

26

Premiers pas avec la POO

La programmation orientée objet (abrégée en POO), c'est un terme qu'on rencontre souvent dans le monde du développement. Sa définition exacte et son utilisation varient en fonction des points de vue, des langages qui l'utilisent, des contraintes, des époques. Une chose est sûre : il s'agit d'un paradigme très répandu et très utilisé.

Ce chapitre a pour but de vous introduire à la programmation orientée objet en définissant les principes qui la régissent et en expliquant ce qu'elle peut nous apporter.

La façon dont nous allons vous présenter la POO va peut-être vous surprendre, comparée à d'autres ouvrages. Cela est dû à une volonté de bien revenir sur ses fondements, de manière à l'exploiter avantageusement.

Comme toute solution, la programmation orientée objet a été conçue pour répondre à certains besoins. Il y aura donc des cas où elle sera plus adaptée et dans d'autres plus limitative. Il convient donc de bien en cerner l'esprit et les principes de base afin de l'utiliser à bon escient. C'est ce que nous essayerons de vous transmettre au cours de cette partie.

26.1. Le principe : des objets bien serviables

Penser en termes de données...

Depuis que nous avons commencé ce livre, nous avons pensé et codé en termes de *structures de données* et d'*algorithmes*. Nous avons conçu notre code comme une suite d'instructions et d'opérations, avec tel algorithme appliqué à telle structure ou tel type, afin d'obtenir telle information en sortie.

Prenons un exemple tiré de la vie réelle. Une laverie met à la disposition des clients des machines à laver, mais à leur charge de s'occuper du reste. C'est à chacun de préparer son linge, d'amener sa lessive, de trier par couleur, de vérifier le poids inséré dans la machine, de lancer celle-ci avec le programme adapté au type de vêtements,

de s'occuper du séchage, etc. On est typiquement dans le cas d'une solution assez complexe, avec plusieurs opérations à effectuer les unes à la suite des autres, des traitements particuliers (tri du linge, tri par couleur, etc.). Tout tourne autour du linge, la "donnée centrale" de notre programme de laverie. Si on devait coder ça, on aurait sans doute une structure pour le linge et plusieurs fonctions comme `tri_linge_par_couleur`, `ajouter_lessive`, etc.

...ou de services ?

La notion de *service* est un des fondements de la POO. Plutôt que de fournir une liste de fonctions, d'opérations, d'algorithmes à appliquer, on va abstraire cette complexité technique, la cacher. On va enfermer, camoufler tout ce qui est complexe et n'offrir qu'une interface beaucoup plus simple à utiliser.

Appliquer ce principe à notre laverie revient à créer un pressing. Toutes les opérations de tri, de dosage et autres sont alors cachées, elles sont déléguées. L'utilisation est beaucoup plus simple. Il suffit de venir, déposer le linge sale et repartir avec du linge propre. Nous n'avons rien d'autre à faire qu'attendre (et, accessoirement, payer). Le pressing s'occupe de nous rendre un service "Laver notre linge" et nous nous contentons de l'utiliser, sans chercher plus loin. On n'a pas besoin de maîtriser les arcanes du nettoyage ; des spécialistes en qui on peut avoir confiance vont en assumer la responsabilité. On passe par une abstraction et ce principe est une des pierres angulaires de la programmation orientée objet.

Pour résumer, la programmation orientée objet, c'est créer des services.

Exemple concret

Vous voulez un exemple concret, avec du code ? Ça tombe bien, la bibliothèque standard en est remplie ! Prenez le cas des chaînes de caractères. Plusieurs fois dans le livre, nous avons mentionné qu'il existe en C++ un type de chaînes de caractères hérité du C. Dans ce langage, manipuler les chaînes est plus compliqué. Il faut passer par des fonctions pour les concaténer ou supprimer un caractère, on doit gérer explicitement la mémoire, etc. Bref, pas très pratique.

Le type `std::string` permet quant à lui de manipuler les chaînes de caractères de façon beaucoup plus aisée. C'est la raison pour laquelle on l'utilise tout le temps. Sauf qu'en interne, `std::string` manipule des chaînes de caractères version C. Simplement voilà, la cuisine qu'il fait pour concaténer deux chaînes ou en vider une troisième ne nous intéresse pas, et d'ailleurs elle n'est pas exposée. C'est donc un très bon exemple de service.

26.2. Un peu de vocabulaire

L'objet

Un objet n'est ni plus ni moins qu'une donnée qu'on manipule. Ainsi, dans le code `std::string chaine`, `chaine` est tout simplement un objet de type `std::string`. De même pour `std::vector<int> tableau`, où notre variable `tableau` est un objet de type `std::vector<int>`.

Le type

Afin de définir quels services sont rendus par un objet, il faut définir son modèle, son type. On parle de la *classe d'un objet*. À partir de cette classe, on pourra créer autant d'objets de ce type qu'on veut. On parle d'*instancier la classe*. Quand on écrit `std::string chaine`, on crée une instance du type `std::string`, instance qui se nomme `chaine` et qui est notre objet.

Note > Des variables de types basiques, comme `int`, `double` ou `bool`, peuvent être considérés comme des objets d'un point de vue théorique. En pratique, en C++, on fera la distinction entre ces types dits natifs et les instances de classes, comme `std::string`.

L'interface d'une classe

Un objet peut nous rendre plus ou moins de services, en fonction de son type. Dans le cas de `std::string`, on peut accéder au premier caractère avec `front`, ajouter un caractère à la fin avec `push_back`, etc. Ces fonctions, qui s'appliquent à une instance donnée d'une classe, sont appelées les *fonctions membres*. Nous en avons déjà un peu parlé au chapitre [Mais où est la doc ?](#). Ce sont elles que nous retrouvons sous la rubrique MEMBER FUNCTIONS.

Note > Quand nous avons écrit `friend auto operator<=>(const Fraction& a, const Fraction& b) = default` dans le chapitre sur le sucre syntaxique, nous avons en fait écrit une fonction membre.

Ce terme "fonction membre" est à mettre en relation avec "fonction libre" [ou *Non-member functions* dans la documentation], qui désigne les fonctions ne faisant partie d'aucun objet. Presque toutes les fonctions que nous avons écrites jusque-là étaient des fonctions libres. Beaucoup de celles que nous avons utilisées aussi, comme la fonction `std::size`.

Toutes les fonctions membres d'une classe, ainsi que toutes les fonctions libres qui peuvent manipuler cette classe, composent ce qu'on nomme *l'interface d'une classe*.

Note > L'interface d'une classe regroupe également d'autres concepts, pas uniquement les fonctions membres et les fonctions libres qui manipulent des instances de cette classe. On peut citer, entre autres, la surcharge d'opérateurs. Nous verrons le reste plus tard dans cette partie.

26.3. En C++, ça donne quoi ?

Créer des types, ça, nous savons le faire depuis que nous avons découvert les structures. Jusqu'ici, nous avons traitées ces dernières comme de simples agrégats de données. Il est temps de changer notre vision des choses et de passer à une approche objet.

Penser services

Reprenons l'exemple du chapitre [Reprendrez-vous un peu de sucre syntaxique ?, où nous avons manipulé des fractions](#). Quels services peut-on attendre de la part d'une fraction ? Comment celle-ci se manipule-t-elle ? Qu'est-il pertinent de faire avec une fraction ? La liste ci-après décrit les services sur lesquels nous allons travailler. Elle n'est pas exhaustive, mais constitue un bon début.

- Obtenir sa valeur réelle.
- La simplifier.
- Calculer sa puissance au degré x .

Penser tests

Maintenant que nous avons défini une liste des services que nous aimerions appliquer à un objet de type `Fraction`, il faut réfléchir aux tests que nous allons écrire pour vérifier que notre implémentation est correcte. Prenez le temps de les écrire, ce sont eux qui vous apporteront la garantie que votre code est de qualité. Voici un exemple.

```
#include <cassert>

int main()
{
    Fraction f1 { 5, 2 };
    assert(f1.valeur_réelle() == 2.5 && "5/2 équivaut à 2.5.");

    f1.simplifier();
    assert(f1.numérateur == 5 && f1.dénominateur == 2 && "5/2 après
simplification donne 5/2.");

    f1 = pow(f1, 2);
    assert(f1.numérateur == 25 && f1.dénominateur == 4 && "5/2 au carré
équivaut à 25/4.");

    Fraction f2 { 258, 43 };
}
```

```

assert(f2.valeur_réelle() == 6 && "258/43 équivaut à 6.");

f2.simplifier();
assert(f2.numérateur == 6 && f2.dénominateur == 1 && "258/43 après
simplification donne 6/1.");

f2 = pow(f2, 2);
assert(f2.numérateur == 36 && f2.dénominateur == 1 && "6/1 au carré
équivaut à 36/1.");

return 0;
}

```

Notez que les tests montrent qu'on aura à manipuler tant des fonctions membres que des fonctions libres.

Fonctions membres

Nous sommes maintenant prêts à définir notre classe. Commençons par les fonctions membres. Celles-ci se définissent au sein même de la classe et s'écrivent comme des fonctions classiques.

```

struct Fraction
{
    double valeur_réelle();
    void simplifier();
};

```

Le prototype n'a rien d'extraordinaire, mais l'implémentation est légèrement inhabituelle. En effet, pour l'écrire, il faut préfixer l'identifiant de la fonction membre par le nom de la classe, avec l'opérateur de résolution de portée `::` en guise de séparateur. C'est nécessaire pour que le compilateur sache que la fonction définie est membre d'une classe.

```

struct Fraction
{
    double valeur_réelle();
    void simplifier();
};

double Fraction::valeur_réelle()
{
}

void Fraction::simplifier()
{
}

```