

Advanced C++ programming

♦ Introduction to C++

- C++: from C and beyond
- Classes, objects and lifetime (vs. JAVA)
- Oriented-Object Programming (inheritance, polymorphism)

♦ Memory management & object manipulation

- References, operators, « copy » object construction
- « move » object construction, lambda functions

♦ Template vs OO programming

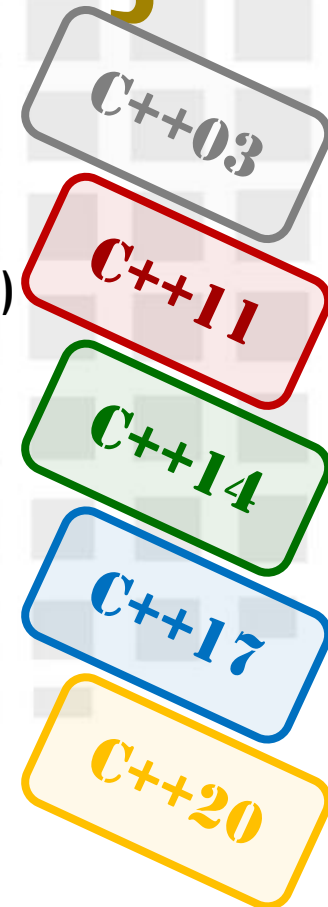
- Template functions and classes

♦ The Standard Template Library



- Containers, iterators and algorithms
- Using sequence & associative containers ...

♦ Smart pointers (STL & Boost)



Introduction à la STL

C++ 11 / C++ 14

`tr1::` (si C++0x)

Smart pointers
(`shared_ptr`)

Générateurs aléatoires
(`poisson`, `gamma`, ...)

Mathématiques
(`bessel`, `laguerre`, ...)

Expressions
régulières

Gestion des entités
(`binders`, `wrapper`, ...)

Nouveaux types
(`tuple`, `hash table`, ...)

Meta-programming
(`type_traits`, ...)

C++ 98

Internationalisation
(`wchar_t`, `wchar_ts`)

C++
(Template)

C++
(Objet)

C89

Standard Template
Library (STL)

Mathématiques
(`complex`, `valarray`)

Gestion des flots
(`iostream`)

Hierarchisation
des exceptions

Graph Library

Meta Programming Library (MPL)

Boost



La « Standard Template Library » (STL)



- **Objectifs**

- Répondre à certains besoins non traités en C++ de base
 - Ranger de manière structurée les ensembles de données manipulées
- Bibliothèque hautement générique
 - Utilisation des « templates »
- Priorité aux « algorithmes »
 - Syntaxe : un même nom de fonction par algorithme
 - Application sur des objets de types différents

- **Implémentation**

- Uniquement dans des fichiers « .h » \Rightarrow un `#include` suffit
- Dans le namespace **std**



La « Standard Template Library » (STL)

- Introduction de nouvelles classes

- « Conteneurs »

- structures proposant un stockage adapté (`vector`, `list`, `map`, ...)
 - temps d'accès connus et donc maîtrisés

- « Algorithmes »

- manipulations classiques des données stockées (`for_each`, `reverse`, ...)
 - indépendance vis-à-vis du type des données stockées

- « Itérateurs »

- généralisation de la notion de pointeurs
 - accès aux données stockées
 - manipulation générique des données (abstraction de leur type)



Les conteneurs (1/5)

- Enjeux

- Stockage **structuré** d'ensemble de données
 - Conteneurs ordonnés
 - Base : vecteurs (`vector`, `string`), files (`deque`), listes (`list`)
 - Spécialisation (files spécifiques) : FIFO (`queue`), LIFO (`stack`), avec priorité, ...
 - Conteneurs associatifs
 - tables (`map`, `multimap`), ensembles (`set`, `multiset`)
- Performances d'accès en lecture / écriture **maitrisées**
 - complexité connue des algorithmes d'accès aux données
- Abstractions pour l'accès aux données
 - opérateurs « génériques » de lecture, d'insertion, de suppression
 - encapsulation de la notion de pointeur
 - ⇒ le concept d'**itérateurs**



Les conteneurs (2/5)

• Exemple : le conteneur « vector »

- Encapsulation des tableaux dynamiques (taille variable)
 - STL gère la stratégie de réallocation lors de l'ajout d'un élément
- Accès direct aux éléments du conteneur

```
#include <vector>
```

```
// Déclaration d'un vecteur d'entiers
std::vector<int> v ;

// Insertion d'un 1er entier
v.push_back (5) ;
std::cout << v[0] ;

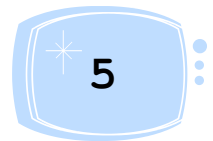
// Déclaration d'une variable entière
int k = 3 ;

// Insertion d'une copie de l'entier 'k'
v.push_back (k) ;
std::cout << v[1] ;

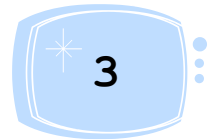
k = 50 ;
std::cout << v[1] ;

// Modifie la valeur du 2nd élément de v
v[1] = 10 ;
std::cout << v[1] ;
```

v = [5]



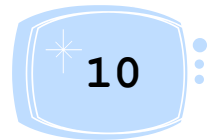
v = [5, 3]



?



v = [5, 10]



Les conteneurs (3/5)

• Méthodes communes à tous les conteneurs

- `bool empty()` : vrai si le conteneur est vide
- `iterator begin()`, `iterator end()` : itérateur sur le début ou la fin du conteneur

• Méthodes communes à presque tous

sauf
array

- `insert()`, `erase()` : insère / supprime un nouvel élément
- `void clear()` : le conteneur doit être vidé

sauf
array

- Appel du destructeur de chaque objet stocké
- Désallocation de la mémoire utilisée

sauf
forward_list

- `int size()` : retourne le nombre d'éléments du conteneur
- ...



Les conteneurs (4/5)

• Insertion d'objets dans n'importe quel conteneur

• Mécanisme d'insertion d'un objet O dans un conteneur

- Une copie ou une appropriation O' de l'objet O est réalisée
- Ce nouvel objet O' est stocké dans le conteneur

⇒ nécessité de prévoir un **constructeur par copie** ou **par appropriation** pour la classe de O si ses constructeurs par défaut ne suffisent pas.

• Attention au phénomène de « slicing »

```
class Shape {
    double x , y ;
}

class Circle : public Shape {
    double radius ;
}
```

```
Container<Shape> C ;
```

```
Shape S ;
Circle R ;
```

```
// Insertion d'un objet de type Shape : Ok
C.push_back (S) ;
```

```
// L'objet de type Circle est tronqué en Shape
C.push_back (R) ;
```

Le container C
ne peut stocker
que des Shape
(pas de place
pour des types dérivés
donc plus spacieux)



Les conteneurs (5/5)

• Insertion d'objets dans n'importe quel conteneur

• Mécanisme d'insertion d'un objet O dans un conteneur

- Une copie ou une appropriation O' de l'objet O est réalisée
- Ce nouvel objet O' est stocké dans le conteneur

⇒ nécessité de prévoir un **constructeur par copie** ou **par appropriation** pour la classe de O si ses constructeurs par défaut ne suffissent pas.

• Sélection de la meilleure technique d'insertion



std::vector::push_back

```
void push_back (const T& value) ;  
void push_back (T&& value) ;
```

```
Container<Circle> C ;  
  
Circle C1 ;  
Circle C2 ;  
  
// Insertion d'un Circle (par copie)  
C.push_back (C1) ;  
  
// Insertion d'un Circle (par appropriation)  
C.push_back (std::move(C2)) ;
```



Les itérateurs (1/4)

• Définition d'un itérateur `it`

- `it` « pointe » sur un élément d'un conteneur
 - Déréférencement pour accéder à la donnée pointée : `*it`
 - Passage à l'élément suivant dans le conteneur : `++it`



`it` est donc lié par nature

- à un conteneur particulier (type des déplacements)
 - au type des données manipulées (taille des éléments)
- Chaque conteneur fournit le type de ses itérateurs `it`
 - au travers des types-membre `iterator` et `const_iterator`

```
// Définition d'un conteneur de Shape
Container<Shape> C ;
```

```
// Récupération du type de l'itérateur associé au conteneur C :
// it permet de modifier l'objet Shape *it
Container<Shape>::iterator it ;
```


```
// Récupération du type de l'itérateur const_iterator :
// it1 ne permet pas la modification de *it1
Container<Shape>::const_iterator it1 ;
```



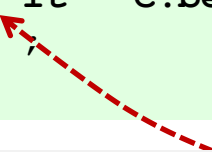
Les itérateurs (2/4)

- Parcours typique du contenu d'un conteneur C
 - Définition de l'intervalle de parcours
 - Itérateur marquant le 1^{er} élément : `C.begin()`
 - « Itérateur » de dépassement : `C.end()`
 - pointe sur « après » le dernier élément du conteneur
 - Création de l'itérateur de parcours : `it`
 - `const_iterator` ou `iterator` selon l'accès aux `*it` dans le parcours
 - Déplacement de cet itérateur `it` sur l'intervalle

```
Container<Shape> C ;
```



```
for (Container<Shape>::iterator it = C.begin() ; it != C.end() ; ++it) {  
    std::cout << *it << std::endl ;  
}
```



`const_iterator` préférable, dans cet exemple : `C.cbegin()` , `C.cend()`



Les itérateurs (3/4)

• Remarques

- Fonctions de **recherche d'un objet** dans un conteneur `C`
 - Elles retournent un itérateur sur l'objet trouvé.
 - Si aucun objet n'est trouvé, elles retournent `C.end()`
- Un itérateur sert aussi à **indiquer une position** dans un conteneur : l'insertion d'un nouvel objet est alors possible
 - L'objet inséré l'est toujours **avant** l'itérateur de position
 - En tête de conteneur : avant `C.begin()`
 - En fin de conteneur : avant `C.end()`
- Comportements **indéterminés** si
 - Incrémentation de `C.end()`
 - Décrémentement de `C.begin()`



Les itérateurs (4/4)

• Familles d'itérateurs `it`

- En fonction des déplacements permis : `++it`, `--it`
- En fonction des accès autorisés par déréférencement `*it`

Famille d'itérateurs		Usage
II	« input »	Accès à l'objet en lecture (une fois) : en r-value L'ordre de parcours par incrémentation n'est pas spécifié
OI	« output »	Accès à l'objet en écriture (une fois) : l-value autorisée L'ordre de parcours par incrémentation n'est pas spécifié
FI	« forward »	II + OO + Ordre de parcours par incrémentation toujours identique
BI	« bidirectional »	FI + Passage à l'objet précédent par décrémentation
RI	« random access »	BI + Accès direct (par index ou déplacement de longueur variable : arithmétique des pointeurs)



Déduction automatique de type



• Mot-clef `auto`

- Lors de la déclaration d'une variable
- Nécessite une valeur initiale (pour en déduire le type)

```
double fn () ;

auto i = 53 ;           // i est de type int
auto d = fn () ;        // d est de type double
auto k ;                // Erreur : pas de déduction possible
```

• Intérêt

- Utile pour des types longs à écrire
- Mais de quel type est l'objet manipulé ?

```
Container<Shape> C ;

// Dans la boucle : it a pour type Container<Shape>::iterator
for (auto it = C.begin() ; it != C.end() ; ++it) {
    std::cout << *it << std::endl ;
}
```

`C.begin()`
`C.cend()`



Boucle « foreach »

C++11

- Syntaxe simplifiée pour parcourir des intervalles, des tableaux ou des « collections »



- Attention à la déclaration de l'élément considéré au sein de la collection

```
Container<Shape> C ;
```

```
// sh : référence à un Shape (qui est donc modifiable)
for (Shape& sh : C) {
    std::cout << sh << std::endl ;
}
```

```
// sh : référence à un Shape
//      non modifiable au travers de sh
for (const Shape & sh : C) {
    std::cout << sh << std::endl ;
}
```

```
// sh est une copie locale de l'objet Shape considéré
for (Shape sh : C) {
    std::cout << sh << std::endl ;
}
```

```
≡ for (auto& sh : C) {
    std::cout << sh << std::endl ;
}
```

```
≡ for (const auto& sh : C) {
    std::cout << sh << std::endl ;
}
```

```
≡ for (auto sh : C) {
    std::cout << sh << std::endl ;
}
```



Les algorithmes (1/4)

- Définition

- Algorithmes de manipulation des données stockées dans les conteneurs (de nombreux sont fournis par STL).

- Principe général

- **Découplage** fort entre algorithmes et conteneurs
- Exemple de l'algorithme **reverse**
 - **reverse** est une fonction globale \neq méthode d'un conteneur
 - **reverse** opère sur un intervalle d'éléments (défini par 2 itérateurs)

```
// v = [2 , 4 , 6]
std::vector<int> v ; ...

// Appel à l'algorithme 'reverse'
reverse (v.begin() , v.end()) ;

// v = [6 , 4 , 2]
```

```
std::list<int> lst ;

// Appel à l'algorithme 'reverse'
reverse (lst.begin() , lst.end()) ;
```



Les algorithmes (2/4)

• Exemples d'algorithmes prédéfinis

- `void swap (T& a, T&b)` : échange les contenus de `a` et `b`
- Les algorithmes suivants travaillent sur les éléments dont la position dans le conteneur est dans `[debut, fin[`
 - `FI remove (FI debut, FI fin, val)` : enlève les objets ayant pour valeur `val` et retourne un itérateur sur la fin du nouvel intervalle ne contenant plus d'élément de valeur `val`.
 - `FI remove_if (FI debut, FI fin, Predicat p)` : \equiv `remove`. Seuls les éléments pour lesquels le foncteur `p` retourne vrai, sont concernés.
 - `void replace (FI debut, FI fin, old, new)` : les éléments ayant pour valeur `old` se voient affectés la valeur `new`.
 - `void sort (RI debut, RI fin)` : trie par ordre croissant (selon `<`)
 - `void sort (RI debut, RI fin, Compare comp)` : trie les éléments par l'ordre croissant défini via la fonction / foncteur `comp`.



Les algorithmes (3/4)

• Un exemple complet : l'algorithme « `for_each` »

- Prototype : `for_each (II debut, II fin, Function f)`
- Fonctionnalité : Applique la fonction f à tous les éléments situés dans l'intervalle $[debut, fin[$

```
#include <iostream>
#include <algorithm>
#include <vector>
using namespace std;

void myfunction (int i) {
    cout << " " << i;
}

struct myclass {
    void operator() (int i) {
        cout << " " << i;
    }
} myobject ;

int main () {
    vector<int> myvector ;
```

```
    myvector.push_back(10) ;
    myvector.push_back(20) ;
    myvector.push_back(30) ;

    // pointer to a function
    cout << "myvector contains:" ;
    for_each (myvector.begin(), myvector.end(), myfunction) ;
    cout << endl ;

    // OR functor: class with overloaded ()
    cout << "\nmyvector contains:" ;
    for_each (myvector.begin(), myvector.end(), myobject) ;
    cout << endl ;

    return 0 ;
}
```



Les algorithmes (4/4)

• Utilisation possible de « lambda functions »

- Exemple : fonction « `count_if` »

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

```
struct functor {
    int a ;
    functor(int _a) : a(_a) { }
    bool operator()(int x) const {
        return a == x ;
    }
} ;
```

```
int main (int argc, char* argv[]) {
    vector<int> v = {100 , 200 , 42 , 400 } ;
    int a = 42 ;
    // Combien de valeur de 'a' dans le vecteur 'v' ?
```

```
count_if (v.begin(), v.end(), functor(a)) ;
```

```
    return 0 ;
}
```

function template

std::count_if

```
template <class InputIterator, class UnaryPredicate>
typename iterator_traits<InputIterator>::difference_type
count_if (InputIterator first, InputIterator last, UnaryPredicate pred);
```

Return number of elements in range satisfying condition

Returns the number of elements in the range `[first,last)` for which `pred` is true.

**Définition du prédicat
localement à l'utilisation**

C++11

```
count_if (v.begin(), v.end(),
    [a](int x){ return x == a; }
) ;
```



Practice



• Using a STL container: “vector”



- Create a forest of trees...
- Walk in the forest... looking for a tree,...
- Select some trees based on an algorithm

• Experimenting slicing...



- What about a container with heterogeneous contents?

