

Initiation aux concepts objets en C++

Du C au C++ - Découverte du paradigme objet

Sommaire

- I. Du C au C++ : concepts, diagramme de classes et vocabulaire
- II. Principales différences entre le C et le C++
- III. Nouveautés avec C++11 / C++14
- IV. Classes et objets
- V. Namespaces et templates
- VI. Exceptions
- VII. Conteneurs : vecteurs, piles et files**
- VIII. Pointeurs, références et surcharges
- IX. Héritage et polymorphisme

Sommaire (détail)

- IV. Conteneurs de la STL
 - A. Vecteur
 - B. Liste
 - C. Set
 - D. Deque
 - E. Pile et file
 - F. Map
 - G. Table de hachage
 - H. Choix du bon conteneur

Objectifs

- Découvrir les différents conteneurs de la STL
- Comment choisir le « bon » conteneur ?
- Savoir utiliser les différents conteneurs en C++

Vecteurs (1/7)

- Tableau dynamique (extensible à « l'infini »)
- Données contiguës en mémoire
- Pas de « trous » dans le tableau
- Accès par indice
- **1^{er} élément à l'indice zéro !!**

Vecteurs – Complexité (2/7)

- $O(1)$ pour l'ajout en queue (push_back)
- $O(1)$ pour suppression en queue (pop_back)
- $O(1)$ pour l'accès au i ème élément
- $O(n)$ pour chercher un élément

Vecteurs (3/7)

```
std::vector<TYPE> myVector;
```

1 2 3

- Type de container (appartient à la lib « std »)
- Nature des éléments dans vecteur (int, float, pointeurs...)
- Nom de la variable

Vecteurs (4/7)

- Inclure <vector>
- Fonctions d'accès aux données :
 - opérateur [] (par indice)
 - front()
 - back()
- Fonctions de modification :
 - push_back()
 - pop_back()
 - insert()
 - erase()

Vecteurs (5/7)

- Parcourir un vecteur :

```
std::vector<float>::iterator it;
std::cout << "Browsing vector..." << std::endl;
for (it=grades.begin(); it!=grades.end(); it++)
{
    std::cout << "Elem = " << *it << std::endl;
}
```

- `grades.begin()` = itérateur sur le premier élément
- `grades.end()` = itérateur sur « l'après dernier élément »

Vecteurs (6/7)

- NOUVEAUTE en C++11 - Parcourir un vecteur

```
// we prefer "const auto&" instead of "auto"
// but it does not matter :)
for (auto elem : grades)
{
    std::cout << "Elem = " << elem << std::endl;
}
```

- Mot clé « auto » qui remplace les itérateurs !!
- Nécessite le flag de compilation : « -std=c++11 »

Vecteurs (7/7)

```
std::vector<float> grades;
grades.push_back(10.5); // add 10.5
grades.push_back(5.24); // add 5.24
grades.push_back(18.75); // add 18.75
grades.push_back(10.0); // add 10.0
std::cout << "First elem : " << grades.front() << std::endl;
std::cout << "Last elem : " << grades.back() << std::endl;
std::cout << "Elem at position 2 : " << grades[2] << std::endl;
grades.erase(grades.begin() + 2); // erase 3rd elem !
std::cout << "New elem at position 2 : " << grades[2] << std::endl;
std::cout << std::endl;

std::vector<float>::iterator it;
std::cout << "Browsing vector..." << std::endl;
for (it=grades.begin(); it!=grades.end(); it++)
{
    std::cout << "Elem = " << *it << std::endl;
}
```

Listes (1/5)

- Listes doublement chainées
- Données non contiguës en mémoire
- Rapide pour les insertions
- Accès par itérateur ! (pas par indice Θ)

Listes – Complexité (2/5)

- $O(1)$ l'ajout en tête/queue(`push_front`/`push_back`)
- $O(1)$ suppression en tête/queue (`pop_front`/`pop_back`)
- $O(i)$ pour l'accès au i ème élément à partir du début
- $O(n)$ pour rechercher un élément
- $O(1)$ pour l'insertion au milieu d'un élément

Listes (3/5)

```
std::Container<TYPE> myVector;
```

The diagram shows the declaration of a vector variable. It consists of three main parts: 'std::Container' (highlighted in green), '<TYPE>' (highlighted in blue), and 'myVector;' (highlighted in purple). Below each part is a horizontal bar of the same color, and below each bar is a number: 1 under 'std::Container', 2 under '<TYPE>', and 3 under 'myVector;'.

- Nom du container (appartient à la lib « std »)
- Nature des éléments dans vecteur (int, float, pointeurs...)
- Nom de la variable

Listes (4/5)

- Include <list>
- Fonctions d'accès aux données :
 - front()
 - back()
- Fonctions de modification :
 - push_back()
 - pop_back()
 - insert()
 - erase()

Listes (5/5)

- Parcourir une liste C++11 :

```
std::list<float> lst;
for (unsigned int i=0; i<1000; ++i)
{
    lst.push_back(i);
}
for (const auto& elem : lst)
{
    std::cout << elem << std::endl;
}
```

- Utilisation boucle de type « foreach »
- Plus besoin d'itérateur !

Set (1/5)

Copyright CodeAnalysis

- Clés uniques (pas de doublons)
- Eléments ordonnés
- Données non contiguës en mémoire
- Implémentation par arbre binaire de recherche

Set – Complexité (2/5)

Copyright CodeAnalysis

- $O(\log(n))$ pour l'insertion
- $O(1)$ pour la suppression (temps amorti) par position
- $O(\log(n))$ pour la suppression par valeur
- $O(\log(n))$ pour recherche d'un élément

Set (3/5)

```
std::Container<TYPE> myVector;
```

The diagram shows the declaration of a vector variable. It consists of three main parts: 'std::Container' (highlighted with a green underline), '<TYPE>' (highlighted with a blue underline), and 'myVector;' (highlighted with a purple underline). Below each part is a corresponding number: '1' under 'std::Container', '2' under '<TYPE>', and '3' under 'myVector;'.

- Nom du container (appartient à la lib « std »)
- Nature des éléments dans vecteur (int, float, pointeurs...)
- Nom de la variable

Set (4/5)

- Include <set>
- Fonctions d'accès aux données :
 - `find()`
 - `count()` – *Retourne 0 ou 1 élément*
- Fonctions de modification :
 - `insert()`
 - `erase()`

Set (5/5)

- Exemple :

```
std::set<float> mySet;
for (unsigned int i=0; i<1000; ++i)
{
    mySet.insert(i);
}

for (const auto& elem : mySet)
{
    std::cout << elem << std::endl;
}

if (mySet.find(500) != mySet.end()) {
    std::cout << "Value 500 has been found !" << std::endl;
}
else {
    std::cout << "Value 500 not found !" << std::endl;
}
```

Deque (1/5)

- Généralisation d'une queue « Double Ended Queue »
- Données non contiguës en mémoire
- Rapide pour les ajouts en début et fin
- Interface similaire au vecteur
- Accès possible par indice !

Deque – Complexité (2/5)

- $O(1)$ l'ajout en tête/queue(`push_front`/`push_back`)
- $O(1)$ suppression en tête/queue (`pop_front`/`pop_back`)
- $O(1)$ pour l'accès au i ème élément
- $O(n)$ pour rechercher un élément
- $>O(n)$ pour l'insertion

Deque (3/5)

```
std::Container<TYPE> myVector;
```

The diagram shows the declaration of a deque container. It consists of three main parts: 'std::Container' (part 1), '<TYPE>' (part 2), and 'myVector;' (part 3). Each part is highlighted with a horizontal bar: a green bar under 'std::Container', a blue bar under '<TYPE>', and a purple bar under 'myVector;'. Below each bar is a corresponding number: '1' under the green bar, '2' under the blue bar, and '3' under the purple bar.

- Nom du container (appartient à la lib « std »)
- Nature des éléments dans vecteur (int, float, pointeurs...)
- Nom de la variable

Deque (4/5)

- Inclure <deque>
- Fonctions d'accès aux données :
 - front()
 - back()
- Fonctions de modification :
 - push_back() / pop_back()
 - push_front() / pop_front()
 - insert()
 - erase()

Deque (5/5)

- Parcourir un deque C++11 :

```
// Allocation of 1000 float
std::deque<float> lst(1000, 0);
for (unsigned int i=0; i<1000; ++i)
{
    lst[i] = i;
}
for (const auto& elem : lst)
{
    std::cout << elem << std::endl;
}
```

- Plus besoin d'itérateur !

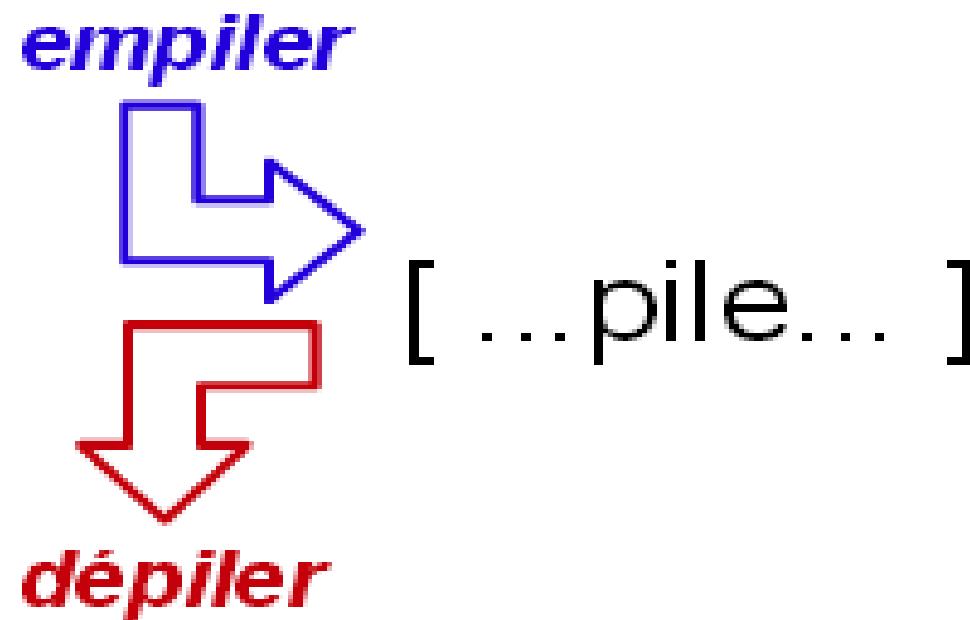
Piles (1/7)

Copyright CodeAnalysis

- LIFO : « Last In First Out »
- Données non contiguës en mémoire
- Utilise un « deque » par défaut
- Accès uniquement au sommet de la pile
- Deux opérations autorisées (empiler et dépiler)

Piles (2/7)

- Principe général :



Piles (3/7)

- Avec un exemple :

Pile Vide	[]
empiler A	[A]
empiler B	[B A]
empiler C	[C B A]
dépiler → C	[B A]
empiler D	[D B A]
dépiler → D	[B A]
dépiler → B	[A]
dépiler → A	[]

Piles (4/7)

```
std::stack<TYPE> grades;
```



- Type de container (appartient à la lib « std »)
- Nature des éléments dans vecteur (int, float, pointeurs...)
- Nom de la variable

Piles (5/7)

- Fonctions d'accès aux données :
 - `top()`
- Fonctions de modification :
 - `push()`
 - `pop()`

Piles (6/7)

Copyright CodeAnalysis

- Parcourir une pile :
 - **Pas de parcours de pile !!**
- Pas de notion d'itérateur pour une pile
- **Rappel : Accès UNIQUEMENT au sommet**

Piles (7/7)

```
std::stack<int> myStack;
for (int i=1; i<=10; i++)
{
    myStack.push(i);
}
std::cout << "Top = " << myStack.top() << std::endl;
myStack.pop();
std::cout << "New top = " << myStack.top() << std::endl;
for (int i=1; i<=9; i++)
{
    myStack.pop();
}
if (myStack.empty()) std::cout << "Stack empty !" << std::endl;
else std::cout << "Left " << myStack.size() << " elem" << std::endl;
```

Files (1/7)

- FIFO : « First In First Out »
- Données non contiguës en mémoire
- Utilise un « deque » par défaut
- Accès uniquement à la tête de la file
- Deux opérations autorisées (enfiler et défiler)

Files (2/7)

- Principe général :



Files (3/7)

- Avec un exemple :

File Vide	[]
enfiler A	[A]
enfiler B	[A B]
enfiler C	[A B C]
défiler → A	[B C]
enfiler D	[B C D]
défiler → B	[C D]
défiler → C	[D]
défiler → D	[]

Files (4/7)

```
std::queue<TYPE> myQueue;
```

1 2 3

- Type de container (appartient à la lib « std »)
- Nature des éléments dans vecteur (int, float, pointeurs...)
- Nom de la variable

Files (5/7)

- Fonctions d'accès aux données :
 - front()
- Fonctions de modification :
 - push()
 - pop()

Files (6/7)

- Parcourir une file :
 - **Pas de parcours de file !!**
- Pas de notion d'itérateur pour une file
- **Rappel : Accès UNIQUEMENT au premier élément**

Files (7/7)

```
std::queue<int> myQueue;
for (int i=1; i<=NB_ELEM; i++)
{
    myQueue.push(i);
}
std::cout << "Front = " << myQueue.front() << std::endl;
myQueue.pop();
std::cout << "New front = " << myQueue.front() << std::endl;
for (int i=1; i<=NB_ELEM-1; i++)
{
    myQueue.pop();
}
if (myQueue.empty()) std::cout << "Queue empty !" << std::endl;
else std::cout << "Left " << myQueue.size() << " elem" << std::endl;
```

Map (1/5)

- Clés uniques (pas de doublons)
- Eléments ordonnés
- Tableaux associatifs (clé => valeur)
- Implémentation par arbre binaire de recherche

Map – Complexité (2/5)

- $O(\log(n))$ pour l'insertion
- $O(1)$ pour la suppression (temps amorti) par position
- $O(\log(n))$ pour la suppression par valeur
- $O(\log(n))$ pour recherche d'un élément

Copyright CodeAnalysis

Map (3/5)

```
std::Container<TYPE> myVector;
```

The diagram shows the declaration of a vector variable. It consists of three main parts: 'std::Container' (highlighted with a green underline), '<TYPE>' (highlighted with a blue underline), and 'myVector;' (highlighted with a purple underline). Below each part is a corresponding number: '1' under 'std::Container', '2' under '<TYPE>', and '3' under 'myVector;'.

- Nom du container (appartient à la lib « std »)
- Nature des éléments dans vecteur (int, float, pointeurs...)
- Nom de la variable

Map (4/5)

- Include <map>
- Fonctions d'accès aux données :
 - operator[]
 - count() – Retourne *0 ou 1 élément*
- Fonctions de modification :
 - insert()
 - erase()

Map (5/5)

- Exemple :

```
std::map<int, std::string> myMap;
for (unsigned int i=0; i<1000; ++i)
{
    std::ostringstream oss;
    oss << "Hello #" << i;
    myMap[i] = oss.str();
}

for (const auto& elem : myMap)
{
    std::cout << elem.first << " => "
          << elem.second << std::endl;
}

auto it = myMap.find(500);
if (it != myMap.end()) {
    std::cout << "Key 500 has been found and value equals = "
          << it->second << std::endl;
}
else {
    std::cout << "Key 500 not found !" << std::endl;
}
```

Hash table (1/5)

- Au choix : clés uniques ou multiples
- Eléments non ordonnés
- Recherche d'un élément très rapide
- Nécessite une fonction de hash
- Implémentation par listes chainées (« buckets »)

Hash table – Complexité (2/5)

- $O(1)$ pour l'insertion (en moyenne)
- $O(1)$ pour la suppression (en moyenne)
- $O(1)$ pour rechercher un élément (en moyenne)

Hash table (3/5)

```
std::unordered_set<Value, HashFunc, EqualityFunc> myHashTable;
std::unordered_map<Key, Value, HashFunc, EqualityFunc> myHashTable2;
```

- Set ou map
- Hashfunc = fonction de hachage
EqualityFunc = prédicat d'égalité

Hash table (4/5)

- Inclure <unordered_set> ou <unordered_map>
- Fonctions d'accès aux données :
 - `find()`
 - `count()`
- Fonctions de modification :
 - `insert()`
 - `erase()`

Hash table (5/5)

- Exemple :

```
std::unordered_set<std::string> myHashTable;
for (unsigned int i=0; i<200; ++i)
{
    std::ostringstream oss;
    oss << "Hello #" << i;
    auto ret = myHashTable.insert(oss.str());
    if (!ret.second)
    {
        std::cout << "\t" << "Error adding " << oss.str() << std::endl;
    }
}

for (unsigned i=0; i<myHashTable.bucket_count(); ++i)
{
    std::cout << "bucket #" << i << " contains:";
    for (auto it = myHashTable.begin(i); it!=myHashTable.end(i); ++it)
        std::cout << " " << *it;
    std::cout << std::endl;
}
```

Hash function - (1/4)

- Calcule une « empreinte » d'un objet
- Permet de ranger les éléments dans les « buckets »
- **Doit être performante et rapide !**
 - Performante : éviter les collisions
 - Rapide : temps de calcul rapide

Hash function - (2/4)

- Exemple fonction très peu performante :

```
struct Hash
{
    std::size_t operator() (const std::string& value) const
    {
        return 0;
    }
};
```

- Dégénération de la table de hachage !

Hash function - (3/4)

- Exemple fonction peu performante :

```
struct Hash2
{
    std::size_t operator() (const std::string& value) const
    {
        std::size_t hash = 0;
        for (unsigned int i=0; i<value.size(); ++i)
        {
            hash += value[i];
        }
        return hash;
    }
};
```

Hash function - (4/4)

- Exemple fonction performante :

```
struct GoodHashFunction
{
    // magic numbers from http://www.isthe.com/chongo/tech/comp/fnv/
    static const size_t InitialFNV = 2166136261U;
    static const size_t FNVMultiple = 16777619;

    std::size_t operator()(const std::string& value) const
    {
        size_t hash = InitialFNV;
        for(size_t i = 0; i < value.size(); i++)
        {
            hash = hash ^ (value[i]);    // xor the low 8 bits
            hash = hash * FNVMultiple; // multiply by the magic number
        }
        return hash;
    }
};
```

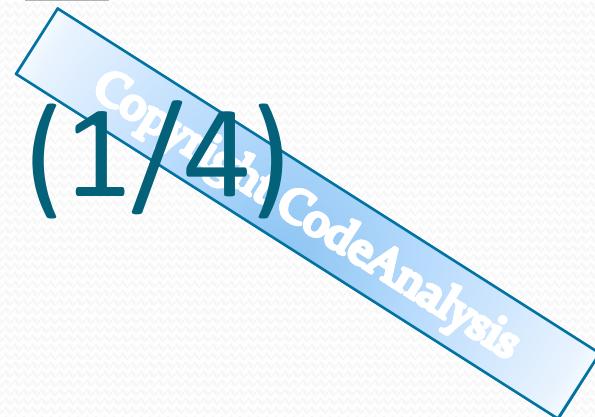
Autres conteneurs

- Tableau statique (Array)
 - Depuis C++11
- Arbres (pas natif en C++) ...
 - Arbres binaires simples
 - Arbres binaires de recherche
 - Arbres n-aire

=> BOOST !

Choix du conteneur - (1/4)

- Se poser les bonnes questions :
 - Ordre important ?
 - Eléments associés à des clés ?
 - Eléments ordonnés ?
 - Doublons autorisés ?
 - Fusions fréquentes ?
 - Besoin de performance ou économie mémoire ?
- Passons à la pratique ! ☺



Choix du conteneur - (2/4)

- Que choisir pour stocker des entiers avec fusion fréquentes ?
 - *Liste car efficace pour les fusions*
- Que choisir pour stocker une grande quantité d'objet sans autoriser les doublons ?
 - *Set ou unordered_set car pas de doublons*
- Que choisir pour associer chaque valeur à une clé non unique ?
 - *Multi_map puisque les clés ne sont pas uniques*

Copyleft CodeAnalysis

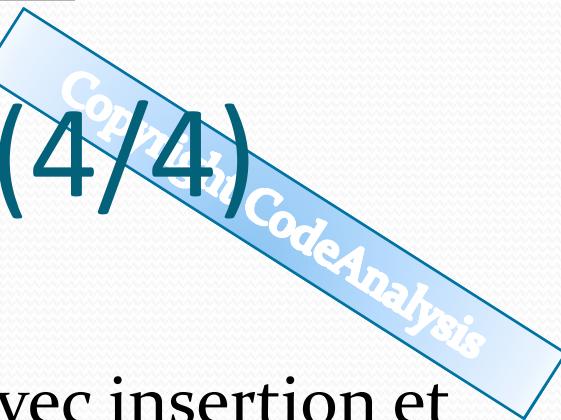
Choix du conteneur - (3/4)

- Que choisir pour stocker une grosse quantité d'entiers avec un accès rapide au *ième* élément ?
 - *Vecteur pour l'accès rapide à un élément*
- Que choisir pour stocker des objets suivant un ordre précis avec accès uniquement au premier ?
 - *Priority_queue / stack / queue*
- Que choisir pour associer une valeur à une clé unique de manière très rapide ?
 - *Vecteur avec pré-allocation sinon table de hachage...*



Choix du conteneur - (4/4)

- Que choisir pour stocker des objets avec insertion et suppression en tête et en queue fréquentes ?
 - *Deque car accès très rapide aux extrémités*



Comparatif - (1/3)

- Propriétés des conteneurs

	Accès rapide [] ?	Doublon autorisé ?	Associatif ?	Ordonné ?
<i>Vector</i>	OUI	OUI	NON	NON
<i>Deque</i>	OUI	OUI	NON	NON
<i>List</i>	NON	OUI	NON	NON
<i>Set</i>	NON	NON	NON	OUI
<i>Map</i>	NON	NON	OUI	OUI
<i>Unordered_set</i>	OUI	NON	NON	NON
<i>Unordered_map</i>	OUI	NON	OUI	NON

Comparatif - (2/3)

- Performance des conteneurs (en moyenne)

	Accès au ième	Insertion en tête	Insertion en queue	Insertion	Recherche
<i>Vector</i>	O(1)	O(n)	O(1)	O(n)	O(n)
<i>Deque</i>	O(1)	O(1)	O(1)	O(n)	O(n)
<i>List</i>	O(n)	O(1)	O(1)	O(n)	O(n)
<i>Set</i>	NC	NC	NC	O(log(n))	O(log(n))
<i>Map</i>	NC	NC	NC	O(log(n))	O(log(n))
<i>Unordered_set</i>	NC	NC	NC	O(log(n))	O(1)*
<i>Unordered_map</i>	NC	NC	NC	O(log(n))	O(1)*

* Temps amorti

Comparatif - (3/3)

- Performance des conteneurs (en moyenne)

	Suppression en tête	Suppression en queue	Suppression	Tri
<i>Vector</i>	O(n)	O(1)	O(n)	O(n.log(n))
<i>Deque</i>	O(1)	O(1)	O(n)	O(n.log(n))
<i>List</i>	O(1)	O(1)	O(n)	O(n.log(n))
<i>Set</i>	NC	NC	O(log(n))**	O(1)
<i>Map</i>	NC	NC	O(log(n))**	O(1)
<i>Unordered_set</i>	NC	NC	O(1)*	NC
<i>Unordered_map</i>	NC	NC	O(1)*	NC

* Temps amorti

** Par valeur sinon O(1) si par position

Prochaine séance

- Pointeurs, références et surcharges

Copyright CodeAnalysis