UE 3.1 C++ : fichiers et conteneurs

Simon Rohou

2025/2026

V1.00
Supports de cours disponibles sur `www.simon-rohou.fr/cours/c++/`
Slides inspirées des cours de Thierry Vaira et de Brian Stout

ENSTA INSTITUT POLYTECHNIQUE DE PARIS

Conteneurs

Fichiers

Section 1

Conteneurs

Bibliothèque standard

C++ possède une **bibliothèque standard** (SL pour *Standard Library*) composée de :

- bibliothèque de flux
- bibliothèque standard du C
- gestion des exceptions
- ...
- Standard Template Library (STL)

Bibliothèque standard

C++ possède une **bibliothèque standard**
(SL pour *Standard Library*) composée de :

- bibliothèque de flux
- bibliothèque standard du C
- gestion des exceptions
- ...
- Standard Template Library (STL)

Nous avons déjà manipulé des éléments de la SL.
Dans ce cours, on se familiarise avec la STL, puis avec les flux.

Standard Template Library (STL)

Problème : vous possédez un tableau de n cases et, pour y insérer un nouvel élément, il vous faut décaler les éléments déjà présents.
Procédure :

1. allouer un nouveau tableau
2. y insérer les n précédents éléments, ainsi que le nouveau
3. remplacer l'ancien tableau par le nouveau

Standard Template Library (STL)

Problème : vous possédez un tableau de n cases et, pour y insérer un nouvel élément, il vous faut décaler les éléments déjà présents.
Procédure :

1. allouer un nouveau tableau
2. y insérer les n précédents éléments, ainsi que le nouveau
3. remplacer l'ancien tableau par le nouveau

La STL implémente de nombreux **types de données** de manière efficace et générique, ce qui **simplifie les opérations** sur les ensembles.

Standard Template Library (STL)

Problème : vous possédez un tableau de n cases et, pour y insérer un nouvel élément, il vous faut décaler les éléments déjà présents.
Procédure :

1. allouer un nouveau tableau
2. y insérer les n précédents éléments, ainsi que le nouveau
3. remplacer l'ancien tableau par le nouveau

La STL implémente de nombreux **types de données** de manière efficace et générique, ce qui **simplifie les opérations** sur les ensembles.

On privilégiera toujours **l'utilisation de la STL** par rapport à une implémentation manuelle : gain en efficacité, robustesse, facilité, et lisibilité pour les développeurs car **standardisation des outils**.

La STL propose des conteneurs

Conteneur (`container`) : objet contenant d'autres objets.
Un conteneur fournit :

- un moyen de gérer un **ensemble** (une **collection**) d'objets
  (au minimum ajout, suppression, parfois insertion, tri, recherche, ...)
- un accès à ces objets (par un **itérateur**)

La STL propose des conteneurs

Conteneur (`container`) : objet contenant d'autres objets.
Un conteneur fournit :

- un moyen de gérer un **ensemble** (une **collection**) d'objets
  (au minimum ajout, suppression, parfois insertion, tri, recherche, ...)
- un accès à ces objets (par un **itérateur**)

La STL propose déjà un certain nombre de conteneurs :

- tableaux (`vector`)
- listes (`list`)
- ensembles (`set`)
- piles (`stack`)
- ...

ENSTA INSTITUT POLYTECHNIQUE DE PARIS

Documentation

Vous entendrez parler de ces quatre lettres... (ou pas)

RTFM

Documentation

Vous entendrez parler de ces quatre lettres... (ou pas)

RTFM

Read

Documentation

Vous entendrez parler de ces quatre lettres... (ou pas)

RTFM

Read The

Documentation

Vous entendrez parler de ces quatre lettres... (ou pas)

RTFM

Read The Fucking

Documentation

Vous entendrez parler de ces quatre lettres... (ou pas)

RTFM

Read The Fucking Manual

Documentation

Vous entendrez parler de ces quatre lettres... (ou pas)

RTFM

Read The Fucking Manual

L'objectif de ce cours n'est pas de faire un inventaire exhaustif des possibilités offertes par la STL. Se référer plutôt à :

`http://www.cplusplus.com/reference/stl/`

Les slides suivantes présentent quelques conteneurs usuels.

ENSTA | INSTITUT POLYTECHNIQUE DE PARIS

Notion de complexité

Il est important de choisir une classe fournie par la STL qui soit **cohérente avec son besoin**.
$\Rightarrow$ certains conteneurs ont des algorithmes plus ou moins efficaces pour accéder à un élément, pour l'insérer, ...

Notion de complexité

Il est important de choisir une classe fournie par la STL qui soit **cohérente avec son besoin**.
$\Rightarrow$ certains conteneurs ont des algorithmes plus ou moins efficaces pour accéder à un élément, pour l'insérer, ...

Pour quantifier l'efficacité, on s'intéresse à son contraire : **la complexité** (notation $\mathcal{O}$).

Notion de complexité

Il est important de choisir une classe fournie par la STL qui soit **cohérente avec son besoin**.
$\Rightarrow$ certains conteneurs ont des algorithmes plus ou moins efficaces pour accéder à un élément, pour l'insérer, ...

Pour quantifier l'efficacité, on s'intéresse à son contraire : **la complexité** (notation $\mathcal{O}$).

Exemple : considérons un tableau de n cases. Si, pour insérer un nouvel élément, il nous faut décaler les éléments déjà présents, la complexité de cette insertion sera alors $\mathcal{O}(n)$:
Il faudra réaliser n opérations.

Notion de complexité

On parle de complexité par ordres de grandeur.

Considérons un conteneur de taille n et un algorithme manipulant ce conteneur (ex : recherche d'élément, insertion, tri..) :

- l'algorithme est dit linéaire (en $\mathcal{O}(n)$) si son temps de calcul est proportionnel à n (*i.e.* au nombre d'éléments)
- de la même façon il peut être :
  - instantané : $\mathcal{O}(1)$
  - logarithmique : $\mathcal{O}(\log(n))$
  - polynomial : $\mathcal{O}(n^k)$
  - exponentiel : $\mathcal{O}(e^n)$
  - ...

Notion de complexité

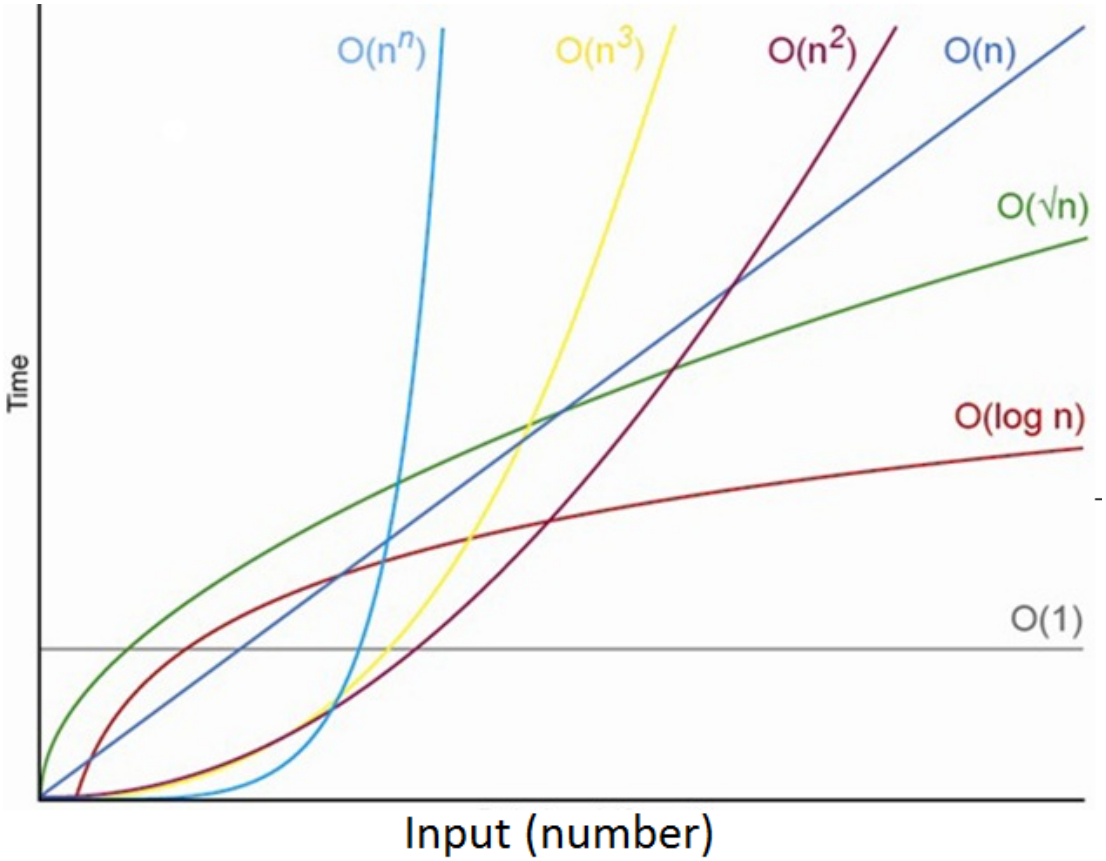

Figure – Illustration de différentes complexités en fonction du nombre d'éléments présents dans le conteneur. Ces notions conditionnent nos choix pour des conteneurs adaptés à nos besoins.

(de randerson112358)

`std::vector` : tableaux dynamiques

Classe `std::vector` :
Conteneur **séquentiel** qui encapsule un **tableau de taille dynamique**.

Les éléments du tableau :

- sont stockés de façon **contigüe**
- sont donc accessibles :
  - non seulement via les **itérateurs** (on en parle plus loin)
  - mais aussi à partir des pointeurs classiques sur un élément

`std::vector` : tableaux dynamiques

Classe `std::vector` :
Conteneur **séquentiel** qui encapsule un **tableau de taille dynamique**.

Les éléments du tableau :

- sont stockés de façon **contigüe**
- sont donc accessibles :
  - non seulement via les **itérateurs** (on en parle plus loin)
  - mais aussi à partir des pointeurs classiques sur un élément

Stockage :

- augmenter la taille d'un `vector` = réallocation mémoire pour garder la continuité = coûteux en terme de performances
- si taille connue par avance ⇒ le renseigner lors de la création

std::vector : exemple

```
#include <vector>

vector<float> v_headings; // un tableau vide de décimaux
// Un tableau de 4 tableaux d'entiers :
vector<vector<int> > v_v_numbers(4);

v_headings.push_back(2.5); // ajout d'éléments à la fin
v_headings.push_back(0.3); // (back) du tableau

float theta0 = v_headings[0]; // theta0 == 2.5
double theta1 = v_headings[1]; // cast automatique en double

// Quelques méthodes proposées avec ce container :
int size = v_headings.size(); // size == 2
bool is_empty = v_v_numbers.empty(); // is_empty == true
```

http://www.cplusplus.com/reference/vector/vector/

Parcourir un conteneur (iterator)

Itérateurs (`iterator`) :

- utilisés pour parcourir une série d'objets
- généralisation des pointeurs :
  itérateurs ⇒ objets qui pointent vers d'autres objets
- incrémenter un itérateur ⇒ désigner l'objet suivant de la série

Exemple sur un tableau (`vector`) :

```
vector<int> v(4, 100); // (100,100,100,100)
for(vector<int>::iterator it = v.begin(); it != v.end(); ++it)
  cout << *it << endl;
```

L'`iterator` est initialisé, puis incrémenté pour parcourir le tableau. Son type est fonction du type du contenu du tableau (ici : `int`).

Note : il existe aussi un itérateur inversé : `reverse_iterator`

(pour info) Les incréments en C++ : ++it ou it++ ?

En C++, on **incrémente** une variable avec ++.
Les deux syntaxes `i++` et `++i` existent.

- `++i` incrémente la valeur de `i`, et l'instruction retourne la valeur incrémentée :

```
i = 1; j = ++i;
(i == 2, j == 2)
```

- `i++` incrémente la valeur de `i`, mais l'instruction retourne la valeur originale de i avant l'incrémentation :

```
i = 1; j = i++;
(i == 2, j == 1)
```

Dans les deux cas, l'opération effectuera l'incrémentation de la variable.
Donc dans une boucle `for`, les deux solutions sont correctes.

Un itérateur fonctionne de la même manière par incrémentation.

`std::vector` : complexité

Différentes complexités selon les actions :

- en **accès** : $\mathcal{O}(1)$
- en **insertion** :
  - meilleur des cas, $\mathcal{O}(1)$ (fin de vecteur, `push_back()`)
  - pire des cas, $\mathcal{O}(n)$ (début de vecteur, `insert()`)

`std::vector` : complexité

Différentes complexités selon les actions :

- en **accès** : $\mathcal{O}(1)$
- en **insertion** :
  - meilleur des cas, $\mathcal{O}(1)$ (fin de vecteur, `push_back()`)
  - pire des cas, $\mathcal{O}(n)$ (début de vecteur, `insert()`)

Voir aussi le conteneur `std::deque` (*Double Ended QUEue*) :

- s'utilise exactement comme `std::vector`
- ne stocke pas les éléments de façon contigüe
- donc plus efficace pour des changements de taille fréquents (réallocations mémoires)

`std::list` : liste doublement chaînée

Conteneur **liste doublement chaînée**, `std::list` :

- insertion en $\mathcal{O}(1)$ en début ou fin de liste (`push_front()`, `push_back()`)
- pas de réallocation si insertion en milieu de liste, donc plus optimisé que `std::vector` (`insert()` avec itérateur)
- accès moins optimisé en milieu de structure :
  - meilleur des cas, $\mathcal{O}(1)$ (fin de liste, `front()`, `back()`)
  - cas général, $\mathcal{O}(n)$ (avec itérateur)

std::list : exemple

```
list<int> lst; // une liste vide
lst.push_back(5); // insertions en O(1) (pas de réallocation)
lst.push_front(6);
lst.push_back(1);

lst.pop_back(); // enlève le dernier élément et supprime '1'
cout << "Taille : " << lst.size() << " entiers" << endl; // = 2

// Utilisation d'un itérateur pour parcourir la liste
for(list<int>::iterator it = lst.begin(); it != lst.end(); ++it)
  cout << *it << endl; // on affiche la valeur pointée par l'
      itérateur (qui est un pointeur)

cout << "Premier element : " << lst.front() << endl; // = 6
cout << "Dernier element : " << lst.back() << endl; // = 5
```

http://www.cplusplus.com/reference/list/list/

`std::map` : dictionnaires

Conteneur **table associative** (ou dictionnaire), `std::map` :

- permet d'associer une clé à une donnée
- prend deux paramètres : type de clé et type de donnée
- insertion en $\mathcal{O}(\log(n))$
- recherche en $\mathcal{O}(\log(n))$

`http://www.cplusplus.com/reference/map/map/`

`std::map` : dictionnaires

Conteneur **table associative** (ou dictionnaire), `std::map` :

- permet d'associer une clé à une donnée
- prend deux paramètres : type de clé et type de donnée
- insertion en $\mathcal{O}(\log(n))$
- recherche en $\mathcal{O}(\log(n))$

Note : le fait d'accéder à une clé via l'opérateur `[]` insère cette clé dans la map (avec le constructeur par défaut pour la donnée).

$\Rightarrow$ l'opérateur `[]` n'est pas adapté pour vérifier si une clé est présente dans la map. Utiliser plutôt la fonction `find()`.

`http://www.cplusplus.com/reference/map/map/`

std::map : exemple

```
map<string,unsigned int> m_days_months; // la clé est une string
m_days_months["january"] = 31;
m_days_months["february"] = 28;
//...

cout << m_days_months.size() << endl; // = 12
for(map<string,unsigned int>::iterator it = m_days_months.begin()
      ; it != m_days_months.end() ; ++it)
  cout << it->first << " : " << it->second << " jours" << endl;

// Nombre de jours du mois de février (2 méthodes) :
cout << m_days_months.find("february")->second << endl;
cout << m_days_months["february"] << endl;

// Test d'existence de clé :
if(m_days_months.find("Vendémiaire") == m_days_months.end()) {
  // la clé n'existe pas
}
```

STL : d'autres conteneurs

Il existe d'autres collections d'objets :

- `std::array`, tableau de taille fixe (C++11)
- `std::stack`, LIFO (last-in first-out)
- `std::queue`, FIFO (first-in first-out)
- `std::deque`, double ended queue
- `std::multimap`, map d'éléments pouvant avoir des clés égales
- `std::pair`, paire d'éléments
- ...

`http://www.cplusplus.com/reference/stl/`

Auto

Le mot-clé `auto` existe en C++ et permet d'améliorer la lisibilité du code. Par exemple, pour itérer un `vector`, au lieu d'écrire :

```
for(std::vector<int>::const_iterator it = v.begin();
      it != v.end(); ++it)
```

On peut écrire

```
for(auto it = v.begin(); it != v.end(); ++it)
```

Auto

Le mot-clé `auto` existe en C++ et permet d'améliorer la lisibilité du code. Par exemple, pour itérer un `vector`, au lieu d'écrire :

```
for(std::vector<int>::const_iterator it = v.begin();
      it != v.end(); ++it)
```

On peut écrire

```
for(auto it = v.begin(); it != v.end(); ++it)
```

Le mot-clé `auto` s'utilise à la place d'un type (sans en être un) et sert à déduire le type d'une expression.
Il peut être associé à des qualificateurs comme les références `&` ou `const`.

Dans l'exemple ci-dessus, le compilateur comprend que `auto` désigne `std::vector<int>::const_iterator`.
Ce qui nous fait **gagner en lisibilité**.

ENSTA INSTITUT POLYTECHNIQUE DE PARIS

"Range-based for loop" avec auto

Apparu dans la version C++11, cette syntaxe permet d'améliorer encore un peu plus la lisibilité d'une boucle `for` sur des éléments d'un conteneur.

Tous les conteneurs de la librairie standard ayant les méthodes `begin`/`end` vont bénéficier de cette syntaxe.

```
// Exemple de contenu prédéfini à l'initialisation :
std::vector<int> v { 1, 2, 3, 4, 5 };

for(auto item : v) // accès aux éléments par valeur (copie)
  std::cout << item << " "; // affiche "1 2 3 4 5 "

for(auto& item : v) // accès aux éléments par référence
  item += 2; // modification directe possible
```

Le mot-clé `auto&` autorise la modification des éléments du conteneur.

Section 2

Fichiers

Généralités

Règle générale pour **créer un fichier** :

- ouvrir le fichier en **écriture**
- y écrire des données
- le fermer

Règle générale pour **lire des données** :

- l'ouvrir en **lecture**
- lire les données en provenance du fichier
- le fermer

Généralités

Règle générale pour **créer un fichier** :

- ouvrir le fichier en **écriture**
- y écrire des données
- le fermer

Règle générale pour **lire des données** :

- l'ouvrir en **lecture**
- lire les données en provenance du fichier
- le fermer

Il existe **deux types** de fichiers :

- les fichiers **textes** (lisibles *via* un simple éditeur de texte)
- les fichiers **binaires** (copie bit à bit de variables)

Fonctionnalités proposées en C++

cstdio ou fstream :
Deux bibliothèques standards pour manipuler les fichiers

- `cstdio`, qui provient du langage C
- `fstream`, approche C++

⇒ On s'intéresse à la **bibliothèque fstream** :

- la classe `std::ofstream`, pour l'écriture de fichiers (`o`, *output*)
- la classe `std::ifstream`, pour la lecture de fichiers (`i`, *input*)

La classe ofstream par l'exemple (écriture de fichier)

```
#include <fstream>
// ...

string filename = "file.txt";
ofstream f(filename); // tentative d'ouverture (constructeur)

if(!f.is_open())
  cout << "Erreur d'ouverture de " << filename << endl;

else
{
  float a = 3.1415;
  f << a << " " << 42; // écriture de variables dans le flux
  string str = "ENSTA Bzh";
  f << endl << str; // saut de ligne dans le fichier
}

f.close(); // fermeture du fichier
```

La classe ifstream par l'exemple (lecture de fichier)

```
#include <fstream>
// ...

string filename = "file.txt";
ifstream f(filename); // tentative d'ouverture (constructeur)

if(!f.is_open())
  cout << "Erreur d'ouverture de " << filename << endl;

else
{
  float a; int b;
  f >> a >> b; // lecture de variables depuis le flux
  string str;
  getline(f, str); // lecture d'une ligne complète
}

f.close(); // fermeture du fichier
```

Sérialisation et flux

La **sérialisation** consiste à transmettre un **objet** à travers un **flux**.
Elle permet par exemple d'enregistrer une instance de classe dans un fichier, ou de la transmettre par web-service.

Sérialisation et flux

La **sérialisation** consiste à transmettre un **objet** à travers un **flux**. Elle permet par exemple d'enregistrer une instance de classe dans un fichier, ou de la transmettre par web-service.

Cette sérialisation transmet des données vers (`ostream`) ou depuis (`istream`) un **flux**. Exemple de support de flux :

- votre terminal
- fichiers (classes `ifstream`, `ofstream`)
- chaînes de caractères (classe `istringstream`)

Sérialisation et flux

La **sérialisation** consiste à transmettre un **objet** à travers un **flux**. Elle permet par exemple d'enregistrer une instance de classe dans un fichier, ou de la transmettre par web-service.

Cette sérialisation transmet des données vers (`ostream`) ou depuis (`istream`) un **flux**. Exemple de support de flux :

- votre terminal
- fichiers (classes `ifstream`, `ofstream`)
- chaînes de caractères (classe `istringstream`)

Exporter un objet sous forme de flux permet donc de le **sérialiser dans un fichier** (avec `ofstream`).

Note : les manipulateurs tels que `endl` sont utilisables dans tous les flux.

Sérialiser un objet

Pour sérialiser un objet d'une classe `Robot`, il suffit de **définir les opérateurs de flux**.

Extrait de `Robot.h` :

```
friend ostream& operator<<(ostream& stream, const Robot& r);
friend istream& operator>>(istream& stream, Robot& r);
```

Extrait de `Robot.cpp` :

```
ostream& operator<<(ostream& stream, const Robot& r) {
  stream << r.name(); // r est const puisqu'on le lit
  return stream;
}

istream& operator>>(istream& stream, Robot& r) {
  stream >> r.m_name; // r n'est pas const car modifié
  return stream;
}
```

Sérialiser un objet

Cela simplifie la manipulation d'objets à travers les fichiers :

```
#include <fstream>
// ...

Robot robot("brave"); // robot.m_name = "brave";
string filename = "file.txt";

ofstream ofst(filename); // ouverture en écriture
// ..tests d'ouverture.., puis :
ofst << robot; // enregistrement de l'objet robot dans le flux
ofst.close(); // fermeture du fichier

ifstream ifst(filename); // ouverture en lecture
// ..tests d'ouverture.., puis :
ifst >> robot; // lecture de l'objet robot depuis le flux
ifst.close(); // fermeture du fichier
```

ENSTA INSTITUT POLYTECHNIQUE DE PARIS