

# Bibliothèque standard C++

## Partie 2/2

ISIMA - ZZZ - 2010



Andréa & Christophe Duhamel  
Loïc Yon  
et David Hill



### Pourquoi utiliser la bibliothèque standard ?

- Fiabilité : collection de classes largement utilisées
- Portabilité : 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

### Quelques exemples...



### Exemple 1

Exemple en C/C++

```
#include <cstdlib>
const int SIZE = 1000;

int main (int, char **)
{
    int    array [SIZE];
    int    n = 0;

    while (cin >> array[n++]);
    --n;

    qsort (array, n, sizeof(int), cmp);

    for (int i = 0; i < n; i++) cout << array[i] << "\n";
}
// Exemple inspiré de
http://www.cs.brown.edu/people/jak/proglang/cpp/stltut/tut.html
```

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

using namespace std;

int main (int, char **)
{
    vector<int> v;
    int    input;

    while (cin >> input) v.push_back (input);

    sort(v.begin(), v.end());

    int n = v.size();
    for (int i = 0; i < n; i++) cout << v[i] << "\n";
}
```

```
#include <algorithm>
#include <vector>
#include <iostream>
#include <iterator>

using namespace std;

int main (int, char **)
{
    vector<int> v;
    int    input;

    while (cin >> input) v.push_back (input);

    sort(v.begin(), v.end());

    copy (v.begin(), v.end(), ostream_iterator<int> (cout,
"\n"));
}
```

```
#include <algorithm>
#include <vector>
#include <iostream>
#include <iterator>

using namespace std;

int main (int, char **)
{
    vector<int> v;
    istream_iterator<int> start (cin);
    istream_iterator<int> end;
    back_inserter<vector<int>> > dest (v);

    copy (start, end, dest);
    sort(v.begin(), v.end());
    copy (v.begin(),
v.end(),
ostream_iterator<int>(cout, "\n"));
}
```



## Classes utilitaires de la STL

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

```
#include <string>

std::string str1("chaîne 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 \***

## Conteneurs de la STL

```
#include <iostream>
#include <string>
#include <sstream>

using namespace std;

int main (int, char **)
{
    stringstream ss;
    ss << "annee :" << 2010 << endl;

    string s = ss.str();
    s += ".";

    // au besoin s.c_str() permet d'accéder au char * interne
    clog << s;
}
```



## 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...)



## 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



## Séquences élémentaires

- 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
    - `void S::push_back (T & elt)` // Ajoute un élément au bout du conteneur
    - `void S::pop_back ()` // Retire un élément au bout du vecteur
    - `T & S::front ()` // Ref sur l'élément en tête de conteneur
    - `const T & S::front () const` // Ref const sur l'élément en tête
    - `T & S::back ()` // Ref sur l'élément en fin de conteneur
    - `const T & S::back () const` // Ref const sur l'élément en fin



## Le vecteur

- header : `<vector>`
- déclaration : `std::vector<T> v;`
- 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  $O(n)$



## Le vecteur (Exemple)

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

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

    IntVector v1;

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

    IntVector 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;
}
```



## La liste

- header : `<list>`
- déclaration : `std::list<T> l;`
- 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)



## La liste

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

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

    IntList 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;
}
```



## La file à double entrée

- « File à double entrée » (ou vecteur circulaire)
- header : `<deque>`
- déclaration : `std::deque<T> d;`
- 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)



## La file à double entrée

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

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

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

    IntDeque 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;
}
```



## Les itérateurs

- Constat
  - On 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 : sortir le pointeur du conteneur (en l'encapsulant !)
  - On souhaite parfois avoir des parcours différents (avant, arrière...)
    - solution naïve : définir la stratégie de parcours dans le conteneur
    - limite : on ne peut pas avoir (proprement) qu'un seul type de parcours
    - solution : 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
  - Notion généralisable en environnement distribué



## 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



## 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()



## Les itérateurs

- Fonctionnalités
  - FNC
  - opérateurs de comparaison != et ==
  - opérateur de déréférenciation \*
  - opérateur d'incrément (pré et post) ++



## 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);
```



## Les foncteurs

- Constat
  - On souhaite souvent appliquer un algorithme sur un conteneur
    - solution naïve : le placer en méthode du conteneur
    - limite : pollution de l'API du conteneur
    - solution : placer les algorithmes dans des classes dédiées
  - On 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
  - Une classe qui implémente un algorithme sur le conteneur
  - ou une sous-classe par variante
  - on accède aux éléments du conteneur par des itérateurs



## Les foncteurs

- Combinaison de deux principes
  - surcharge de l'opérateur ( )
    - syntaxe : type\_retour A::operator ( ) (paramètres)
    - intérêt : un objet se comporte comme une fonction
    - l'algorithme souhaité est implémenté dans l'opérateur ( )
    - les arguments de l'algorithme sont placés en argument
  - NB : peut aussi être utilisé pour remplacer l'opérateur [ ]



## Les foncteurs : exemple du comparateur

### Principe

- pas d'état interne
- opérateur ( ) prenant les deux objets à comparer

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

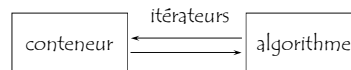
Comparator cmp;
A      a1, a2;

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



## Les foncteurs

- Manipulation globale de conteneurs
  - 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 des itérateurs



## Exemple : un générateur de nombres pairs

- Principe
  - état interne conservé par les attributs
  - opérateur ( ) sans paramètres pour la génération des nombres

```
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
```



## Quelques liens ...

- Super mémento  
<http://www.medini.org/download/stlqr/1.32/stlqr-a4.pdf>
- Documentation officielle SGI  
<http://www.sgi.com/tech/stl/>
- Représentation UML de la STL  
[http://frog.isima.fr/bruno/retrieve.htm?archive=stl\\_containers](http://frog.isima.fr/bruno/retrieve.htm?archive=stl_containers)
- Sur le C++ en général  
<http://www.cplusplus.com/reference/>



## Pour aller plus loin : la bibliothèque boost

boost  
C++ LIBRARIES

"... one of the most highly regarded and expertly designed C++ library projects in the world."  
— Herb Sutter and Andrei Alexandrescu, C++ Coding Standards

**WELCOME TO BOOST.ORG!**  
Boost provides free peer-reviewed portable C++ source libraries.  
We emphasize libraries that work well with the C++ Standard Library. Boost libraries are intended to be widely useful, and usable across a broad spectrum of applications. The Boost license encourages both commercial and non-commercial use.  
We aim to establish "best practice" and provide reference implementations so that Boost libraries are suitable for eventual standardization. Ten Boost libraries are already included in the C++ Standards Committee's Library Technical Report (TR1) and will be in the new C++0x Standard now being finalized. C++0x will also include several more Boost libraries in addition to those from TR1. More Boost libraries are proposed for TR2.

**GETTING STARTED**  
Boost works on almost any modern operating system, including UNIX and Windows variants. Follow the Getting Started Guide to download and install Boost. Popular Linux and Unix distributions such as Fedora, Debian, and Ubuntu include pre-built Boost packages. Boost may also already be available on your organization's internal web server.

**BACKGROUND**  
Read on with the introductory material to help you understand what Boost is about and to help in educating your organization about Boost.

**DOWNLOADS**  
• Version 1.44.0 (release notes)  
August 13th, 2010 17:00 GMT  
• Boost Jam 3.1.10 (release notes)  
March 24th, 2010 15:00 GMT  
More Downloads... (RSS)

**NEWS**  
• Version 1.44.0  
New Libraries: Meta State Machine, Polygon, Updated Libraries: Accumulators, Aho, Config, Fuzzy, Fuzzy, Fusion, Hash, Iterators, Math, MPL, Multi-index, Containers, Proto, Range, Spirit, Thread, TR1, Type Traits, ublas, Utility, Unit, Wave, Xpressive. Updates for Quickbook and Boostbook.  
August 13th, 2010 17:00 GMT  
• Version 1.43.0  
New Libraries: FunctionalFactory, FunctionalForward, Major Update: Range, Updated Libraries: Accumulators, Aho,...

**GET BOOST**

SEARCH Google

WELCOME  
Getting Started  
Download  
Library  
Mailing Lists  
Reporting Bugs  
Wiki

Introduction  
Community  
Development  
Support