Points clés sur les conteneurs en C++

Michel Simatic, Amina Guermouche

Télécom SudParis

List

1. Définition :

```
std::list<int> l; /* definition d'une liste
    vide*/
std::list<int> l = { 7, 5, 16, 8 }; /*
    definition d'une liste de 4 elements*/
```

2. Quelques fonctions utiles de la classe list :
   - `push_front` : ajout en tête
   - `push_back` : ajout en queue
   - `pop_front` : supprime le premier élément
   - `pop_back` : supprime le dernier élément
   - `insert(iterator position, valeur)`
3. Parcours : iterator ou range based loop comme les vecteur

Différence entre vecteurs et listes

Opération	vecteur	list	remarques
parcours	O(n)	O(n)	
ajout fin	O(n)	O(1)	tableau O(1) s'il y a de la place
ajout	O(n)	O(1)	pour le tableau, il faut copier les données
suppression à la fin	O(1)	O(1)	
suppression ailleurs	O(n)	O(1)	il faut copier les données du tableau
recherche pas trié	O(n)	O(n)	
recherche trié	O(log n)	O(n)	On n'a pas moyen d'accéder à une "case" particulière avec la liste

map

1. Définition : Une map est une séquence ordonnée de couples (clé, valeur) (`(key, value)`)

```
std::map<char,int> ma_map; /* definition d'
    une map vide*/
```

2. Parcours range based loop ou iterator
3. Accès aux éléments :

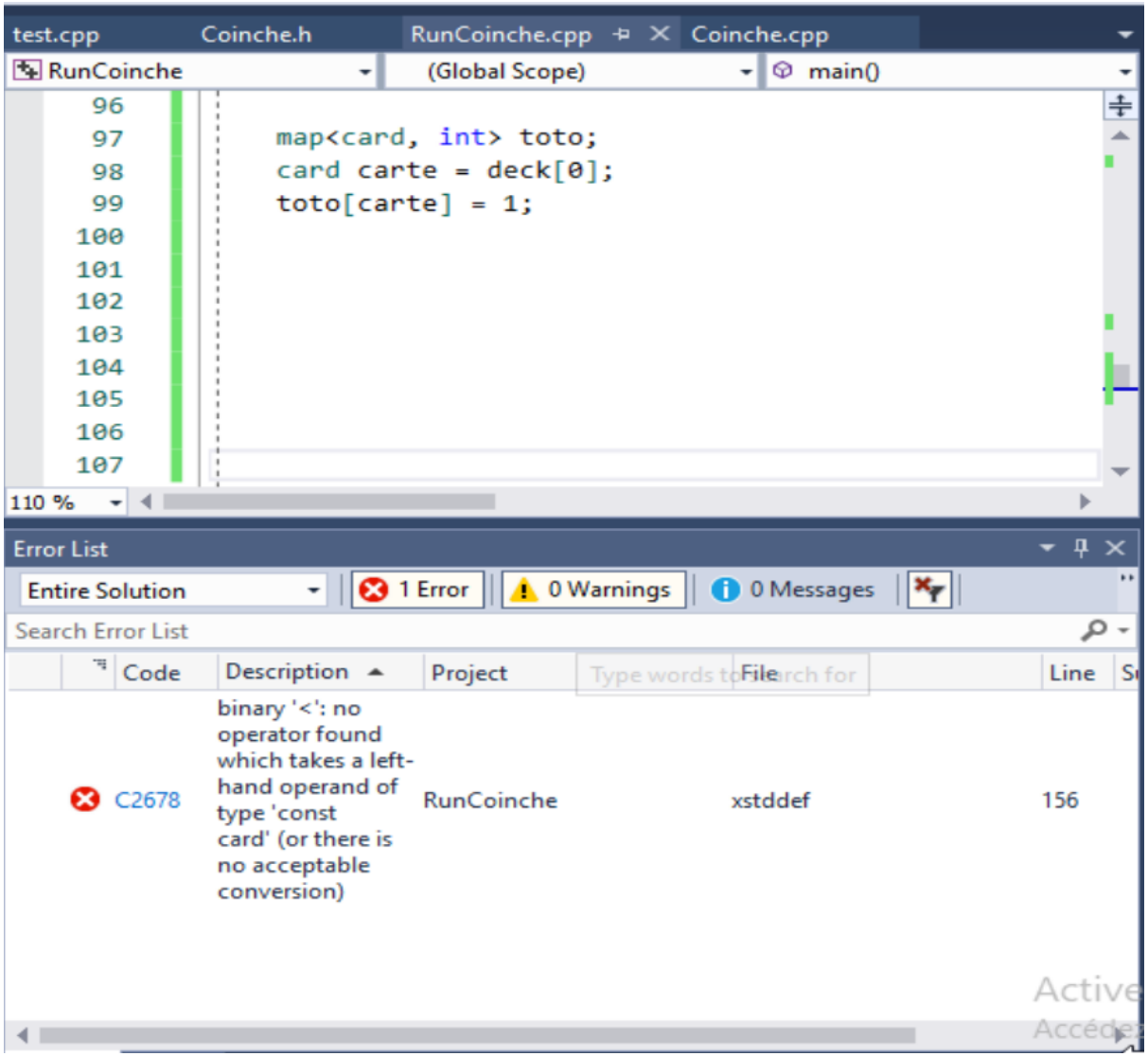
```
for(const auto & data : ma_map)
    cout << i.first << '' '' << i.second <<
        endl; /* i.first represente la cle, i.
        second la valeur */
```

4. Remarque : Il est possible d'accéder aux éléments en faisant `ma_map[cle]`

Quelques remarques sur les map

- Une map est triée selon l'ordre de clé (ordre croissant par défaut ou ordre défini par l'utilsateur)
- Dans la STL, une map est implémentée en arbre binaire de recherche
- Lors de l'insertion d'un élément, la map vérifie si un élément avec la même clé existe déjà. Si c'est le cas, le nouvel élément n'est pas inséré.
- Étant donné que le tri par défaut dans une map utilise l'opérateur $<$ (strict weak order), il est nécessaire de le surcharger pour des types définis par l'utilisateur. Sinon, il y aura des erreurs de compilation lors de l'insertion d'éléments.

Surcharge de l'opérateur <



Surcharge de l'opérateur <

- Condition du strict weak order :
  - $a > a$: false
  - $(a < b)$ et $b < c \rightarrow a < c$
  - $!(a < b)$ et $!(b < a) \rightarrow a \equiv b$

```
bool operator<(const card &a, const card &b)
{
  return a.couleur < b.couleur ||
         (a.couleur == b.couleur && a.valeur < b.
             valeur) ||
         (a.couleur == b.couleur && a.valeur == b.
             valeur && a.force < b.force); /*
             conditions necessaires pour que a < b
             */
}
```

multimap

1. Définitnion : Une multimap est une map dans laquelle plusieurs valeurs peuvent avoir la même clé
2. Une multimap est triée selon la fonction de comparaison des clés
3. Deux éléments avec une même clé seront trié selon l'ordre d'insertion
4. Une multimap est également représentée sous forme d'arbre binaire de recherche

```
multimap <int, int> ma_map;          /* empty
    multimap container*/
```

Unordered map

1. Une unoredred map est également une séquence de couple <clé, valeur>
2. Une unordered map est représentée par une table de hashage dont les clés sont transformées en indices de la table grâce à la fonction de hashage.
3. Les clés ne sont pas triées

```
unordered_map<string, double> umap;
```

Différence entre une map et une unordered map

Opération	map	unordered_map	Remarques
recherche	O(log n)	O(1)	si toutes les valeurs sont sur le même hash
ajout	O(log n)	O(1)	O(n) pour la unordered map
parcours	trié	non trié	

Quel conteneur choisir ?

- Utiliser un vecteur sauf si vous avez une bonne raison de ne pas le faire
- Utiliser une map quand vous voulez faire une recherche basée sur une valeur
- Utiliser une unordered map si vous voulez faire plusieurs recherches dans un ensemble assez large et que vous n'avez pas besoin d'un parcours trié

Transformation d'un conteneur vers un autre 1/2

1. Boucle classique avec `push_back`
2. 2 iterators

```
map<int,int> m;
/* remplir la map*/
vector<pair<int,int>> v (m.begin(), m.end());
```

3. transform : `transform(iterator position_initiale_source, iterator position_finale_source, inserter, operation)`
   - inserter : Un inserter est un iterator indiquant comment vont être insérés les éléments dans le container

Transformation d'un container vers un autre container 2/2

```
pair<int , int> return_pair(const pair<int ,int> &p){
  return p;
}

int main(){
map<int ,int> m;
/* remplir la map*/
vector<pair<int ,int>> v;

transform(m.begin(), m.end(), back_inserter(v), /*
    back_inserter fait appel a push_back*/
return_pair);
```

Tri d'un tableau 1/2

- `sort(iterator debut, iterator fin, operation de comparaison)`

1. Définition d'une fonction de comparaison

```
bool my_lt(const pair<int,int>&a, const pair<
    int,int> &b)
{
  return a < b /* supposons qu'on a surcharge
      l'operateur < pour une pair*/
}
int main()
{
  vector<pair<int,int>> v;
  /* remplir v*/
  sort(v.begin(), v.end(), my_lt);
  ...
}
```

Tri d'un tableau 2/2

2. Lambda expression (depuis C++11)

```
vector<pair<int,int>> v;
/* remplir v*/
sort(v.begin(), v.end(),
[] (const pair<int, int>& a, const pair<int,
    int> &b) -> bool
{
  return a < b
}
);
```

3. Dissection : [capture] (paramètre) -> type de retour corps
4. Remarque : Le type de retour n'est pas nécessaire et le compilateur le détecte de la même manière que `auto`. Ainsi -> bool n'est pas nécessaire dans l'exemple. Il faut cependant le spécifier dans des cas complexes, par exemple lorsqu'il y a plus de 1 return (dans le cas de if par exemple).

Intérêt des lambda expression

- Visibilité des variables de la fonction contenant la lambda expression
  1. [] : pas d'accès aux variables externes à la lambda expression
  2. [&] : accès à toutes les variables de la fonction par référence
  3. [=] : accès à toutes les variables de la fonction par copie
  4. [a,&b] : accès à a par copie et à b par référence

```
vector <int> v1;
vector <int> v2;
sort(v1.begin(),v2.begin(),
      [&] (const int &a, const int &b) /* acces a v1
            et v2 par reference*/
      {
        return ...
      });
```

Suppression d'un élément d'une liste 1/2

- `remove` : Supprime les éléments égaux à une valeur

```
std::remove(v.begin(), v.end(), 1);
```

- `remove_if` : Supprime les éléments vérifiant une condition

```
std::remove_if(v.begin(), v.end(), [](const
    int &a){return a < 0;}); // supprime tous
     les elements negatifs
```

Remarque : Ces fonctions ne suppriment pas l'espace alloués aux éléments supprimés. Les éléments sont cependant inacessibles. Ce comportements est dû au fait que ces fonctions ne sont pas uniquement appliquées aux conteneurs.

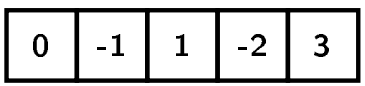
Suppression d'un élément d'une liste 2/2

- Combiner `erase` et `remove` (ou `remove_if`) pour supprimer les éléments ⇒ Idiome *erase remove*
- `remove` : Supprime les éléments égaux à une valeur

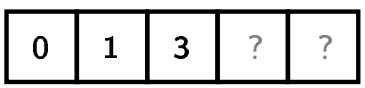
```
v.erase(std::remove(v.begin(), v.end(), 1));
```

- `remove_if` : Supprime les éléments vérifiant une condition

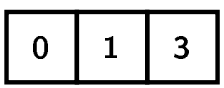
```
v.erase(std::remove_if(v.begin(), v.end(),
    [](const int &a){return a < 0;})); //
    supprime tous les elements negatifs
```



Vecteur initial



`remove_if`



`erase remove_if`

Lambda expressions dans le transform

```
map<int,int> m;
/* remplir la map*/
vector<pair<int,int>> v;

transform(m.begin(), m.end(), back_inserter(v), /*
    back_inserter fait appel a push_back*/
[](const pair<int, int> &p){
  return p;
}

/* si on veut que v ne contiennent que les valeurs
    */
vector<int> v1;
transform(m.begin(), m.end(), back_inserter(v1),
[](const pair<int, int> &p){
  return p.second;
}
```