

# Créer des classes

[rappel sur la sémantique de valeur, à quoi cela correspond](#)

Deux aspect : structurer les données et leur appliquer des traitements. Dans les chapitres précédents, vu la partie traitement : l'algorithmique. La POO vise à fournir un méthode pour structurer les données.

## Classe et objet

Classe = type définit dans le code, objet = ce qui apparaît en mémoire dans le code. objet est l'instanciation d'une classe. Il peut y avoir plusieurs objets qui sont instancier à partir d'une classe .

La syntaxe, pour une classe qui ne fait rien (vide) :

```
class A { // une classe
};

A a1;    // un premier objet, qui s'appelle "a1" et de type "A"
A a2;    // un second objet, qui s'appelle "a2" et de type "A"
```

(attention au point-virgule après les crochets). Il est possible d'utiliser également le mot clé `struct` (la différence sera expliquée ensuite). Dans les 2 cas, cela permet de créer une classe

```
struct B {
};
```

on peut avoir plusieurs objets instancié à partir d'une même classe, mais on ne peut pas avoir 2 classes avec le même nom (tout comme on ne

pouvais pas avoir 2 variables avec le même nom)

En pratique, une classe est un type. Par exemple, vous avez déjà vu des exemples de classe : string, vector, array. Leur utilisation est identique à n'importe quel type de la bibliothèque standard.

Le nom des classes suit les mêmes règles (caractères autorisés) que les noms de variables ou de fonction.

Il est classique lors que l'on écrit des codes d'exemples de donner des noms bateau aux noms. Vous avez déjà vu par exemple i, j, k pour nommer des entiers, s ou str pour des chaînes, f(), g(), h(), foo(), bar() pour des fonctions. Pour les classes, on donne souvent une lettre majuscule : A, B, C, etc.

Ces choix de noms ne sont bien sûr pas à utiliser dans un vrai projet, uniquement pour des exemples ou poser une question sur un forum.

Comme il n'y a pas de différence entre les classes de la lib standard et vos propres classes, vous pouvez les utiliser n'importe où, comme vous le faisiez pour les autres types.

```
class A {};           // pour définir un type (A est une classe)

A a {};             // pour définir une variable (a est un
objet)
void f(A a) {}      // comme paramètre de fonction
A g() {}            // comme paramètre de retour de fonction

f(a);               // comme argument de fonction
a = g();            // comme résultat de fonction
auto a = f();       // avec auto

template<typename T>
void h(T t) {}      // fonction template

h<A>();              // comme argument template de fonction
std::vector<A> v;   // comme argument template d'une classe
```

## Les membres d'une classe

Une classe rassemble des variables et des fonctions. S'appellent variables membres ou attribues et fonctions membres ou méthodes. Déclare membres de la même façon que d'habitude :

```
struct A {  
    int i {};  
  
    int f() {  
        cout << "appel de f()" << endl;  
    }  
};
```

Pour les appeler, il faut créer un objet puis utiliser ses membres (comme vous l'avez fait pour les classes de la lib standard) :

```
A a {};  
  
a.i = 123;  
cout << a.i << endl;  
  
a.f();
```

Les membres d'une classe ne sont pas au même niveau (on parle de portée) que les fonctions libres, il n'y a pas de conflit entre les noms et ils est donc possible de donner le même nom à une fonction et une fonction membre.

```
void f() {} // fonction libre  
  
struct A {  
    f() {} // fonction membre  
};  
  
int main() {  
    f(); // appel de la fonction libre  
}
```

```
A a {};  
a.f(); // appel de la fonction membre  
}
```

Vous avez déjà rencontré cette situation avec les fonctions `begin` et `end` par exemple. Pour rappel, ces fonctions peuvent s'appeler comme des fonctions libres ou des fonctions membres :

```
vector<int> v {};  
  
begin(v); // fonction libre  
v.begin(); // fonction membre
```

Pour déclarer de tels fonctions, il faut simplement créer une fonction membre et une fonction libre qui prend un paramètre :

```
struct A {  
    void f() { ... } // la fonction membre  
};  
  
void f(A const& a) { // on utilise une référence sur  
    l'objet, pour ne pas le copier  
    a.f();           // on appelle la fonction membre  
}  
  
A a {};  
a.f();  
f(a);
```

Les deux fonctions font la même chose.

Définitions :

- déclaration : on dit au compilateur qu'un identifiant existe (A, i, etc)
- définition : on dit au compilateur à quoi correspond à un identifiant
- implémentation : on dit au compilateur comment on fait

Exemple :

```
void f(); // déclaration (on dit que "f" existe) et
          définition (on dit que "f" est
                // une fonction, qui ne prend aucun paramètre et
                // retourne rien

void f() { ... } // implémentation

class A; // déclaration de A

class A { // définition de A
    int i {};
    void f();
};

void A::f() { // implémentation de A::f
};
```

### Type incomplet

Un type est complet lorsqu'il est entièrement défini. Un type qui est simplement déclaré et pas encore défini est incomplet par exemple. Pour les fonctions et classes, on a vu comment séparer déclaration et définition.

Pour une variable, possible aussi de séparer, en utilisant le mot-clé "extern" (cf ailleurs). Utile avec des libs.

```
extern int a; // déclaration

a += 1; // erreur, a est simplement déclaré (ie le
        // compilateur sait que l'identifiant "a"
        // existe, mais il ne sait pas à quoi cela
        // correspond)

int a {}; // ok, définition
```

ODR (one definition rule) : on peut avoir plusieurs déclaration, mais une

seule définition header guard : éviter plusieurs définition d'une même classe

```
class A {};  
class A {}; // erreur, double définition  
  
class B;  
class B; // ok, double déclaration
```

Par contre, avant d'utiliser il faut que cela soit défini :

```
class A;  
class A; // ok, double déclaration  
  
A a {}; // erreur, non défini  
  
class A {};  
A a {}; // ok, défini
```

déclaration anticipé

```
class A {  
    B b {}; // erreur, B n'est pas connu à ce niveau  
};  
  
class B {  
};
```

```
class B; // déclaration anticipée de B  
  
class A {  
    B b {}; // ok, le compilateur sait que B existe (même  
            // s'il ne sait pas à quoi cela correspond  
};  
  
class B {  
};
```

problématique de double inclusion de classe dans plusieurs fichiers

## Visibilité des membres d'une classe

3 types de visibilité :

- public (publique) : le membre est visible depuis l'extérieur de la classe ;
- private (privé) : le membre est n'est pas visible depuis l'extérieur de la classe ;
- protected (protégé) : le membre est visible que depuis les classes dérivées.

Le troisième cas est lié à la notion d'héritage, sera vu dans les classes à sémantique d'entité.

Par exemple, une classe A avec un membre f() ou i. Pour appeler le membre, on utilise l'opérateur . (comme déjà fait avec begin et end par exemple) :

```
A a {}; // défini a
a.i = 0; // accès à membre i
a.f(); // accès à membre f()
```

Bien faire attention aux notions de variables et types. A est un type, il permet de déclarer une variable. a est une variable, on peut appeler . dessus.

Dans ce code, on utilise la classe A, on est l'extérieur de la classe. Comme on a accès aux membres, on a donc un accès public. Avec un membre privé, essayer d'accéder depuis l'extérieur produit une erreur du compilateur :

```
main.cpp:9:7: error: 'i' is a private member of 'A'
    a.i = 123;
      ^
main.cpp:2:9: note: implicitly declared private here
    int i {};
      ^
main.cpp:10:7: error: 'f' is a private member of 'A'
```

```
    a.f();
      ^
main.cpp:3:10: note: implicitly declared private here
    void f() {}
      ^
2 errors generated.
```

On définit la visibilité des membres en utilisant les mots-clés `public`, `private` et `protected`. Par exemple :

```
class A {
public:
    int i {};
    void foo() {}
private:
    int j {};
    void g() {}
};
```

La déclaration de visibilité s'applique tant qu'un autre identificateur n'est pas spécifiée. Donc dans ce code, `i` et `foo` sont publique et `j` et `g` sont privé.

```
A a {};
a.i = 123; // ok
a.f();    // ok
a.j = 123; // erreur
a.g();    // erreur
```

Par défaut, le mot-clé `class` crée une classe avec des membres en visibilité privée, on omette le `private` s'il est en premier. Donc écrire :

```
class A {
private:
    int i {};
    void foo() {}
};
```

est équivalent à :

```
class A {
    int i {};
```

```
void foo() {}  
};
```

Le mot-clé struct est similaire à class et permet de définir une classe. La seule différence est que struct a une visibilité publique par défaut :

```
class A {  
    int i {};  
};  
  
struct B {  
    int i {};  
};  
  
A a {};  
a.i = 123; // erreur, i est privé avec class si public n'est  
pas spécifié  
  
B b {};  
b.i = 123; // ok, i est publique avec struct par défaut
```

### Accesseurs

Des accesseurs sont des fonctions membres spécifiques, qui permettent de lire et modifier des variables membres. Cela permet de mettre les variables membres en private et contrôler l'accès aux variables.

```
// sans accesseurs  
class A {  
public:  
    int m_i {};  
};  
  
// avec accesseurs  
class B {  
private:  
    int m_i {};  
public:  
    int getI() const { return m_i }  
};
```

```
void setI(int i) { m_i = i; }  
};  
</note>
```

On le appelle souvent "getter" et "setter".

A première vue, semble respecter encapsulation, mais vision des classes comme ensemble de données et pas comme un prestataire de services. Analogie : si on avait un classe Portefeuille, première approche correspond à "voilà mon portefeuille, sers toi", la seconde à "voilà X euros". La seconde est beaucoup plus sécurisée...  
</note>

==== portée, statique et espace de noms ====

On peut remarque que l'on a des syntaxes similaire. Un seul identifiant par portée. Un récapitulatif :

```
<code cpp>  
// déclaration fonction  
void f(); // fonction libre dans la portée globale ::  
  
namespace myspace {  
void f(); // fonction libre dans la portée myspace  
}  
  
class A {  
    void f(); // fonction membre  
    static g(); // fonction membre static  
};  
  
// utilisation  
  
int main() {  
    f(); // fonction libre de ::  
    ::f(); // autre syntaxe, avec portée globale explicite  
  
    myspace::f(); // libre de myspace  
  
    A a {};
```

```
a.f(); // membre non statique  
  
A::g(); // membre static  
}
```

Portée : espace de noms (global ou user), classe, fonction dans lequel une identifiant est défini. Utilisation de l'opérateur de portée ::

## **Nameslookup et signature de fonction**

Quand on rencontre l'utilisation d'un identifiant, comment trouver la fonction qui correspond ? Template, spécialisation template, surcharge, multiple définition (qu'est ce qui rentre dans la signature ?)

==== Encapsulation ====

On a vu que les classes déclarées que vous déclarez sont identiques à celle de la lib standard. La réciproque est vraie : les classes de la lib standard sont identiques au code que vous pouvez écrire. Elles sont écrites en C++, avec la même syntaxe que vous utiliser (des exercices à la fin du cours proposent de réécrire ces classes).

En particulier, il est tout à fait possible d'aller regarder le code C++ des classes et fonctions de la lib standard pour voir comment ils sont implémentés.

Mais la question importante est : avez-vous eu besoin de connaître le code de ces classes et fonctions pour les utiliser ?

La réponse est bien sûr non (heureusement, comme vous les utiliser depuis le début du cours, si vous ne pouviez pas les utiliser sans voir leur code, vous seriez un peu bloqué). Pour utiliser une classe et une fonction, vous avez simplement besoin :

- de connaître son nom et ce qu'elle fait ;
- et de connaître la liste de ses paramètres.

Si une classe ou fonction est correctement conçue, il est généralement possible de savoir ce qu'elle fait et le rôle de chaque paramètres, rien d'avec leurs noms. Si on regarde par exemple la fonction `std::sort` :

```
template< class RandomIt >
void sort( RandomIt first, RandomIt last );
```

Le nom signifie `tri`, on comprend ce qu'elle permet de trier quelque chose. Cette fonction prend deux paramètres, de type `RandomIt`, qui s'appellent `first` et `last`. On comprend donc que cette fonction permet de trier une collection entre un premier élément et un dernier.

Pour `RandomIt`, il faut bien sûr savoir ce qu'est un [RandomAccessIterator](#) pour savoir à quoi cela correspond. Mais (normalement) vous savez que cela correspond à des itérateurs, par exemple dans `vector`.

L'ensemble des informations que l'on donne sur une classe ou une fonction et qui permet de les utiliser s'appelle l'interface d'une classe. Celle-ci contient les noms des classes, fonctions et paramètres que les utilisateurs peuvent utiliser, ainsi que la documentation.

En pratique, cela signifie que pour utiliser une classe, vous pouvez écrire :

doivent déjà connaître la différence entre définition et implémentation ? dans le chapitre sur les fonctions ?

```
void f(); // définition d'une fonction libre

struct A {
    int f(); // définition d'une fonction membre
};
```

Le code de la fonction n'est pas nécessaire pour comprendre comment utiliser cette classe. Le code permettant d'appeler la définition (d'une classe ou d'une fonction). Le bloc de code `{}` est remplacé par `;`

Bien sûr, un moment donné, il faut donner le code correspondant à la fonction. Ce code s'appelle l'implémentation de la fonction. Pour implémenter une fonction membre, il faut indiquer la classe correspondante, en utilisant l'opérateur de portée `::` :

```
// implémentation d'une fonction libre
```

```

void f() {
    ...
}

// implémentation d'une fonction membre
void A::f() {
    ...
}

```

On va même pouvoir aller plus loin et séparer les définitions et implémentation dans 2 fichiers séparés. Le premier dans .h et second dans .cpp. Compilation de .cpp et inclusion de .h

Remarque : sauf tempalte

On peut remarquer un avantage très intéressant de cette séparation : on peut modifier l'implémentation, sans que cela impact la définition. Cela implique donc que l'on peut modifier le code interne d'une classe ou d'une fonction sans avoir besoin de modifier le code qui l'utilise.

Pour être concret, si on écrit une fonction d'incrémntation, on pourra écrire :

```

template<typename T>
void next(T & value) {
    value += 1;
}

int i {};
next(i);

auto it = begin(v);
next(it);

```

Par la suite, on réalise que ce code ne fonctionne pas avec les itérateurs de `std::list` (qui n'est pas un `Randomlterator`, mais un `BidirectionalIterator`,

qui ne propose pas l'opérateur +  .

On peut alors corriger le code en utiliser l'opérateur ++ :

```
template<typename T>
void next(T & value) {
    ++value;
}
```

Le code précédent continue de fonctionner (le code est maintenable) et on peut à présent utiliser `std::list` (le code est évolutif).

La séparation entre définition et implémentation s'appelle l'encapsulation. Ce principe permet de gagner en maintenabilité et évolutivité du code. Plus les classes et fonctions seront correctement encapsulées, pour votre code gagnera en qualité.

Il ne sera pas toujours possible de séparer correctement définition et implémentation. Ce n'est pas grave, il faut juste être conscient que cela aura un impact sur la maintenabilité et l'évolutivité.

Une erreur classique est d'exposer les détails interne d'une classe. Par exemple, si vous avez une classe qui contient un `vector` et que vous voulez pouvoir modifier les éléments de ce `vector`, vous pouvez écrire:

```
struct A {
    vector<int> v {};
};

// ou équivalent, un accesseur :

class B {
    vector<int> v {};
public:
    vector<int> & get_v { return v; }
};
```

On expose `vector` vers l'extérieur, il fait partie de l'interface. Si un jour on utilise un `std::list`, le code utilisateur ne sera pas forcément correct

Possible d'éviter cela en ne mettant pas `vector` en interface. Par exemple, on peut proposer des fonctions `begin` et `end`, comme pour les conteneurs (il faut fournir les versions `const` et non `const`. Pourquoi ? Comment savoir ce qu'il faut fournir ?) :

```

class A {
    Container v {};
public:
    using container = vector<int>;
    using iterator = container ::iterator;
    using const_iterator = container ::iterator;

    iterator      begin()      { return begin(v); }
    const_iterator begin() const { return begin(v); }
    iterator      end()        { return end(v); }
    const_iterator end() const  { return end(v); }
};

```

Autre solution, design pattern visiteur, ie prendre une fonction à appliquer sur chaque élément :

```

class A {
    vector<int> v {};
public:
    template<typename Function>
    void apply(Function f) {
        std::for_each(begin(v), end(v), f);
    }
};

```

===== Principe de responsabilité unique =====

Faire une seule chose et le faire bien

===== Classe template =====

De la même manière que l'on a pu définir des fonctions template, il est possible de définir des classes template. Un exemple de classe template, c'est vector ou array.

La déclaration d'une classe template est similaire à une fonction, il faut ajouter template avec la liste des paramètres template :

```

template<typename T>
class A {
};

```

Les paramètres template peuvent être utilisés dans l'ensemble de la classe, pour la déclaration d'une variable membre ou comme paramètre de fonction membre.

```
template<typename T>
struct A {
    T value {};
    T f(T param);
};
```

### séparation implémentation et définition

Pour instancier une classe template, il faut spécifier les arguments template. Contrairement aux fonctions template, le compilateur ne peut pas déduire les arguments template à partir des arguments de fonction.

```
A<int> a_int {};  
A<double> a_double {};
```

<b>Chapitre précédent</b>	<b>Sommaire principal</b>	<b>Chapitre suivant</b>
---------------------------	---------------------------	-------------------------

[Cours, C++](#)