Introduction à la STL

ISIMA - ZZ2 - 2009

Christophe Duhamel
Andréa Duhamel

ISIMA Introduction

- Présence récurrente en développement des mêmes
  - 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++

2

ISIMA Pourquoi utiliser la STL

- Fiabilité : collection de classes largement utilisées
- Portabilité : librairie standard, totalement en C++
- Efficacité : utilisation intensive de la généricité et des structures de données optimisées pour garantir les meilleures performances
- Compréhensibilité : toutes les classes suivent les mêmes conventions d'utilisation

3

ISIMA Classes utilitaires de la STL

- Classe chaîne de caractères : `string`
  - gestion automatique de la mémoire (FNC)
  - surcharge des opérateurs classiques (+, <<)
  - définie dans le namespace `std`, header `<string>`
  - utilisation

```
#include <string>
…
std::string str1("chaine lue : "), str2;
std::cin >> str2;
std::cout << str1 + str2 << std::endl;
str2[0] = '\0';
str1 += str2;
std::cout << str1 << std::endl;
```

→ utiliser le plus possible **`string`** à la place de **`char *`** 4

ISIMA Classes utilitaires de la STL

- Classe exception : `exception`
  - fonctionnalités de base (FNC)
  - méthode `what()` : renvoie une description (`char *`)
  - définie dans le namespace `std`, header `<exception>`
  - définition

```
class exception
{
  public:
    exception () throw();
    exception (const exception& rhs) throw();
    exception & operator= (const exception& rhs) throw();
    virtual ~exception() throw();
    virtual const char * what() const throw();
};
```

→ dériver les exceptions personnelles de `exception`

5

ISIMA Conteneurs de la STL

- Trois grandes classes de conteneurs
  - séquences élémentaires
    - vecteur, liste et file à double entrée
  - adaptations des séquences élémentaires
    - pile, file et 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, allocation…)

6

ISIMA Conteneurs de la STL

- Fonctionnalités communes à tous les conteneurs
  - Forme Normale de Coplien
  - dimensionnement automatique de la capacité
    - lorsque l'insertion d'un élément viole la capacité
    - doublement de la capacité
    - permet une adaptation rapide à la taille « finale »
  - quelques méthodes

```
int C::size () const     // nombre d'éléments
int C::max_size () const // nombre max
bool C::empty () const   // prédicat de vacuité
void C::swap (C & cnt)   // échange de contenu
void C::clear ()         // purge
```

7

ISIMA Séquences élémentaires

- Fonctionnalités communes à toutes les séquences
  - Insertion : insert

    `Utilise les itérateurs: voir plus loin.`
  - Suppression : erase

    `Utilise les itérateurs: voir plus loin.`
  - accès en bordure de la séquence

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

8

ISIMA Le vecteur

- Vecteur
  - header : `<vector>`
  - déclaration : `std::vector<T> vec;`
  - méthodes spécifiques

```
int V::capacity () const
void V::reserve (int nb)

      X & V::operator[] (int idx)
const X & V::operator[] (int idx) const
```

  - intéressant par ses accès en O(1)
  - déconseillé pour les insertions/suppressions O(n)

9

ISIMA Le vecteur

- exemple

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

int main (int, char **)
{
  typedef std::vector<int> ivector;

  ivector v1;
  for (int i = 0; i < 4; ++i)
    v1.push_back(i);

  ivector v2(4); // taille initiale 4
  for (int i = 0; i < 4; ++i)
    v2[i] = i;

  std::cout << (v1 == v2) ? "Ok" : "Pas Ok"
            << std::endl;

  return 0;
}
```

10

ISIMA La liste

- Liste
  - header : `<list>`
  - déclaration : `std::list<T> lst;`
  - méthodes spécifiques

```
void L::push_front (const T & elt)
void L::pop_front ()

void L::remove (const T & elt)
void L::sort ()
void L::sort (Comparator cmp)
void L::merge(list<T> & l)
splice, remove_if, unique
```

  - intéressant pour les insertions/suppressions en O(1)
  - déconseillé pour les accès directs en O(n)

11

ISIMA La liste

- exemple

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

int main(int, char **)
{
  typedef std::list<int> ilist;

  ilist l;
  for (int i = 0; i < 4; ++i)
    l.push_back((10 + 3*i) % 5);

  l.sort();    // 0 1 3 4
  l.reverse(); // 4 3 1 0

  // affiche le debut et la fin
  std::cout << l.front() << ' ' << l.back()
            << std::endl;

  return 0;
}
```

12

ISIMA La file à double entrée

- File à double entrée (vecteur circulaire)
  - header : `<deque>`
  - déclaration : `std::deque<T> deq;`
  - méthodes spécifiques

```
int D::capacity () const
void D::reserve (int nb)
void D::push_front (const T & elt)
void D::pop_front ()

      X & D::operator[] (int idx)
const X & D::operator[] (int idx) const
```

  - tous les avantages de `vector`
  - gestion des insertions/suppressions en tête en O(1)

13

ISIMA La file à double entrée

- exemple

```
#include <iostream>
#include <deque>

int main(int, char **)
{
  typedef std::deque<int> ideque;

  ideque v1;
  for (int i = 0; i < 4; ++i)
    v1.push_front(i);

  ideque v2(4); // taille initiale 4
  for (int i = 0; i < 4; ++i)
    v2[i] = i;

  std::cout << (v1 == v2) ? "Ok" : "Pas Ok"
            << std::endl;
  return 0;
}
```

14

ISIMA Les itérateurs

- Constat
  - souhaite souvent parcourir les éléments d'un conteneur
    - solution naïve : définir un pointeur sur la cellule courante dans le conteneur
    - limite : on ne peut pas avoir deux parcours en même temps
    - solution temporaire : sortir le pointeur du conteneur (en l'encapsulant !)
  - souhaite parfois avoir des parcours différents (avant, arrière…)
    - solution naïve : définir la stratégie de parcours dans le conteneur
    - limite : on ne peut avoir (proprement) qu'un seul type de parcours
    - solution temporaire : sortir la stratégie du conteneur (en l'encapsulant !)
- Itérateur
  - classe qui définit l'accès à un élément courant du conteneur
  - classe qui définit sa stratégie de parcours

15

ISIMA Les itérateurs

- Stratégies d'accès et de parcours des conteneurs
  - chaque itérateur définit sa propre stratégie de parcours
  - typiquement un pointeur sur un élément du conteneur
  - doit connaître l'implémentation de son conteneur
  - défini comme une classe imbriquée dans le conteneur

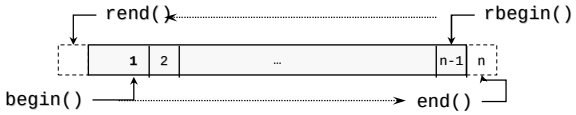
```
class Conteneur
{
  public:
    …
    class Iterateur { … };
    …
};
```

  - possibilité d'avoir plusieurs itérateurs en même temps
  - incompatibilité des itérateurs de conteneurs différents

16

ISIMA Les itérateurs

- 4 types d'itérateur par conteneur
  - conteneur::iterator
  - conteneur::const_iterator
  - conteneur::reverse_iterator
  - conteneur::const_reverse_iterator
- Balises fournies par le conteneur



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

17

ISIMA Les itérateurs

- Fonctionnalités
  - FNC
  - opérateurs de comparaison `!=` et `==`
  - opérateur de déréférenciation `*`
  - opérateur de préincrémentation `++`

18

ISIMA Les itérateurs

- 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 `find()`
    - permet une opération immédiate sur l'objet
    - complexité de l'accès suivant : O(1)

```
conteneur c;
conteneur::iterator it = find(c.begin(), c.end(), elt);
do_something(*it);
```

19

ISIMA Séquences élémentaires (le retour)

- Fonctionnalités communes à toutes les séquences
  - Insertion

```
S::iterator S::insert (S::iterator before, T & elt)
S::iterator S::insert (S::iterator before, int nb, T & elt)
S::iterator S::insert (S::iterator before, S::const_iterator
                       first, S::const_iterator last)
```

  - Suppression

```
S::iterator S::erase (S::iterator pos)
S::iterator S::erase (S::const_iterator first,S::const_iterator last)
```

  - accès en bordure de la séquence

déjà vu

20

ISIMA Les foncteurs

- Constat
  - souhaite souvent appliquer un algorithme sur un conteneur
    - solution naïve : le placer en méthode du conteneur
    - limite : pollution de l'interface du conteneur
    - solution : placer les algorithmes dans des classes dédiées
  - souhaite parfois avoir plusieurs versions de l'algorithme
    - solution naïve : utiliser la surcharge (polymorphisme faible)
    - limite : on n'est pas sûr d'appeler la bonne version
    - solution : utiliser l'héritage
- Foncteur
  - classe qui implémente un algorithme sur le conteneur
  - une sous-classe par variante
  - accès aux éléments du conteneur par les itérateurs

21

ISIMA Les foncteurs

- Combinaison de deux principes
  - surcharge de l'opérateur `()`
    - arité spécifiée par le concepteur
    - syntaxe : `type_retour A::operator() (paramètres)`
    - intérêt : un objet se comporte comme une fonction
    - note : peut aussi être utilisé pour remplacer l'opérateur `[]`
    - l'algorithme souhaité est implémenté dans l'opérateur `()`
    - les arguments de l'algorithmes sont placés en argument
  - Séparation des algorithmes et des conteneurs

22

ISIMA Les foncteurs

- Exemple : générateur de nombres pairs
  - principe
    - état interne conservé par les attributs
    - opérateur `()` sans paramètres pour la génération des nombres
  - code

```
class GenPair
{
  protected:
    unsigned val;
  public:
    GenPair () {val = 0;}
    unsigned operator() (void) {val += 2; return val;}
};
GenPair gen;
std::cout << gen() << ' ' << gen() << std::endl; //
… affiche 2 4
```

23

ISIMA Les foncteurs

- Exemple : comparateur
  - principe
    - pas d'état interne
    - opérateur `()` prenant les deux objets à comparer
  - code

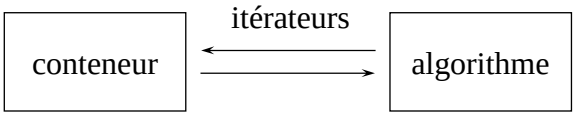
```
class Comparator
{
  public:
    Comparator () {}
    bool operator() (const A & a1, const A & a2) const
    {return (a1.val() < a2.val());}
};

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

24

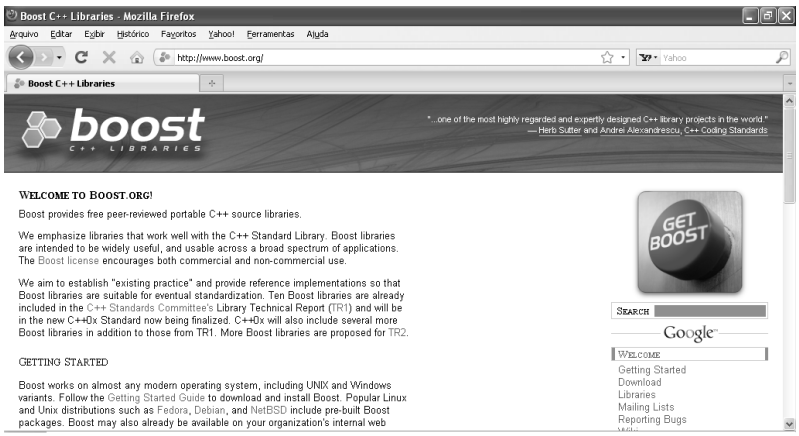
ISIMA Les foncteurs

- Manipulation globale du conteneur
  - trois entités
    - des conteneurs 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



25

ISIMA Pour aller plus loin : la librairie boost



26