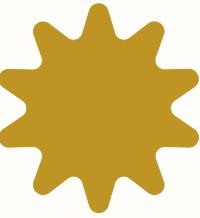


**POLYTECH SORBONNE**

# **PROGRAMMATION OBJET AVANCÉE**



Millan Mégane  
(megane.millan@inria.fr)  
2024 - 2025



# PRÉSENTATION DU COURS

## Objectif du cours

- Savoir utiliser les outils de compilation (cmake...)
- Maitriser le C++

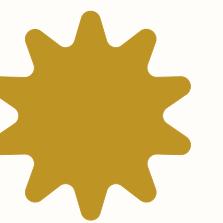
## Déroulement

- 6 séances de cours (2h)
- 6 séances de TP (4h)
- 1 séance d'évaluation du projet

## Évaluation

- 90% Projet et 10% TP



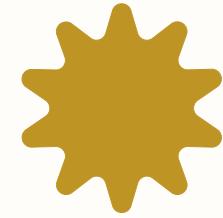


## PROGRAMME & CONTENU

Programmation objet  
en C++

C++ et compilation  
avec CMake

Concept de C++  
avancé



# PRÉSENTATION DU PROJET

## Projet

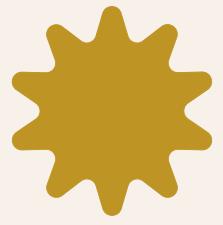
- Faire un jeu vidéo en utilisant vos connaissances en programmation objet et en C++

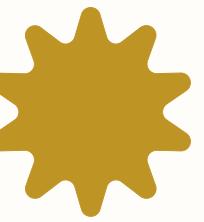
## Évaluation du projet

- Evaluation du code (propreté, commentaire, doc, tests unitaires...)
- Présentation de 10 min et 10 min de question



# RAPPEL PROGRAMMATION OBJET

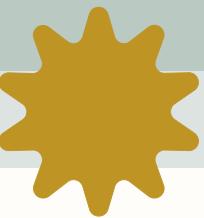




## RAPPEL

- **Programmation Objet** - Modéliser des concepts abstraits sous formes de familles de classes d'objets
  - **Héritage** - Une ou plusieurs classes ont une base commune et on peut donc factoriser du code, en utilisant l'héritage
  - **Polymorphisme** - Si l'argument d'une fonction a des classes dérivées, alors on peut passer des objets de ces classes dérivées à cette fonction



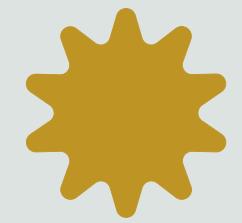


## RÉSUMÉ

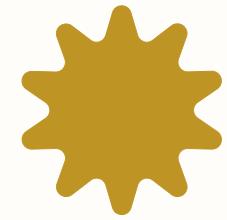
---

### **Une bonne classe**

- Pas ou peu d'attributs publics
- Avoir un sens – représenter un concept
- Une interface simple
  - peu de méthodes publiques
- **Et doit être documentée**



# **PROGRAMMATION ORIENTÉE OBJET EN C++**

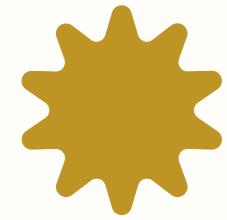


## C++

### Philosophie

- Statiquement typé, généraliste, efficace et “portable”
- Supporte beaucoup de style de programmation (objet ou non...)
- Compatible en grande partie avec le C
- Ne nécessite pas d'environnement particulier





# C++

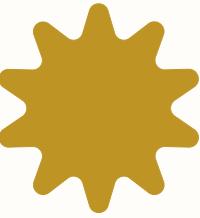
## Quand l'utiliser ?

- Besoin de rapidité, et d'abstraction
- Programmation système
- Programmation bas niveau

## Mais

- Gestion de la mémoire, pas forcément automatique (pointeurs)
- Parfois complexe à debugger (segmentation fault)
- Syntaxe qui peut devenir illisible





# SYNTAXE DE BASE

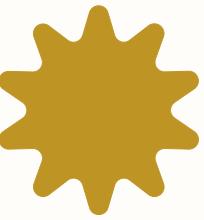
```
# include <iostream>

int main(int argc , char ** argv)
{
    std::cout << " Hello world " << std::endl ;
    return 0;
}
```

Pour compiler :

```
g ++ -o hello hello.cpp
```

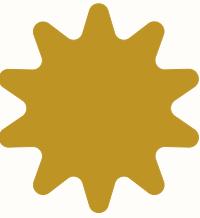




# SYNTAXE DE BASE

```
class Test {  
public :  
Test ( int x , int y )  
{  
_x = x ;  
_y = y ;  
std::cout << " Constructeur " << std::endl ;  
}  
  
int getX () { return _x ; }  
int getY () { return _y ; }  
  
int _x ;  
int _y ;  
}
```





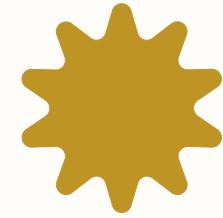
# SYNTAXE DE BASE

Comme en C, on sépare les définitions de l'implémentation

```
# ifndef A_hpp  
# define A_hpp  
class A {  
    int methode () ;  
}  
# endif
```

```
# include "a.hpp "  
int A::methode ()  
{  
    return 1;  
}
```



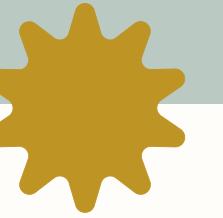


## SYNTAXE DE BASE

En C++, on peut définir facilement quel accès et portée on veut donner à nos méthodes et attributs

- **Privée** - Accessible seulement à l'intérieur de la classe
- **Protected** - Accessible pour la classe et les classes dérivées
- **Public** - Accessible partout





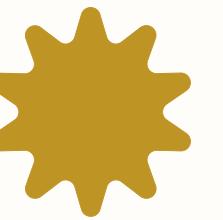
## Un peu de pratique

Créez une classe Rectangle avec :

- 2 attributs - longueur et largeur
- 2 méthodes - calculAire et calculPerimetre

Compilez et testez votre classe



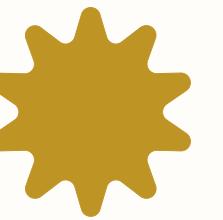


# CONSTRUCTEURS ET DESTRUCTEURS

Fonctionne globalement comme en Python

```
class Test
{
public :
    // constructeur
    Test () {
        _x = 0;
        _y = 0;
        _z = new float [42];
    }
protected :
    int _x ;
    int _y ;
    float * _z ;
};
```



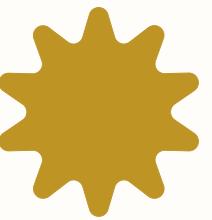


# CONSTRUCTEURS ET DESTRUCTEURS

Ecriture simplifiée pour les attributs  
(Effet identique)

```
class Test
{
public :
    // constructeur
    Test () :
        _x (0) , _y (0) {
        _z = new float [42];
    }
protected :
    int _x ;
    int _y ;
    float * _z ;
};
```



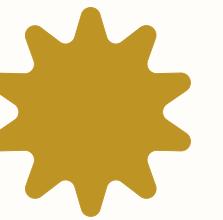


# CONSTRUCTEURS ET DESTRUCTEURS

```
class Test
{
    public :
        // constructeur par defaut
        Test () : _x (0) , _y (0) , _z (new float [42])
        { /* code*/ }
        // Autres constructeurs
        Test ( int x , int y ) : _x ( x ) , _y ( y ) , _z(NULL) { /*code */ }
        // Constructeur par copie
        Test ( const Test & o) : _x ( o . _x ) , _y ( o . _y ) , _z (new float [42]){
            for ( size_t i = 0; i < 42; ++ i )
                _z [ i ] = o . _z [ i ];
        }

    protected:
        int _x;
        int _y;
        float* _z;
};
```



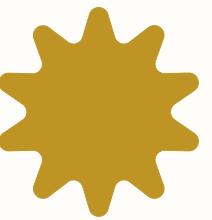


## CONSTRUCTEUR PAR COPIE

Le constructeur par copie est important en C++ - il est invoqué lorsqu'une variable est déclarée avec une initialisation.

```
Test t ; // Constructeur par defaut  
Test t2 = t ; // Appel du constructeur par copie  
Test t3 ( t ) ; // idem
```





## DESTRUCTEUR

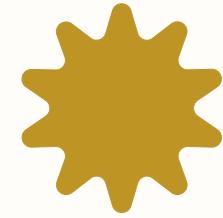
Le destructeur est une méthode automatiquement appelée quand un objet est détruit.

Nécessaire pour détruire aussi les objets liés et libérer la mémoire Il existe un destructeur par défaut (qui ne fait rien).

```
class Test
{
    public :
        Test () : _z ( new float [42]) {} // constr. par defaut
        ~Test () { // Destructeur
            delete [] _z ;
        }
    protected :
        float* _z ;
};
```

Il n'y a qu'un seul destructeur par classe (aucun argument, ne renvoie rien).





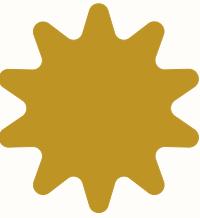
# OPÉRATEURS

Comme en python, on peut redéfinir tous les opérateurs (`=, ==, !=, ||, +, -, += ...`)  
**Chaque opérateur sera une méthode de la classe.**

## Prototype

```
Test& operator=(const Test& o):
```



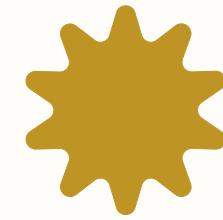


# HÉRITAGE

```
class A
{
    private:
        int x2;
    protected:
        int x3;
};

class B : public A
{
    public:
        void print() {
            std::cout << x3 << std::endl;
        }
        // On ne peut pas accéder à x2
};
```





# POLYMORPHISME

## Rappel

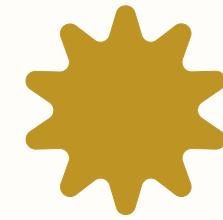
- Si une classe **B** dérive d'une classe **A**, alors toute fonction ayant pour argument une instance de **A** peut être appelée avec une instance de la classe **B**

```
A* a = new A();
B* b = new B();
A* b2 = new B(); // okay, mais uniquement avec des pointeurs

void une_fonction(A* obj) {....}

une_fonction(a);
une_fonction(b); // subsomption
une_fonction(b2); // idem
```





# POLYMORPHISME

Ne fonctionne pas avec un passage par valeur !

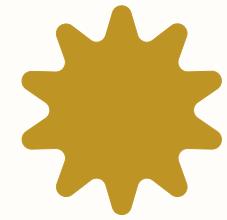
```
A a = new A();
B b = new B();

void une_fonction(A* obj) {....}

une_fonction(a);
// une_fonction(b); incorrect car copie partielle
```

- Ne fonctionne pas avec un passage par valeur !





# OPÉRATEURS

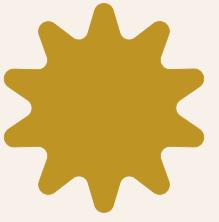
Comme en python, on peut redéfinir tous les opérateurs (`=, ==, !=, ||, +, -, += ...`)  
**Chaque opérateur sera une méthode de la classe.**

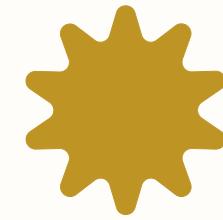
## Prototype

```
Test& operator=(const Test& o):
```



# **SPÉCIFICITÉS DU C++**





# PASSAGE PAR RÉFÉRENCE

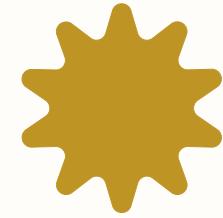
- Une référence est un “alias” vers le même espace mémoire qu’une autre variable, mais avec un autre nom
  - Les deux variables ont **la même adresse mémoire**

```
int x = 42;
int& y = x; // référence vers x

std::cout << "x " << x << std::endl ; // 42
std::cout << "y " << y << std::endl ; // 42 aussi

y = 23;
std::cout << "x " << x << std::endl ; // 23
std::cout << "y " << y << std::endl ; // 23 aussi
```





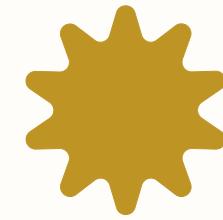
# PASSAGE PAR RÉFÉRENCE

- Utilisation typique - **Passage de variable par référence**
- Permet de lire et modifier directement la variable passée en argument d'une fonction sans la copier (donc plus rapide)

```
void func(int &arg)
{
    arg = 43;
}

int main (int argc, char *argv[])
{
    int x = 23;
    func(x) ;
    std::cout << "x " << x << std::endl ;
}
```

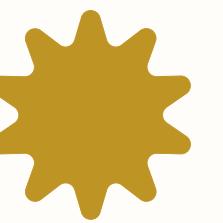




# PASSAGE PAR RÉFÉRENCE

- Différence entre les références et les pointeurs
  - **Référence** : Alias pour une variable existante. Une fois une référence liée à une variable, elle ne peut pas être modifiée pour référer à un autre objet. N'occupe pas d'espace mémoire supplémentaire.
  - **Pointeur** : Variable qui contient l'adresse mémoire d'une autre variable. Un pointeur peut être réaffecté pour pointer vers une autre adresse. Occupe de la mémoire pour stocker l'adresse.





# NAMESPACE

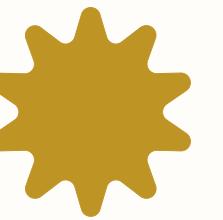
- Fonctionnalité qui permet d'organiser le code en regroupant des classes, fonctions, constantes ou variables sous un même nom logique.
- Eviter les conflits de noms lorsque des projets ou des bibliothèques utilisent des noms identiques.

```
#include <iostream>

namespace MonEspaceDeNoms {
    int x = 42;
    void afficher() {
        std::cout << "Valeur de x : " << x << std::endl;
    }
}

int main() {
    // Accès aux membres du namespace
    std::cout << MonEspaceDeNoms::x << std::endl; // Affiche 42
    MonEspaceDeNoms::afficher(); // Appelle la fonction afficher()
    return 0;
}
```





# NAMESPACE

Pour simplifier l'accès, vous pouvez utiliser le mot-clé **using**:

- **Using namespace** : Cela rend tous les membres accessibles sans le préfixe du namespace.
- **Using spécifique** : Cela rend un membre spécifique accessible.

```
#include <iostream>

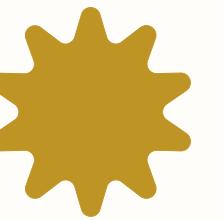
namespace MonEspaceDeNoms {
    int x = 42;
    void afficher() {
        std::cout << "Valeur de x : " << x << std::endl;
    }
}

int main() {
    using namespace MonEspaceDeNoms; // On importe tout
    afficher(); // Appelle afficher() sans préfixe
    std::cout << x << std::endl; // Accès direct à x

    // Alternative : importer un membre spécifique
    using MonEspaceDeNoms::x;
    std::cout << x << std::endl;

    return 0;
}
```





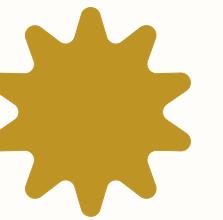
# NAMESPACE

- **Organisation** - Regroupe des entités logiquement liées
- **Evitement des conflits** - Résout les conflits de noms dans des projets complexes
- **Portée locale** avec les namespaces anonymes (limite la portée à un fichier particulier)

```
namespace {
    void fonctionLocale() {
        std::cout << "Fonction accessible uniquement dans ce fichier." << std::endl;
    }
}

int main() {
    fonctionLocale(); // Accessible ici
    return 0;
}
```





# NAMESPACE

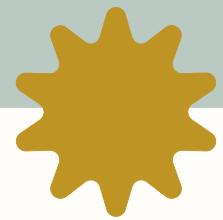
Les **namespaces** sont largement utilisés dans les bibliothèques standard de C++, comme le namespace **std** :

```
#include <iostream>

using namespace std;

int main() {
    cout << "Hello, world!" << endl; // Pas besoin de préfixe std::
    return 0;
}
```



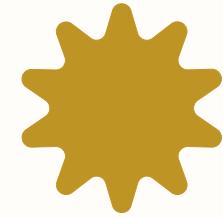


## Un peu de pratique

- Placez la classe Rectangle dans un namespace nommé Geometrie pour éviter d'éventuels conflits de noms.
- Implémentez une fonction afficherDimensions (en dehors de la classe) qui prend une référence constante à un objet Rectangle et affiche ses dimensions.
- Surchargez l'opérateur == pour comparer deux rectangles (basé sur leur aire).

Testez le tout dans la fonction main.





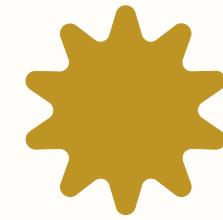
# CONST

Le mot-clef **const** permet de manipuler des objets en "lecture seule".

- Utilisé
  - Avec des **références** - la référence permet de lire mais pas de modifier l'objet
  - Avec des **pointeurs** - on peut lire mais non modifier l'objet pointé
  - Dans la **signature des méthodes** - la méthode ne peut pas modifier **\*this**, seulement le lire
  -

**Détecté à la compilation** - si les const ne sont pas respecté, le programme ne compile pas !





# CONST

## Déclaration d'une variable constante

```
const int age = 25;  
age = 30; // Erreur : impossible de modifier une variable `const`
```

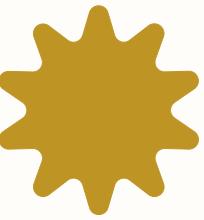
## Pointeurs Constants

```
int a = 10;  
int *const ptr = &a; // Le pointeur est constant, mais pas la valeur pointée
```

```
const int b = 20;  
const int *ptr = &b; // La valeur pointée est constante
```

```
const int c = 30;  
const int *const ptr = &c; // Ni le pointeur, ni la valeur ne peuvent changer
```



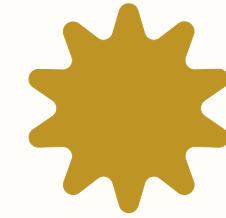


# CONST

## Méthodes constantes

```
class Personne {  
private:  
    std::string nom;  
public:  
    Personne(std::string n) : nom(n) {}  
    void afficher() const { // Ne peut pas modifier les attributs de l'objet  
        std::cout << "Nom: " << nom << std::endl;  
    }  
};
```





# CONSTEXPR

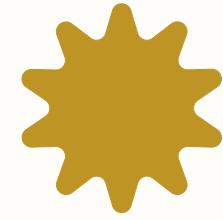
Le mot-clé **constexpr** en C++ est une extension de **const** qui permet de garantir qu'une expression peut être évaluée à la compilation.

## Différences entre **const** et **constexpr**

- **const** garantit qu'une variable ne peut pas être modifiée après son initialisation.
- **constexpr** garantit que la valeur est connue et évaluée à la compilation, ce qui permet d'optimiser les performances.

```
constexpr int carre(int x) {
    return x * x;
}
constexpr int val = carre(5); // Calculé à la compilation
```



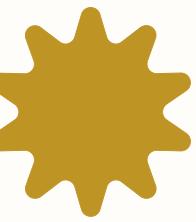


# CONSTEXPR

## Conditions d'utilisation :

- **constexpr** doit contenir uniquement des expressions pouvant être évaluées à la compilation.
- Les fonctions **constexpr** doivent retourner une valeur connue à la compilation.



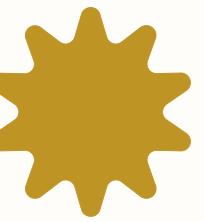


# STATIC

Le mot-clé **static** en C++ est utilisé pour modifier le comportement d'une variable, d'une méthode ou d'une fonction.

Il change la durée de vie et/ou la portée d'un élément.





# STATIC

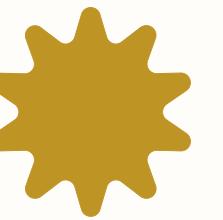
## Variables locales static

- Une variable **static** dans une fonction conserve sa valeur entre les appels.

```
void compteur() {
    static int count = 0;
    count++;
    std::cout << "Appel : " << count << std::endl;
}
```

Chaque appel à `compteur()` incrémente `count` sans réinitialisation.





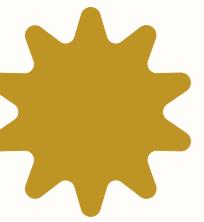
# STATIC

## Membre static

- Une variable ou méthode **static** appartient à la classe et non aux instances.
- Partagée par toutes les instances de la classe.

```
class Exemple {  
public:  
    static int valeurPartagee;  
    static void afficherValeur() {  
        std::cout << "Valeur partagée : " << valeurPartagee << std::endl;  
    }  
};  
  
int Exemple::valeurPartagee = 10; // Initialisation obligatoire
```

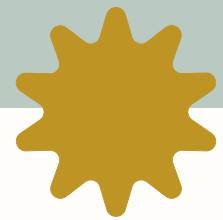




## STATIC VS NON STATIC

<b>Caractéristiques</b>	<b>Membre static</b>	<b>Membre non static</b>
<b>Appartenance</b>	À la classe	À une instance spécifique
<b>Mémoire</b>	Partagée entre toutes les instances	Unique pour chaque instance
<b>Accès</b>	Accessible sans créer d'instance	Nécessite une instance pour accéder

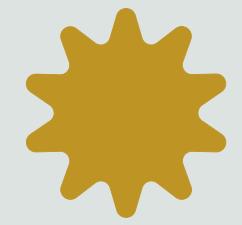




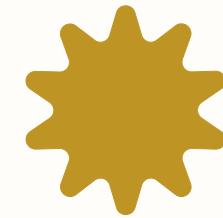
## Un peu de pratique

- Trouvez quelles fonctions de la classe Rectangle peuvent utiliser le mot clé **const**
- Ajoutez un membre **static** à la classe Rectangle pour compter les instances créées.
  - Mettez à jour ce compteur dans le constructeur.
  - Ajoutez une méthode **static** pour accéder à ce compteur.





## NOTIONS AVANCÉES



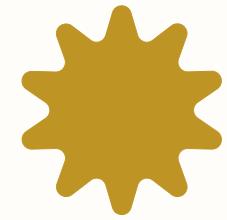
# CLASSE ABSTRAITE

Comme en Python, il est possible de faire des **classes abstraites**, qui ne seront pas instanciées.

- Pas de définition explicite comme en python.
  - Une classe est abstraite si elle contient au moins une fonction purement virtuelle.

```
virtual void methode() = 0;
```



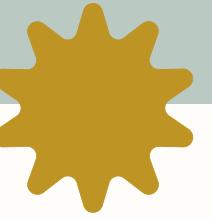


# CLASSE ABSTRAITE

```
class Forme {
public:
    virtual void dessiner() const = 0; // Virtual pour le polymorphisme
    virtual ~Forme() {} // Destructeur virtuel
};

class Cercle : public Forme {
public:
    void dessiner() const override { // Redéfinition
        std::cout << "Dessiner un cercle" << std::endl;
    }
};
```

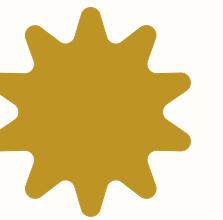




## Un peu de pratique

- Créez une classe abstraite Forme qui va devenir le parent de la classe Rectangle.





## CODE GENERIQUE

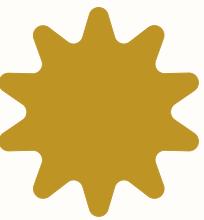
Pour faire du code générique, on peut :

- Utiliser le **polymorphisme**

**Exemple** - Fonction de **swap** de valeur. Obligation de redéfinir pour chaque type.

- Duplication de code inutile !



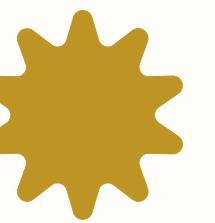


## CODE GENERIQUE

### Défauts

- Extensibilité - nécessite de redéfinir les fonctions pour chaque type
- Fiabilité - Perte d'information de type (cast en **Object**)
- Lenteur - Appel d'une fonction virtuelle au runtime



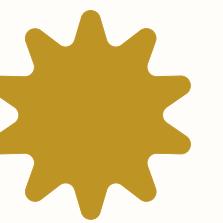


## CODE GENERIQUE

```
// int
void swap ( int &a , int & b ) {
    int tmp = a ;
    a = b;
    b = tmp ;
}

// double
void swap ( double &a , double & b ) {
    double tmp = a ;
    a = b;
    b = tmp ;
}
```



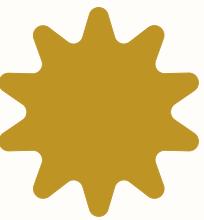


# TEMPLATE

La solution pour rendre son code générique en C++ - **les templates**

```
template < typename T >
void swap ( T & a , T & b )
{
    T tmp = a ;
    a = b;
    b = tmp ;
}
```



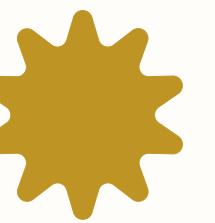


## TEMPLATE

Quand on appelle une fonction générique, le code spécifique est généré par le compilateur

- Le code n'est vérifié par le compilateur que s'il est instancié
- Temps de compilation allongés
  - On parse à chaque fois tout le code
  - Message d'erreur parfois très longs et très peu lisibles
    - Taille du binaire plus gros





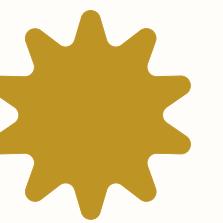
# TEMPLATE

Le mot clé **typename** permet de récupérer automatiquement le nom du type d'un argument.

```
template < typename T1, typename T2>
void fonction(T1& a, T2& b)
{
    // Algorithme
}
class A{...};
int main ()
{
    float x , y ;
    fonction<float , float>fonction (x , y) ;

    int m ;
    A a;
    fonction (m , a) ; // deduction automatique
}
```





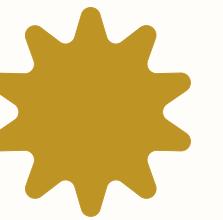
# TEMPLATE

On peut également donner des types par défaut.

```
template<typename T = float, typename X = int>
T myFunction (X x )
{
    T result = static_cast <T>( x ) / 2;
    return result ;
}

int main ()
{
    int a = myFunction <int , float>(23) ; //returns 11 (int)
    float b = myFunction <float , float>(23) ; // returns 11.5
    float c = myFunction<>(23) ; // returns 11.0 (float)
    return 0;
}
```





## PARAMÈTRES CONSTANTS

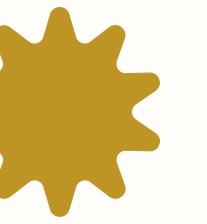
On peut également passer des paramètres en template.

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

template <int N , typename X>
void fill (X & x) {
    for (size_t i = 0; i < x . size () ; ++i)
        x[i] = N;
}

int main (){
    std :: vector <float> v(10) ;
    fill <42 , std::vector<float>>( v ) ;
    fill <42>( v ) ; // deduction du 2eme parametre
    for(int i = 0; i < 10; i++)
        std::cout << v [ i ] << std::endl ; // prints 42
    return 0;
}
```





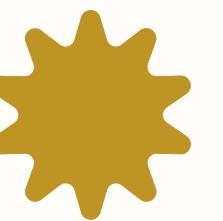
# CLASSE TEMPLATE

```
template<typename T>
class A {
public :
    const T& x() const { return _x; }
    void x(const T& t);
private :
    T _x;
};

template < typename T >
void A <T>:: x(const T& t){
    _x = t ;
}

int main () {
A<int> a;
// Example usage
a.x(42);
std::cout << "a.x() : " << a.x () << std::endl ;
return 0;
}
```





## TYPEDEF

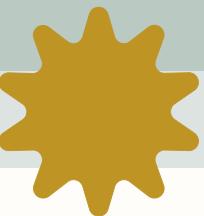
**Typedef** permet de définir un alias pour un type.

```
typedef int * intptr_t;
typedef unsigned int uint_t;

typedef std::vector<std::vector<int>> mat_t;

mat_t m;
mat_t::iterator it = m.begin();
// Compare with :
std::vector<std::vector<int>>::iterator it = m.begin();
```

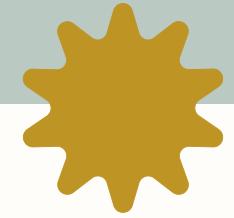




## CONCLUSION

---

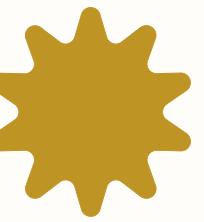
- Un mécanisme très puissant
- Une autre forme d'abstraction
- Exécution rapide (pas de coût pour l'abstraction)
- Mais compilation très lente
  - Très utilisé dans le C++ moderne
  - Mais peut très vite devenir compliqué



## Un peu de pratique

- Créez une classe template Pair qui peut stocker deux formes ou deux rectangles, ou deux carrés....
- Ajoutez des méthodes pour récupérer et modifier chaque élément de la paire.
- Testez cette classe en créant des paires de formes, de rectangles, ou même une paire où le premier élément est une forme et le deuxième un rectangle.
- Créez une fonction template aireDifference qui prend deux paires de formes ou de rectangles et retourne la différence d'aire entre les deux paires.



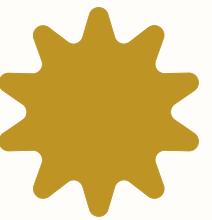


## RAPPEL POINTEUR DE FONCTION

En C++, les pointeurs ont perdu un peu de leur utilité suite à l'introduction des références.

Cependant, il est un domaine où les pointeurs sont irremplaçables, ce sont les pointeurs sur des fonctions.





## RAPPEL POINTEUR DE FONCTION

En C++, les pointeurs ont perdu un peu de leur utilité suite à l'introduction des références.

Cependant, il est un domaine où les pointeurs sont irremplaçables, ce sont les **pointeurs sur des fonctions**.

```
#include <cmath>
using namespace std;

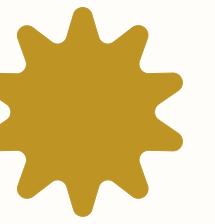
//Liste des fonctions "calculables"

//double f(double x) { return x*x;}
double f(double x) { return 1/x;}
//double f(double x) { return sqrt(x);}
//double f(double x) { return exp(x);}
//...

double minimum(double a,double b)
{
    double min(100000);

    for(double x=a; x<b; x+= 0.01)
        min = min< f(x)? min : f(x);
    return min;
}
```





# RAPPEL POINTEUR DE FONCTION

```
//Type énuméré représentant les fonctions calculables
enum Fonctions{CARRE, INVERSE, RACINE, EXPONENTIELLE};

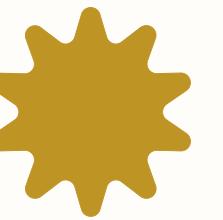
//Liste des fonctions "calculables"

double carre(double x) { return x*x;}
double inverse(double x) { return 1/x;}
double racine(double x) { return sqrt(x);}
double exponentielle(double x) { return exp(x);}

double minimum(double a,double b,Fonctions fonction_choisie)
{
    double min(100000);
    switch(fonction_choisie)
    {
        case CARRE:
            for(double x=a; x<b; x+= 0.01) //On parcourt l'intervalle
                min = min< carre(x)? min : carre(x);
            break;
        case INVERSE:
            for(double x=a; x<b; x+= 0.01) //On parcourt l'intervalle
                min = min< inverse(x)? min : inverse(x);
            break;
        //...
    };
    return min;
}
```

On peut utiliser le type **enum**.





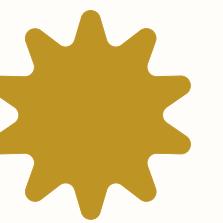
## RAPPEL POINTEUR DE FONCTION

Une fonction n'est pas un objet ou une variable, il n'est pas possible de passer une fonction directement en argument à notre fonction minimum.

```
int (*pointeur_1)(int);
// Déclaration d'un pointeur nommé "pointeur_1" qui pourra pointer
// sur des fonctions recevant un int et renvoyant un int.

int (*pointeur_2)(int,double);
// Déclaration d'un pointeur nommé "pointeur_2" qui pourra pointer
// sur des fonctions recevant un int et un double et renvoyant un int.
```





# RAPPEL POINTEUR DE FONCTION

## Affecter un pointeur sur fonction

```
#include <string>
using namespace std;

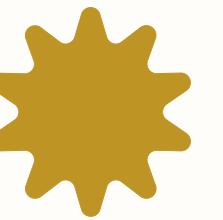
int fonction(double a,string phrase) //Une jolie fonction
{
    //blablabla
}

int main()
{
    int (*monPointeur)(double,string); //On déclare un pointeur sur fonction

    monPointeur = fonction; //Et on le fait pointer sur "fonction"
    // Ou bien monPointeur = &fonction;

}
```





# RAPPEL POINTEUR DE FONCTION

## Le cas particulier des fonctions membres de classe

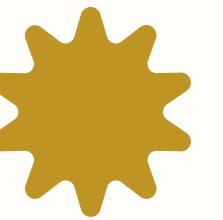
```
class A{
public:
    int fonction(int a);
    //...
};

int A::fonction(int a){return a*a;}
int main()
{
    int(*ptr)(int) = fonction;      //non

    int(*ptr)(int) = A::fonction;  //non

    int (A::*ptr)(int) = &A::fonction; //oui
}
```





# RAPPEL POINTEUR DE FONCTION

## Utiliser ce pointeur

```
int maximum(int a,int b) //Retourne le plus grand de deux entiers
{
    return a>b ? a : b;
}

int main()
{
    int (*ptr) (int,int); //Un joli pointeur

    ptr = maximum; //que l'on affecte à la fonction "maximum"

    int resultat = (*ptr)(1,2); //On calcule le maximum de 1 et 2 via la fonction pointée
    //Notez l'utilisation obligatoire des ()

    int resultat_2 = ptr(3,4); //Et on fait la même chose pour 3 et 4
    //Notez l'absence de *
}
```



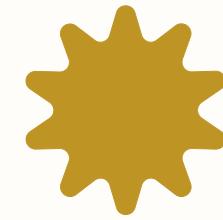
## Un peu de pratique

- Réécrire le code suivant en utilisant les pointeurs de fonctions

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

//Liste des fonctions "calculables"
//double f(double x) { return x*x;}
double f(double x) { return 1/x;}
//double f(double x) { return sqrt(x);}
//double f(double x) { return exp(x);}

double minimum(double a,double b)
{
    double min(100000);
    for(double x=a; x<b; x+= 0.01)
        min = min< f(x)? min : f(x);
    return min;
}
int main()
{
    cout << "De quelle fonction voulez-vous chercher le minimum ?" << endl;
    cout << "1 -- x^2" << endl;
    cout << "2 -- 1/x" << endl;
    cout << "3 -- racine de x" << endl;
    cout << "4 -- exponentielle de x" << endl;
    int reponse;
    cin >> reponse;
    cout << "Le minimum de la fonction entre 3 et 4 est: " << minimum(3,4) << endl;
    return 0;
}
```



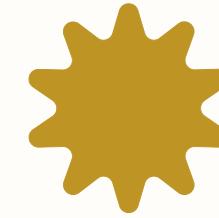
# EXPRESSION LAMBDA

**Fonction, potentiellement anonyme, pour des opérations locales.**

```
[zone de capture] (paramètres de la lambda) -> type de retour { instructions }
```

- **Capture**: par défaut, totale isolation = aucune variable externe disponible. Permet de modifier des variables extérieures
- **Paramètres**
- **Type de retour**
- **Instructions**



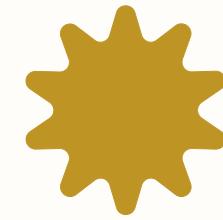


# EXPRESSION LAMBDA

## Pourquoi ?

- **Optimisation** : Le compilateur optimise mieux les lambdas que les fonctions classiques.
- **Portée** : Une fonction est globale alors qu'une lambda peut rester locale.
- **Choix** : C++ permet d'utiliser les deux, mais les lambdas sont plus adaptées dans certains cas.





# EXPRESSION LAMBDA

## Capture

On peut choisir de capturer une variable par valeur ou par référence

## Attention

Lorsque l'on capture par référence, il faut s'assurer que la durée de vie de la lambda ne dépassera pas celle de la variable référencée.

```
int value = 0;

// Capture par valeur.
const auto display_copy = [value]() { std::cout << value << std::endl; };

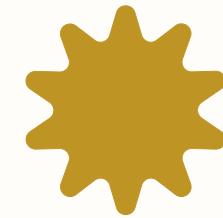
// Capture par référence.
const auto display_ref = [&value]() { std::cout << value << std::endl; };

display_copy(); // => 0
display_ref(); // => 0

value = 5;

display_copy(); // => 0
display_ref(); // => 5
```





# EXPRESSION LAMBDA

```
int main()
{
    auto lambda = [](int v){ std::cout << v << std::endl; };
    lambda(3);
    return 0;
}

// pourrait être traduit par le compilateur avec :

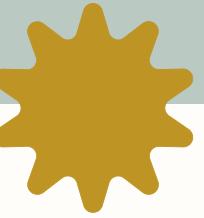
void nom_autogenere_incomprehensible(int v)
{
    std::cout << v << std::endl;
}

int main()
{
    void (*lambda)(int) = &nom_autogenere_incomprehensible;
    lambda(3);
    return 0;
}
```

## Compilation

Le type de la lambda va être généré au cours de la compilation.

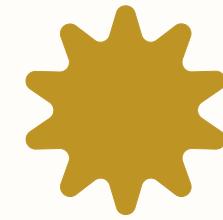
Donc il faut utiliser **auto** ou bien de la wrapper à l'intérieur d'un objet de type **std::function**, si la lambda renvoie une valeur.



## Un peu de pratique

- Écrire une lambda qui compare deux valeurs et l'utiliser pour trouver le maximum dans un tableau.





# FONCTEURS

En C, on utilise souvent des pointeurs de fonctions :

- Pour passer une fonction en argument d'une autre
- Pour faire des **callbacks**

```
typedef float (*fun_t)(float);

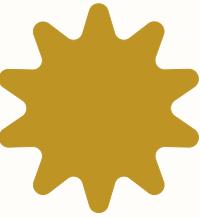
float add1(float x){
    return x + 1;
}

float sub2(float x){
    return x - 2;
}

void apply (float* tab, int n, fun_t f){
    for(int i = 0; i < n ; ++i)
        tab [ i ] = (* f ) ( tab [ i ] );
}

int main (){
    float* x = (float *)calloc (10 , sizeof(float));
    apply(x, 10, add1);
    apply(x , 10, sub2);
    free(x) ;
    return 0;
}
```





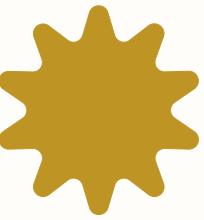
# FONCTEURS

```
struct Add1{
    float operator()(float x) const {
        return x + 1;
    }
};

struct Sub2 {
    float operator()(float x) const {
        return x - 2;
    }
};

int main() {
    Sub2 sub;
    Add1 add;
    float v = sub(2); // v = 0
    float y = add(2); // y = 3
    return 0;
}
```

En C++, on fait des **foncteurs(functors)** - des objets appelables comme des fonctions avec un opérateur()



# FONCTEURS

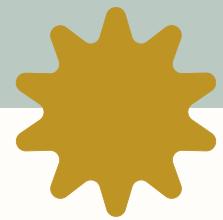
```
class Functor {
public :
    virtual float operator()(float x) const = 0;
};

class Add1 : public Functor {
public :
    float operator()(float x) const override {
        return x + 1;
    }
};

void apply(float* tab, int n, const Functor & f){
    for(int i = 0; i < n; ++i)
        tab[i] = f(tab[i]);
}

int main(){
    float x[] = {1, 2, 3, 4};
    apply(x, 4, Add1());
    return 0;
}
```

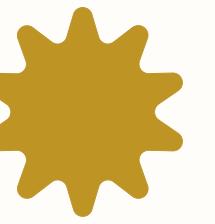




## Un peu de pratique

- Ajoutez une fonction template appliquerFoncteur qui utilise un foncteur pour appliquer une transformation à chaque élément d'une Pair. Par exemple, un foncteur pourrait augmenter ou réduire les dimensions des formes de la paire



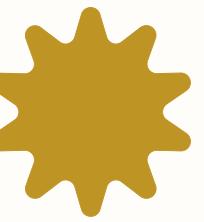


## CAST

Il existe plusieurs méthodes pour “caster” une variable:

- `dynamic_cast`
- `static_cast`
- `const_cast`
- `reinterpret_cast` (à éviter)





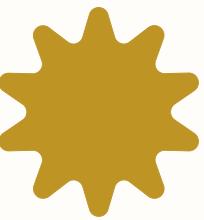
## CAST

B dérive de A. C est une autre classe.

- **dynamic\_cast** convertit un pointeur ou une référence d'un type à un autre en vérifiant que la conversion est valide.
  - Si la conversion est invalide, renvoie NULL (pour un pointeur) ou lance une exception (pour une réf.)
  - Vérification du type des objets à l'exécution → lent !

```
A* a = new A();
B* b = new B();
A* b_dcast_a = dynamic_cast<A*>(b);
// -> ok (*b est bien un A)
C* b_dcast_c = dynamic_cast<C*>(b);
// classe C non liée -> b_dcast_c == NULL
B* a_dcast_b = dynamic_cast<B*>(a);
// *a n'est pas un objet B complet -> a_dcast_b == NULL
B* b_dcast_2 = dynamic_cast<B*>(b_dcast_a);
// -> ok ! (car *b_dcast_a est bien un B !)
```





## CAST

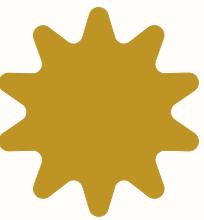
B dérive de A. C est une autre classe.

**static\_cast** ne fait pas de vérification à l'exécution ( $\rightarrow$  rapide)

- Vérification du type des pointeurs/références à la compilation
  - Si la conversion est manifestement impossible (classes non apparentées), ne compile pas
  - Sinon, fonctionne... mais permet aussi de convertir un pointeur ou une réf. d'un type de base vers un type de dérivé (dangereux !)

```
A* a = new A();
B* b = new B();
A* b_scast_1 = static_cast<A*>(b);
// -> ok (*b est bien un A)
C* b_scast_2 = static_cast<C*>(b);
// classe C non liée -> ne compile pas
B* a_scast = static_cast<B*>(a);
/* -> passe... mais attention, *a_scast n'est pas un objet B valide ! */
```





## CAST

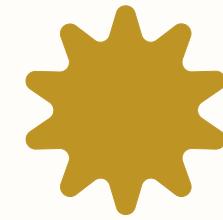
D'autres moins utiles :

- **reinterpret\_cast** permet de faire pratiquement n'importe quoi (convertir un pointeur vers un type complètement différent, etc.), généralement à éviter
- **const\_cast** permet (uniquement) de convertir un pointeur ou une référence const en pointeur ou une référence non-const, ou vice versa.

```
class A { ... };
class B { ... };

void A::methode(const B& bobject)
{
    B& bobject_non_const = const_cast<B&>(bobject);
    /* On peut maintenant appeler les méthodes non-const de
       bobject_non_const (mais est-ce une bonne idée ?) */
}
```





# CONTENEURS EN C++

Quatre types de conteneurs standards:

- **std::vector<T>** - Tableau redimensionnable (liste en python)
- **std::list<T>** - Liste chaînée (pas d'équivalent)
- **std::set<T>** - Ensemble (set en python)
- **std::map<T>** - Table associative (dictionnaire en python)

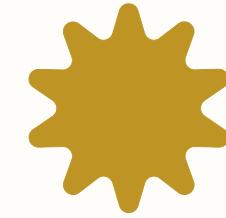
**Pas d'arbre, ni de graphe.**

Opérations typiques:

- Accès à un élément
- Supprimer un élément
- Ajouter un élément
- Rechercher un élément

**Chaque type diffère par les temps d'exécution de chaque opération**





# STD::VECTOR

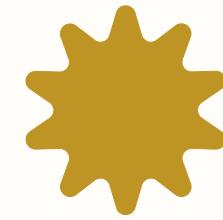
Principales méthodes:

- **v.push\_back(e)** - Ajout d'un élément e **à la fin**
- **v.pop\_back()** - Suppression du **dernier** élément
- **v.clear()** - Suppression de tous les éléments
- **v[i]** - Surcharge de l'opérateur [] qui renvoie l'élément à la case i (référence)
- **v.front()** et **v.back()** - Renvoie respectivement le premier ou le dernier élément (référence)
- **v.size()** - Renvoie le nombre d'éléments

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

int main(){
    std::vector<int> v(10);
    v.push_back(34);
    v[0] = 19;
    for(int& el : v)
    {
        std::cout << v[i] << std::endl;
    }
    return 0;
}
```





# STD::VECTOR

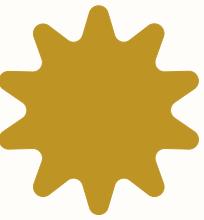
## Caractéristiques

- **Ajout et suppression à la fin** - rapide
- **Suppression au milieu** - lent (décalage de tous les éléments après la suppression)
- **Accès/changement à/d' un élément à un index donné (accès aléatoire)** - rapide

## A ne pas utiliser:

- quand on doit supprimer beaucoup d'élément avec des accès aléatoires (tous sauf à la fin)





# STD::LIST

On doit les parcourir dans l'ordre  
o Concept des itérateurs

```
#include <list>
#include <iostream>

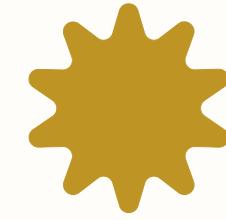
int main() {
    std::list<int> my_list;
    my_list.push_back(42);
    my_list.push_back(4242);

    for (std::list<int>::const_iterator it = my_list.begin(); it != my_list.end(); ++it)
        std::cout << *it << std::endl;

    std::list<int>::iterator it = my_list.begin();
    *it = 2; // changement du premier élément

    return 0;
}
```



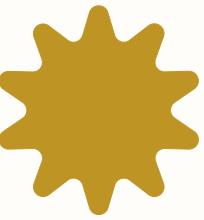


# STD::LIST

## Méthodes pour les itérateurs

- **std::list<T>::iterator** - type de l'itérateur
- **std::list<T>::const\_iterator** - type de l'itérateur (mais en const)
- **++it** - itérateur suivant
- **--it** - itérateur précédent
- **it->méthode()** - méthode de l'objet pointé par it
- **my\_list.begin()** - itérateur sur le premier élément
- **my\_list.end()** - itérateur sur la fin de la liste (**après le dernier élément**)
- **my\_list.insert(iterator pos, const T& t)** - insert t avant pos





## STD::LIST

Le parcours avec des itérateurs fonctionne également avec des vecteurs

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

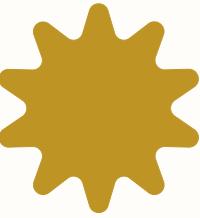
int main() {
    std::list<int> my_list;
    my_list.push_back(42);
    my_list.push_back(4242);

    for (std::list<int>::const_iterator it = my_list.begin(); it != my_list.end(); ++it)
        std::cout << *it << std::endl;

    std::list<int>::iterator it = my_list.begin();
    *it = 2; // changement du premier élément

    return 0;
}
```





# STD::LIST

On peut donc faire des algorithmes génériques

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

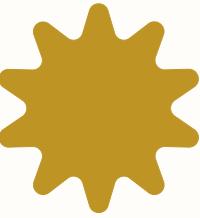
template<typename T>
void print(const T& my_list) {
    for (typename T::const_iterator it = my_list.begin(); it != my_list.end(); ++it)
        std::cout << *it << std::endl;
}

int main() {
    std::list<int> my_list;
    print(my_list);

    std::vector<int> my_vec;
    print(my_vec);

    return 0;
}
```





# STD::LIST

Ou encore

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

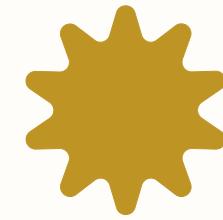
template<typename T>
void print(const T& i1, const T& i2) {
    for (T it = i1; it != i2; ++it)
        std::cout << *it << std::endl;
}

int main() {
    std::list<int> my_list;
    print(my_list.begin(), my_list.end());

    std::vector<int> my_vec;
    print(my_vec.begin(), my_vec.end());

    return 0;
}
```

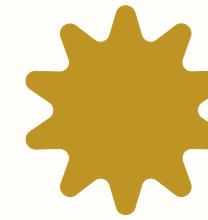




# CONCLUSION

	<b>std::list</b>	<b>std::vector</b>
Ajout à la fin	$O(1)$	$O(1)$
Ajout au début	$O(1)$	$O(n)$
Accès aléatoire	$O(n)$	$O(1)$
Suppression à la fin	$O(1)$	$O(1)$
Suppression au début	$O(1)$	$O(n)$
Suppression après un élément	$O(1)$	$O(n)$
Recherche	$O(n)$	$O(n)$
Tri	$O(n \log(n))$	$O(n \log(n))$

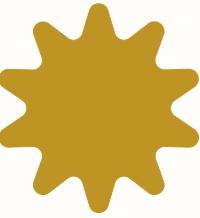




## REMARQUES

- Pas de conteneurs avec des références
- **std::vector<A> a(10)** - le vecteur contient 10 objets A, on appelle 10 fois le constructeur par défaut
  - Le constructeur par défaut doit exister
  - **Pas de polymorphisme** - on ne peut stocker que des objets A et non des objets de classes dérivées
  - gestion assez lourde, par ex. pour remplacer un élément par un autre : utilise **operator=** de A
  - **généralement peu approprié** sauf pour des classes très simples
- **std::vector<A\*> a(10)** - le vector contient 10 pointeurs
  - on construit ces objets quand/comme on veut (avec new)
  - Polymorphisme **OK**
  - copier un élément = copier un pointeur (rapide)
  - mais **attention** aux pointeurs nuls, aux partages d'objets involontaires, à gérer le cycle de vie des objets (new/delete), etc

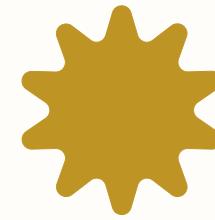




## STD::SET

- On souhaite représenter un ensemble
  - **non ordonné**
  - sans éléments dupliqués
- Opérations qui doivent être rapides
  - Ajout d'un élément
  - Suppression d'un élément
  - **Savoir si un élément appartient à l'ensemble**

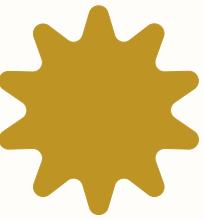




## STD::SET

- Représente l'ensemble par un arbre de recherche  
→ Nécessite qu'un **operator<** soit défini pour T (ou alors une fonction ou un foncteur de comparaison doit être fourni)
- (n = taille de l'ensemble)
- **void insert (const T& t)** - ajout d'un élément → O(log(n))
- **iterator find (const T& t)** - recherche d'un élément → O(log(n)) (renvoie un itérateur sur l'élément ou std::set::end s'il n'est pas trouvé)
- **size\_t size ()** - renvoie le nombre d'éléments





# STD::SET

```
#include <list>
#include <set>
#include <iostream>

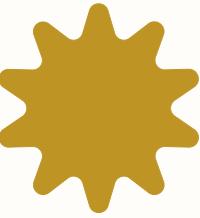
int main() {
    std::set<int> s;
    s.insert(42);
    s.insert(42);
    s.insert(24);

    // Parcours
    for (std::set<int>::const_iterator it = s.begin(); it != s.end(); ++it)
        std::cout << *it << std::endl;

    // Recherche
    std::set<int>::iterator it = s.find(42);
    if (it != s.end())
        std::cout << "trouve la reponse" << std::endl;
    else
        std::cout << "je cherche encore" << std::endl;

    return 0;
}
```



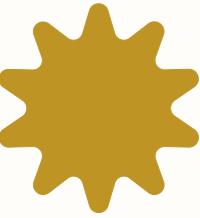


## STD::MAP

### Opérations rapides ( $\log(n)$ )

- Ajout de couple clé/valeur : **ma\_table[key] = val**
- Accès par clé : **ma\_table[key]**
- Savoir si une clé est dans la table : **find** (cf std::set)
- Récupérer et itérer sur l'ensemble des clés
  - itérateurs `it->first`
  - clé `it->second` : élément





# STD::MAP

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

int main() {
    typedef std::map<std::string, int> month_t;
    typedef month_t::iterator month_iterator_t;

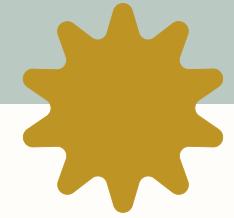
    month_t months;
    months["january"] = 31;
    months["february"] = 28;
    months["march"] = 31;

    std::cout << months["january"] << std::endl;

    month_iterator_t it = months.find("february");
    if (it != months.end())
        std::cout << it->first << " => " << it->second << std::endl;

    return 0;
}
```





## Un peu de pratique - Gestion d'un Inventaire de Magasin

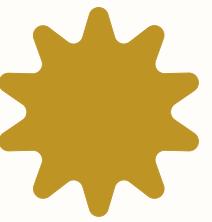
Vous allez implémenter un programme en C++ qui gère l'inventaire d'un magasin en utilisant std::vector, std::set et std::map.

- **Stocker les articles disponibles** : Utilisez un std::set<std::string> pour garder une liste unique des produits disponibles en magasin.
- **Gérer le stock de chaque article** : Utilisez un std::map<std::string, int> pour associer chaque article à sa quantité en stock.
- **Enregistrer les ventes** : Utilisez un std::vector<std::pair<std::string, int>> pour enregistrer les articles vendus et leur quantité.

### Tâches à réaliser

- Ajouter un article au stock (dans std::set et std::map).
- Vendre un article (mettre à jour std::map et ajouter l'entrée dans std::vector).
- Afficher l'état du stock.
- Afficher l'historique des ventes.

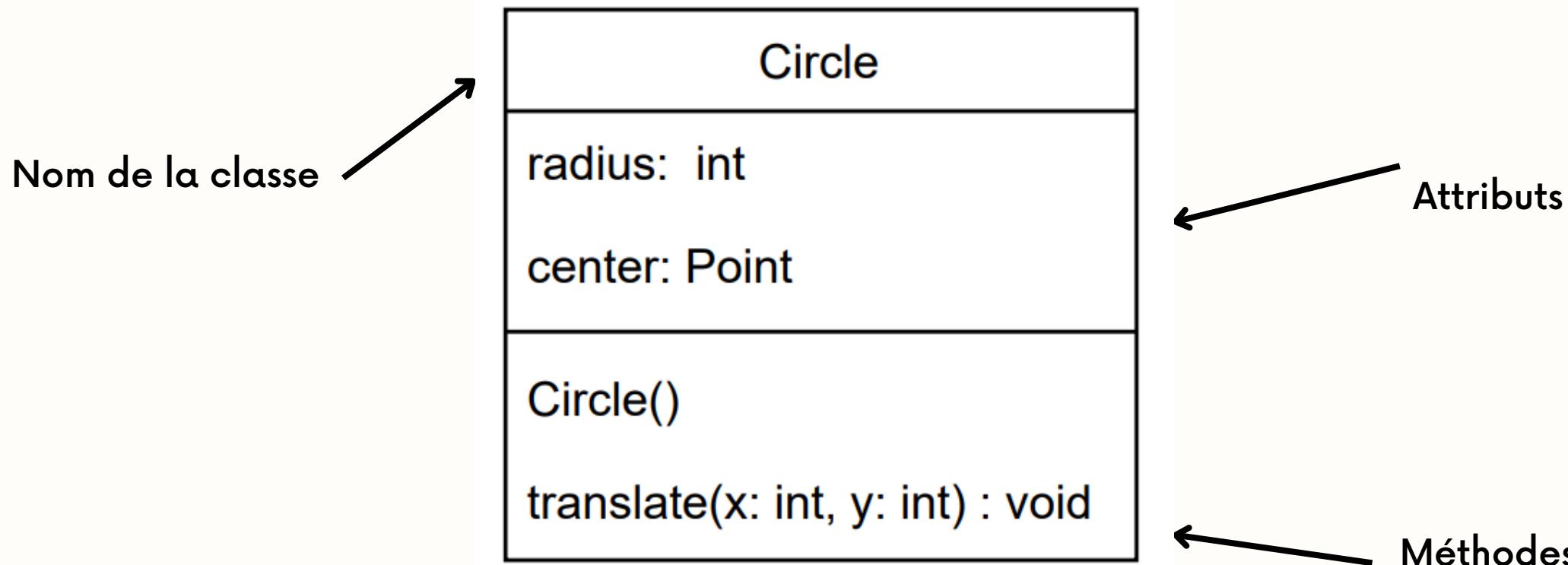


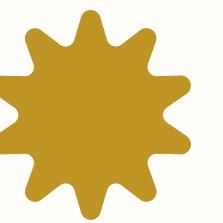


# REPRESENTATION DES CLASSES

**Comment représenter ses programmes ?**

- UML - Unified Modeling Language
- On ne verra ici que les «diagramme de classes»

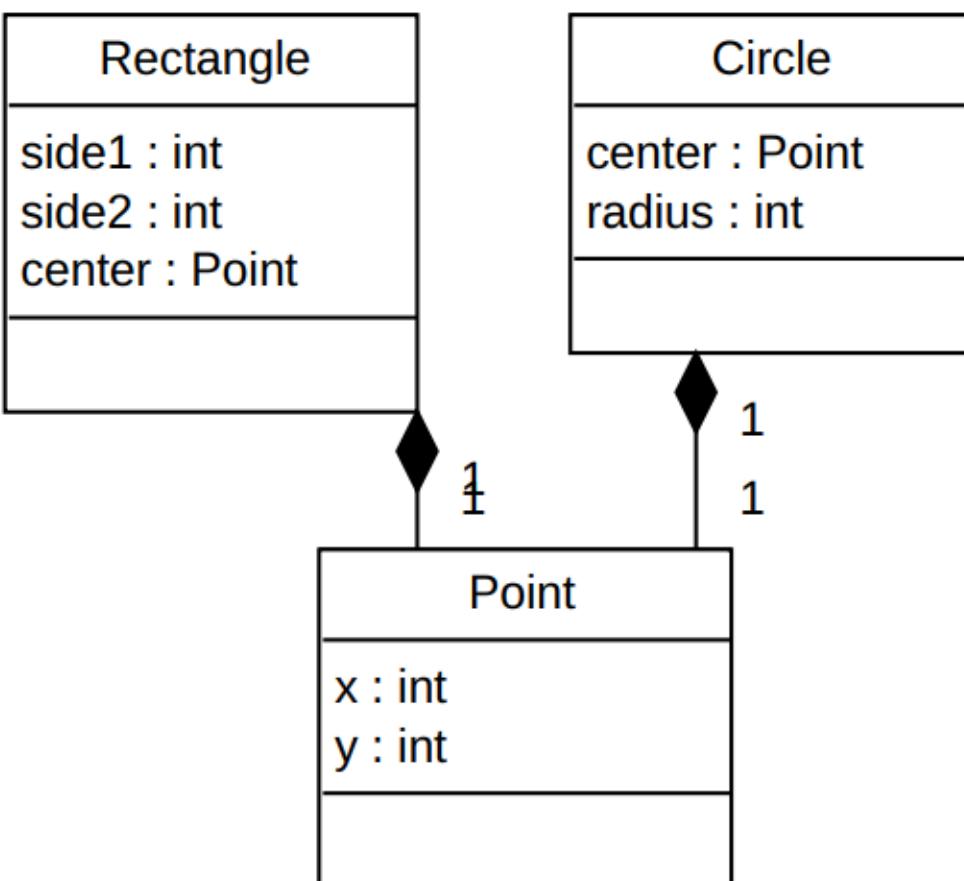


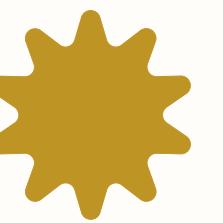


# REPRESENTATION DES CLASSES

## Les relations

- **Composition** – Losange plein

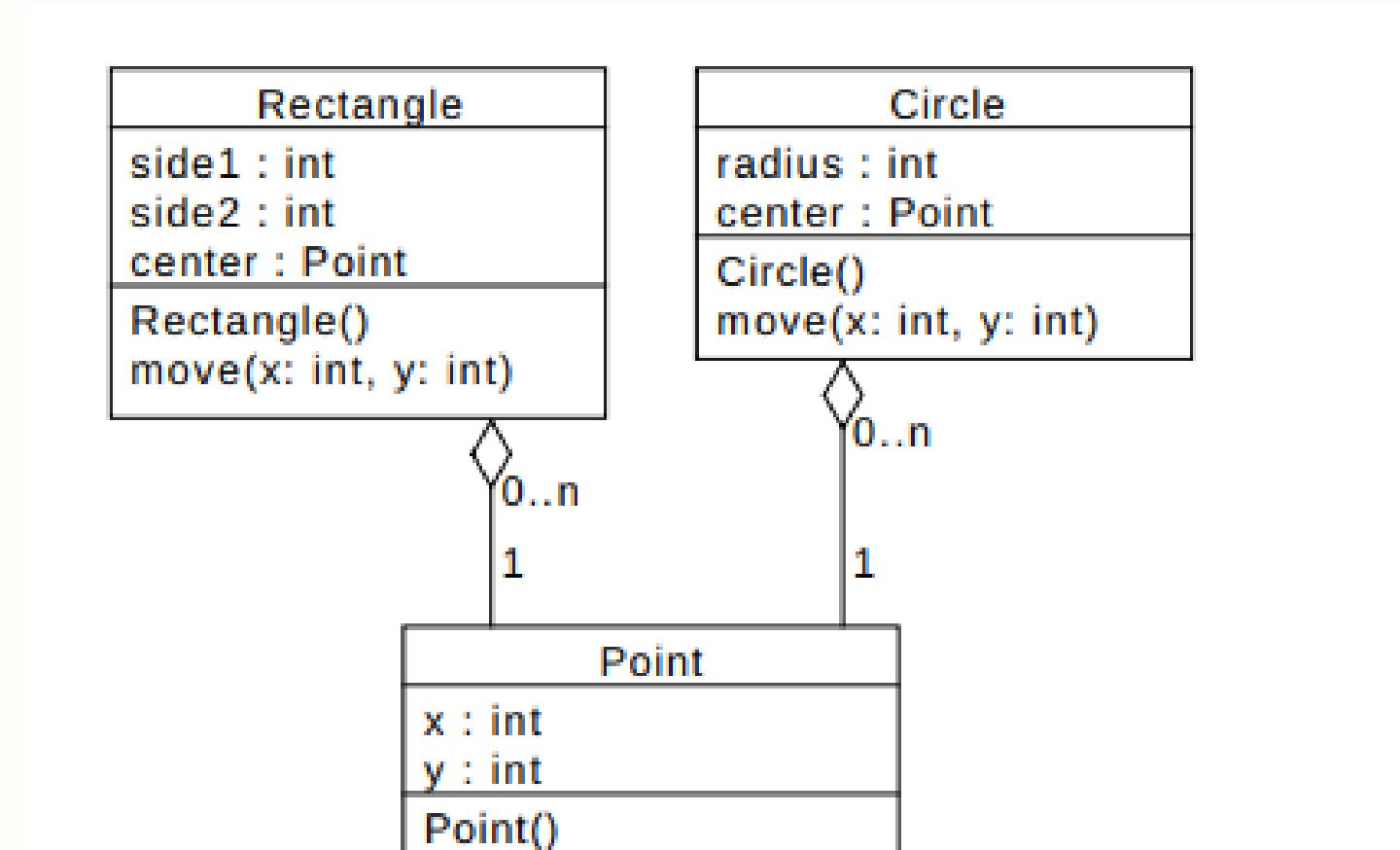


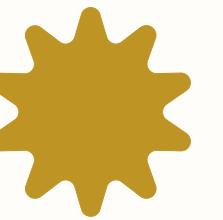


# REPRESENTATION DES CLASSES

## Les relations

- **Agrégation** - Losange vide

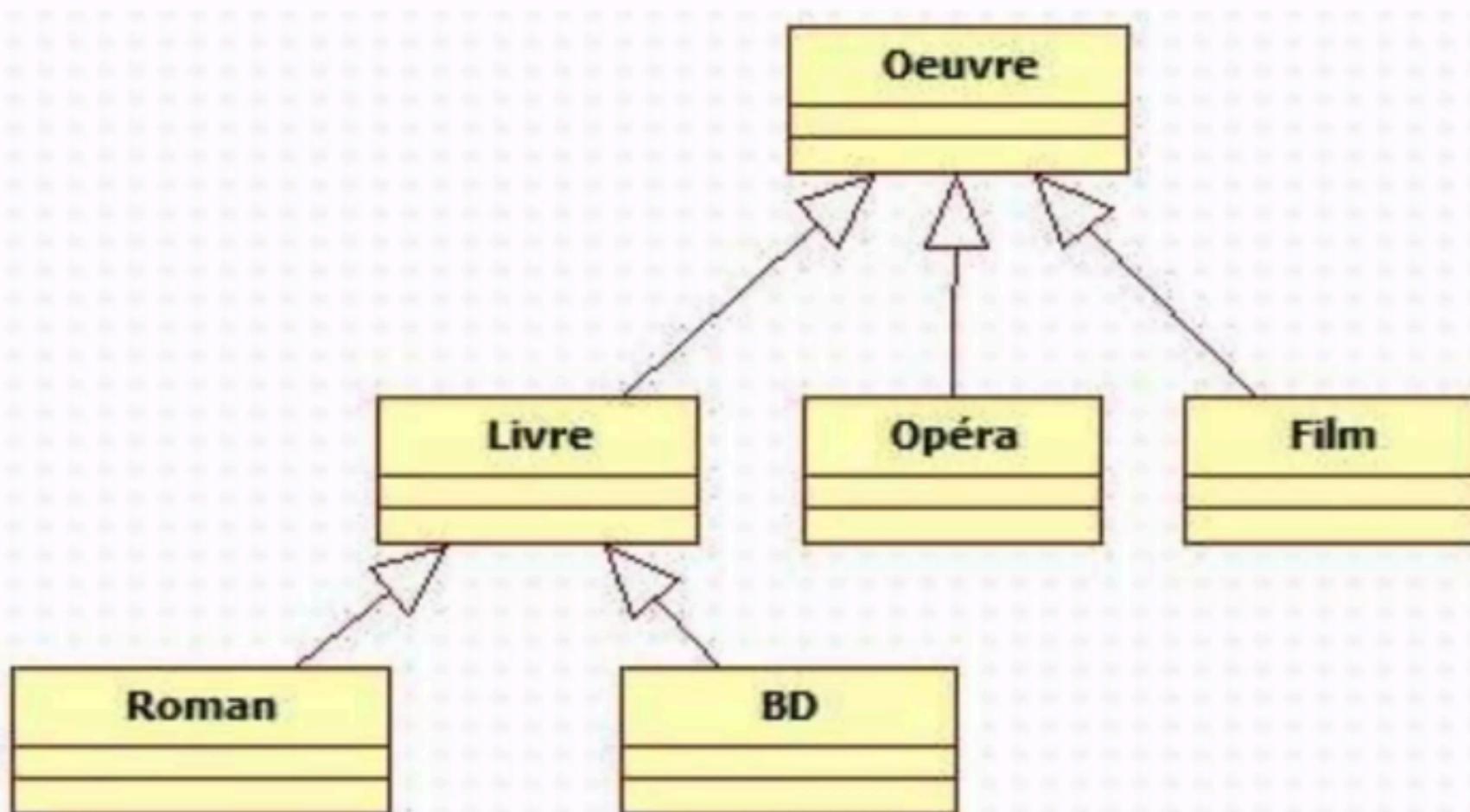


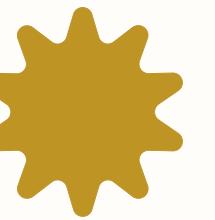


# REPRESENTATION DES CLASSES

## Les relations

- Héritage - Flèche





# REPRESENTATION DES CLASSES

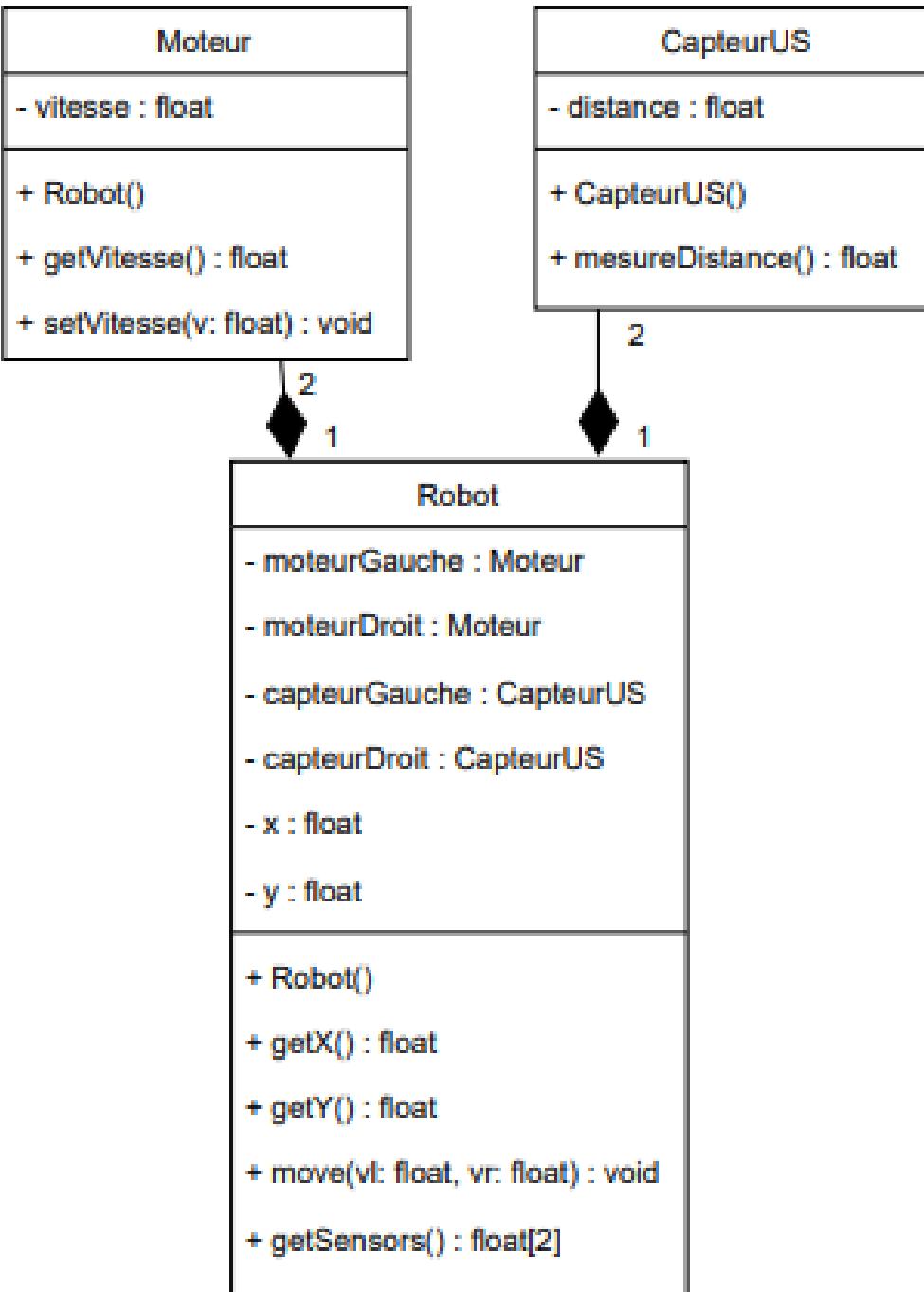
## Attributs et Méthodes

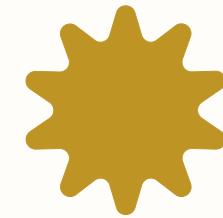
- **Privée** - représenter avec -
- **Public** - représenter avec +
- **Protected** - représenter avec #

Circle
-radius : int
-area : int
-center : Point
+Circle(r : int,c : Point)
+Circle()
+move(x : int,y : Point)
+getRadius() : int
+getCenter() : Point
+setRadius(x : int)
+newOperation()



# Un peu de pratique





# MÉTA PROGRAMMATION

## Rappel

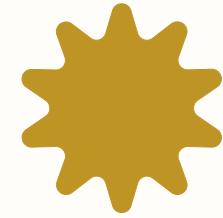
- Les templates permettent d'écrire du code générique, indépendant du type utilisé.
- Génération automatique de code par le compilateur à partir d'un seul modèle.

```
template <class T>
const T& Max(const T& x, const T& y) {
    return x > y ? x : y;
}

int i = Max(5, 9);
double d = Max(1.5, 8.6);
std::string s = Max(std::string("bonjour"), std::string("hello"));
```



Le compilateur génère les versions spécifiques pour **int**, **double**, **std::string** automatiquement.



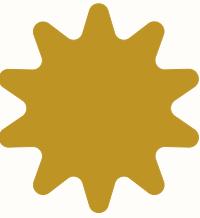
# MÉTA PROGRAMMATION

- **Métaprogrammation** = Exécution de code à la compilation.
- Permet d'optimiser les calculs et d'automatiser la génération de code.
- **Avantages** : Réduction du code écrit, meilleure performance d'exécution
- **Inconvénients** : Temps de compilation plus long.

## Applications

- ✓ Calculs mathématiques optimisés (factorielle, sinus...).
- ✓ Optimisation d'algorithmes classiques (ex: tri-bulle).
- ✓ Génération automatique de code (design patterns : Fabrique, Visiteur).
- ✓ Analyse syntaxique et transformations de code.





# MÉTA PROGRAMMATION

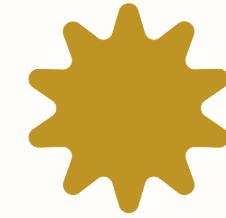
## Exemple

```
template<unsigned int N> struct Fact
{
    enum {Value = N * Fact<N - 1>::Value};
};

template<> struct Fact<0>
{
    enum {Value = 1};
};

// Ici, x vaudra 24 avant même que vous ne lanciez votre programme - coût à l'exécution : 0 sesterce
unsigned int x = Fact<4>::Value;
```





# MÉTA PROGRAMMATION

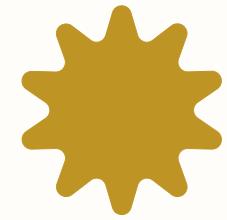
## Quelques règles

- **Problème** : Les fonctions classiques sont évaluées à l'exécution et non à la compilation.
- **Solution** : Remplacer les fonctions par des structures (struct) et utiliser des enum pour stocker les valeurs calculées.s

```
// Fonction Classique
int Identite(int N) {
    return N;
}
unsigned int x = Identite(5); // x connu seulement à l'exécution
```

```
// Fonction en métaprogramming
template<int N> struct Identite {
    enum { Value = N };
};
unsigned int x = Identite<5>::Value; // x connu du compilateur
```





# MÉTA PROGRAMMATION

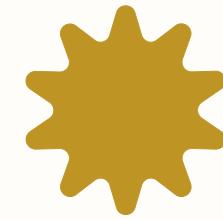
## Les conditions : spécialisation et opérateur ternaire

- If/else classique - évalué à l'exécution.

```
template<bool Condition> struct Test {};
template<> struct Test<true>
{
    static void Do()
    {
        DoSomething();
    }
};
template<> struct Test<false>
{
    static void Do()
    {
        DoSomethingElse();
    }
};

/* Code habituel
if (Condition)
    DoSomething();
else
    DoSomethingElse();
*/
// Avec notre structure Test
Test<Condition>::Do();
```





# MÉTA PROGRAMMATION

## Pourquoi utiliser le CRTP ?

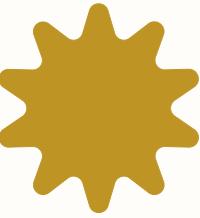
1. Éviter le coût des fonctions virtuelles (virtual)
2. Permettre l'optimisation du code par le compilateur → Meilleure inlining et élimination du code mort.
3. Générer du code spécifique à chaque classe dérivée tout en gardant un code générique dans la classe de base.
4. Factoriser du code sans perdre en performance

```
//Approche Classique
class Base {
public:
    virtual void implementation() = 0; // Fonction virtuelle pure
    void interface() { implementation(); }
    virtual ~Base() = default;
};

class Derived : public Base {
public:
    void implementation() override {
        std::cout << "Implémentation de Derived" << std::endl;
    }
};

int main() {
    Derived d;
    d.interface(); // Appelle implementation() de Derived
}
```





# POLYMPORPHISME STATIQUE

```
template <typename Derived>
class Shape {
public:
    void draw() const {
        static_cast<const Derived*>(this)->draw_impl();
    }
};

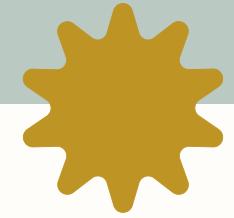
class Circle : public Shape<Circle> {
public:
    void draw_impl() const {
        std::cout << "Dessine un cercle" << std::endl;
    }
};

class Square : public Shape<Square> {
public:
    void draw_impl() const {
        std::cout << "Dessine un carré" << std::endl;
    }
};

int main() {
    Circle c;
    Square s;

    c.draw(); // Affiche "Dessine un cercle"
    s.draw(); // Affiche "Dessine un carré"
}
```

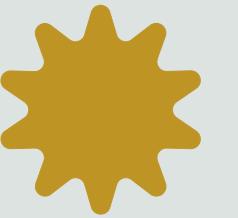




## Un peu de pratique

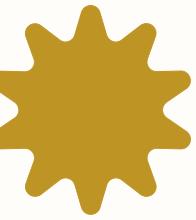
- Refaire les classes Forme, Rectangle et Cercle en utilisant le CRTP





# INTRODUCTION À CMAKE ET CTEST

---



## CONTEXTE

### Qu'est-ce que c'est ?

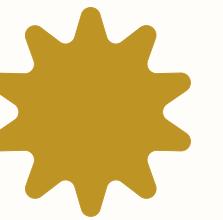
CMake, CTest, CPack, CDash sont des outils opensource

- développés par Kitware (VTK - Visualisation ToolKit) depuis 2001
- **CMake** - Gestion de la compilation
- **CTest** - Tests unitaires
- **CPack** - Construction de packages
- **CDash** - Intégration Continue

### Adopter une méthodologie d'intégration continue

- Vise à l'automatisation des tâches (compilation, tests unitaires et fonctionnels, tests de performances, validation, documentation ...)





## CONTEXTE

Pour faire quoi ?

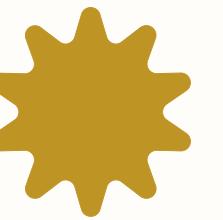
### CMake

- Aide à générer le Makefile pour différentes plateformes
  - Compilation
- Permet de localiser les dépendances
  - Recherche de bibliothèque
  - Facilite le portage ou la gestion de différents compilateur

### CTest/Boost

- Tests unitaires
- Peut s'utiliser sans CMake





## EXEMPLE

Programme simple, sans dépendances.

```
#include <iostream>

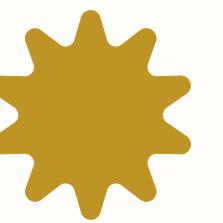
int main()
{
    std::cout << "Hello World !" << std::endl;
}
```

Structure

```
project/
| -- src/
|   -- programme.cpp
```

**Objectif** - compiler le code, l'installer et le tester





## EXAMPLE

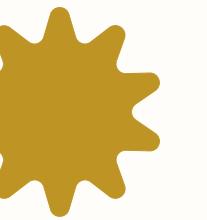
**Compilation manuelle, simple**

```
g++ prog.cpp
```

**Génération d'un exécutable**

```
cpp@cpp:~/project/src$ ./a.out  
' Hello , world !'
```





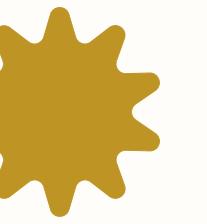
# EXAMPLE

## Création manuelle d'un Makefile

```
project/  
| -- src/  
|   | -- prog.cpp  
|   | -- Makefile  
| -- bin/
```

```
cpp@cpp:~/project/src$ make -n  
g++ -c prog.cpp  
g++ -o a.out prog.o  
cpp@cpp:~/project/src$ make  
cpp@cpp:~/project/src$ make install  
mv a.out ../bin/.  
cpp@cpp:~/project/src$ cd ../bin  
cpp@cpp:~/project/bin$ ./a.out
```





# EXEMPLE

## Exemple de Makefile

```
CC = g++
OBJS = prog.o
.SUFFIXES: .o .cpp

.cpp.o:
    $(CC) -c $(CFLAGS) $<

EXE = a.out

all: $(EXE)

$(EXE): $(OBJS)
    $(CC) $(LDFLAGS) -o $@ $(OBJS)

clean:
    rm -f $(OBJS) $(EXE)

install:
    mv $(EXE) ../bin/.
```



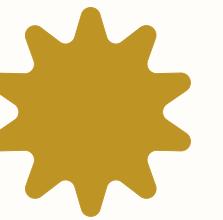


## EXAMPLE

### Construction en utilisant CMake

```
project/  
| -- src/  
|   | -- prog.cpp  
|   | -- CMakeLists.txt  
| -- bin/  
| -- CMakeLists.txt
```





# EXAMPLE

## CMakeLists.txt à la racine

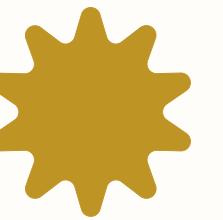
```
# CMake version
cmake_minimum_required(VERSION 2.8.10 FATAL_ERROR)

# Define project name and language
project(MyProject CXX)

# Define executable names
set(EXE MyProgram)

# Define directories
set(SRC ${CMAKE_SOURCE_DIR}/src)
add_subdirectory(${SRC})
```





# EXEMPLE

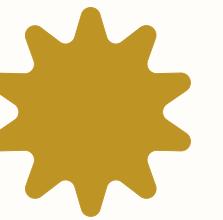
## CMakeLists.txt associé aux sources

```
# Set up the source files compilation
# Add the source files
set(SRC
    prog.cpp
)

# Define the executable in terms of the source files
add_executable(${EXE} ${SRC})

# How to install this executable
install(TARGETS ${EXE} RUNTIME DESTINATION bin)
```



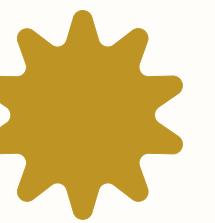


# EXEMPLE

## Utilisation de cmake

```
cpp@cpp:~/project$ mkdir build && cd build
cpp@cpp:~/project/build$ ccmake ..
cpp@cpp:~/project/build$ make
Scanning dependencies of target a.out
[100%] Building CXX object src/CMakeFiles/a.out.dir/prog.o
Linking CXX executable a.out
[100%] Built target a.out
cpp@cpp:~/project/build$ make install
Install the project ...
-- Install configuration: "Release"
-- Installing: ~/project/bin/a.out
```





# MAKEFILE VS CMAKE

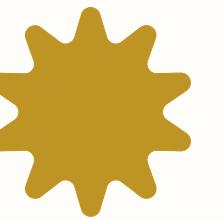
## Makefile

```
main.o: main.c main.h  
    cc -c main.c  
  
MyProgram: main.o  
    cc -o MyProgram main.o -lm -lz
```

## CMakeLists.txt

```
project(MyProject C)  
  
add_executable(MyProgram main.c)  
  
target_link_libraries(MyProgram z m)
```





## CMAKE

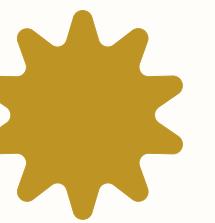
**Pourquoi ne pas se contenter d'appeler le compilateur ? (g++)**

Dans les projets informatiques, la construction de exécutable est une tâche souvent complexe qui dépend

- d'un grand nombre de fichiers
- de nombreuses dépendances
- du cas à traiter (compilation conditionnelle, ...)
- de nombreux éléments externes (bibliothèques, ...)
- et de la manière dont ils sont installés
- de la plateforme
- de l'environnement sur le plateforme

**La plupart du temps, on ne contrôle pas ces dépendances.**





# EXEMPLE AVEC LIEN

**Code (Même structure du projet que précédemment)**

```
#include <Eigen/Dense>
#include <iostream>

int main() {
    // Créer une matrice 2x2
    Eigen::Matrix2d mat;
    mat << 1, 2,
          3, 4;

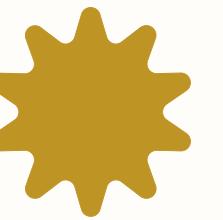
    // Afficher la matrice
    std::cout << "La matrice est :\n" << mat << std::endl;

    // Calculer l'inverse de la matrice
    Eigen::Matrix2d mat_inv = mat.inverse();

    // Afficher l'inverse
    std::cout << "L'inverse de la matrice est :\n" << mat_inv << std::endl;

    return 0;
}
```





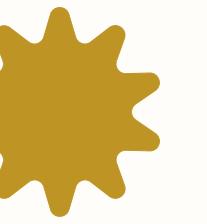
## EXEMPLE AVEC LIEN

### Compilation manuelle

```
g++ -I /path/to/eigen main.cpp -o MyProgram
```

**Problème** - Il faut parfois chercher le chemin de ses dépendances





# EXEMPLE AVEC LIEN

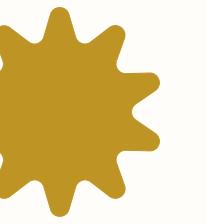
## Makefile

```
# Nom de l'exécutable
EXE = MyProgram
# Chemin vers le répertoire contenant Eigen
EIGEN_DIR = /path/to/eigen
# Fichiers source
SRC = src/main.cpp

# Flags de compilation
CXXFLAGS = -I$(EIGEN_DIR)

# Cible par défaut
all: $(EXE)
# Règle de compilation
$(EXE): $(SRC)
    $(CXX) $(CXXFLAGS) $(SRC) -o $(EXE)
# Règle de nettoyage
clean:
    rm -f $(EXE)
# Règle d'installation (si besoin)
install: $(EXE)
    mv $(EXE) ../bin/
```





# EXEMPLE AVEC LIEN

## CMakeLists.txt à la racine

```
# cmake version
cmake_minimum_required(VERSION 2.8.10 FATAL_ERROR)

# Define project name and language
project(MyProject CXX)

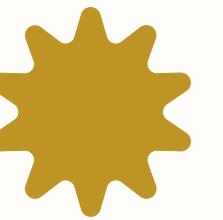
# Find Eigen3
find_package(Eigen3 REQUIRED)

# Define executable names
set(EXE MyProgram)

# Define directories
set(SRC ${CMAKE_SOURCE_DIR}/src)

# Add subdirectory for source files
add_subdirectory(${SRC})
```





# EXEMPLE AVEC LIEN

## CMakeLists.txt sources

```
# Set up the source files compilation
# Add the source files
set(SRC
    prog.cpp
)

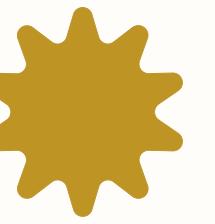
# Define the executable in terms of the source files
add_executable(${EXE} ${SRC})

# Add the needed libraries (Eigen in this case)
target_link_libraries(${EXE} Eigen3::Eigen)

# Include Eigen3 directories
include_directories(${EIGEN3_INCLUDE_DIR})

# How to install this executable
install(TARGETS ${EXE} RUNTIME DESTINATION bin)
```





# EXEMPLE AVEC LIEN

## CMakeLists.txt sources

```
# Set up the source files compilation
# Add the source files
set(SRC
    prog.cpp
)

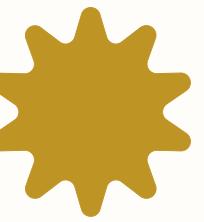
# Define the executable in terms of the source files
add_executable(${EXE} ${SRC})

# Add the needed libraries (Eigen in this case)
target_link_libraries(${EXE} Eigen3::Eigen)

# Include Eigen3 directories
include_directories(${EIGEN3_INCLUDE_DIR})

# How to install this executable
install(TARGETS ${EXE} RUNTIME DESTINATION bin)
```





# CONFIGURATION

## Variables principales

- Type de l'exécutable

```
CMAKE_BUILD_TYPE=[Debug , Release]
```

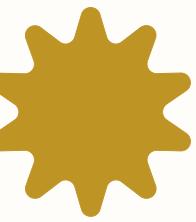
Permet ou non d'exécuter les assert (uniquement en Debug)....

- Répertoire d'installation

```
CMAKE_INSTALL_PREFIX=[/usr/local , C:\Program Files]
```

Impact direct = répertoire où cmake va chercher les bibliothèques externes, et installer les exécutables.





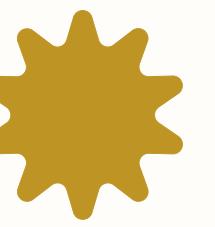
## CTest

**CTest** utilise le même fichier **CMakeLists.txt** (comme les tests définis dans le Makefile)

**CTest** organise la vérification des tests unitaires. Il permet de retourner la valeur 0 lorsque le test est effectué avec succès et renvoie une autre information dans le cas contraire. Le programme testé peut être

- un binaire
- un script
- ...





# CTEST

## Exemple de test utilisant Catch2

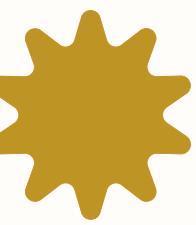
```
#define CATCH_CONFIG_MAIN
#include <catch2/catch.hpp>
#include <Eigen/Dense>

TEST_CASE("Matrix inverse test") {
    Eigen::Matrix2d mat;
    mat << 1, 2,
          3, 4;

    Eigen::Matrix2d mat_inv = mat.inverse();

    REQUIRE(mat_inv * mat == Eigen::Matrix2d::Identity());
}
```



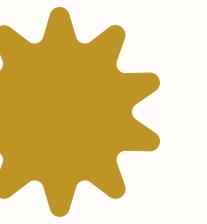


# CTEST

## Structure du répertoire

```
project/
|
|   └── src/                      # Code source principal
|       ├── prog.cpp               # Fichier source principal
|       └── CMakeLists.txt         # CMake pour la compilation du programme
|
|   └── tests/                     # Tests unitaires
|       ├── test_program.cpp     # Test unitaire (exemple avec Catch2)
|       └── CMakeLists.txt       # CMake pour configurer les tests
|
|   └── bin/                       # Dossier où l'exécutable sera installé
|
└── CMakeLists.txt               # CMake principal pour le projet
└── build/                       # Dossier de compilation (généré par CMake)
```





# CTEST

## CMakeLists.txt à la racine

```
# cmake version
cmake_minimum_required(VERSION 2.8.10)

# Define project name and language
project(MyProject CXX)

# Enable testing
enable_testing()

# Find Eigen3
find_package(Eigen3 REQUIRED)

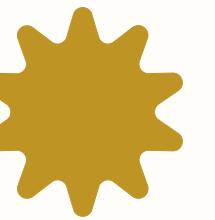
# Define executable names
set(EXE MyProgram)

# Define directories
set(SRC ${CMAKE_SOURCE_DIR}/src)

# Add subdirectory for source files
add_subdirectory(${SRC})

# Optionally, add tests here
add_subdirectory(tests)
```





# CTEST

## CMakeLists.txt dans tests/

```
# Set up the test files compilation
set(TEST_SRC
    test_program.cpp
)

# Define the test executable
add_executable(test_program ${TEST_SRC})

# Link Eigen3 and the main program to the test
target_link_libraries(test_program Eigen3::Eigen ${EXE})

# Add the test to CTest
add_test(NAME MyTest COMMAND test_program)
```

**Lancer les tests** - ctest (dans le répertoire build)

