

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

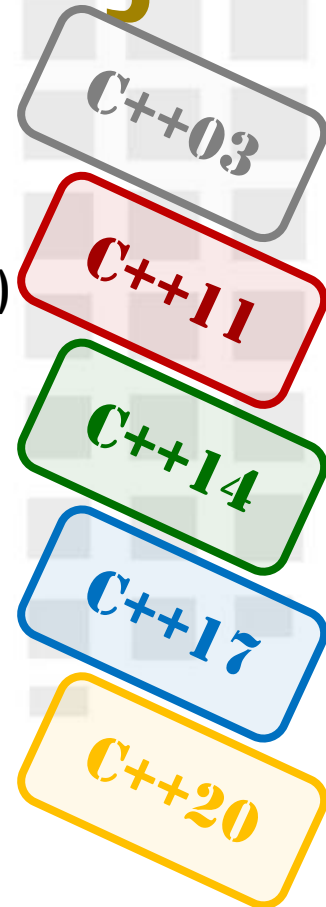
- Template functions and classes

♦ The Standard Template Library

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



♦ Smart pointers (STL & Boost)



Les conteneurs séquentiels

(1/12)

• Types de base

- **array**<T> : tableaux de taille fixe
- **forward_list**<T> : listes à chainage avant
- **list**<T> : listes à double chainage
- **vector**<T> : tableaux dynamiques
- **deque**<T> : tableaux circulaires

C++11**C++11**

• Adaptateurs implémentables grâce à ces types

- **stack**<> : piles
- **queue**<> : files
- **priority_queue**<> : files avec priorités

Exemple de piles implémentées avec des deque :

```
stack<T, Container=deque<T>>
```



Les conteneurs séquentiels

(2/12)



• `array<T, n>` : tableau de taille fixe

• Propriétés

- Accès direct par l'opérateur `[]` (allocation contigüe) : $\mathcal{O}(1)$
- Pas d'allocation / insertion / suppression dynamique d'élément
- Pas de surcoût par rapport aux tableaux classiques du C
- Utilisation possible des itérateurs / algorithmes

10	15	0	2	...	4	19
----	----	---	---	-----	---	----

• Quelques méthodes

- Retourne la taille **n** du tableau (= nombre d'éléments) \neq `sizeof`
 - `constexpr size_type size()`
- Remplissage avec un élément : $\sim \mathcal{O}(\text{size}())$
 - `void fill(const T& val)`
- Retourne une référence sur l'élément en position k : $\mathcal{O}(1)$

l'opérateur `[]`
ne le fait pas

- `reference at (size_type k)`
- Remarque : **at** signale toute tentative d'accéder à une position hors du tableau.



Les conteneurs séquentiels

(3/12)

C++11

• `array<T,n>`

```
// Inserting into a vector
#include <iostream>
#include <array>
using namespace std;

// default initialization:
// non-local = static storage:
// zero-initialized: {0,0,0}
array<int,3> v ;
```

```
int main () {
    for (it = myarray.begin() ;
         it < myarray.end() ;
         ++it)
        cout << " " << *it ;
    cout << endl ;

    for (auto x:myarray)
        cout << " " << x ;
    cout << endl ;

    return 0;
}
```

```
// default initialization (local = automatic storage):
// uninitialized: {?,?,?}
array<int,3> first ;
```

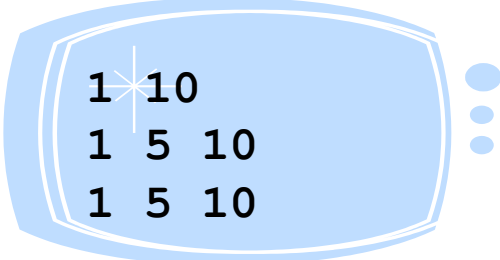


```
// initializer-list initializations:
array<int,3> second = {10,20} ;    // {10,20,0}
array<int,3> third = {1,2,3} ;     // {1,2,3}
```

```
// copy initialization:
array<int,3> myarray = third;      // copy: {1,2,3}
```

```
// assign some values
myarray[2] = 10 ;
myarray.at(1) = 5 ;
```

```
// read some values
cout << myarray[0] << " " << myarray.at(2) << endl ;
```



1	10
1	5 10
1	5 10



Les conteneurs séquentiels

(4/12)

- Plusieurs types de constructeurs

- Sans argument, avec une taille initiale, avec taille et valeur initiales, avec un intervalle d'itérateurs.

- Quelques méthodes communes

- `void insert(iterator it, ...)` : insère juste avant `it`, soit une valeur, soit `n` fois la valeur, soit un ensemble de données spécifiées par un intervalle d'itérateurs.
- `iterator erase(iterator a [, iterator b])` : efface le ou les éléments indiqués par un itérateur `a` ou un intervalle `[a,b[` et retourne un itérateur sur la position suivante.
- `reference front()`, `reference back()` : retourne une référence sur l'élément en tête ou en fin de conteneur (\neq `begin` ou `end`).
- `void clear ()` : vide la séquence



Les conteneurs séquentiels

(5/12)

- **vector<T> : tableau dynamique**

- Propriétés

- Accès direct par l'opérateur [] : $\mathcal{O}(1)$
- Comportement en insertion / suppression
 - insertion : réallocation possible \Rightarrow tous les itérateurs & références sont invalidés
 - suppression : les itérateurs & références suivant la position effacée sont invalidés



- Quelques méthodes

- Insertion, suppression en fin de tableau : $\mathcal{O}(1)$
 - void **push_back** (const T& obj), void **pop_back** (const T& obj)
- Insertion, suppression au cœur du tableau : $\sim \mathcal{O}(\text{size}())$
 - void **insert**(iterator it, ...), iterator **erase**(iterator a)
- Retourne une référence sur l'élément en position n : $\mathcal{O}(1)$
 - reference **at** (size_type n)
 - Remarque : **at** signale toute tentative d'accéder à une position hors du vecteur, l'opérateur [] ne le fait pas.



Les conteneurs séquentiels

(6/12)



• vector<T>

```
// Inserting into a vector
#include <iostream>
#include <vector>
using namespace std;

int main () {

    // Setup : 100 100 100
    vector<int> myvector (3,100) ;
    vector<int>::iterator it ;

    cout << "myvector contains:";
    for (it = myvector.begin() ;
         it < myvector.end() ;
         ++it)
        cout << " " << *it ;
    cout << endl ;

    return 0;
}
```

```
it = myvector.begin() ;           // 100 100 100
it = myvector.insert (it,200) ;   // ^
// "it" points to the new 200     // 200 100 100 100
                                   // ^
```

```
myvector.insert (it,2,300) ;
// 300 300 200 100 100 100
// "it" no longer valid
```



```
it = myvector.begin() ;           // 300 300 200 100 ...
                                   // ^
```

```
vector<int> anothervector (2,400) ;
myvector.insert (it+2 ,
                 anothervector.begin() ,
                 anothervector.end()) ;
// 300 300 400 400 200 100 100 100
// "it" no longer valid

int myarray [] = { 501,502,503 } ;
myvector.insert (myvector.begin() , myarray, myarray+3) ;
// 501 502 503 300 300 400 400 200 100 100 100
```

501 502 503 300 300 400
400 200 100 100 100



Inside « std::vector » (1/7)

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

using namespace std ;

struct A {

    int k = 0 ;

    A(int i)      { k = i ;   cout << "(" << k << ")" ctor : " << this << endl ; }
    A(const A& a) { k = a.k ; cout << "Copy ctor [" << k << "]" " << &a << " -> " << this << endl ; }
    A(A&& a)      { k = a.k ; cout << "Move ctor [" << k << "]" " << &a << " -> " << this << endl ; }
    A& operator= (const A& a) { k = a.k ; cout << "Copy = [" << k << "]" " << &a << " -> " << this << endl ;
                               return *this ; }
    A& operator= (A&& a)      { k = a.k ; cout << "Move = [" << k << "]" " << &a << " -> " << this << endl ;
                               return *this ; }
    ~A() { cout << "~ dtor [" << k << "]" " << this << endl ; }
} ;
```



```
int main() {
    std::vector<A> v ;
    std::cout << "return : " << std::endl ;
    return 0 ;
}
```



return :



Inside « std::vector » (2/7)

- Add a first object to an empty vector

```
int main() {
    std::vector<A> v ;
    std::cout << "a1 : " << std::endl ;
    A a1(1) ;
    v.push_back(a1) ;

    std::cout << "return : " << std::endl ;
    return 0 ;
}
```




```
a1 :
(1) ctor : 0x16b8e3220
Copy ctor [1] 0x16b8e3220 -> 0x155e067d0

return :
~ dtor [1] 0x16b8e3220
~ dtor [1] 0x155e067d0
```



Inside « std::vector » (3/7)

- Add a second object to the vector



```


int main() {
    std::vector<A> v ;

    std::cout << "a1 : " << std::endl ;
    A a1(1) ;
    v.push_back(a1) ;

    std::cout << "a2 : " << std::endl ;
    A a2(2) ;
    v.push_back(a2) ;

    std::cout << "return : " << std::endl ;
    return 0 ;
}

```



```

a1 :
(1) ctor : 0x16b0df220
Copy ctor [1] 0x16b0df220 -> 0x15be067d0

a2 :
(2) ctor : 0x16b0df21c
Copy ctor [2] 0x16b0df21c -> 0x15be067e4
Copy ctor [1] 0x15be067d0 -> 0x15be067e0
~ dtor [1] 0x15be067d0

return :
~ dtor [2] 0x16b0df21c
~ dtor [1] 0x16b0df220
~ dtor [2] 0x15be067e4
~ dtor [1] 0x15be067e0

```



Inside « std::vector » (4/7)

• “Giving” a third element to the vector

```
int main() {

    std::vector<A> v ;

    std::cout << "a1 : " << std::endl ;
    A a1(1) ;
    v.push_back(a1) ;

    std::cout << "a2 : " << std::endl ;
    A a2(2) ;
    v.push_back(a2) ;

    std::cout << "a3 : (move) " << std::endl ;
    A a3(3) ;
    v.push_back(std::move(a3)) ;

    std::cout << "return : " << std::endl ;
    return 0 ;
}
```



```
a1 :
(1) ctor : 0x16b437220
Copy ctor [1] 0x16b437220 -> 0x127e067d0
a2 :
(2) ctor : 0x16b43721c
Copy ctor [2] 0x16b43721c -> 0x127e067e4
Copy ctor [1] 0x127e067d0 -> 0x127e067e0
~ dtor [1] 0x127e067d0
```

```
a3 : (move)
(3) ctor : 0x16b437218
Move ctor [3] 0x16b437218 -> 0x127e067d8
Copy ctor [2] 0x127e067e4 -> 0x127e067d4
Copy ctor [1] 0x127e067e0 -> 0x127e067d0
~ dtor [2] 0x127e067e4
~ dtor [1] 0x127e067e0
```

```
return :
~ dtor [3] 0x16b437218
~ dtor [2] 0x16b43721c
~ dtor [1] 0x16b437220
~ dtor [3] 0x127e067d8
~ dtor [2] 0x127e067d4
~ dtor [1] 0x127e067d0
```



Inside « std::vector » (5/7)

- Tell the vector that moving elements is allowed

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

```
using namespace std ;
```

```
struct A {
```

```
    int k = 0 ;
```

```
    A(int i) { k = i ; cout << "(" << k << ") ctor : " << k << endl ; }
```

```
    A(const A& a) { k = a.k ; cout << "Copy ctor [" << k << "]" << endl ; }
```

```
    A(A&& a) noexcept { k = a.k ; cout << "Move ctor [" << k << "]" << endl ; }
```

```
    A& operator= (const A& a) { k = a.k ; cout << "Copy = [" << k << "]" << endl ;  
                                return *this ; }
```

```
    A& operator= (A&& a) { k = a.k ; cout << "Move = [" << k << "]" << endl ;  
                           return *this ; }
```

```
    ~A() { cout << "~ dtor [" << k << "]" << this << endl ; }
```

```
};
```



```
a1 :
(1) ctor : 0x16b437220
Copy ctor [1] 0x16b437220 -> 0x127e067d0
a2 :
(2) ctor : 0x16b43721c
Copy ctor [2] 0x16b43721c -> 0x127e067e4
Move ctor [1] 0x127e067d0 -> 0x127e067e0
~ dtor [1] 0x127e067d0
```

a3 : (move)

```
(3) ctor : 0x16b437218
Move ctor [3] 0x16b437218 -> 0x127e067d8
Move ctor [2] 0x127e067e4 -> 0x127e067d4
Move ctor [1] 0x127e067e0 -> 0x127e067d0
~ dtor [2] 0x127e067e4
~ dtor [1] 0x127e067e0
```

return :

```
~ dtor [3] 0x16b437218
~ dtor [2] 0x16b43721c
~ dtor [1] 0x16b437220
~ dtor [3] 0x127e067d8
~ dtor [2] 0x127e067d4
~ dtor [1] 0x127e067d0
```



Inside « std::vector » (6/7)

• Erase the first object

```
int main() {

    std::vector<A> v ;

    std::cout << "a1 : " << std::endl ;
    A a1(1) ;
    v.push_back(a1) ;

    std::cout << "a2 : " << std::endl ;
    A a2(2) ;
    v.push_back(a2) ;

    std::cout << "a3 : (move) " << std::endl ;
    A a3(3) ;
    v.push_back(std::move(a3)) ;

    std::cout << "erase of 1st" << std::endl ;
    v.erase(v.begin()) ;

    std::cout << "return : " << std::endl ;

    return 0 ;
}
```



```
a1 :
(1) ctor : 0x16f9f3220
Copy ctor [1] 0x16f9f3220 -> 0x156e067d0
a2 :
(2) ctor : 0x16f9f321c
Copy ctor [2] 0x16f9f321c -> 0x156e067e4
Move ctor [1] 0x156e067d0 -> 0x156e067e0
~ dtor [1] 0x156e067d0
a3 : (move)
(3) ctor : 0x16f9f3218
Move ctor [3] 0x16f9f3218 -> 0x156e067d8
Move ctor [2] 0x156e067e4 -> 0x156e067d4
Move ctor [1] 0x156e067e0 -> 0x156e067d0
~ dtor [2] 0x156e067e4
~ dtor [1] 0x156e067e0
```

erase of 1st

```
Move = [2] 0x156e067d4 -> 0x156e067d0
Move = [3] 0x156e067d8 -> 0x156e067d4
~ dtor [3] 0x156e067d8
```

return :

```
~ dtor [3] 0x16f9f3218
~ dtor [2] 0x16f9f321c
~ dtor [1] 0x16f9f3220
~ dtor [3] 0x156e067d4
~ dtor [2] 0x156e067d0
```



Inside « std::vector » (7/7)

• Add another object

```
int main() {

    std::vector<A> v ;

    std::cout << "a1 : " << std::endl ;
    A a1(1) ;
    v.push_back(a1) ;

    std::cout << "a2 : " << std::endl ;
    A a2(2) ;
    v.push_back(a2) ;

    std::cout << "a3 : (move) " << std::endl ;
    A a3(3) ;
    v.push_back(std::move(a3)) ;

    std::cout << "erase of 1st" << std::endl ;
    v.erase(v.begin()) ;

    std::cout << "a4 : (move) " << std::endl ;
    A a4(4) ;
    v.push_back(std::move(a4)) ;

    std::cout << "return : " << std::endl ;

    return 0 ;
}
```



```
a1 :
(1) ctor : 0x16f9f3220
Copy ctor [1] 0x16f9f3220 -> 0x156e067d0
a2 :
(2) ctor : 0x16f9f321c
Copy ctor [2] 0x16f9f321c -> 0x156e067e4
Move ctor [1] 0x156e067d0 -> 0x156e067e0
~ dtor [1] 0x156e067d0
a3 : (move)
(3) ctor : 0x16f9f3218
Move ctor [3] 0x16f9f3218 -> 0x156e067d8
Move ctor [2] 0x156e067e4 -> 0x156e067d4
Move ctor [1] 0x156e067e0 -> 0x156e067d0
~ dtor [2] 0x156e067e4
~ dtor [1] 0x156e067e0
erase of 1st
Move = [2] 0x156e067d4 -> 0x156e067d0
Move = [3] 0x156e067d8 -> 0x156e067d4
~ dtor [3] 0x156e067d8

a4 : (move)
(4) ctor : 0x16f9f31fc
Move ctor [4] 0x16f9f31fc -> 0x156e067d8

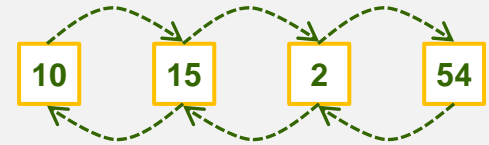
return :
~ dtor [4] 0x16f9f31fc
~ dtor [3] 0x16f9f3218
~ dtor [2] 0x16f9f321c
~ dtor [1] 0x16f9f3220
~ dtor [4] 0x156e067d8
~ dtor [3] 0x156e067d4
~ dtor [2] 0x156e067d0
```



Les conteneurs séquentiels

(7/12)

- **`list<T>`** : liste à chainage double



- Propriétés

- Itérateur bi-directionnel $\mathcal{O}(\text{size}())$, pas d'accès aléatoire.
- Comportement en insertion / suppression
 - temps constant
 - préservation de la validité des itérateurs et références

- Quelques méthodes

- Insertion, suppression, lecture en début et fin de liste : $\mathcal{O}(1)$
 - void **`push_front`** (const T& obj), void **`push_back`** (const T& obj)
 - void **`pop_front`** (const T& obj), void **`pop_back`** (const T& obj)
 - reference **`front`**(), reference **`back`**()
- Supprime toutes les occurrences de obj dans la liste : $\mathcal{O}(\text{size}())$
 - void **`remove`** (const T& obj)
 - Remarque : **`erase`** supprime des éléments selon leur positions, **`remove`** supprime des éléments selon leurs valeurs.



Les conteneurs séquentiels

(8/12)

• `list<T>`

```
// Erasing from list
#include <iostream>
#include <list>

using namespace std ;

int main () {
    int i;
    list<int> mylist;
    list<int>::iterator it1,it2;

    // set some values:
    for (i=1; i<10; i++)
        mylist.push_back(i*10);

    cout << "mylist contains:";
    for (it1=mylist.begin() ;
        it1!=mylist.end();
        ++it1)
        cout << " " << *it1;
    cout << endl;

    return 0;
}
```

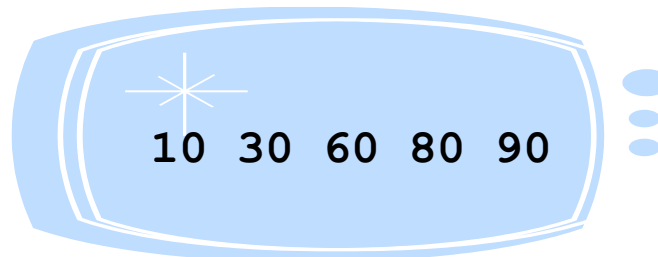
```
// 10 20 30 40 50 60 70 80 90
it1 = it2 = mylist.begin(); // ^^
advance (it2,6);           // ^
++it1;                     // ^

it1 = mylist.erase (it1);   // 10 30 40 50 60 70 80 90
                           // ^

it2 = mylist.erase (it2);   // 10 30 40 50 60 80 90
                           // ^

++it1;                     // ^
--it2;                     // ^

it1 = mylist.erase (it1,it2); // 10 30 60 80 90
                           // ^
```



Les conteneurs séquentiels

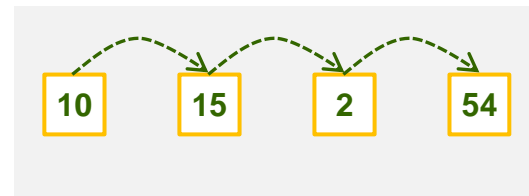
(9/12)



- **forward_list<T>** : liste à chainage avant

- Propriétés

- Itérateur mono-directionnel $\mathcal{O}(\text{size}())$
- Pas d'accès aléatoire.
- Comportement en insertion / suppression
 - temps constant, préservation de la validité des itérateurs et références



- Quelques méthodes

- Pas de fonction membre **size()** : pas de stockage de la taille de la liste (\neq **list**) dans un souci d'optimisation des performances
- Insertion, suppression, lecture en début de liste : $\mathcal{O}(1)$
 - void **push_front** (const T& obj), void **pop_front** (const T& obj)
 - reference **front**()
- Supprime toutes les occurrences de obj dans la liste : $\mathcal{O}(\text{size}())$
 - void **remove** (const T& obj)



Les conteneurs séquentiels


(10/12)

C++11

• forward_list<T>

```
// Erasing example
#include <iostream>
#include <forward_list>
#include <iterator>

using namespace std ;

int main () {
    
    return 0;
}
```

```
forward_list<int> lst = { 1, 2, 3, 4, 5, 6, 7, 8, 9 } ;
                        // 1 2 3 4 5 6 7 8 9

// ERROR: No function erase
// lst.erase (lst.begin()) ;

// Remove first element
lst.erase_after (lst.before_begin()) ; // 2 3 4 5 6 7 8 9

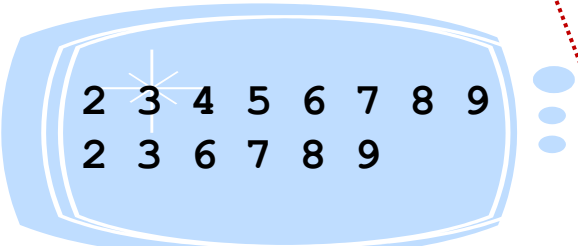
for (auto n : lst) cout << n << " " ;
cout << endl ;

                                // 2 3 4 5 6 7 8 9
                                //   ^
                                //           ^

auto fi= next (lst.begin()) ;
auto la= next (fi, 3) ;

// Remove a range of elements
// after 'fi' to just before 'la'
auto it = lst.erase_after (fi, la) ; // 2 3 6 7 8 9
                                   //           ^

for (auto n : lst) cout << n << " " ;
cout << endl ;
```



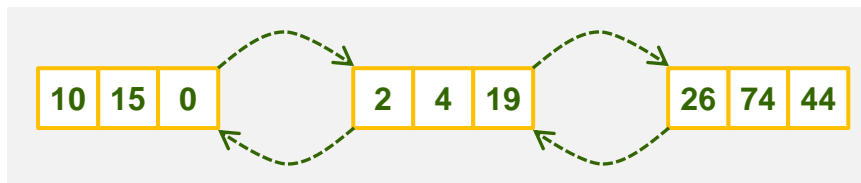
2 3 4 5 6 7 8 9
2 3 6 7 8 9



Les conteneurs séquentiels

(11/12)

- **deque<T>** : « *double-ended queue* »
 - Comportement similaire à un **vector**
 - Accès direct aux éléments (opérateur [])
 - Itération bi-directionnelle
 - Ajout / suppression en fin de conteneur : $\mathcal{O}(1)$
 - Avantages par rapport à un **vector**
 - Pas de réallocation de tout le vecteur lorsque le conteneur grossit
 - Ajout / suppression en début de conteneur : $\mathcal{O}(1)$
 - Différences par rapport à un **vector**
 - Contigüité du rangement des éléments en mémoire non garantie
 - chaînage par bloc



Les conteneurs séquentiels

(12/12)

• deque<T>

```
// Inserting into a deque
#include <iostream>
#include <deque>
#include <vector>

using namespace std ;

int main () {
    deque<int> mydeque;
    deque<int>::iterator it;

    // set some initial values
    for (int i=1; i<6; i++)
        mydeque.push_back(i) ;
    // : 1 2 3 4 5

    it = mydeque.begin() ;
    ++it ;

    it = mydeque.insert (it,10) ;
    // : 1 10 2 3 4 5
    // "it" now points to the
    // newly inserted 10
```

```
mydeque.insert (it,2,20) ;
// 1 20 20 10 2 3 4 5
// "it" no longer valid!
```

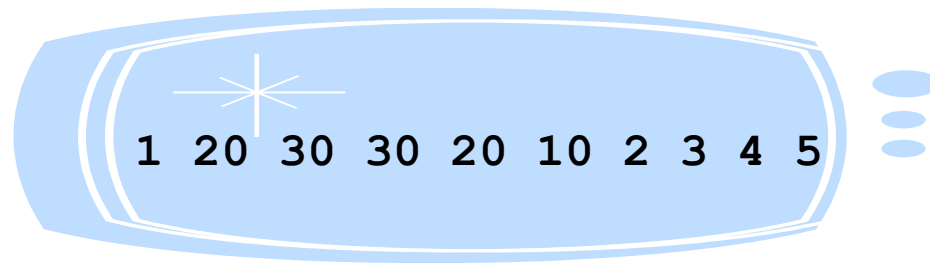


```
it = mydeque.begin()+2 ;
```

```
vector<int> myvector (2,30) ;
mydeque.insert (it,myvector.begin(),myvector.end());
// 1 20 30 30 20 10 2 3 4 5
```

```
cout << "mydeque contains:" ;
for (it = mydeque.begin(); it < mydeque.end(); ++it)
    cout << " " << *it;
cout << endl ;
```

```
return 0;
}
```



Les conteneurs associatifs (1/8)

• Principe général

- On stocke un élément de type **<clé, donnée>**
 - La donnée de type T contient **l'information effective**
 - La clé de type K sert à **organiser** le stockage (recherches optimisées)
- Familles des conteneurs associatifs **ordonnés**
 - fourniture d'un opérateur de **comparaison** `compare<K>` (par défaut `<`)
- Famille des conteneurs associatifs **non-ordonnés**
 - fourniture d'opérateurs d'**égalité** `equal_to<K>` et de **hashage** `hash<K>`

Les données sont les clés
des conteneurs « ensemble »

	Clé unique	Clés multiples
Ensembles	<code>set<T></code> <code>unordered_set<T></code>	<code>multiset<T></code> <code>unordered_multiset<T></code>
Associations	<code>map<K, T></code> <code>unordered_map<K, T></code>	<code>multimap<K, T></code> <code>unordered_multimap<K, T></code>



Les conteneurs associatifs (2/8)

- **Implémentation par la STL**

- **Conteneurs ordonnés**

- Structures en arbre « rouge et noir » (arbre binaire)
- Insertion, suppression, recherche : complexité en $\mathcal{O}(\log(n))$
- Constructeurs
 - 1^{er} cas : une relation d'ordre sur les clés (<) existe déjà
 - 2^{ème} cas : (re)définition d'une relation d'ordre (pointeur sur une fonction ou foncteur)

- **Conteneurs non-ordonnés**

- Organisation en « buckets » reposant sur une fonction de hachage
- Insertion, suppression, recherche : pire cas en $\mathcal{O}(n)$, amortie en $\mathcal{O}(1)$
- Accès individuel à un élément plus rapide mais moins efficace pour parcourir un sous-ensemble.



Les conteneurs associatifs (3/8)

- **set<T> : Ensembles**

- Propriétés

- Deux éléments d'un **set** sont forcément **différents** \Rightarrow l'insertion d'un nouvel élément ne se fera que si aucun élément ayant la même valeur n'est déjà présent dans l'ensemble
- L'ajout et la suppression d'un élément n'invalident pas les itérateurs référençant d'autres éléments de l'ensemble

- Implémentation

- A tout moment, le rangement est **ordonné** selon les valeurs des données qui sont considérées comme des clés (et la relation d'ordre sous-jacente) \Rightarrow optimisation des algorithmes ensemblistes (intersection, union, ...)



Les conteneurs associatifs (4a/8)

• set<T>

```
#include <iostream>
#include <set>
using namespace std;

int main () {
    set<int> myset;
    set<int>::iterator it;
    pair<set<int>::iterator,bool> ret;

    // set some initial values :
    // 10 20 30 40 50
    for (int i=1; i<=5; i++)
        myset.insert(i*10) ;

    cout << "myset contains:";
    for (it = myset.begin();
         it != myset.end();
         ++it)
        cout << " " << *it ;
    cout << endl;

    return 0;
}
```

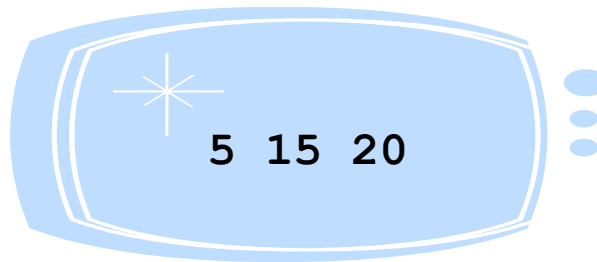
```
ret = myset.insert(20);    // no new element inserted
if (ret.second==false)
    it = ret.first;        // "it" now points to 20

myset.insert (it,25);      // max efficiency inserting
                          // 10 20 25 30 40 50

int myints[]={5,10,15};
myset.insert (myints,myints+3); // 10 already in set
                          // 5 10 15 20 25 30 40 50

it = myset.begin();
it++;                     // "it" points now to 10
myset.erase (it);         // 5 15 20 25 30 40 50
myset.erase (40);         // 5 15 20 25 30 50

it = myset.find (25);      // "it" points now to 25
myset.erase (it, myset.end());
                          // 5 15 20
```



Les conteneurs associatifs (4b/8)

• set<T>

```
#include <set>
#include <iostream>
#include <algorithm>
```

```
struct Foo {
    int x ;

    Foo (int _x) : x(_x) {}
    ~Foo() {}
} ;
```

```
struct FooOps {
    bool operator() (const Foo& a, const Foo& b ) {
        return a.x > b.x ;
    }
    void operator() (const Foo& a) {
        std::cout << a.x << " " ;
    }
} ;
```

```
int main() {
    // std::set<Foo>      foo_set2 ;
    std::set<Foo, FooOps> foo_set1 ;

    foo_set1.insert (Foo(2)) ; // foo_set2.insert (Foo(2)) ;
    foo_set1.insert (Foo(1)) ; // foo_set2.insert (Foo(1)) ;
    foo_set1.insert (Foo(3)) ; // foo_set2.insert (Foo(3)) ;
    foo_set1.insert (Foo(2)) ; // foo_set2.insert (Foo(2)) ;

    std::for_each (foo_set1.begin(), foo_set1.end(), FooOps()) ;
    std::for_each (foo_set1.begin(), foo_set1.end(),
        [] (const Foo& p) { std::cout << p.x << " " ; }
    ) ;

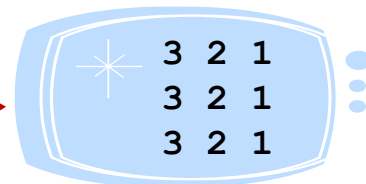
    for (const Foo& p : foo_set1)
        std::cout << p.x << " " ;

    return 0 ;
}
```

Erreur de compilation :
L'opérateur < (comparaison stricte)
n'est pas défini pour la classe Foo
et std::set en a besoin

Le foncteur FooOps fournit
l'opérateur() (const Foo&, const Foo&)
pour permettre une comparaison stricte
entre les éléments de l'ensemble foo_set1

Le foncteur FooOps fournit
l'opérateur() (const Foo&)
invocé par l'algorithme for_each



Les conteneurs associatifs (5/8)

• unordered_set<T>

local_it
bucket iterator

```
// unordered_set begin/end example
#include <iostream>
#include <string>
#include <unordered_set>
```

```
using namespace std;
```

```
int main () {
    unordered_set<string> myset = {
        "Mercury", "Venus", "Earth",
        "Mars", "Jupiter", "Saturn",
        "Uranus", "Neptune" };
```

```
// Full set
```

```
cout << "myset: " << endl ;
for (auto it = myset.begin() ;
     it != myset.end() ; ++it )
    cout << " " << *it ;
cout << endl ;
```

```
return 0;
```

```
}
```

it
container iterator

```
// By buckets
```

```
cout << "myset's buckets: " << endl ;
for (unsigned i = 0; i < myset.bucket_count() ; ++i) {
    cout << "bucket #" << i << " contains: " ;
    for (auto local_it = myset.begin(i) ;
         local_it != myset.end(i) ; ++local_it)
        cout << " " << *local_it ;
    cout << endl ;
```

```
myset: Venus Jupiter Neptune ...
myset's buckets contain:
bucket #0 contains:
bucket #1 contains: Venus
bucket #2 contains: Jupiter
bucket #3 contains:
bucket #4 contains: Neptune Mercury
bucket #5 contains:
bucket #6 contains: Earth
bucket #7 contains: Uranus Saturn
bucket #8 contains: Mars
bucket #9 contains:
bucket #10 contains:
```



Les conteneurs associatifs (6/8)

- **map<K, T> : Associations**

- Entités pointées par les itérateurs : `pair<const K, T>`

```
map<K,T>::iterator it;  
(*it).first;      // the key value (of type K)  
(*it).second;    // the mapped value (of type T)  
(*it);           // the "element value" (of type pair<const K,T>)
```

- A tout moment, le rangement est **ordonné** selon les valeurs des clés (et la relation d'ordre sous-jacente)

- Accès direct aux données par utilisation de leur clé



- Opérateur `[]` : recherche ou/et insertion !!!
- Méthode `find` : recherche (sans effet de bord)



Les conteneurs associatifs (7/8)

• map<K, T>

```
#include <iostream>
#include <map>
using namespace std;
```

```
int main () {
    map<char, string> mymap ;
    mymap['a'] = "A" ;    mymap['b'] = "B" ;
    mymap['c'] = "C" ;    mymap['d'] = "D" ;
    mymap['e'] = mymap['a'] ;

    map<char, string>::iterator it ;
    it = mymap.find('b') ;    // it points to objet with key 'b'
    mymap.erase (it) ;

    mymap.erase (mymap.find('d')) ;

    cout << (int) mymap.size() << " elements." << endl ;
    for (it = mymap.begin() ; it != mymap.end(); it++)
        cout << (*it).first << " => " << (*it).second << endl ;

    cout << "'a' => " << mymap.find('a')->second << endl ;
    cout << "'b' => " << mymap['b'] << endl ;
    cout << "'c' => " << mymap['c'] << endl ;
    cout << "'d' => " << mymap['d'] << endl ;
    cout << (int) mymap.size() << " elements." << endl ;

    return 0 ;
}
```

3 elements

a => A

c => C

e => A

'a' => A

'b' =>

'c' => C

'd' =>

5 elements



[] ≠ find



Les conteneurs associatifs (8/8)

• map<K, T>

```
// Constructing maps
#include <map>
using namespace std;

bool fncomp (char lhs, char rhs) { return lhs < rhs ; }

struct classcomp {
    bool operator() (const char& lhs, const char& rhs) const {
        return lhs < rhs ;
    }
};

int main () {
    map<char,int> first ;
    first['a']=10; first['b']=30; first['c']=50; first['d']=70;

    map<char,int> second (first.begin(),first.end()) ;

    map<char,int> third (second) ;

    map<char,int,classcomp> fourth ;           // classcomp as Compare

    bool(*fn_pt)(char,char) = fncomp ;        // fncomp as Compare
    map<char,int,bool(*) (char,char)> fifth (fn_pt) ;

    return 0;
}
```



Retour sur les conteneurs (1/2)

- **Choix d'un conteneur particulier**
 - Quelles sont les données à représenter ?
 - Lien entre ces données
 - relation d'ordre ou non
 - Mode de représentation privilégié
 - orienté « nœud » ou allocation mémoire contigüe
 - Quels sont les types d'accès prédominants ?
 - Mode de navigation
 - accès direct, accès séquentiel (en avant, en arrière, ...)
 - Compromis entre les performances
 - d'accès en lecture, d'insertion de nouvelles données, de suppression de données



Retour sur les conteneurs (2/2)

• Phénomène de « slicing » / polymorphisme

• Rappel du problème



- STL : insertion des éléments uniquement par **copie** ou **appropriation**
- On insère un objet de type **Circle** dans une **list<Shape>**
- Que stocke-t-on vraiment ?

• Idée :



- Ne pas stocker des copies d'éléments mais des **pointeurs** sur ces éléments \Rightarrow on retrouve alors les intérêts du polymorphisme

• Inconvénients

- La gestion de la mémoire reste alors à la charge du programmeur !
 - Allocation par **new** / libération par **delete**
 - Source de nombreux « bugs »



Faire appel à des « **smart pointers** »

