Programmation
avancée
Master 1

Programmation avancée

C++

Sylvain Théry

thery@unistra.fr

ICube

ICUBE

UNIVERSITÉ DE STRASBOURG

Objets / Classes

Le C++ est une évolution du C qui permet de gérer les objets et l'héritage et intègre la notion de paramétrage à la compilation (template)

Classe: définition (description d'un objet)

- membres (données)
- méthodes (fonctions)

Objet: instanciation de la classe dans une variable en mémoire.

Définition

```
class Nom [: public Parent]
{
    private:
    …           // accès autorisé uniquement dans les
                méthodes de la classe
    protected:
    …           // accès aussi autorisé au classes dérivées
    public:
    …           // accès aussi autorisé depuis l'extérieur
};
```

Classes: utilisation

Accès aux membres et méthodes publiques d'un objet directement ou par ref (.):

```
Vec3f v(1.0,2.0,3.0);
float l = v.length();
```

Accès aux membres et méthodes publiques d'un objet par pointeur (->):

```
Vec3f vptr = new Vec3f(1.0,2.0,3.0);
float l = v->length();
```

Classes implémentation

Par convention on écrit:
- la déclaration de la classe dans un fichier xxx.h
- l'implémentation (définition) des méthodes dans un fichier xxx.cpp

On peut mettre l'implémentation le xxx.h et déclarer la fonction inline

Les fonctions inline sont récopiées à la place d'être appelée.

forward déclaration

Par défaut si on utilise une classe *A* dans la définition d'une classe *B*, il faut inclure *a.h* dans *b.h*

Afin de minimiser les inclusions, si on n'utilise que des pointeurs et des références sur *A* dans b.h on peut alors inclure *a.h* uniquement dans *b.cpp*
On doit alors faire une forward declaration de *A* dans *b.h*:

```
class A; // forward declaration
class B
{
    A* m_ptrA;
…
```

minimiser les inclusions → réduction des temps de compil

Pb des inclusions multiples

Pour éviter qu'un fichier xxx.h soit inclus plusieurs fois, et aussi éviter les inclusions cycliques, on utilise les macros #ifdef #define #endif

```
#ifndef __MA_CLASSE__
#define __MA_CLASSE__

.....

#endif
```

#pragma once fait la même chose, mais pas standard.

Fichier entête C++

Depuis la standardisation du C++, tous les fichiers d'entête de la librarie standard C++ (classes & fonctions du namespace std::) ont perdu l'extension *.h*

Exemples:

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

Le fichiers de la librarie standard C changent de nom:

<*math.h*> devient <*cmath*>

<*stdlib.h*> devient <*cstdlib*>

Pointeur

Variable (typée T*) contenant l'adresse d'une autre variable (de type T)

Déclaration: *T* ptr;*
Récupérer l'adresse d'une variable: *ptr = &var;*
Récupérer la variable par son pointeur: *T x=*ptr;*

La valeur *NULL* (***nullptr*** C++11) utilisée pour initialisé un pointeur à une valeur invalide (0)

Accéder aux membre d'une classe par un pointeur sur l'objet:
    ptr->membre

Arithmétique des pointeurs:
    ptr++ ajoute sizeof(T) à la valeur de ptr !

Allocation Dynamique

Contrairement au C, l'allocation dynamique de mémoire en C++ est typée grâce à l'utilisation de la fonction new

```
T* ptr1 = new T(param);   // alloue un objet de type T
T* ptr2 = new T[N]; // alloue un tableau de N objets de type T
```

Renvoit nullptr en cas d'erreur, si N=0 ptr2 est non null mais invalide !

La fonction delete permet de libérer la mémoire:

```
delete ptr1;  //delete pour les ptr alloués avec new
delete[ ] ptr2; //delete [ ] pour les ptr alloués avec new T[ ]
delete nullptr; // ne fait rien
```

Remarques:

- new appelle le constructeur et delete le destructeur de la classe T
- l'allocation de tableau appelle toujours le constructeur sans paramètre sur tous les éléments.
- delete[ ] appelle le destructeur sur toutes les cases du tableau.

Allocation Dynamique

Un POD (Plain Object DataType) est :
- un type de base
- une struct

Si T est un POD:

```
T* ptr1 = new T;
T* ptr2 = new T[N];   → alloue et laisse la mémoire intacte

T* ptr1 = new T( );
T* ptr2 = new T[N]( );    → alloue et initialize à zéro
```

Allocation Dynamique

Allocation / libération & nullptr

new renvoit nullptr en cas d'erreur.

delete nullptr; ne fait rien

Bonnes pratiques:

- Allocation: tester si new a renvoyé nullptr

- Libération:

```
delete ptr;  // désalloue le bloc pointé par ptr (si != nullptr)
ptr = nullptr;  // pas de pb si on refait delete ptr
```

Cast de pointeurs & références

Une instance d'objet a un type définivement fixé lors de sa déclaration.

Exemple:

int b = 3; // b est de type *int* pour toute sa durée vie.

Par contre elle peut être référencée par un des pointeurs ou des références de types différents. On utilise le cast:

```
int* b_ptr = &b; // normal
char* b_ptr_char = reinterpret_cast<char*>(b_ptr);
short& b_ref_sht = reinterpret_cast<short&>(b);
```

Cast de pointeurs & références

Pour les objets, plusieurs cast possible en C++ : T* res = xxxx_cast<T*>(ptr);

dynamic_cast

vérifie à l'éxécution que la conversion base* vers derivée* est possible, peut renvoyer NULL en cas d'échec

static_cast

même conversion vérifiée à la compilation
void* → T*
T* → const T*

const_cast

permet de supprimer l'attribut const **NE PAS UTILISER**

reinterpret_cast

T1*→T2* ou T*→int sans aucune vérification **DANGEUREUX**

Cast de type simple

A la place utiliser le cast du C:

```
float y = (float)x;
```

On peut utiliser

```
float y = static_cast<float>(x);
```

Ou l'appel du constructeur:

```
float y = float(x);
float y(x);
```

Référence

Variable (typée T&) permettant d'accéder à une variable de type T. C’est comme un alias de la variables.

C'est un pointeur déguisé, avec des contraintes:
- la référence doit être initialisée (ptr != null)
- on ne peut pas modifier la valeur du ptr

Déclaration: T& ref = var;

On utilise la référence de la même manière que la variable référencée.

Utilisation principale: passage de paramètres par référence

Paramètre par référence


```
void incremente(int x) // param par copie
{
    x=x+1;
}
```

Ne marche pas car on modifie une copie locale

```
void incremente(int& x) // param par ref
{
    x=x+1;
}
```

Fonctionne car le x de la fonction est une référence qui pointe la même variable que celle utilisée lors de l'appel

Une référence à la même taille mémoire qu'un pointeur (4/8)
=> gain de performance si le paramètre est "gros"

utilisation des références


```
T x = …;
T y = …;

T& r0;          // erreur de compilation !!

T& r1 = x;      // déclare une ref r1 sur x

T& r2 = r1;     // déclare une 2ieme ref r2 sur r1 (donc sur x)

r1 = y;         // écrase r1 (donc x) avec le contenu de y
```

On ne peut pas changer une référence (le pointeur) !

const et références

Le mot clé **const** permet de préciser si une variable, un paramètre ou une méthode est constante.

void setPos(const Vec3f& P);

la variable référencée ne sera pas modifiée dans la fonction

const Vec3f& getPos(int I) const;

la référence retournée ne pourra pas être modifiée

la méthode ne modifie pas l'objet

const et pointeur

const peut aussi s'appliquer sur les pointeurs
Attention la position du const est importante:

avant T*

const int ptr;*

*int const *ptr;*

Définit un pointeur sur une variable constante

après T*

*int * const ptr = ...*

Définit un pointeur constant sur une variable.
Ici comme dans une ref, l'adresse ne peut être modifiée

On peut évidemment écrire:

const int const ptr = ...*

const & paramètre par copie

const peut aussi s'appliquer sur de simples paramètres:

exemple:

```
void fonction(const int x)
{
    …
    x++; // erreur de compilation
    ….
}
```

But: empêcher la modification des paramètres d'une fonction.

Importance de l'utilisation du const

const peut permettre au compilateur de faire certaines optimisation, mais il permet surtout au programmeur de vérifier qu'il n'y a pas d'effet de bord dans son code

Une méthode est déclarée const car elle me doit pas modifier l'objet. Si elle renvoit une ref ou un pointeur non const sur un membre de la classe ce n'est plus cohérent !

De même appliquer une méthode non const sur une référence constante à un objet, entrainera une erreur.

Amène souvent à dédoubler certains accesseurs:
1 version const et 1 version non const

Il faut utiliser const dès la conception !

constexpr

Depuis le C++11 on peut indiquer (et vérifier) qu'une expression ou une fonction est évaluable à la compilation

Qualifier constexpr

Une constexpr est utilisable
- comme paramètre template
- dans du SFINAE

Mutable

Le mot clé **mutable** placé devant un membre d'une classe permet de l'exclure des règles du const:

Un membre mutable peut être modifié dans une méthode const

Pratique pour les membres qui ne font pas sémantiquement parti de l'objet.

Exemple: compteur d'accès en lecture:

```
class X
{
    float m_val;
    mutable int m_counter_of_get;
     ...
    float get_value() const
    {
        m_counter_of_get++;
        return m_val;
    }
};
```

Friend

Le mot clé ***friend*** permet d'indiquer au compilateur que la classe est ami avec une fonction ou une autre classe.

C'est à dire que cette fonction peut accéder aux membres privés et protégés de la classe.

A éviter, mais parfois utile

Aide au debuggage: assert

assert(expr_bool)

arrête le programme si expr_bool est fausse

affiche le fichier, n° ligne, et la fonction courante

est ignoré si le code n'est pas compilé en debug

Aide au debuggage: static_assert

static_assert(constexpr_bool, message)

arrête la compilation si *constexpr_bool* est fausse

constexpr_bool doit donc être évaluable à la compil

Qualifier: **constexpr**

Membre statique

static permet de définir un membre comme étant commun à toute les instances de la classe.

Tout membre statique doit être initialisé dans le fichier d'implémentation (.cpp)

dans a.h :

```
class A
{
    static int s_nb;
}
```

dans a.cpp :

```
int A::s_nb = 0;
```

La valeur est modifiable au cours du temps !

Méthode statique

static permet aussi de définir une méthode appelable sans instance de la classe.

Un méthode statique ne peut accéder ou modifier que des membres statiques.

Appel:

ClassName::staticMethod();

Si obj est une instance de ClassName, on peut aussi écrire:

obj.staticMethod();

static const

L'utilisation conjointe de static et const permet de définir une constante:

```
class A
{
    static const int NB = 42;
};
```

A privilégier à la place des #define (non typés !)

Attention: l'initialisation dans la déclaration n'est possible que pour les int (dans la norme !)

Constructeurs

Des méthodes très particulières !

- porte le nom de la classe
- pas de type de retour

Elles sont appelées lors de la création de l'objet:

- variable (locale, globale, muette, membre)
- allocation dynamique

On peut en définir autant que l'on veut (paramètres différents)

Constructeurs

Exemple:

```
class Vec3f
{
    Vec3f();  // default
    Vec3f(float v);
    Vec3f(float v0, float v1, float v2);
    Vec3f(const Vec3f& vec); // copie
```

Constructeur par défaut

C'est le constructeur sans argument

Il est généré par le compilateur si il n'est pas défini.

Attention si vous définissez un autre constructeur, vous devez définir le constructeur sans argument !

Ce constructeur est nécessaire par exemple pour la déclaration (statique) d'un tableau d'objets:

Vec3f tv[4]; // nécessite le constructeur par défaut

Constructeurs implémentation

Le constructeur sert à initialiser les membres de la classe. Il est conseillé d'utiliser la **liste d'intialisations**

*Vec3f(float v1, float v2, float v3)**:***
    m_x(v1),
    m_y(v2),
    m_z(v3)
{ }

Permet d'initialiser les membres (variables, références, pointeurs)
Permet d'appeler un constructeur d'une classe parent
Indispensable pour initialiser les références.
Pas d'appel de fonction en paramètres

Constructeurs de copie

Le constructeur de copie est utilisé (une version par défaut est générée si il n'est pas défini) lors de :
- passage de paramètres par copie
- retour de fonction
- duplication d'objet (affectation si pas d'opérateur =)

Attention:

```
Vec3f v2 = v1;   // equivalent a v2(v1): copy constr.

Vec3f v2;
v2  = v1;     // utilisation de l'operateur d'affectation =
```

Constructeurs (de copie) privé

Il parfois utile d'interdire la copie d'un objet.

Afin d'empêcher l'utilisation du constructeur de copie et de l'opérateur d'affectation définis par défaut, il faut les définir en “private”

Il est aussi possible de mettre tous les constructeurs en private !

Plus propre depuis C++11:

faire suivre la déclaration de “ = delete “

braced init list


```
int tableau[4] = {2,4,6,8};      // C99
POD p = {3.14f, 'X'};            //C++03
std::map<int,std::string> m{{1,"un"},{2,"deux"},{3,"trois"}}; // C++11
```

```
struct POD
{
    float ff;
    char cc;
};
```

(Sequence) Constructeur avec une braced init list en paramètre.

```
class A
{
    int data_[5];
public:
    A(std::initializer_list<int> inil)
    {
        assert(inil.size()<=5);
        int j=0;
        for (int x: inil)
            data_[ j++ ] = x;
    }
};
```

→ `A a = {5,4,3,2,1};`
`A b{5,4,3,2,1};`

Constructeur explicite

Le mot clé **explicit** placé devant un constructeur permet d'empêcher son utilisation par le compilateur pour faire de la conversion implicite.

```
class A
{
    explicit A(int i);
};
```

```
class B
{
    B(const A& a);
};
```

```
B b(3); // interdit grace à explicit
```

Sinon on peut contruire un A implicitement à partir d'un int (constructeur *A(int)*) si B b(3)

Return Value Optimisation

Le compilateur peut se permettre des optimisations si celles-ci ne modifient pas le comportement du programme
déclaration + affectation sur la même ligne → R.V.O.

```
class A { …
     A(const A&) { std::cout << "copie” << std::endl; }
};

A f( ) {…
  return A(...); // construit un A et le renvoit
}

A obj = f( ); // construction de A puis copie dans obj ?
                → non, construction directe dans obj !
```

Remarque: tous les compilateurs n'optimisent pas les même cas

Destructeur

Méthode appelée lors de la destruction de l'objet, Avant la libération de la mémoire.

Nommée ~Nom_de_la_classe()

Remarque: le destructeur est automatiquement appelé aussi sur les paramètres et variables locales (portée des variables)

Opérateurs

En C++ on peut définir l'utilisation des opérateurs sur ses propres classes.

Prog. avancée exemple:

```
Vec3f  v1,v2,v3;
…
v1 = v2+v3;
```

On peut **surcharger** les opérateurs:
+, -, /, *, +=, -=, *=, /=, ++, --, ==, <, [ ], ()

Attention à bien respecter la signature des opérateurs.
Prog. avancée exemple l'affectation (=) renvoit une ref sur l'objet !

Opérateurs

Remarques

- [ ] ne peut prendre qu'un paramètre
- ( ) peut prendre plusieurs paramètres

On peut surcharger les opérateurs existants, à utiliser avec précaution :

- * & -> new delete

Autres surchargeable, mais attention à la lisibilité :

- && || !

A ne pas confondre avec :

- & | ~

Opérateurs prefix /postfix

++ et -- peuvent être préfixés:

```
b = ++a;   // a = a+1; b = a; incrémente puis renvoit
```

ou postfixés:

```
b == a++; // b = a; a = a+1; renvoit puis incrémente
```

++a :

```
T&  T::operator++()
```

a++ :

```
T  T::operator++(int)       param int non utilisé
```

Opérateurs

Exemple:

```
float& Vec3f::operator[ ] (int I)
{
    switch(i)
    {
        case 0: return m_x; break;
        case 1: return m_y; break;
        case 2: return m_z; break;
    }
}
```

Opérateurs & const

Certains opérateurs peuvent (& doivent) être défini **const**

```
float& Vec3f::operator[ ] (int I);
const float& Vec3f::operator[ ] (int I) const;
    ou float Vec3f::operator[ ] (int I) const;
```

L'opérateur défini en const s'applique uniquement sur les références constantes d'objet.

Pour les objets et les références c'est la première version qui sera utilisée.

Opérateurs & priorités

::
a++, a--, ( ), [ ], ., ->, xx_cast
++a, --a, +a, -a, !, ~, *ptr, &var,
a*b, a/b, a%b
a+b, a-b
>>, <<
<, >, >=, <=
==, !=
&
^
|
&&
||
a?b:c
=, +=, -=, *=, /=, %=, ..
,

Pointeur this

Dans une classe la variable **this** désigne le pointeur sur l'objet courant.

On doit parfois indiquer au compilateur qu'une méthode fait partie de la classe (héritage+template)

```
this->methode()
```

Si on veut retourner l'objet courant:

```
return *this;
```

lvalue & rvalue

lvalue: représente un objet occupant une zone mémoire identifiable

rvalue: expression qui n'est pas une lvalue;
expression qui ne peut être que à droite d'un =

L'opérateur d'affectation = attend une lvalue à sa gauche

```
X+4 = 3 ; // erreur car x+4 n'est pas une lvalue
7 = f(2);  // erreur car 7 n'est pas une lvalue
sin(0.0) = 7;  // erreur car sin(0.0) n'est pas une lvalue
                                    sin(0.0) est une variable temporaire
```

Héritage

Permet d'enrichir une classe avec de nouvelles données et des nouvelles méthodes.

Permet de faire du polymorphisme (cours à suivre)

Héritage private public

Si on veut uniquement enrichir une classe en rendant les méthodes de celle-ci inaccessibles, on utilisera l'héritage ***private***.

Dans les autres case et en particulier si on veut utiliser le polymorphisme on utilisera l'héritage ***public***

Héritage & accès

Tous membres/méthodes définis en *public* et en *protected* dans une classe sont accessibles dans les classes qui en héritent (récursivement)

Tous membres/méthodes définis en private ne sont accessibles que dans les méthodes de la classe elle-même.

Héritage & surcharge

On peut surcharger une méthode de la classe parent en la redéfinissant (même déclaration)

La méthode de la classe parent est alors cachée, mais accessible en la préfixant par *ClasseParent::methode(…);*

Depuis C++11 on peut faire suivre la déclaration de la surcharge (virtuelle) par override

Héritage & constructeurs

Le constructeur par défaut de la classe parent est automatiquement appelé par tous les constructeurs d'une classe dérivée.

Attention l'appel est fait avant d'entrer dans le code du constructeur.

Si on veut appeler un autre constructeur il faut le faire explicitement dans la liste d'initialisations.

Héritage & destructeur

Le destructeur de la classe parent est automatiquement appelé par le destructeur d'une classe dérivée.

L'appel se fait après l'exécution du code du destructeur de la classe dérivée.

Classe: bonnes pratiques

Définir des types locaux:

- Self (type de this) et Inherit (type du parent)
- autres si ils sont utiles

Utiliser override pour marquer les méthodes surchargées

Utiliser =delete pour empêcher le compilateur de générer un constructeur

Regrouper les méthodes sémantiquement

→ on peut mettre public/protected/private plusieurs fois.

Namespace

Les namespace ou espace de nommage permettent de ranger et d'isoler du code.
Utile pour les noms de classe et de fonction très communs qui peuvent être dupliqués dans un gros projet.

```
namespace Projet
{
    int prj1;
    void fonc1();
}

Projet::prj1 = 2;
Projet::fonc1();
```

Namespace

On peut imbriquer les namespaces et définir des chemins d'accès:

::Projet::Prog. avancée::Nodes::

l'utilisation de using namespace xxx permet de choisir un point départ pour la recherche des noms.
Exemple: *using namespace std; // à éviter*

L'utilisation de :: en debut de chemin définit un chemin absolu.

Système similaire au chemin de fichiers Unix

Flux

En C++ il est déconseillé d'utiliser la librairie C pour les entrées-sorties. A éviter donc:

printf, scanf, fprintf, fscanf, …

Pour interagir avec la console, les fichiers, etc.. il faut utiliser les flux

Flux

Quoi	type entrée	type sortie
I/O console	std::istream	std::ostream
I/O fichier	std::ifstream	std::ofstream
I/O chaine	std::istringstream	std::ostringstream

Opérateurs:

- << pour écrire dans un flux
- >> pour lire dans un flux
- remarque: les opérateurs renvoit le flux

Console

Sortie standard:

```
std::cout << variable ;
```

Entrée standard:

```
std::cin >> variable ;
```

Sortie erreur

```
std::cerr << variable ;
```

Saut de ligne: `std::endl;`
Flush: `std::flush;`

Flux commandes

On peut envoyer des commandes dans le flux:

std::cout << std::boolalpha ;
affiche true/false au lieu de 1/0

std::cout << std::hex ;
affiche les entiers en hexadécimal

std::cout << std::setprecision(9) ;
change la précision d'affichage des réels, le nombre total de chiffres affichés (6 par défaut)

std::fixed std::scientific
std::setw(nb) std::setfill(chr) std::left std::right
…

Exemples


```
int i=1;
std::cout << “i = “<<i << std::endl;
std::string str=”Entree une valeur ”;
std::cout << str;
std::cin >> i;
std::cout << “i = “<<i << std::endl;
```

Opérateurs de flux

Fonctionnent pour tous les types prédéfinis, nécessitent la surcharge de l'opérateur de flux de ostream pour les classes.

Optionnellement dans la classe *A*:

```
friend std::ostream& operator<<(std::ostream& out, const A& a) ;
```

En **dehors** de la classe

```
std::ostream& operator<<(std::ostream& out, const A& a)
{
    out << a.m_x;
    return out;
}
```

Utilisable pour la console, les fichiers, les stringstream

stringstream

Très pratique (équivalent du sscanf) les stringstream permettent de lire/écrire des flux dans/depuis une chaine de caractères C++ (std::string)

```
std::stringstream ss;
ss << “valeur de i=”<< a;          // flux dans le stringstream
std::string chaine = ss.str();     // récuperation de toute la chaine

std::string ch;
do
{                                          // ou utilisation du stringstream
     ss >> ch;                             // pour lire les mots 1 a 1
     std::cout << ch<< std::endl;
} while (!ss.eof());
```

Flux commandes

On peut envoyer des commandes dans le flux:

std::cout << std::boolalpha ;
affiche true/false au lieu de 1/0

std::cout << std::hex ;
affiche les entiers en hexadécimal

std::cout << std::setprecision(9) ;
change la précision d'affichage des réels, le nombre total de chiffres affichés (6 par défaut)

std::fixed std::scientific
std::setw(nb) std::setfill(chr) std::left std::right ss
…

Flux exemple


```cpp
#include <iostream>
#include <fstream>
#include <sstream>


void affiche(std::ostream& s)
{
    s << "Hello"<< std::endl;
    for (auto i: {1,3,5,7})
        s << i << " / ";
    s << std::endl;
}

int main()
{
    affiche(std::cout);     // envoye dans la console

    std::ofstream fich("sortie.txt");
    affiche(fich);          // envoye dans le fichier
    fich.close();

    std::stringstream sstr;
    affiche(sstr);          // envoye dans une chaine

    std::cout << sstr.str();

    return 0;
}
```

Chaîne de caractères C++

En C++ il est déconseillé d'utiliser char* pour les chaînes de caractères. La librarie standard fournit la classe std::string (#include <string>)

Avantages de std::string:

std::string chaine = “Hello “+ch1+” et “+ch2;

+ de nombreuses méthodes très utiles (recherche, sous-chaines, comparaison, remplacement ...)

C++11 (C++0x)

Norme du langage (Septembre 2011)

Supportée par les compilateurs les récents
(éviter VisualC++ avant 2015, gcc 4.9 min)

Exemple de nouveautés:

- using (typedef template)
- auto (déduction automatique du type)
- lambda expression
- for range
- *move constructeur (rvalue reference)*
- unique_ptr / shared_ptr
- template avec nombre de paramètres variable
- SFINAE

C++11 using

using à la place de typedef

Exemple:

```
using  MyInt = int32_t;
```

Avantage, peut être template:

```
template <typename T>
using Vec3 = Vecteur<3,T>
```

→ *Vec3<float>* → *Vecteur<3,float>*

Ne pas hésiter à définir des types locaux (à des classes ou fonctions) pour simplifier l'écriture.

C++11 auto

auto : type déterminé par inférence

Exemple:

```
auto it = my_vec.begin( );
auto* ptr = …
auto& ref = …
const auto& cref= …
```

A utiliser avec …

decltype v détermine le type de la variable v

```
using T = decltype f(x);
```

C++11 lambda

Fonctions anonyme:

auto f = [&] (int I, int j) { …. } → float

[ ] ne capture rien de l'environnement
[&] capture tout l'environnement par référence.
[=] capture tout l'environnement par copie.
[this] capture l'objet courant par référence.
[a, &b] capture a par copie et b par référence.

C++11 lambda

Pratique pour passer "du code" en paramètre

pour lancer des threads:

```
std::thread th( [ & ] ( ) { ... } );
```

pour appliquer une fonction à tous les elts d'un container

```
std::vector<int> v={1,2,3,4,5};
std::for_each(v.begin(), v.end(), [ ] (int& x) { x=x*3;} );
```

C++11 for range

Pour parcourir un container (vector/list/map/...) on utilise les iterateurs (généralisation de pointeurs).

```
std::vector<int> vect = {1,2,3,4,5};
for(std::vector<int>::iterator it = vect.begin(); it != vect.end(); ++it)
    std::cout << *it << std::endl;
```

En C++11 on peut faire:

```
for(auto i : vect)   // ou auto& i : vect  ou const auto& i : vect
    std::cout << i << std::endl;
```

S'applique à tout objet qui fournit:
- une sous-classe iterator avec les opérateurs ++ != *
- les méthodes begin et end (→ iterator)

C++11 for range


```
class MyTable
{
    static const int NB=20;
    int vec_[NB];
public:
    class iterator
    {
        int* ptr_;
    public:
        iterator(int* v,int i):ptr_(v+i)
        {}

        iterator& operator ++()
        {
            ++ptr_;
            return *this;
        }

        bool operator !=(const iterator& it)
        {
            return ptr_ != it.ptr_;
        }

        int& operator *()
        {
            return *ptr_;
        }
    };
```

```
    iterator begin()
    {
        return iterator(vec_,0);
    }

    iterator end()
    {
        return iterator(vec_,NB);
    }
};


int main()
{
    MyTable ta;

    int cpt=3;
    for(auto& x: ta)
        x = cpt++;

    for(const auto& x: ta)
        std::cout << x << " / ";
    std::cout << std::endl;

    return 0;
}
```

Rvalue reference

T& est une référence (sur une lvalue de type T).

T&& est une référence sur une rvalue de type T.

void fonc(**T&&** a);

fonc ne peut être appelé qu'avec un param rvalue:
ex: fonc(a+b);
fonc(f2(c)); // avec T f2(...)

On peut *caster* une lvalue-ref en rvalue-ref :
static_cast<T&&>(a);
std::move(a);

Rvalue reference

Quel intérêt d'écrire void fonc(T&& a); ?

On a le droit de modifier le param a, de lui voler ses ressources car il sera détruit juste après (rvalue)

```
class T
{
    …
    int* data_;
};

fonc(T&& a)
{
    int* local = a.data_; // je recupere data (sans copier)
    a.data_ = nullptr; //  necessaire pour empêcher le delete
}
```

C++11 move constructeur move affectation

Constructeur et affectation qui implémentent le déplacement des données du paramètre vers *this*.

```
A(A&& a);   // move constructor
A& A::operator = (A&& a); // move affectation
```

L'objet *a* devient invalide car on a transféré ses données (copie de pointeurs, swap de container, …).

Ils sont appelés automatiquement quand ça a du sens:

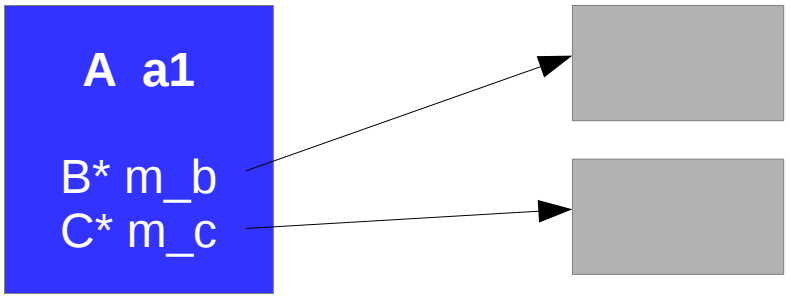
→ sur une rvalue (retour de fonction, appel au constructeur).

→ sur un std::forward de rvalue.

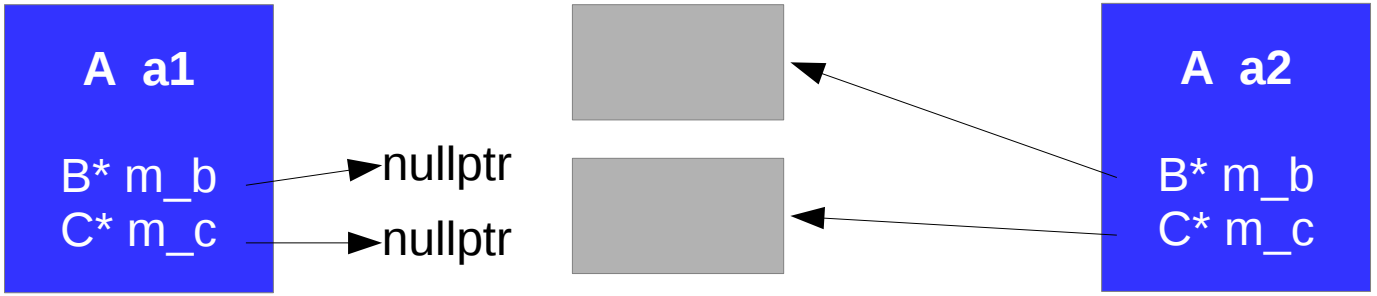
On peut l'appeler explicitement par A a2 = std::move(a1);

Permet d'éviter des copies inutiles → performance

C++11 move



A a2 = std::move(a1);



SmartPointer unique

std::unique_ptr<T>

est un pointeur de T (T*) géré par le compilateur:
- lorsque l'objet unique_ptr est détruit, il y a désallocation.
- lorsque l'objet unique_ptr est écrasé, il y a désallocation.

std::unique_ptr<T> p1(T*) // création à partir d'un pointeur

Pour écraser un unique_ptr:
  p1= std::move(p2); *// transfert p2 sur p1 → p2 invalide*

On n'a pas le droit de dupliquer (unique !)
  p1= p2; *// affectation interdite sinon plus d'unicité*

SmartPointer shared

std::shared_ptr<T>

est un pointeur de T (T*) géré par le compilateur:
- lorsque plus personne ne pointe vers la mémoire, celle-ci est désallouée

Déclaration: *std::shared_ptr<T> p(T*);*

Déclaration avec allocation de l'objet pointé:
std::shared_ptr<T> p = std::make_shared<T>(params)
~ std::shared_ptr<T> p(new T(params))