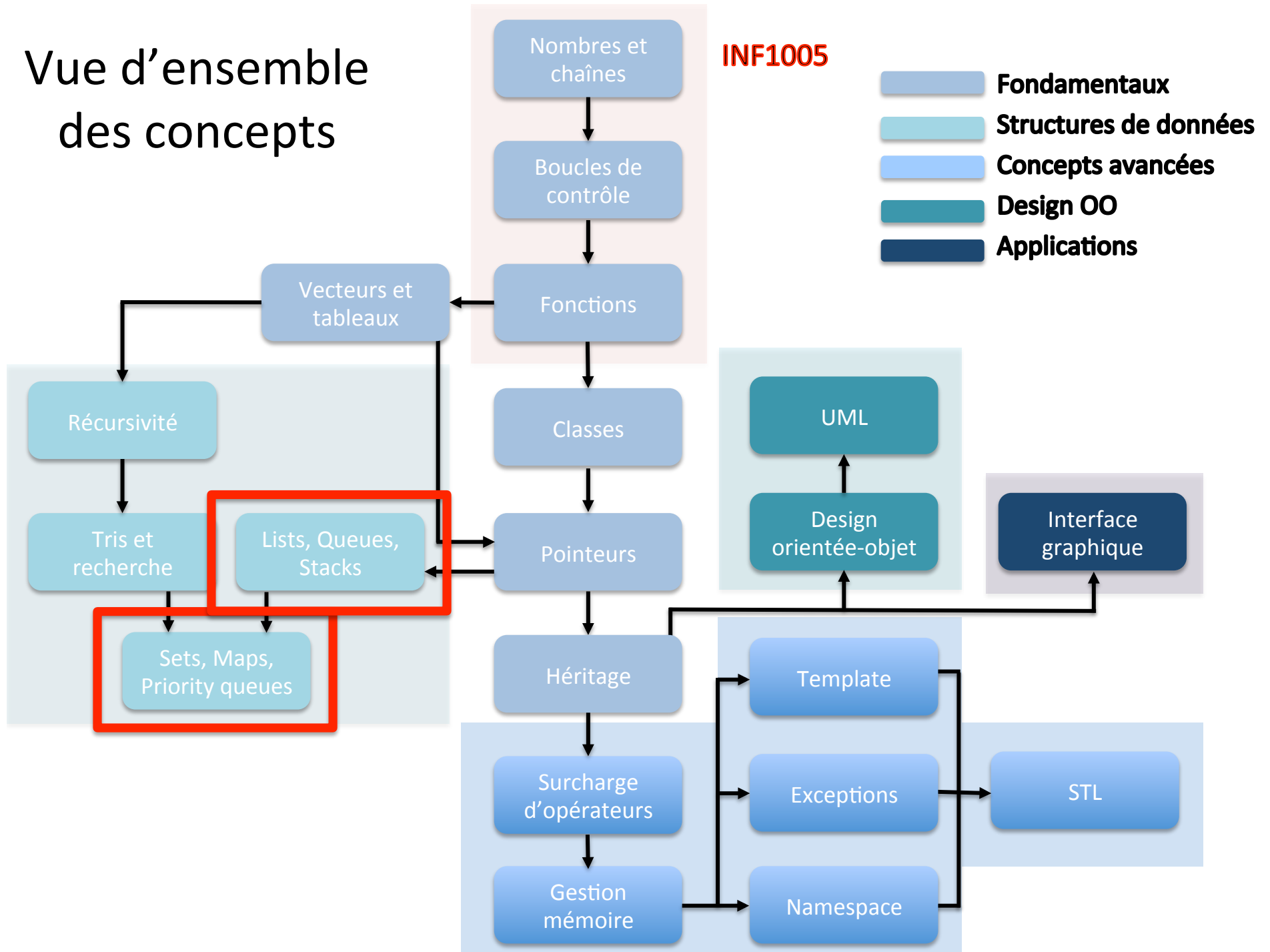


Programmation orientée objet

Conteneurs STL

Vue d'ensemble des concepts

INF1005



Aperçu

- Introduction
- Éléments
- Classification par concepts :
 - Conteneur simple
 - À itération vers l'avant
 - Réversible
 - À accès aléatoire
- Classification par organisation :
 - Séquentiels
 - Associatifs
 - Adaptateurs

Introduction

- **Conteneur** : objet qui contient d'autres objets : ses éléments.
- Chaque conteneur a sa façon distincte de manipuler et de stocker ses items (ex : tableau, heap, liste liée).
- **Les conteneurs possèdent leurs éléments.** Lorsqu'ils sont détruits, leurs éléments le sont aussi .
- Ils possèdent des classes imbriquées :
 - Type des éléments contenus (`conteneur::value_type`)
 - Type des itérateurs associés (`conteneur::iterator`)
 - Type du pointeur vers un élément (`conteneur::pointer`)

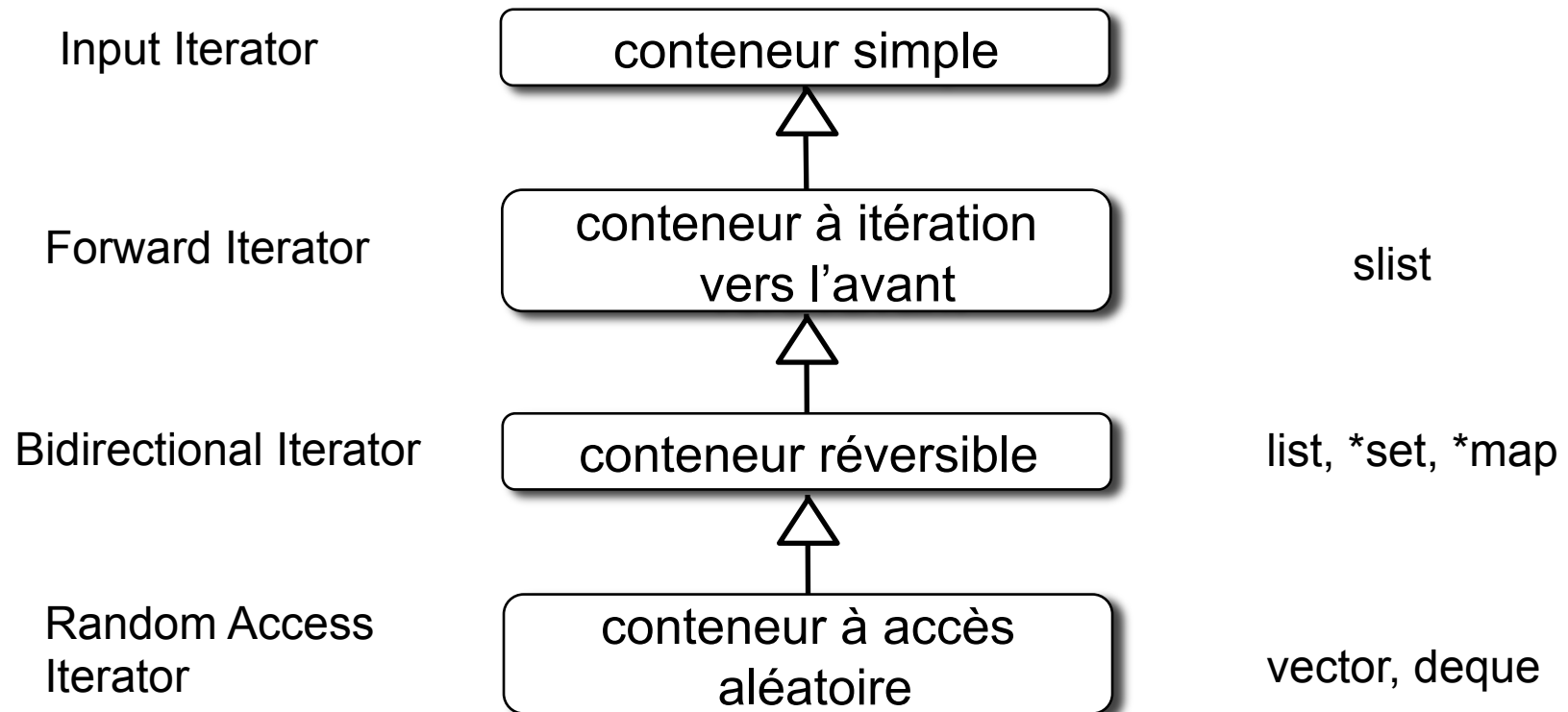
Spécifications pour les éléments

- Les éléments qui sont insérés dans un conteneur doivent posséder :

- Un constructeur par copie
- Un destructeur
- Opérateur =
- Un constructeur par défaut (facultatif)
- Opérateur == (facultatif)
- Opérateur < (facultatif)

Pourquoi? Clarifier si on veut agrégation ou composition

Classification par concepts



Conteneur simple

- Répond à la définition d'un conteneur (intro) :
 - Contient d'autres objets.
 - Possède ses éléments.
 - Permet de manipuler ses éléments.
 - Inclut des classes imbriquées
- Ne garantie **pas** que ses éléments soient **ordonnés**.
- Ne garantie **pas que plusieurs itérateurs soient actifs** en même temps pour le même conteneur.
- *Tous les conteneurs de la STL font partie de ce concept.*

À itération vers l'avant (Forward Container)

- Possède un itérateur avançant (`Forward Iterator`)
- Ses éléments sont stockés selon **un ordre défini**.
- Permet :
 - La comparaison d'égalité élément par élément.
 - La comparaison operator `<` des éléments.
 - L'application d'algorithmes à **passage multiple**.
- Permet à ses itérateurs :
 - De repasser plusieurs fois sur le même intervalle (ordre défini).
 - Uniquement d'avancer.

À itération vers l'avant (Forward Container) (suite)

- *Tous les conteneurs de la STL font partie de ce concept.*
- `slist<T>` : (singly linked list)
 - Seul conteneur à être purement un `Forward Container`.
 - Une liste à liens simples dont chaque nœud a un pointeur vers l'élément suivant, mais pas vers le précédent.

Réversible

- Possède un **itérateur bidirectionnel**.
- Permet toutes les opérations faites sur les conteneurs à itérations vers l'avant.
- Itérateurs peuvent **aussi reculer** (bidirectionnel).
- Degré de flexibilité maximal de la *plupart des conteneurs* de la STL :
 - `list<T>`
 - `set<T>` et `map<T,S>`
 - `multiset<T>` et `multimap<T,S>`
 - `hash_set<T>` et `hash_map<T,S>`

Réversible (exemples)

- Déclarations de conteneurs réversibles :
 - `list<int> listeDeInts;`
 - `map<string, double> notesDesEleves;`
- Déclarations d'itérateurs bidirectionnels :
 - `set<Point>::iterator it = unSet.begin();`
 - `hash_map<char, int>::reverse_iterator revIt = hMap.rbegin();`
- Déplacements d'itérateurs dans des conteneurs :
 - `++it; // avance d'une position`
 - `--it; // recule d'une position`
 - `++revIt; // recule d'une position`
 - `--revIt; // avance d'une position`

À accès aléatoire

(Random Access Container)

- Possède un itérateur à accès aléatoire (`Random Access Iterator`).
- Se déplace dans **les deux directions**, les itérateurs peuvent aussi faire des **sauts**.
- *Seulement les conteneurs séquentiels basés sur des tableaux* sont de cette catégorie :
 - `vector<T>`
 - `deque<T>`
- On peut utiliser **l'opérateur []** pour accéder directement aux éléments.

À accès aléatoire (exemples)

- Utilisation de l'opérateur [] :

- `vector<int> vectDeInts;`

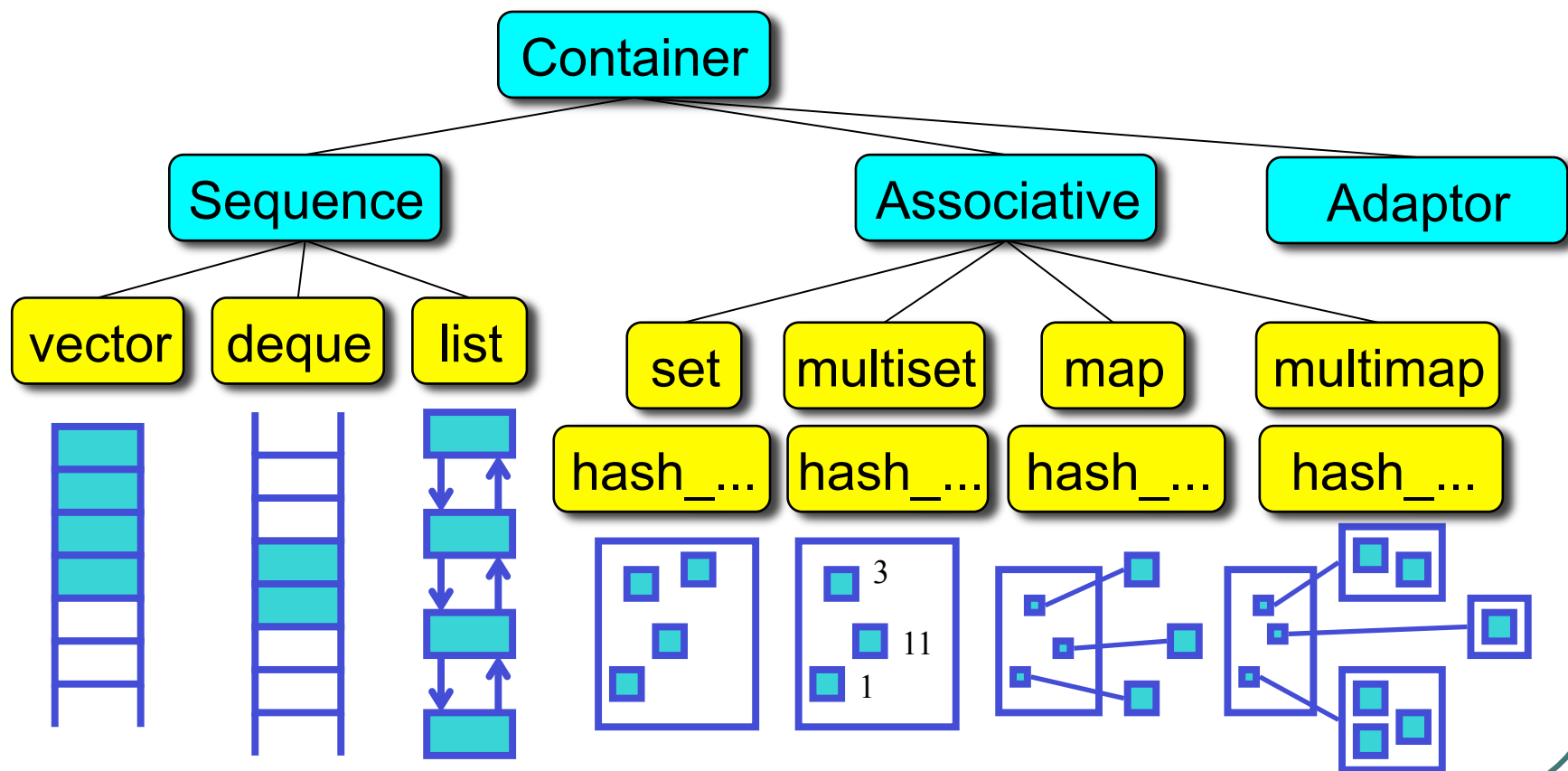
```
vectDeInts.push_back(10);  
vectDeInts.push_back(25);  
vectDeInts.push_back(40);  
cout << vectDeInts[1] << endl;    // Imprime 25 à la console
```

- Déplacement d'un itérateur par un saut :



- `deque<double> dequeDeDoubles;`

```
dequeDeDoubles.push_back(-26.0);  
dequeDeDoubles.push_front(37.9);  
dequeDeDoubles.push_front(13.78);  
  
deque<double>::iterator it = dequeDeDoubles.begin();  
it += 2;  
cout << *it << endl; // Imprime -26.0 à la console
```

Classification par organisation



Classification par organisation (suite)

- Plus pratique pour classer les conteneurs.
- Organisation des éléments
 - **Séquentiels** : conteneurs avec **une organisation linéaire**. 
 - **Associatifs** : conteneurs optimisés pour la **recherche par clefs**. 
- Choix d'une interface pour un conteneur déjà existant :
 - **Adaptateurs** : restreignent l'interface et fournissent des méthodes correspondant à un comportement particulier (Ex : LIFO).

Conteneurs

(méthodes communes)

- Conteneurs: petit groupe de méthodes en commun :
 - `begin()` : retourne un itérateur qui pointe au début du conteneur.
 - `end()` : retourne un itérateur qui pointe après le dernier élément.
 - `size()` : retourne le nombre d'éléments contenus.
 - `max_size()` : retourne le nombre maximal d'éléments que peut posséder le conteneur (déterminé par système d'exploitation).
 - `empty()` : indique si le conteneur est vide.
 - `swap(Conteneur c)` : échange les éléments des deux conteneurs.

Les conteneurs séquentiels

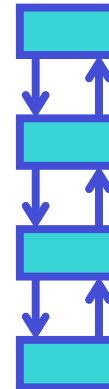
vector



deque



list



Séquentiels (conteneurs)

- **vector<T>** : tableau dynamique à capacité variable qui permet l'accès à ses items par l'opérateur d'élément [] et une **insertion par l'arrière**.
- **deque<T>** : tableau dynamique semblable à **vector<T>** qui permet une **insertion efficace par l'avant en plus de l'arrière**.
- **list<T>** : liste liée à liens doubles, rapide peu importe l'endroit d'insertion, mais qui n'a **pas d'accès direct** à ses éléments.

Exemple d'utilisation d'un conteneur

```
vector<int> nombres;  
nombres.push_back(3);  
nombres.push_back(8);  
...  
nombres.push_back(42);  
...  
vector<int>::iterator pos = nombres.begin();  
  
// on élimine les nombres pairs  
while (pos != nombres.end()) {  
    if (*pos % 2 == 0)  
        pos = nombres.erase(pos);  
    else  
        ++pos;  
}
```

erase est une méthode qui élimine l'item pointé par l'itérateur passé en paramètre et retourne un itérateur sur l'élément qui suit l'élément retiré.

Exemple d'utilisation d'un conteneur (suite)

```
typedef vector<int> Conteneur;  
Conteneur nombres;  
nombres.push_back(3);  
nombres.push_back(8);  
...  
nombres.push_back(42);  
...  
Conteneur::iterator pos = nombres.begin();  
  
// on élimine les nombres pairs  
while (pos != nombres.end()) {  
    if (*pos % 2 == 0)  
        pos = nombres.erase(pos);  
    else  
        ++pos;  
}
```

En utilisant **typedef**, on simplifie l'écriture du code.

Exemple d'utilisation d'un conteneur (suite)

```
typedef list<int> Conteneur;  
Conteneur nombres;  
nombres.push_back(3);  
nombres.push_back(8);  
...  
nombres.push_back(42);  
...  
Conteneur::iterator pos = nombres.begin();  
  
// on élimine les nombres pairs  
while (pos != nombres.end()) {  
    if (*pos % 2 == 0)  
        pos = nombres.erase(pos);  
    else  
        ++pos;  
}
```

On peut changer pour une liste et le code reste inchangé.

Exemple d'utilisation d'un conteneur

```
typedef deque<int> Conteneur;  
Conteneur nombres;  
nombres.push_back(3);  
nombres.push_back(8);  
...  
nombres.push_back(42);  
...  
Conteneur::iterator pos = nombres.begin();  
  
// on élimine les nombres pairs  
while (pos != nombres.end()) {  
    if (*pos % 2 == 0)  
        pos = nombres.erase(pos);  
    else  
        ++pos;  
}
```

On peut aussi changer pour un **deque** et le code est toujours le même!

Séquentiels

(constructeurs communs)

- Tous les conteneurs séquentiels possèdent les mêmes cinq constructeurs :
 - `vector<T>()` : constructeur par défaut qui crée un vecteur vide.
 - `vector<T>(const vector<T>& vect)` : constructeur par copie.
 - `vector<T>(taille)` : remplit de `taille` éléments construits par défaut.
 - `vector<T>(taille,valeur)` : remplit de `taille` éléments égaux à `valeur`.
 - `vector<T>(itérateur,itérateur)` : remplit en copiant les éléments compris dans l'intervalle des itérateurs.
- Les conteneurs séquentiels `list<T>` et `deque<T>` ont les mêmes constructeurs.

Séquentiels

(méthodes communes)

- `insert(it, x)` : insère la valeur `x` juste avant l'élément pointé par `it`.
- `insert(it, i, j)` : les itérateurs `i` et `j` délimitent un intervalle de valeur qui est inséré juste avant l'élément pointé par `it`.
- `erase(it)` : retire l'élément pointé par `it`.
- `erase(it1, it2)` : retire les éléments compris dans l'intervalle `[it1, it2[`.
- `clear()` : vide le conteneur en détruisant tous les éléments
- `resize(n)` : redimensionne le conteneur pour qu'il possède exactement `n` éléments (retrait ou ajout d'éléments selon le cas).

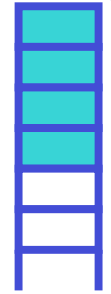
Séquentiels (suite)

- Les éléments sont placés dans un ordre strictement linéaire qui est souvent l'ordre d'arrivée.
- L'emplacement d'un élément ne dépend pas de ses caractéristiques.
- On peut les remplir/lire/vider par l'avant :
 - `slist<T>`, `list<T>` et `deque<T>`
avec `push_front(valeur)`, `front()` et `pop_front()`
- Et/ou par l'arrière :
 - `vector<T>`, `list<T>` et `deque<T>`
avec `push_back(valeur)`, `back()` et `pop_back()`

Séquentiels (suite)

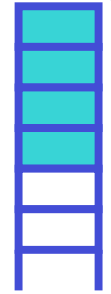
- Ils permettent tous d'insérer ou de retirer un élément au milieu, mais `list<T>` et `slist<T>` le font en temps constant.
- Pour les autres, c'est une opération coûteuse.
- Selon la classification par concepts :
 - `vector<T>` est à accès aléatoire.
 - `deque<T>` est à accès aléatoire.
 - `list<T>` est réversible.
 - `slist<T>` est à itération vers l'avant (Forward Container).

Vecteur



- Par la surcharge de l'opérateur d'indexation `[]`, il s'utilise comme un tableau C++.
- Il ne faut pas confondre taille et capacité :
 - `unVect.size()` : taille du tableau, le nombre d'éléments contenus.
 - `unVect.capacity()` : capacité du tableau, taille maximale du vector.
- Si on insère plus d'éléments que le capacité le permet, le vecteur augmentera automatiquement sa capacité.
- Les méthodes `insert()` et `erase()` sont très coûteuses, car tous les éléments qui suivent l'endroit d'insertion ou de retrait doivent être décalés.
- Seules les méthodes d'insertion et de retrait `push_back()` et `pop_back()` se font en temps constant si la capacité le permet.

Vecteurs – capacité et taille



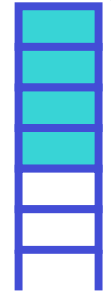
```
int main()
{
    vector<int> unTableau;
    cout << "capacite initiale:  " << unTableau.capacity()
        << endl;
    for (int i = 1; i < 10; ++i) {
        unTableau.push_back(i);
        cout << "capacite:  " << unTableau.capacity()
            << endl;
    }
    for (int j = 0; j < unTableau.size(); ++j)
        cout << unTableau[j] << ' ';
    cout << endl;
    return(0);
}
```

Initialement, la capacité est 0.

La capacité sera égale à 16 si elle doublée à chaque réajustement.

La taille sera égale à 10.

Vecteurs – capacité et taille



- Si on sait déjà que la taille d'un vecteur peut atteindre une certaine valeur, on peut fixer la capacité, ce qui évitera des réajustements coûteux:

```
vector< int > v;  
v.reserve(100);
```

Capacité fixée à 100. Il faudra donc 100 éléments dans le vecteur avant de devoir réajuster la taille du tableau interne.

Construction d'un conteneur séquentiel

```
int main()
{
    vector< int > coll;

    cout << coll.size();

    vector<int>::iterator iter = coll.begin();

    cout << *iter;

    ...
}
```

Le constructeur par défaut construit un conteneur vide.

La valeur retournée sera zéro.

Erreur!
Le vecteur est vide!

Construction d'un conteneur séquentiel (suite)

```
int main()
{
    list< int > coll;

    cout << coll.size();

    list<int>::iterator iter = coll.begin();

    cout << *iter;

    ...
}
```

Le constructeur par défaut construit un conteneur vide.

La valeur retournée sera zéro.

Erreur!
La liste est vide!

Construction d'un conteneur séquentiel (suite)

```
int main()  
{  
    vector< int > coll(10) ;  
  
    cout << coll.size() ;  
  
    cout << coll[1] ;  
    ...  
}
```

Initialise le conteneur avec 10 éléments. Ici, chaque élément aura la valeur par défaut du type, c'est-à-dire 0.

La valeur retournée sera 10.

La valeur 0 sera affichée.

Construction d'un conteneur séquentiel (suite)

```
int main()
{
    vector< Clock > coll(10) ;

    cout << coll.size() ;

    cout << coll[1].get_hours() ;
    ...
}
```

Initialise le conteneur avec 10 objets de la classe **Clock**. Chaque objet sera construit en utilisant le constructeur par défaut de la classe **Clock**.

La valeur retournée sera 10.

L'heure du deuxième item sera affichée.

Construction d'un conteneur séquentiel (suite)

```
int main()
{
    list< Clock > coll(10);

    cout << coll.size();

    list< Clock >::iterator iter = coll.begin();
    ++iter;
    cout << iter->get_hours();
    ...
}
```

Initialise le conteneur avec 10 objets de la classe **Clock**. Chaque objet sera construit en utilisant le constructeur par défaut de la classe **Clock**.

La valeur retournée sera 10.

Affichera l'heure du deuxième objet de la liste.

Construction d'un conteneur séquentiel (suite)

```
int main()
{
    vector< int > coll(10,18) ;

    cout << coll.size() ;

    cout << coll[1] ;
    ...
}
```

Initialise le conteneur avec 10 éléments. Ici, la valeur 18 sera copiée à chaque position du vecteur.

La valeur 18 sera affichée.

Construction d'un conteneur séquentiel (suite)

```
int main()
{
    vector< int > v;
    for (int i = 0; i < 10; ++i)
        v.push_back(i*i);

    list< int > list(v.begin(), v.end());

    ...
}
```

La liste sera construite en y plaçant tous les éléments contenus dans le vecteur **v**.



Deque

- Signifie «double-ended queue».
- Très semblable à `vector<T>` :
 - Opérateur d'indexation `[]`
 - Tableau dynamique à capacité variable sans toutefois définir de méthode `capacity()`.
 - Méthodes `insert()` et `erase()` toujours aussi lentes.
 - Ajout et retrait à la fin en temps constant.
- Permet en plus la manipulation en temps constant du premier élément avec `push_front()`, `front()` et `pop_front()`.



Liste

- Implémentée comme une liste liée à liens doubles.
- Accès direct au premier et au dernier éléments.
- Peu importe l'endroit où pointe un itérateur d'une liste, l'insertion et le retrait est en temps constant.
- Par contre, il n'y a aucun accès direct aux éléments du milieu.
- Il faut parcourir la liste du début ou de la fin pour atteindre un élément (temps linéaire).



Liste (méthodes spécifiques)

- **merge(liste)** : permet de combiner deux listes triées dans le conteneur appelant tout en conservant l'ordre des éléments si les deux listes sont triées avec l'opérateur $<$.
- **merge(liste, prédicat binaire)** : même opération mais en utilisant un prédicat binaire passé en argument pour le tri.
- **remove(valeur)** : retire toutes les occurrences de la valeur dans la liste.
- **remove_if(prédicat)** : retire tous les éléments qui retournent vrai selon un prédicat.
- **reverse()** : inverse la position de tous les éléments de la liste.
- **sort()** : trie les éléments selon l'opérateur $<$.
- **sort(prédicat binaire)** : trie selon le prédicat binaire .



Prédicat

- Si une fonction retourne un `bool`, elle est appelée prédicat.
 - **Prédicat unaire** (ou simplement **Prédicat**) : un seul paramètre.
 - **Prédicat binaire** : deux paramètres de même type.
- Exemple de prédicat binaire :
`bool sontSecant(Cercle cercle1, Cercle cercle2)` : retourne vraie si les cercles ont deux points en commun.



Listes – exemple 1

```
int main()
{
    list< char > uneListe;
    for (char c = 'a'; c <= 'z'; ++c) {
        uneListe.push_back(c);
    }
    while (!uneListe.empty()) {
        cout << uneListe.front() << ' ';
        uneListe.pop_front();
    }
    cout << endl;
    return(0);
}
```

On remplit une liste
de toutes les lettres
de a à z.

On affiche tous les
éléments de la liste.



Listes – exemple 2

```
bool voyelle(char c)
{
    return (c == 'a' ||
            c == 'e' ||
            c == 'i' ||
            c == 'o' ||
            c == 'u') ;
}
```



Listes – exemple 2 (suite)

```
int main()
{
    list<char> uneListe;
    for (char c='a'; c<='z'; ++c)
        uneListe.push_back(c);
    uneListe.remove_if(voyelle);
    uneListe.reverse();
    while (!uneListe.empty()) {
        cout << uneListe.front() << ' ';
        uneListe.pop_front();
    }
    cout << endl;
    return(0);
}
```



Listes – exemple 2 (suite)

```
int main()
{
    list<char> uneListe;
    for (char c='a'; c<='z'; ++c)
        uneListe.push_back(c);
    uneListe.remove_if(voyelle);
    uneListe.reverse();
    while (!uneListe.empty()) {
        cout << uneListe.front() << ' ';
        uneListe.pop_front();
    }
    cout << endl;
    return(0);
}
```

On remplit une liste
de toutes les lettres
de a à z.



Listes – exemple 2 (suite)

```
int main()
{
    list<char> uneListe;
    for (char c='a'; c<='z'; ++c)
        uneListe.push_back(c);
    uneListe.remove_if(voyelle);
    uneListe.reverse();
    while (!uneListe.empty()) {
        cout << uneListe.front() << ' ';
        uneListe.pop_front();
    }
    cout << endl;
    return(0);
}
```

On retire toutes les voyelles.



Listes – exemple 2 (suite)

```
int main()
{
    list<char> uneListe;
    for (char c='a'; c<='z'; ++c)
        uneListe.push_back(c);
    uneListe.remove_if(voyelle);
    uneListe.reverse();
    while (!uneListe.empty()) {
        cout << uneListe.front() << ' ';
        uneListe.pop_front();
    }
    cout << endl;
    return(0);
}
```

On inverse la liste.



Listes – exemple 2 (suite)

```
int main()
{
    list<char> uneListe;
    for (char c='a'; c<='z'; ++c)
        uneListe.push_back(c);
    uneListe.remove_if(voyelle);
    uneListe.reverse();
    while (!uneListe.empty()) {
        cout << uneListe.front() << ' ';
        uneListe.pop_front();
    }
    cout << endl;
    return(0);
}
```

On affiche le résultat.



Parcours d'une liste (exemple 3)

```
list<Point> listePoint;
```

```
listePoint.push_back(Point(10,10));
```

```
listePoint.push_back(Point(4,5));
```

```
listePoint.push_back(Point(1,1));
```

```
listePoint.push_back(Point(3,8));
```

```
list<Point>::iterator pos = listePoint.begin();
```

```
list<Point>::iterator fin = listePoint.end();
```

```
while (pos!=fin) {
```

```
    cout <<*pos;
```

```
    pos++;
```

```
}
```

```
listePoint.sort();
```

Sortie:

(10,10)

(4,5)

(1,1)

(3,8)

pos++ pointe
sur l'élément
suivant

Tri par <



Trier List opérateur <

```
bool Point::operator < (const Point &P)
```

```
{ return (x_*x_+y_*y_) < (P.x_*P.x_+P.y_*P.y_) ; }
```

```
...  
listePoint.sort();  
...  
while (pos!=fin)  
{  
    cout <<*pos;  
    pos++;  
}
```

Sortie:
(1,1)
(4,5)
(3,8)
(10,10)

Tri selon
l'opérateur <

Trier par prédicat binaire

Fonction globale

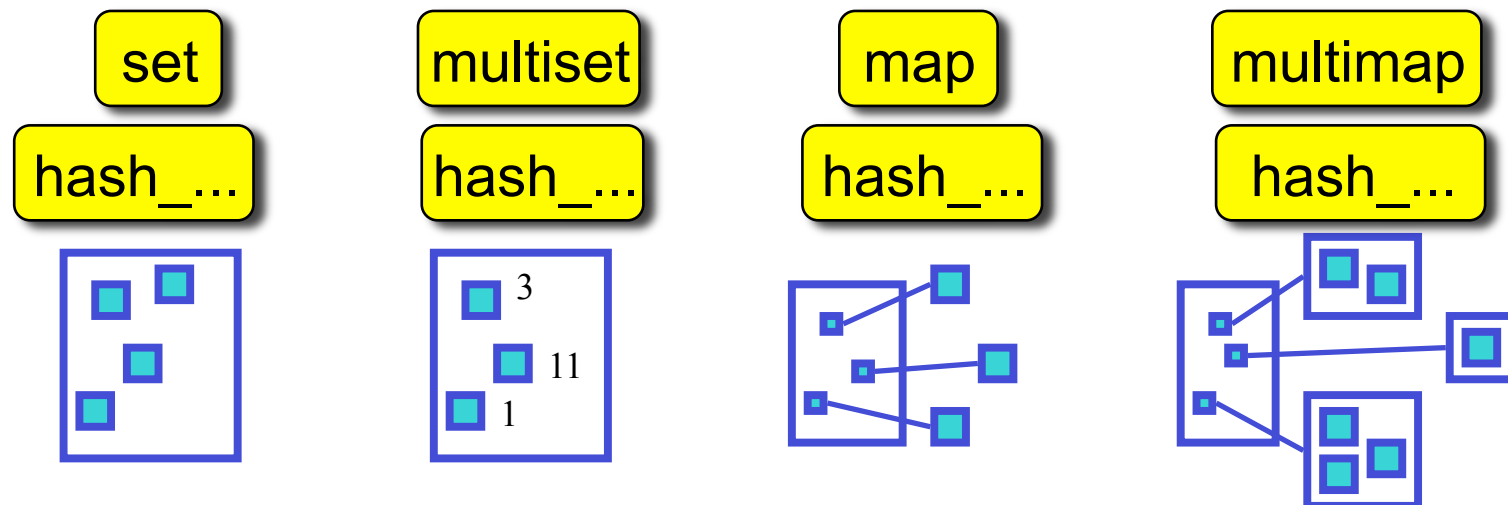


```
bool operateur(const Point & P1, const Point & P2)
{ return P1.getX() < P2.getX(); }
```

```
listePoint.sort(operateur);
pos = listePoint.begin();
while (pos != fin)
{
    cout << *pos;
    pos++;
}
```

Tri selon le prédicat globale

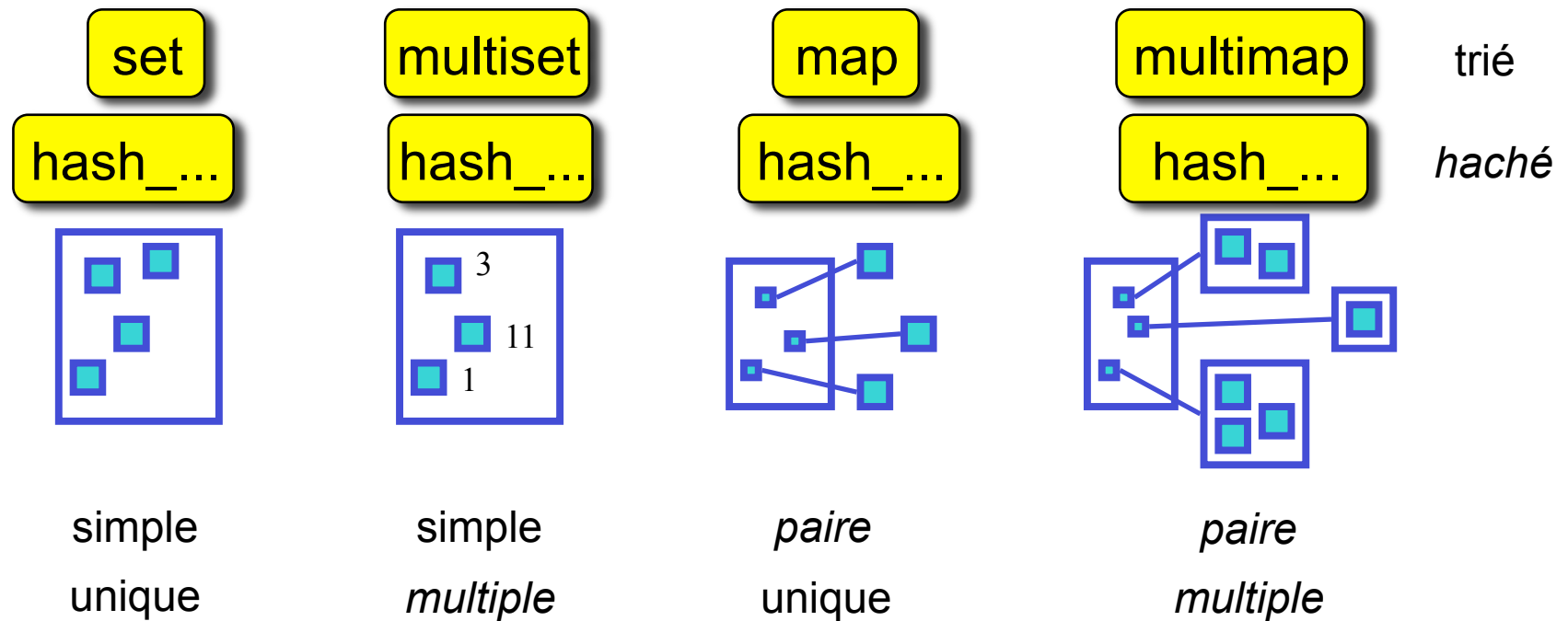
Les conteneurs associatifs



Conteneurs Associatifs

- On peut subdiviser les conteneurs associatifs en plusieurs familles :
 - **Simple** : Les éléments sont leur propre clef.
 - **Paire** : Éléments du type `pair<const key, data>` où chaque valeur (data) est associé à une clef (const key).
 - **Trié** : Les clefs sont triés en ordre croissant.
 - **Haché** : Les clefs sont traduites selon une table de hachage
 - **Unique** : Un élément ne peut être présent qu'une seule fois.
 - **Multiple** : Un élément peut être présent plusieurs fois.

Les conteneurs associatifs



Associatifs

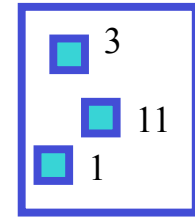
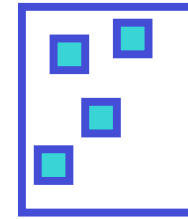
(méthodes communes)

- `erase(k)` : retire tous les éléments qui s'identifient à la clef `k`.
- `erase(it)` : retire l'élément pointé par `it`.
- `erase(it1, it2)` : retire les éléments compris dans l'intervalle `[it1, it2[`.
- `clear()` : retire et détruit tous les éléments du conteneur.
- `find(k)` : retourne un itérateur qui pointe sur un élément dont la clef est `k` ou `end()` si cette clef n'est pas présente.
- `count(k)` : retourne le nombre d'éléments qui sont associés à la clef `k`.

Associatifs (suite)

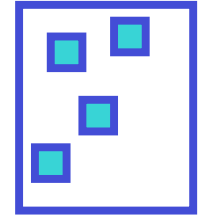
- La recherche d'élément à partir d'une clef; elle peut être la valeur elle-même (simple) ou une valeur associée (key-value paire)
- La clef est toujours déclaré `const`.
- La clef ne permet pas d'insérer une valeur à un endroit précis.
- La clef détermine la position d'insertion ou de retrait:
 - `insert(value)` : Pour ajouter un élément
 - `erase(clef ou itérateur)` : Pour retirer un élément

Associatifs : Simple



- Tous ceux qui se nomment `*set` (`set`, `multiset`, `hash_set`, `hash_multiset`).
- Une valeur **est** une clef.
- Un itérateur pointant sur un élément d'un conteneur associatif simple **n'est pas mutable** (Pour garantir le tri, si applicable!) :
 - ```
set<int>::iterator it = unSet.begin();
*it = 3; // Refusé à la compilation
```
- Pour modifier un élément 'x', il faut :
  - Retirer : `unSet.erase(x);`
  - Modifier : `x = 3;`
  - Réinsérer : `unSet.insert(x);`
- Classes imbriquées :
  - `conteneur::iterator` (équivalent à `conteneur::const_iterator`)
  - `conteneur::key_type` (équivalent à `conteneur::value_type`)

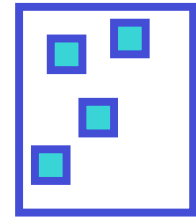




## Conteneur *set* - exemple

---

- Supposons un programme qui doit stocker une liste de noms d'étudiants inscrits.
- On peut utiliser un set, qui les conservera en ordre alphabétique.
- Ainsi, il sera facile d'afficher la liste en ordre alphabétique.
- Il sera aussi facile de vérifier si un étudiant est inscrit.

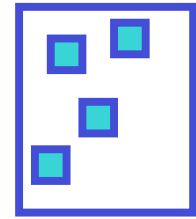


## Conteneur *set* – exemple (suite)

---

```
int main()
{
 set< string > inscrits;
 inscrits.insert("xiao chen");
 inscrits.insert("mehdi");
 inscrits.insert("roberta");

 set< string >::iterator iter;
 for (iter = inscrits.begin();
 iter < inscrits.end(); ++iter) {
 cout << *iter;
 }
 return(0);
}
```



## Conteneur *set* – exemple (suite)

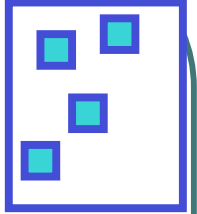
---

```
int main()
{
 set< string > inscrits;
 inscrits.insert("xiao chen");
 inscrits.insert("mehdi");
 inscrits.insert("roberta");

 if (inscrits.find("mehdi") != inscrits.end())
 cout << "mehdi est inscrit";

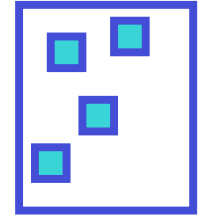
 return(0);
}
```

# Conteneur *set* – exemple d'utilisation de la valeur de retour



```
int main()
{
 string nom;
 set< string > inscrits;
 cout << "Entrez un nom: " << endl;
 cin >> nom;
 while (nom != "stop") {
 if (inscrits.insert(nom).second)
 cout << "Nom ajouté" << endl;
 else
 cout << "Le nom existe déjà" << endl;
 cout << "Entrez un nom: " << endl;
 cin >> nom;
 }
 return(0);
}
```

Si le nom existe déjà, l'insertion échoue et la seconde valeur de la paire retournée est *false*.

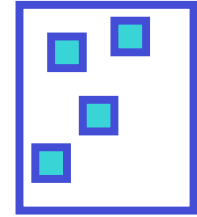


## Exemple : set

```
bool Point::operator < (const Point &P) const
{ return (x_*x_+y_*y_) < (P.x_*P.x_+P.y_*P.y_); }

set<Point> setPoint;
setPoint.insert(Point(10,10));
setPoint.insert(Point(4,5));
setPoint.insert(Point(1,1));
setPoint.insert(Point(3,8));
set<Point>::iterator courant= setPoint.begin();
set<Point>::iterator final = setPoint.end();
while (courant != final)
{ cout <<*courant;
 courant++;
}
```

Insertion en  
respectant  
l'ordre <



# Set: insertion unique

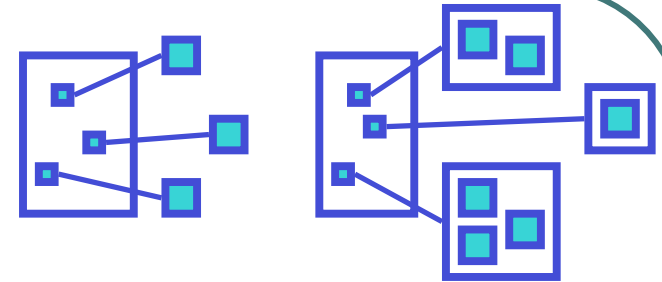
```
set<Point> setPoint;
pair <set<Point>::iterator, bool> retour;
```

```
retour = setPoint.insert(Point(10,10));
if (retour.second)
 cout << "insertion "<<endl;
else
 cout << "non insertion"<<endl;
```

```
retour = setPoint.insert(Point(10,10));
if (retour.second)
 cout << "insertion "<<endl;
else
 cout << "non insertion"<<endl;
```

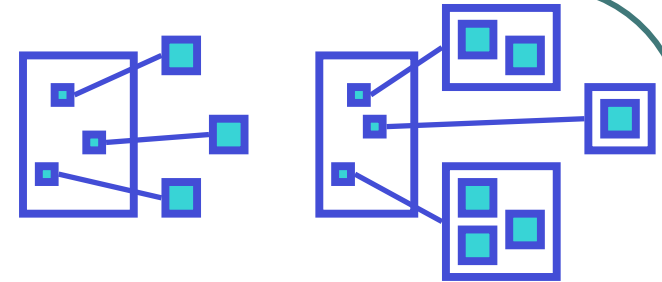
Faux: déjà  
inséré

# Associatifs : Paire



- Avant de parler des conteneurs paires, il faut savoir que leurs éléments sont des objets de type `pair<S, T>` :
  - ```
#include <utility>
template <typename F, typename S>
class pair{
public :
    /* ... */
    F first;
    S second;
};
```
- Il n'y a pas de méthodes `get*()` et `set*()`; les attributs `first` et `second` sont directement accessibles.
- C'est un conteneur spécialement utile pour stocker une paire d'éléments de types hétérogènes:
 - `pair<string, double>("Jeremy", 16.42);`
 - `make_pair("Jeremy", 16.42);`

Associatifs (suite)



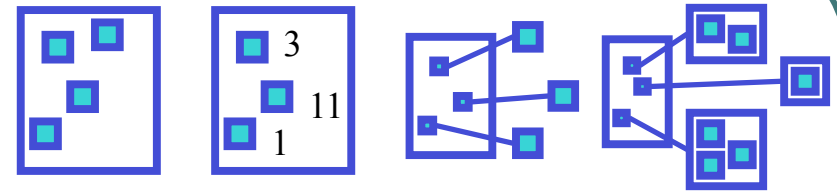
- Tous ceux qui se nomment `*map` (`map`, `multimap`, `hash_map`, `hash_multimap`)
- Chaque élément est une paire clef/valeur : `pair<key, data>`
- Un itérateur pointant sur un élément d'un conteneur associatif paire n'est pas mutable (pour garantir le tri, si applicable), mais **permet de modifier la valeur** (data):

```
map<int, double>::iterator it = unMap.begin();
*it = make_pair(2, 5.9);           // Refusé à la compilation
(*it).first = 2;                   // Refusé à la compilation
(*it).second = 5.9;                // Accepté
```

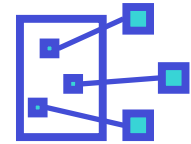
`make_pair()` :
construit un `pair`
en déduisant le
type des arguments

- Classes imbriquées :
 - `conteneur::iterator` (différent de `conteneur::const_iterator`)
 - `conteneur::key_type`
 - `conteneur::data_type`
 - `conteneur::value_type` (`pair<const key_type, data_type>`)

Associatifs : Trié



- Tous ceux qui n'ont pas le préfixe `hash_`
 - `set<T>`, `map<T>`, `multiset<T>`, `multimap<T>`
- Les clefs doivent **surcharger l'opérateur spécifié pour le tri**.
- L'opérateur par défaut pour le tri des clefs est l'opérateur `<`.
- Pour en spécifier un autre, il faut l'indiquer au constructeur :
 - `map<string, double, greater<string> >`; // `operator > (string,string)`
- En tout temps, les éléments d'un conteneur associatif trié sont ordonnés selon l'ordre spécifié par le constructeur.

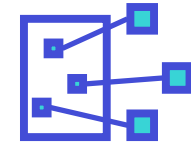


Conteneur *map* - exemple

```
#include <map>
map<string, float> coll;

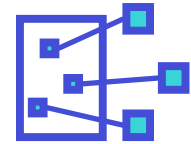
coll["VAT"] = 0.15;    // VAT est la clé
coll["Pi"] = 3.1416;
coll["an arbitrary number"] = 4983.223;
coll["Null"] = 0;

cout << coll["Pi"];
```



Conteneur *map* - exemple

- Soit un annuaire téléphonique qui stocke, pour un ensemble de personnes, leurs numéros de téléphone.
- Nous définirons une classe Annuaire qui permettra d'effectuer les opérations suivantes:
 - Ajouter un numéro
 - Chercher un numéro
 - Retirer un numéro
 - Afficher tous les numéros triés par nom
 - Afficher tous les numéros triés par numéro

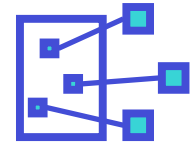


Conteneur *map* - exemple

```
class Annuaire
{
public:
    void ajouter_numero(string nom, int numero);
    void retirer_numero(string nom);
    int chercher_numero(string nom) const;
    void afficher_par_nom() const;
    void afficher_par_numero() const;
private:
    map< string, int > entrees_;
    typedef map<string,int>::const_iterator IterateurConst;
};
```

Nous aurons souvent besoin d'un itérateur constant, parce que les trois dernières méthodes sont déclarées const, et ne n'auront donc pas le droit de modifier l'état de l'objet.

Conteneur *map* – exemple (suite)



```
void Annuaire::ajouter_numero(string nom, int numero)
{
    entrees_[nom] = numero;
}
```

```
void Annuaire::retirer_numero(string nom)
{
```

```
    map< string, int >::iterator pos =
        entrees_.find(nom);
```

```
    if (pos != entrees_.end())
        entrees_.erase(pos);
```

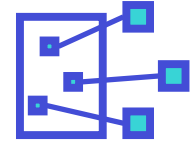
```
}
```

Ici on ne doit pas utiliser un itérateur constant, puisque l'objet pointé sera éliminé. L'état de l'annuaire sera donc modifié.

On recherche le nom dans le map.

Si l'item recherché se trouve dans le map, on l'élimine.

Conteneur *map* – exemple (suite)



```
int Annuaire::chercher_numero(string nom) const  
{
```

Itérateur constant utilisé
puisque la méthode est
déclarée const.

```
    IterateurConst pos = entrees_.find(nom);
```

```
    if (pos != entrees_.end())
```

```
        return pos->second;
```

```
    else
```

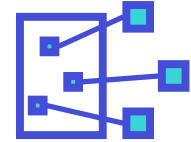
```
        return 0;
```

```
}
```

On recherche le
nom dans le
map.

Si l'item recherché se
trouve dans le map, on
retourne le numéro.

Conteneur *map* – exemple (suite)



```
void Annuaire::afficher_par_nom() const  
{
```

```
    IterateurConst pos = entrees_.begin();
```

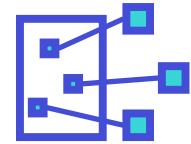
Par le biais de l'itérateur, on parcourt tous les items du map.

```
    while (pos != entrees_.end()) {  
        cout << "Nom:  " << pos->first  
        << "  Tel:  " << pos->second << endl;  
        ++pos;  
    }  
}
```

On affiche la clé (c'est-à-dire le nom dans ce cas-ci).

On affiche l'item stocké (le numéro dans ce cas-ci).

Conteneur *map* – exemple (suite)



```
void Annuaire::afficher_par_numero() const
{
    IterateurConst pos = entrees_.begin();

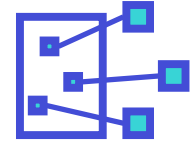
    map<int, string > entrees_inversees;

    while (pos != entrees_.end()) {
        entrees_inversees.insert(make_pair(pos->second, pos->first));
        ++pos;
    }

    map<int, string >::const_iterator pos_inv = entrees_inversees.begin();

    while (pos_inv != entrees_inversees.end()) {
        cout << "Tel:  " << pos_inv->first
        << "  Nom:  " << pos_inv->second << endl;
        ++pos_inv;
    }
}
```


Conteneur *map* – exemple (suite)



```
void Annuaire::afficher_par_numero() const
{
    IterateurConst pos = entrees_.begin();

    map<int, string > entrees_inversees;

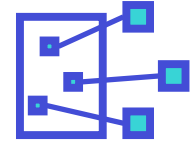
    while (pos != entrees_.end()) {
        entrees_inversees.insert(make_pair(pos->second, pos->first));
        ++pos;
    }

    map<int, string >::const_iterator pos_inv = entrees_inversees.begin();

    while (pos_inv != entrees_inversees.end()) {
        cout << "Tel:  " << pos_inv->first
        << "  Nom:  " << pos_inv->second << endl;
        ++pos_inv;
    }
}
```

Il nous faut un itérateur pour parcourir le map et un autre pour tester la fin du parcours.

Conteneur *map* – exemple (suite)



```
void Annuaire::afficher_par_numero() const
{
    IterateurConst pos = entrees_.begin();

    map<int, string > entrees_inversees;

    while (pos != entrees_.end()) {
        entrees_inversees.insert(make_pair(pos->second, pos->first));
        ++pos;
    }

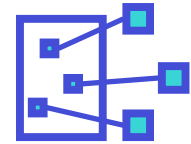
    map<int, string >::const_iterator pos_inv = entrees_inversees.begin();

    while (pos_inv != entrees_inversees.end()) {
        cout << "Tel:  " << pos_inv->first
              << "  Nom: " << pos_inv->second << endl;
        ++pos_inv;
    }
}
```

Il nous faut un autre map qui contient les même données, mais en utilisant le numéro comme clé.

Quelle hypothèse est-ce que nous faisons ici?

Conteneur *map* – exemple (suite)



```
void Annuaire::afficher_par_numero() const
{
```

```
    IterateurConst pos = entrees_.begin();
```

```
    map<int, string > entrees_inversees;
```

```
    while (pos != entrees_.end()) {
        entrees_inversees.insert(make_pair(pos->second, pos->first));
        ++pos;
    }
```

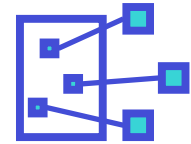
On remplit le nouveau map à partir des entrées courantes de l'annuaire.

À noter l'utilisation de la fonction `make_pair` qui construit une paire.

```
    map<int, string >::const_iterator pos_inv = entrees_inversees.begin();
```

```
    while (pos_inv != entrees_inversees.end()) {
        cout << "Tel:  " << pos_inv->first
        << "  Nom:  " << pos_inv->second << endl;
        ++pos_inv;
    }
}
```

Conteneur *map* – exemple (suite)



```
void Annuaire::afficher_par_numero() const
{
    IterateurConst pos = entrees_.begin();

    map<int, string > entrees_inversees;

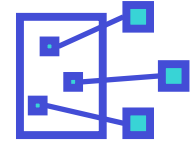
    while (pos != entrees_.end()) {
        entrees_inversees.insert(pair< int, string >(pos->second,pos->first));
        ++pos;
    }

    map<int, string >::const_iterator pos_inv = entrees_inversees.begin();

    while (pos_inv != entrees_inversees.end()) {
        cout << "Tel:  " << pos_inv->first
        << "  Nom:  " <<  pos_inv->second << endl;
        ++pos_inv;
    }
}
```

On pourrait aussi créer la paire en utilisant le constructeur.

Conteneur *map* – exemple (suite)



```
void Annuaire::afficher_par_numero() const
{
    IterateurConst pos = entrees_.begin();

    map<int, string > entrees_inversees;

    while (pos != entrees_.end()) {
        entrees_inversees.insert(pair< int, string >(pos->second,pos->first));
        ++pos;
    }
```

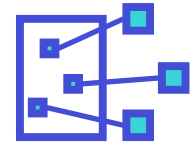
Encore une fois, il faut utiliser des itérateurs constants parce que la méthode n'a pas le droit de modifier l'objet.

```
map<int, string >::const_iterator pos_inv = entrees_inversees.begin();

while (pos_inv != entrees_inversees.end()) {
    cout << "Tel:  " << pos_inv->first
    << "  Nom:  " << pos_inv->second << endl;
    ++pos_inv;
}
```

Il nous faut deux nouveaux itérateurs pour parcourir le second map.

Conteneur *map* – exemple (suite)



```
void Annuaire::afficher_par_numero() const
{
    IterateurConst pos = entrees_.begin();

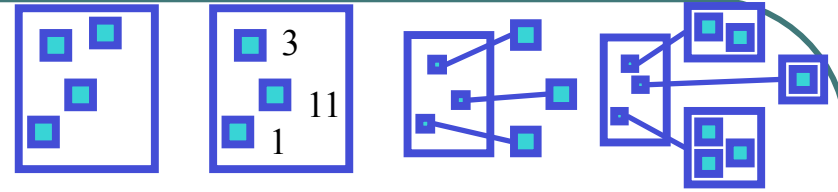
    map<int, string > entrees_inversees;

    while (pos != entrees_.end()) {
        entrees_inversees.insert(pair< int, string >(pos->second,pos->first));
        ++pos;
    }

    map<int, string >::const_iterator pos_inv = entrees_inversees.begin();

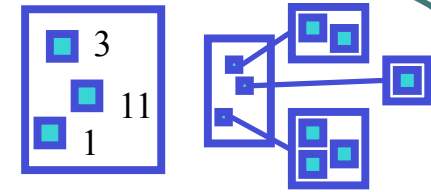
    while (pos_inv != entrees_inversees.end()) {
        cout << "Tel:  " << pos_inv->first
        << "  Nom:  " << pos_inv->second << endl;
        ++pos_inv;
    }
}
```

On affiche les items du second map.



Associatifs : Haché (Hashed)

- Tous les conteneurs associatifs préfixés de `hash_`
 - `hash_set<T>`, `hash_map<T>`, `hash_multiset<T>`, `hash_multimap<T>`
- Les clefs ne sont **pas triées**, mais réparties selon une table de hachage.
- Le nombre d'entrées dans la table et la fonction de hachage peuvent être spécifiés à la construction du conteneur :
 - ```
hash_set<char*> eleves(256, // nb entrées
 (buckets)
 hash<char*>); // fonction de hachage
```
- Ces conteneurs sont **très rapides pour l'insertion et la recherche d'éléments** (temps constant en moyenne).

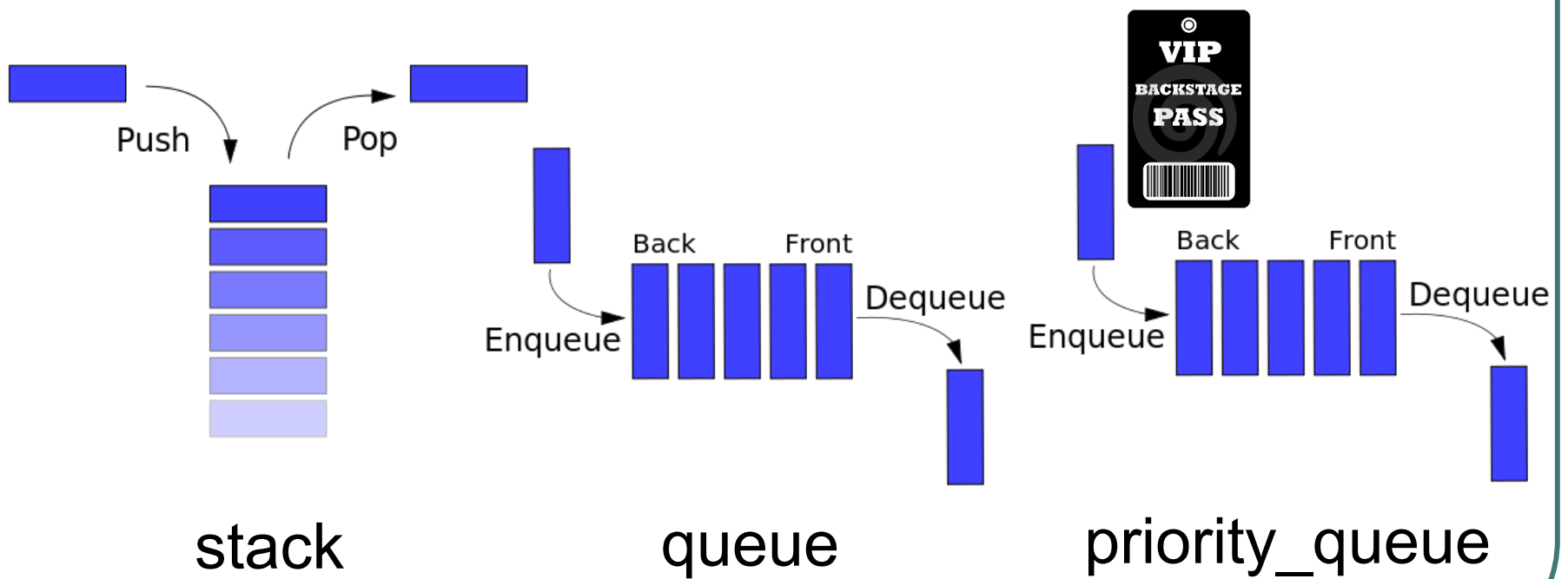


# Associatifs : Unique/Multiple

- Unique :
  - Tous les conteneurs associatifs **sans** le mot `multi`.
  - `count(T x)` retourne soit 1 ou 0.
  - `insert(T x)` retourne une paire `<iterator, bool>` où
    - `*iterator == x`; est toujours vrai.
    - `bool` indiquant si l'insertion a réussi (aucun élément égal à `x` avant l'insertion).
- Multiple :
  - Tous les conteneurs associatifs **avec** le mot `multi`.
  - `count(T x)` retourne un entier `>= 0`
  - `insert(T x)` retourne un itérateur pointant à l'endroit où `x` est placé.



# Les adaptateurs



# Adaptateurs

---

- **stack** : adaptateur qui agit à la façon d'une pile (LIFO) en définissant les méthodes appropriées
- **queue** : adaptateur qui agit à la façon d'une file (FIFO) en définissant les méthodes appropriées
- **priority\_queue** : adaptateur qui agit comme une file de priorité en triant ses éléments de sorte que les premiers retirés soient les plus prioritaires.

# Adaptateurs (suite)

---

- Les adaptateurs ne sont pas des conteneurs.
- Ils sont plutôt une **couche d'abstraction** qui permet d'utiliser les conteneurs séquentiels suivant un comportement spécifique :
  - **LIFO** (Last In First Out) : comme une pile d'assiettes .
  - **FIFO** (First In First Out) : comme une file d'attente à la caisse.
  - **File de priorité** : comme les files d'attente à La Ronde (flash).
- Ils renomment des méthodes pour que les méthodes soient plus adaptés à la situation :
  - Ex : `pop()` qui cache un `pop_back()` de `deque`
  - Ex : `push()` qui cache un `push_back()` sur un `vector`

# Adaptateurs (suite)

---

- Ils ont tous un conteneur par défaut sur lequel ils opèrent :
  - `deque<T>` pour `stack` et `queue`
  - `vector<T>` pour `priority_queue`
- Mais on peut changer ce conteneur à la construction :
  - Ex : `queue<int, list<int> > myQueue;`
- Peu importe le conteneur à la base de l'adaptateur, les méthodes restent les mêmes, **seul le temps d'exécution et l'espace mémoire peuvent changer.**

# Adaptateur *stack* de la STL

---

```
template <typename T, typename C = deque< T > >
class stack
{
public:
 int size() const;
 void push(const T& x);
 T& top();
 void pop();
protected:
 C items_;
};
```

On peut passer un type de conteneur. Par défaut, il s'agira d'un **deque**.

Conteneur générique.

# Adaptateurs

## (méthodes communes)

---

- Ces méthodes font référence au conteneur à la base et permettent de connaître le nombre d'éléments contenus :
  - `size()` : retourne le nombre d'éléments dans le conteneur
  - `empty()` : indique si le conteneur est vide
- Bien que `c.empty()` soit équivalent à `c.size() == 0`, `empty()` peut être beaucoup plus rapide, car elle est assurée d'être fait en temps constant tandis que `size()` peut être proportionnel à la taille
- Aucun adaptateur ne permet d'itération à travers ses éléments. L'idée est d'**interdire toutes manipulations dangereuses ou sans liens avec les fonctionnalités** de l'adaptateur.

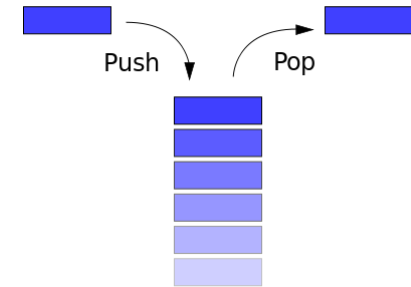
# Adaptateur *stack* de la STL

---

```
template <typename T, typename C = deque< T > >
void stack< T, C >::push(const T& x)
{
 items_.push_back(x);
}
```

La méthode **push\_back()**  
doit être définie pour le  
conteneur utilisé.

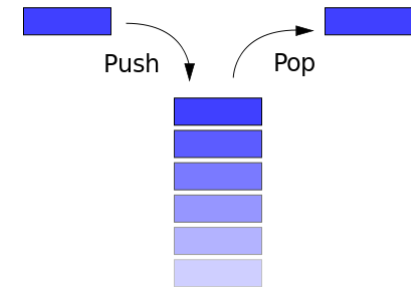
# Stack



- Pile basée sur le système LIFO (Last In First Out).
- Seul l'élément du dessus peut être manipulé.
- Son conteneur de base peut être `vector`, `deque` ou `list`
- Offre les méthodes suivantes :
  - `top()` : inspecter l'élément du dessus
  - `push()` : insérer un élément sur le dessus
  - `pop()` : retirer l'élément du dessus, mais pas le retourner (`void pop()`)



# Stack (exemple)



```
• int main()
{
```

```
 stack<double> unStack;
```

Déclaration du stack

```
 unStack.push(12.45);
 unStack.push(19.12);
 unStack.push(11.88);
 unStack.push(13.94);
```

Remplissage du stack

```
 while(!unStack.empty())
 {
 cout << unStack.top() << endl;
 unStack.pop();
 }
```

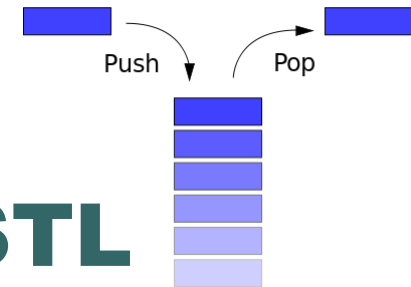
Affichage :

13.94  
11.88  
19.12  
12.45

```
 return 0;
```

```
}
```

# Adaptateur *stack* de la STL

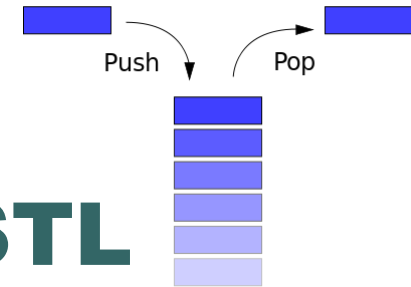


```
int main()
{
 stack< int, vector<int> > unePile;
 unePile.push(23);
 unePile.push(2);
 unePile.push(19);

 while (!unePile.empty()) {
 cout << unePile.top();
 unePile.pop();
 }
 return(0);
}
```

Si on n'inclut pas cet argument, le conteneur sera un **deque**.

# Adaptateur *stack* de la STL

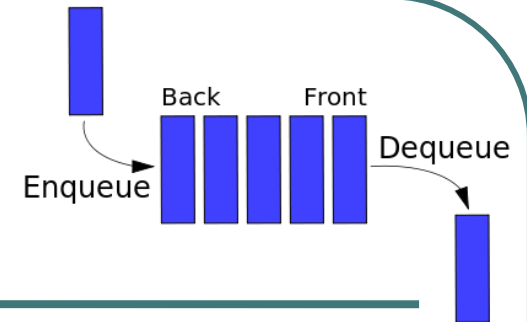


- Remarquez que le conteneur caché dans la pile est déclaré *protected*
- On peut donc facilement étendre la classe
- Voici comment on peut y ajouter une méthode pour vider la pile:

```
template <class T>
class Pile : public stack<T>
{
 public:
 void vider();
};

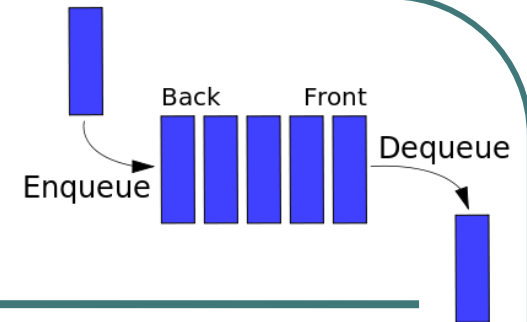
template <class T>
void Pile< T >::vider()
{
 items_.clear();
};
```

# Queue



- File basée sur le système FIFO (First In First Out)
- Les éléments sont ajoutés à la fin et retirés au début
- Son conteneur de base peut être `deque` ou `list`
- Offre les méthodes suivantes :
  - `front()` : inspecter l'élément du devant
  - `back()` : inspecter l'élément de l'arrière
  - `push()` : insérer un élément à la fin
  - `pop()` : retirer l'élément du début, mais pas le retourner (`void pop()`)

# Queue (exemple)



```
• int main()
{
```

```
 queue<double> uneQueue;
```

Déclaration de la queue

```
 uneQueue.push(12.45);
 uneQueue.push(19.12);
 uneQueue.push(11.88);
 uneQueue.push(13.94);
```

Remplissage de la queue

```
 while(!uneQueue.empty())
```

```
 {
```

```
 cout << uneQueue.front() << endl;
 uneQueue.pop();
```

```
 }
```

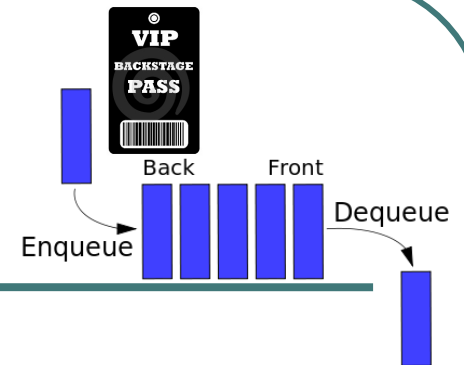
```
 return 0;
```

```
}
```

Affichage inversé par rapport à stack :

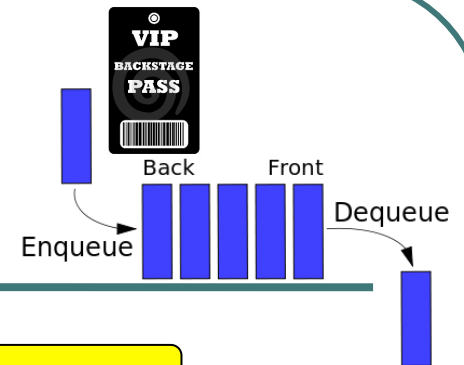
12.45  
19.12  
11.88  
13.94

# Priority\_queue



- File basée sur le système de priorité
- À l'interne, il s'agit d'un `vector<T>`
- Par défaut, les éléments sont triés avec l'opérateur<
- Il offre les méthodes suivantes :
  - `top()` : inspecter l'élément le plus prioritaire
  - `push()` : insérer un élément dans la file
  - `pop()` : retirer l'élément le plus prioritaire, mais pas le retourner (`void pop()`)

# Priority\_queue (exemple)



```
int main()
{
 priority_queue<double> unePQueue;

 unePQueue.push(12.45);
 unePQueue.push(19.12);
 unePQueue.push(11.88);
 unePQueue.push(13.94);

 while(!unePQueue.empty())
 {
 cout << unePQueue.top() << endl;
 unePQueue.pop();
 }

 return 0;
}
```

Déclaration de la queue

Remplissage de la queue

Affichage :

19.12  
13.94  
12.45  
11.88

# Adaptateurs (void pop())

---

- Certaines personnes se demandent pourquoi `pop()` ne retourne pas l'élément retiré :
  - C'est une question de performance :
    - L'objet ne peut pas être renvoyé par référence, car il est détruit dans le conteneur
    - Un renvoi par valeur s'impose avec le temps de copie que cela implique
    - Si en plus, l'objet renvoyé n'est pas utilisé, on peut gaspiller beaucoup de temps!
  - La boucle pour afficher et sortir les éléments du conteneur en deux temps dans les exemples :

```
while(!adaptateur.empty())
{
 cout << adaptateur.top() << endl;
 adaptateur.pop();
}
```