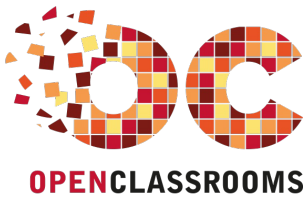
Introduction à C++ 2011 (C++0x)

Par antoyo



www.openclassrooms.com

Dernière mise à jour le 16/08/2011

Sommaire

Sommaire 2
Lire aussi 1
Introduction à C++ 2011 (C++0x) 3
Pourquoi une nouvelle révision du langage C++ ? 3
Préparation du compilateur pour utiliser C++ 2011 4
Installation 4
Configuration de la compilation 4
Les énumérations fortement typées 5
Les énumérations 5
Les énumérations fortement typées 6
Le « bug » des chevrons 7
Les listes d'initialisateurs 7
Côté utilisateur 7
Côté créateur 8
Le mot-clé auto 8
La boucle basée sur un intervalle 12
Initialisation d'un pointeur 13
Les fonctions anonymes et les fermetures 14
Fonctions anonymes 14
Fermetures 15
Les tableaux à taille fixe 17
Déclaration et initialisation 17
Méthodes 18
Exemple 18
Un nouveau type de conteneur : le tuple 19
Déclaration et initialisation 19
Accéder aux éléments du tuple 20
Exemple 20
Gestion du temps 21
Obtenir le temps actuel 21
Autres unités 22
Les initialisateurs d'attributs 23
Partager 25

Introduction à C++ 2011 (C++0x)

Par antoyo

Mise à jour : 16/08/2011
Difficulté : Intermédiaire Durée d'étude : 3 heures

Bonjour à tous !
Vous avez entendu parler de la nouvelle norme C++, ces derniers temps ?
C++0x et C++1x vous disent-ils quelque chose ?
Vous voulez en apprendre plus sur cette norme ?
Alors, vous êtes au bon endroit !

Ce tutoriel a pour but de vous montrer quelques nouveautés de la norme C++ 2011 qui sont déjà utilisables par les compilateurs actuels.
Sommaire du tutoriel :

- Pourquoi une nouvelle révision du langage C++ ?
- Préparation du compilateur pour utiliser C++ 2011
- Les énumérations fortement typées
- Le « bug » des chevrons
- Les listes d'initialisateurs
- Le mot-clé auto
- La boucle basée sur un intervalle
- Initialisation d'un pointeur
- Les fonctions anonymes et les fermetures
- Les tableaux à taille fixe
- Un nouveau type de conteneur : le tuple
- Gestion du temps
- Les initialisateurs d'attributs

Pourquoi une nouvelle révision du langage C++ ?

La norme actuelle du C++ date de 1998, puis a été mise à jour en 2003.
Nous pouvons nous demander pourquoi faire une nouvelle norme maintenant.

Pour répondre à la question, je cite Wikipédia :

Citation : Wikipédia

> Un langage de programmation comme le C++ suit une évolution qui permettra aux programmeurs de coder plus rapidement, de façon plus élégante et permettant de faire du code maintenable.

Certaines nouveautés permettent en effet d'écrire du code C++ plus simple, comme vous le verrez dans quelques instants.

En outre, le matériel informatique évoluant constamment, en particulier les processeurs, les besoins sont maintenant différents en matière de programmation.
Ayant aujourd'hui plusieurs cœurs, les processeurs peuvent exécuter plusieurs instructions en même temps.
Des classes permettant l'utilisation de la programmation concurrente sont ainsi apparues dans cette norme du C++.

Est-ce que la nouvelle norme du C++ est rétro-compatible avec l'ancienne ?

Presque.

La plupart du code déjà écrit n'aura donc pas besoin d'être modifié.

Préparation du compilateur pour utiliser C++ 2011

Attention : tout ceci est expérimental et ne devrait pas être utilisé en production.
Utilisez ces nouveautés à vos risques et périls.

De plus, les compilateurs actuels n'implémentent pas toutes les nouveautés de la norme C++0x.
Les codes de ce tutoriel ont été testés avec g++ 4.6 sous GNU/Linux.

Je vais vous montrer comment configurer g++ 4.6 sur Ubuntu pour utiliser les nouveaux standards du C++.

Installation

Premièrement, ajoutez ce ppa à vos sources :

Code : Console

```
deb http://ppa.launchpad.net/ubuntu-toolchain-
r/test/ubuntu YOUR_UBUNTU_VERSION_HERE main
```

en modifiant YOUR_UBUNTU_VERSION HERE par votre version d'Ubuntu (natty, maverick, lucid ou karmic).

Vous pouvez maintenant mettre à jour le compilateur g++ :

Code : Console

```
sudo apt-get install g++
```

Après l'installation, tapez la commande suivante pour vérifier que vous avez au moins la version 4.6 de g++ :

Code : Console

```
g++ -v
```

La dernière ligne de la sortie devrait vous le confirmer :

Code : Console

```
gcc version 4.6.1 20110409 (prerelease) (Ubuntu 4.6.0-3~ppa1)
```

Configuration de la compilation

Maintenant, vous allez essayer de compiler ce programme :

Code : C++

```
#include <iostream>

int main() {
    int tableau[5] = {1, 2, 3, 4, 5};
    for(int &x : tableau) {
        std::cout << x << std::endl;
    }
    return 0;
}
```

en utilisant la ligne de commande habituelle :

Code : Console

```
g++ main.cpp -o BoucleFor
```

J'ai la bonne version de g++, mais j'obtiens tout de même une erreur du compilateur.
Que dois-je faire ?

Vous devez simplement lui demander poliment de compiler en utilisant la nouvelle norme.
Sinon, vous pouvez avoir une erreur de ce type :

Code : Console

```
main.cpp: In function 'int main()':
main.cpp:5:18: error: range-based-for loops are not allowed in C++98 mode
```

Parfois, vous pouvez avoir un avertissement du genre :

Code : Console

```
main.cpp:3:1: warning: scoped enums only available with -std=c++0x or -
std=gnu++0x [enabled by default]
```

qui vous indique un peu mieux que faire.

Lancez donc la dernière commande avec un nouvel argument :

Code : Console

```
g++ -std=c++0x main.cpp -o BoucleFor
```

Et voilà, ça compile !

Le précédent code source vous met-il l'eau à la bouche ?
Si c'est la cas, alors continuez votre lecture, vous n'avez pas terminé de découvrir de nouvelles façons d'écrire du code plus simplement qu'avant !

Les énumérations fortement typées

Pour commencer, voyons quelque chose de facile : les énumérations fortement typées.

Les énumérations

Vous connaissez déjà les énumérations, que l'on peut créer comme ceci :

Code : C++

```
enum Direction { HAUT, DROITE, BAS, GAUCHE };
```

On peut ensuite créer une variable utilisant cette énumération :

Code : C++

```
Direction direction = HAUT;
```

Puis, nous pouvons afficher cette variable :

Code : C++

```
std::cout << direction << std::endl;
```

Cela fonctionne, car il y a une conversion implicite vers un entier.
C'est comme si nous avions fait ceci :

Code : C++

```
std::cout << static_cast<int>(direction) << std::endl;
```

Les énumérations fortement typées

Passons aux nouveautés.
Les énumérations fortement typées se créent en ajoutant le mot-clé **class** après **enum**. :

Code : C++

```
enum class Direction { Haut, Droite, Bas, Gauche };
```

Par la suite, pour l'utiliser, il faut utiliser le nom de l'énumération, suivi par l'opérateur de résolution de portée et du nom de la valeur que l'on souhaite utilisée :

Code : C++

```
Direction direction = Direction::Haut;
```

C'est la première différence avec les énumérations que vous connaissez.

La seconde, c'est qu'il n'y a pas de conversion implicite vers un entier.
Ainsi, le code suivant ne compilera pas :

Code : C++

```
std::cout << direction << std::endl;
```

Pour qu'il compile, il faut **explicitement** convertir la variable en entier :

Code : C++

```
std::cout << static_cast<int>(direction) << std::endl;
```

Mais, à quoi ça sert d'utiliser ces énumérations ?
C'était bien plus simple avant, non ?

Ces énumérations n'ont pas été créées pour le simple plaisir de compliquer les choses.
Elles permettent d'assurer une certaine sécurité en codant.

En effet, vous ne voulez sûrement pas faire des opérations mathématiques avec votre énumération.
Le code suivant n'aurait aucun sens :

Code : C++

```
int nombre = 5 + Direction::Droite;
```

Ce code provoque une erreur de compilation et c'est bien, car ça n'a pas de sens d'ajouter une direction à un nombre.

Le « bug » des chevrons

Vous souvenez-vous de la façon de créer un vecteur de vecteurs ?
Il y a un petit point à ne pas oublier ; il faut mettre un espace entre les deux derniers chevrons :

Code : C++

```
std::vector<std::vector<int> > nombres;
```

Sinon, >> pourrait être confondu avec l'opérateur de flux.

Eh bien, savez-vous quoi ?
Il est maintenant possible d'écrire ceci en C++0x :

Code : C++

```
std::vector<std::vector<int>> nombres;
```

Il n'y a plus d'erreur de compilation.

Les listes d'initialisateurs

Côté utilisateur

Pour créer un vecteur avec différentes valeurs initiales, il faut écrire plusieurs lignes de code :

Code : C++

```
std::vector<int> nombres;
nombres.push_back(1);
nombres.push_back(2);
nombres.push_back(3);
nombres.push_back(4);
nombres.push_back(5);
```

Il est maintenant possible de faire beaucoup plus court avec une liste d'initialisateurs :

Code : C++

```
std::vector<int> nombres = { 1, 2, 3, 4, 5 };
```

C'est comme si on initialisait un tableau normal :

Code : C++

```
int nombres2[] = { 1, 2, 3, 4, 5 };
```

sauf qu'on peut le faire avec les conteneurs de la STL.

Voici un code complet présentant comment utiliser une liste d'initialisateurs avec un `std::map` :

Code : C++

```
#include <iostream>
#include <map>
#include <ostream>
#include <string>
```

```
int main() {
    std::map<int, std::string> nombres = { { 1, "un" }, { 2, "deux"
}, { 3, "trois" } };
    std::cout << nombres[1] << std::endl;
}
```

Simple, non ?

Côté créateur

Vous avez utilisé les listes d'initialisateurs, mais aimeriez-vous créer des fonctions qui les utilisent ?
C'est ce que nous allons voir ici.

Nous allons créer une fonction très banale qui ne fait qu'afficher les éléments passés à la fonction à l'aide de la liste d'initialisateurs.

Notre fonction `main()` sera :

Code : C++

```
int main() {
    afficherListe({ 1, 2, 3, 4, 5 });
    return 0;
}
```

Nous souhaitons implémenter la fonction `afficherListe()` qui affichera les éléments envoyés.

La fonction aura l'allure suivante :

Code : C++

```
void afficherListe(std::initializer_list<int> liste) {
    //Affichage de la liste.
}
```

Étant donné que nous avons indiqué `int` entre les chevrons, nous acceptons seulement les entiers dans la liste.

Le code de la fonction est donc :

Code : C++

```
void afficherListe(std::initializer_list<int> liste) {
    for(std::initializer_list<int>::iterator i(liste.begin()) ; i !=
liste.end() ; ++i) {
        std::cout << *i << std::endl;
    }
}
```

N'oubliez pas d'inclure l'entête :

Code : C++

```
#include <initializer_list>
```

Le mot-clé auto

Le mot-clé **auto** est un mot-clé utilisable en C++0x qui est différent du mot-clé **auto** qui était utilisé avant. Il est possible de le mettre à la place du type de telle sorte que ce type est automagiquement déterminé à la compilation en fonction du type retourné par l'objet utilisé pour l'initialisation. 

On parle ici d'inférence de types.

Donc, il est possible d'écrire le code suivant :

Code : C++

```
auto nombre = 5;
```

La variable nombre sera de type entier (`int`).
Mais, je vous interdis de faire cela ! 
Si vous initialisez toutes vos variables avec le mot-clé **`auto`**, votre code va vite devenir illisible.

Vous devez utiliser ce mot-clé le moins possible.

As-tu des exemples d'utilisation ?

Oui, en voilà un :

Code : C++

```
for(auto i(nombres.begin()) ; i != nombres.end() ; ++i) {
    std::cout << *i << std::endl;
}
```

Au lieu de cela :

Code : C++

```
for(std::vector<int>::iterator i(nombres.begin()) ; i !=
nombres.end() ; ++i) {
    std::cout << *i << std::endl;
}
```

Mais, encore là, c'est limite.

Si vous utilisez beaucoup de fois un même itérateur, le mieux est de créer un **`typedef`** :

Code : C++

```
typedef std::vector<int>::iterator iter_entier;
for(iter_entier i(nombres.begin()) ; i != nombres.end() ; ++i) {
    std::cout << *i << std::endl;
}
```

Vous pouvez également utiliser ce mot-clé lorsqu'une même fonction peut retourner différents types de données.

Vous devez faire attention à deux points en particulier :

1. Le mot-clé **`auto`** n'est pas comme un type qui change au cours de l'exécution du programme. Si j'assigne un entier à une variable dont le mot-clé **`auto`** est utilisé à l'initialisation, cette variable sera et restera un entier.
2. Vous devez obligatoirement *initialiser* une variable déclarée avec le mot-clé **`auto`** à la déclaration. Autrement dit, il est impossible de faire :

Code : C++

```
auto variable;
```

Si je déclare une variable avec le mot-clé **auto**, puis-je donner le même type à une autre variable ?

Oui, avec le mot-clé `decltype` :
Ce mot-clé détermine le type d'une expression.

Code : C++

```
auto variable = 5;
decltype(variable) autreVariable;
```

Ainsi, nous sommes sûr que `autreVariable` aura le même type (`int`) que `variable`.

L'inférence de type est très utile lorsque nous utilisons la programmation générique.
Considérez ce programme :

Code : C++

```
#include <iostream>

template <typename T>
T maximum(const T& a, const T& b) {
    if(a > b) {
        return a;
    }
    else {
        return b;
    }
}

template <typename T>
T minimum(const T& a, const T& b) {
    if(a < b) {
        return a;
    }
    else {
        return b;
    }
}

int main() {
    int a(10), b(20);
    auto plusGrand = maximum(a, b);
    decltype(plusGrand) plusPetit = minimum(a, b);
    std::cout << "Le plus grand est : " << plusGrand << std::endl;
    std::cout << "Le plus petit est : " << plusPetit << std::endl;
    return 0;
}
```

Ce code détermine, grâce à des fonctions génériques, le plus grand et le plus petit nombres.

Ce qui est intéressant, c'est que nous n'avons besoin que de modifier la première ligne de la fonction `main()` si nous voulons essayer ce code avec un autre type.
Remplacez-la par celle-ci et tout fonctionne :

Code : C++

```
double a(10.5), b(20.5);
```

De plus, les mots-clés **auto** et `decltype` sont obligatoires dans certains cas utilisant la programmation générique.

Considérons l'exemple suivant (assez tordu, je dois l'admettre) :
Vous avez un programme de gestion des notes obtenues à l'école.

Les examens comportent dix questions (chacune vaut 10 points).
Votre programme stocke les notes de deux manières :

- Un nombre entier sur 100 (le pourcentage) ;
- Un nombre réel sur 1 (le pourcentage divisé par 100).

Vous avez créé une fonction `ajouterDixPourcents()` surchargée pour les entiers et les réels :

Code : C++

```
int ajouterDixPourcents(int nombre) {
    if(nombre + 10 <= 100) {
        return nombre + 10;
    }
    else {
        return 100;
    }
}

float ajouterDixPourcents(float nombre) {
    if(nombre + 0.1 <= 1) {
        return nombre + 0.1;
    }
    else {
        return 1;
    }
}
```

Pour une raison inconnue  , vous créez également une fonction générique `ajouter()` qui appellera la fonction `ajouterDixPourcents()`.
Cette fonction doit retourner le même type que celui retourné par cette dernière.
Mais, comment faire pour déterminer ce type ?

Nous pourrions utiliser le mot-clé **auto**, non ?

Impossible, car il se base sur le type de l'objet qu'on lui affecte. Le compilateur ne saura pas sur quel objet se baser.

Avec `decltype()`, alors ?

Encore une fois, c'est impossible.
En effet, quelle expression lui passerions-nous pour déterminer le type de retour de la fonction `ajouter()` ?

La solution est d'utiliser les deux mots-clés en utilisant une syntaxe alternative pour le prototype de la fonction ! (Vous ne pouviez pas vraiment le deviner. )

Voilà enfin la fonction `ajouter()` :

Code : C++

```
template <typename T> auto ajouter(T nombre) ->
decltype(ajouterDixPourcents(nombre)) {
    return ajouterDixPourcents(nombre);
}
```

Il est impossible d'utiliser `decltype(ajouterDixPourcents(nombre))` à la place du type de retour, car `nombre` n'a pas encore été parsé par le compilateur.

Vous pouvez maintenant utiliser votre fonction générique `ajouter()` dans votre code :

Code : C++

```
int main() {
    int pourcentageEntier(42);
    float pourcentageFlottant = pourcentageEntier / 100.f;
    std::cout << pourcentageFlottant << " = " << pourcentageEntier
<< "%" << std::endl;

    pourcentageEntier = ajouter(pourcentageEntier);
    pourcentageFlottant = ajouter(pourcentageFlottant);
    std::cout << pourcentageFlottant << " = " << pourcentageEntier
<< "%" << std::endl;
}
```

Étant donnée que nous utilisons **auto** et `decltype` dans une fonction dont le prototype est plutôt long, je vous donne un exemple plus simple pour comprendre :

Code : C++

```
auto cinq() -> int {
    return 5;
}
```

Nous utilisons le mot-clé **auto** à la place d'indiquer un type de retour.
Il faut donc préciser le type de retour après la parenthèse fermante sous la forme suivante :

Code : C++

```
-> type
```

type peut être un type prédéfini comme `int`, mais la plupart du temps, nous utiliserons `decltype(expression)`.

La boucle basée sur un intervalle

Vous vous rappelez du code montré dans la deuxième partie de ce tutoriel ?
Cette boucle ressemblant au **foreach** de nombreux autres langages ?

Eh bien, il est maintenant possible d'utiliser une telle boucle en C++ !

Alors qu'autrefois nous étions obligés d'utiliser l'algorithme `for_each` :

Code : C++

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

using namespace std;

void afficherElement(int element) {
    cout << element << endl;
}

int main() {
    vector<int> nombres = { 1, 2, 3, 4, 5 };
    for_each(nombres.begin(), nombres.end(), afficherElement);

    return 0;
}
```

Nous pouvons maintenant utiliser la syntaxe beaucoup plus simple de la boucle **for** :

Code : C++

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

using namespace std;

int main() {
    vector<int> nombres = { 1, 2, 3, 4, 5 };
    for(int &element : nombres) {
        cout << element << endl;
    }

    return 0;
}
```

Comment cela fonctionne-t-il ?

Prenons la ligne du **for** :

Code : C++

```
for(int &element : nombres)
```

Sur cette ligne, nous trouvons d'abord la déclaration de la variable qui prendra chacune des valeurs du vecteur situé après les deux-points (:).
Donc, au premier tour de boucle, `element` vaudra 1.
Au deuxième tour, `element` aura pour valeur 2.
Et ainsi de suite.

Étant donné que nous utilisons une référence sur l'élément, nous pouvons le modifier sans problème :

Code : C++

```
for(int &element : nombres) {
    ++element;
}
```

Pour parcourir un std::map, le principe est le même :

Code : C++

```
map<int, string> nombres = { { 1, "un" }, { 2, "deux" }, { 3,
"trois" } };
for(auto i : nombres) {
    cout << i.first << " => " << i.second << endl;
}
```

Initialisation d'un pointeur

En C, vous aviez l'habitude d'initialiser un pointeur avec la macro `NULL` :

Code : C

```
int *pointeur = NULL;
```

Mais, nous évitons ceci en C++, car nous détestons utiliser des macros.

Vous avez donc pris l'habitude d'initialiser vos pointeurs à 0 :

Code : C++

```
int *pointeur(0);
```

Ça fonctionne toujours, mais sachez que vous pouvez maintenant initialiser vos pointeurs avec le nouveau mot-clé `nullptr` :

Code : C++

```
int *pointeur(nullptr);
```

Maintenant, impossible de ne pas voir du premier coup d'œil qu'il s'agit d'un pointeur.

Enfin, sachez qu'il est impossible d'utiliser `nullptr` comme un entier, ce qui était possible avec NULL.
En effet, ce code compile :

Code : C++

```
int nul = NULL;
```

Mais, pas celui-ci :

Code : C++

```
int nul = nullptr;
```

Quelle en est la raison ?

Parce que `nullptr` est de type `std::nullptr_t` (un peu comme **`true`** et **`false`** sont de type `bool`), alors que `NULL` est une macro qui vaut `0`.

Les fonctions anonymes et les fermetures

Fonctions anonymes

Une fonction anonyme, communément appelée fonction lambda, est une fonction qui n'a pas … de nom.
C'est aussi simple que cela.
On peut envoyer de telles fonctions en paramètre à des fonctions ou bien les stocker dans une variable.

Par exemple, au lieu d'envoyer un foncteur à std::for_each, nous pouvons lui envoyer une fonction anonyme.
C'est pratique dans le cas où nous ne comptons pas réutiliser la fonction ailleurs.
Reprenons l'exemple donné plus haut et utilisons à la place une fonction anonyme :

Code : C++

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

using namespace std;

int main() {
    vector<int> nombres = { 1, 2, 3, 4, 5 };
    for_each(nombres.begin(), nombres.end(), [](int element) {
        cout << element << endl;
    });
```

```
    return 0;
}
```

Nous allons étudier la partie qui doit vous sembler étrange :

Code : C++

```
[](int element) {
    cout << element << endl;
}
```

Premièrement, il y a des crochets ; nous verrons plus tard à quoi ils servent.

Ensuite, il y a directement les paramètres de la fonction, sans le nom devant.
C'est tout à fait normal, les fonctions anonymes n'ont pas de nom.

Le reste est comme une fonction ordinaire : il y a l'accolade ouvrante, les instructions et l'accolade fermante.

Vous pouvez assigner une fonction anonyme à une variable, comme ceci :

Code : C++

```
std::function<int (int, int)> addition = [](int x, int y) -> int {
    return x + y;
};
```

Premièrement, il y a le type std::function<> que nous devons assigner à la variable.
Entre les chevrons, nous devons mettre le type de retour, suivi du type des paramètres, entre parenthèses, de la fonction anonyme.
En pratique, nous donnerons **auto** comme type à une variable dans laquelle nous voulons mettre une fonction anonyme :

Code : C++

```
auto addition = [](int x, int y) -> int {
    return x + y;
};
```

Dans cet exemple, il y a une petite nouveauté :

Code : C++

```
[](int x, int y) -> int {
    return x + y;
}
```

Après la parenthèse fermante, il y a `-> int`.
Ce code est facultatif et il indique le type de retour de la fonction anonyme.
Cela peut être utile de le mettre dans le cas où nous donnons le type **auto** à la variable ou lorsque nous envoyons une fonction anonyme à une autre fonction.
Ainsi, nous saurons rapidement quel type de variable retourne la fonction anonyme.

Si nous ne le mettons pas, le type sera calculé par `decltype()`.

Fermetures

Une fermeture est une fonction qui capture des variables à l'extérieur de celle-ci.
Il faut préciser quelles sont les variables capturées.

Nous devons le faire entre les crochets ([]) qui se trouvent au début d'une fonction anonyme.
Pour capturer un variable, il faut écrire sont nom entre crochet.

Sachez qu'il y a deux façons de capturer une variable :

- par référence : de cette façon, les modifications apportées à la variable s'appliqueront à la variable capturée ;
- par valeur : c'est une copie de la variable capturée ; les modifications n'affecteront pas cette dernière.

Pour capturer une variable par référence, il suffit de précéder son nom par l'esperluette (&), comme ceci :

Code : C++

```
[&somme](std::vector<int> vecteur {
    //...
}
```

Si nous voulons capturer une variable par valeur, il suffit d'écrire son nom :

Code : C++

```
[x](int &a) {
    a = x;
}
```

Voyons un exemple complet :

Code : C++

```
int somme(0);
std::function<void (std::vector<int>)> sommeVecteur =
[&somme](std::vector<int> vecteur) {
    for(int &x : vecteur) {
        somme += x;
    }
};

std::vector<int> nombres = { 1, 2, 3, 4, 5 };

sommeVecteur(nombres);
std::cout << somme << std::endl;
```

Nous avons la confirmation que la variable somme a bien été modifiée : le nombre 15 s'affiche.

Il est possible de capturer toutes les variables par référence ou par valeur.

Voici une liste pour en résumer la syntaxe :

- [] : pas de variable externe ;
- [x, &y] : x capturée par valeur, y capturée par référence ;
- [&] : toutes les variables externes sont capturées par référence ;
- [=] : toutes les variables externes sont capturées par valeur ;
- [&, x] : x capturée par valeur, toutes les autres variables, par référence ;
- [=, &x] : x capturée par référence, toutes les autres variables, par valeur.

Bien que cela peut paraître tentant, il vaut mieux éviter de capturer toutes les variables externes lorsque possible (c'est-à-dire tout le temps ).
En effet, cela pourrait créer des problèmes de nom avec des variables extérieures.

Bon, vous êtes prévenus, mais voici tout de même un exemple :

Code : C++

```
int x(10);
auto afficherVariableX = [x]() {
    std::cout << x << std::endl;
};

afficherVariableX();
```

Comme vous pouvez le constater, la variable externe x a été affichée.

Les tableaux à taille fixe

Vous vous souvenez des tableaux statiques du Cours C++ ?
Ceux que nous pouvons déclarer comme suit :

Code : C++

```
int tableauFixe[5];
```

Eh bien, il y a maintenant un conteneur de la STL pour ce genre de tableau.

Mais pourquoi créer un conteneur pour une chose existant déjà dans le langage ?

Le but de ce conteneur est de pouvoir utiliser les tableaux statiques de la même manière que les autres conteneurs de la STL tout en offrant des performances semblables aux tableaux statiques que vous connaissez déjà.

Premièrement, vous devez inclure cet en-tête :

Code : C++

```
#include <array>
```

Déclaration et initialisation

Pour déclarer un tableau fixe de cinq entiers, procéder comme suit :

Code : C++

```
std::array<int, 5> tableauFixe;
```

Regardez ce schéma pour mieux comprendre :

std::array<TYPE, TAILLE> NOM;

Vous pouvez en même temps initialiser le tableau avec la liste d'initialisateurs que vous connaissez déjà bien :

Code : C++

```
std::array<int, 5> tableauFixe = { 1, 2, 3, 4, 5 };
```

La taille ne peut pas être une variable, mais peut être une constante.

Donc, ce code ne compilera pas :

Code : C++

```
int taille = 5;
std::array<int, taille> tableauFixe = { 1, 2, 3, 4, 5 };
```

Mais, ce code compilera :

Code : C++

```
int const taille = 5;
std::array<int, taille> tableauFixe = { 1, 2, 3, 4, 5 };
```

C'est d'ailleurs une bonne idée de créer une constante, car si la taille du tableau venait à être modifiée, il suffirait de changer la valeur de la constante.

Il est ensuite possible de parcourir le tableau avec une boucle basée sur un intervalle :

Code : C++

```
for(int nombre : tableauFixe) {
    std::cout << nombre << std::endl;
}
```

Méthodes

Pour obtenir la taille d'un tableau, utiliser l'habituelle méthode `size()`.

Il est toujours possible d'accéder aux éléments d'un tableau avec l'opérateur [] et la méthode at() :

Code : C++

```
tableauFixe[0] = 10;
tableauFixe.at(1) = 40;
std::cout << tableauFixe[1] << std::endl;
std::cout << tableauFixe.at(0) << std::endl;
```

La méthode at() fait une vérification des limites.
Si vous lui passer en paramètre un indice hors limite, une exception `std::out_of_range` sera levée.

Attention !
Il n'y a aucune vérification des limites pour l'opérateur [].
Si vous lui passer un indice en dehors du tableau, le compilateur ne repèrera pas l'erreur et votre programme pourrait se terminer de façon inattendue.

Les méthodes `front()` et `back()` permettent respectivement d'obtenir le premier et le dernier élément du tableau.

La méthode `empty()` nous indique si le tableau est vide.

Il est possible de modifier la valeur de tous les éléments par une autre avec la méthode `fill()`.

Enfin, il est également possible d'utiliser les itérateurs avec les tableaux statiques en utilisant les méthodes `begin()` et `end()`.

Exemple

Voici un exemple de code utilisant ces méthodes :

Code : C++

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

int main() {
    std::array<int, 5> tableauFixe = { 1, 2, 3, 4, 5 }; //Création
d'un tableau à taille fixe de cinq entiers.

    //Affichage de tous ses éléments.
    for(int nombre : tableauFixe) {
        std::cout << nombre << std::endl;
    }

    std::cout << "Taille : " << tableauFixe.size() << std::endl;
//Affichage de sa taille.

    //Modification d'éléments.
    tableauFixe[0] = 10;
    tableauFixe.at(1) = 40;
    std::cout << tableauFixe[1] << std::endl; //Affichage d'un
élément précis sans vérification de limite.
    std::cout << tableauFixe.at(0) << std::endl; //Même chose sauf
qu'il y a une vérification de limite.
    std::cout << tableauFixe.front() << std::endl; //Afficher le
premier élément.
    std::cout << tableauFixe.back() << std::endl; //Afficher le
dernier élément.

    if(not tableauFixe.empty()) {
        std::cout << "Le tableau n'est pas vide." << std::endl;
    }

    tableauFixe.fill(5); //Modifier tous les éléments pour la valeur
5.

    for(std::array<int, 5>::iterator i(tableauFixe.begin()) ; i !=
tableauFixe.end() ; ++i) {
        std::cout << *i << std::endl;
    }
    return 0;
}
```

Un nouveau type de conteneur : le tuple

Commençons par définir ce qu'est un tuple :

Citation : Wikipédia

> Un tuple est une collection de dimension fixe d'objets de types différents.

Un tuple permet donc de créer un tableau statique, comme nous venons de voir, mais avec des types d'objet différents.

L'en-tête à inclure est :

Code : C++

```
#include <tuple>
```

Déclaration et initialisation

Voyons un exemple de déclaration d'un tuple :

Code : C++

```
std::tuple<int, double, std::string> nomTuple;
```

Ici, nous créons un tuple nommé `nomTuple` pouvant contenir un entier, un nombre réel et une chaîne de caractères.

La forme générale pour déclarer un tuple est :

std::tuple<type1, type2, …> nomTuple;

Si nous voulons initialiser le tuple lors de l'initialisation, nous devons envoyer des arguments au constructeur.

Imaginons un tuple qui contient des informations sur une personne.
Nous pourrions retenir son âge, sa taille et son prénom.

Il serait possible de déclarer et d'initialiser ce tuple comme suit :

Code : C++

```
std::tuple<int, double, std::string> personne(22, 185.4, "Jack");
```

Accéder aux éléments du tuple

Pour accéder aux éléments d'un tuple, nous devons utiliser la fonction `std::get()`.

Par exemple, si nous voulons modifier l'âge de notre bon ami Jack, nous devons faire ceci :

Code : C++

```
std::get<0>(personne) = 23;
```

Ensuite, nous pouvons afficher son âge :

Code : C++

```
std::cout << "Âge : " << std::get<0>(personne) << std::endl;
```

Exemple

Voici un exemple :

Code : C++

```
#include <iostream>
#include <string>
#include <tuple>

int main() {
    std::tuple<int, double, std::string> personne(22, 185.4,
"Jack"); //Création d'un tuple.
    std::get<0>(personne) = 23; //Modification d'un objet d'un
tuple.

    //Affichage des objets du tuple.
    std::cout << "Âge : " << std::get<0>(personne) << std::endl <<
"Taille : " << std::get<1>(personne) << std::endl << "Prénom : " <<
std::get<2>(personne) << std::endl;

    typedef std::tuple<int, double, std::string> tuple_personne;
```

```
        std::cout << "Taille du tuple : " <<
    std::tuple_size<tuple_personne>::value << std::endl; //Affichage de
    la taille du tuple.
        tuple_personne jack(23, 185.4, "Jack");
        if(personne == jack) { //Comparaison de tuples.
            std::cout << "Les tuples sont identiques." << std::endl;
        }
        return 0;
    }
```

Comme vous pouvez le constater, nous pouvons vérifier que les valeurs de deux tuples sont égales grâce à l'opérateur ==.

Gestion du temps

Il y a maintenant une interface pour gérer le temps en C++.
Voyons tout de suite comment elle fonctionne.

Avant tout, incluez l'en-tête :

Code : C++

```
#include <chrono>
```

Obtenir le temps actuel

Pour obtenir le temps actuel, ce qui est utile pour faire une action à chaque X secondes, nous devons appeler la fonction `std::chrono::system_clock::now()` :

Code : C++

```
std::chrono::time_point<std::chrono::system_clock> temps =
std::chrono::system_clock::now();
```

Ou plus simplement :

Code : C++

```
auto temps = std::chrono::system_clock::now();
```

Pour faire un test, nous allons appeler deux fois cette fonction, avec un appel à usleep() entre chaque appel.
Ensuite, nous afficherons le temps pris par le programme.

Code : C++

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

int main() {
    auto temps1 = std::chrono::system_clock::now();
    usleep(100000);
    auto temps2 = std::chrono::system_clock::now();

    std::cout << (temps2 - temps1).count() << " microsecondes." <<
std::endl;

    return 0;
}
```

Le résultat suivant peut s'afficher à l'écran :

Code : Console

```
100090 microsecondes.
```

Et voilà, nous avons pu calculer le temps pris par la fonction usleep() pour mettre en pause le programme.

Autres unités

On peut également décider d'afficher le temps dans une autre unité, par exemple en nanosecondes :

Code : C++

```
std::chrono::nanoseconds nbrNanoSecondes = (temps2 - temps1);
std::cout << nbrNanoSecondes.count() << " nanosecondes." <<
std::endl;
```

Le résultat :

Code : Console

```
100090000 nanosecondes.
```

J'ai essayé d'obtenir le résultat en millisecondes et ça ne fonctionne pas.
Est-ce normal ?

Oui, car il y aurait une perte de précisions étant donné que le nombre retourné est un entier.
Pour obtenir tout de même le nombre de millisecondes, il faut faire ceci :

Code : C++

```
std::chrono::duration<double, std::milli> nbrMilliSecondes = (temps2
- temps1);
std::cout << nbrMilliSecondes.count() << " millisecondes." <<
std::endl;
```

Toutes les unités que vous avez utilisées jusqu'à présent était des **typedef**s sur la classe `std::chrono::duration`.
Par exemple, pour les nanosecondes, le **typedef** est :

Code : C++

```
typedef duration<int64_t, nano>  nanoseconds;
```

Le problème avec les millisecondes dans notre exemple est que le type utilisé est un entier.

Or, le nombre de millisecondes doit être de type `double`, sinon il y aurait une perte de précision.
C'est pourquoi nous devons utiliser la classe `std::chrono::duration`.

Pour obtenir le nombre de secondes, nous pouvons faire ceci :

Code : C++

```
std::chrono::duration<double, std::ratio<1>> nbrSecondes = (temps2 -
temps1);
```

Ou plus simplement :

Code : C++

```
std::chrono::duration<double> nbrSecondes = (temps2 - temps1);
```

Les initialisateurs d'attributs

Les initialisateurs d'attributs permettent… d'initialiser les attributs (ah oui ?) d'une classe ou d'un autre type de donnée.

Pour ce faire, il suffit d'écrire la valeur des attributs, dans le même ordre qu'ils sont déclarés dans la classe, entre accolades, séparés par une virgule.

Nous allons utiliser la classe suivante (qui a deux attributs publics) pour nos premiers tests :

Code : C++

```
struct Paire {
    int nombre1;
    int nombre2;
};
```

Dans notre code, il est maintenant possible d'utiliser cette syntaxe pour initialiser les attributs :

Code : C++

```
int main() {
    Paire paire{ 5, 25 }; //Initialise nombre1 à 5 et nombre2 à 25.
    return 0;
}
```

Nous pouvons ensuite modifier les attributs de l'objet en utilisant la même syntaxe :

Code : C++

```
paire = { 3, 9 };
```

Il est également possible d'utiliser cette syntaxe pour retourner un objet :

Code : C++

```
Paire getPaire() {
    return { 10, 100 }; //Retourne une Paire dont nombre1 = 10 et
nombre2 = 100.
}
```

Dans notre fonction `main()`, nous pouvons tester cette fonction :

Code : C++

```
paire = getPaire();
```

Mais on nous a dit qu'il faut éviter à tout prix les attributs publics.
Alors, nous n'utiliserons pas beaucoup cette syntaxe, non ?

Eh bien… vous pouvez même l'utiliser avec des attributs privés, par l'intermédiaire du constructeur.

En voici un exemple :

Code : C++

```
#include <iostream>

class Paire {
    public:
        Paire(int nombre1, int nombre2) : nombre1_(nombre1),
nombre2_(nombre2) {}

    private:
        int nombre1_;
        int nombre2_;
};

Paire getPaire() {
    return { 10, 100 }; //Retourne une Paire dont nombre1_ = 10 et
nombre2_ = 100.
}

int main() {
    Paire paire{ 5, 25 }; //Initialise nombre1_ à 5 et nombre2_ à
25.
    paire = { 3, 9 };
    paire = getPaire();
    return 0;
}
```

À chaque fois, le constructeur est utilisé pour initialiser les attributs.

Vous devez également savoir que la liste d'initialisateurs a priorité sur les initialisateurs d'attributs.
Pour faire des tests, ajoutez ce constructeur :

Code : C++

```
Paire(std::initializer_list<int> liste) : nombre1_(*liste.begin()),
nombre2_(*liste.end()) {
    std::cout << "Liste d'initialisateurs." << std::endl;
}
```

Veuillez noter que cette syntaxe est utilisable pour tous les types, comme vous pouvez le constater en regardant l'exemple suivant :

Code : C++

```
bool booleen{false};
int nombre{42};
int tableau[5] = { 1, 2, 3, 4, 5 };
std::vector<int> vecteurNombres{ 1, 2, 3 };
std::string chaine{"Hello World!"};
int* pointeurSurEntier{nullptr};
```

Que de nouveautés dans la nouvelle norme, n'est-ce pas ?
Pourtant, nous n'avons pas tout vu, loin de là. 
Pour en apprendre encore plus, voici quelques liens :

- http://en.wikipedia.org/wiki/C%2B%2B0x
- http://fr.wikipedia.org/wiki/C%2B%2B1x
- http://aubedesheros.blogspot.com/search/label/c%2B%2B
- http://www2.research.att.com/~bs/C++0xFAQ.html

Et si vous voulez savoir si telle ou telle fonctionnalité est implémentée dans gcc, allez sur ce site.

Enfin, si vous voulez apprendre à utiliser les threads de la nouvelle norme (utilisable avec g++ 4.6), il y a une excellente série d'articles pour les utiliser en C++ sur ce site.

Attention, si vous utilisez g++, vous devrez ajouter un argument à la compilation (`-pthread`) pour pouvoir utiliser les threads.
Par exemple :

Code : Console

```
g++ -std=c++0x -pthread thread.cpp -o Thread
```

Merci à Babilomax pour ses commentaires et remarques très utiles.