La sémantique d'entité

Jusque maintenant, on a vu qu'une seule sémantique, la sémantique de valeur. Pourquoi ? Principalement parce que la bibliothèque standard contient majoritairement des classes à sémantique de valeur. Les classes à sémantiques d'entités seront plus utilisées dans le code métier.

Cette sémantique impose des contraintes sur l'interface des classes, ce que l'on pourrait voir comme une limitation, mais cette sémantique est compatible avec un concept important en programmation orientée objet : l'héritage (qui sera vu dans le chapitre suivant).

Critère d'unicité

A quoi correspond la sémantique d'entité ? Aux objets comme on le conçoit habituellement. Prenons pas exemple une table et de chaises. Chaque objet est *unique*, deux chaises ont beau être identique, ce sont bien deux objets différents. Quoi que l'on fasse avec les chaises (sauf les détruire...), les chaises restent individualisables. C'est le principal critère de la sémantique d'entité : l'unicité.

Au contraire, la sémantique de valeur ne reconnait pas cette unicité. Par exemple, si on considère la valeur “5”, que l'on entre directement cette valeur dans le code, que cette valeur soit le résultat d'un calcul ou le retour d'une fonction, la valeur est toujours la même, il n'est pas possible de distinguer la valeur “5” provenant de l'une ou l'autre moyen de la créer.

```
const int i = 5;
const int j = 3 + 2;
const int k = get_5();
```

Pour prendre un exemple issu de l'univers informatique, prenons par exemple les fenêtres d'un système. Chaque fenêtre est individualisable,

en général par leur position et leur dimensions. Même si deux fenêtres semblent identiques en tous points, il reste possible de les déplacer séparément, d'en fermer une seule, etc. Ce sont bien des entités, au sens “objet unique”.

En terme de code, cela signifie que l'on va pouvoir écrire :

```
auto w1 = create_widget("Windows 1");
auto w2 = create_widget("Windows 2");
```

Conséquences sur l'interface des classes

Fondamentalement, la syntaxe pour créer une classe à sémantique d'entité est la même que pour la sémantique de valeur. Ce qui va distinguer l'une ou l'autre sémantique sera les fonctions que l'on va créer ou non.

Pour rappel, pour définir une classe, il suffit d'utiliser le mot clé `class` ou `struct`, suivi du nom de la classe et de sa définition. Les règles pour le nom des classes sont identiques à celles des noms de variables et fonctions (n'importe quelle lettre majuscule ou minuscule, le caractère souligné et les chiffres sauf en première position).

```
class MyClass {
};
```

La définition de la classe est donnée dans un bloc, entre crochets. Il ne faut pas oublier le point-virgule à la fin de la définition de la classe.

Accessibilité des membres

La différence entre `class` et `struct` est que dans le premier cas, les membres ne sont pas accessibles depuis l'extérieur de la classe, dans le second cas, ils le sont. Pour changer l'accessibilité, il faut utiliser les mots-clés `public` et `private`.

En pratique, il est classique d'indiquer systématiquement l'accessibilité.

```
class MyClass {
private:
    int un_variable_privee {};

public:
    int un_variable_publique {};
};
```

Constructeur et destructeur

Pour rappel, un constructeur est une fonction membre particulière qui est appelée lorsqu'un objet est créé, tandis que le destructeur est *systématiquement* appelé lorsque l'objet est détruit. Il peut exister plusieurs constructeurs dans une classe, mais un seul destructeur. Le nom d'un constructeur est toujours le nom de la classe, celui du destructeur est le nom de la classe précédé d'un tile `~`.

```
class MyClass {
public:
    MyClass();                          // constructeur par
défaut
    MyClass(int i, int j, int k);       // constructeur avec
paramètres

    ~MyClass();                         // destructeur
};
```

Si on ne fournit aucun constructeur, le compilateur va créer différents constructeurs par défaut. On verra par la suite qu'il est souvent intéressant d'interdire explicitement la création par défaut de certains types de constructeur ou au contraire de demander la création explicite de certains constructeur. Par exemple, si on fournit un constructeur avec paramètres, le constructeur par défaut sera désactivé. Il faudra alors demander sa création explicite avec le mot-clé `default`. De même, si on souhaite desactiver le constructeur par copie, on pourra utiliser le mot-clé `delete`.

```
class MyClass {
public:
    MyClass() = default;                // constructeur par défaut
    MyClass(int i, int j, int k);       // constructeur avec paramètres
    MyClass(MyClass const&) = delete;   // constructeur par copie
};
```

Idem pour le destructeur, si on n'en fournit pas, le compilateur proposera un destructeur par défaut.

Le plus souvent, ces constructeurs et destructeur seront publics, mais il peut être intéressant de les mettre en privé. Dans ce cas, il ne sera pas possible de créer ces objets directement en appelant le constructeur. On pourra alors créer par exemple une fonction (ou classe) dédiée pour la création des objets (voir le design pattern *factory*). Il faudra alors mettre cette fonction (ou classe) en `friend` pour qu'elle soit autorisée à accéder aux membres privés.

```
MyClass create_objet();

class MyClass {
private:
    MyClass() = default;              // le constructeur est privé
    friend MyClass create_objet();    // la fonction est amie
};
```

Copie et clonage

L'une des conséquences de la sémantique d'entité est qu'il n'est pas possible de copier un objet. Il faut donc obligatoirement désactiver le constructeur par copie et l'opérateur par copie, pour éviter que le compilateur les créés.

```
class MyClass {
public:
    MyClass(MyClass const&) = delete;
    MyClass& operator=(MyClass const&) = delete;
};
```

Concernant la sémantique de déplacement, il n'y a pas d'obligation de la désactiver. Cela va dépendre de la sémantique que vous souhaitez donner à votre classe.

L'explication de pourquoi il n'est pas possible de copier une classe à sémantique d'entité est lié à l'héritage et l'impossibilité de créer un constructeur virtuel. Vous verrez dans le prochain chapitre ces notions et le pourquoi, pour le moment, accepter cette idée telle quelle.

Opérateurs de comparaison

De la même façon, cela n'a pas de sens de comparer des classes à sémantiques d'entité, puisque par définition, tous les objets sont différents. Cela n'a donc pas de sens de vouloir par exemple déterminer quel objet est supérieur à un autre. Par contre, il est possible de comparer les objets selon d'un de ses membres qui serait à sémantique de valeur. Par exemple, les chaises peuvent être comparés selon leur poids ou leur taille, les fenêtres peuvent être comparées selon leur position ou leur dimension.

```
widget w1, w2;

std::cout << std::boolalpha;
std::cout << (w1 > w2) << std::endl;       // n'a pas de
sens
std::cout << (w1.x > w2.x) << std::endl;   // ok
```

De la même façon, cela n'a pas de sens de tester l'égalité ou la différence avec la sémantique d'entité, puisque par définition les entités

sont uniques. Elles ne peuvent être égale qu'à elles mêmes et différentes dans tous les autres cas.

```
widget w1, w2;

std::cout << std::boolalpha;
std::cout << (w1 == w2) << std::endl;   // n'a pas de sens,
puisque toujours faux
std::cout << (w1 == w1) << std::endl;   // n'a pas de sens,
puisque toujours vrai
std::cout << (w1 != w2) << std::endl;   // n'a pas de sens,
puisque toujours vrai
```

Il est donc préférable de désactiver explicitement ces opérations, pour éviter de réaliser ces tests. Comme par défaut, seul les opérateurs `==` et `!=` sont définies, il faut les désactiver.

```
class MyClass {
public:
    bool operator==(MyClass const&) = delete;
    bool operator!=(MyClass const&) = delete;
};
```

On pourrait penser que puisque ces tests ne posent pas de problèmes particuliers, on pourrait quand même les conserver, même s'ils retournent un résultat prévisible. En pratique, avec la notion d'héritage que vous verrez dans le prochain chapitre, l'implémentation de ces fonctions n'est pas trivial. Il est donc pertinent de les désactiver, pour éviter les problèmes de comparaison que poserait les fonctions créée par défaut par le compilateur.
Il sera parfois intéressant de pouvoir quand même tester l'unicité des entités, mais cette problématique ne sera pas forcement simple. Nous reviendrons sur ce problème dans un prochain chapitre.

Opérateurs arithmétiques

Pour la même raison que les opérateurs de comparaison, cela n'a pas de sens de définir les opérateurs arithmétiques sur des entités. Seuls les membres à sémantiques de valeur pourraient avoir des opérateurs arithmétiques.

Conclusion

On voit que les classes à sémantique d'entité interdisent de nombreuses fonctions de base que l'on a rencontré dans les classes à sémantiques de valeur (redéfinir des opérateurs de comparaison ou des opérateurs arithmétiques). L'intérêt de cette sémantique sera qu'elle autorise l'héritage de classe, que nous allons voir dans le prochain chapitre.

Chapitre précédent	Sommaire principal	Chapitre suivant

Cours, C++