Modélisation et programmation

Cours 5 : Introduction à la STL

ENSIMAG 2A – IF

Standard Template library

Concepts

La STL est une collection de conteneurs, d'algorithmes génériques et d'outils associés.

- Les conteneurs : permet de contenir des objets de n'importe quel type, et de manipuler cette liste (ajouter, enlever, accéder).
- Les itérateurs : sont une abstraction des pointeurs et permettent de parcourir les conteneurs de manière transparente.
- Les functeurs : fonctions ou objets surchargeant l'opérateur () pour affecter les opérations sur les éléments du conteneurs
- Les algorithmes génériques : liste d'algorithme que l'on peut appliquer sur les conteneurs. Il y a par exemple le trie, la recherche, la fusion, la séparation.

L'objectif de ce chapitre n'est pas d'être exhaustif, mais uniquement

- informer sur l'existence des principaux fonctionnalités.
- donner les points d'entrées pour faire prendre conscience que seule la documentation de la STL vous fournira tous les cas d'usages.

Les itérateurs

Introduction

- Les itérateurs généralisent les pointeurs.
- Ils sont utilisés pour accéder aux données du conteneur.
- Ils permettent de séparer les conteneurs et les algorithmes.
- Un algorithme peut manipuler plusieurs types de conteneurs à travers les itérateurs qui leurs sont associés.
- Un intervalle d'éléments dans un conteneur peut être représenté par deux itérateurs : *début* et *fin*. L'intervalle ainsi définit est [*début*, *fin*[.
- *fin* n'est pas inclus dans l'intervalle
  - permet d'identifier les recherches infructueuse.
  - permet de représenter l'intervalle vide.
- Les méthodes associées sont begin() et end().

Déclarations des itérateurs

- Les itérateurs de la STL sont définis à l'intérieur des classes des conteneurs

```
1   template <typename T>
2   class list
3   {
4    public :
5     class iterator
6     {
7       ...
8     };
9     ...
10  };
```

- Il est donc nécessaire de faire référence au type du conteneur pointé lorsqu'il est déclaré:

```
1   list<double>::iterator it;
```

Classification

- Un itérateur peut avoir différents comportements:
  - input_iterator: lecture seule (opérations *,==,=-,++)
  - output_iterator: écriture seule (opérations *,++)
  - forward_iterator: lecture et écriture avec un déplacement vers l'avant.
  - bidirectional_iterator: lecture et écriture avec un déplacement vers l'avant ou vers l'arrière.
  - random_iterator: lecture et écriture avec un déplacement vers l'avant ou vers l'arrière. Permet la comparaison arithmétique.
- Les itérateurs ainsi présentés sont classés du moins permissif au plus permissif.

Itérateur d'entrée

- La fonctionnalité de base de tout itérateur doit être de pouvoir accéder à la valeur pointée: l'itérateur d'entrée doit donc être déférençable

```
1    x = *it;
```

- L'itérateur d'entrée doit pouvoir également être incrémenté pour parcourir le conteneur associé

```
1    ++it;
```

- Il n'accepte pas forcément que l'on modifie la valeur pointée : c'est le caractère mutable.
- Exemple : istream_iterator pour manipuler les flux d'entrées comme cin.

Itérateur de sortie

- La fonctionnalité de base de cet itérateur est de stocker des données par assignation

```
1    *it = x;
2    it++ = x;
```

- L'itérateur de sortie n'est pas toujours déférençable.
- Il n'y a pas de distance ou de type de valeur associés avec un itérateur de sortie.
- Exemple : ostream_iterator pour manipuler les flux de sorties comme cout

Itérateur avançant

- Il permet toutes les opérations des itérateurs d'entrées et de sorties
  - Déférencement.
  - Mutabilité (modification de la valeur pointée).
  - Incrémentation de l'itérateur pour pointer vers la prochaine valeur.
  - Test d'égalités entre pointeurs et valeurs pointés.
- Il peut repasser plusieurs fois sur le même intervalle.
- Il ne pas se déplacer (incrément ou décrément) que d'une position à la fois.
- Exemple : tous les itérateurs associés aux conteneurs de la STL, les pointeurs arithmétiques.

Itérateur bidirectionnel

- Il ajoute le décrément à l'itérateur avançant.
- On le décrémente en utilisant l'opérateur --.
- Il ne pas se déplacer (incrément ou décrément) que d'une position à la fois.
- Exemple : tous les itérateurs associés aux conteneurs de la STL, les pointeurs arithmétiques.

Itérateur à accès aléatoire

- Il permet toutes les opérations des pointeurs arithmétiques C.
  - Incrémentation : ++it;
  - Décrémentation : -it;
  - Déférencement : x = *it;
  - Assignation : *it = x;
  - Egalité : it1 = it2;
  - Saut: it += 21;
  - Opérateur d'élément : it[i] = x;
- Ne peut s'utiliser qu'avec des conteneurs supportant l'accès aléatoire : vector, array, deque.

Les itérateurs de la STL

- T* ⇒ accès aléatoire
- vector<T>::iterator ⇒ accès aléatoire
- deque<T>::iterator ⇒ accès aléatoire
- array<T>::iterator ⇒ accès aléatoire
- list<T>::iterator ⇒ bidirectionnel
- set<T>::iterator ⇒ bidirectionnel
- map<T>::iterator ⇒ bidirectionnel
- multiset<T>::iterator ⇒ bidirectionnel
- multimap<T>::iterator ⇒ bidirectionnel
- hash_set<T>::iterator ⇒ bidirectionnel
- hash_map<T>::iterator ⇒ bidirectionnel

Propriétés des itérateurs de la STL

- const_iterator définit un itérateur qui n'es pas mutable

```
1  vector<Point*> points;
2  vector<Point*>\dotsconst_iterator cIt;
3  /*...*/
4  for(cIt = points.begin(), cIt != points.end(); ++cIt)
5   cIt->deplacer();
```

- Dans une méthode déclarée **const**, seul un const_iterator peut être utilisé pour manipuler les attributs

```
1  int Scene::getNbSpheres() const
2  {
3   int nbSpheres = 0;
4   vector<Objects*>::iterator it = objects_.begin();
5   for( it; it != objects_.end(); ++it)
6    if ( it->type() == "sphere")
7     nbSpheres++;
8   retrun nbSpheres;
9  }
```

Itérateurs d'insertion

- Ce sont des adaptateurs d'itérateurs.
- Ils permettent d'insérer des éléments dans un conteneur sans écraser les éléments existants.
- Ce sont des itérateurs de sorties.
- insert_iterator : insérer à un endroit précis dans le conteneur.

```
1    *i_it = x;  //conteneur.insert(it,x)
```

- back_insert_iterator : insérer directement à la fin du conteneur.

```
1    *bi_it = x;  //conteneur.push_back(x)
```

- front_insert_iterator : insérer directement au début du conteneur.

```
1    *fi_it = x;  //conteneur.push_front(x)
```

Itérateurs renversés

- Ce sont des adaptateurs d'itérateurs.
- reverse_iterator : construit à partir d'un itérateur à accès aléatoire.
- reverse_bidirectional_iterator : construit à partir d'un itérateur à bidirectionnel.
- L'opérateur ++ se comporte comme l'opérateur -- sur les itérateurs normaux.
- Il est possible de renvoyer les itérateurs inversés
  - rbegin() : pointe sur le dernier élément.
  - rend() : pointe avant le premier élément.

Les conteneurs

Les conteneurs

- Chaque conteneur est défini par une classe générique.
- On les instancie en précisant le type de leurs éléments.
- Les conteneurs offrent des méthodes qui leurs sont spécifiques, tout en partageant un petit nombre d'entre elles.
- Ce qui les distingue est leur comportement et leur structure interne.
- On les classe en trois groupes:
  1. les conteneurs séquentiels.
  2. les conteneurs associatifs.
  3. les adaptateurs.

Méthodes communes

- begin() : retourne un itérateur qui pointe au début du conteneur.
- end() : retourne un itérateur qui pointe après le dernier élément.
- size() : retourne le nombre d'éléments contenus.
- max_size() : retourne le nombre maximal d'éléments que le conteneur peut posséder.
- empty() : indique si le conteneur est vide.
- swap(Conteneur c) : échange les éléments des deux conteneurs.

Les conteneurs séquentiels

- Contiennent une suite linéaire d'objets. Ils sont de trois types.
- vector<value_type>: fourni un accès aléatoires aux objets en temps constant. L'ajout et la suppression d'éléments se réalisent en temps constants à la fin du vecteur et en temps linéaire au milieu du vecteur.
- deque<value_type>: accès aléatoire aux objets en temps constant. L'ajout et la suppression d'éléments se réalisent en temps constants à la fin et au début et en temps linéaire au milieu.
- list<value_type>: pas d'accès aléatoire. L'ajout et la suppression d'éléments se réalisent en temps constant.

Exemple d'utilisation

```
1 ➤  typedef deque<int> Conteneur;
2    Conteneur nombres;
3    nombres.push_back(3);
4    nombres.push_back(8);
5    ...
6    nombres.push_back(42);
7    ...
8    Conteneur::iterator pos = nombres.begin();
9    // supprime les nombres pairs
10   while (pos != nombres.end()) {
11    if (*pos % 2 &=& 0)
12      pos = nombres.erase(pos);
13    else
14      ++pos;
15   }
```

Constructeurs

- Les conteneurs séquentiels possèdent tous les mêmes constructeurs
  - vector<T>() : constructeur par défaut
  - vector<T>(**const** vector<T>& vect) : constructeur par recopie
  - vector<T>(taille) : construit un conteneur de taille taille
  - vector<T>(taille, valeur): construit un conteneur de taille taille avec les éléments égaux à valeur.
  - vector<T>(iterateur, iterateur) : constructeur par recopie d'un intervalle d'itérateurs.

Identification

- Simple : les éléments sont leur propre clé.
- Paire : les éléments sont de type pair<**const** key, data>.
- Trié : les clés sont triés en ordre ascendant.
- Haché : les clés sont traduites selon une table de hachage.
- Unique : un élément ne peut-être présent qu'une seule fois.
- Multiple : un élément peut-être présent plusieurs fois.

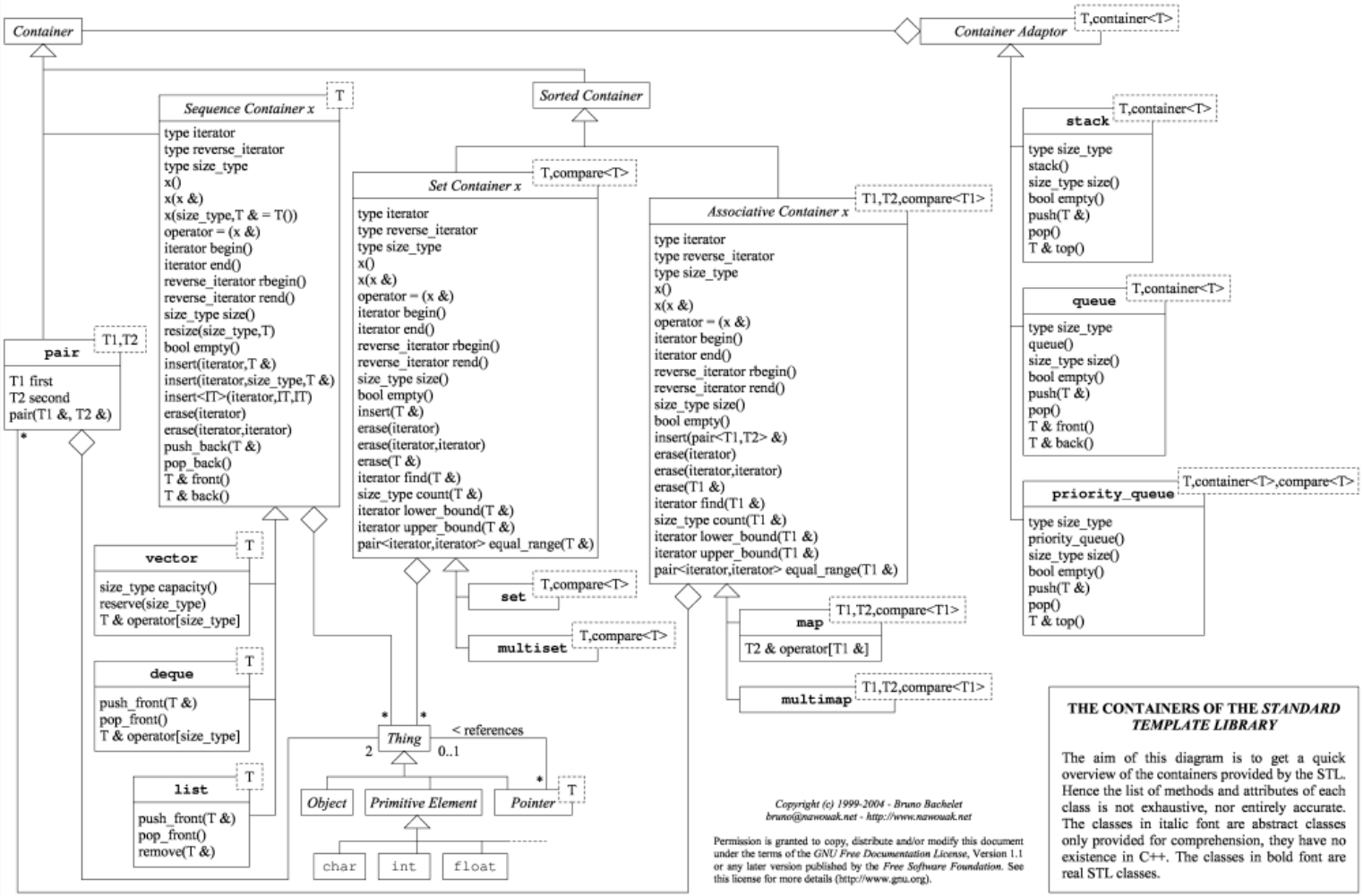
Les conteneurs associatifs

- Permettent de construire des ensembles et des tables de hachage.
- set<key_type>: contient une liste de clés unique. Permet de retrouver et de tester rapidement les clés.
- multi_set<key_type>: contient une liste de clés éventuellement dupliquées. Permet de retrouver et de tester rapidement les clés.
- map<key_type,value_type>: c'est une table de hachage. Contient une suite ordonnées d'un seul type basée sur la valeur des clés.
- multi_map<key_type,value_type>: permet d'avoir des clés dupliquées.
- hash_set<key_type>: contient une liste de clé unique classé à l'aide d'une fonction de hachage.

Caractéristiques

- L'emplacement d'un élément dépend fortement de ses caractéristiques.
- La recherche d'élément à partir d'une clé : elle peut-être la valeur elle-même ou une valeur associée.
- La clé est toujours déclaré **const**.
- La clé ne peut pas être utilisé pour insérer une valeur à un endroit précis.
- La clé détermine la position d'insertion ou de retrait.

Aperçu


Container
Container Adaptor
T,container<T>
Sequence Container x
T
type iterator
type reverse_iterator
type size_type
x()
x(x &)
x(size_type,T & = T())
operator = (x &)
iterator begin()
iterator end()
reverse_iterator rbegin()
reverse_iterator rend()
size_type size()
resize(size_type,T)
bool empty()
insert(iterator,T &)
insert(iterator,size_type,T &)
insert<IT>(iterator,IT,IT)
erase(iterator)
erase(iterator,iterator)
push_back(T &)
pop_back()
T & front()
T & back()
Sorted Container
Set Container x
T,compare<T>
type iterator
type reverse_iterator
type size_type
x()
x(x &)
operator = (x &)
iterator begin()
iterator end()
reverse_iterator rbegin()
reverse_iterator rend()
size_type size()
bool empty()
insert(T &)
erase(iterator)
erase(iterator,iterator)
erase(T &)
iterator find(T &)
size_type count(T &)
iterator lower_bound(T &)
iterator upper_bound(T &)
pair<iterator,iterator> equal_range(T &)
Associative Container x
T1,T2,compare<T1>
type iterator
type reverse_iterator
type size_type
x()
x(x &)
operator = (x &)
iterator begin()
iterator end()
reverse_iterator rbegin()
reverse_iterator rend()
size_type size()
bool empty()
insert(pair<T1,T2> &)
erase(iterator)
erase(iterator,iterator)
erase(T1 &)
iterator find(T1 &)
size_type count(T1 &)
iterator lower_bound(T1 &)
iterator upper_bound(T1 &)
pair<iterator,iterator> equal_range(T1 &)
stack
T,container<T>
type size_type
stack()
size_type size()
bool empty()
push(T &)
pop()
T & top()
queue
T,container<T>
type size_type
queue()
size_type size()
bool empty()
push(T &)
pop()
T & front()
T & back()
priority_queue
T,container<T>,compare<T>
type size_type
priority_queue()
size_type size()
bool empty()
push(T &)
pop()
T & top()
pair
T1,T2
T1 first
T2 second
pair(T1 &, T2 &)
vector
T
size_type capacity()
reserve(size_type)
T & operator[size_type]
deque
T
push_front(T &)
pop_front()
T & operator[size_type]
list
T
push_front(T &)
pop_front()
remove(T &)
set
T,compare<T>
multiset
T,compare<T>
map
T1,T2,compare<T1>
T2 & operator[T1 &]
multimap
T1,T2,compare<T1>
Thing
< references
2
0..1
Object
Primitive Element
Pointer
T
char
int
float
Copyright (c) 1999-2004 - Bruno Bachelet
bruno@nawouak.net - http://www.nawouak.net
Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.1 or any later version published by the Free Software Foundation. See this license for more details (http://www.gnu.org).
THE CONTAINERS OF THE STANDARD TEMPLATE LIBRARY
The aim of this diagram is to get a quick overview of the containers provided by the STL. Hence the list of methods and attributes of each class is not exhaustive, nor entirely accurate. The classes in italic font are abstract classes only provided for comprehension, they have no existence in C++. The classes in bold font are real STL classes.

- stack<value_type>: adapte le conteneur deque pour fournir les fonctionnalités d'une pile.
- queue<value_type>: adapte le conteneur deque pour fournir les fonctionnalités d'une file.
- priority_queue<value_type>: adapte le conteneur vector afin de gérer la priorité. L'élément du dessus et le plus grand.
- Aucun de ces adaptateurs ne permet d'itérations à travers ses éléments.

Foncteurs

Foncteurs

- C'est un objet qui peut-être appelé comme une fonction.
- Un foncteur peut être générateur f(), unaire g(x) ou binaire h(x,y).
- On peut également classifier les foncteurs en fonction du type de retour.
- Opérations arithmétiques

```
1   vector<double> V1(100), V2(100), V3(100);
2   iota(V1.begin(),V1.end(),1);
3   fill(V2.begin(),V2.end(),10);
4   transform(V1.begin(),V1.end(),V2.begin(),
5     V3.begin(), modulus<int>());
```

- Comparaisons

```
1   vector<double>::iterator first_nonzero =
2   find_if(V3.begin(), V3.end(),
3     bind2nd(not_equal_to<double>(), 15));
```

Foncteurs (2)

- Opérations logiques

```
1  vector<bool> V;
2  transform(V.begin(), V.end(), V.begin(),
3    logical_not<bool>());
```

- Opérations d'identité

```
1  map<int, double> M;
2  M[1] = 0.3; M[47] = 0.8; M[33] = 0.1;
3  transform(M.begin(), M.end(),
4    ostream_iterator<double>(cout," "),
5    select2nd<map<int, double>::value_type>());
```

- Les utilisateurs peuvent créer leurs propres foncteurs en définissant une classe.

Algorithmes

Les algorithmes génériques

- Fournissent un ensemble de fonctions qui permettent de modifier, interpréter ou analyser les données d'un conteneur.
- Certains algorithmes ne fonctionnent que sur une famille précise de conteneurs.
- Les algorithmes sont classées en 4 catégories:
  - Algorithmes *non mutants*

```
1 std::for_each(v.begin(), v.end());
```

  - Algorithmes *mutants*

```
1 std::fill(v.begin(), v.end(),0.);
```

  - Algorithmes de *tris*

```
1 std::sort(v.begin(), v.end());
```

  - Algorithmes *numériques*

```
1 sum = std::accumulate(v.begin(), v.end(),0);
```