int> ensemble2 (desEntiers2, desEntiers2+5); // the range (first,last)

  cout << "L'ensemble set 1 contient :";
  for (set<int>::iterator it=ensemble1.begin(); it!=ensemble1.end(); ++it)
  {
    cout << ' ' << *it;
  }

  cout << endl;

  cout << "L'ensemble set 2 contient :";
  for (set<int>::iterator it=ensemble2.begin(); it!=ensemble2.end(); ++it)
  {
    cout << ' ' << *it;
  }

  cout << endl;

  return 0;
}
```

Exemple d'utilisation d'un conteneur **set**

À l'exécution, on obtient deux ensembles ordonnés sans doublons :

```
L'ensemble set 1 contient : 13 23 42 65 75 100
L'ensemble set 2 contient : 13 23 75
```

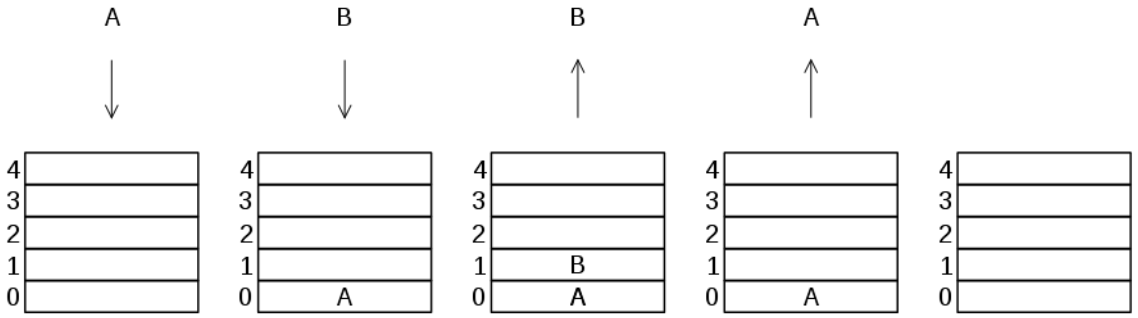
Complexité

- **Insertion :** O(log(n))
- **Recherche :** O(log(n))

La pile (stack)

Une **pile (« *stack* » en anglais)** est une **structure de données basée sur le principe « Dernier arrivé, premier sorti », ou LIFO (*Last In, First Out*)**, ce qui veut dire que les derniers éléments ajoutés à la pile seront les premiers à être récupérés.

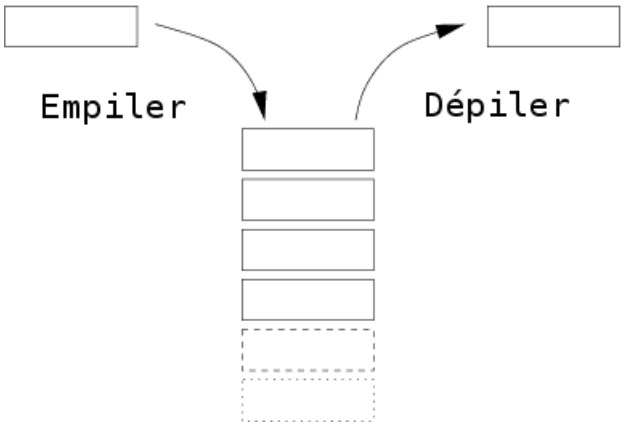
Lien : http://www.cplusplus.com/reference/stack/stack/



Le fonctionnement est celui d'une pile d'assiettes : on ajoute des assiettes sur la pile, et on les récupère dans l'ordre inverse, en commençant par la dernière ajoutée.

Une **pile** est utilisée en général pour **gérer un historique de données** (pages webs visitées, ...) ou **d'actions** (les fonctions « Annuler » de beaucoup de logiciels par exemple). La pile est utilisée aussi pour tous les paramètres d'appels et les variables locales des fonctions dans les langages compilés.

Voici quelques fonctions communément utilisées pour manipuler des **piles** :
- « Empiler » : ajoute ou dépose un élément sur la pile
- « Dépiler » : enlève un élément de la pile et le renvoie
- « La pile est-elle vide ? » : renvoie « vrai » si la pile est vide, « faux » sinon
- « La pile est-elle pleine ? » : renvoie « vrai » si la pile est pleine, « faux » sinon
- « Nombre d'éléments dans la pile » : renvoie le nombre d'éléments présents dans la pile
- « Taille de la pile » : renvoie le nombre maximum d'éléments pouvant être déposés dans la pile
- « Quel est l'élément de tête ? » : renvoie l'élément de tête (le sommet) sans le dépiler



Exemple avec le type `stack` de la STL :

```
#include <iostream>
#include <stack>
using namespace std;

/* Les opérations de base :
    void pile.push(valeur); // Empiler
    T    pile.top();        // Retourne la valeur du haut de la pile
    void pile.pop();        // Dépiler
    bool pile.empty();      // Retourne true si la pile est vide sinon false
    void pile.clear();      // Vider la pile

   Lien : http://www.cplusplus.com/reference/stack/stack/ */

int main()
{
    stack<int> pile;
    pile.push(4);
    pile.push(2);
    pile.push(1);

    cout << "Taille de la pile : " << pile.size() << endl;
    while (!pile.empty())
    {
        cout << pile.top() << endl;
        pile.pop();
    }
    cout << "Taille de la pile : " << pile.size() << endl;

    return 0;
}
```

Exemple d'utilisation d'une `stack`

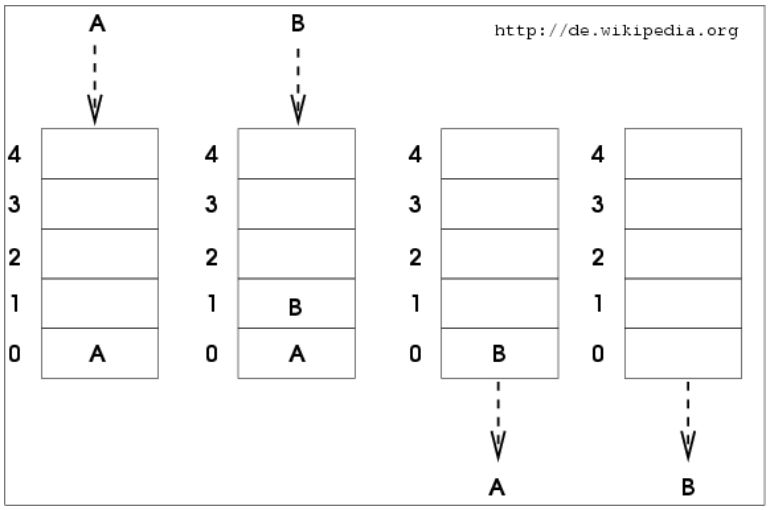
Complexité

Opérations `push`, `pop`, `top` et `size` : O(1)

La file (queue)

Une **file (ou file d'attente, « *queue* » en anglais)** est une **structure de données basée sur le principe « Premier arrivé, premier sorti », ou FIFO (*First In, First Out*)**, ce qui veut dire que les premiers éléments ajoutés à la file seront les premiers à être récupérés.

Lien : http://www.cplusplus.com/reference/queue/queue/



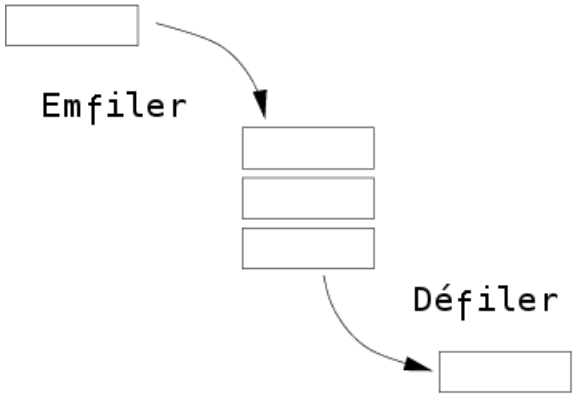
Le fonctionnement est celui d'une file d'attente : les premières personnes arrivées sont les premières personnes à sortir de la file.

Une liste chaînée dont on n'utilise que les opérations `ajouterQueue` et `retirerTête` constitue une queue. Si la queue se base sur un tableau, la structure enregistre deux indices, l'un correspondant au dernier arrivé, l'autre au prochain à sortir.

Les queues servent à organiser le traitement séquentiel des blocs de données d'origines diverses. C'est une technique fiable pour être sûr d'effectuer les opérations dans un ordre logique.

Une **file** est utilisée en général pour mémoriser temporairement des transactions qui doivent attendre pour être traitées (notions de serveur et de *spool*) et pour créer toutes sortes de mémoires tampons (« *buffers* »).

Voici quelques fonctions communément utilisées pour manipuler des **piles** :
- « Enfiler » : ajoute un élément dans la file (*enqueue*)
- « Défiler » : renvoie le prochain élément de la file, et le retire de la file (*dequeue*)
- « La file est-elle vide ? » : renvoie « vrai » si la file est vide, « faux » sinon
- « Nombre d'éléments dans la file » : renvoie le nombre d'éléments dans la file



Exemple avec le type queue de la STL :

```
#include <iostream>
#include <queue>

using namespace std;

/* Les opérations de base :
    void file.push(valeur); // Empiler
    T    file.front();    // Retourne la valeur la plus "ancienne"
    T    file.back();     // Retourne la valeur la moins "ancienne"
    void file.pop();      // Dépiler la valeur la plus "ancienne"
    bool file.empty();    // Retourne true si le tas est vide sinon false
    void file.size();     // Retourne la taille du tas

   Lien : http://www.cplusplus.com/reference/queue/queue/ */

int main()
{
    queue<int> file;

    file.push(1);
    file.push(4);
    file.push(2);

    cout << "Taille de la file : " << file.size() << endl;
    while (!file.empty())
    {
        cout << file.front() << endl;
        file.pop();
    }
    cout << "Taille de la file : " << file.size() << endl;
    return 0;
}
```

Exemple d'utilisation d'une queue

Il existe aussi des files à priorité qui permettent de récupérer l'élement de plus grande valeur. Elles permettent d'implémenter efficacement des planificateurs de tâches, où un accès rapide aux tâches d'importance maximale est souhaité. Les files à priorité sont utilisés dans certains algorithmes de tri et de recherche.

Exemple avec le type priority_queue de la STL :

```
#include <iostream>
#include <queue>

using namespace std;

/* Les opérations de base :
    void tas.push(valeur); // Empiler
    T    tas.top();      // Retourne la plus grande valeur selon l'opérateur <
    void tas.pop();      // Dépiler la plus grande valeur
    bool tas.empty();    // Retourne true si le tas est vide sinon false
    void tas.size();     // Retourne la taille du tas
```

```
   Lien : http://www.cplusplus.com/reference/queue/priority_queue/ */

int main()
{
    priority_queue<int> tas;

    tas.push(1);
    tas.push(4);
    tas.push(2);

    cout << "Taille du tas : " << tas.size() << endl;
    while (!tas.empty())
    {
        cout << tas.top() << endl;
        tas.pop();
    }

    return 0;
}
```

Exemple d'utilisation d'une `priority_queue`

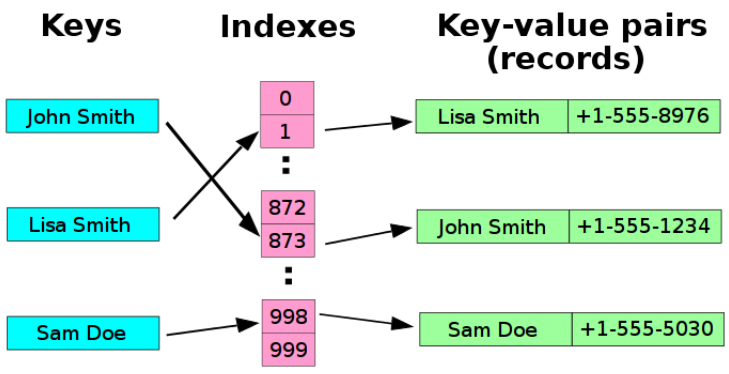
Complexité

Opérations `push_back`, `pop_front`, `front` et `size` : O(1)

Table de hachage

Une table de hachage (*hash table*) est une structure de données qui permet une association clé-élément. Il s'agit d'un tableau ne comportant pas d'ordre (contrairement à un tableau ordinaire qui est indexé par des entiers). On accède à chaque élément de la table par sa clé. L'accès s'effectue par une **fonction de hachage** qui transforme une clé en une valeur de hachage (un nombre) indexant les éléments de la table.

- On accède à chaque élément de la table via sa clé.
- L'accès à un élément se fait en transformant la clé en une valeur de hachage par l'intermédiaire d'une fonction de hachage.
- Le hachage est un nombre qui permet de localiser les éléments dans le tableau, c'est l'index de l élément dans le tableau.



Le fait de créer une valeur de hachage à partir d'une clé peut engendrer un problème de « **collision** », c'est-à-dire que deux clés différentes, voire davantage, pourront se retrouver associées à la même valeur de

hachage et donc au même élément de la table. Pour diminuer les risques de collisions, il faut donc premièrement choisir avec soin sa fonction de hachage. Ensuite, un mécanisme de résolution des collisions sera à implémenter.

Il existe deux types de tables de hachage dans la STL :

- `hash_set<K>` : table de hachage simple, stocke seulement des clés de type K.
- `hash_map<K,T>` : table de hachage double, stocke des clés de type K associées à des valeurs de type T. À une clé donnée ne peut être stockée qu'une seule valeur.

Liens : https://www.sgi.com/tech/stl/hash_set.html et https://www.sgi.com/tech/stl/hash_map.html

`hash_set` et `hash_map` font partie de la STL mais ne sont pas intégrés à la bibliothèque standard C++. Les compilateurs GNU C++ et Visual C++ de Microsoft les ont quand même implémentés.

La nouvelle norme C++ 11 (`-std=c++11`) propose des conteneurs similaires : `unordered_set` et `unordered_map`.

Liens : http://www.cplusplus.com/reference/unordered_set/unordered_set/ et http://www.cplusplus.com/reference/unordered_map/unordered_map/

```cpp
#include <iostream>
#include <string>
#include <unordered_map>

using namespace std;

// g++ unordered-map.cpp -std=c++0x

// g++ >= 4.8
// g++ unordered-map.cpp -std=c++11

int main()
{
    unordered_map<string, string> hashtable;

    //hashtable.emplace("www.wikipedia.fr", "78.109.84.114");
    //cout << "Adresse IP : " << hashtable["www.wikipedia.fr"] << endl;

    hashtable.insert(make_pair("www.cplusplus.com", "167.114.170.15"));
    hashtable.insert(make_pair("www.google.fr", "216.58.204.67"));
    cout << "Adresse IP de www.google.fr : " << hashtable["www.google.fr"] << endl << endl;

    cout << "La table : " << endl;
    for (auto itr = hashtable.begin(); itr != hashtable.end(); itr++)
    {
        cout << (*itr).first << " -> " << (*itr).second << endl;
    }

    return 0;
}
```

Exemple d'utilisation d'un `unordered_map`

On peut créer sa propre fonction de hachage avec un foncteur [1] :

```
#include <iostream>
#include <string>
#include <unordered_map>

using namespace std;

// Foncteur de hachage
class Hachage
{
    public:
        size_t operator()(const string &s) const
        {
            cout << "hash : " << hash<string>()(s) << endl;
            return hash<string>()(s);
        }
};

int main()
{
    unordered_map<string, string, Hachage> hashtable;

    hashtable.insert(make_pair("www.wikipedia.fr", "78.109.84.114"));
    hashtable.insert(make_pair("www.cplusplus.com", "167.114.170.15"));
    hashtable.insert(make_pair("www.google.fr", "216.58.204.67"));

    cout << "La table : " << endl;
    for (auto itr = hashtable.begin(); itr != hashtable.end(); itr++)
    {
        cout << (*itr).first << " -> " << (*itr).second << endl;
    }
    cout << endl;

    cout << "Adresse IP de www.google.fr : " << hashtable["www.google.fr"] << endl;

    return 0;
}
```

On peut utiliser `unordered_map` avec ses propres classes à condition de définir l'opérateur `==` :

```
#include <iostream>
#include <string>
#include <unordered_map>

using namespace std;

class Fabricant
{
    private:
        string nom;

    public:
```

1. Un foncteur (*Function Object*) est un objet qui se comporte comme une fonction (en surchargeant l'opérateur `()`).

```
        Fabricant(string nom)
        {
            this->nom = nom;
        }

        string getNom() const
        {
            return nom;
        }

        bool operator==(const Fabricant &f) const
        {
            return nom == f.nom;
        }
};

class Modele
{
    private:
        string nom;
        int annee;

    public:
        Modele(string nom, int annee)
        {
            this->nom = nom;
            this->annee = annee;
        }

        string getNom() const
        {
            return nom;
        }

        int getAnnee() const
        {
            return annee;
        }

        bool operator==(const Modele &m) const
        {
            return (nom == m.nom && annee == m.annee);
        }
};

class Hachage
{
    public:
        size_t operator()(const Modele &m) const
        {
            return hash<string>()(m.getNom()) ^ hash<int>()(m.getAnnee());
        }
};
```

```
int main()
{
    unordered_map<Modele, Fabricant, Hachage> catalogue;

    Modele zoe("Zoe", 2012);
    Modele megane3("Megane III", 2008);
    Modele clio3("Clio III", 2005);
    Modele bipper("Bipper", 2007);
    Fabricant renault("Renault");
    Fabricant peugeot("Peugeot");

    catalogue.insert(make_pair(zoe, renault));
    catalogue.insert(make_pair(megane3, renault));
    catalogue.insert(make_pair(clio3, renault));
    catalogue.insert(make_pair(bipper, peugeot));

    for (auto &itr : catalogue)
    {
        cout << itr.second.getNom() << " " << itr.first.getNom() << " " << itr.first.
           getAnnee() << endl;
    }

    return 0;
}
```

Autres (*boost*)

Boost est une collection de bibliothèques logicielles utilisées en programmation C++. En fait, Boost se veut un laboratoire d'essais destiné à expérimenter de nouvelles bibliothèques pour le C++. Plusieurs de ses bibliothèques ont déjà été intégrées à la bibliothèque standard C++.

Boost propose notamment d'autres conteneurs : tampon circulaire (`boost::circular_buffer`), tableaux de bits dynamiques (`boost::dynamic_bitset`), les matrices et les tableaux à dimensions multiples (`boost::multi_array`) ...

Liens : http://www.boost.org/ et https://cpp.developpez.com/faq/cpp/?page=Boost

Tri d'un conteneur (`sort`)

La STL fournit une fonction générique (très performante) `sort()` pour effectuer un tri de comparaison. Il faut inclure le fichier d'en-tête `<algorithm>`.

Lien : http://www.cplusplus.com/reference/algorithm/sort/

La fonction `sort()` est capable de trier des tableaux :

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

using namespace std;

int main()
```

```
{
  int tableau[] = { 23, 5, -10, 0, 0, 321, 1, 2, 99, 30 };
  size_t taille = sizeof(tableau) / sizeof(tableau[0]);

  cout << "Avant : " << endl;
  for (size_t i = 0; i < taille; ++i)
  {
     cout << tableau[i] << ' ';
  }
  cout << endl;

  // Tri :
  sort(tableau, tableau + taille);

  cout << "Après : " << endl;
  for (size_t i = 0; i < taille; ++i)
  {
     cout << tableau[i] << ' ';
  }
  cout << endl;

  return 0;
}
```

Tri d'un tableau

La fonction `sort()` peut évidemment trier un `vector` : elle prend alors en premier argument un itérateur vers le premier élément du vecteur (`vec.begin()`) et en second argument un itérateur vers le dernier élément (`vec.end()`) :

Pour trier un conteneur de type `list`, il faut utiliser la fonction membre `list::sort()`.

```
#include <algorithm>
#include <iostream>
#include <vector>
using namespace std;

int main()
{
  vector<int> vec { 23, 5, -10, 0, 0, 321, 1, 2, 99, 30 }; // -std=c++0x ou -std=c++11
  cout << "Avant : " << endl;
  for (int i = 0; i < vec.size(); ++i)
  {
     cout << vec[i] << ' ';
  }
  cout << endl;

  // Tri :
  sort(vec.begin(), vec.end());

  cout << "Après : " << endl;
  for (int i = 0; i < vec.size(); ++i)
  {
```

```
    cout << vec[i] << ' ';
  }
  cout << endl;
  return 0;
}
```

Tri d'un vector

Bien entendu, cette fonction fonctionne également très bien avec des `float`, des `double`, des `string` ou tout autre type déjà défini.

Certains conteneurs sont déjà triés. C'est le cas de `map` et de `set`. Pour ce type de conteneurs, la fonction `sort()` n'a donc pas d'utilité.

La fonction `sort()` peut également trier des objets, avec l'obligation d'avoir défini l'opérateur `<` :

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

using namespace std;

class Objet
{
    private:
        int valeur;

    public:
        Objet(int v=0)
        {
            valeur = v;
        }

        int getValeur() const
        {
            return valeur;
        }

        void setValeur(int v)
        {
            valeur = v;
        }

        // Définit l'opérateur < pour le tri :
        bool operator < (const Objet &o) const
        {
            return valeur < o.valeur;
        }
};

int main()
{
    const int NB_OBJETS = 10;
```

```
    Objet objets[NB_OBJETS];
    int nbObjets = 0;

    cout << "Un tableau d'objets :" << endl;
    for(int i=0;i<NB_OBJETS;i++)
    {
        objets[i].setValeur(NB_OBJETS-i);
        cout << objets[i].getValeur() << endl;
    }

    // choisir le nombre d'objets à trier :
    nbObjets = NB_OBJETS;
    sort(objets, objets + nbObjets);

    cout << "Un tableau d'objets triés :" << endl;
    for(int i=0;i<NB_OBJETS;i++)
    {
        cout << objets[i].getValeur() << endl;
    }
}
```

Tri d'objets

La fonction `sort()` est aussi une fonction surchargée dont la deuxième version admet un **prédicat** [2] comme troisième argument. Dans ce cas, les valeurs seront comparées via le prédicat (une fonction de tri). On utilisera un **foncteur** [3] pour le prédicat. Ceci permettra de définir ces propres critères de tri. On pourra par exemple trier par ordre décroissant (avec l'opérateur `>`).

La liste des prédicats utilisables pour les algorithmes de tris :

- `std::equal_to<T>` : utilise l'opérateur `==`
- `std::greater<T>` : utilise `>`
- `std::greater_equal<T>` : utilise `>=`
- `std::less<T>` : utilise `<`
- `std::less_equal<T>` : utilise `<=`

Lien : http://www.cplusplus.com/reference/functional/

La STL fournit les foncteurs pour ces prédicats, par exemple `greater` et `less` :

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

using namespace std;

int main()
{
  vector<int> vec { 23, 5, -10, 0, 0, 321, 1, 2, 99, 30 }; // -std=c++0x ou -std=c++11

  cout << "Avant : " << endl;
  for (int i = 0; i < vec.size(); ++i)
  {
```

2. Un prédicat est une fonction renvoyant un booléen.
3. Un foncteur (*Function Object*) est un objet qui se comporte comme une fonction (en surchargeant l'opérateur `()`).

```
    cout << vec[i] << ' ';
  }
  cout << endl;

  // Tri décroissant :
  sort(vec.begin(), vec.end(), greater<int>());

  cout << "Après : " << endl;
  for (int i = 0; i < vec.size(); ++i)
  {
    cout << vec[i] << ' ';
  }
  cout << endl;

  return 0;
}
```

Tri décroissant d'un vector

Il est également possible d'écrire ses propres foncteurs :

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

using namespace std;

class Objet
{
   private:
      int valeur;

   public:
      Objet(int v=0)
      {
         valeur = v;
      }

      int getValeur() const
      {
         return valeur;
      }

      void setValeur(int v)
      {
         valeur = v;
      }

      // prédicat pour less
      bool operator < (const Objet &o) const
      {
         return valeur < o.valeur;
      }

      // prédicat pour greater
      bool operator > (const Objet &o) const
```

```
        {
            return valeur > o.valeur;
        }
};

// Foncteur pour le tri ascendant
class TriAscendant
{
    public:
        bool operator() (const Objet &a, const Objet &b) const
        {
            return a.getValeur() < b.getValeur();
        }
};

// Foncteur pour le tri descendant
class TriDescendant
{
    public:
        bool operator() (const Objet &a, const Objet &b) const
        {
            return a.getValeur() > b.getValeur();
        }
};

int main()
{
    const int NB_OBJETS = 10;
    Objet objets[NB_OBJETS];
    int nbObjets = 0;

    cout << "Un tableau d'objets : ";
    for(int i=0;i<NB_OBJETS;i++)
    {
        objets[i].setValeur(i+1);
        cout << objets[i].getValeur() << ' ';
    }
    cout << endl;

    // choisir le nombre d'objets à trier :
    nbObjets = NB_OBJETS;

    // Tri descendant :
    sort(objets, objets + nbObjets, TriDescendant());
    //sort(objets, objets + nbObjets, greater<Objet>());

    cout << "Un tableau d'objets triés (descendant) : ";
    for(int i=0;i<NB_OBJETS;i++)
    {
        cout << objets[i].getValeur() << ' ';
    }
    cout << endl;
```

```
    // Tri ascendant :
    //sort(objets, objets + nbObjets); // par défaut
    sort(objets, objets + nbObjets, TriAscendant());
    //sort(objets, objets + nbObjets, less<Objet>());

    cout << "Un tableau d'objets triés (ascendant) : ";
    for(int i=0;i<NB_OBJETS;i++)
    {
        cout << objets[i].getValeur() << ' ';
    }
    cout << endl;

    return 0;
}
```

Différents tris d'objets

Pour les conteneurs « triés » (comme map et set), il est possible de modifier le type de tri (en utilisant un foncteur) que l'on précisera à l'instanciation. Exemples : http://www.cplusplus.com/reference/map/map/map/ et http://www.cplusplus.com/reference/set/set/set/.

La STL fournit d'autres fonctions de tri :

- `stable_sort()` : très similaire à `sort()` en plus lent, mais permet de préserver l'ordre des éléments identiques
- `partial_sort()` : tri des valeurs en choisissant une rangée de valeurs
- `partial_sort_copy()` : identique à `partial_sort()`, mais les éléments sont copiés dans un nouveau tableau
- `nth_element()` : tri des données de sorte que l'élément à une certaine position soit mis à sa bonne position dans un tableau entièrement trié, et de façon à ce que les éléments précédents soient inférieurs, et les éléments suivants soient supérieurs à cette donnée (sans pour autant trier les valeurs précédentes et supérieures)

Lien : http://bakura.developpez.com/tutoriel/cpp/tri/

Suppresion des éléments d'un conteneur (`clear`, `erase` et `remove`)

On distingue deux situations qui dépendent de la nature de ce qui est stocké dans le conteneur :

- s'il s'agit d'**un objet**, il n'est pas utile de le détruire, il le sera lorsqu'il est retiré du conteneur, ou lorsque le conteneur est détruit.
- s'il s'agit d'**un pointeur sur un objet**, il faut le détruire car un pointeur n'est pas un objet. Si cette destruction n'est pas faite, le programme présentera une fuite de mémoire.

Pour les conteneurs `vector`, on utilisera les méthodes :

- `clear()` pour supprimer tous les éléments du conteneur
- `erase()` pour supprimer un élément unique ou une gamme d'éléments du vecteur à partir de leur position

La mémoire allouée ne sera pas libérée ce qui laissera la capacité du `vector` inchangée.

```cpp
#include <vector>
#include <iostream>
#include <algorithm>
using namespace std;

int main( )
{
    std::vector<int> vec{0, 1, 2, 3, 4, 5, 6, 7, 8, 9};

    cout << "vector [" << vec.size() << "," << vec.capacity() << "] : ";
    for (int i = 0; i < vec.size(); ++i)
    {
        cout << vec[i] << ' ';
    }
    cout << endl;
    // vector [10,10] : 0 1 2 3 4 5 6 7 8 9

    // supprime un élément unique
    vec.erase(vec.begin());

    cout << "vector [" << vec.size() << "," << vec.capacity() << "] : ";
    for (int i = 0; i < vec.size(); ++i)
    {
        cout << vec[i] << ' ';
    }
    cout << endl;
    // vector [9,10] : 1 2 3 4 5 6 7 8 9

    // supprime une gamme d'éléments
    vec.erase(vec.begin()+2, vec.begin()+5);

    cout << "vector [" << vec.size() << "," << vec.capacity() << "] : ";
    for (int i = 0; i < vec.size(); ++i)
    {
        cout << vec[i] << ' ';
    }
    cout << endl;
    // vector [6,10] : 1 2 6 7 8 9

    // supprime tous les éléments
    vec.clear();
    //vec.erase(vec.begin(), vec.end());

    cout << "vector [" << vec.size() << "," << vec.capacity() << "] : ";
    for (int i = 0; i < vec.size(); ++i)
    {
        cout << vec[i] << ' ';
    }
    cout << endl;
    // vector [0,10] :
    return 0;
}
```

Suppression d'éléments dans un `vector`

Pour le conteneur list, il faut utiliser la fonction advance() pour déplacer les itérateurs à utiliser ensuite avec erase() :

```
std::list<int> l{0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
list<int>::iterator first, last;

// supprime une gamme d'éléments
first = l.begin();
advance(first, 2);
last = l.begin();
advance(last, 5);

l.erase(first, last);

// list [7] : 0 1 5 6 7 8 9
```

Suppression d'éléments dans une `list`

Pour supprimer un élément à partir de sa valeur, il faut utiliser conjointement la fonction `remove()` et la méthode `erase()` pour le conteneur `vector` :

```
#include <vector>
#include <iostream>
#include <algorithm>
using namespace std;

int main( )
{
    std::vector<int> vec{0, 1, 2, 3, 4, 5, 6, 7, 8, 9};

    cout << "vector [" << vec.size() << "," << vec.capacity() << "] : ";
    for (int i = 0; i < vec.size(); ++i)
    {
        cout << vec[i] << ' ';
    }
    cout << endl; // vector [10,10] : 0 1 2 3 4 5 6 7 8 9

    // retire la valeur 5 (mais pas l'élément)
    remove(vec.begin(), vec.end(), 5);

    cout << "vector [" << vec.size() << "," << vec.capacity() << "] : ";
    for (vector<int>::iterator it = vec.begin(); it != vec.end(); ++it)
    {
        cout << *it << ' ';
    }
    cout << endl; // vector [10,10] : 0 1 2 3 4 6 7 8 9 9

    // retire l'élément de valeur 9
    vec.erase(remove(vec.begin(), vec.end(), 9), vec.end());

    cout << "vector [" << vec.size() << "," << vec.capacity() << "] : ";
    for (vector<int>::iterator it = vec.begin(); it != vec.end(); ++it)
    {
        cout << *it << ' ';
```

```
    }
    cout << endl; // vector [8,10] : 0 1 2 3 4 6 7 8

    return 0;
}
```

Pour le conteneur `list`, il suffit d'utiliser la méthode membre `remove()`.

Pour les `[multi]set` et les `[multi]map`, on utilisera la méthode membre `erase()` (qui a exactement le même rôle que `list::remove()`), par contre le paramètre est la **clé** de l'élément à effacer et non sa valeur.

Pour détruire les pointeurs d'un conteneur, on peut utiliser un **foncteur** :

```
#include <list>
#include <iostream>
#include <algorithm>

using namespace std;

// Foncteur servant à libérer un pointeur (applicable à n'importe quel type)
class Delete
{
    public:
        template <class T> void operator()(T*& p) const
        {
            cout << "delete " << p << endl;
            delete p;
            p = NULL;
        }
};

int main()
{
   // Création d'une liste de pointeurs
   std::list<int*> l;
   l.push_back(new int(5));
   l.push_back(new int(0));
   l.push_back(new int(1));
   l.push_back(new int(6));

   // Destruction de la liste : attention il faut bien libérer les pointeurs avant la
      destruction de la liste !
   std::for_each(l.begin(), l.end(), Delete());

   return 0;
}
```

Suppression des pointeurs dans un conteneur

Lien : http://cpp.developpez.com/faq/cpp/?page=Conteneurs

Recherche d'éléments (find)

La STL fournit une fonction générique `find()` pour rechercher des éléments dans un conteneur. Il faut inclure le fichier d'en-tête `<algorithm>`.

Lien : http://www.cplusplus.com/reference/algorithm/find/

```
#include <iostream>
#include <algorithm>  // pour find
#include <vector>
#include <list>

using namespace std;

int main ()
{
  int tableau[] = { 10, 20, 30, 40 };
  size_t taille = sizeof(tableau) / sizeof(tableau[0]);
  int *p;

  p = find(tableau, tableau + taille, 30);

  if (p != (tableau + taille))
    cout << "Element trouvé : " << *p << '\n';
  else
    cout << "Element non trouvé !\n";

  vector<int> vec { 10, 20, 30, 40 };
  vector<int>::iterator itV;

  itV = find(vec.begin(), vec.end(), 20);
  if (itV != vec.end())
    cout << "Element trouvé : " << *itV << '\n';
  else
    cout << "Element non trouvé !\n";

  list<int> l { 10, 20, 30, 40 };
  list<int>::iterator itL;

  itL = find(l.begin(), l.end(), 50);
  if (itL != l.end())
    cout << "Element trouvé : " << *itL << '\n';
  else
    cout << "Element non trouvé !\n";

  return 0;
}
```

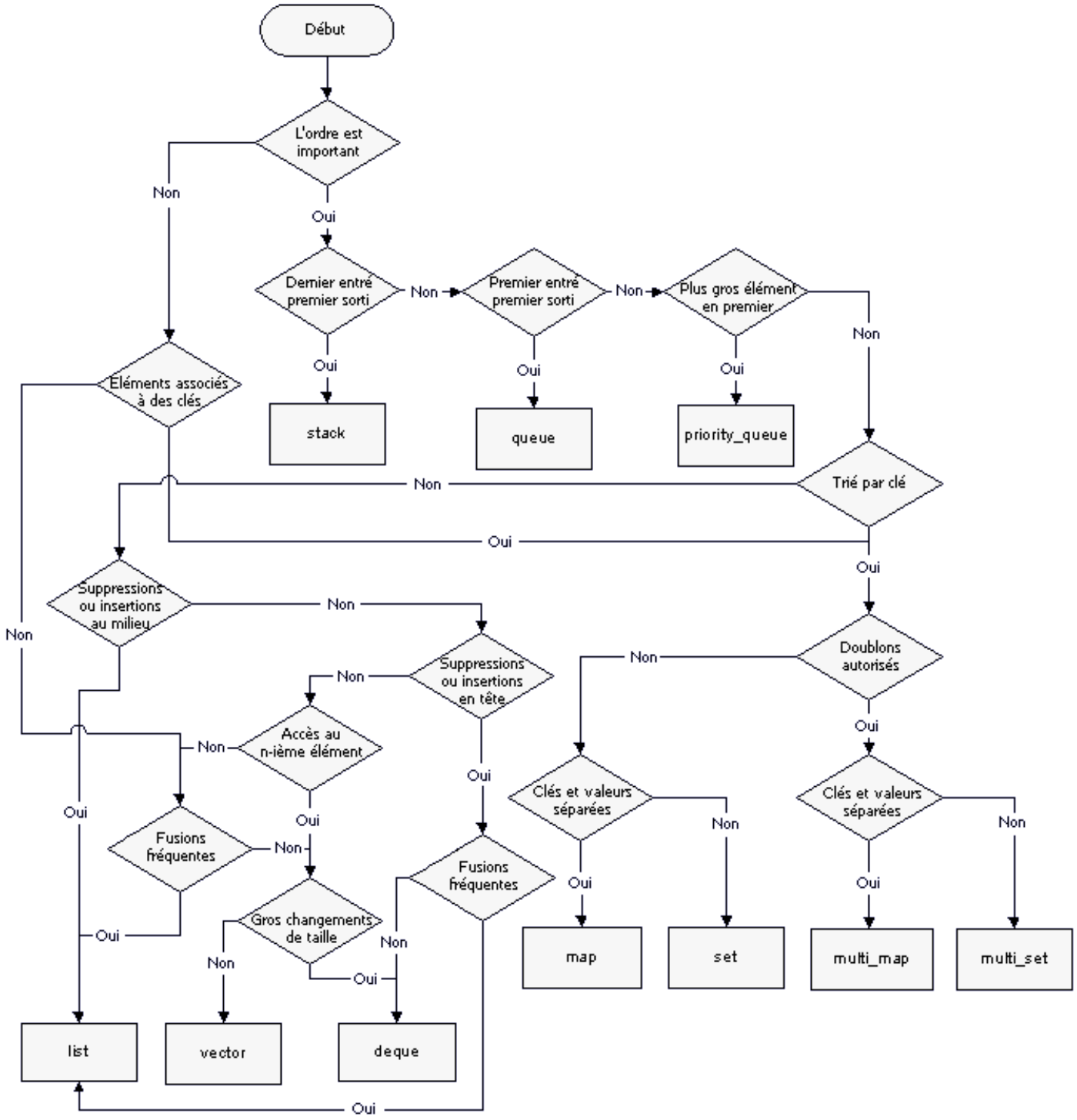
Recherche d'éléments dans un `vector`

Pour les `[multi]set` et les `[multi]map`, on utilisera la méthode membre `find()`.

Choix d'un conteneur

La panoplie de conteneurs proposée par la STL est conséquente : conteneurs ordonnés, associatifs, listes chaînées ... Le choix du "bon" conteneur dépend principalement des opérations que l'on va effectuer sur les données : suppression, ajout, recherche ...

Voici un schéma récapitulatif qui aidera à faire son choix (extrait du site www.developpez.com) :



Il existe un nouveau conteneur en C++ 11 (`-std=c++0x` ou `-std=c++11`) : `array` pour représenter un tableau de taille fixe.

Comparaison des complexités

Groupe	Fonction	vector	deque	list	set	multiset	map	multimap	bitset	stack	queue	priority_queue
Itérateurs	begin	O(1)	O(1)	O(1)	O(1)	O(1)	O(1)	O(1)	-	-	-	-
	end	O(1)	O(1)	O(1)	O(1)	O(1)	O(1)	O(1)	-	-	-	-
Accès	front	O(1)	O(1)	O(1)	-	-	-	-	-	-	O(1)	-
	back	O(1)	O(1)	O(1)	-	-	-	-	-	-	O(1)	-
	top	-	-	-	-	-	-	-	-	O(1)	-	O(1)
	operator[]	O(1)	O(1)	-	-	-	Log	-	O(1)	-	-	-
	at	O(1)	O(1)	-	-	-	-	-	-	-	-	-
Modifieurs	assign	O(n)	O(n)	O(n)	-	-	-	-	-	-	-	-
	insert	O(n+m)	O(m)	O(m)	Log	Log	Log	Log	-	-	-	-
	erase	O(n)	O(m)	O(m)	O(1)+	O(1)+	O(1)+	O(1)+	-	-	-	-
	clear	O(n)	O(n)	O(n)	O(n)	O(n)	O(n)	O(n)	-	-	-	-
	push_front	-	O(1)	O(1)	-	-	-	-	-	-	-	-
	pop_front	-	O(1)	O(1)	-	-	-	-	-	-	-	-
	push_back	O(1)	O(1)	O(1)	-	-	-	-	-	-	-	-
	pop_back	O(1)	O(1)	O(1)	-	-	-	-	-	-	-	-
	push	-	-	-	-	-	-	-	-	O(1)	O(1)	O(1)
	pop	-	-	-	-	-	-	-	-	O(1)	O(1)	O(1)
Opérations	find	-	-	-	Log	Log	Log	Log	-	-	-	-

☞ *Extrait du livre "C++ : L'essentiel du code et des commandes" de Vincent Gouvernelle (Ed. Pearson Education France)* https://books.google.com/books?isbn=2744022810

Exemple détaillé : utilisation d'un vector

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

using namespace std;

int main()
{
    // créer un tableau d'entiers vide
    std::vector<int> v;

    // ajouter l'entier 10 à la fin
    v.push_back( 10 );

    // afficher le premier élément (10)
    cout << "Premier element : " << v.front() << '\n';

    // afficher le dernier élément (10)
    cout << "Dernier element : " << v.back() << '\n';

    // enlever le dernier élément
    v.pop_back(); // supprime '10'

    // le tableau est vide
    if ( v.empty() )
    {
        cout << "Tout est normal : tableau vide\n";
    }

    // redimensionner le tableau
    // resize() initialise tous les nouveaux entiers à 0
    v.resize( 10 );

    // quelle est sa nouvelle taille ?
    cout << "Nouvelle taille : " << v.size() << '\n'; // affiche 10

    // sa taille est de 10 : on peut accéder directement aux
    // 10 premiers éléments
    v[ 9 ] = 5;

    // intitialiser tous les éléments à 100
    std::fill( v.begin(), v.end(), 100 );

    // vider le tableau
    v.clear(); // size() == 0

    // on va insérer 50 éléments
    // réserver (allouer) de la place pour au moins 50 éléments
    v.reserve( 50 );
```

```
    // vérifier que la taille n'a pas bougé (vide)
    cout << "Taille : " << v.size() << '\n';

    // capacité du tableau = nombre d'éléments qu'il peut stocker
    // sans devoir réallouer (modifié grâce à reserve())
    cout << "Capacite : " << v.capacity() << '\n'; // au moins 50, sûrement plus

    for ( int i = 0; i < 50; ++i )
    {
        // grâce à reserve() on économise de multiples réallocations
        // du fait que le tableau grossit au fur et à mesure
        v.push_back( i );
    }

    // afficher la nouvelle taille
    cout << "Nouvelle taille : " << v.size() << '\n'; // affiche 50

    // rechercher l'élément le plus grand (doit être 49)
    cout << "Max : " << *std::max_element( v.begin(), v.end() ) << '\n';

    // tronquer le tableau à 5 éléments
    v.resize( 5 );

    // les trier par ordre croissant
    std::sort( v.begin(), v.end() );

    // parcourir le tableau
    for ( size_t i = 0, size = v.size(); i < size; ++i )
    {
        // attention : utilisation de l'opérateur []
        // les accès ne sont pas vérifiés, on peut déborder !
        cout << "v[" << i << "] = " << v[ i ] << '\t';
    }
    cout << '\n';

    // utilisation de at() : les accès sont vérifiés
    try
    {
        v.at( v.size() ) = 10; // accès en dehors des limites !
    }
    catch ( const std::out_of_range &e )
    {
        cerr << "at() a levé une exception std::out_of_range : " << e.what() << endl;
    }

    // parcours avec un itérateur en inverse
    for ( std::vector<int>::reverse_iterator i = v.rbegin(); i != v.rend(); ++i )
    {
        cout << *i << '\t';
    }
    cout << '\n';

    // on crée un tableau v2 de taille 10
```

```
    std::vector<int> v2( 10 );
    v2.at( 9 ) = 5; // correct, le tableau est de taille 10

    // on crée un tableau v3 de 10 éléments initialisés à 20
    std::vector<int> v3( 10, 20 );

    // faire la somme de tous les éléments de v3
    // on doit obtenir 200 (10 * 20)
    cout << "Somme : " << std::accumulate( v3.begin(), v3.end(), 0 ) << '\n';

    // on recopie v3 dans v
    v = v3;

    // on vérifie la recopie
    if ( std::equal( v.begin(), v.end(), v3.begin() ) )
    {
        cout << "v est bien une copie conforme de v3\n";
    }

    return 0;
}
```

Utilisation détaillée d'un vector

On obtient à l'exécution :

```
Premier element : 10
Dernier element : 10
Tout est normal : tableau vide
Nouvelle taille : 10
Taille : 0
Capacite : 50
Nouvelle taille : 50
Max : 49
v[0] = 0      v[1] = 1      v[2] = 2      v[3] = 3      v[4] = 4
at() a levé une exception std::out_of_range : vector::_M_range_check
4      3      2      1      0
Somme : 200
v est bien une copie conforme de v3
```

Liste des exemples

Exemple d'utilisation d'un `vector` . . . 4
Exemple d'utilisation d'un `iterator` . . . 5
Exemple d'utilisation d'une `list` . . . 6
Exemple d'utilisation d'une `map` . . . 8
Exemple d'utilisation d'un `pair` . . . 9
Exemple d'utilisation d'un `set` . . . 11
Exemple d'utilisation d'une `stack` . . . 13
Exemple d'utilisation d'une `queue` . . . 15
Exemple d'utilisation d'une `priority_queue` . . . 15
Exemple d'utilisation d'un `unordered_map` . . . 17
Tri d'un tableau . . . 20
Tri d'un `vector` . . . 21
Tri d'objets . . . 22
Tri décroissant d'un `vector` . . . 23
Différents tris d'objets . . . 24
Suppression d'éléments dans un `vector` . . . 27
Suppression d'éléments dans une`list` . . . 28
Suppression des pointeurs dans un conteneur . . . 29
Recherche d'éléments dans un `vector` . . . 30
Utilisation détaillée d'un vector . . . 33