ISIMA

PARTIE VIII

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
  - Ancienne doc (qui reste pratique): **`http://www.boost.org/sgi/stl`**

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, publique…)

- 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 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 ?

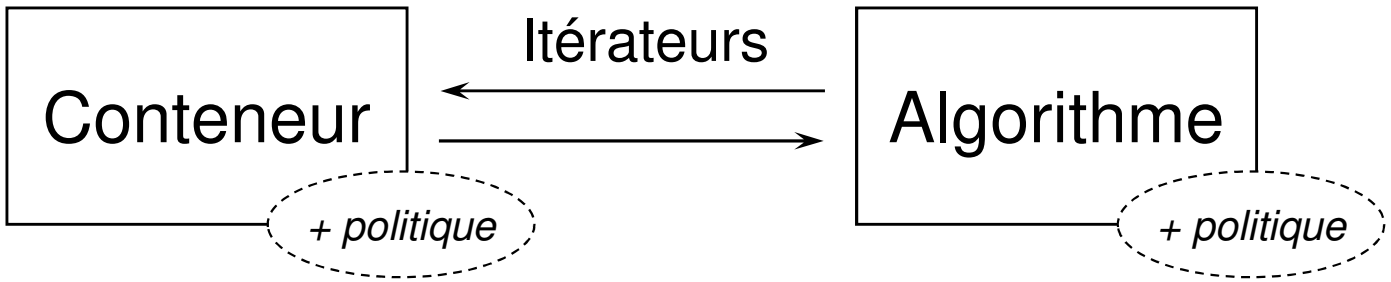
  ⇒ Abstraction du conteneur et de la stratégie de parcours: les «itérateurs»

- 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 $\Rightarrow$ 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)*

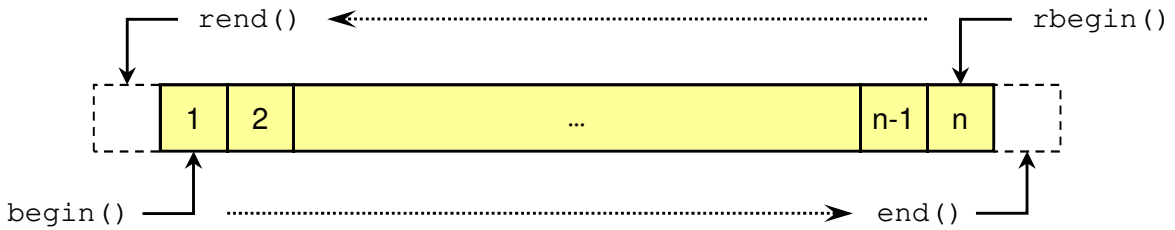
- Interface d'un itérateur en C++
  - Forme normale de Coplien
    - Constructeur par défaut
    - Constructeur par copie
    - Opérateur d'affectation
    - Destructeur
  - API et sémantique du pointeur (arithmétique partielle)
    - Opérateurs de comparaison «**!=**» et «**==**»
      - Attention: 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) // Accès avec écriture
     v *= 2;
    ```
  - ```
    for (float v : t) // Accès avec lecture seule
     std::cout << v << " ";
    ```

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

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

- A suivre…