

ISEN

ALL IS DIGITAL!

LILLE



yncréa



Le Langage C++

Quelques classes STL et types C++11/14

- Le type “array”
- Le type “unique_ptr”
- Le type “tuple”
- Les « expressions lambdas »

Quelques difficultés liées aux tableaux de taille fixe :

- Il n'est généralement pas possible de connaître sa taille;
- A partir du nom du tableau, il est possible d'accéder à n'importe quel emplacement mémoire (bug mémoire potentiel).

En C/C++ :

```
int tab[] = {1,2,3};
```

Quel est la taille du tableau tab ?

Dans la fonction/procédure où il est déclaré : `sizeof(tab)/sizeof(tab[0])`.

Mais quand il est passé en paramètre ?

```
void my_function(int tab[]){  
    for (int i = 0; i < size; ++i){ //size ??  
        tab[i] = i + 1;  
    }  
}
```

Un “array” est un « container » qui stocke un ensemble de données de même type, contigu en mémoire, et de dimension fini (nécessite le fichier d’entête <array>) :

```
std::array<type, taille> tab;
```

Par exemple :

```
constexpr int M = 3, N = 6;
```

```
array<int, M> tab1; // 3 éléments de type int initialisés à 0
```

```
array<int, N> tab2{1,2,3}; // 6 éléments de type int dont les trois premiers sont explicitement initialisés à 1, 2, 3 et les autres à 0
```

```
array<M,array<int,N>> tab3; //matrice (MxN)
```

C’est un « container », la classe array dispose de quelques méthodes déjà vues avec la classe vector:

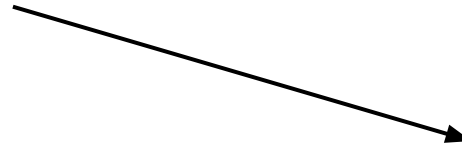
at, operator[], front(), back(), begin...end, rbegin..rend, size(), fill(...).

Par exemple :

```
size_t n = tab2.size();  
for (auto &iter : tab2) {  
    iter = -1;  
}
```

Exemple d'un passage d'une variable de type « array » dans une fonction/procédure

```
template<size_t size>
void dummy(array<int, size> &tab){
    sort(tab.begin(), tab.end());
    cout<<tab.size()<<endl;
    for (auto &itab : tab) cout << itab << " ";
    cout<<endl;
}
```



```
10
1 2 3 4 5 6 7 8 9 10
```

```
int main() {
    array<int, 10> tab;
    int count = 0;
    for (auto itab = tab.begin(); itab != tab.end(); ++itab, ++count) *itab = tab.size() - count;
    dummy(tab);
}
```

Exemple :

```
int main() {
    array<int, 3> tab2{3,2,1};
    my_print(tab2);
    std::fill(tab2.begin(), tab2.end(), -1);
    my_print(tab2);
    std::fill_n(tab2.begin(), tab2.size(), -2);
    my_print(tab2);
    for_each(tab2.begin(), tab2.end(), init);
    my_print(tab2);

    array<int, 3> tab3{3,2,1};
    sort(tab3.begin(), tab3.end(), order);
    my_print(tab2);
    return 0;
}
```

```
void init(int &val){
    val = -3;
}
```

```
struct myclass {
    bool operator() (const int &i, const int &j) {
        return i < j;
    } order;
}
```

```
template<size_t size>
void my_print(array<int, size> &tab){
    for (auto &itab : tab) {
        cout << itab << " ";
    }
    cout << endl;
}
```

```
3 2 1
-1 -1 -1
-2 -2 -2
-3 -3 -3
1 2 3
```

Quelques exemples de problèmes liés à l'utilisation des pointeurs et des fonctions new/delete :

La mémoire est allouée mais jamais libérée :

```
int n = 1000000;
for (int i; i < n; ++i) {
    buggy_procedure1();
}
```

```
void buggy_procedure1(){
    ....
    tab = new int[1000];
    ...
}
```

La mémoire est désallouée plusieurs fois :

```
auto tab = new int[1000];
buggy_procedure2(tab);
delete [] tab;
```

```
void buggy_procedure2(int *tab){
    ...
    delete [] tab;
    ...
}
```

L'adresse donnée au destructeur n'est pas la bonne :

```
auto tab = new int[1000];
++tab;
delete [] tab;
```

...

Quelques exemples de problèmes liés à l'utilisation des pointeurs et des fonctions new/delete :
La mémoire est allouée mais jamais libérée :

```
int n = 1000000;  
...  
for (int i; i < n; ++i) {  
    auto my_class(Buggy_class(n));  
}  
...
```



```
class Buggy_class{  
...  
private:  
    int *my_wonderfull_array_;  
  
public:  
    Buggy_class(const size_t &size) :  
        my_wonderfull_array_(new int[size]) {  
    }  
    ...  
    ~Buggy_class(){  
        // La mémoire my_wonderfull_array_ n'est  
        // jamais libérée  
    }  
};
```


unique_ptr est un « pointeur intelligent » qui prend en charge un pointeur de telle sorte qu'il ne puisse être référencé qu'une seule fois et est automatiquement détruit quand il n'est plus référencé.

Exemple de déclarations :

```
#include<memory>
```

```
unique_ptr<type> my_pointer1; //initialisé à nullptr
```

```
unique_ptr<int> my_pointer2(new int[10]); //initialisé à vers un pointeur d'entier avec 10 éléments
```

```
my_pointer1 = make_unique<int>(10); //initialisé à vers un pointeur d'entier avec 10 éléments C++14
```

```
auto my_pointer2 = make_unique<int>(10); //initialisé à vers un pointeur d'entier avec 10 éléments C++14
```

Quelques exemples de problèmes potentiels liés à l'utilisation des pointeurs et des fonctions new/delete :

La mémoire est allouée mais jamais libérée :

```
int n = 1000000;  
for (int i; i < n; ++i) {  
    buggy_procedure1();  
}
```

```
void buggy_procedure1(){  
....  
    auto tab = new int[1000];  
    auto tab = make_unique<int>(1000);  
}
```

La mémoire est désallouée plusieurs fois :

```
auto tab = make_unique<int>(1000);  
buggy_procedure2(tab);  
delete [] tab;
```

```
...  
}  
  
void buggy_procedure2(unique_ptr<int> &tab){  
    // delete [] tab; //impossible  
    tab = nullptr; // éventuellement mais la mémoire  
    // est libérée  
}
```

L'adresse donnée au destructeur n'est pas la bonne :

```
auto tab = make_unique<int>(1000);  
++tab;  
delete [] tab; // le pointeur associé doit normalement être géré par l'unique_ptr  
// Elle est automatiquement libérée quand elle n'est plus référencée
```

Dans un prochain cours nous reviendrons sur ce sujet qui est plus complexe que cette présentation.

Quelques exemples de problèmes liés à l'utilisation des pointeurs et des fonctions new/delete :
~~La mémoire est allouée mais jamais libérée :~~

```
int n = 1000000;  
...  
for (int i; i < n; ++i) {  
    auto my_class(Buggy_class(n));  
}  
...
```



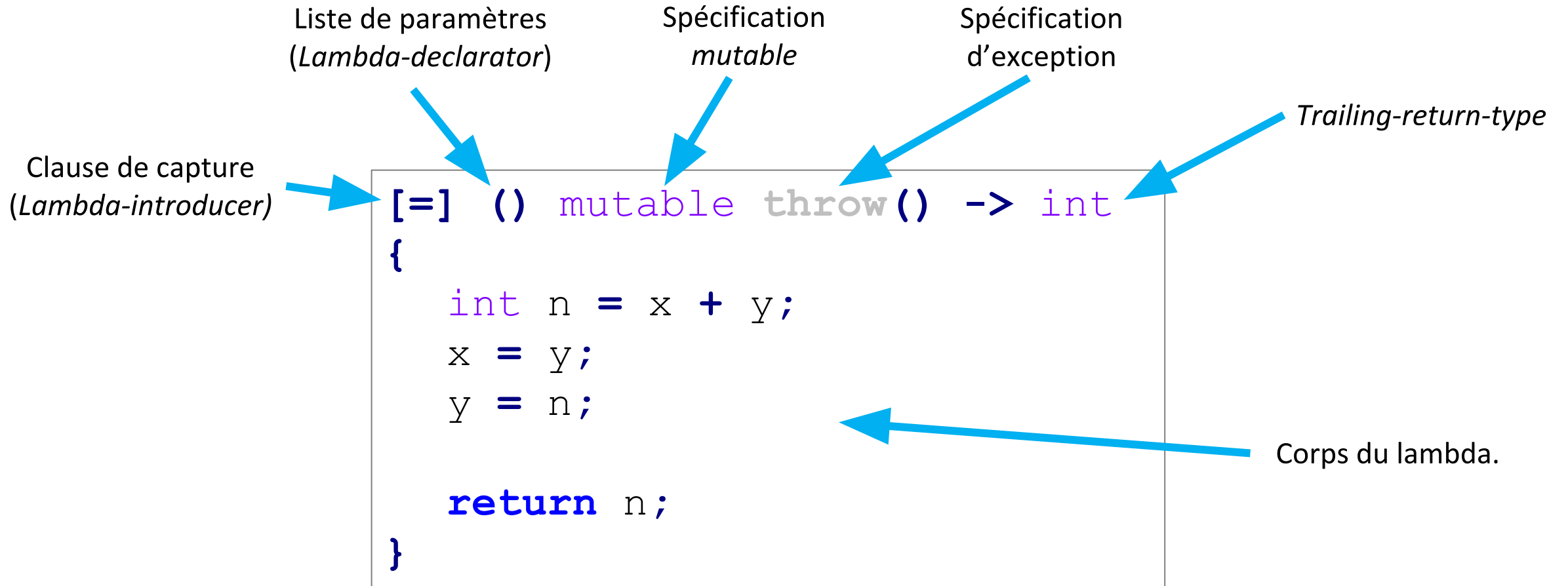
```
class Buggy_class{  
...  
private:  
    int *my_wonderfull_array_;  
    unique_ptr<int> my_wonderfull_array_;  
  
public:  
    Buggy_class(const size_t &size) :  
        my_wonderfull_array_(make_unique<int>(size)) {  
    }  
    ...  
    ~Buggy_class(){}  
};
```

- Depuis C ++ 11
- Fonctions éventuellement anonymes.
- Un moyen pratique de définir une fonction à l'emplacement où elle est appelée (peut permettre au compilateur de légères optimizations additionnelles) ou être passée comme argument.
- Utilisées pour encapsuler quelques lignes de code.

```
#include<iostream>
#include <algorithm>
#include<array>
using namespace std;
int main()
{
    array<int, 10> vec = {1,2,3,4,5,6,7,8,9, 100};
    sort(vec.begin(), vec.end(), [](const int &a, const int &b) {return a >
b; } );
    for (auto &item : vec) cout << item << " ";
    return 0;
}
```

100 9 8 7 6 5 4 3 2 1

Les éléments d'une expression lambda



Spécifie les symboles visibles dans le champ où la fonction est déclarée

Une liste de symboles peut être adoptée comme suit:

- `[a,&b]` **a** est capturé par valeur et **b** est capturé par référence.
- `[this]` capture le pointeur **this**
- `[&]` capture tous les symboles par référence.
- `[=]` capture tous les symboles par valeur.
- `[]` ne capture rien

```
void abssort(float* x, unsigned n) {  
    std::sort(x, x + n, [](const float &a, const float &b) {return (abs(a) < abs(b));} );  
}
```

```
auto func1 = [](int i) {cout << i << ":";};
```

```
func1(42);
```

42

```
int r = 0, s = 0;
```

```
auto l = [r, &s] () mutable {
```

```
    // Capture r par valeur et s par référence
```

```
    r++; // Capturé par valeur, ne peut pas être modifié sans mutable
```

```
    s++; // Capturé par référence c'est OK
```

```
};
```

```
l();
```

```
auto mesg = [](string message) { std::cout<<"Hello " << message<< std::endl; };
```

```
mesg("everyone");
```

Hello everyone


```
#include<iostream>
#include <algorithm>
#include<array>
#include<string>
using namespace std;

int main() {
    array<int, 10> vec = { 1,2,3,4,5,6,7,8,9, 100};
    auto print = [](const int &a){cout << a + 1 <<" ";};
    for_each(vec.begin(), vec.end(),print);
}
```

```
1 2 3 4 5 6 7 8 9 100
```

Data sorting

```
#include <stdio.h>
#include <stdlib.h>
#include <array>
#include <algorithm>
#include <iostream>

using namespace std;
#define _SIZE_ 6
constexpr int M = _SIZE_;

int compare ( const void * first, const void * second ) {
    return (*(int*)second < *(int*)first);
}

struct My_compare {
    bool operator() (const int &first, const int &second) {
        return second < first;
    }
} my_compare;
```

```
int main() {

    int tabc[_SIZE_] = { 10, 50, 30, 20, 40, 60 };
    qsort( tabc, _SIZE_, sizeof(int), compare);
    for( int i=0; i<_SIZE_; i++ ) {
        printf( "%d ", tabc[i] );
    }
    printf( "\n" );

    array<int, 6> tabcpp = { 10, 50, 30, 20, 40, 60 };
    sort(tabcpp.begin(), tabcpp.end(), my_compare);

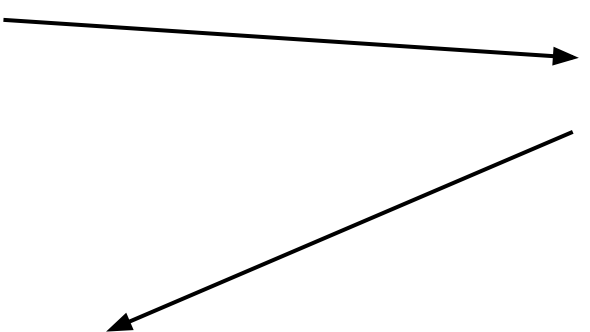
    sort(tabcpp.begin(), tabcpp.end(), [](const int &first,
const int &second){
        return second < first;});
    for (auto &itab : tabcpp) cout << itab << " ";
    return 0;
}
```

Problème : Comment une fonction peut retourner un ensemble de données ?

Une solution “classique” :

“Emballer” (wrapper) les données à retourner dans une structure :

```
struct My_return{  
    int my_int;  
    string my_string;  
};  
  
My_return my_function(){  
    return {100,"mon texte"};  
}  
  
int main() {  
    auto my_return = my_function();  
    auto my_int    = my_return.my_int;  
    auto my_string = my_return.my_string;  
    return 0;  
}
```



Il est nécessaire de définir autant de type de structure que de type de données à retourner.

Un “tuple” est une collection hétérogène de données de taille fixée.

Problème : Comment une fonction peut retourner un ensemble de données ?

Avec un tuple (C++17):

```
auto my_function(){
    return tuple{100,"mon texte"};
}
```

```
int main() {
    auto [my_int, my_string] = my_function();
    cout <<my_int<<" "<<my_string<<endl;
    return EXIT_SUCCESS;
}
```

Un peu moins simple en C++ 11:

```
tuple<int, string> my_function(){
    return make_tuple(100,"mon texte");
}
```

```
int main() {
    int my_int;
    string my_string;
    tie(my_int, my_string) = my_function();
    cout <<my_int<<" "<<my_string<<endl;
    return EXIT_SUCCESS;}

```

```
auto [first, second, third, fourth] = tuple{20, 30, "a string", 10.5f};
```

```
int first, second;
string third;
float fourth;
```

```
tie (first, second, third, fourth) = make_tuple(20, 30, "a string", 10.5f);
auto my_tuple = make_tuple(20, 30, "a string", 10.5f);
auto first1    = get<0>(my_tuple);
auto second1   = get<1>(my_tuple);
auto third1    = get<2>(my_tuple);
auto fourth1   = get<3>(my_tuple);
```

Exercice 1

Exécuter et commenter le programme C++ suivant :

```
#include <iostream>
#include <set>
using namespace std;

int main() {
    int a[] = {7, 4, 9, 1, 3, 4, 8, 2, 7, 5,
3, 6, 10, 4, 8, 10, 1, 2};
    multiset<int> s(&a[0], &a[17]);
    multiset<int>::iterator p = s.begin();
    while (p != s.end()) cout << *p++ << " ";
    return 0;
}
```

Reprendre le même code avec un container de type set.

Exercice 2

En vous basant sur la classe fraction et des algorithmes STL, écrire un programme qui permet de lire une liste de fractions à partir d'un fichier et de les afficher dans l'ordre croissant, ainsi que :

- d'afficher la somme totale des fractions.
- de supprimer les valeurs répétées,
- de supprimer les valeurs négatives.
- d'afficher la valeur minimale,
- d'afficher la valeur maximale.