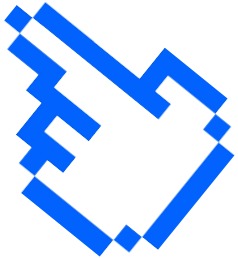
La Plateforme
La grande école du numérique pour tous

POO en C++

Sommaire



/ Spécificités en C++

Syntaxe

Constructeur

Destructeur

```
#pragma once

class MyParentClass
{
public:
    MyParentClass(int a);
    ~MyParentClass();
};

class MyPrinterClass : public MyParentClass
{
public:
    MyPrinterClass(int a);
    ~MyPrinterClass();
    void print();
}
```

Heritage

Constructeur

```
class MyClass
{
  MyClass(int a);
  MyClass(char *str);
}
```

Un constructeur porte le nom de sa classe.
Une classe peut avoir plusieurs constructeurs.

Destructeur

```
class MyClass
{
  MyClass(int a);
  ~MyClass();
}
```

Le destructeur (qui commence par un ~) sert essentiellement à libérer les ressources acquises par une instance.

New / Delete

```
MyClass *instance = new MyClass();
delete instance;
```

Les mots clé new et delete appellent respectivement le constructeur et destructeur.

/ Définition

Définition / Déclaration

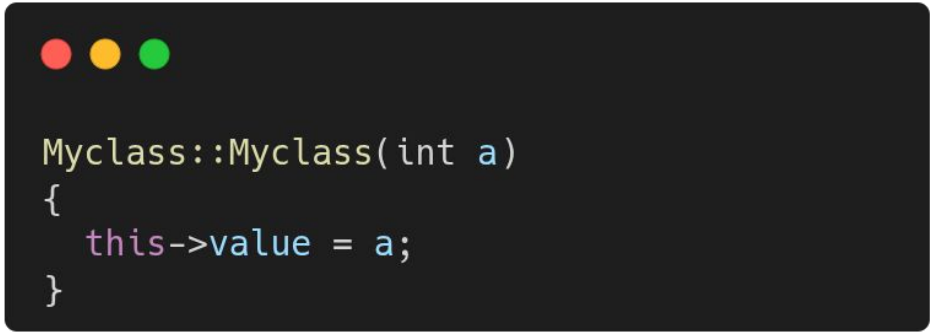
```
class MyClass
{
  MyClass(int a);
  MyClass(char *str);
}
```

La déclaration se fait en general dans un fichier .hpp.

```
MyClass::MyClass(int a)
{
  std::cout << a << std::endl;
}

MyClass::MyClass(char *str)
{
  std::cout << str << std::endl;
}
```

La définition se fait dans un fichier .cpp.

This



```
Myclass::Myclass(int a)
{
  this->value = a;
}
```

Le mot clé this fait référence à l'instance de la classe.

Initialisation par liste d'initialisateurs

```
MyClass::MyClass(int first, int second)
    : value1(first), value2(second)
{
}
```

Les ":" après le constructeur permet d'initialiser des attributs avant même l'entrée dans le constructeur.

Initialisation par liste d’initialisateurs

1. **Performance** (évite une double initialisation)
   - Sans la liste, les membres sont **d'abord initialisés avec leur valeur par défaut**, puis **réassignés** dans le corps du constructeur → double travail inutile.
   - Avec une liste d'initialisation → les valeurs sont directement affectées à la construction.

2. Const et Références
   - Les **membres const** ou les **références (&) doivent obligatoirement** être initialisés dans la liste.
   - Impossible de le faire dans le corps du constructeur.

/ Contrôle d'accès

Public

```
class MyClass
{
  public:
  int value1;
  void print_hello();
}
```

Public permet l'accès aux membres à n'importe quel autre élément.

Private

```
class MyClass
{
private:
  int a;
public:
  int getA();
  void setA();
}
```

Private ne permet l'accès aux membres que à la classe elle même.

Protected

```
class MyClass
{
private:
  int a;
protected:
  int getA();
  void setA();
}

class Child : public MyClass
{
public:
  void printA()
  {
    std::cout << this.getA() << std::endl;
  }
}
```

Protected permet l'accès à la classe elle même, ainsi qu'aux classes qui héritent d'elle.

/ Static

Static

```
class Compteur {
public:
    static int nbObjets;  // Déclaration d'un attribut static

    Compteur() {
        nbObjets++;
    }
};

int Compteur::nbObjets = 0; // Initialisation OBLIGATOIRE en
dehors de la classe

int main() {
    Compteur a;
    Compteur b;
    std::cout << Compteur::nbObjets << std::endl;  // Affiche
2
}

}
```

Un membre statique n'est pas associé à une instance mais à la classe.
Il est accessible depuis n'importe où avec la syntaxe Class::membre, mais jamais avec this.
Il est partagé par tous les objets de la classe.

opérateur de résolution de portée :: pour identifier la classe à laquelle elle appartient.

Les membres statiques doivent impérativement être initialisés.

Quel que soit le nombre d'objets de la classe créés, il n'existe qu'une seule copie du membre statique.

/ Polymorphisme

/ polymorphisme

- “poly” = plusieurs
- “morphisme” = formes.

= Code qui fonctionne de différentes manières selon le type qui l'utilise

technique puissante qui permet d'écrire un code **plus générique, modulaire et facile à maintenir** en POO

Deux types de polymorphisme

le polymorphisme statique (ou compile-time polymorphism)

- réalisé à l'aide de **surcharge de fonctions et de templates.**
- La décision sur quelle fonction ou template utilisé est prise **au moment de la compilation,** en fonction du type statique des arguments passés.

polymorphisme dynamique (ou run-time polymorphism).

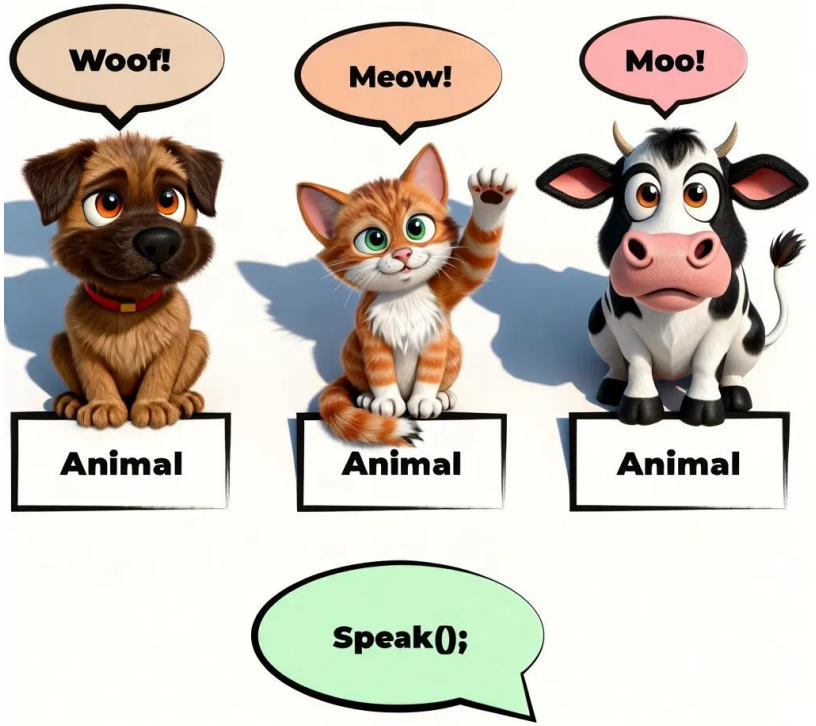
- réalisé à l'aide de classes de base et de classes dérivées, en utilisant des **fonctions virtuelles**
- L'appel de cette fonction est résolu **lors de l'exécution,** en fonction du type réel de l'objet.

Le polymorphisme dynamique, c’est quoi ?

= C'est la capacité d'un objet à prendre plusieurs formes

permet de **"surcharger" les méthodes de la classe mère** pour redéfinir leurs comportements sans changer leur signature.

superclasse commune



Méthode virtuelle vs méthode virtuelle pure

La fonction virtuelle

- a sa définition dans la classe mère.
- a une **implémentation par défaut** dans la classe de base.
- Seules les classes filles qui en ont **besoin** la redéfinissent sinon elles gardent **implémentation de base.**

La fonction virtuelle pure

- n’a pas sa définition dans la classe mère et n'a pas d'implémentation dans la classe de base.
- Classes filles **doivent la redéfinir** et donc fournir une implémentation
- Le fait qu’elle soit pure, rend la **classe abstraite**

```cpp
class Animal {
public:
    virtual void parler() {
        std::cout << "Je suis un animal." << std::endl;
    }
};

class Chien : public Animal {
public:
    void parler() override {
        std::cout << "Wouf !" << std::endl;
    }
};

class Chat : public Animal {
public:
    void parler() override {
        std::cout << "Miaou !" << std::endl;
    }
};

void faireParler(Animal* animal) {
    animal->parler();
}
```

```cpp
Chien monChien;
Chat monChat;

faireParler(&monChien);
faireParler(&monChat);
```

Il permet d'utiliser un pointeur ou une référence vers une classe de base pour appeler des méthodes d'une classe dérivée. Autrement dit, on appelle une méthode sans *savoir exactement* à quel type d'objet tu as affaire.

Ce qui se passe :

- À la compilation, faireParler() accepte un Animal*.
- À l'exécution, **le bon parler() est appelé dynamiquement** grâce au mot-clé virtual.
- Le C++ utilise une **vtable (table de fonctions virtuelles)** pour cela.

On ne connaît pas le type de l'animal

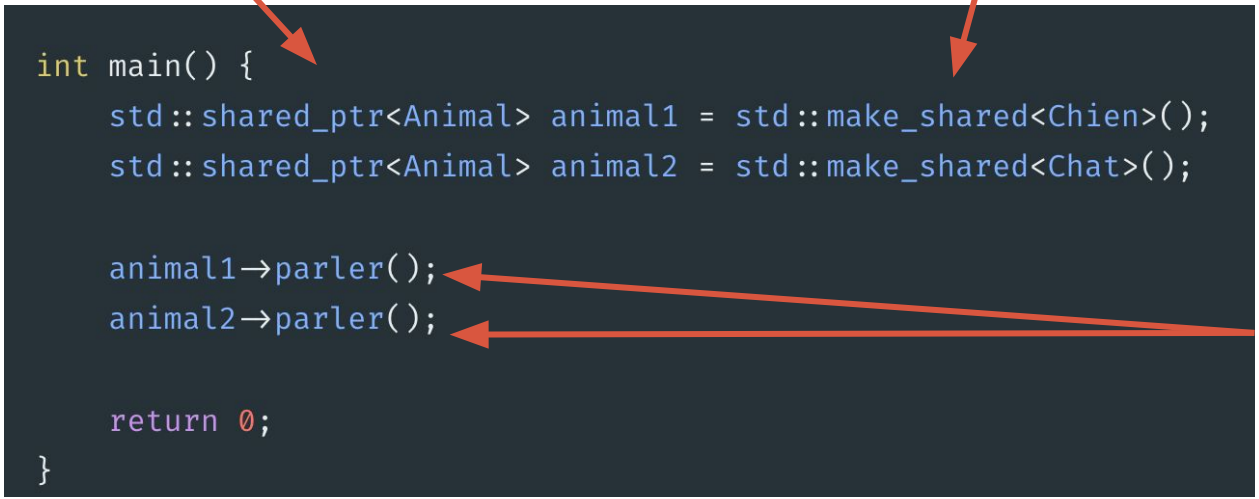
Destructeur virtuel pour assurer une destruction propre

Si le destructeur de Animal n'est pas virtual et que l'on détruit un objet dérivé via un pointeur de type Animal*, seul le destructeur de Animal sera appelé, et les ressources spécifiques à la classe dérivée ne seront pas libérées — c'est une fuite de mémoire.

```cpp
#include <iostream>
#include <memory>

class Animal {
public:
    virtual void parler() const {
        std::cout << "L'animal fait un bruit." << std::endl;
    }
    virtual ~Animal() {}
};

class Chien : public Animal {
public:
    void parler() const override {
        std::cout << "Woof!" << std::endl;
    }
};

class Chat : public Animal {
public:
    void parler() const override {
        std::cout << "Miaou!" << std::endl;
    }
};
```

Permet plusieurs copies du pointeur tout en gérant automatiquement la destruction de l'objet lorsque le dernier shared_ptr qui le pointe est détruit ou réinitialisé.

fonction est utilisée pour créer un std::shared_ptr de manière efficace



Appels polymorphiques

/ Abstraction

/ Classes abstraites

- Une classe abstraite ne **peut pas être instanciée.**
- Une classe est abstraite si elle possède **au moins une méthode abstraite** (virtuelle pure)**.**
- Elles sont donc des **schémas** pour des classes enfant.

Méthode virtuelle

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

class Animal {
public:
    virtual void speak() {
        cout << "Some generic animal sound" << endl;
    }
};

class Dog : public Animal {
public:
    void speak() override {
        cout << "Woof!" << endl;
    }
};

int main() {
    Animal* a = new Dog();
    a->speak(); // Affiche "Woof!"
    delete a;
}
```

Méthode virtuelle pure (abstraite)

```
class Animal
{
public:
  Animal(string name);
  virtual void makeSound() = 0;
}
class Dog : public Animal
{
public:
  Dog(string name);
  void makeSound() override
  {
     std::cout << "Woof!" << std::endl;
  }
}
```

Méthode virtuelle pure (abstraite)

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

Le mot clé virtual ainsi que l'initialisation a 0 de la méthode la rendent virtuelle pure (abstraite).
Il s'agit d'une syntaxe spéciale comprise par le compilateur

Override

Une méthode abstraite doit impérativement être implémentée dans les classes enfant.

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

```
void makeSound() override
{
  std::cout << "Woof!" << std::endl;
}
```

Interface

```
class Interface
{
public:
  virtual void action() = 0;
  virtual ~Interface() {};
}
```

Une interface est une classe dont toutes les méthodes sont abstraites.

Le destructeur virtuel permet d'utiliser aussi le destructeur de la classe enfant.

/ Surcharge

Surcharge de fonctions

```
class Print {
public:
  void show(int a) {
    std::cout << "Integer " << a << std::endl;
  }
  void show(double a) {
    std::cout << "Double " << a << std::endl;
  }
}
```

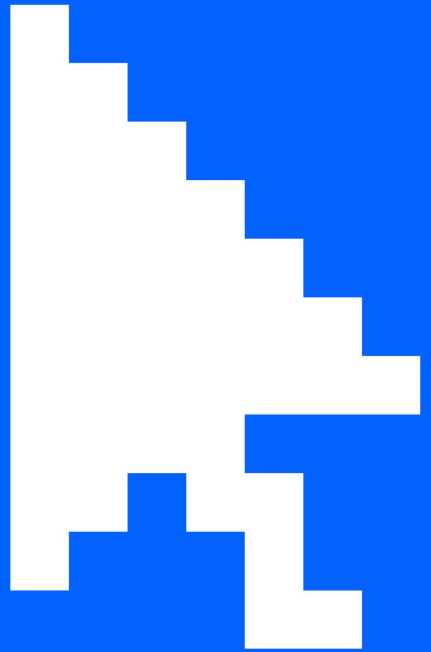
Des méthodes peuvent avoir le même nom, tant qu'elles n'ont pas la même signature.

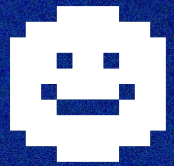
Surcharge d'opérateurs

```
class Point
{
public:
   int x, y;
   Point(int x, int y) : x(x), y(y){}
   Point operator+(Point p) {
         return Point(x + p.x, y + p.y);
      }
}
```

Ici l'opérateur + est surchargé :
un comportement est défini si l'on additionne deux instances de la classe Point.

Des Questions ?





La Plateforme