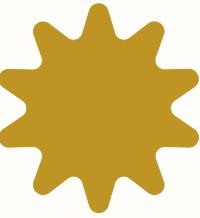


**POLYTECH SORBONNE**

# **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++

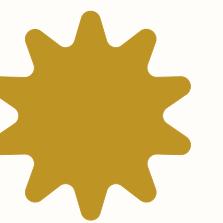
## Déroulement

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

## Évaluation

- 90% Projet et 10% TP



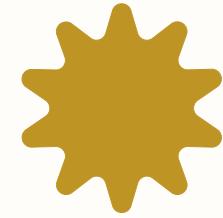


## PROGRAMME & CONTENU

Programmation objet  
en C++

C++ et compilation  
avec CMake

Concept de C++  
avancé



# PRÉSENTATION DU PROJET

## Projet

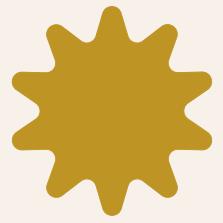
- Faire un jeu vidéo en utilisant vos connaissances en programmation objet et en C++

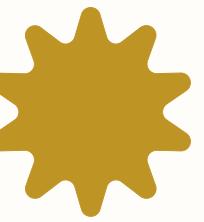
## Évaluation du projet

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



# RAPPEL PROGRAMMATION OBJET

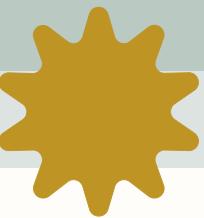




## RAPPEL

- **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



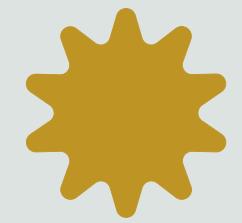


## RÉSUMÉ

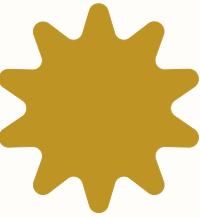
---

### **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++**

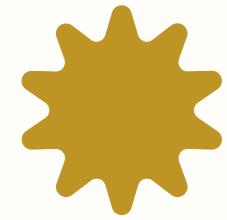


**C++**

## **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++**

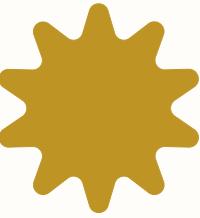
## **Quand l'utiliser ?**

- 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

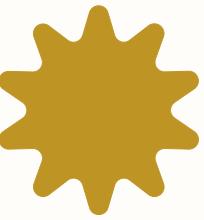
```
# include <iostream>

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

Pour compiler :

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

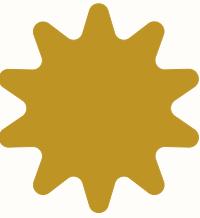




# SYNTAXE DE BASE

```
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 ;  
}
```





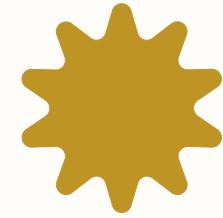
# SYNTAXE DE BASE

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;  
}
```



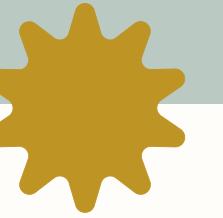


## SYNTAXE DE BASE

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





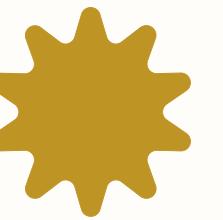
## Un peu de pratique

Créez une classe Rectangle avec :

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

Compilez et testez votre classe



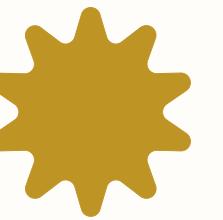


# CONSTRUCTEURS ET DESTRUCTEURS

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 ;
};
```



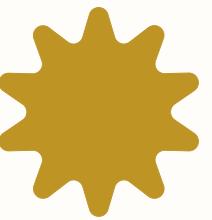


# CONSTRUCTEURS ET DESTRUCTEURS

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 ;
};
```



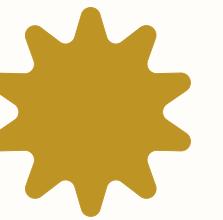


# CONSTRUCTEURS ET DESTRUCTEURS

```
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;
};
```



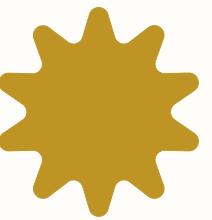


## CONSTRUCTEUR PAR COPIE

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

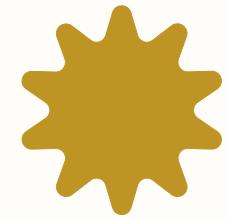
Le destructeur est une méthode automatiquement appelée quand un objet est détruit.

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).





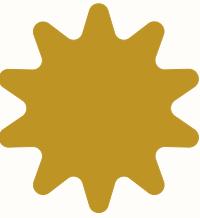
# OPÉRATEURS

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

## Prototype

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



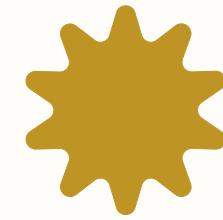


# HÉRITAGE

```
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

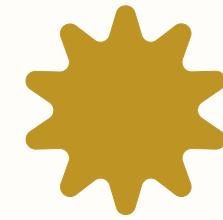
- Si une classe **B** dérive d'une classe **A**, alors toute fonction ayant pour argument une instance de **A** peut être appelée avec une instance de la classe **B**

```
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 !

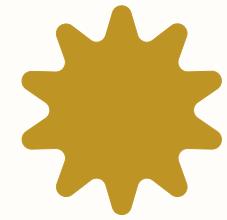
```
A a = new A();
B b = new B();

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

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

- Ne fonctionne pas avec un passage par valeur !





# OPÉRATEURS

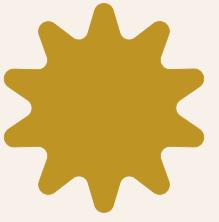
Comme en python, on peut redéfinir tous les opérateurs (`=, ==, !=, ||, +, -, += ...`)  
**Chaque opérateur sera une méthode de la classe.**

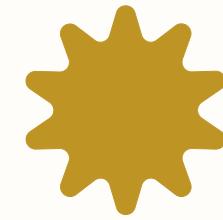
## Prototype

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



# **SPÉCIFICITÉS DU C++**





# PASSAGE PAR RÉFÉRENCE

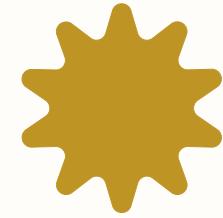
- Une référence est un “alias” vers le même espace mémoire qu’une autre variable, mais avec un autre nom
  - Les deux variables ont **la même adresse mémoire**

```
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
```





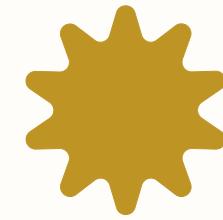
# PASSAGE PAR RÉFÉRENCE

- 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 ;
}
```

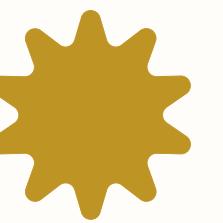




# PASSAGE PAR RÉFÉRENCE

- 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

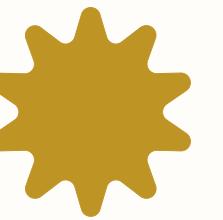
- Fonctionnalité qui permet d'organiser le code en regroupant des classes, fonctions, constantes ou variables sous un même nom logique.
- Eviter les conflits de noms lorsque des projets ou des bibliothèques utilisent des noms identiques.

```
#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>

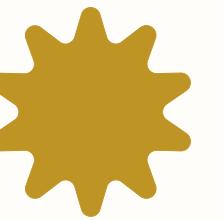
namespace MonEspaceDeNoms {
    int x = 42;
    void afficher() {
        std::cout << "Valeur de x : " << x << std::endl;
    }
}

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;
}
```





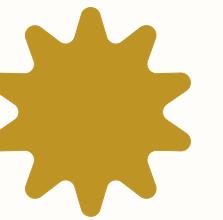
# NAMESPACE

- **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

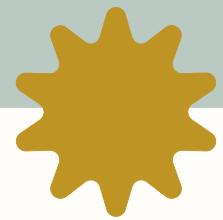
Les **namespaces** sont largement utilisés dans les bibliothèques standard de C++, comme le namespace **std** :

```
#include <iostream>

using namespace std;

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



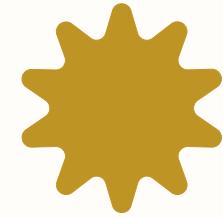


## Un peu de pratique

- 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.





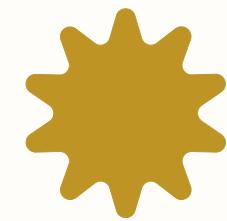
# CONST

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`
```

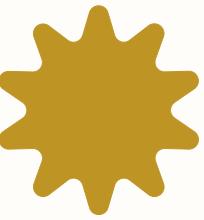
## Pointeurs Constants

```
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
```



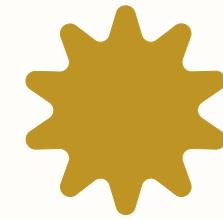


# CONST

## 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

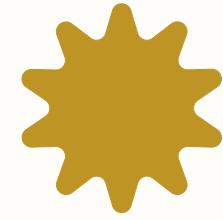
Le mot-clé **constexpr** en C++ est une extension de **const** qui permet de garantir qu'une expression peut être évaluée à la compilation.

## 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
```



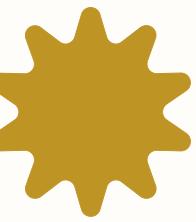


# CONSTEXPR

## 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.



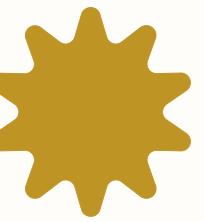


# STATIC

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

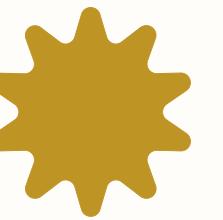
## Variables locales static

- 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.





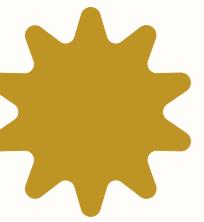
# STATIC

## 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
```

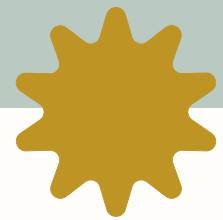




## STATIC VS NON STATIC

<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

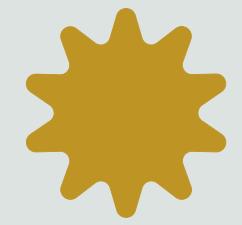




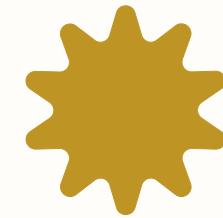
## Un peu de pratique

- 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



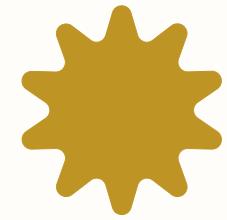
# CLASSE ABSTRAITE

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;
```



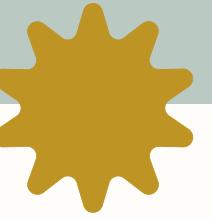


# CLASSE ABSTRAITE

```
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;
    }
};
```

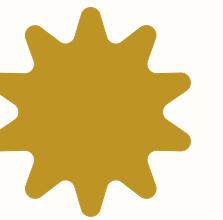




## Un peu de pratique

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





## CODE GENERIQUE

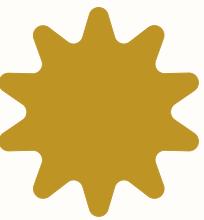
Pour faire du code générique, on peut :

- Utiliser le **polymorphisme**

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

- Duplication de code inutile !



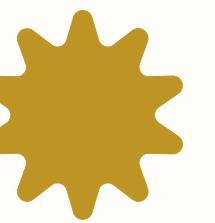


## CODE GENERIQUE

### 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



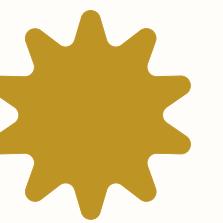


## CODE GENERIQUE

```
// 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 ;
}
```



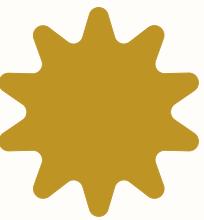


# TEMPLATE

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 ;
}
```



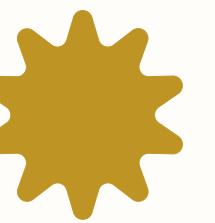


## TEMPLATE

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





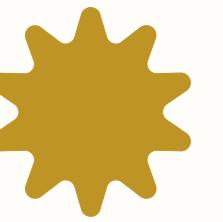
# TEMPLATE

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
}
```





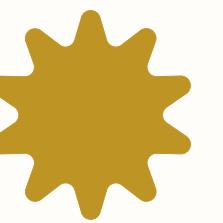
## TEMPLATE

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

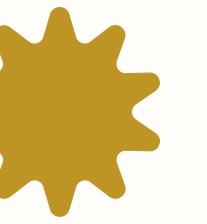
On peut également passer des paramètres en template.

```
# 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;
}
```





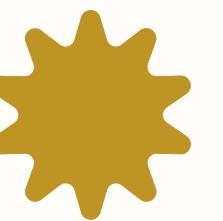
# CLASSE TEMPLATE

```
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

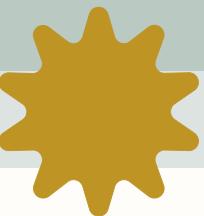
**Typedef** permet de définir un alias pour un type.

```
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();
```

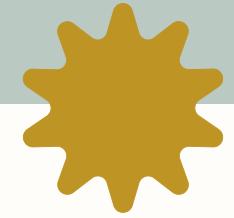




## CONCLUSION

---

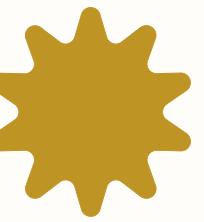
- Un mécanisme très puissant
- Une autre forme d'abstraction
- Exécution rapide (pas de coût pour l'abstraction)
- Mais compilation très lente
  - Très utilisé dans le C++ moderne
  - Mais peut très vite devenir compliqué



## 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.



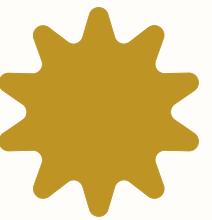


## RAPPEL POINTEUR DE FONCTION

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;

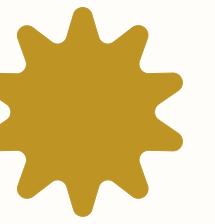
//Liste des fonctions "calculables"

//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};

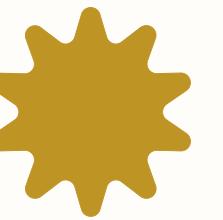
//Liste des fonctions "calculables"

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**.





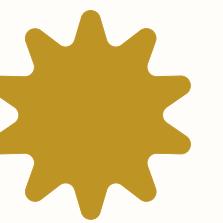
## RAPPEL POINTEUR DE FONCTION

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;

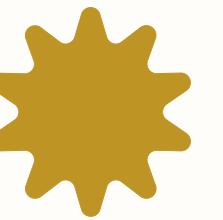
int fonction(double a,string phrase) //Une jolie fonction
{
    //blablabla
}

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

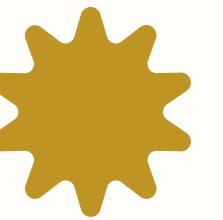
```
class A{
public:
    int fonction(int a);
    //...
};

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 *
}
```



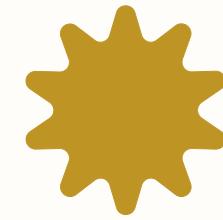
## Un peu de pratique

- 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;
}
```



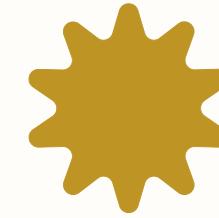
# EXPRESSION LAMBDA

**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**



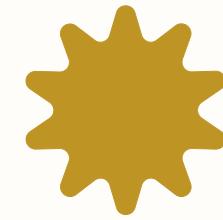


# EXPRESSION LAMBDA

## 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; };

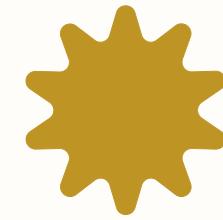
// Capture par référence.
const auto display_ref = [&value]() { std::cout << value << std::endl; };

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 :

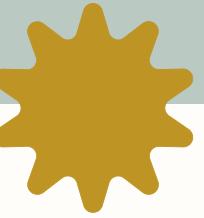
void nom_autogenere_incomprehensible(int v)
{
    std::cout << v << std::endl;
}

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.

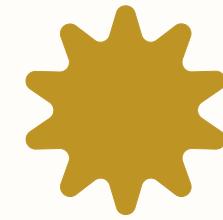
Donc il faut utiliser **auto** ou bien de la wrapper à l'intérieur d'un objet de type **std::function**, si la lambda renvoie une valeur.



## 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);

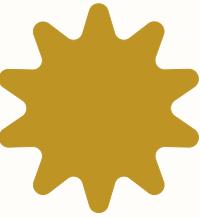
float add1(float x){
    return x + 1;
}

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;
}
```





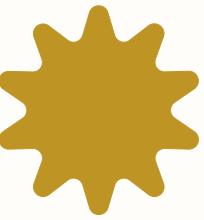
# FONCTEURS

```
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

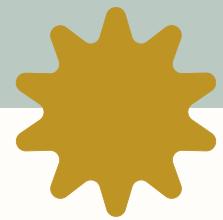
```
class Functor {
public :
    virtual float operator()(float x) const = 0;
};

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;
}
```

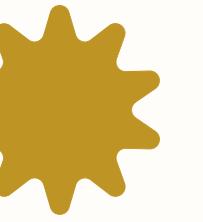




## Un peu de pratique

- 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



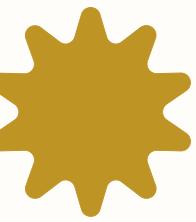


## CAST

Il existe plusieurs méthodes pour “caster” une variable:

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





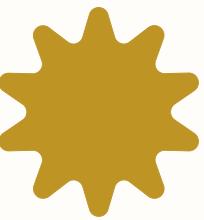
## CAST

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

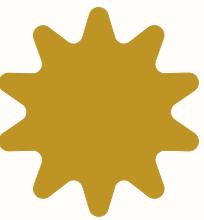
B dérive de A. C est une autre classe.

**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

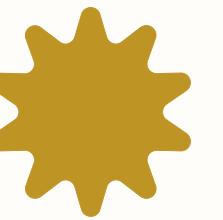
D'autres moins utiles :

- **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 ?) */
}
```



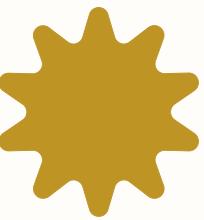


# INTRODUCTION À CMAKE

## Qu'est-ce que CMake ?

- Outil **open-source** pour gérer le processus de compilation d'un projet.
- Génère des fichiers de build adaptés à différents systèmes (équivalent de Makefile pour Make, projet Visual Studio, etc.).
- **CMakeLists.txt** = fichier texte simple pour configurer le projet.



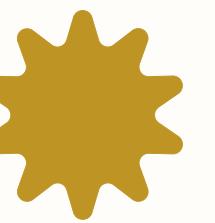


# INTRODUCTION À CMAKE

## Pourquoi utiliser CMake ?

- **Portabilité** : Compatible avec de nombreux systèmes d'exploitation et compilateurs.
- **Modularité** : Permet de gérer des projets complexes avec plusieurs sous-projets.
- **Facilité d'utilisation** : Simplifie les configurations complexes en automatisant les détails technique.



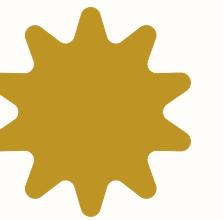


# INTRODUCTION À CMAKE

## Structure d'un projet CMake simple

```
mon_projet/
| -- CMakeLists.txt
| -- src/
|   | -- main.cpp
| -- include/
|   | -- mon_projet.h
```





# CYCLE DE COMPIRATION

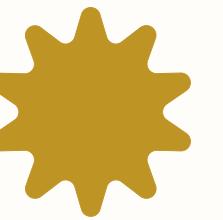
## Création du dossier de compilation

```
mkdir build # ou un autre nom  
cd build
```

## Génération des instructions de compilation & Compilation

```
cmake .. # et options si besoin avec -D  
make -j4 # j4 = 4 threads de compilation
```





# INTRODUCTION À CMAKE

## CMakeLists.txt

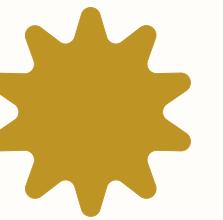
```
cmake_minimum_required(VERSION 3.10)
project(MonProjet)

# Spécifiez le standard C++
set(CMAKE_CXX_STANDARD 17)
set(CMAKE_CXX_STANDARD_REQUIRED True)

# Ajouter l'exécutable
add_executable(mon_executable src/main.cpp)

# Inclure le répertoire include
include_directories(include)
```





# INTRODUCTION À CMAKE

## CMakeLists.txt

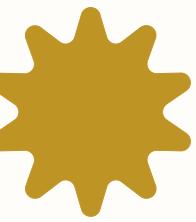
```
cmake_minimum_required(VERSION 3.10)
project(MonProjet)

# Spécifiez le standard C++
set(CMAKE_CXX_STANDARD 17)
set(CMAKE_CXX_STANDARD_REQUIRED True)

# Ajouter l'exécutable
add_executable(mon_executable src/main.cpp)

# Inclure le répertoire include
include_directories(include)
```





# INTRODUCTION À CMAKE

## Commande utiles

### 1. **cmake\_minimum\_required**

- Spécifie la version minimale de CMake requise.

### 2. **project**

- Définit le nom du projet et les langages utilisés.

### 3. **add\_executable**

- Déclare l'exécutable principal du projet.

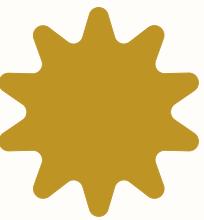
### 4. **include\_directories**

- Ajoute des répertoires pour trouver les fichiers d'en-tête.

### 5. **target\_link\_libraries**

- Lie des bibliothèques externes à l'exécutable.





# INTRODUCTION À CMAKE

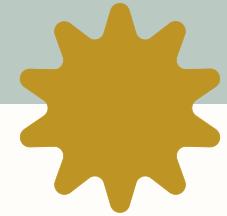
## Ajouter des fichiers multiples

```
add_executable(mon_executable  
    src/main.cpp  
    src/module1.cpp  
    src/module2.cpp  
)
```

## Ajouter une bibliothèque externe

```
add_library(ma_bibliotheque src/lib.cpp)  
target_include_directories(ma_bibliotheque PUBLIC include)  
  
# Lier la bibliothèque à l'exécutable  
add_executable(mon_executable src/main.cpp)  
target_link_libraries(mon_executable ma_bibliotheque)
```





## 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.

