



# Programmation Orientée Objet

Maxime MARIA

[maxime.maria@u-pem.fr](mailto:maxime.maria@u-pem.fr)



# Les templates

## Utilisation de la STL (*Standard Template Library*)

# Les templates

# Notions


- Les fonctions templates
- Les classes templates

# Le problème

- Prenons une fonction "somme"...

```
int somme(const int &a, const int &b)
{
    return a + b;
}
```

```
int i = 7, j = 9;
std::cout << somme(i, j) << std::endl;
```

 « 16 »

# Le problème

- Prenons une fonction "somme"...

```
int somme(const int &a, const int &b)
{
    return a + b;
}
```

```
int i = 7, j = 9;
std::cout << somme(i, j) << std::endl;
```

→ « 16 »

- Avec des floats ?

```
float x = 7.9, y = 1.7;
std::cout << somme(x, y) << std::endl;
```

→ « 8 »



# Le problème

- Prenons une fonction "somme"...

```
int somme(const int &a, const int &b)
{
    return a + b;
}
```

```
int i = 7, j = 9;
std::cout << somme(i, j) << std::endl;
```

→ « 16 »

- Avec des floats ? Surcharge !

```
float x = 7.9, y = 1.7;
std::cout << somme(x, y) << std::endl;
```

→ « 9,6 »

```
float somme(const float &a, const float &b)
{
    return a + b;
}
```



# Le problème

- Prenons une fonction "somme"...

```
int somme(const int &a, const int &b)
{
    return a + b;
}
```

```
int i = 7, j = 9;
std::cout << somme(i, j) << std::endl; → « 16 »
```

- Avec des floats ? Surcharge !

```
float x = 7.9, y = 1.7;
std::cout << somme(x, y) << std::endl; → « 9,6 »
```

```
float somme(const float &a, const float &b)
{
    return a + b;
}
```

- Comment éviter la surcharge pour tous les types ?





# Patron de fonction

- Fonction générique / Fonction template
  - On définit la fonction avec un type générique
  - Appelable avec tous les types (respectant l'implémentation)
  - Mot-clef `template`



# Patron de fonction

- Fonction générique / Fonction template
  - On définit la fonction avec un type générique
  - Appelable avec tous les types (respectant l'implémentation)
  - Mot-clef `template`

Déclaration avec  
`template`

Entre chevrons `<>`

`typename` indique au compilateur  
que `Type` désigne le type générique

```
template<typename Type>  
Type somme(const Type &a, const Type &b)  
{  
    return a + b;  
}
```



# Patron de fonction

- Fonction générique / Fonction template
  - On définit la fonction avec un type générique
  - Appelable avec tous les types (respectant l'implémentation)
  - Mot-clef `template`

Déclaration avec  
`template`

Entre chevrons `<>`

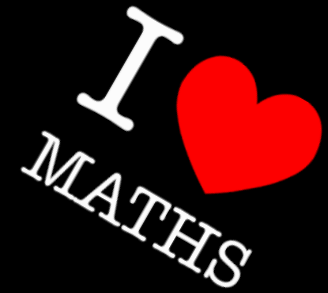
`typename` indique au compilateur  
que `Type` désigne le type générique

```
template<typename Type>  
Type somme(const Type &a, const Type &b)  
{  
    return a + b;  
}
```

- NB : `typename` peut être remplacé par `class`



# Utilisation

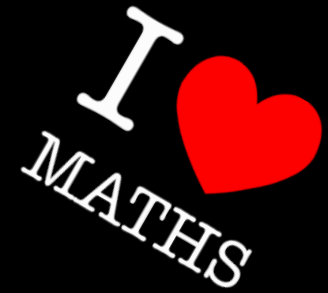


- Comme une fonction usuelle

```
template<typename Type>  
Type somme(const Type &a, const Type &b)  
{  
    return a + b;  
}
```

```
int i = 7, j = 9;  
std::cout << somme(i, j) << std::endl; → « 16 »  
float x = 7.9, y = 1.7;  
std::cout << somme(x, y) << std::endl; → « 9,6 »
```

# Utilisation



- Comme une fonction usuelle

```
template<typename Type>
Type somme(const Type &a, const Type &b)
{
    return a + b;
}
```

```
int i = 7, j = 9;
std::cout << somme(i, j) << std::endl; → « 16 »
float x = 7.9, y = 1.7;
std::cout << somme(x, y) << std::endl; → « 9,6 »
```

```
int somme(const int &a, const int &b)
{
    return a + b;
}
```

```
float somme(const float &a, const float &b)
{
    return a + b;
}
```

- Détermination statique (à la compilation)
  - Génération des fonctions selon les types nécessaires

# Utilisation



- Comme une fonction usuelle

```
template<typename Type>  
Type somme(const Type &a, const Type &b)  
{  
    return a + b;  
}
```



Déclaration ET définition  
dans le fichier d'en-tête  
(.hpp)

```
int i = 7, j = 9;  
std::cout << somme(i, j) << std::endl; → « 16 »  
float x = 7.9, y = 1.7;  
std::cout << somme(x, y) << std::endl; → « 9,6 »
```

```
int somme(const int &a, const int &b)  
{  
    return a + b;  
}
```

```
float somme(const float &a, const float &b)  
{  
    return a + b;  
}
```

- Détermination statique (à la compilation)
  - Génération des fonctions selon les types nécessaires

# Lien avec le passé



- Les vecteurs sont des templates ?

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

# Lien avec le passé



- Les vecteurs sont des templates ?

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

VRAI



- Mais c'est pas une fonction pourtant ?
  - Non c'est une classe template, on va voir ça...



# Lien avec le passé



- Les vecteurs sont des templates ?

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



- Mais c'est pas une fonction pourtant ?
  - Non c'est une classe template, on va voir ça...
- C'est du polymorphisme ?

# Lien avec le passé



- Les vecteurs sont des templates ?

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

VRAI



- Mais c'est pas une fonction pourtant ?
  - Non c'est une classe template, on va voir ça...

- C'est du polymorphisme ?

VRAI

- Polymorphisme paramétrique
- Statique : déterminé à la compilation



# Types utilisables

- Tous les types :
  - Types primitifs, pointeurs, références, classes...
- Doivent respecter l'implémentation !
  - Ici, Type doit surcharger l'opérateur +

```
template<typename Type>  
Type somme(const Type &a, const Type &b)  
{  
    return a + b;  
}
```



# Détermination du type générique

- À la compilation (statique)

```
template<typename Type>  
Type somme(const Type &a, const Type &b)  
{  
    return a + b;  
}
```

# Détermination du type générique

- À la compilation (statique)
- Détermination automatique

```
template<typename Type>  
Type somme(const Type &a, const Type &b)  
{  
    return a + b;  
}
```

OK {

```
int i = 7, j = 9;  
std::cout << somme(i, j) << std::endl; → « 16 »  
float x = 7.9, y = 1.7;  
std::cout << somme(x, y) << std::endl; → « 9,6 »
```

# Détermination du type générique

- À la compilation (statique)
- Détermination automatique

```
template<typename Type>  
Type somme(const Type &a, const Type &b)  
{  
    return a + b;  
}
```

OK { 

```
int i = 7, j = 9;  
std::cout << somme(i, j) << std::endl; → « 16 »  
float x = 7.9, y = 1.7;  
std::cout << somme(x, y) << std::endl; → « 9,6 »
```

- Pas toujours possible !

```
std::cout << somme(x, i) << std::endl; → IMPOSSIBLE
```

- Spécification explicite du type

```
std::cout << somme<float>(x, i) << std::endl; → « 14,9 »
```

↓  
i est converti en float

# Un peu plus loin...

- Plusieurs types génériques

```
template<typename T, typename U>  
T fonction(const U &param)  
{  
    // [...]  
}
```

```
int res = fonction<int, float>(79.f);
```



# Un peu plus loin...

- Plusieurs types génériques

```
template<typename T, typename U>  
T fonction(const U &param)  
{  
    // [...]  
}
```

```
int res = fonction<int, float>(79.f);
```

- Type générique par défaut

```
class UneClasse;  
  
template<typename Type = UneClasse>  
void fonction(const Type &param)  
{  
    // [...]  
}
```

Type par défaut  
(une classe !)





# Constante template

- Constante

```
template<int nbFois>
void afficherYoXFois()
{
    for (int i = 0; i < nbFois; ++i)
    {
        std::cout << "YO !" << std::endl;
    }
}
```

```
afficherYoXFois<3>();
```

↓  
« YO !  
YO !  
YO ! »



# Constante template

- Constante

```
template<int nbFois>
void afficherYoXFois()
{
    for (int i = 0; i < nbFois; ++i)
    {
        std::cout << "YO !" << std::endl;
    }
}
```

```
afficherYoXFois<3>();
```

↓  
« YO !  
YO !  
YO ! »

- Types de constantes possibles

- Type primitif
- Pointeur/référence d'objet
- Pointeur/référence de fonction



# Un exemple plus complexe

Type des données

Type du résultat, float par défaut

```
template<typename TypeData, typename TypeRes = float>
TypeRes moyenne(const std::vector<TypeData> &data)
{
    TypeRes res(0);

    for (unsigned int i = 0; i < data.size(); ++i)
    {
        res += data[i];
    }

    return res / data.size();
}
```

Initialisation OK en C++ 11



Équivalent

```
std::vector<int> vec = { 79, 17, 35, 6, 7, 16 };
```

```
std::cout << moyenne(vec) << std::endl;
std::cout << moyenne<>(vec) << std::endl;
std::cout << moyenne<int>(vec) << std::endl;
std::cout << moyenne<int, float>(vec) << std::endl;
```

« 26,6667 »

```
std::cout << moyenne<int, int>(vec) << std::endl;
```

« 26 »

# Spécialisation de fonction template

- Comportement de la somme pour des strings

```
template<typename Type>  
Type somme(const Type &a, const Type &b)  
{  
    return a + b;  
}
```

```
std::string str1("IMAC"), str2("rocks !");  
std::cout << somme(str1, str2) << std::endl;
```

→ « IMACrocks ! »



# Spécialisation de fonction template

- Comportement de la somme pour des strings

```
template<typename Type>  
Type somme(const Type &a, const Type &b)  
{  
    return a + b;  
}
```

Spécialisation pour des strings

```
template<>  
std::string somme<std::string>(const std::string &a, const std::string &b)  
{  
    return a + " " + b;  
}
```

Ajoute un espace entre les deux strings

```
std::string str1("IMAC"), str2("rocks !");  
std::cout << somme(str1, str2) << std::endl;
```

« ~~IMACrocks !~~ »

« IMAC rocks ! »



# Spécialisation de fonction template

- Comportement de la somme pour des strings

Respecter l'ordre !  
Spécialisation après

```
template<typename Type>  
Type somme(const Type &a, const Type &b)  
{  
    return a + b;  
}
```

Spécialisation pour des strings

```
template<>  
std::string somme<std::string>(const std::string &a, const std::string &b)  
{  
    return a + " " + b;  
}
```

Ajoute un espace entre les deux strings

```
std::string str1("IMAC"), str2("rocks !");  
std::cout << somme(str1, str2) << std::endl;
```

« ~~IMACrocks !~~ »

« IMAC rocks ! »



# Notions

- Les fonctions templates
- Les classes templates

# Patron de classe

- Classe générique / Classe template
- Même principe que pour une fonction

```
template<typename Type>  
class Vector3  
{  
};
```

Type générique





# Patron de classe

- Classe générique / Classe template
- Même principe que pour une fonction

```
template<typename Type>  
class Vector3  
{  
private:  
    Type m_x;  
    Type m_y;  
    Type m_z;  
};
```

Type générique

Attributs



# Patron de classe

- Classe générique / Classe template
- Même principe que pour une fonction

```
template<typename Type>  
class Vector3  
{  
public:  
    Vector3(const Type &x, const Type &y, const Type &z)  
        : m_x(x), m_y(y), m_z(z)  
    {}  
private:  
    Type m_x;  
    Type m_y;  
    Type m_z;  
};
```

Type générique

Constructeur

Attributs



# Patron de classe

- Classe générique / Classe template
- Même principe que pour une fonction

```
template<typename Type>
class Vector3
{
public:
    Vector3(const Type &x, const Type &y, const Type &z)
        : m_x(x), m_y(y), m_z(z)
    {}
    Type getX() const
    {
        return m_x;
    }
    void setX(const Type &x)
    {
        m_x = x;
    }
private:
    Type m_x;
    Type m_y;
    Type m_z;
};
```

Type générique

Constructeur

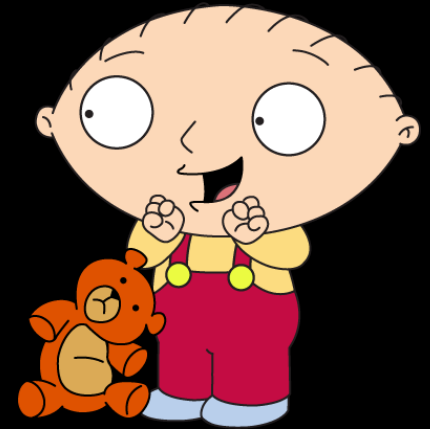
Des méthodes (getter/setter)

Attributs



# Utilisation

- Instanciation d'un objet
  - Comme d'habitude + type entre chevrons
  - Comme un `std::vector` !



```
template<typename Type>
class Vector3
{
public:
    Vector3(const Type &x, const Type &y, const Type &z)
        : m_x(x), m_y(y), m_z(z)
    {}
    Type getX() const
    {
        return m_x;
    }
    void setX(const Type &x)
    {
        m_x = x;
    }
private:
    Type m_x;
    Type m_y;
    Type m_z;
};
```

```
Vector3<float> vecFloat(79.f, 17.f, 35.f);

Vector3<int> vecInt(6, 7, 16);

std::cout << vecFloat.getX() << std::endl;

vecInt.setX(79);
```

« 79.0 »

# Alias de type

- Donne un autre nom à un type
- `typedef Type Alias;`

```
typedef Vector3<int> Vector3i;  
typedef Vector3<float> Vector3f;
```

```
Vector3f vecFloat(79.f, 17.f, 35.f);  
Vector3i vecInt(6, 7, 16);
```

# Alias de type

- Donne un autre nom à un type
- `typedef Type Alias;`

```
typedef Vector3<int> Vector3i;  
typedef Vector3<float> Vector3f;
```

```
Vector3f vecFloat(79.f, 17.f, 35.f);  
Vector3i vecInt(6, 7, 16);
```



# Alias de type

- Donne un autre nom à un type

- `typedef Type Alias;`

```
typedef Vector3<int> Vector3i;  
typedef Vector3<float> Vector3f;
```

```
Vector3f vecFloat(79.f, 17.f, 35.f);  
Vector3i vecInt(6, 7, 16);
```



- En C++11: `using Alias = Type;`

```
using Vector3i = Vector3<int>;  
using Vector3f = Vector3<float>;
```

```
Vector3f vecFloat(79.f, 17.f, 35.f);  
Vector3i vecInt(6, 7, 16);
```



# Alias de type

- Donne un autre nom à un type

- `typedef Type Alias;`

```
typedef Vector3<int> Vector3i;  
typedef Vector3<float> Vector3f;
```

```
Vector3f vecFloat(79.f, 17.f, 35.f);  
Vector3i vecInt(6, 7, 16);
```



- En C++11 : `using Alias = Type;`

```
using Vector3i = Vector3<int>;  
using Vector3f = Vector3<float>;
```

```
Vector3f vecFloat(79.f, 17.f, 35.f);  
Vector3i vecInt(6, 7, 16);
```



- `typedef` ne permet pas les alias templates

```
template<typename Type>  
using Vec = std::vector<Type>;
```

Équivalent { `std::vector<float> vec1;`  
`Vec<float> vec2;`



# UTILISATION DE LA STL

## *(Standard Template Library)*

# Standard Template Library (STL)

- Bibliothèque générique (template)
- Normalisée ISO
- Contenant :
  - Des conteneurs
  - Des itérateurs
  - Des algorithmes



# Standard Template Library (STL)

- Bibliothèque générique (template)
- Normalisée ISO
- Contenant :
  - Des conteneurs
  - Des itérateurs
  - Des algorithmes



# Les conteneurs

- Ensemble de classes (namespace `std`)
  - Pour stocker des éléments
- Syntaxe générale :
  - `std::typeConteneur<typeElement> conteneur;`
- Voyons les principaux conteneurs...



# Conteneurs de séquences

- `vector` :
  - Tableau dynamique
- `stack` :
  - Pile (LIFO)
- `queue` :
  - File (FIFO)
- `deque` :
  - File à deux bouts
- `priority_queue` :
  - File de priorité
- `list` :
  - Liste doublement chaînée



# Conteneurs associatifs

- `pair` :
  - Paire d'éléments
- `set` :
  - Ensemble d'éléments ordonnés, sans doublon
- `map` :
  - Table associative clef/donnée, sans doublon
- `multiset` :
  - Comme `set` mais doublons possibles
- `multimap` :
  - Comme `map` mais doublons possibles



# Méthodes

- Méthodes communes :
  - `empty` : Retourne true si le conteneur est vide, false sinon
  - `size` : Retourne le nombre d'éléments contenus
  - `swap` : Échange les contenus de 2 conteneurs de même type

# Méthodes

- Méthodes communes :
  - `empty` : Retourne true si le conteneur est vide, false sinon
  - `size` : Retourne le nombre d'éléments contenus
  - `swap` : Échange les contenus de 2 conteneurs de même type
- Plein d'autres ! (en fonction du type de conteneur) :
  - `insert`
  - `erase` } Nécessitent l'utilisation d'itérateurs !
- Consultez la doc !





# std::vector

- Tableau dynamique
  - Contigu en mémoire
  - Insertion/suppression en  $O(1)$  par la fin

```
#include <vector>
```

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

← Création d'un vecteur vide

vec 

# std::vector

- Tableau dynamique
  - Contigu en mémoire
  - Insertion/suppression en  $O(1)$  par la fin

```
#include <vector>
```

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

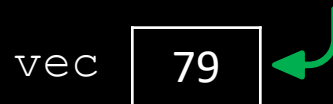


Création d'un vecteur vide

```
vec.push_back(79);
```



Insertion par l'arrière

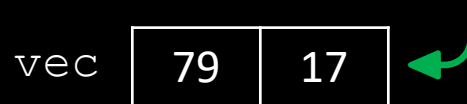


# std::vector

- Tableau dynamique
  - Contigu en mémoire
  - Insertion/suppression en  $O(1)$  par la fin

```
#include <vector>
```

```
std::vector<int> vec; ← Création d'un vecteur vide  
vec.push_back(79); ← Insertions par l'arrière  
vec.push_back(17);
```

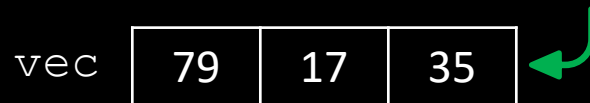


# std::vector

- Tableau dynamique
  - Contigu en mémoire
  - Insertion/suppression en  $O(1)$  par la fin

```
#include <vector>
```

```
std::vector<int> vec; ← Création d'un vecteur vide  
vec.push_back(79); ← Insertions par l'arrière  
vec.push_back(17);  
vec.push_back(35);
```



# std::vector

- Tableau dynamique
  - Contigu en mémoire
  - Insertion/suppression en  $O(1)$  par la fin

```
#include <vector>
```

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

← Création d'un vecteur vide

```
vec.push_back(79);
```

← Insertions par l'arrière

```
vec.push_back(17);
```

```
vec.push_back(35);
```

```
vec.pop_back();
```

← Suppression par l'arrière



# std::vector

- Tableau dynamique
  - Contigu en mémoire
  - Insertion/suppression en  $O(1)$  par la fin

```
#include <vector>
```

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

 ← Création d'un vecteur vide

```
vec.push_back(79);
```

 ← Insertions par l'arrière

```
vec.push_back(17);
```

```
vec.push_back(35);
```

```
vec.pop_back();
```

 ← Suppression par l'arrière

vec

79	17
----	----

- Éléments accessibles en  $O(1)$  via `[]` : `vec[0]`, `vec[1]` ...

# std::deque



- File à deux bouts : *Double ended queue*
  - Contigu en mémoire
  - Insertion/suppression en  $O(1)$  par le début ET la fin

```
#include <deque>
```

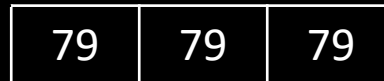
# std::deque



- File à deux bouts : *Double ended queue*
  - Contigu en mémoire
  - Insertion/suppression en  $O(1)$  par le début ET la fin

```
#include <deque>
```

```
std::deque<int> file(3,79);
```

 ← Création d'une file de 3 « 79 »

file



# std::deque



- File à deux bouts : *Double ended queue*
  - Contigu en mémoire
  - Insertion/suppression en  $O(1)$  par le début ET la fin

```
#include <deque>
```

```
std::deque<int> file(3,79);
```

← Création d'une file de 3 « 79 »

```
file.push_front(1);
```

← Insertion par l'avant



# std::deque



- File à deux bouts : *Double ended queue*
  - Contigu en mémoire
  - Insertion/suppression en  $O(1)$  par le début ET la fin

```
#include <deque>
```

```
std::deque<int> file(3,79);
```

← Création d'une file de 3 « 79 »

```
file.push_front(1);
```

← Insertion par l'avant

```
file.push_back(9);
```

← Insertion par l'arrière



# std::deque



#include <deque>

- File à deux bouts : *Double ended queue*
  - Contigu en mémoire
  - Insertion/suppression en  $O(1)$  par le début ET la fin

```
std::deque<int> file(3,79);
```

← Création d'une file de 3 « 79 »

```
file.push_front(1);
```

← Insertion par l'avant

```
file.push_back(9);
```

← Insertion par l'arrière

```
file.pop_front();
```

← Suppression par l'avant



# std::deque



#include <deque>

- File à deux bouts : *Double ended queue*
  - Contigu en mémoire
  - Insertion/suppression en  $O(1)$  par le début ET la fin

```
std::deque<int> file(3,79);
```

← Création d'une file de 3 « 79 »

```
file.push_front(1);
```

← Insertion par l'avant

```
file.push_back(9);
```

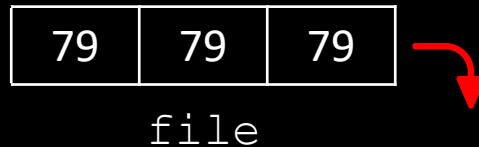
← Insertion par l'arrière

```
file.pop_front();
```

← Suppression par l'avant

```
file.pop_back();
```

← Suppression par l'arrière



# std::deque

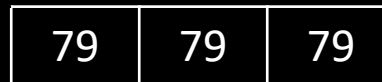


- File à deux bouts : *Double ended queue*

#include <deque>

- Contigu en mémoire
- Insertion/suppression en  $O(1)$  par le début ET la fin

```
std::deque<int> file(3,79); ← Création d'une file de 3 « 79 »  
file.push_front(1); ← Insertion par l'avant  
file.push_back(9); ← Insertion par l'arrière  
file.pop_front(); ← Suppression par l'avant  
file.pop_back(); ← Suppression par l'arrière
```



file

- Éléments accessibles en  $O(1)$  via `[]` : `file[0]`, `file[1]`...

# std::stack

- Pile (LIFO : *Last In First Out*)
  - Accès et modification possible QUE par le haut ( $O(1)$ )

```
#include <stack>
```



# std::stack

- Pile (LIFO : *Last In First Out*)
  - Accès et modification possible QUE par le haut ( $O(1)$ )

```
#include <stack>
```

```
std::stack<int> pile;
```

← Création d'une pile vide

pile



# std::stack

- Pile (LIFO : *Last In First Out*)
  - Accès et modification possible QUE par le haut ( $O(1)$ )

```
#include <stack>
```

```
std::stack<int> pile;
```

← Création d'une pile vide

```
pile.push(17);
```

← Insertion

17
----

pile





# std::stack

- Pile (LIFO : *Last In First Out*)
  - Accès et modification possible QUE par le haut ( $O(1)$ )

`#include <stack>`

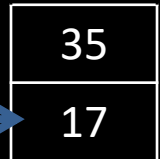
```
std::stack<int> pile;
```

← Création d'une pile vide

```
pile.push(17);  
pile.push(35);
```

← Insertions

Inaccessible



pile



# std::stack

- Pile (LIFO : *Last In First Out*)
  - Accès et modification possible QUE par le haut ( $O(1)$ )

`#include <stack>`

```
std::stack<int> pile;
```

← Création d'une pile vide

```
pile.push(17);  
pile.push(35);  
pile.push(79);
```

← Insertions

Inaccessibles



pile



# std::stack

- Pile (LIFO : *Last In First Out*)
  - Accès et modification possible QUE par le haut ( $O(1)$ )

#include <stack>

```
std::stack<int> pile;
```

← Création d'une pile vide

```
pile.push(17);
```

```
pile.push(35);
```

```
pile.push(79);
```

← Insertions

Inaccessibles



pile

```
int a = pile.top();
```

← Lecture de la valeur sortie : a = 79



# std::stack

- Pile (LIFO : *Last In First Out*)
  - Accès et modification possible QUE par le haut ( $O(1)$ )

`#include <stack>`

```
std::stack<int> pile;
```

← Création d'une pile vide

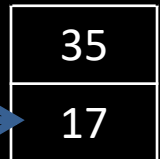
```
pile.push(17);
```

```
pile.push(35);
```

```
pile.push(79);
```

← Insertions

Inaccessible →



pile

```
int a = pile.top();
```

← Lecture de la valeur sortie :  $a = 79$

```
pile.pop();
```

← Suppression du haut de la pile



# std::queue

- File (FIFO : *First In First Out*)

```
#include <queue>
```

# std::queue

- File (FIFO : *First In First Out*)

```
#include <queue>
```

C'est bon...  
On a compris...



# Nota Bene

- Insertions en  $O(1)$  uniquement si la taille *size()* est inférieure à la capacité *capacity()*
- Sinon, réallocation  $\rightarrow$  copie des éléments contenus  $O(n)$
- Capacité d'un conteneur : *capacity()*
  - Taille mémoire allouée à la construction ( $>$  taille effective)
  - But : permettre l'insertion en  $O(1)$  (on parle de coût amorti)
- Récupérer cette mémoire : *shrink\_to\_fit()* (C++11)

# std::map

- Table associative
  - Paire de Clef/Élément
  - Clef (indice) de n'importe quel type : doit implémenter < !

```
#include <map>
```




# std::map

- Table associative
  - Paire de Clef/Élément
  - Clef (indice) de n'importe quel type : doit implémenter < !

```
#include <map>
```

Création d'une map : "tableau" d'entiers avec des indices de type string



```
std::map<std::string, int> map;
```

# std::map



#include <map>

- Table associative
  - Paire de Clef/Élément
  - Clef (indice) de n'importe quel type : doit implémenter < !

Création d'une map : "tableau" d'entiers avec des indices de type string



```
std::map<std::string, int> map;
```

Insertions :  
Si la clef n'existe pas, elle est  
automatiquement créée

```
{ map["Deux-Sèvres"] = 79;  
  map["Paris"] = 75;
```

# std::map



#include <map>

- Table associative
  - Paire de Clef/Élément
  - Clef (indice) de n'importe quel type : doit implémenter < !

Création d'une map : "tableau" d'entiers avec des indices de type string

Insertions :  
Si la clef n'existe pas, elle est  
automatiquement créée

```
std::map<std::string, int> map;  
  
map["Deux-Sèvres"] = 79;  
map["Paris"] = 75;  
  
std::cout << map["Deux-Sèvres"] << std::endl;
```

« 79 »

# std::map



#include <map>

- Table associative
  - Paire de Clef/Élément
  - Clef (indice) de n'importe quel type : doit implémenter < !

Création d'une map : "tableau" d'entiers avec des indices de type string

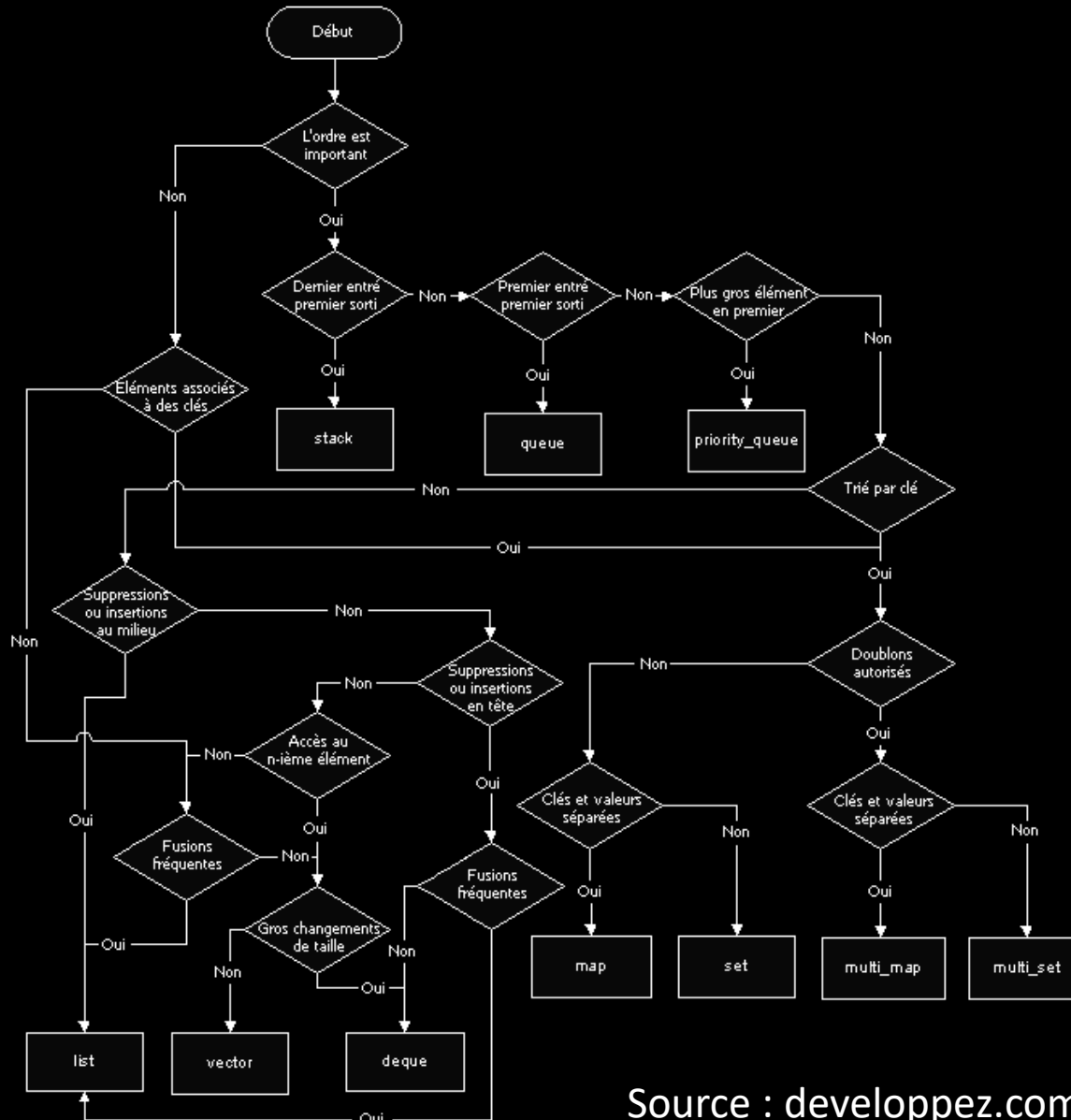
Insertions :  
Si la clef n'existe pas, elle est  
automatiquement créée

```
std::map<std::string, int> map;  
  
map["Deux-Sèvres"] = 79;  
map["Paris"] = 75;  
  
std::cout << map["Deux-Sèvres"] << std::endl;
```

« 79 »

- Parcours par itérateur !

# Quel conteneur choisir ?



Source : [developpez.com](http://developpez.com)

# Standard Template Library (STL)

- Bibliothèque générique (template)
- Normalisée ISO
- Contenant :
  - Des conteneurs
  - Des itérateurs
  - Des algorithmes



# Itérateurs

- Abstraction de pointeurs
  - Permet de se déplacer dans un conteneur
  - Utilisation similaire pour tous les conteneurs

# Itérateurs

- Abstraction de pointeurs
  - Permet de se déplacer dans un conteneur
  - Utilisation similaire pour tous les conteneurs

- Syntaxe générale : mot-clef `iterator`

- `conteneur::iterator it;`



*e.g.*



Via les deux points

```
std::vector<int>,  
std::deque<float>, ...
```



# Itérateurs

- Abstraction de pointeurs
  - Permet de se déplacer dans un conteneur
  - Utilisation similaire pour tous les conteneurs

- Syntaxe générale : mot-clef `iterator`

- `conteneur::iterator it;`

*e.g.*

`std::vector<int>,  
std::deque<float>, ...`

Via les deux points

Un exemple  
peut-être ?  
Soit L, une `std::list...`

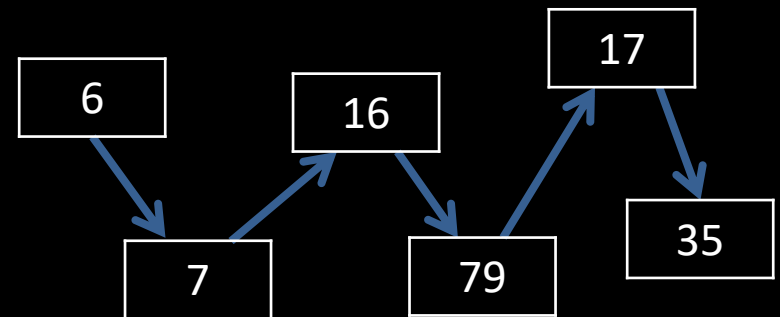


# std::list

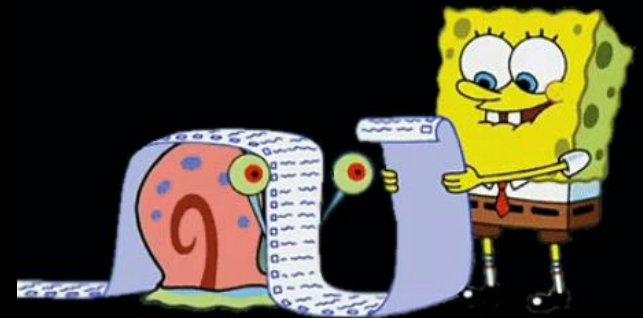
- Liste doublement chaînée
  - Modification en  $O(1)$  par le début ET la fin

```
#include <list>
```

- Différence avec deque ?
  - Pas contigu en mémoire
  - Éléments non accessibles via [ ]



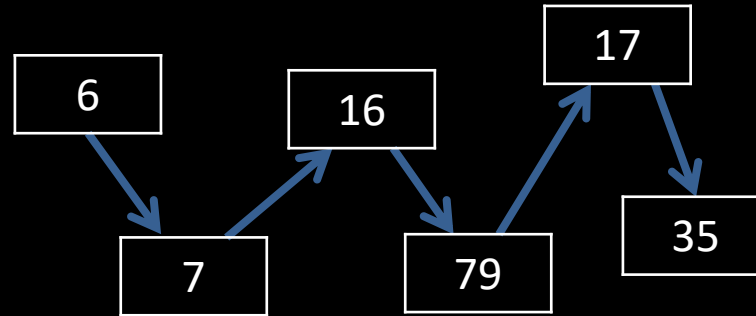
- Parcours via un itérateur !



# Utilisation d'un itérateur

- Déclaration d'une liste, insertion de 6 entiers

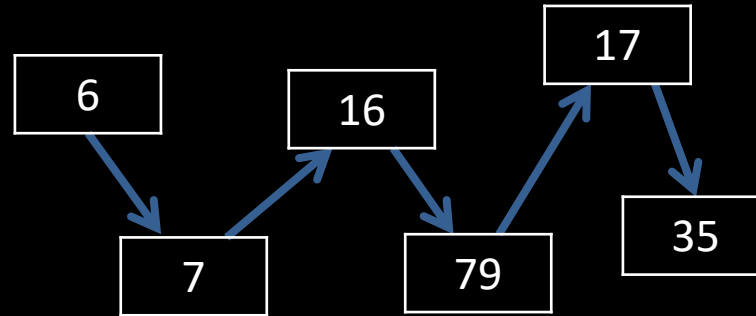
```
std::list<int> list;  
  
list.push_back(6);  
list.push_back(7);  
list.push_back(16);  
list.push_back(79);  
list.push_back(17);  
list.push_back(35);
```



# Utilisation d'un itérateur

- Déclaration d'une liste, insertion de 6 entiers

```
std::list<int> list;  
  
list.push_back(6);  
list.push_back(7);  
list.push_back(16);  
list.push_back(79);  
list.push_back(17);  
list.push_back(35);
```

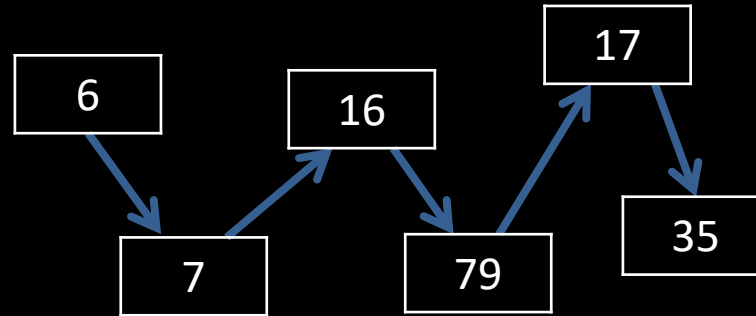


- Déclaration d'un itérateur `std::list<int>::iterator it;`

# Utilisation d'un itérateur

- Déclaration d'une liste, insertion de 6 entiers

```
std::list<int> list;  
  
list.push_back(6);  
list.push_back(7);  
list.push_back(16);  
list.push_back(79);  
list.push_back(17);  
list.push_back(35);
```



- Déclaration d'un itérateur
- Parcours de la liste

```
std::list<int>::iterator it;
```

Itérateur sur le premier élément

Itérateur sur le dernier élément

```
for (it = list.begin(); it != list.end(); ++it)  
{  
    std::cout << *it << std::endl;  
}
```

Incrémentation

Accès à la donnée comme avec un pointeur, via \*

# Les différents itérateurs

- On déclare toujours via `::iterator`
  - En réalité, il existe 5 sortes d'itérateurs :
    - Input
    - Output
    - Forward
    - Bidirectionnal
    - Random Access
- } Utilisés pour les conteneurs



# Itérateurs pour conteneurs

- Bidirectionnal iterator :
  - Déplacement uniquement via `++` `--` (de un en un)
  - Utilisé pour `list`, `map`, `set...` (mémoire non contigüe)

# Itérateurs pour conteneurs

- Bidirectionnal iterator :
  - Déplacement uniquement via ++ -- (de un en un)
  - Utilisé pour `list`, `map`, `set`... (mémoire non contigüe)
- Random access iterator
  - Accès possible "par le milieu"
  - Utilisé pour `vector`, `deque`... (mémoire contigüe)



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

6	7	16	79	35
---	---	----	----	----

```
std::vector<int>::iterator it = v.begin() + 3;
```

`*it`





# Types d'itérateurs

- 4 classiques :
  - `iterator`
    - Parcours du début à la fin
  - `const_iterator`
    - Parcours du début à la fin
    - Éléments accessibles uniquement en lecture
  - `reverse_iterator / const_reverse_iterator`
    - Sens opposé
- Utilisez "`const_`" dès que possible !

# Standard Template Library (STL)

- Bibliothèque générique (template)
- Normalisée ISO
- Contenant :
  - Des conteneurs
  - Des itérateurs
  - Des algorithmes



# Les algorithmes

```
#include <algorithm>
```

- La STL propose une multitude d'algorithmes
  - Consultez la doc !
- D'un point de vue général, il faut maîtriser :
  - Les itérateurs



# Les algorithmes

```
#include <algorithm>
```

- La STL propose une multitude d'algorithmes
  - Consultez la doc !
- D'un point de vue général, il faut maîtriser :
  - Les itérateurs
  - Les foncteurs



# Les algorithmes

```
#include <algorithm>
```

- La STL propose une multitude d'algorithmes
  - Consultez la doc !
- D'un point de vue général, il faut maîtriser :
  - Les itérateurs
  - Les foncteurs
- Prenons un exemple : le tri d'un vecteur



# std::sort



- Tri des éléments dans l'ordre croissant
  - Via l'opérateur <

Initialisation OK en C++ 11

```
std::vector<int> vecInt = { 79, 7, 16, 6 };
```



79	7	16	6
----	---	----	---

```
std::sort(vecInt.begin(), vecInt.end());
```



6	7	16	79
---	---	----	----

Interval de tri

# std::sort



- Tri des éléments dans l'ordre croissant
  - Via l'opérateur <

Initialisation OK en C++ 11

```
std::vector<int> vecInt = { 79, 7, 16, 6 };
```

79	7	16	6
----	---	----	---

```
std::sort(vecInt.begin(), vecInt.end());
```

6	7	16	79
---	---	----	----

Interval de tri

- Avec des strings ? < définit l'ordre alphabétique

```
std::vector<std::string> vecStr;  
vecStr.push_back("Lettre");  
vecStr.push_back("Alphabet");  
vecStr.push_back("Mot");
```

Lettre	Alphabet	Mot
--------	----------	-----

```
std::sort(vecStr.begin(), vecStr.end());
```

Alphabet	Lettre	Mot
----------	--------	-----

# std::sort



- Tri des éléments dans l'ordre croissant
  - Via l'opérateur <

Initialisation OK en C++ 11

```
std::vector<int> vecInt = { 79, 7, 16, 6 };
```

79	7	16	6
----	---	----	---

```
std::sort(vecInt.begin(), vecInt.end());
```

6	7	16	79
---	---	----	----

Interval de tri

- Avec des strings ? < définit l'ordre alphabétique

```
std::vector<std::string> vecStr;  
vecStr.push_back("Lettre");  
vecStr.push_back("Alphabet");  
vecStr.push_back("Mot");
```

Lettre	Alphabet	Mot
--------	----------	-----

```
std::sort(vecStr.begin(), vecStr.end());
```

Alphabet	Lettre	Mot
----------	--------	-----

- Comment changer le comportement du tri ?



# Foncteur

- Abstraction de fonction
- Objet possédant une surcharge de l'opérateur `()`
  - Agit comme une fonction lorsqu'il est passé en argument

# Foncteur

- Abstraction de fonction
- Objet possédant une surcharge de l'opérateur ( )
  - Agit comme une fonction lorsqu'il est passé en argument
- Exemple : foncteur additionnant deux entiers

```
class Addition
{
public:
    int operator()(const int a, const int b)
    {
        return a + b;
    }
};
```

Surcharge de l'opérateur ( )



# Foncteur

- Abstraction de fonction
- Objet possédant une surcharge de l'opérateur ( )
  - Agit comme une fonction lorsqu'il est passé en argument
- Exemple : foncteur additionnant deux entiers

```
class Addition
{
public:
    int operator()(const int a, const int b)
    {
        return a + b;
    }
};
```

Surcharge de l'opérateur ( )

Déclaration d'un foncteur

```
Addition foncteur;
```

Utilisation comme une fonction

```
std::cout << foncteur(6, 7) << std::endl;
```

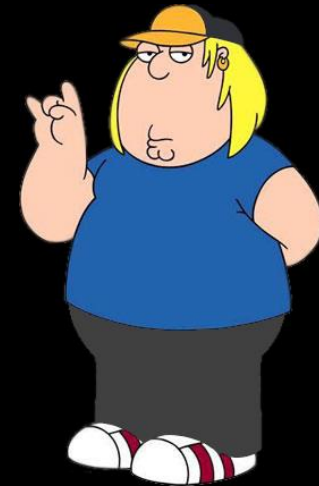
« 13 »

# Foncteurs prédéfinis

- La STL propose des foncteurs prédéfinis

- Arithmétiques
- Comparaisons
- Logiques...
- Consultez la doc !

```
#include <functional>
```



- Exemple : `std::plus`

Foncteur prédéfini : addition de deux entiers

```
std::plus<int> foncteur;
```

```
std::cout << foncteur(6, 7) << std::endl;
```

« 13 »

# Algorithmes et foncteurs

- Comment changer le comportement du tri ?

```
std::vector<std::string> vecStr;  
vecStr.push_back("Lettre");  
vecStr.push_back("Alphabet");  
vecStr.push_back("Mot");
```



Lettre	Alphabet	Mot
--------	----------	-----

```
std::sort(vecStr.begin(), vecStr.end());
```



Alphabet	Lettre	Mot
----------	--------	-----

# Algorithmes et foncteurs

- Comment changer le comportement du tri ?

```
std::vector<std::string> vecStr;  
vecStr.push_back("Lettre");  
vecStr.push_back("Alphabet");  
vecStr.push_back("Mot");
```

Lettre	Alphabet	Mot
--------	----------	-----

```
std::sort(vecStr.begin(), vecStr.end());
```

Alphabet	Lettre	Mot
----------	--------	-----

```
class CompareLongueurStr  
{  
public:  
    bool operator()(const std::string &a, const std::string &b)  
    {  
        return a.length() < b.length();  
    }  
};
```

Foncteur comparant la  
longueur de deux strings

```
CompareLongueurStr foncteur;
```

```
std::sort(vecStr.begin(), vecStr.end(), foncteur);
```

Foncteur en argument

Mot	Lettre	Alphabet
-----	--------	----------

# Fonction Lambda (C++ 11)

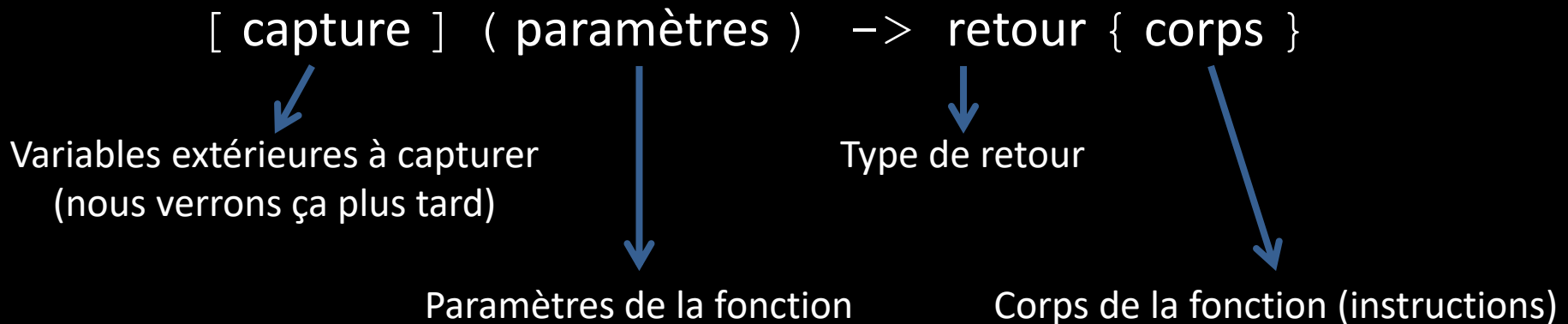


- = Fonction anonyme
- Ici, utilisation dans les algorithmes de la STL

# Fonction Lambda (C++ 11)



- = Fonction anonyme
- Ici, utilisation dans les algorithmes de la STL
- Syntaxe générale :

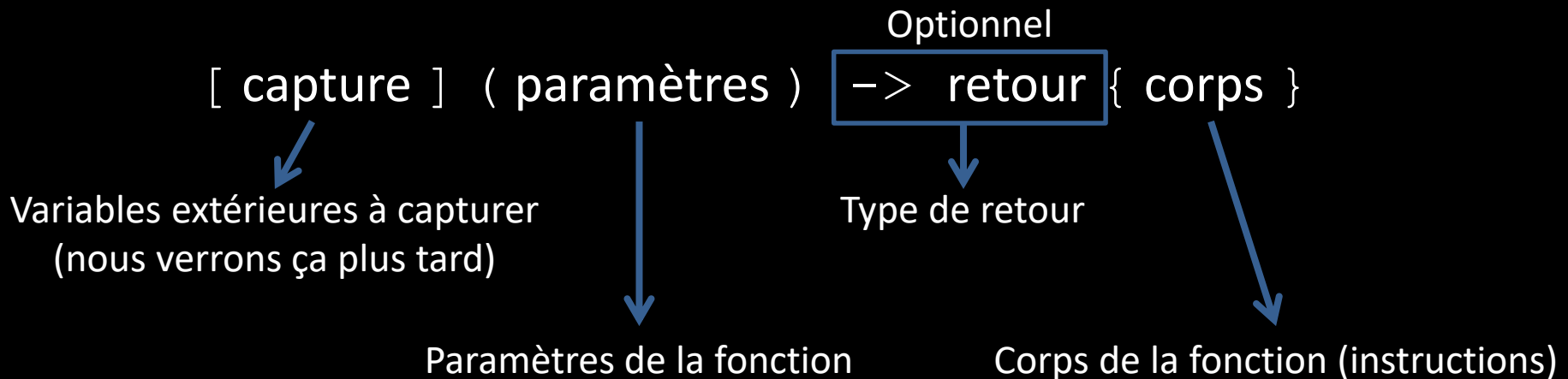




# Fonction Lambda (C++ 11)



- = Fonction anonyme
- Ici, utilisation dans les algorithmes de la STL
- Syntaxe générale :



# Utilisation comme foncteur

- Toujours le même exemple...

```
std::vector<std::string> vecStr;  
vecStr.push_back("Lettre");  
vecStr.push_back("Alphabet");  
vecStr.push_back("Mot");
```

Lettre	Alphabet	Mot
--------	----------	-----

```
std::sort(vecStr.begin(), vecStr.end());
```

Alphabet	Lettre	Mot
----------	--------	-----

```
std::sort( vecStr.begin(), vecStr.end(),
```

Rien à  
capturer

```
[ ](const std::string &a, const std::string &b)  
{  
    return a.length() < b.length();  
}
```

Fonction  
Lambda

Mot	Lettre	Alphabet
-----	--------	----------

# Déclaration d'une fonction lambda

- Dans une variable `std::function<typeFoncteur>`

Type de retour puis paramètre(s)

`#include <functional>`

```
std::function<bool(const std::string &, const std::string &)> foncteur;  
  
foncteur = [](const std::string &a, const std::string &b)  
{  
    return a.length() < b.length();  
};  
  
std::sort(vecStr.begin(), vecStr.end(), foncteur);
```

- Possibilité de l'utiliser plusieurs fois !



# [capture]

- Pour utiliser des variables déclarées en dehors de la fonction lambda



# [capture]

- Pour utiliser des variables déclarées en dehors de la fonction lambda
- Deux façons de capturer
  - Par valeur : variable copiée

```
foncteur = [variable] // [...]
```

- Par référence : possibilité de modifier la variable

```
foncteur = [&variable] // [...]
```



# [capture]

- Pour utiliser des variables déclarées en dehors de la fonction lambda
- Deux façons de capturer
  - Par valeur : variable copiée

```
foncteur = [variable] // [...]
```

- Par référence : possibilité de modifier la variable

```
foncteur = [&variable] // [...]
```

- Liste de captures

```
foncteur = [&variable1, variable2, &variable3] // [...]
```



# Exemple

- Calculer la somme des éléments d'un vecteur
  - Utilisation de l'algorithme de parcours `for_each`

Capture par  
référence  
(modifiable)

```
int somme(0);

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

std::function<void (const int)> foncteur;

foncteur = [&somme](const int i) { somme += i; };

std::for_each(vec.begin(), vec.end(), foncteur);

std::cout << somme << std::endl;
```

Application  
du foncteur

Parcours du  
vecteur

« 15 »



**FIN DU CM !**

**Renseignez vous sur Boost !**

**<https://theboostcpplibraries.com/>**

