

## PARTIE VI

# Bibliothèque standard STL

Bruno Bachelet

Loïc Yon

- Historique
- Classes utilitaires
- Rappels
  - Espaces de nommage
  - Exceptions
- Principes généraux
  - Itérateurs
  - Foncteurs
- Conteneurs de la STL
  - Conteneurs de séquences
  - Conteneurs adaptateurs
  - Conteneurs associatifs

- Développement récurrent des mêmes composants
  - ❑ Structures de données: vecteur, pile, file, ensemble...
  - ❑ Algorithmes: chercher, trier, insérer, extraire...
  
- Éviter de réinventer la roue
  - ❑ Temps perdu (codage, débogage, optimisation)
  - ❑ Utiliser l'existant (bibliothèques)
  
- Tous les langages modernes ont une bibliothèque
  - ❑ Java, C#, Perl, Python
  - ❑ C++

- Concept de la généricité dès les années 70
- Alexander Stepanov
  - ❑ Premiers développements de la STL en 1979
  - ❑ Portage en ADA en 1987
  - ❑ Portage en C++ en 1992
- Normalisée en 1998
  - ❑ Avant: STL = *Standard Template Library*
  - ❑ Après: *Standard C++ Library*
  - ❑ Implication de HP puis SGI
  - ❑ Documentation: <http://martinbroadhurst.com/stl>

## ■ Chaîne de caractères

### □ Limite du `char *` en C

- Gestion de mémoire absente
- Source classique de fuites mémoires
- Pas de vrai type de données
- Pas de support d'opérations simples (e.g. concaténation)

### □ Classe `string`

- Gestion interne de la mémoire (forme normale de Coplien)
- Surcharge des opérateurs classiques (`+`, `<<`)
- Entête `<string>`

## ■ Exceptions

### □ Contrôle primitif des erreurs en C

- Utilisation du retour des fonctions
- Variable globale (`errno`)
- Prise en compte facultative de l'erreur

### □ Classe `exception`

- Bloc `try / catch`, mot-clé `throw`
- Gestion obligatoire
- Entête `<exception>`

# Rappels: espaces de nommage (1/3)

---

- En anglais: «*namespaces*»
- Permettent d'organiser les composants en modules
  - Mais leur fonction est très limitée
  - Déterminent simplement une zone avec un nom
  - Aucune règle d'accessibilité (privé, publique...)
- Evitent les collisions de nom
  - Exemple: `std::vector` ≠ `boost::mpl::vector` ≠ `boost::fusion::vector`
- Permettent de grouper des fonctions et des classes
  - Interface d'une classe = méthodes mais aussi fonctions
    - Les opérateurs externes notamment
  - Appel de fonction résolu selon le *namespace* des arguments
    - Dans le cas d'une fonction surchargée dans plusieurs *namespaces*

# Rappels: espaces de nommage (2/3)

---

- Mot-clé: `namespace`
  - `namespace monespace { /* Code */ }`
  - A rajouter sur tous les composants du module
  - Bien penser au `.hpp` et au `.cpp`

- Imbrication possible

```
namespace monespace {  
    void f(void);  
    ...  
    namespace monsousespace {  
        void g(void);  
        ...  
    }  
}
```



# Rappels: espaces de nommage (3/3)

---

- Utiliser un composant provenant d'un *namespace*
  - ❑ `monespace::f();`
  - ❑ `monespace::monsousespace::g();`
- Importer un symbole: **déclaration** «`using`»
  - ❑ `using std::vector;`
  - ❑ `vector<int> v;`
  - ❑ `std::string s;`
- Importer tous les symboles: **directive** «`using`»
  - ❑ `using namespace std;`
  - ❑ `vector<int> v;`
  - ❑ `string s;`
- Conseils pratiques
  - ❑ Ne jamais mettre d'importation dans un fichier entête (`.hpp`)
  - ❑ Préférer les déclarations aux directives dans un fichier d'implémentation (`.cpp`)
- Possibilité de créer des alias
  - ❑ `namespace fus = boost::fusion;`

- Pour gérer les erreurs: les «exceptions»
- Mécanisme qui permet de séparer
  - ❑ La détection d'une erreur
  - ❑ La prise en charge de l'erreur
- Exemple: code de calcul + interface graphique
  - ❑ Le code de calcul détecte des erreurs
  - ❑ L'interface graphique est informée et affiche un message dans une fenêtre
- Permet de conserver une modularité
- Exception = objet qui est créé lorsqu'une erreur survient

# Exceptions: transmission (2/6)

---

- Mot-clé «**throw**» dans une méthode
  - Au lieu de gérer l'erreur localement, l'erreur est transmise à la méthode appelante
  - On dit qu'une exception est «levée» / «lancée»
  - **if (erreur) throw std::string("oops !");**
  - Interruption de la suite normale du code
- L'objet transmis contient des renseignements sur l'erreur

# Exceptions: détection (3/6)

- Pour détecter une exception...
- Il faut surveiller
  - Bloc «**try**» définit une zone de surveillance
  - **try** {  
    // Code susceptible de lancer une exception  
}
  - **throw** ⇒ suspension de l'exécution normale
- Il faut rattraper et traiter les exceptions
  - Bloc «**catch**» décrit le traitement d'une exception
  - **catch(const exception & e) { /\* Gestion exception \*/ }**
  - Reprise de l'exécution suspendue par «**throw**»
- Plusieurs «**catch**» peuvent se succéder
  - Le premier qui correspond au type de l'erreur sera exécuté
  - Donc placement des «**catch**» du plus spécifique au moins spécifique
  - **catch(const MonException & e) { ... }**  
    **catch(const std::exception & e) { ... }**  
    **catch(...)**

# Exceptions: détection (4/6)

- Obligation de rattraper toutes les exceptions potentielles
  - Gestion immédiate: «**catch**» dans la méthode
  - Possibilité de «renvoyer» à la méthode appelante avec «**throw**»

- Exemple

```
void lectureFichier(const std::string & nom)
{ /* Lecture des données d'un fichier */ }

void traitement(void) {
    try {
        lectureFichier("mon_fichier.dat");
        // Code susceptible de lever un objet «exception»
    }

    catch(const ExceptionFichier & e)
    { std::cout << "Erreur ouverture fichier !" << std::endl; }

    catch(const std::exception & e)
    {std::cout << "Erreur dans les données !" << std::endl; }

    // Exécuté si aucune exception ou exception traitée
    std::cout << "Fin du traitement" << std::endl;
}
```

# Exceptions: classes standards (5/6)

---

- Si possible, utiliser une classe standard
  - ❑ `invalid_argument`, `out_of_range`, `overflow_error`...
  
- Sinon, créer ses classes d'exceptions
  - ❑ Spécialiser la classe de base `std::exception` ou une de ses sous-classes
  - ❑ Encapsuler des informations sur l'erreur
  - ❑ Redéfinir la méthode `what ( )` pour retourner un message décrivant l'erreur

- Toujours avoir un catch «universel»
  - ❑ `catch (...) { traitement }`
  - ❑ Permet de gérer les imprévus
  - ❑ Placé en général au niveau le plus haut
  - ❑ Dans la fonction «`main`» par exemple
  
- Fournir des garanties en cas d'exception
  - ❑ Que se passe-t-il en cas d'exception en plein milieu d'une série d'opérations ?
  - ❑ Garantir une certaine cohérence
    - ⇒ «*Exception safety*»

- Aucune garantie
  - ❑ Les données peuvent se retrouver dans un état incohérent
  - ❑ Fuite mémoire, crash possible
  
- Garantie «*no leak*»
  - ❑ Pas de fuite mémoire ou d'erreur de pointeur
  
- Garantie «*invariants preserved*»
  - ❑ Les données restent dans un état cohérent
  - ❑ Effet de bord possible
  
- Garantie «*no change*»
  - ❑ Les données conservent leurs valeurs originales
  - ❑ Pas d'effet de bord
  
- Garantie «*no throw*»
  - ❑ Toutes les opérations s'exécutent avec succès
  - ❑ Aucune exception ne sort de la méthode



- Exemple: opérateur d'affectation d'un vecteur d'entiers

- Aucune garantie

```
Vecteur & Vecteur::operator = (const Vecteur & v) {  
    if (this != &v) {  
        delete [] tab;  
        tab = new int[v.size];  
        size = v.size;  
        for (unsigned i = 0; i<size; ++i) tab[i] = v.tab[i];  
    }  
  
    return *this;  
}
```

- Rappel: «new» peut lever une exception

⇒ Incohérence: «tab» a été libéré et «size» n'est pas nulle

- Garantie «no change»

```
Vecteur & Vecteur::operator = (const Vecteur & v) {  
    if (this != &v) {  
        int * t = new int[v.size];  
        delete [] tab;  
        tab = t;  
        size = v.size;  
        for (unsigned i = 0; i<size; ++i) tab[i] = v.tab[i];  
    }  
  
    return *this;  
}
```

- «**this**» n'est pas modifié avant l'exception éventuelle

- Inconvénients

- ❑ Code plus compliqué
- ❑ Plus difficile de mutualiser du code avec le constructeur

- Autre solution: l'idiome «*copy-and-swap*»
  - `Vecteur & Vecteur::operator = (const Vecteur & v) {`  
    `Vecteur v2(v);`  
    `this->swap(v2);`  
    `return *this;`  
}
  - `void Vecteur::swap(Vecteur & v) {`  
    `std::swap(size, v.size);`  
    `std::swap(tab, v.tab);`  
}
- Exception levée lors de la copie  
⇒ «**v**» et «**this**» restent dans leur état initial
- **swap** ⇒ échange des contenus

- «*copy-and-swap*»  $\Rightarrow$  garantie «*no change*»
  - A condition que «**swap**» soit garanti «*no throw*»
- Avantage: réutilisation du code du constructeur de copie
- Attention aux performances de «**swap**» !
  - Tous les attributs ne seront pas des entiers ou des pointeurs
  - Utiliser «**std::swap**» pour les classes de la bibliothèque standard
  - Proposer un «**swap**» efficace pour vos classes
  - En C++11, utilisation des opérateurs «de mouvement»
- Possibilité d'une écriture encore plus compacte
  - Passage de l'argument par copie
  - ```
Vecteur & operator = (Vecteur v) {  
    this->swap(v);  
    return *this;  
}
```

- Séparation du conteneur et des algorithmes
  - ❑ Principe: «petit mais costaud»
    - Classes spécialisées
    - Uniquement les méthodes essentielles
- Stratégies d'accès / parcours aux conteneurs
  - ❑ Impossible de toutes les prévoir
  - ❑ Séparer ces stratégies et les conteneurs  $\Rightarrow$  itérateurs
- Algorithmes sur les conteneurs
  - ❑ Impossible de tous les prévoir
  - ❑ Séparer les algorithmes et les conteneurs
  - ❑ Algorithmes «à trous»  $\Rightarrow$  foncteurs

- Pour parcourir une collection: l'itérateur
  - ❑ Pointe sur un élément d'une collection
  - ❑ Permet de passer d'un élément à un autre
- Plusieurs itérateurs  $\Rightarrow$  parcours simultanés
- API indépendante de la véritable structure de données
- Différentes stratégies d'accès et de parcours
  - ❑ Accès en lecture ou lecture/écriture
  - ❑ Sens de parcours
- Implémentation d'un itérateur
  - ❑ Il doit souvent connaître l'implémentation de son conteneur
  - ❑ Deux possibilités
    - Classe amie
    - Classe imbriquée
  - ❑ Dans les 2 cas, il apparaît comme un type imbriqué

- Indépendants du conteneur sous-jacent
  - Il peut même ne pas y avoir de conteneur
    - Séquences générées à la volée, lecture/écriture dans un flux...
  - Utilisation homogène quelque soit le conteneur
- Permettent de représenter des sous-séquences
- Par rapport à un parcours avec index
  - Beaucoup plus efficace pour certaines structures de données
    - Exemples: liste, arbre
  - Différents types de parcours possibles sur une même séquence
    - Exemples: parcours préfixe, infixé et postfixé
  - Modification de la séquence en cours d'itération possible

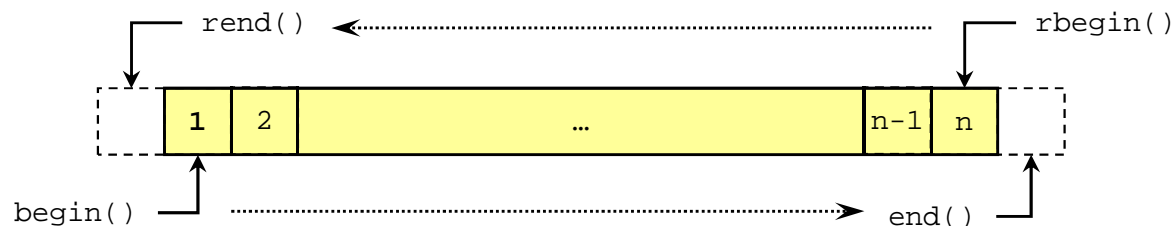
## ■ Fonctionnalités

- ❑ Forme normale de Coplien
  - Constructeur par défaut
  - Constructeur par copie
  - Opérateur d'affectation
  - Destructeur
- ❑ Opérateurs de comparaison **!=** et **==**
  - Attention: ne pas utiliser l'opérateur **<**
- ❑ Opérateur de déréférencement **\***
- ❑ Opérateurs d'incrémentation **++** (pré- et post-fixé)

- Manipulation identique à celle des pointeurs  
⇒ tableaux et conteneurs manipulables indifféremment



- 4 types d'itérateur par conteneur
  - Types imbriqués
    - `conteneur::iterator`
    - `conteneur::const_iterator`
    - `conteneur::reverse_iterator`
    - `conteneur::const_reverse_iterator`
  - **const** = accès en lecteur seule
  - **reverse** = parcours inversé, dernier → premier
- Balises fournies par le conteneur



- Parcours premier → dernier: **begin()**, **end()**
- Parcours dernier → premier: **rbegin()**, **rend()**

- Exemples d'utilisation

- Parcours d'un conteneur

- `Conteneur c;`

- ...

- `Conteneur::iterator it;`

- `for (it = c.begin(); it != c.end(); ++it)`  
`do_something(*it);`

- Valeur de retour de l'algorithme «`find`»

- Permet une opération immédiate sur l'objet

- Complexité de l'accès au suivant:  $O(1)$

- `Conteneur c;`

- ...

- `Conteneur::iterator it;`

- `it = find(c.begin(), c.end(), elt);`

- `do_something(*it);`

- Tous les itérateurs ne fournissent pas les mêmes fonctionnalités
  - de parcours
    - Exemple: impossible de reculer un itérateur sur une liste simplement chaînée
  - de manipulation de l'élément
    - Exemple: impossible de modifier un élément
- «Concepts» pour spécifier différents types d'itérateurs
- Important pour écrire des algorithmes
  - Documenter les fonctionnalités requises
  - Proposer des implémentations spécialisées pour certains itérateurs

# Concepts d'itérateurs (2/3)

---

## ■ `InputIterator`

- ❑ Accès à l'élément en lecture + avancée dans la séquence

## ■ `OutputIterator`

- ❑ Accès à l'élément en écriture + avancée dans la séquence

## ■ `ForwardIterator`

- ❑ `InputIterator` + `OutputIterator`

## ■ `BidirectionalIterator`

- ❑ `ForwardIterator` + recul dans la séquence

## ■ `RandomAccessIterator`

- ❑ `BidirectionalIterator` + «saut» dans la séquence

# Concepts d'itérateurs (3/3)

## ■ Exemple

```
template <typename InputIterator, typename OutputIterator>
OutputIterator copy(InputIterator first, InputIterator last,
                   OutputIterator result)
{
    while (first != last) *result++ = *first++;
    return result;
}
```

## ■ Exemple de spécialisation: `std::advance(it, n)`

- ❑ `it` doit modéliser `InputIterator`
- ❑ Si `it` modélise `BidirectionalIterator`, `n` peut être négatif
- ❑ Temps constant si `it` modélise `RandomAccessIterator`
  - `it += n;`
- ❑ Temps linéaire sinon
  - `if (n > 0)`  
    `while (n-- > 0) ++it;`

- Représentation d'une fonction par un objet
  - Permet l'écriture d'algorithmes «à trous»
  - A l'exécution, on passe un foncteur
  - Le foncteur comble les trous de l'algorithme
  
- Intérêts
  - Paramétrisation des algorithmes
    - D'autres solutions sont possibles (cf. design patterns)
  - Possibilité d'avoir un état interne
    - Attributs utiles pour mémoriser l'état et les paramètres
  
- Depuis C++11: Expressions lambdas  
⇒ Génération automatique de foncteurs

- Exemple: algorithme de tri

```
template <typename T>
void trier(vector<T> & v) {
    for (int i = 0; i < v.size()-1; ++i)
        for (int j = i+1; j < v.size(); ++j)
            if (v[j] < v[i])
                { T tmp = v[i]; v[i] = v[j]; v[j] = tmp; }
}
```

- Pas très flexible

- ❑ **T** doit implémenter l'opérateur <
- ❑ Comment faire un tri décroissant ?

- Solution: passer la relation d'ordre en paramètre

- Algorithme de tri

```
template <typename T,typename R>
void trier(vector<T> & v, const R & rel) {
    for (int i = 0; i<v.size()-1; ++i)
        for (int j = i+1; j<v.size(); ++j)
            if (rel.estAvant(v[j],v[i]))
                { T tmp = v[i]; v[i] = v[j]; v[j] = tmp; }
}
```

- Relation d'ordre

- ❑ `template <typename T> class Ordre {`  
    `public: bool estAvant(const T & a,const T & b)`  
    `const { return x; }`  
};
- ❑ Ordre croissant:  $x = a < b$
- ❑ Ordre décroissant:  $x = a > b$
- ❑ Exemple d'appel: `trier(v,Ordre<int>());`



- Foncteur = objet qui a l'apparence d'une fonction

⇒ Surcharge de l'opérateur ( )

- Opérateur ( )

- Arité spécifiée par le concepteur
  - Peut donc remplacer l'opérateur [ ] (e.g. une matrice)
- Syntaxe: *type\_retour* **operator**( ) (*paramètres*)

- Relation d'ordre

```
template <typename T> class Inferieur {  
    public: bool operator ( ) (const T & a,const T & b) const  
    { return (a<b); }  
};
```

- Algorithme de tri

```
template <typename T,typename R>  
void trier(vector<T> & v, const R & rel) {  
    for (int i = 0; i<v.size()-1; ++i)  
        for (int j = i+1; j<v.size(); ++j)  
            if (rel(v[j],v[i]))  
                { T tmp = v[i]; v[i] = v[j]; v[j] = tmp; }  
}
```

# Exemples de foncteurs (1/2)

---

## ■ Exemple: comparateur

### □ Principe

- Pas d'état interne
- Opérateur `()` prenant les deux objets à comparer

### □ Code foncteur

```
class Comparateur {  
    public:  
        bool operator() (const A & a1, const A & a2) const  
        { return (a1.val() < a2.val()); }  
};
```

### □ Code appel

```
Comparateur cmp;  
A a1, a2;  
std::cout << cmp(a1,a2) << std::endl;
```

# Exemples de foncteurs (2/2)

---

## ■ Exemple: générateur de nombres pairs

### □ Principe

- Etat interne conservé par les attributs
- Opérateur `()` sans paramètres pour la génération des nombres

### □ Code foncteur

```
class GenerateurPair {  
    protected: int val;  
    public:  
        GenerateurPair(void) : val(0) {}  
        int operator() (void) { val+=2; return val; }  
};
```

### □ Code appel

- `GenerateurPair gen;`
- `std::cout << gen() << ' ' << gen() << std::endl;`

- Classes de base (au sens propre)
  - Exposent les types des paramètres et de retour
  - `std::unary_function<Arg, Result>`
  - `std::binary_function<Arg1, Arg2, Result>`
  
- Foncteurs prédéfinis
  - Arithmétiques: addition, soustraction, multiplication, division...
    - `plus<T>`, `minus<T>`, `multiplies<T>`, `divides<T>`...
  - Comparaisons: inférieur, supérieur, égal...
    - `less<T>`, `less_equal<T>`, `equal_to<T>`...
  - Opérateurs logiques: et, ou, non
    - `logical_and<T>`, `logical_or<T>`...
  - Utilisent simplement les opérateurs correspondants

# Manipulation de foncteurs

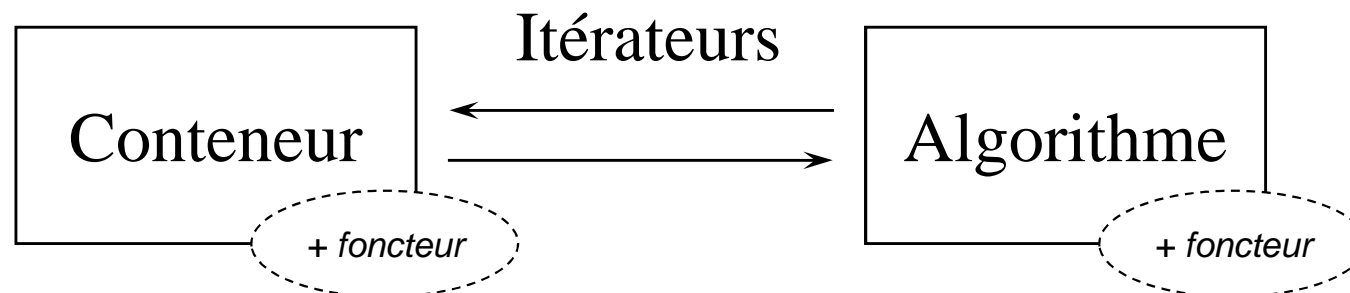
---

- Création de foncteur
  - A partir d'une fonction
    - `ptr_fun( pointeur fonction )`
  - A partir d'une méthode
    - Passage par pointeur de l'objet: `mem_fun( pointeur méthode )`
    - Passage par référence de l'objet: `mem_fun_ref( pointeur méthode )`
- Adaptation de foncteur
  - Négation: `not1`, `not2`
  - Fixation d'un paramètre: `bind1st`, `bind2nd`
- Exemple: compter les chaînes non vides dans un vecteur

```
vector<string> v;  
...  
int nbNonVides =  
    count_if(v.begin(), v.end(),  
             not1( mem_fun_ref(&string::empty) ) );
```
- Simplification avec les lambdas en C++11

# Séparation conteneur-algorithmes

- Manipulation globale du conteneur
  - Trois entités
    - Un conteneur pour le stockage des objets
    - Des itérateurs pour les accès aux objets
    - Des algorithmes pour la manipulation des objets
  - Fonctionnement conjoint
    - Les algorithmes opèrent sur le conteneur via les itérateurs



# Conteneurs de la STL (1/3)

---

- Trois grandes classes de conteneurs
  - Séquences élémentaires
    - Vecteur, liste, file à double entrée
  - Adaptations des séquences élémentaires
    - Pile, file, file à priorité
  - Conteneurs associatifs
    - Ensemble avec/sans unicité
    - Association avec clé unique/multiple
  
- Remarques
  - Tous définis dans le namespace «**std**»
  - Utilisation intensive de la généricité
    - Type de données
    - Allocateur de mémoire
    - Comparateur
    - ...

- Choix du conteneur
  - Selon les fonctionnalités disponibles
    - Un morceau d'API commun
    - Un morceau d'API spécifique à chaque conteneur
  - Selon la complexité des opérations
    - Opérations en  $O(1)$ ,  $O(\log n)$ ,  $O(n)$
  - Critères de choix
    - Chercher le conteneur le plus «naturel» pour l'algo voulu
    - Analyser la complexité du traitement
    - Chercher le conteneur offrant la meilleure complexité globale



## ■ Fonctionnalités communes

- ❑ Forme Normale de Coplien
- ❑ Dimensionnement automatique de la capacité

### ■ Exemple du vecteur

- ❑ Lorsque l'insertion d'un élément viole la capacité
- ❑ Augmentation de la capacité

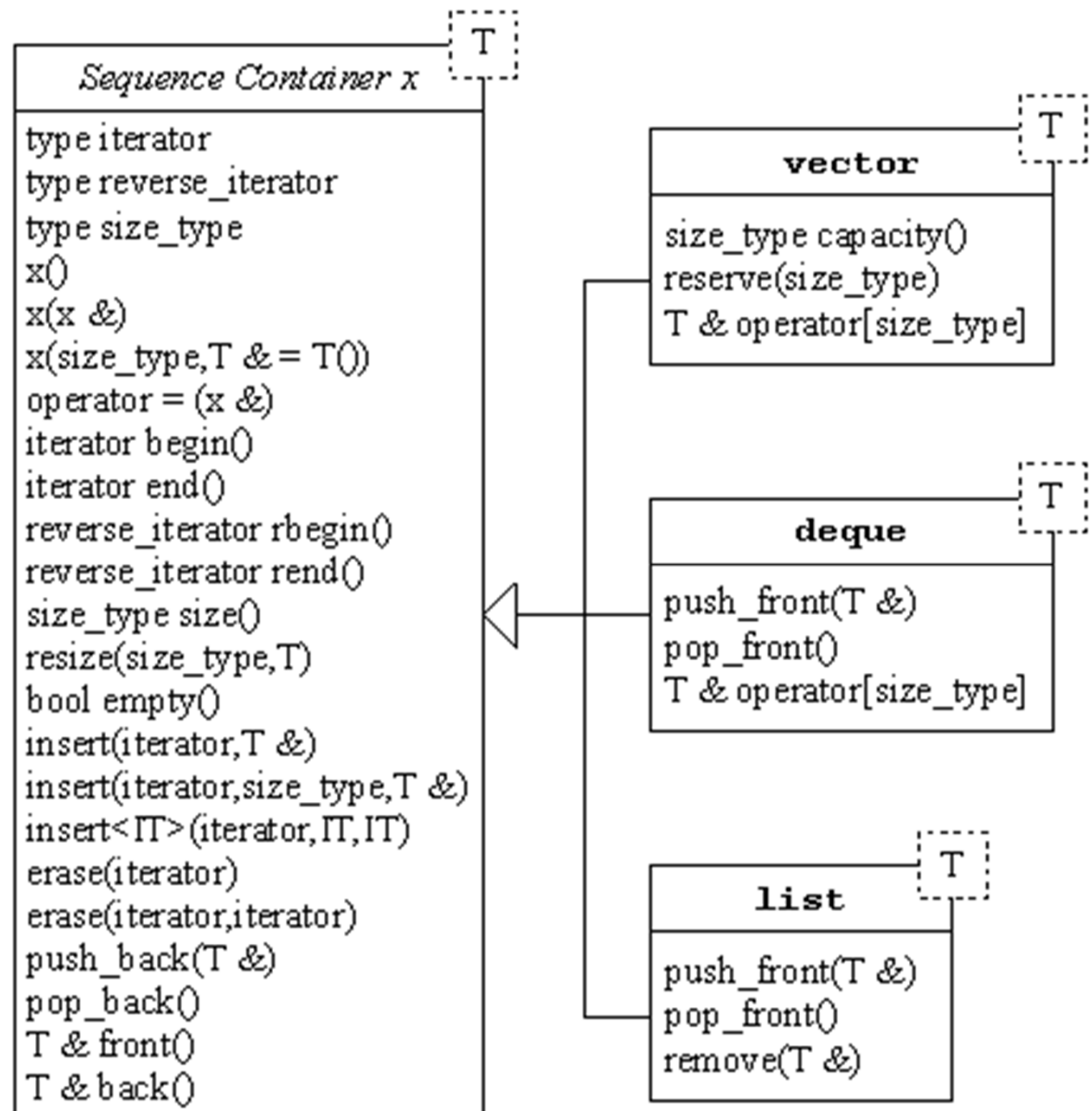
## ❑ Balises des itérateurs

## ❑ Quelques méthodes

- `size_t C::size() const` // Nombre d'éléments
- `size_t C::max_size() const` // Nombre max d'éléments
- `bool C::empty() const` // Est vide ?
- `void C::swap(C & cnt)` // Echange de contenu
- `void C::clear()` // Vide le conteneur

# Conteneurs en séquences (1/2)

- Vecteur (**vector**)
- Liste (**list**)
- File à double entrée (**deque**)



# Conteneurs en séquences (2/2)

## ■ Méthodes communes

### □ Insertion (avant la position indiquée)

- `void S::insert(S::iterator pos,T & elt)`
- `void S::insert(S::iterator pos,int nb,T & elt)`
- `template <typename InputIterator>`  
`void S::insert(S::iterator pos,`  
`InputIterator debut,InputIterator fin)`

### □ Suppression

- `S::iterator S::erase(S::iterator pos)`
- `S::iterator S::erase(S::iterator debut,S::iterator fin)`

### □ Accès / ajout en tête et fin

- `void S::push_back(const T & elt)`
- `void S::pop_back()`
- `T & S::front()`
- `const T & S::front() const`
- `T & S::back()`
- `const T & S::back() const`

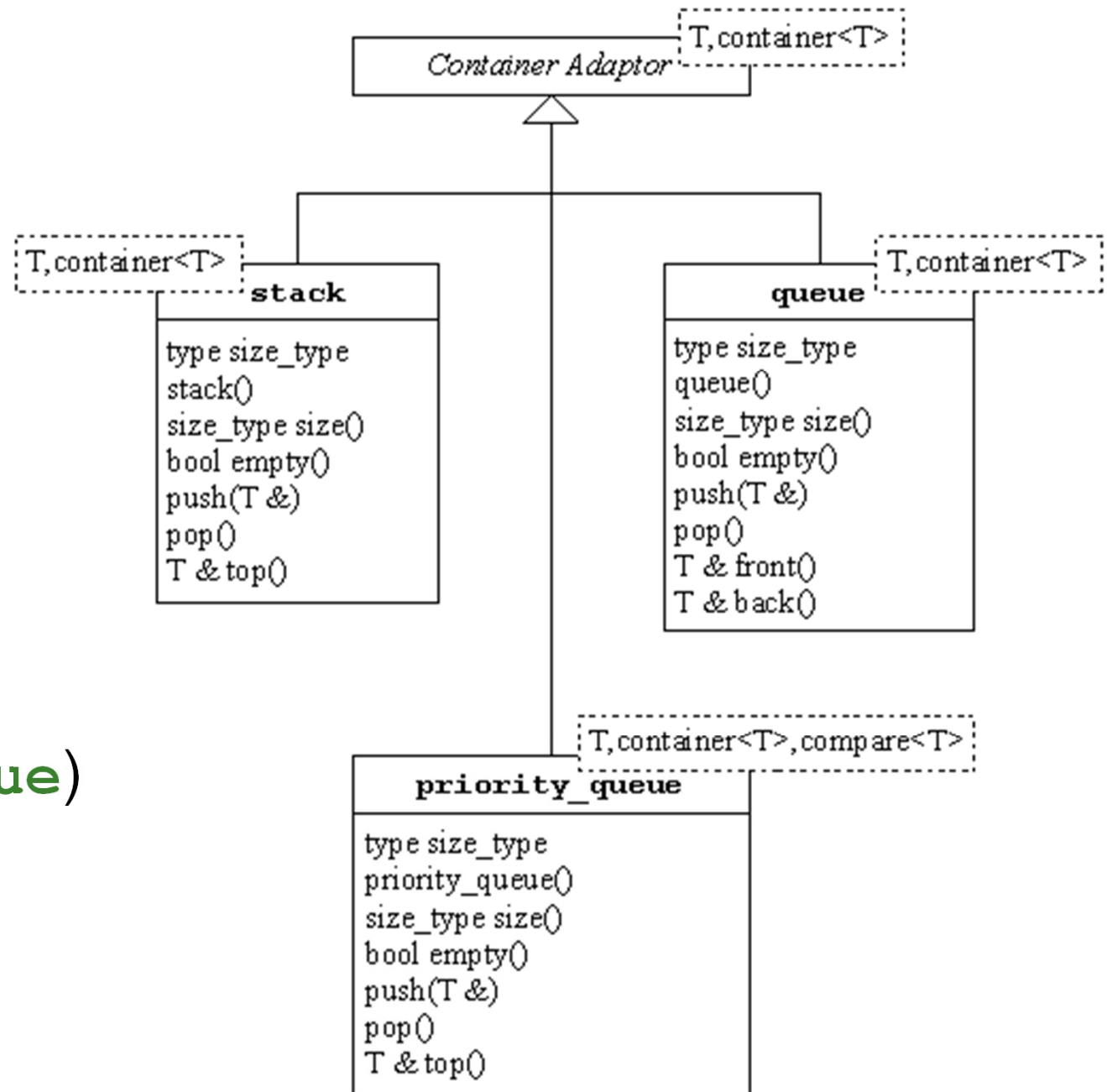
- Tableau qui se redimensionne automatiquement
  - Éléments contigus en mémoire (compatibilité avec les tableaux C)
- Efficacité
  - + Accès direct aux éléments (opérateur `[]`) en  $O(1)$
  - + Ajout / suppression en fin en  $O(1)$  (amorti)
  - Ajout / suppression ailleurs en  $O(n)$
- Utilisation
  - Entête: `<vector>`
  - Déclaration: `std::vector<T> v;`
  - Possède un autre paramètre template facultatif
    - Allocateur, gestionnaire de la mémoire interne
- Méthodes spécifiques
  - Contrôle capacité
    - `size_t V::capacity() const` // Capacité actuelle du vecteur
    - `void V::reserve(size_t nb)` // Ajustement de la capacité
  - Accès par index aux éléments
    - `X & V::operator[](size_t idx)` // Lecture/écriture
    - `const X & V::operator[](size_t idx) const` // Lecture seule

- Liste doublement chaînée
- Efficacité
  - + Ajout / suppression n'importe où en  $O(1)$
  - Pas d'accès direct aux éléments
- Utilisation
  - Entête: `<list>`
  - Déclaration : `std::list<T> l;`
  - Possède aussi un paramètre facultatif pour l'allocateur
- Méthodes spécifiques
  - Ajout / suppression en tête
    - `void L::push_front(const T & elt)`
    - `void L::pop_front()`
  - Suppression d'un élément
    - `void L::remove(const T & elt)`
  - Autres algorithmes spécifiques
    - `sort, merge, splice, remove_if, unique...`

- Similaire au vecteur sauf
  - ❑ Opérations en tête possibles
  - ❑ Contiguïté des éléments non garantie
- Efficacité
  - + Accès direct aux éléments (opérateur `[]`) en  $O(1)$
  - + Ajout/suppression en tête et fin en  $O(1)$  (amorti)
  - Ajout/suppression ailleurs en  $O(n)$
- Utilisation
  - ❑ Entête: `<deque>`
  - ❑ Déclaration: `std::deque<T> d;`
  - ❑ Possède aussi un paramètre facultatif pour l'allocateur
- Méthodes spécifiques
  - ❑ Pas de contrôle de capacité
  - ❑ Ajout / suppression en tête
    - `void D::push_front(const T & elt)`
    - `void D::pop_front()`
  - ❑ Accès par index aux éléments
    - `X & D::operator[](size_t idx)`
    - `const X & D::operator[](size_t idx) const`

# Conteneurs adapteurs (1/3)

- Pile  
(**stack**)
- File  
(**queue**)
- File à priorité  
(**priority\_queue**)

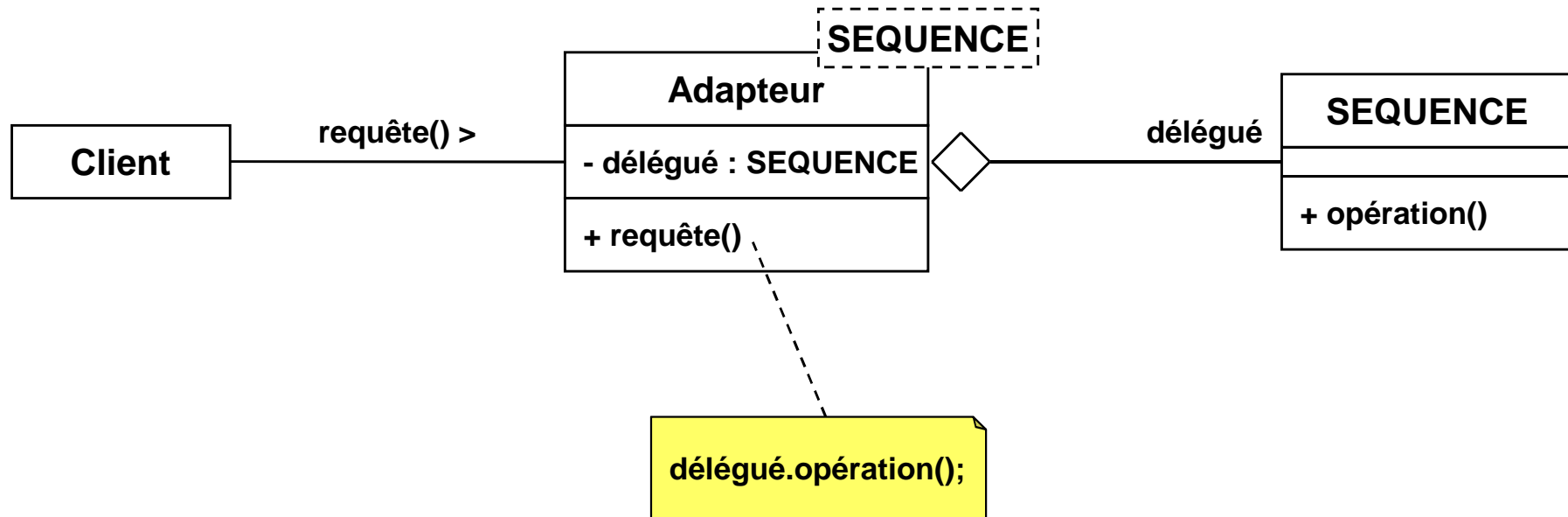


- Définis à partir d'un conteneur en séquence
  - ❑ Celui-ci est paramétrable
  - ❑ Utilise la structure de données du conteneur
  
- Propose une API spécifique
  - ❑ Celle-ci est réduite
  - ❑ Pas d'itérateurs
  
- Mécanisme de délégation
  - ❑ Agrégation du conteneur
  - ❑ Délégation des opérations au conteneur



# Conteneurs adapteurs (3/3)

## ■ Mécanisme de délégation



- Accès seulement au sommet de la pile
  - Pas de possibilité de voir les éléments empilés
- Utilisation
  - Entête: `<stack>`
  - Déclaration
    - `std::stack<T> s; // Conteneur par défaut = deque<T>`
    - `std::stack<T, std::vector<T> > s;`
- Méthodes spécifiques
  - Empilement / dépilement
    - `size_t S::push(const T & elt)`
    - `void S::pop()`
  - Accès au sommet
    - `T & S::top()`
    - `const T & S::top() const`
  - Comparaison de piles (car impossible de voir l'empilement)
    - `bool operator==(const stack<T> & s1, const stack<T> & s2)`
    - `bool operator<(const stack<T> & s1, const stack<T> & s2)`

- Structure FIFO (*First In First Out*)
  - ❑ Ajout en fin, retrait en tête
  - ❑ Pas de possibilité de voir les élément dans la file
- Utilisation
  - ❑ Entête: `<queue>`
  - ❑ Déclaration
    - `std::queue<T> q; // Conteneur par défaut = deque<T>`
    - `std::queue<T, std::list<T> > q;`
    - Ne peut pas utiliser `std::vector` (n'a pas `pop_front()`)
- Méthodes spécifiques
  - ❑ Ajout / retrait
    - `size_t Q::push(const T & elt)`
    - `void Q::pop()`
  - ❑ Accès aux extrémités
    - `front()`, `back()`
  - ❑ Comparaison de files
    - Opérateurs `==` et `<`

- File d'attente à priorité

- ❑ Ajout en fin, retrait de l'élément le plus «grand»  $\Rightarrow$  foncteur comparateur
- ❑ Pas de possibilité de voir les éléments dans la file

## ■ Utilisation

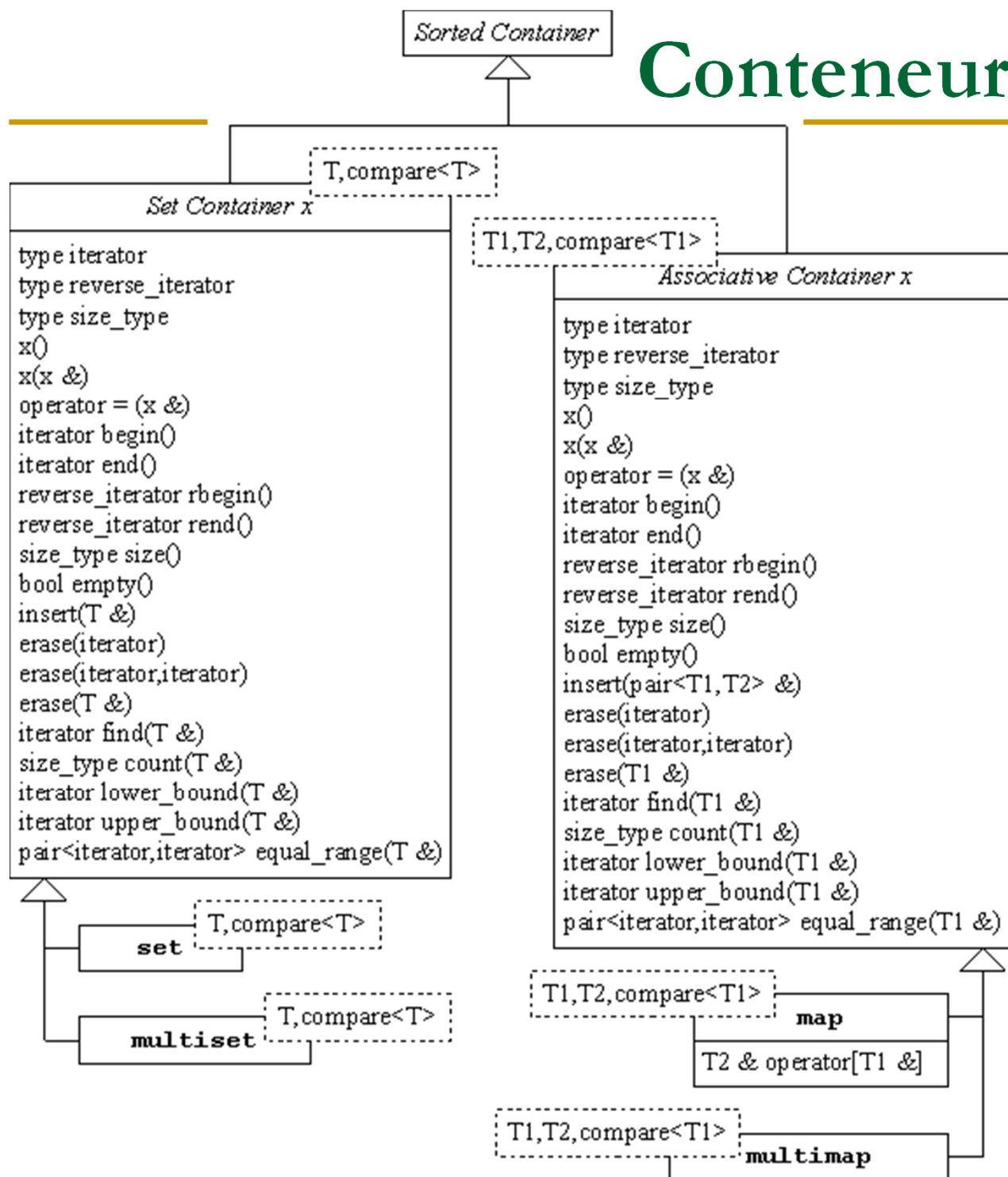
- ❑ Entête: `<queue>`
- ❑ Déclaration

- `std::priority_queue<T> p;` // Conteneur par défaut = `vector<T>`  
// Comparateur par défaut = `less<T>`
- `std::priority_queue<T,std::deque<T>,std::greater<T>> q;`
- Ne peut pas utiliser `std::list` (n'a pas `operator[]`)

## ■ Méthodes spécifiques

- ❑ Constructeur (qui attend un objet comparateur)
  - `P::P(Compareteur & c = Compareteur())`
- ❑ Ajout / retrait
  - `size_t P::push(const T & elt)`
  - `void P::pop()`
- ❑ Accès au plus grand
  - `T & P::top()`
  - `const T & P::top() const`

# Conteneurs associatifs (1/5)



- Ensemble avec unicité (**set**)
- Ensemble sans unicité (**multiset**)
- Association avec unicité (**map**)
- Association sans unicité (**multimap**)

- Principe de l'association
  - ❑ Associer une clé à chaque élément
  - ❑ On accède à l'élément par sa clé
- Structure utilisée pour l'association: `std::pair`

```
template <typename T1,typename T2>
struct pair {
    T1 first;
    T2 second;
    pair(void) {}

    pair (const T1 & t1,const T2 & t2)
        : first(t1), second(t2) {}
};
```

- Création d'une paire

- ❑ `p = pair<int,double>(13,27.14);`
- ❑ Obligé d'écrire les types paramètres de la paire

- Pour éviter d'écrire les types: `std::make_pair()`

```
template <typename T1, typename T2>
pair<T1,T2> make_pair(const T1 & cle,
                      const T2 & elt)
{ return pair<T1,T2>(cle,elt); }
```

- Utilise le polymorphisme statique

- ❑ `p = make_pair(13,27.14);`

# Conteneurs associatifs (2/5)

---

- Conteneurs associatifs triés sur la clé
  - Nécessitent une relation d'ordre sur les clés  
⇒ foncteur comparateur
  - Représentation interne typique: RB-tree
- Ensembles
  - `set` ou `multiset`
  - L'élément contient sa clé
- Associations
  - `map` ou `multimap`
  - Les éléments stockés sont des associations clé-valeur
  - `first` = clé
  - `second` = valeur associée
- Clé unique ou multiple ?
  - Unicité ⇒ `set` ou `map`
  - Multiplicité ⇒ `multiset` ou `multimap`



## ■ Attention !

- ❑ `set` et `multiset` possèdent un seul paramètre: `set<V>`
- ❑ `map` et `multimap` possèdent deux paramètres: `map<K,V>`
- ❑ Pour `set` et `multiset`, `T = V`
- ❑ Pour `map` et `multimap`, `T = pair<K,V>`

## ■ Méthodes communes

### ❑ Constructeurs

- `A::A(void)`
- `template <typename InputIterator>`  
`A::A(InputIterator deb, InputIterator fin)`
- Paramètre facultatif: le comparateur de clés

## ■ Méthodes communes

### □ Insertions

- `pair<A::iterator,bool> A::insert(const T & elt)`
- `A::iterator`  
`A::insert(A::iterator pos,const T & elt)`
- `template <typename InputIterator>`  
`void A::insert(InputIterator deb,`  
`inputIterator fin)`

### □ Suppressions

- `void A::erase(A::iterator pos)`
- `void A::erase(A::iterator deb,A::iterator fin)`
- `A::size_type A::erase(const A::key_type & cle)`

# Conteneurs associatifs (5/5)

---

## ■ Méthodes communes

### □ Accès aux éléments

- `A::size_type A::count(const A::key_type & cle) const`
  - Nombre d'éléments ayant la clé fournie
- `A::iterator A::find(const A::key_type & cle) const`
  - Itérateur sur le premier élément ayant la clé fournie ou `A::end()` sinon
- `A::iterator A::lower_bound(const A::key_type & cle) const`
  - Itérateur sur le 1<sup>er</sup> élément dont la clé n'est pas inférieure à celle fournie
- `A::iterator A::upper_bound(const A::key_type & cle) const`
  - Itérateur sur le 1<sup>er</sup> élément dont la clé est supérieure à celle fournie
- `pair<A::iterator, A::iterator> A::equal_range(const A::key_type & cle) const`
  - Fournit un encadrement des éléments ayant la clé fournie

- Conteneur trié d'éléments contenant leur propre clé
- Utilisation
  - Entêtes: `<set>` / `<multiset>`
  - Déclaration
    - `std::set<V> s; // Comparateur par défaut = less<V>`
    - `std::set<V,greater<V> > s;`
    - Possède aussi un paramètre facultatif pour l'allocateur
- Méthodes spécifiques
  - Insertion dans «`set`»
    - `pair<S::iterator,bool> S::insert(const V & elt)`
  - Insertion dans «`multiset`»
    - `M::iterator M::insert(const V & elt)`

## ■ Fonctions spécifiques

### □ Fonctions ensemblistes (entête `<algorithm>`)

- Entre deux ensembles  $[deb1, fin1)$  et  $[deb2, fin2)$
- Ensembles décrits par des itérateurs

```
template <typename InputIterator1, typename InputIterator2,  
          typename OutputIterator>  
OutputIterator set_union(InputIterator1 deb1,  
                        InputIterator1 fin1,  
                        InputIterator2 deb2,  
                        InputIterator2 fin2,  
                        OutputIterator res)
```

- `InputIterator1`: type des itérateurs du 1<sup>er</sup> ensemble
- `InputIterator2`: type des itérateurs du 2<sup>nd</sup> ensemble
- `OutputIterator`: type des itérateurs pour l'ensemble résultat

### ■ Exemples

- `bool includes(deb1, fin1, deb2, fin2)`
- `OutputIterator set_intersection(deb1, fin1, deb2, fin2, res)`
- `OutputIterator set_difference(deb1, fin1, deb2, fin2, res)`
- `OutputIterator set_symmetric_difference(deb1, fin1, deb2, fin2, res)`

- Conteneur trié d'éléments associés à une clé
- Utilisation
  - Entêtes: `<map>` / `<multimap>`
  - Déclaration
    - `std::map<K,V> s; // Comparateur par défaut = less<K>`
    - `std::map<K,V,greater<K> > s;`
    - Possède aussi un paramètre facultatif pour l'allocateur
- Méthodes spécifiques
  - `pair<M::iterator,bool>`  
`M::insert(const pair<K,V> &)`
  - `V & M::operator[](const K & cle)`

- Remarques sur l'opérateur `[ ]`
  - Permet un accès indexé similaire au vecteur
  - Index = clé
  - Complexité d'accès en  $O(\log n)$
  - Attention: si la clé n'existe pas dans le conteneur, elle est ajoutée et associée à l'élément par défaut (`v()`)
  - Il est conseillé d'utiliser l'opérateur `[ ]` pour
    - l'écriture (insertion)
    - la lecture dont on est sûr de l'existence de la clé
  - Si on n'est pas sûr de l'existence d'une clé
    - Appel préalable à `find()` ou `count()`
    - Utilisation des itérateurs pour parcourir

# Types de données internes

---

- Les conteneurs STL définissent des types internes
  - ❑ Embarqués dans les classes
- Pour tous les conteneurs
  - ❑ `C::value_type`: type des éléments stockés
    - Pour les associations: `pair<K,V>`
  - ❑ `C::reference`: type d'une référence sur un élément stocké
  - ❑ `C::const_reference`: type d'une référence constante sur un élément stocké
  - ❑ `C::size_type`: type d'entier utilisé pour compter les éléments
  - ❑ `C::iterator` et variations: types des itérateurs du conteneur
- Pour les conteneurs associatifs
  - ❑ `C::key_type`: type des clés
    - Pour les associations: `K`
    - Pour les ensembles: `V`
  - ❑ `C::data_type`: type des valeurs
  - ❑ `C::key_compare`: comparateur des clés



- Collection de fonctionnalités classiques
  - Copier
  - Chercher
  - Trier
  - Insérer, supprimer, modifier
  - Partitionner, fusionner
  - Réorganiser
  
- Remarques
  - Tous définis dans le namespace `std`
  - Définis indépendamment des conteneurs

- Manipulent des itérateurs
  - ❑ Générique: possibilité de passer n'importe quel itérateur/pointeur
  - ❑ Itérateur de début et de fin = séquence où lire les éléments
  - ❑ Parfois itérateur de sortie pour écrire le résultat
  
- Souvent paramétrés par une opération
  - ❑ Algorithme à trous
  - ❑ Générique: possibilité de passer un foncteur ou un pointeur de fonction
  - ❑ Comparateur, prédicat, générateur...

# Boucle «pour chaque»

---

- Applique une opération à chaque élément d'une séquence
- Paramétré par une opération unaire

- Exemple

```
void ajouterPrefix(std::string & adresse)
{
    adresse = "http://www." + adresse;
}
...
std::vector<std::string> adresses;
...
for_each(adresses.begin(), adresses.end(), ajouterPrefix);
```

- «Vraie» boucle **foreach** dans C++11

- Recherche d'un élément par valeur
  - `itérateur find(début, fin, valeur)`
- Recherche d'un élément ayant une propriété donnée
  - `itérateur find_if(début, fin, prédicat)`
- Comptage du nombre d'éléments
  - égaux à une valeur : `entier count(début, fin, valeur)`
  - ayant une propriété : `entier count_if(début, fin, prédicat)`
- Test d'égalité de deux séquences
  - `booléen equal(début1, fin1, début2, fin2)`

## ■ Copie de tous les éléments

- `itérateur copy(début, fin, résultat)`

## ■ Copie des éléments ayant une certaine propriété

- `copy_if` oublié dans C++03 !
- Présent dans C++11
- Équivalence: `remove_copy_if(... not1(predicat));`

## ■ Exemple

```
vector<int> v = ...;
int buffer[5];
list<int> l;
copy(v.begin(), v.end(), buffer);
copy(buffer, buffer + 5, back_inserter(l));
copy(l.begin(), l.end(), ostream_iterator<int>(cout, " "));
```

- «Suppression» d'une valeur
  - `itérateur remove(début, fin, valeur)`
- «Suppression» des éléments ayant une certaine propriété
  - `itérateur remove_if(début, fin, prédicat)`
- Attention ! `remove` déplace seulement à la fin de la séquence
  - Retourne un itérateur sur la «nouvelle fin»
  - Coupler avec `Conteneur::erase` pour supprimer vraiment
  - Exemple

```
v.erase(remove(v.begin(), v.end(), 42), v.end() );
```
- Versions non modifiantes
  - `itérateur remove_copy(début, fin, résultat, valeur)`
  - `itérateur remove_copy_if(début, fin, résultat, prédicat)`

- Remplacement d'une valeur
  - `void replace(début, fin, ancienneValeur, nouvelleValeur)`
- Remplacement des éléments ayant une certaine propriété
  - `void replace_if(début, fin, prédicat, nouvelleValeur)`
- Versions non modifiantes
  - `itérateur replace_copy(début, fin, résultat, ancienneValeur, nouvelleValeur)`
  - `itérateur replace_copy_if(début, fin, résultat, prédicat, nouvelleValeur)`

- Appliquer une opération à chaque élément et stocker le résultat dans une autre séquence
  - `itérateur transform(début, fin, résultat, opérationUnaire)`
  - Exemple

```
list<string> adressesModifiees;  
transform(adresses.begin(), adresses.end(),  
          back_inserter(adressesModifiees),  
          ajouterPrefixe);
```
- Appliquer une opération binaire sur les éléments de deux séquences (deux-à-deux)
  - `itérateur transform(début1, fin1,  
 début2,  
 résultat,  
 opérationBinaire)`



- Tri avec opérateur <
  - `void sort(début, fin)`
- Tri avec comparateur personnalisé
  - `void sort(début, fin, comparateur)`
- Tri stable
  - Conserve l'ordre des éléments équivalents
  - `void stable_sort(début, fin[, comparateur])`
- Mélange
  - Par défaut, utilise `rand()`
  - `void random_shuffle (début, fin[, générateurAléatoire])`

## ■ Inversion

- ❑ `void reverse(début, fin)`
- ❑ `void reverse_copy(début, fin, résultat)`

## ■ Rotation

- ❑ Déplace les éléments de façon à ce que milieu soit au début
- ❑ `void rotate(début, milieu, fin)`
- ❑ `void rotate_copy(début, milieu, fin, résultat)`

# Remplissage d'une séquence

---

- Valeur fixe = passage d'une valeur
- Valeur variable = passage d'un générateur
  - Pointeur de fonction ou foncteur
- Début et fin fournis...
  - Remplissage de toute la séquence
  - `void fill(début, fin, valeur)`
  - `void generate(début, fin, générateur)`
- ...ou nombre d'éléments fournis
  - `void fill_n(résultat, n, valeur)`
  - `void generate_n(résultat, n, générateur)`

- Minimum / Maximum
  - Entre deux valeurs
    - `min/max(a, b)`
  - D'une séquence
    - `itérateur min/max_element(début, fin[, compareur])`
  
- Échange de deux valeurs
  - `void swap(a, b)`
  - Par défaut, copie dans un temporaire
  - Peut/doit être spécialisé pour être plus efficace
    - Par exemple, échange d'un attribut pointeur pour éviter la copie des éléments pointés
  - Très utilisé
    - Dans les algorithmes
    - Pour «compresser» un conteneur STL
      - La méthode `clear()` ne réduit pas la capacité du conteneur
    - Pour implémenter l'opérateur `=` (*copy-and-swap*)
      - Garantir une copie «sécurisée»

# Quelques concepts liés à la STL

---

## ■ Itérateur

- ❑ «Pointeur» sur un élément d'un conteneur
- ❑ Permet la séparation conteneur-algorithmes

## ■ Foncteur

- ❑ «Fonction» représentée sous la forme d'un objet
- ❑ Sa classe surcharge l'opérateur ( )
- ❑ Permet de paramétrer les algorithmes

## ■ Allocateur

- ❑ Objet chargé de la gestion de la mémoire dans un conteneur
- ❑ Permet d'adapter l'allocation mémoire
  - Optimisation: «pools» d'objets
  - Sécurité: multithreading
  - Support: mémoire en fichier

## ■ Traits

- ❑ API commune qui permet de connaître les caractéristiques d'un conteneur
  - Exemple: les types internes
- ❑ Permet aux conteneurs d'être interchangeables

## ■ Avantages

- ❑ Ensemble de fonctionnalités courantes
- ❑ Code performant et fiable

## ■ Inconvénients

- ❑ Ceux des codes génériques
  - Code instancié plusieurs fois
  - Peu de vérification préalable sur les types paramètres  
⇒ erreurs de compilation difficiles à déchiffrer
- ❑ Peu de vérification de cohérence
  - Exemple: débordement des itérateurs

## ■ Evolution

- ❑ Standard C++11
- ❑ Intégration de certaines bibliothèques Boost
  - *threads*, *regex*, *RNG*, tables de hachage...