MODULE 0

C++11 : Bref aperçu

Objectifs de ce module :

- ✓ ***Aperçu C++11***

Table des matières

Sujets ***Page***

Introduction

Quelques nouveautés

Le chevron >>

Un petit irritant qui est maintenant réglé en C++11 est la possibilité d'utiliser le double crochet >> pour fermer la définition lors de l'usage des templates. Les compilateurs C++ actuels traitent toujours une séquence de deux signes *supérieur à* comme un opérateur de décalage binaire vers la droite. En conséquence, lors de l'imbrication de l'utilisation de patrons, les programmeurs sont obligés d'insérer un espace entre les deux chevrons fermants.

Par exemple, en C++03, ce code provoque une erreur de compilation :

```
#include <vector>
std::vector<std::vector<int>> matrix;
// Attention ! Écrire plutôt : “std::vector<std::vector<int> >”
```

C++11 tentera de détecter automatiquement si les symboles doivent jouer le rôle de chevrons fermants ou d'opérateur de décalage binaire.

C++03 avant	C++11 maintenant

#include <vector>
 	
using namespace std;
 	
int main()
 	
{
 	
vector<vector<float> > v;
 	
}
	#include <vector>
	
	using namespace std;
	
	int main()
	
	{
	
	vector<vector<float>>// OK C++11
	
	}

Le pointeur constant « nullptr »

Ce pointeur constant a été ajouté pour éviter certaines ambiguïtés tel que montré ci-dessous.

C++03 avant	C++11 maintenant

void fonc(char *);
void fonc (int);
int main()
{
char c = '*';
int i = 5;
fonc(i); // appelle fonc(int)
fonc(&c); // appelle fonc(char *)
fonc(0); // ambiguïté...
}
	void fonc(char *);
	void fonc (int);
	int main()
	{
	char c = '*';
	int i = 5;
	fonc(i); // appelle fonc(int)
	fonc(&c); // appelle fonc(char *)
	fonc(nullptr); // appelle fonc(char *)
	}

Le nouveau mot-clé nullptr a été proposé comme constante du langage avec le caractère particulier d'être assignable à tous les types de pointeurs.

En effet, contrairement au C où la macro préprocesseur est généralement définie avec #define NULL ((void*)0), en C++ il est interdit d'assigner un void* à un pointeur d'un type différent. L'usage était donc de définir NULL avec l'entier 0. Ce comportement restera compatible, mais il sera aussi possible d'écrire :

T* ptr = nullptr;

La constante NULL définie comme l'entier 0 ne permettait pas au compilateur de déterminer quelle surcharge de f choisir dans le code suivant:

```
void f(int);
void f(void*);
 f(0); // Entier 0 ou pointeur nul?
```

Le mot clé nullptr est une constante du type nullptr_t, non convertible en entier. Pour appeler la fonction f avec un pointeur NULL, la surcharge est correctement choisie en C++11 dans le code suivant:

```
void f(int);          f(0);  // Entier 0, pas d,ambiguïté
void f(void*);        f(nullptr); // Convertible en void*, mais pas en int
```

Le mot-clé « auto »

C'est un des ajouts qui était déjà présent lors de l'élaboration du standard de C++03 mais qui a été consolider davantage en C++11.

Le mot clé **`auto`** se voit assigner une nouvelle sémantique par le nouveau standard. Nous connaissions son unique sémantique d'indicateur de classe de stockage pour une variable. En effet, déclarer une variable automatique revenait à indiquer au compilateur qu'elle était valide seulement dans l'espace où elle était déclarée ; ce comportement étant aussi celui par défaut, le mot clé était inutile. Dans le nouveau standard, il change de sémantique et prend la place du type dans la déclaration. Le type sera alors automatiquement décidé par correspondance avec le type retourné par l'objet utilisé pour l'initialisation de la variable.

Les variables étant déclarées avec `auto` devront donc impérativement être initialisées. Exemple :

```
void f();

X *g(int);

auto a = 3.14159; // a est double

auto b = 0; // b est int

auto c; // illégal

auto d = f(); // illégal

auto e = g(3); // e est X*
```

Mais son usage devient vraiment incontournable lorsqu'il s'agit de boucler à l'aide d'itérateurs.

Voyons un exemple :

C++03	std::vector<int> vect;
	// ...
	vect = {1,2,3,4,5};
	for(std::vector<int>::const_iterator cit = vect.begin(); cit != vect.end();cit++)
	{
	  std::cout << *cit;
	}

C++11	std::vector<int> vect;
	// ...
	vect = {1,2,3,4,5};
	
	for(auto it = vect.begin(); it != vect.end(); it++)
	{
	  std::cin >> *it;
	}

Le mot clé auto permet donc de déduire le type d'une donnée en fonction du type de l'expression pour l'initialiser.

Il y avait une redondance dans le code en c++03 qui venait du fait que l'on répétait la portion de la déclaration suivante « std::vector<int>::const_iterator ».

On peut faire encore plus simple s'il s'agit de passer des éléments d'un tableau ou d'un vecteur comme nous le verrons plus loin.

Les listes d'initialisateur

On peut maintenant facilement initialiser un vecteur à la déclaration en utilisant la notation { et }. Exemple :

C++03	C++11

#include <vector>
 	
using namespace std;
 	
int main()
 	
{
 	
int tab[] = {2,3,5,7,11};
 	
vector<int> v;
 	
v.push_back(2);
 	
v.push_back(3);
 	
v.push_back(5);
 	
v.push_back(7);
 	
v.push_back(11);
 	
}
	#include <vector>
	
	using namespace std;
	
	int main()
	
	{
	
	int tab[] ={2,3,5,7,11};
	
	vector<int> v ={2,3,5,7,11};
	
	}

Les boucles « range-for »

Une des belles caractéristiques du C++11. Vous vous souvenez que l'on utilisait le mot-clé auto lors du parcours d'un vecteur avec un itérateur. Dans le cas des boucles « range-for », nous utiliserons à nouveau le mot-clé auto pour parcourir la liste des éléments.

Que faisait-on en C++03 pour parcourir les éléments d'un tableau? Quelque chose comme ceci :

C++03	C++11

#include <iostream>
 	
#include <vector>
 	
using namespace std;
 	
int main()
 	
{
 	
int x[10] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
 	
int nbElem = sizeof(x) / sizeof(int);
 	
for (int i = 0; i < nbElem; i++)
 	
{
 	
cout << "i = " << x[i] << endl;
 	
}
 	
}
	#include <iostream>
	
	#include <vector>
	
	using namespace std;
	
	int main()
	
	{
	
	vector<int> v = {1,2,3,4,5};
	
	int x[10] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
	
	for(auto i : x) {
	cout << i << " ";
	}
	
	for (auto i : v)
	
	{
	
	cout << i << endl;
	
	}
	
	for (auto &i : v)
	
	{
	
	i*=2;
	
	}
	
	for (auto i : v)
	
	{
	
	cout << i << " ";
	
	}
	
	}

Dans le code précédent, il est intéressant de noter l'usage du caractère « & » dans le code suivant :

```
for (auto &i : v)
  {
      i*=2;
  }
```

L'éperluette permet donc de travailler avec la référence et permet donc de modifier la valeur de chaque élément du vecteur. Dans l'exemple, chaque élément est doublé.

Quelques ajouts intéressants

<array>

Les « array » sont aussi efficace en terme d'espace mémoire que les tableaux statiques usuels accédés par les []. En fait, cette classe ajoute une couche de méthodes et de fonctions globales de façon telle que les array peuvent être utilisés comme conteneur standard.

Contrairement aux autres conteneurs standards, les arrays ont une longueur fixe et ne peuvent donc pas dynamiquement se contracter ou s'étendre.

Exemple :

```
#include <array>
#include <iostream>

using namespace std;

int main()
{
        array<int,5> tab ={2,3,5,7,11};

        array<double,4> Tab = {5.1, -1.0, 5.56, 6.11};

         int x[10] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };


        cout << "Nombre d'élément de array Tab: " << Tab.size() << endl;

        cout << "Tab.at(1) :" << Tab.at(1) << "  ou  Tab[1] :" << Tab[1] << endl;

        cout << "Premier élément de Tab : " <<Tab.front() << endl;   // 5.1
        cout << "Dernier élément de Tab : " << Tab.back() << endl;     // 6.11
}
```

<chrono>

Chrono est le nom de l'entête mais aussi le nom du sous-espace de nom. Tous les éléments dans cet espace de nom sont définis directement sous l'espace de nom std::chrono.

Exemple d'un calcul de temps entre 2 instructions :

C++03	C++11

int main()
{
clock_t avant = clock();
// traitement
clock_t apres = clock();
clock_t elapsed =apres –avant;
cout << static_cast<double
	#include <iostream>
	#include <chrono>
	
	using namespace std;
	using namespace std::chrono;
	
	int main()

```
>(elapsed)/CLOCKS_PER_SEC* 1000.0

<< " ms." << endl;

}
```

```
{

  auto avant = system_clock::now();

  for (int i = 0; i <2000; i++)

  {

    for (int j = 0; j < i; j++)

   {

       cout << "*";

   }

       cout << endl;

  }

  auto apres = system_clock::now();

 cout << "Temps pour affichele pattern: " <<
duration_cast<milliseconds>(apres-avant).count()  <<
"ms" << endl;

}
```

Exemple pour provoquer un temps d'attente de 300 ms

```
#include <iostream>
#include <chrono>
#include <thread>

using namespace std;
using namespace std::chrono;

int main()
{
        // définition d'une unité de temps
        milliseconds attente_300ms(300);  // 300 ms

        auto avant = system_clock::now();

        this_thread::sleep_for(attente_300ms);

        auto apres = system_clock::now();

        cout << "Temps d'attente de " << duration_cast<milliseconds>(apres-avant).count()  << "
ms"<< endl;


}
```

Génération de nombre aléatoire <random>

Cet entête contient les éléments pour générer des nombres aléatoires.

Cette bibliothèque permet de produire des nombres aléatoires en utilisant des combinaisons de générateurs et de distributions.

- Generateurs: Objets générant des nombres aléatoires uniformes.
- Distributions: Objets qui transforme une séquence de nombre aléatoire généré par un générateur dans une séquence qui suit une variable de distribution spécifique comme des distributions de type uniforme, normale ou binomiale.

Exemple d'une génération d'un nombre aléatoire entre 1 et 6 :

```
#include <iostream>
#include <random>

using namespace std;


int main()
{

   std::random_device  rd;

  std::uniform_int_distribution<int> distribution(1,6);
  std::default_random_engine generateur(rd());

  auto des = distribution(generateur);  // génère un nombre aléatoire entre 1 et 6

  cout << "Valeur du dé: " << des << endl;


}
```

Quelques éléments de <algorithm>

partition

Réorganise les éléments d'un intervalle situé entre deux éléments déterminés en faisant en sorte que les premiers éléments qui répondent au critère se retrouvent avant ceux qui ne répondent pas au critère. Partition retourne un itérateur qui pointe au premier éléments du second groupe.

Exemple :

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

using namespace std;

bool EstInferieurA10 (int i)
{
   return( i < 10);
}


int main()
{
        vector<int> MonVecteur = {20, 5, 11, -8, 25};

        auto  Position = partition(MonVecteur.begin(), MonVecteur.end(), EstInferieurA10);

        for(auto i : MonVecteur)
        {
          cout << i << " | ";
        }
        cout << " La position de l'autre groupe de valeur :  " << distance(MonVecteur.begin(), Position);
}
```

Le contenu du vecteur sera égal à : {-8, 5, 11, 20, 25}

L'exécution de ce programme nous donne :

-8 | 5 | 11 | 20 | 25 | La position de l'autre groupe de valeur : 2

La position de l'autre groupe de valeur qui ne réponde pas au critère.

sort

Tri les éléments d'un groupe en ordre croissant ou selon l'ordre spécifié par la fonction donnée en 3e paramètre..

Exemple1 : Trier en ordre croissant

```
vector<int> MonVecteur = {32,71,12,45,26,80,53,33};

sort(MonVecteur.begin(), MonVecteur.end());  // Trie en ordre croissant par défaut.
```

Le contenu du vecteur : {12, 26, 32, 33, 45, 53, 71, 80}

Exemple 2 : Trier seulement les éléments 2 à 4

```
sort(begin(MonVecteur)+2, begin(MonVecteur)+4);
```

Exemple 3 : Trier en ordre décroissant en passant par une fonction

```
bool Decroissant (int i, int j)
{
   return( i >j);
}
  vector<int> MonVecteur = {32,71,12,45,26,80,53,33};

  // Trie en ordre décroissant par l'intermédiaire d'une fonction.
 sort(MonVecteur.begin(), MonVecteur.end(), Decroissant);
```

On obtient :

25 | 20 | 11 | 5 | -8

minmax_element

Trouve le plus grand et le plus petit élément dans un intervalle de valeurs.

Les résultats sont emmagasinés dans une paire d'itérateur qui contient l'emplacement du plus petit (first) et l'emplacement du plus grand (second). On doit ensuite utiliser l'étoile (*) pour aller chercher les contenus de ces itérateurs si on veut faire afficher les valeurs.

Exemple 1 : Faire afficher le plus petit et le plus grand élément d'un vecteur

```
vector<int> MonVecteur = {-2, 9, -11, 99, 11, 87};

auto Resultat = minmax_element(MonVecteur.begin(), MonVecteur.end());

cout << "Le plus petit : " << *Resultat.first<< endl;
cout << "Le plus grand : " << *Resultat.second << endl;
```

Exercices

Complétez le programme suivant en complétant les éléments demandés dans ce programme:

```
//
// Incluez ce dont vous avec besoin
//


#include <iostream>
#include <chrono>
using namespace std;


int main()
{
  cout << "Combien d'elements? ";
  int n;
  if (!(cin >> n) || n < 0) return -1;
  //
  // Créez un conteneur capable de contenir n éléments  vecteur ou array
  //


  //
  // Générez n entiers aleatoires entre 1 et 100 dans ce conteneur
  //
  // <random>, <algorithm>
  // Utilisez uniform_int_distribution, default_random_engine et generate_n()
  //


  //
  // Partitionnez le conteneur pour que les pairs soient d'un côté et les impairs de l'autre
  //
  // <algorithm>
  // Utilisez partition()
```

```
//

//
// Triez les pairs en ordre croissant
//
// <algorithm>
// Utilisez sort()
//

//
// Triez les impairs en ordre decroissant
//
// <algorithm>
// Utilisez sort()
//

//
// Affichez les elements du conteneur, avec " | " entre chacun
//
// <algorithm>
// Utilisez for_each()
//

//
// Transformez chaque nombre en son négatif
//
// <algorithm>
// Utilisez transform()
//

//
// Affichez les elements du conteneur, avec " | " entre chacun
```

//

// <algorithm>

// Utilisez for_each()

//

//

// Transformez chaque nombre en son négatif

//

// Utilisez une boucle for simplifiée

//

//

// Affichez les elements du conteneur, avec " | " entre chacun

//

// <algorithm>

// Utilisez for_each()

//

//

// Mélangez les

//

// <algorithm>

// Utilisez random_shuffle()

//

//

// Affichez les elements du conteneur, avec " | " entre chacun

//

// <algorithm>

// Utilisez for_each()

//

```
  //
  // Indiquez la position et la valeur du plus petit et du plus gros élément
  //
  // <algorithm>
  // Utilisez minmax_element()
  //


  //
  // Y a-t-il un élément qui soit divisible par trois? Si oui, lequel est-ce et à quel indice est-il?
  //


}
```