Points clés sur les vecteurs en C++

Michel Simatic, Amina Guermouche

Télécom SudParis

Quelques fonctions utiles

1. Déclaration et initialisation :

```
std::vector<int> v(5); /*definit un vecteur
                         de 5 entiers*/
```

2. Quelques fonctions utiles de la classe vector (Chapitre 19 pages 667-677) :
   - `push_back` : Ajout à la fin du vecteur
   - `pop_back` : retire de la fin du vecteur
   - `size` : retourne le nombre d'éléments du vecteur
   - `erase` : supprime un éléments ou un intervalle d'un vecteur et déplace les éléments suivants

Parcours

1. Boucle for :

```
for (int i = 0; i < v.size(); i++)
  std::cout << v[i];
```

2. Iterator : un itérateur est un objet qui pointe vers des éléments d'un tableau, d'un container (map, vector, ...), ...

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

3. Range based loop (depuis C++ 11) :

```
for (int i : v)
  std::cout << i;
```

Range based for loop

```
for (int i : v)
  std::cout << i;
```

- i : une variable dont le type est celui des éléments de la séquence représentée par v

Range based for loop

```
for (int i : v)
  std::cout << i;
```

- i : une variable dont le type est celui des éléments de la séquence représentée par v

```
for (auto i : v)
  std::cout << i;
```

- auto : Spécifie que le type de la variable sera déduit automatiquement lors de son initialisation.
- Depuis C++ 11

Plus de détails sur la fonction `erase`

- Suppression d'un seul élément :

```
v.erase (v.begin()+2);
```

- Suppression des éléments entre deux positions :

```
v.erase (v.begin(),v.begin()+3);
```

Remarque : La fonction retourne l'iterateur pointant sur l'élément qui suit le dernier élément supprime

Paramètres d'une fonction (Chapitre 8)

- Passage par valeur :

```
void print(vector<double> v)
{
  for(auto i : v)
    cout << i << ',';
}
```

Paramètres d'une fonction (Chapitre 8)

- Passage par valeur :

```
void print(vector<double> v)
{
  for(auto i : v)
    cout << i << ',';
}
```

  - Copie des paramètres
  - Modification locale à la fonction
  - ☹ Et si la variable est un vecteur de 8MB ?
  - ☹ En plus, on ne fait qu'afficher ce vecteur

Passage par const-reference

```
void print(const vector<double>& v)
  {
    for(auto i : v)
      cout << i << ',';
  }

void f(int x)
  {
    vector<double> v1(100);
    ...
    print(v1);
  }
```

- v est fait référence (*refers back*) à la variable définie ailleurs
- Pas de copie
- `const` : Étant donné que v n'est pas modifiée dans cette fonction, le mot clé const permet d'éviter qu'on la modifie par erreur

Passage par reference

```
void print(vector<double>& v)
    {
      for(int i=0; i < v.size(); i++)
          v[i] ++;
    }
```

- v est fait référence (*refers back*) à la variable définie ailleurs
- Pas de copie
- Une modification du paramètre est visible à l'extérieur de la fonction

Quel passage de paramètre choisir ?

- Objets très petits (un ou deux doubles ou entiers) : par valeur
- Objets larges pas modifiés : par const-reference
- Préférez retourner un résultat au lieu de passer le paramètre par référence
- Utiliser le passage par référence seulement quand il le faut. Par exemple :
  - Pour manipuler les vecteurs et autres objets larges,
  - pour des fonctions modifiants plusieurs objets car il n'est pas possible de retourner plus d'un objet. Il est cependant déconseillé d'écrire des fonctions modifiants plusieurs objets.

Et le passage par adresse dans tout ça ?

- Préférez le passage par référence
- Utilisez le passage par adresse quand c'est nécessaire

```
f(T *t) /* a utiliser si t peut etre null car une
    reference ne peut pas etre null*/
```

Quelques remarques sur les références

```
void main
{
  int i = 7;
  int& r = i; /* r est une reference vers i */
  r = 9; /* i devient 9*/
  i = 10; /* i et r valent 10*/
  int j = 0;
  r = j; /* On ne peut pas reassigner les
      references. r est toujours une reference sur
      i. r et i valent 0 mais &r et &i sont les
      memes*/
}
```

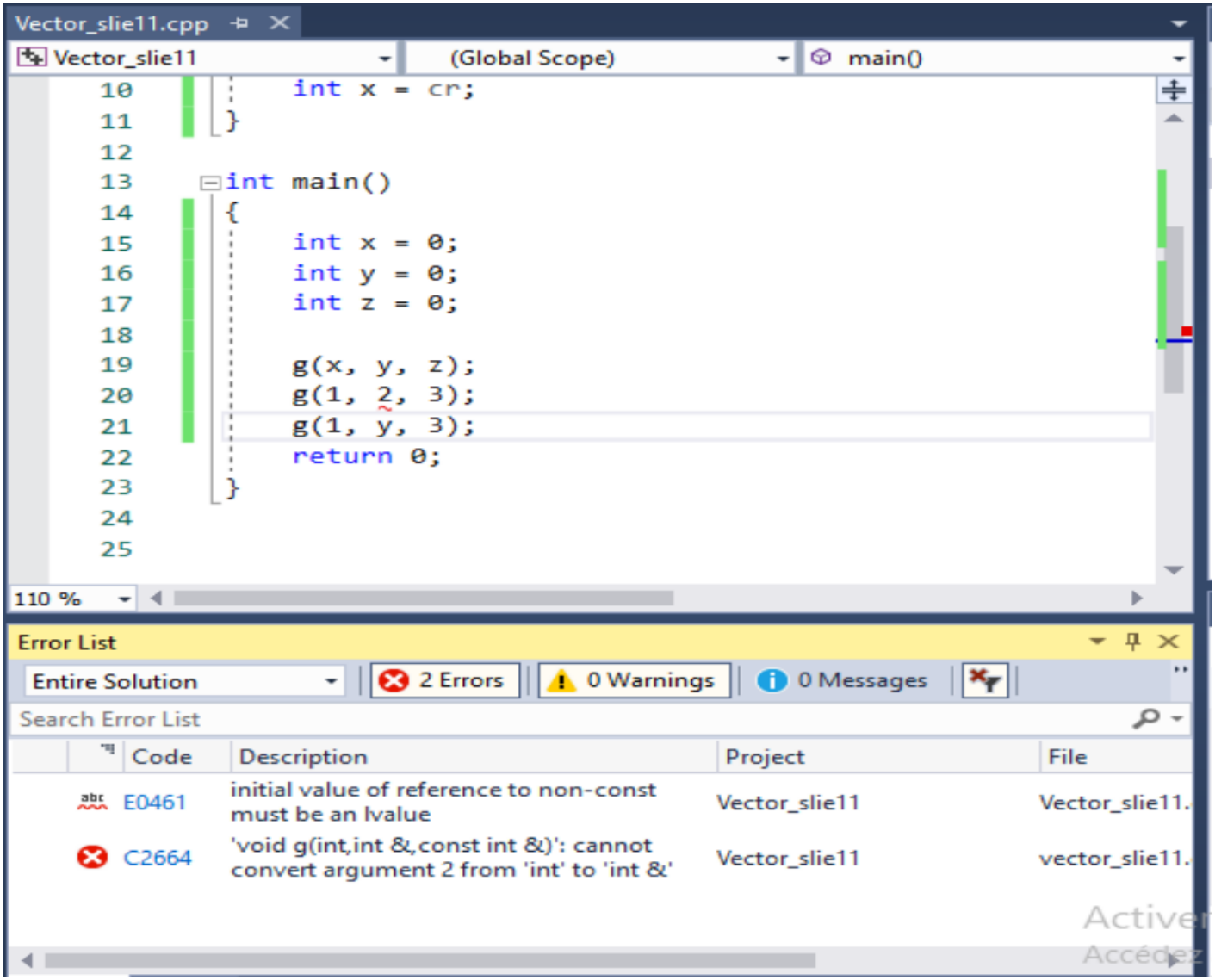
Quelques remarques sur les références

```
void g(int a, int &r, const int& cr)
{
  ++a;
  ++r;
   int x = cr;
}
int main()
{
   int x = 0;
   int y = 0;
   int z = 0;

   g(x, y, z); /*x = 0, y = 1, z = 0*/


}
```

Quelques remarques sur les références

```
void g(int a, int &r, const int& cr)
{
  ++a;
  ++r;
   int x = cr;
}
int main()
{
   int x = 0;
   int y = 0;
   int z = 0;

   g(x, y, z); /*x = 0, y = 1, z = 0*/
   g(1,2,3);/* erreur : une reference a besoin d'une variable qu'elle
        reference*/
}
```

Quelques remarques sur les références



Quelques remarques sur les références

```
void g(int a, int &r, const int& cr)
{
  ++a;
  ++r;
   int x = cr;
}
int main()
{
   int x = 0;
   int y = 0;
   int z = 0;

   g(x, y, z); /*x = 0, y = 1, z = 0*/
   g(1,2,3) ;/* erreur : une reference a besoin d'
       une variable qu'elle reference*/
   g(1,y,3); /* ok pour le const-ref */
}
```

Revenons à auto

```
for(auto i : v) /* une copie de chaque element est
    creee : on ne peut pas changer les elements de
    v en modifiant i*/
  ...

for(auto& i : v) /* a utiliser lorsqu'on veut
    modifier les valeur de v */
  ...

for (const auto& i : v) /* lorsqu'on accede aux
    elements de v en lecture seule*/
```

Remarque : Le fonctionnement est le même si le type est utilisé au lieu de auto

Pour plus de détails `https://blog.petrzemek.net/2016/08/17/auto-type-deduction-in-range-based-for-loops/`

À lire pour la séance prochaine

Chapitre 21