Isima

# PARTIE IX

# Bibliothèque standard

Bruno Bachelet

Loïc Yon

# Plan

- Généralités
  - Historique
  - Espaces de nommage
- Grands principes
  - Séparation conteneurs-algorithmes
  - Itérateurs
  - Foncteurs / Lambdas
- Conteneurs
  - Conteneurs de séquences
  - Conteneurs adapteurs
  - Conteneurs associatifs
- Algorithmes

# Historique

- Concept de généricité introduit dès les années 70

  ⇒ Développement de structures de données
  et d'algorithmes génériques

- *Standard Template Library*
  - Travaux d'Alexander Stepanov
  - Premiers développements en 1979
  - Portage en ADA en 1987
  - Portage en C++ en 1992

- Normalisée en 1994, puis intégrée à la norme C++98

- STL fait partie de la bibliothèque standard du C++
  - Concerne la partie conteneurs (structures de données) et algorithmes

# Espaces de nommage *(1/4)*

- *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, public…)

- Evitent les collisions de nom
  - `std::vector` ≠ `boost::mpl::vector` ≠ `boost::fusion::vector`

- Permettent de regrouper des fonctions et des classes
  - Interface d'une classe = méthodes mais aussi fonctions
    - Les opérateurs binaires (externes) notamment
  - Résolution de la surcharge d'une fonction
    - Les surcharges dans les *namespaces* des arguments sont considérées

# Espaces de nommage *(2/4)*

- Mot-clé «`namespace`» ⇒ délimite un bloc

```
namespace monespace {
 class A { … };
 void f();
 using t = …;
}
```

- Tous les composants à l'intérieur du bloc sont préfixés
  - `monespace::A`, `monespace::f`, `monespace::t`…

- Peut être ouvert autant de fois que nécessaire

```
namespace monespace { class A; }
…
namespace monespace { void f(); }
```

- S'utilise aussi bien dans «`.hpp`» que dans «`.cpp`»
  - Une déclaration et sa définition doivent être dans le même *namespace*

# Espaces de nommage *(3/4)*

- Imbrication de *namespaces* possible

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

- Utiliser un composant provenant d'un *namespace*
  - `monespace::f();`
  - `monespace::monsousespace::g();`
- Préfixe « `::` » seul ⇒ référence au *namespace* global
- Possibilité de créer des alias
  - `namespace fus = boost::fusion;`

# Espaces de nommage *(4/4)*

- Possibilité d'«importer» des symboles
  - Pour éviter d'écrire le préfixe

- Importer un symbole: **déclaration** «`using`»
  - `using std::vector;`
  - `vector<int> v; // Utilisation implicite de «std::vector»`
  - `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`)

# Principes de la STL

- «Petit mais costaud»
  - Fournir des classes compactes
  - Spécialisées / centrées autour d'une fonctionnalité
  - Avec uniquement les méthodes essentielles
- Séparation des conteneurs et des algorithmes
  - Impossible de prévoir tous les algorithmes d'un conteneur

  ⇒ Algorithmes définis à part des conteneurs
- Stratégies d'accès / parcours des conteneurs
  - Pourquoi lier un algorithme à une stratégie de parcours ?
  - Pourquoi lier un algorithme à un conteneur spécifique ?

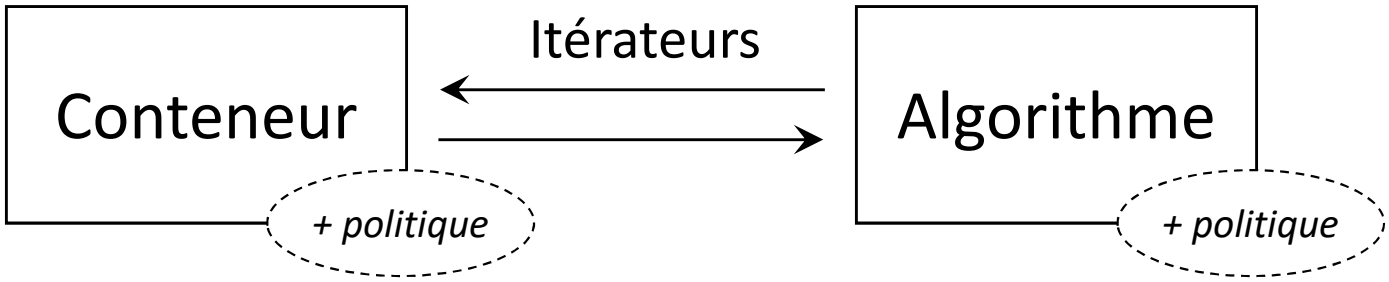
  ⇒ Abstraction du conteneur et de la stratégie de parcours: les «itérateurs»
- Algorithmes «génériques»
  - Pouvoir utiliser un algorithme dans un maximum de situations (e.g. tri)

  ⇒ Algorithmes «à trous» via les «politiques» (e.g. foncteurs, lambdas)

# Interaction conteneur-algorithmes

- En général, trois entités nécessaires pour manipuler un conteneur
  - Un conteneur pour le stockage des objets
  - Des itérateurs pour les accès aux objets
  - Des algorithmes pour la manipulation des objets
  - Optionnel: politiques pour paramétrer les algorithmes et/ou les conteneurs

- Fonctionnement conjoint conteneur-algorithmes
  - Les algorithmes opèrent sur le conteneur via des itérateurs



# Itérateurs *(1/2)*

- Parcourir un conteneur ⇒ un intermédiaire
  - Permet des parcours simultanés
  - Permet de parcourir une sous-partie du conteneur
  - Permet de faire abstraction du conteneur
  - Permet différentes stratégies d'accès (lecture/écriture) et de parcours (sens)
- Un itérateur est un objet
  - Qui pointe sur un élément d'un conteneur
  - Qui permet de passer d'un élément à un autre dans le conteneur
- API indépendante de la véritable structure de données
- Il s'agit d'un *design pattern* commun

# Itérateurs *(2/2)*

- Rend un algorithme indépendant du conteneur sous-jacent
  - Manipulation homogène de l'itérateur quel que soit le conteneur
  - Il peut même ne pas y avoir de conteneur derrière un itérateur
    - Séquences générées à la volée, lecture/écriture dans un flux...
- Par rapport à un parcours avec indice
  - Beaucoup plus efficace pour certaines structures de données
    - Exemples: liste, arbre
  - Différents types de parcours possibles sur une même séquence
    - Exemples: parcours préfixe, infixe et postfixe
  - Modification de la séquence en cours d'itération possible
- Implémentation d'un itérateur
  - Il doit souvent connaître l'implémentation de son conteneur
  - Deux possibilités: classe amie ou classe imbriquée
  - Le conteneur doit fournir des méthodes pour produire des itérateurs

# Itérateurs C++ *(1/3)*

- Interface d'un itérateur en C++
  - Forme normale de Coplien
    - Constructeur par défaut
    - Constructeur par copie
    - Opérateur d'affectation
    - Destructeur
  - API et sémantique du pointeur (arithmétique partielle)
    - Opérateurs de comparaison «!=» et «==»
      - Attention: ne pas utiliser l'opérateur «<»
    - Opérateur de déréférencement «*»
    - Opérateurs d'incrémentation «++» (préfixé et postfixé)

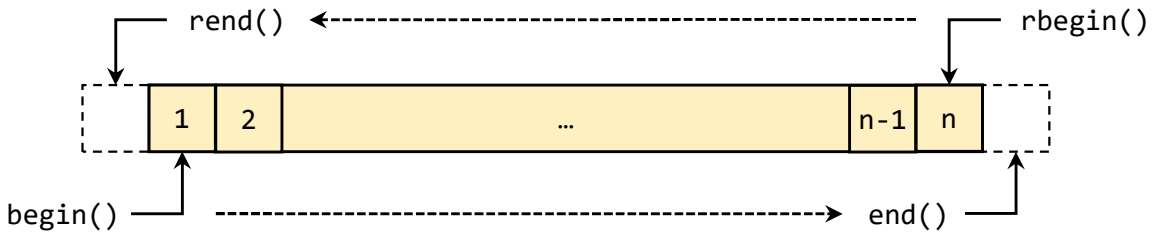
- Manipulation similaire à celle des pointeurs

  ⇒ tableaux et conteneurs STL manipulables indifféremment

# Itérateurs C++ *(2/3)*

- 4 types d'itérateurs associés à chaque conteneur
  - Types imbriqués
    - *type_conteneur*`::iterator`
    - *type_conteneur*`::const_iterator`
    - *type_conteneur*`::reverse_iterator`
    - *type_conteneur*`::const_reverse_iterator`
  - `const` = accès en lecteur seule
  - `reverse` = parcours inversé (dernier → premier)

- «Balises» (itérateurs repères) fournies par le conteneur




  - Du premier au dernier: *conteneur*`.begin()` → *conteneur*`.end()`
  - Du dernier au premier: *conteneur*`.rbegin()` → *conteneur*`.rend()`

# Itérateurs C++ *(3/3)*

- Parcours d'un conteneur à l'aide d'itérateurs
  - ```
    conteneur_t c;
    …
    conteneur_t::iterator it; // Accès avec écriture

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

- L'algorithme «`find`» retourne un itérateur sur l'élément trouvé
  - Sinon retourne la balise de fin du conteneur
  - Permet une opération immédiate sur l'objet
  - Complexité de l'accès au suivant: *O(1)*
  - ```
    conteneur_t c;
    …
    conteneur_t::iterator it;

    it = std::find(c.begin(),c.end(),elt);
    if (it != c.end()) do_something(*it);
    ```

# Boucle «*for*» simplifiée *(1/2)*

- Depuis C++11, syntaxe simplifiée pour le parcours de collections
  - Pour les tableaux de taille fixe (i.e. taille connue à la compilation)
  - Pour les conteneurs standards (ou tout conteneur respectant l'API)

- `for (`*élément* `:` *conteneur*`)`
  - *élément* = variable qui représente l'élément parcouru à chaque itération

- Exemple pour les tableaux
  - `float t[10];`
  - `for (float & v : t) v *= 2; // Accès avec écriture`
  - `for (float v : t) std::cout << v << " "; // Accès avec lecture seule`

- Code équivalent à une boucle avec indices
  - `for (unsigned i = 0; i < 10; ++i) t[i] *= 2;`
  - `for (unsigned i = 0; i < 10; ++i) std::cout << t[i] << " ";`

# Boucle «*for*» simplifiée *(2/2)*

- Exemple pour les conteneurs standards
  - `std::list<Point> points;`
  - `for (Point & p : points) p.x += dx;`
  - `for (const Point & p : points) std::cout << p.x << " ";`
- Code équivalent à une boucle avec itérateurs
  - ```
    std::list<Point>::iterator it = points.begin();
    while (it != points.end()) {
     (*it).x += dx; // Accès avec écriture
     ++it;
    }
    ```
  - ```
    std::list<Point>::const_iterator it = points.begin();
    while (it != points.end()) {
     std::cout << (*it).x << " "; // Accès lecture seule
     ++it;
    }
    ```
- S'applique à toute classe disposant de l'API des itérateurs

# Concepts d'itérateurs *(1/3)*

- 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
- Importants pour écrire des algorithmes génériques
  - Documenter les fonctionnalités requises par les itérateurs
  - 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
- `BidirectionalIterator`
  - `InputIterator` + recul dans la séquence
- `RandomAccessIterator`
  - `BidirectionalIterator` + «saut» dans la séquence
- Concepts formalisés en C++20 (cf. `Legacy*Iterator`)
  - Liste complète: `http://en.cppreference.com/w/cpp/iterator`

# Concepts d'itérateurs *(3/3)*

- Avant C++20, concepts implicites dans les noms des paramètres

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

- Spécialisation en fonction du concept: `std::advance(it,n)`
  - Contrainte: «`it`» doit modéliser «`InputIterator`»
  - Si «`it`» modélise «`BidirectionalIterator`» ⇒ autoriser «`n`» négatif
  - Si «`it`» modélise «`RandomAccessIterator`»
    - Implémentation en temps constant *O(1)*: `it += n;`
  - Sinon
    - Implémentation en temps linéaire *O(n)*: `if (n > 0) while (n-- > 0) ++it;`

# Politiques *(1/3)*

- Politique = fonction représentée sous la forme d'un objet
  - Permet l'écriture d'algorithmes «à trous»
  - A l'exécution, on passe une politique à l'algorithme
  - La politique comble les trous de l'algorithme
- Il s'agit du design pattern «stratégie»
- Intérêts
  - Paramétrisation des algorithmes
  - Possibilité d'avoir un état interne (via les attributs)
- La bibliothèque standard privilégie les «foncteurs»
  - Foncteur = politique qui a l'apparence d'une fonction
    ⇒ fonctions et foncteurs manipulables indifféremment
- Depuis C++11: expressions lambdas
  ⇒ génération implicite de foncteurs

# Politiques *(2/3)*

- 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]) std::swap(v[i],v[j]);
}
```

- Pas très flexible
  - «T» doit implémenter l'opérateur «<»
  - Comment faire un tri décroissant ?

# Politiques *(3/3)*

- Solution: passer la relation d'ordre en paramètre sous forme d'objet

```
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])) std::swap(v[i],v[j]);
}
```

- Implémentation d'une politique (relation d'ordre)
  - ```
    template <typename T> class OrdreCroissant {
     public: bool estAvant(const T & a, const T & b) const { return a < b; }
    };
    ```
  - ```
    template <typename T> class OrdreDecroissant {
     public: bool estAvant(const T & a, const T & b) const { return a > b; }
    };
    ```
  - Exemple d'appel: `trier(v,OrdreCroissant<int>());`

# Foncteurs *(1/2)*

- Foncteur = politique qui a l'apparence d'une fonction
  - Surcharge de l'opérateur «()»
  - Arité en fonction du besoin (e.g. relation d'ordre ⇒ 2 arguments)
  - Syntaxe: *type_retour* `operator()(`*arguments*`)`

- Exemple: relation d'ordre

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

- Exemple: 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])) std::swap(v[i],v[j]);
}
```

# Foncteurs *(2/2)*

- Objet ⇒ possibilité d'un état interne conservé par les attributs
- Exemple: générateur de nombres pairs
  - Opérateur «`()`» (non constant) et sans paramètres
  - Retourne le prochain nombre pair
- Implémentation

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

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

# Foncteurs standards

- Arithmétique: addition, soustraction, multiplication, division...
  - `plus<T>`, `minus<T>`, `multiplies<T>`, `divides<T>`...
- Comparaison: inférieur, supérieur, égal...
  - `less<T>`, `less_equal<T>`, `equal_to<T>`...
- Logique: et, ou, non
  - `logical_and<T>`, `logical_or<T>`...
- Utilisent simplement les opérateurs correspondants
  - `plus<T>` ⇒
```
T operator()(const T & a, const T & b) const
{ return a + b; }
```
  - `less<T>` ⇒
```
bool operator()(const T & a, const T & b) const
{ return a < b; }
```
  - `logical_and<T>` ⇒
```
bool operator()(const T & a, const T & b) const
{ return a && b; }
```

# Algorithmes génériques

- Algorithme générique ⇒ algorithme à trous
  - `std::sort(v.begin(),v.end(),std::greater<int>());`
  - Dernier paramètre = foncteur ou pointeur de fonction

- Implémentation à l'aide de la généricité
  - ```
    template <typename IT, typename COMP>
    void sort(const IT & debut, const IT & fin,
              const COMP & comparer);
    ```
  - `comparer()` ⇒ appel opérateur «`()`» si foncteur
  - `comparer()` ⇒ appel fonction si pointeur de fonction

- Souvent, créer un foncteur est fastidieux
  - Trouver un nom
  - Ecrire la classe
  - Pour un usage souvent ponctuel
    ```

# Les lambdas

- Expression lambda (depuis C++11)
  - Permet l'écriture d'une fonction à la volée
  - Pour un usage ponctuel
  - Fonction «anonyme»
  - Fonction «contextualisée» (cf. mécanisme de capture)

- Exemple
  - ```
    std::sort(v.begin(),v.end(),
              [] (int x, int y) { return x > y; });
    ```
  - Tri par ordre décroissant

- Syntaxe: `[`*capture*`] (`*arguments*`) -> `*retour*` { `*code*` }`
  - Arguments, retour, code = éléments d'une fonction normale
    - Remarque: utilisation de la syntaxe alternative (depuis C++11) de retour de fonction
  - Capture = liste des variables du contexte «capturées» par la lambda
    ```

# Lambda sans capture

- `[] (int x, int y) { return x > y; }`
  - Type de retour déduit automatiquement
    - A condition que tous les retours soient du même type
  - Equivalent à: `[] (int x, int y) -> bool { return x > y; }`

- Equivalent au foncteur suivant
  - ```
    struct Anonyme {
     bool operator()(int x, int y) const
     { return x > y; }
    };
    ```
  - Remarque: opérateur «`()`» constant

- Ou à la fonction suivante
  - `inline bool anonyme(int x, int y) { return x > y; }`
    ```

# Implémentation des lambdas

- Implémentation libre des lambdas
  - Souvent sous la forme d'un foncteur
  - Mais pour les lambdas sans capture, une fonction suffit
- Dépend donc du compilateur

  ⇒ impossible de connaître *a priori* le type d'une lambda
- Mais possibilité de stocker une lambda dans une variable
  - `auto f = [] (int x, int y) { return x > y; };`
  - `if (f(v[i],v[j])) …`
  - `auto` ⇒ type de la lambda identifié par le compilateur pour déclarer «`f`»
- Et aussi de «capter» le type d'une lambda
  - `using lambda_t = decltype(f);`
  - `typeid(lambda_t).name()` ⇒ *`main::{lambda(int, int)#2}`* (g++ 4.8.3)
  - `decltype` ⇒ type de l'expression (ici variable «`f`») demandé au compilateur

# Capture de variables *(1/5)*

- Une lambda peut utiliser des variables de son contexte
  ⇒ mécanisme de «capture»

- Exemple: filtrer les valeurs d'un échantillon
  - `std::replace_if(v.begin(),v.end(),filtre,-1);`
  - `filtre` = prédicat (politique de «test»)
  - Prédicat testé sur chaque élément «`x`» ⇒ `if (filtre(x)) …`
  - Prédicat vérifié ⇒ valeur de «`x`» remplacée par «`-1`»

- Capture des données de l'intervalle du filtre

```
int min = …;
int max = …;
…
auto filtre = [min,max] (int x) { return (x<min || x>max); };
```

- Variables capturées listées dans «`[…]`»
  - Variable utilisée sans être capturée ⇒ erreur

# Capture de variables *(2/5)*

- Deux types de captures
  - [*variable*] ⇒ capture par copie
  - [&*variable*] ⇒ capture par référence

- Capture par copie = copie de la variable capturée
  - Modification de la variable dans le contexte ⇒ aucun impact dans la lambda

- Exemple

```
int min = 5;
int max = 7;
…
auto filtre = [min,max] (int x) { return (x<min || x>max); };
…
min = 3;
max = 10;
…
replace_if(v.begin(),v.end(),filtre,-1); // filtre = [5;7]
```

# Capture de variables *(3/5)*

- Capture par référence = référence sur la variable capturée
  - Evite la copie (important pour les gros objets)
  - Modification de la variable dans le contexte ⇒ impact dans la lambda
  - Attention à la durée de vie des variables capturées par référence

- Exemple

```
int min = 5;
int max = 7;
…
auto filtre = [&min,&max] (int x) { return (x<min || x>max); };
…
min = 3;
max = 10;
…
replace_if(v.begin(),v.end(),filtre,-1); // filtre = [3;10]
```

# Capture de variables *(4/5)*

- Capture automatique possible
  - Variable utilisée ⇒ variable capturée
  - Seules les variables utilisées dans la lambda sont capturées
  - `[]` ⇒ aucune capture
  - `[=]` ⇒ capture automatique par copie
  - `[&]` ⇒ capture automatique par référence

- Exemple: capture automatique par copie

```
int min = 5;
int max = 7;
…
auto filtre = [=] (int x) { return (x<min || x>max); };
```

# Capture de variables *(5/5)*

- Capture de «`this`»
  - `[this]` ⇒ capture du pointeur de l'objet du contexte

- Exemple

```
class Statistique {
 private:
  int min_;
  int max_;

 public:
  …
  void filtrer(vector<int> & v) const {
   auto filtre = [this] (int x) {
    return (x < this->min_ || x > this->max_);
   };

   replace_if(v.begin(),v.end(),filtre,-1);
  }
};
```

# Foncteur et capture *(1/2)*

- Lambda avec capture ⇒ implémentation par foncteur
- Capture par copie
  - `[min,max] (int x) { return (x<min || x>max); };`
- Foncteur équivalent

```
struct Anonyme {
 int min;
 int max;

 Anonyme(int a, int b) : min(a), max(b) {}

 bool operator()(int x) const {
  return (x<min || x>max);
 }
};
```

# Foncteur et capture *(2/2)*

- Capture par référence
  - `[&min,&max] (int x) { return (x<min || x>max); };`

- Foncteur équivalent

```
struct Anonyme {
 int & min;
 int & max;

 Anonyme(int & a, int & b) : min(a), max(b) {}

 bool operator()(int x) const {
  return (x<min || x>max);
 }
};
```

# Lambda constante

- Opérateur « ( ) » constant dans les exemples précédents

- Rappel: dans une méthode constante…
  - Les attributs deviennent constants
  - Mais attention au cas des pointeurs/références
  - Les pointeurs/références sont constants mais pas les objets référencés !

- Par défaut, une lambda est «constante»
  ⇒ implémentation d'un foncteur avec opérateur « ( ) » constant

- Lambda constante
  ⇒ les variables capturées par copie sont constantes
  - Car les variables deviennent des attributs du foncteur
  - Capture par copie ⇒ attribut valeur ⇒ variable capturée constante
  - Capture par référence ⇒ attribut référence ⇒ variable capturée modifiable

# Lambda non constante

- Lambda non constante ⇒ mot-clé «`mutable`»

  ⇒ Modification possible des variables capturées par copie

  ⇒ Foncteur avec opérateur «`()`» non constant

- Exemple: produire des nombres pairs

```
int cpt = 32;
…
auto gen = [cpt] () mutable {
 cpt += 2;
 return cpt;
};
…
std::generate(v.begin(),v.end(),gen);
```

- Attention: une lambda peut donc être un objet non constant
  - `template <typename LAMBDA> void algo(const LAMBDA &)` ⇒ erreur possible

# Abstraction du type de fonction

- Trois manières de modéliser une fonction
  - Pointeur de fonction
    - Une méthode est considérée comme une fonction dont le 1er argument est le pointeur de l'objet
  - Foncteur
    - Objet avec opérateur « ( ) »
  - Lambda
    - Type inconnu
    - Implémentation comme fonction ou foncteur
- Types différents, mais même manière d'être appelés
- Comment faire abstraction de ces trois types ?
- Objectif: algorithme recevant indifféremment en paramètre un pointeur de fonction, un foncteur ou une lambda

# Abstraction de fonction par généricité *(1/2)*

- Une approche: abstraction par un paramètre générique
  - Avantage: très efficace
    - Instanciation adaptée au type de modélisation
  - Inconvénient: difficile de contrôler le paramètre
    - Comment être sûr qu'il représente bien une fonction ?
    - Solution alternative: abstraction par un «adapteur», mais surcoût (cf. `std::function`)

- Passage par référence constante ?
  - `template <typename F> void algo(const F & f);`
  - Problème pour les lambdas/foncteurs non constants

- Passage par référence non constante ?
  - `template <typename F> void algo(F & f);`
  - Problème pour les *rvalues* ou les pointeurs de fonction
  - Et souvent une lambda est une *rvalue*: `algo([…] (…) {…});`

# Abstraction de fonction par généricité *(2/2)*

- Passage par copie ?

  ⇒ inefficacité

- Solution: passage par «référence universelle»
  - Appelée aussi *forwarding reference* (cf. *collapsing rules*)
  - `template <typename F> void algo(F && f);`
  - Accepte des valeurs constantes ou non
  - Accepte des *lvalues* ou des *rvalues*

- Exemple

```
template <typename IT, typename GEN>
void generate(const IT & debut, const IT & fin, GEN && generer) {
 for (IT it = debut; it != fin; ++it) *it = generer();
}
```

# Conteneurs de la STL *(1/3)*

- Quatre grandes classes de conteneurs
  - Séquences élémentaires
    - Vecteur, liste, file à double entrée
    - Tableau statique (depuis C++11)
  - Adaptations des séquences élémentaires
    - Pile, file, file à priorité
  - Conteneurs associatifs triés
    - Ensemble avec/sans unicité
    - Association avec clé unique/multiple
  - Conteneurs associatifs non triés (depuis C++11)

- Utilisation intensive de la généricité
  - Type de données
  - Allocateur de mémoire
  - Comparateur
  - …

# Conteneurs de la STL *(2/3)*

- 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)*
  - Parfois amortie

- 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

# Conteneurs de la STL *(3/3)*

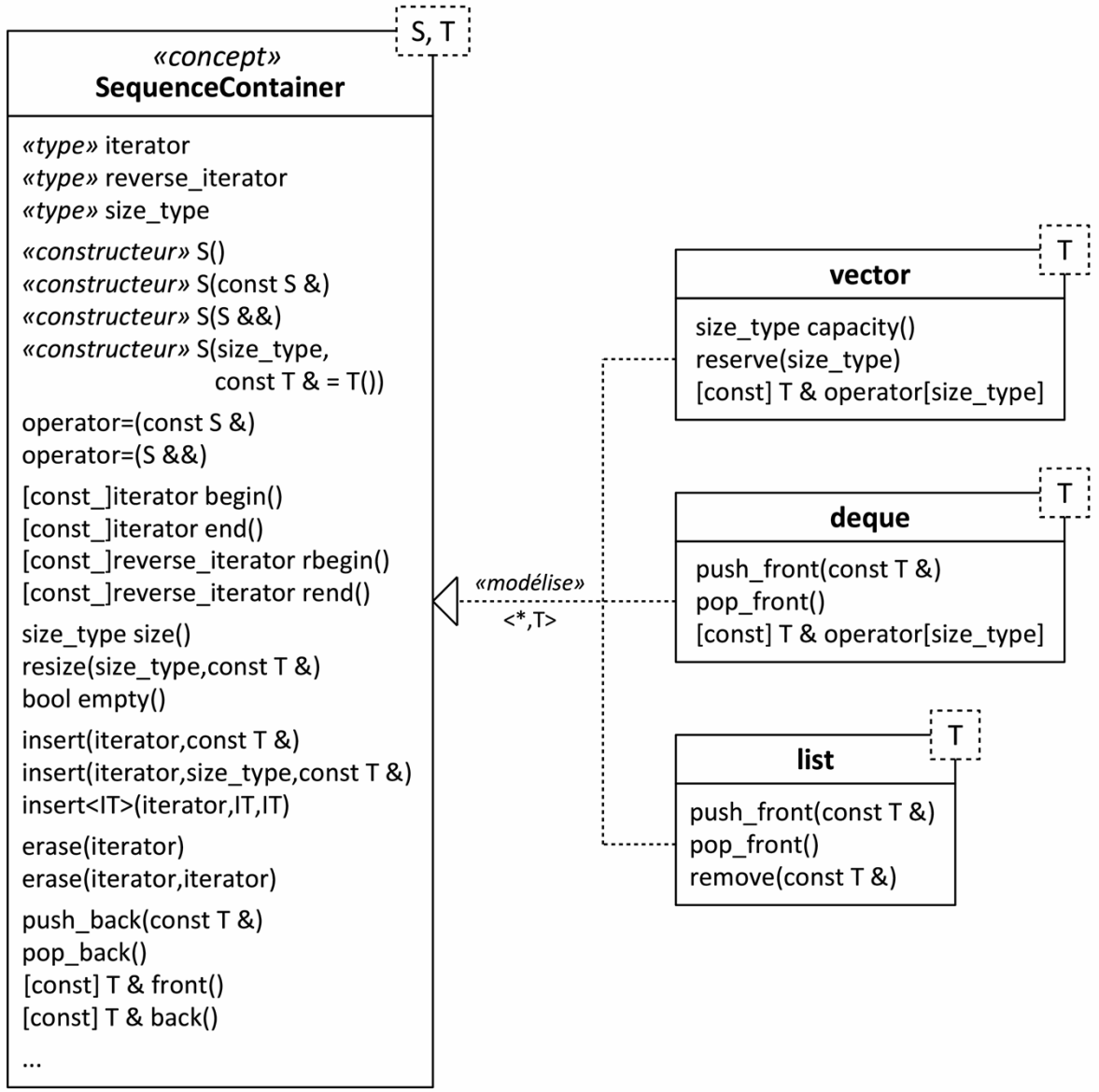
- Des fonctionnalités communes
- Forme normale de Coplien (+ opérateurs de mouvement)
- Dimensionnement automatique de la capacité
  - Excepté pour le tableau statique (`array`)
  - Exemple du vecteur
    - Lorsque l'insertion d'un élément survient en limite de capacité
    - Augmentation de la capacité
- Balises des itérateurs
- Quelques méthodes
  - `size_type C::size() const` `// Nombre d'éléments`
  - `size_type C::max_size() const` `// Nombre max d'éléments`
  - `bool C::empty() const` `// Est vide ?`
  - `void C::swap(C & conteneur)` `// Echange de contenu`
  - `void C::clear()` `// Vide le conteneur`

# Conteneurs en séquence *(1/2)*

- Vecteur (`vector<T>`)
- Liste doublement chaînée (`list<T>`)
- File à double entrée (`deque<T>`)

## Depuis C++11

- Tableau (`array<T,N>`)
- Liste simplement chaînée (`forward_list<T>`)



# Conteneurs en séquence *(2/2)*

- Quelques méthodes communes

- Insertion (avant la position indiquée)
  - `void S::insert(S::iterator pos, 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`
    ```

# Vecteur

- Tableau qui se redimensionne automatiquement
  - Eléments 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;`
- Méthodes spécifiques
  - Contrôle capacité
    - `size_type V::capacity() const   // Capacité actuelle`
    - `void      V::reserve(size_type nb) // Ajustement capacité`
  - Accès par indice aux éléments
    - `X &       V::operator[](size_type id) // Lecture/écriture`
    - `const X & V::operator[](size_type id) const // Lecture seule`

# Liste

- 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;`
- 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)`

# File à double entrée *(1/2)*

- Similaire au vecteur + possibilité d'opérations en tête
  - 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;`
- Méthodes spécifiques
  - Ajout / suppression en tête
    - `void D::push_front(const T & elt)`
    - `void D::pop_front()`
  - Accès par indice aux éléments
    - `X &       D::operator[](size_type id)`
    - `const X & D::operator[](size_type id) const`

# File à double entrée *(2/2)*

```
std::deque<std::string> file;

file.push_back("...");
file.push_front("B");
file.push_front("A");
file.push_back("Y");
file.push_back("Z");

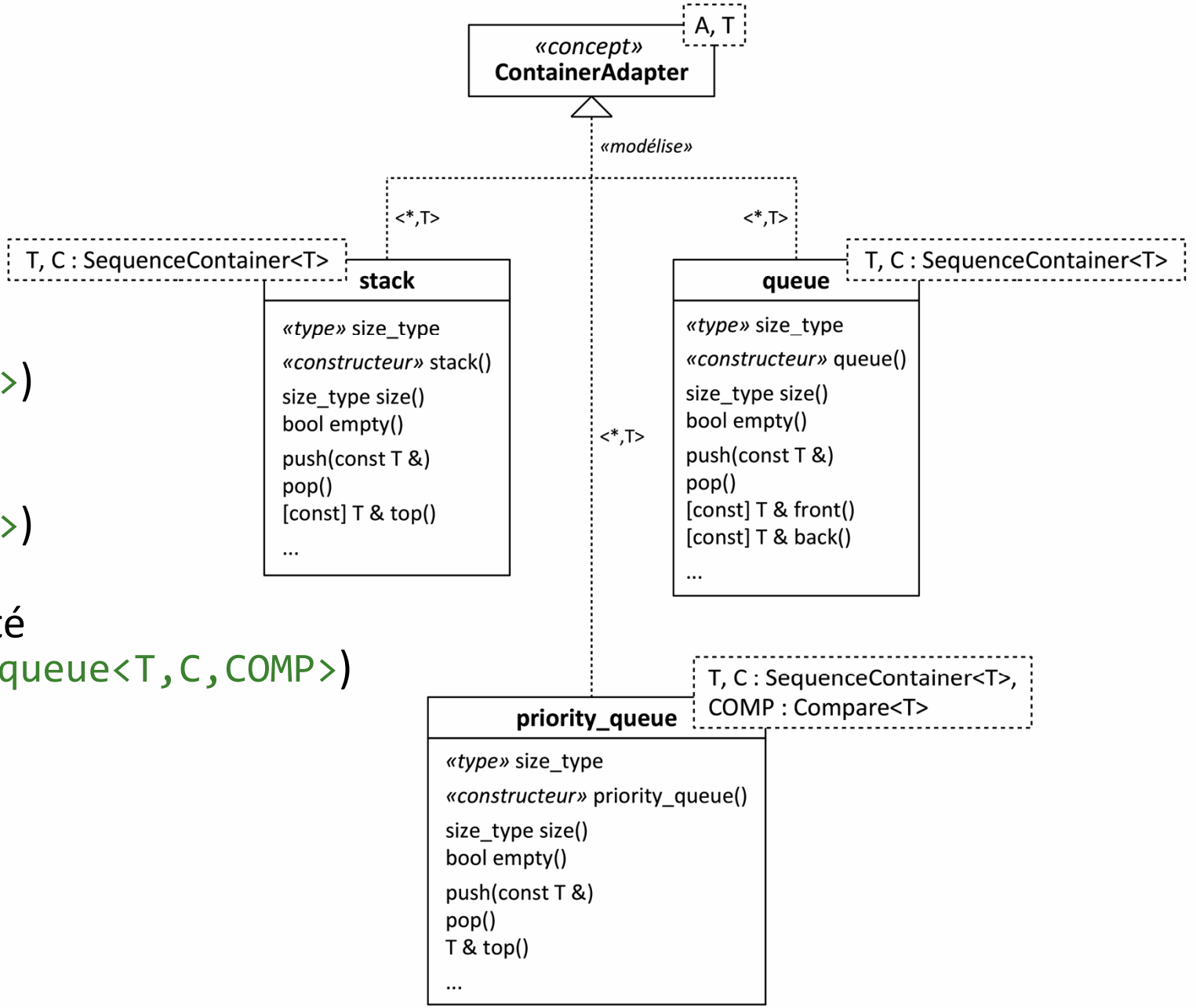
for (const std::string & s : file)
 std::cout << s << " "; // ⇒ A B ... Y Z

file.pop_front();
file.pop_back();

for (const std::string & s : file)
 std::cout << s << " "; // ⇒ B ... Y
```

# Conteneurs adapteurs *(1/3)*

- Pile (`stack<T,C>`)
- File (`queue<T,C>`)
- File à priorité (`priority_queue<T,C,COMP>`)

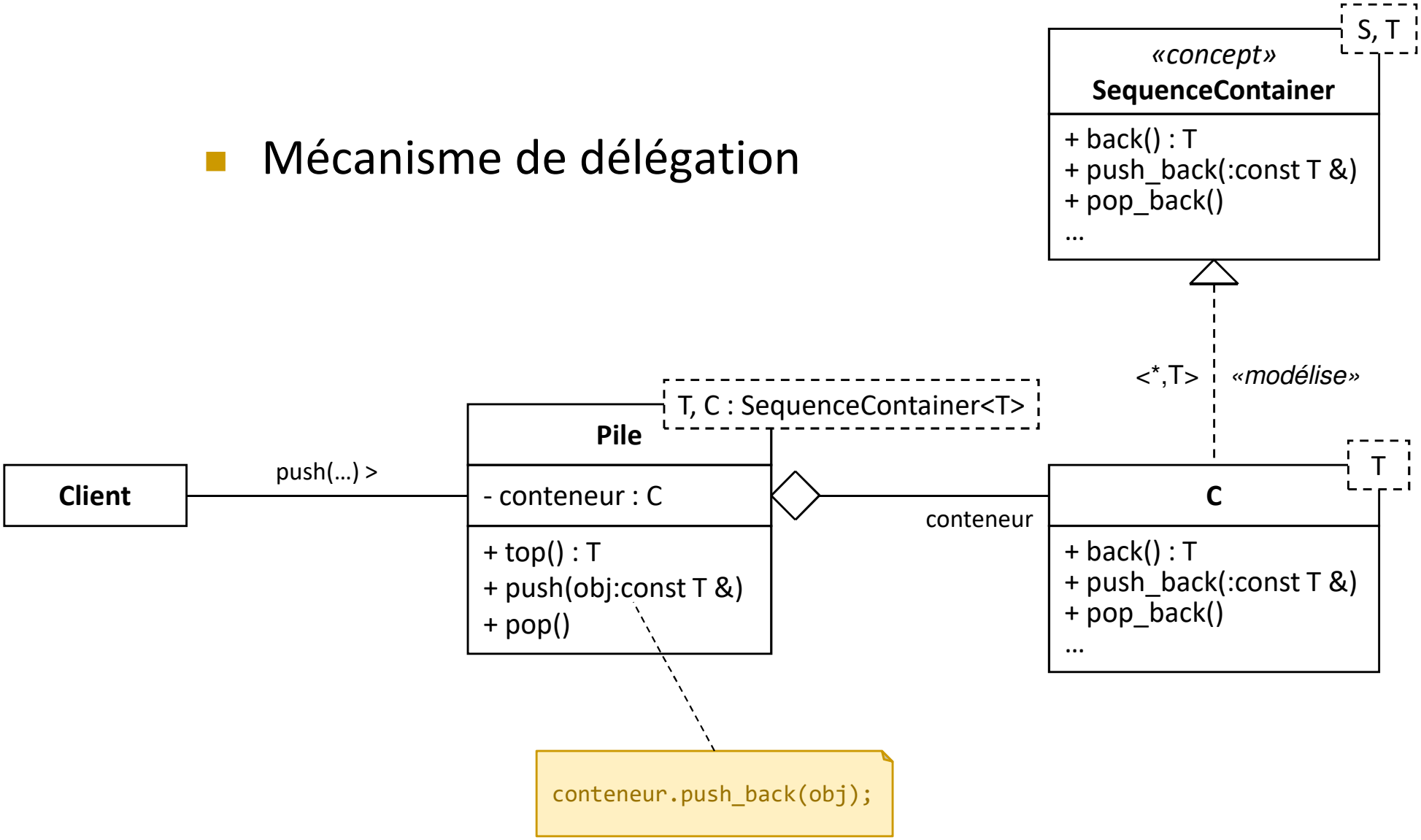


# Conteneurs adapteurs *(2/3)*

- 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



# Pile

- Accès seulement au sommet de la pile
  - Pas de possibilité de voir les éléments empilés
- Utilisation
  - Entête: `<stack>`
  - Déclaration
    - `std::stack<T> s; // C = deque<T>`
    - `std::stack<T,std::vector<T>> s;`
- Méthodes spécifiques
  - Empilement / dépilement
    - `void S::push(const T & elt)`
    - `void S::pop()`
  - Accès au sommet
    - `T &       S::top()`
    - `const T & S::top() const`

# File

- Structure FIFO (*First In First Out*)
  - Ajout en fin, retrait en tête
  - Pas de possibilité de voir les éléments dans la file
- Utilisation
  - Entête: `<queue>`
  - Déclaration
    - `std::queue<T> q; // C = deque<T>`
    - `std::queue<T,std::list<T>> q;`
    - Ne peut pas utiliser «`vector`» (manque «`pop_front`»)
- Méthodes spécifiques
  - Ajout / retrait
    - `void Q::push(const T & elt)`
    - `void Q::pop()`
  - Accès aux extrémités
    - `T & Q::front() | const T & Q::front() const`
    - `T & Q::back()  | const T & Q::back() const`

# File à priorité *(1/2)*

- File d'attente à priorité
  - Ajout en fin, retrait de l'élément le plus «grand» ⇒ politique «comparateur»
  - Pas de possibilité de voir les éléments dans la file
- Utilisation
  - Entête: `<queue>`
  - Déclaration
    - `std::priority_queue<T> p; // C = vector<T>, COMP = less<T>`
    - `std::priority_queue<T,std::deque<T>,std::greater<T>> q;`
    - Ne peut pas utiliser «`list`» (manque opérateur «`[]`»)
- Méthodes spécifiques
  - Constructeur (qui attend une politique «comparateur»)
    - `P::P(const COMP & c = COMP())`
  - Ajout / retrait
    - `void P::push(const T & elt)`
    - `void P::pop()`
  - Accès au plus grand
    - `T &       P::top()`
    - `const T & P::top() const`

# File à priorité *(2/2)*

```
std::priority_queue<int,std::vector<int>,std::greater<int>> file;

// std::greater ⇒ ordre décroissant
// ⇒ le plus petit est le plus prioritaire

file.push(23);
file.push(12);
file.push(99);
file.push(3);

while (!file.empty()) {
 int v = file.top();

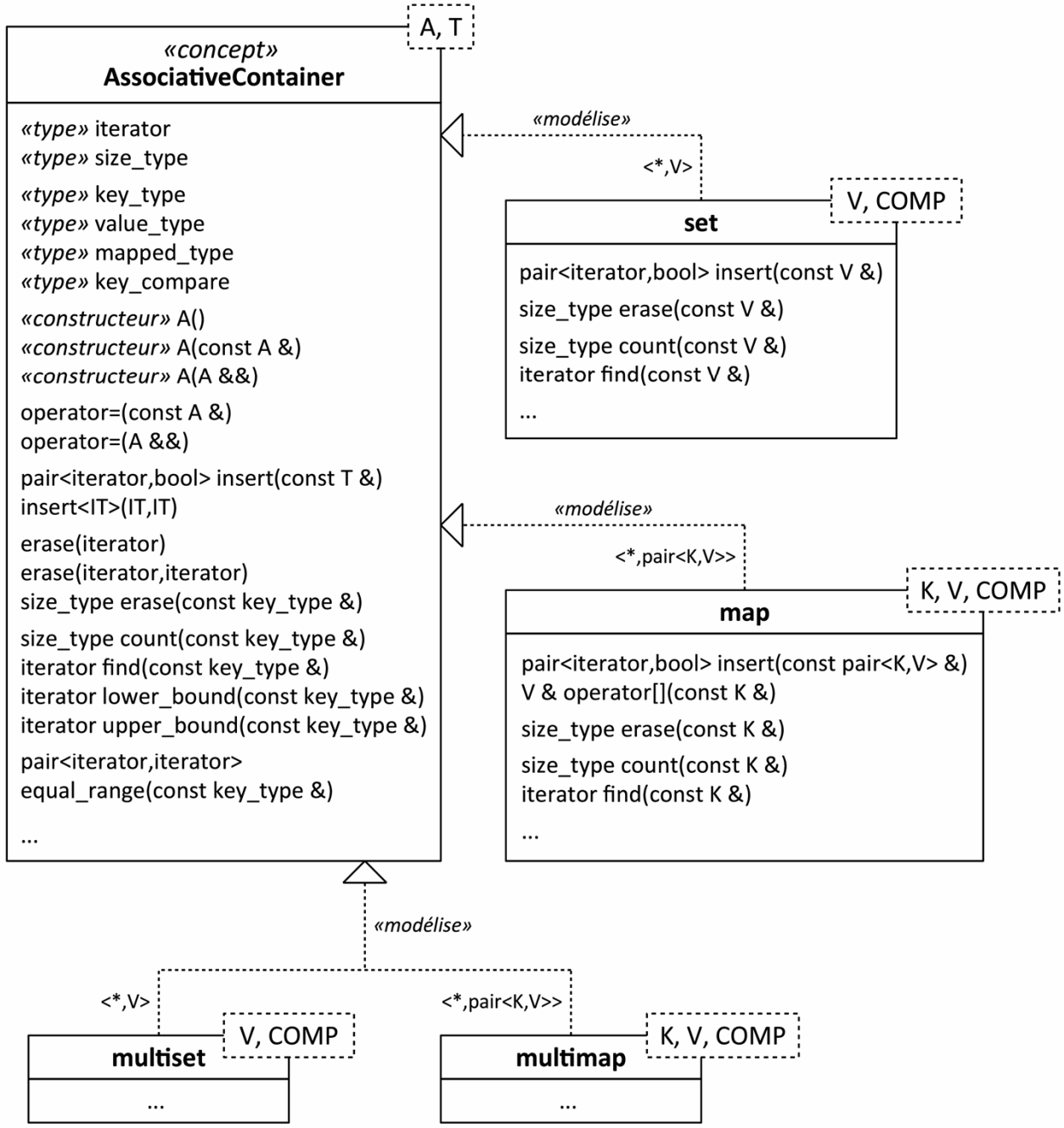
 std::cout << v << " ";
 file.pop(); // Dépilement plus petit → plus grand
} // ⇒ 3 12 23 99
```

# Conteneurs associatifs *(1/5)*

- Ensemble avec unicité (`set<V,COMP>`)
- Ensemble sans unicité (`multiset<V,COMP>`)
- Association avec unicité (`map<K,V,COMP>`)
- Association sans unicité (`multimap<K,V,COMP>`)

## Depuis C++11

- Versions non triées
  - `unordered_set`
  - `unordered_multiset`
  - `unordered_map`
  - `unordered_multimap`



# Paire *(1/2)*

- 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() {}

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

# Paire *(2/2)*

- Création d'une paire
  - `p = std::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>
std::pair<T1,T2> make_pair(const T1 & a, const T2 & b)
{ return std::pair<T1,T2>(a,b); }
```

- Utilise la déduction de types du compilateur
  - `p = std::make_pair(13,27.14);`

  ⇔ `std::make_pair<int,double>(13,27.14);`

# Conteneurs associatifs *(2/5)*

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

# Conteneurs associatifs *(3/5)*

- «`set`» et «`multiset`» ⇒ un seul paramètre
  - `set<V>` | `multiset<V>`
  - Eléments stockés: `T = V`
- «`map`» et «`multimap`» ⇒ deux paramètres
  - `map<K,V>` | `multimap<K,V>`
  - Eléments stockés: `T = std::pair<const K,V>`
- Quelques méthodes communes *(1/3)*
- Constructeur (qui attend une politique «comparateur»)
  - `A::A(const COMP & c = COMP())`
- Insertion
  - `pair<A::iterator,bool> A::insert(const T & elt)`
  - ```
    template <typename InputIterator>
    void A::insert(InputIterator deb, InputIterator fin)
    ```
    ```

# Conteneurs associatifs *(4/5)*

- Quelques méthodes communes *(2/3)*

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

- Recherche élément
  - `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

# Conteneurs associatifs *(5/5)*

- Quelques méthodes communes *(3/3)*

- Intervalle

  - `A::iterator A::lower_bound(const A::key_type & cle) const`
  - Itérateur sur le 1er élément dont la clé est égale ou supérieure à celle fournie

  - `A::iterator A::upper_bound(const A::key_type & cle) const`
  - Itérateur sur le 1er 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
    (intervalle défini par «`lower_bound`» et «`upper_bound`»)

# Ensembles *(1/2)*

- Conteneur trié d'éléments contenant leur propre clé
- Utilisation
  - Entête: `<set>`
  - Déclaration
    - `std::set<V> s; // COMP = less<V>`
    - `std::set<V,std::greater<V>> s;`
- Méthodes spécifiques
  - Insertion dans «`set`»
  - `pair<S::iterator, bool> S::insert(const V & elt)`
  - Insertion dans «`multiset`»
  - `M::iterator M::insert(const V & elt)`

# Ensembles *(2/2)*

```
struct Personne { std::string nom; std::string prenom; };

struct CompPersonne {
 bool operator()(const Personne & p1, const Personne & p2) const {
  return p1.nom < p2.nom || (p1.nom == p2.nom && p1.prenom < p2.prenom);
 }
};

std::multiset<Personne,CompPersonne> ensemble;

ensemble.insert(Personne{"Doe","John"});
ensemble.insert(Personne{"Smith","John"});
ensemble.insert(Personne{"Doe","Jane"});

for (const Personne & p : ensemble) {
 std::cout << p.nom << ";" << p.prenom << " ";
} // ⇒ Doe;Jane Doe;John Smith;John
```

# Associations *(1/4)*

- Conteneur trié d'éléments associés à une clé
- Utilisation
  - Entête: `<map>`
  - Déclaration
    - `std::map<K,V> s; // COMP = less<K>`
    - `std::map<K,V,std::greater<K>> s;`
- Méthodes spécifiques
  - Insertion
  - `pair<M::iterator,bool> M::insert(const pair<const K,V> &)`
  - Accès élément
  - `V & M::operator[](const K & cle)`

# Associations *(2/4)*

- Remarques sur l'opérateur «`[]`»
- Permet un accès indexé similaire au vecteur
  - Indice = 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

# Associations *(3/4)*

- Exemple *(1/2)*

```
std::multimap<std::string,Personne> asso;

asso.insert(std::make_pair("homme",Personne{"Smith","John"}));
asso.insert(std::make_pair("homme",Personne{"Doe","John"}));
asso.insert(std::make_pair("femme",Personne{"Doe","Jane"}));

for (const std::pair<std::string,Personne> & a : asso) {
 std::cout << a.first << "=" << a.second.nom
                      << ";" << a.second.prenom << " ";
} // ⇒ femme=Doe;Jane homme=Smith;John homme=Doe;John
```

# Associations *(4/4)*

- Exemple *(2/2)*

```
using iter_t = std::multimap<std::string,Personne>::iterator;

std::pair<iter_t,iter_t> hommes = asso.equal_range("homme");

iter_t it = hommes.first;

while (it != hommes.second) {
 std::cout << it->second.nom << ";"
           << it->second.prenom << " ";
 ++it;
} // ⇒ Smith;John Doe;John
```

# Types de données internes

- Les conteneurs définissent des types internes
  - Embarqués dans les classes
- Pour tous les conteneurs
  - `C::value_type`: type des éléments stockés
    - Pour les associations: `pair<const K,V>`
  - `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::mapped_type`: type des valeurs
  - `C::key_compare`: comparateur des clés

# Algorithmes de la STL

- Fonctionnalités classiques/récurrentes
  - Chercher, compter
  - Copier, insérer, supprimer
  - Remplir, transformer, trier
  - …
- Indépendants des conteneurs ⇒ manipulation d'itérateurs
  - Paramètres génériques: tout type d'itérateur/pointeur
  - Itérateurs de début et de fin = séquence où lire les éléments
  - Parfois itérateur de sortie pour écrire le résultat
- Algorithmes à trous ⇒ souvent paramétrés par une politique
  - Paramètre générique: pointeur de fonction, foncteur ou lambda
  - Politique = comparateur, prédicat, générateur...

# Paramétrage par lambda

- Exemple: `for_each` ⇒ traiter chaque élément d'une séquence

```
std::vector<int> v = {3,5,7,13};

// Parcours avec modification des éléments: +1 pour chacun
std::for_each(v.begin(),v.end(),[] (int & x) { x += 1; });

for (int x : v) std::cout << x << " "; // ⇒ 4 6 8 14

int s = 0;

// Parcours sans modification des éléments: calcul somme
std::for_each(v.begin(),v.end(),[&s] (int x) { s += x; });

std::cout << "s = " << s << std::endl; // ⇒ s = 32
```

# Interrogation

- Recherche d'un élément dans une séquence
  - `it = std::find(it_debut,it_fin,valeur)`
    ⇒ recherche *x = valeur*
  - `it = std::find_if(it_debut,it_fin,predicat)`
    ⇒ recherche *x* tel que *prédicat(x) = vrai*
  - Si non trouvé ⇒ `it` = `it_fin`

- Exemple (suite précédent)

```
auto it = std::find_if(v.begin(),v.end(),
                       [] (int x) { return x%3 == 0; });
if (it != v.end()) std::cout << *it << std::endl; // ⇒ 6
```

- Comptage des éléments d'une séquence
  - `n = std::count(it_début,it_fin,valeur)`
  - `n = std::count_if(it_début,it_fin,predicat)`

# Copie

- Copie des éléments d'une séquence dans une autre
  - *[itérateur_début, itérateur_fin] → [itérateur_destination, …]*
  - Tous: `it = std::copy(it_debut,it_fin,it_dest)`
  - Filtre: `it = std::copy_if(it_debut,it_fin,it_dest,predicat)`
  - «`it_dest`» pointe sur le 1er élément où faire la copie
    - Utilisation des opérateurs «`*`» et «`++`» pour faire la copie et avancer
  - «`it`» pointe sur la valeur de «`it_dest`» en fin de copie

- Exemple

```
int buffer[100];
std::list<int> liste;

// Copie vecteur → tableau
// Itérateur tableau = pointeur
int * fin = std::copy(v.begin(),v.end(),buffer);

// Copie tableau → liste
// «back_inserter» ⇒ itérateur qui appelle «push_back»
std::copy(buffer,fin,std::back_inserter(liste));
```

# Suppression / Remplacement

- Suppression d'éléments
  - `it = std::remove(it_debut,it_fin,valeur)`
  - `it = std::remove_if(it_debut,it_fin,predicat)`
  - Déplace seulement les éléments à la fin de la séquence
  - Retourne un itérateur sur la «nouvelle fin»
  - Enchaîner avec méthode «`erase`» pour supprimer réellement
  - Exemple: `v.erase(std::remove(v.begin(),v.end(),14),v.end())`

- Remplacement des éléments
  - `std::replace(it_debut,it_fin,valeur,nouvelleValeur)`
  - `std::replace_if(it_debut,it_fin,predicat,nouvelleValeur)`

# Transformation

- Appliquer une opération à chaque élément d'une séquence
  - Résultat dans une autre séquence
  - `it = std::transform(it_debut,it_fin,it_dest,opUnaire)`
  - Ou une opération binaire sur les éléments de 2 séquences (deux-à-deux)
  - `it = std::transform(it_debut1,it_fin1,it_debut2,it_dest,opBinaire)`

- Exemple

```
std::vector<int> v1 = {1,2,3,4,5}, v2, v3;

std::transform(v1.begin(),v1.end(),std::back_inserter(v2),
               [] (int x) { return 2*x; });

for (int x : v2) std::cout << x << " "; // ⇒ 2 4 6 8 10

std::transform(v1.begin(),v1.end(),v2.begin(),std::back_inserter(v3),
               [] (int x, int y) { return x + y; });

for (int x : v3) std::cout << x << " "; // ⇒ 3 6 9 12 15
```

# Remplissage

- Remplir une séquence avec la même valeur
  - `std::fill(it_debut,it_fin,valeur)`
  - `std::fill_n(it_debut,n,valeur)`
- Ou avec une valeur produite à chaque appel d'une fonction
  - `std::generate(it_debut,it_fin,generateur)`
  - `std::generate_n(it_debut,n,generateur)`
- Exemple

```
std::vector<int> v1(4), v2(3);
int n = 0;

std::generate(v1.begin(),v1.end(),[&n] () { return n+=2; });
for (int x : v1) std::cout << x << " "; // ⇒ 2 4 6 8

std::generate_n(v2.begin(),v2.size(),[&n] () { return n+=2; });
for (int x : v2) std::cout << x << " "; // ⇒ 10 12 14
```