

Cours C++

Programation orientée objet - Classes en C++

Plein de tableaux...

```
vector<string> prénoms_étudiants = {"A", "B", "C"};  
vector<string> noms_étudiants = {"D", "E", "F", "G"};  
vector<date> datesnaissance_étudiants = {date(2022,10,4), ...};  
vector<float> notes_étudiants (4);  
notes_étudiants[2] = 19;
```

Qu'a-t-on envie de faire ?

→ Regrouper dans une "boite" / structure

```
struct Étudiant {  
    string prénom;  
    string nom;  
    date date_naissance;  
    float note;  
};  
  
vector<Étudiant> étudiants = {  
    { "A", "D", date(2022,10,4), 0 },  
    ...  
};  
étudiants[2].note = 19;
```

Qu'a-t-on envie de faire ?

→ Regrouper dans une "boite" / structure

```
struct Étudiant {  
    string prénom;  
    string nom;  
    date date_naissance;  
    float note;  
};  
  
vector<Étudiant> étudiants = {  
    { "A", "D", date(2022,10,4), 0 },  
    ...  
};  
étudiants[2].note = 19;
```

Un code infernal

```
// appareil A
canal_comm_a = 192.168.1.5:3000
login(canal_comm_a, "utilisateur", "mdp")
calibration_a = 0.74 // W/A
dernière_commande = 0

// appareils B
port_usb = "/dev/usb5"
p = open(port_usb, 115200)
voltage = +inf
// ... deux autres appareils comme B ...

wait(5 sec) // attendre que A démarre
envoyer(canal_comm_a, "initialize, 42")
wait(1 sec)
```

```
while (voltage > 0) {

    char buffer[100]
    voltage = atof( read(p, buffer, 100) )
    if (voltage > 16) afficher erreur "..."
    r = write(p, "reset")
    if (r < 0) afficher erreur "..."
    // ... deux autres appareils comme B ...

    puissance = ... // faire des calculs

    I = puissance*calibration_a
    envoyer(canal_comm_a, "setcurr {I:.4e} 0")
    dernière_commande = I
    envoyer(canal_comm_a, "bidule")
    envoyer(canal_comm_a, "chose")
    état = envoyer(canal_comm_a, "currstate")
    if (état > 4 or état < 0)
        afficher erreur "..."
}
```

Qu'a-t-on envie de faire ?

Fonction du programme noyée dans les détails techniques,
redondance...

Qu'a-t-on envie de faire ?

Cacher, "emballer" :

- ▶ les routines spécifiques à chaque appareil → faire des *fonctions* pour dédupliquer & regrouper
- ▶ les canaux de communications, calibrations, états... → dans une boîte / *structure*
- ▶ combiner ces deux idées en une *interface* claire et simple

Qu'a-t-on envie de faire ?

Cacher, "emballer" :

- ▶ les routines spécifiques à chaque appareil → faire des *fonctions* pour dédupliquer & regrouper
- ▶ les canaux de communications, calibrations, états... → dans une boîte / *structure*
- ▶ combiner ces deux idées en une *interface* claire et simple

Qu'a-t-on envie de faire ?

Cacher, "emballer" :

- ▶ les routines spécifiques à chaque appareil → faire des *fonctions* pour dédupliquer & regrouper
- ▶ les canaux de communications, calibrations, états... → dans une boîte / *structure*
- ▶ combiner ces deux idées en une *interface* claire et simple

Qu'a-t-on envie de faire ?

Cacher, "emballer" :

- ▶ les routines spécifiques à chaque appareil → faire des *fonctions* pour dédupliquer & regrouper
- ▶ les canaux de communications, calibrations, états... → dans une boîte / *structure*
- ▶ combiner ces deux idées en une ***interface*** claire et simple

Idéalement

```
// appareil A
appareil_a = AppareilTypeA(192.168.1.5, "utilisateur", "mdp", 0.74 /*W/A*/)

// appareils B
appareil_b1 = AppareilTypeB("/dev/usb5")
appareil_b2 = AppareilTypeB("/dev/usb7")
appareil_b3 = AppareilTypeB("/dev/usb8")

appareil_a.wait_and_initialize()

while (appareil_b1.last_voltage > 0) {

    v1 = appareil_b1.read_voltage()
    v2 = appareil_b2.read_voltage()
    v3 = appareil_b3.read_voltage()

    puissance = ... // faire des calculs

    appareil_a.set_power(puissance)
    if (appareil_a.état > 4 or appareil_a.état < 0)
        afficher erreur "..."
```

Notion de classe

- ▶ Programmation orientée objet : analyse du problème en termes *de nature et de structure des données*, et des *actions* sur ces structures
- ▶ \neq programmation impérative (pensée en terme de procédure, d'instructions)
- ▶ Une classe est
 1. une généralisation de la notion de type (int, double,...)
 2. une association de *données membres* (\equiv *données* \equiv *membres* \equiv *attributs*) et d'opérations s'y rapportant, les *méthodes* (\equiv *fonctions membres*)
- ▶ Mais la POO, c'est bien plus que ça...

Notion de classe

- ▶ Programmation orientée objet : analyse du problème en termes *de nature et de structure des données*, et des *actions* sur ces structures
- ▶ \neq programmation impérative (pensée en terme de procédure, d'instructions)
- ▶ Une classe est
 1. une généralisation de la notion de type (int, double,...)
 2. une association de *données membres* (\equiv *données* \equiv *membres* \equiv *attributs*) et d'opérations s'y rapportant, les *méthodes* (\equiv *fonctions membres*)
- ▶ Mais la POO, c'est bien plus que ça...

Notion de classe

- ▶ Programmation orientée objet : analyse du problème en termes *de nature et de structure des données*, et des *actions* sur ces structures
- ▶ \neq programmation impérative (pensée en terme de procédure, d'instructions)
- ▶ Une classe est
 1. une généralisation de la notion de type (int, double,...)
 2. une association de *données membres* (\equiv *données* \equiv *membres* \equiv *attributs*) et d'opérations s'y rapportant, les *méthodes* (\equiv *fonctions membres*)
- ▶ Mais la POO, c'est bien plus que ça...

Notion de classe

- ▶ Programmation orientée objet : analyse du problème en termes *de nature et de structure des données*, et des *actions* sur ces structures
- ▶ \neq programmation impérative (pensée en terme de procédure, d'instructions)
- ▶ Une classe est
 1. une généralisation de la notion de type (int, double,...)
 2. une association de *données membres* (\equiv *données* \equiv *membres* \equiv *attributs*) et d'opérations s'y rapportant, les *méthodes* (\equiv *fonctions membres*)
- ▶ Mais la POO, c'est bien plus que ça...

Notion de classe

- ▶ Programmation orientée objet : analyse du problème en termes *de nature et de structure des données*, et des *actions* sur ces structures
- ▶ \neq programmation impérative (pensée en terme de procédure, d'instructions)
- ▶ Une classe est
 1. une généralisation de la notion de type (int, double,...)
 2. une association de *données membres* (\equiv *données* \equiv *membres* \equiv *attributs*) et d'opérations s'y rapportant, les *méthodes* (\equiv *fonctions membres*)
- ▶ Mais la POO, c'est bien plus que ça...

Notion de classe

- ▶ Programmation orientée objet : analyse du problème en termes *de nature et de structure des données*, et des *actions* sur ces structures
- ▶ \neq programmation impérative (pensée en terme de procédure, d'instructions)
- ▶ Une classe est
 1. une généralisation de la notion de type (int, double,...)
 2. une association de *données membres* (\equiv *données* \equiv *membres* \equiv *attributs*) et d'opérations s'y rapportant, les *méthodes* (\equiv *fonctions membres*)
- ▶ Mais la POO, c'est bien plus que ça...

Déclaration de classe : attributs

- Classe → généralisation de la notion de type :
collection de variables

particule.h

```
class Particule {  
public: // visibilité des membres (cf. cours encapsulation)  
  
    // Déclaration des attributs = données membres  
    double masse; // kg  
    double charge; // C  
    vec3 impulsion; // kg·m/s  
};
```

👉 Ne pas oublier le point virgule à la fin de la déclaration

Déclaration de classe : attributs

- ▶ Classe → généralisation de la notion de type :
collection de variables

particule.h

```
class Particule {  
public: // visibilité des membres (cf. cours encapsulation)  
  
    // Déclaration des attributs = données membres  
    double masse; // kg  
    double charge; // C  
    vec3 impulsion; // kg·m/s  
};
```

👉 Ne pas oublier le point virgule à la fin de la déclaration

Instance de classe

Une fois que l'on a déclaré la classe (\equiv le type), on peut créer une variable de ce type :

```
Particule my_electron; // <- instance de la classe Particule
```

En mémoire, `my_electron` est simplement une collection `{double, double, vec3}`. On appelle ça une *instance*, ou un *objet*.

Accès aux membres :

```
cout << my_electron.charge;
```

```
my_electron.masse = 9.109e-31;
```

Instance de classe

Une fois que l'on a déclaré la classe (\equiv le type), on peut créer une variable de ce type :

```
Particule my_electron; // <- instance de la classe Particule
```

En mémoire, `my_electron` est simplement une collection `{double, double, vec3}`. On appelle ça une *instance*, ou un *objet*.

Accès aux membres :

```
cout << my_electron.charge;
```

```
my_electron.masse = 9.109e-31;
```

Déclaration de classe : méthodes

- Classe → association d'attributs et de méthodes (fonctions membres)

particule.h

```
class Particule {  
public:  
  
    // Déclaration des attributs = données membres  
    double masse; // kg  
    double charge; // C  
    vec3 impulsion; // kg·m/s  
  
    // Déclaration des méthodes = fonctions membres  
    void initialise(double masse_MeV, int charge_u, vec3 impulsion_SI);  
    void afficher();  
    double masse_en_MeV();  
    double énergie_en_MeV();  
};
```

Méthodes

Méthode = fonction qui agit sur une instance :

```
my_electron.initialise(...); // agit sur my_electron
```

```
my_proton.initialise(...); // agit sur my_proton
```

Ça n'est pas une fonction globale !

Méthodes

Méthode = fonction qui agit sur une instance :

```
my_electron.initialise(...); // agit sur my_electron
```

```
my_proton.initialise(...); // agit sur my_proton
```

Ça n'est pas une fonction globale !

Définition de méthodes de classes

- Identique à la définition d'une fonction mais préfixée du nom de la classe et de l'opérateur de portée ::

particule.cpp

```
#include <iostream>
```

```
#include "particule.h"
```

```
void Particule::initialise(double masse_MeV, int charge_u, vec3 impulsion_SI) {  
    masse = masse_MeV * J_per_MeV / (c*c);  
    charge = charge_u * unit_charge;  
    impulsion = impulsion_SI;  
}
```

```
void Particule::afficher() {  
    std::cout << "m=" << masse_en_MeV() << " MeV/c2, q=" << charge/unit_charge << ...  
}
```

```
double Particule::énergie_en_MeV() { //  $E^2 = m^2c^4 + p^2c^2$   
    c2 = c * c;  
    return sqrt( pow(masse*c2, 2) + impulsion.norme_carrée()*c2 ) / J_per_MeV;  
}
```

Définition de méthodes de classes

► Implicitement :

particule.cpp

```
#include <iostream>
```

```
#include "particule.h"
```

```
void Particule::initialise(double masse_MeV, int charge_u, vec3 impulsion_SI) {  
    INSTANCE.masse = masse_MeV * J_per_MeV / (c*c);  
    INSTANCE.charge = charge_u * unit_charge;  
    INSTANCE.impulsion = impulsion_SI;  
}
```

```
void Particule::afficher() {  
    std::cout << "m=" << INSTANCE.masse_en_MeV() << " MeV/c², q=" << INSTANCE.charge/unit_charge << endl;  
}
```

```
double Particule::énergie_en_MeV() { //  $E^2 = m^2c^4 + p^2c^2$   
    c2 = c * c;  
    return sqrt( pow(INSTANCE.masse*c2, 2) + INSTANCE.impulsion.norme_carrée()*c2 ) / J_per_MeV;  
}
```

Définition de méthodes de classes

- Implicitement, en d'autres termes :

particule.cpp

```
#include <iostream>
#include "particule.h"

void Particule::initialise(double masse_MeV, int charge_u, vec3 impulsion_SI) {
    MOI.masse = masse_MeV * J_per_MeV / (c*c);
    MOI.charge = charge_u * unit_charge;
    MOI.impulsion = impulsion_SI;
}

void Particule::afficher() {
    std::cout << "m=" << MOI.masse_en_MeV() << " MeV/c², q=" << MOI.charge/unit_charge
}

double Particule::énergie_en_MeV() { //  $E^2 = m^2c^4 + p^2c^2$ 
    c2 = c * c;
    return sqrt( pow(MOI.masse*c2, 2) + MOI.impulsion.norme_carrée()*c2 ) / J_per_MeV;
}
```

Définition de méthodes de classes

- Implicitement, en vrai C++ :

particule.cpp

```
#include <iostream>
```

```
#include "particule.h"
```

```
void Particule::initialise(double masse_MeV, int charge_u, vec3 impulsion_SI) {  
    this->masse = masse_MeV * J_per_MeV / (c*c);  
    this->charge = charge_u * unit_charge;  
    this->impulsion = impulsion_SI;  
}
```

```
void Particule::afficher() {  
    std::cout << "m=" << this->masse_en_MeV() << " MeV/c2, q=" << this->charge/unit_cha  
}
```

```
double Particule::énergie_en_MeV() { //  $E^2 = m^2c^4 + p^2c^2$   
    c2 = c * c;  
    return sqrt( pow(this->masse*c2, 2) + this->impulsion.norme_carrée()*c2 ) / J_per_M  
}
```

Exemple d'utilisation de la classe particule

test_particule.cpp

```
#include<iostream>
using namespace std;
#include "particule.h"

int main() {
    // Instanciation d'objets de type particule
    Particule my_electron, my_proton;
    my_electron.initialise(0.511, -1, {2e-23,-4e-23,0});
    my_proton.initialise(938.0, +1, {0,0,0});

    cout << "my_electron :" << endl;
    my_electron.afficher();
    double E = my_electron.energie_en_MeV();
    cout << "énergie en MeV = " << E << endl;
}
```

Exemple d'utilisation de la classe particule : pointeurs

test_particule.cpp

```
#include<iostream>
using namespace std;
#include "particule.h"

int main() {
    // Instanciation d'objets de type particule
    Particule * ptr_electron = new Particule;
    ptr_electron->initialise(0.511, -1, {2e-23,-4e-23,0});

    cout << "my_electron :" << endl;
    my_electron->afficher();
    double E = my_electron->energie_en_MeV();
    cout << "energie en MeV = " << E << endl;

    delete ptr_electron;
}
```

Objets constants

Rappel : le mot-clé **const** interdit la modification de l'objet auquel il s'applique.

```
const int i = 42;  
i = 3; // erreur de compilation
```

Comment exprimer que :

```
void f (const Particule& p) {  
    p.afficher(); // possible  
    p.initialise(...); // impossible  
}
```

Le compilateur doit savoir quelles méthodes modifient l'objet.

Objets constants

Rappel : le mot-clé **const** interdit la modification de l'objet auquel il s'applique.

```
const int i = 42;  
i = 3; // erreur de compilation
```

Comment exprimer que :

```
void f (const Particule& p) {  
    p.afficher(); // possible  
    p.initialise(...); // impossible  
}
```

Le compilateur doit savoir quelles méthodes modifient l'objet.

Objets constants

Rappel : le mot-clé **const** interdit la modification de l'objet auquel il s'applique.

```
const int i = 42;  
i = 3; // erreur de compilation
```

Comment exprimer que :

```
void f (const Particule& p) {  
    p.afficher(); // possible  
    p.initialise(...); // impossible  
}
```

Le compilateur doit savoir quelles méthodes modifient l'objet.

Méthodes constantes

Le prototype d'une méthode doit se terminer par **const** si celle-ci ne modifie pas l'objet :

```
class Particule {  
    // ...  
    void initialise(double masse_MeV, int charge_u, vec3 impulsion_SI);  
    void afficher() const;  
    double masse_en_MeV() const;  
};
```

☞ Ces méthodes ne peuvent pas modifier les membres de la classe

- ▶ Par défaut les méthodes peuvent modifier l'objet.
- ▶ En cas d'oubli de **const**, la méthode *n'est pas utilisable sur des objets constants* !

Méthodes constantes

Le prototype d'une méthode doit se terminer par **const** si celle-ci ne modifie pas l'objet :

```
class Particule {  
    // ...  
    void initialise(double masse_MeV, int charge_u, vec3 impulsion_SI);  
    void afficher() const;  
    double masse_en_MeV() const;  
};
```

☞ Ces méthodes ne peuvent pas modifier les membres de la classe

- ▶ Par défaut les méthodes peuvent modifier l'objet.
- ▶ En cas d'oubli de **const**, la méthode *n'est pas utilisable sur des objets constants* !