Cela ne va pas. Les explications sont confuses, pas dans le bonne ordre, les notions importantes ne sont pas présentées en premier.

1. La sémantique d'entité n'est pas une nouvelle syntaxe pour écrire des classes, on y retrouve les principes déjà présentés dans la sémantique de valeur : fonctions et variables membres, accessibilités public/private, les constructeurs/destructeur. 2. Distinction entité/valeur ne se fait pas sur la syntaxe (les règles d'écriture des classes), mais sur la sémantique (comme on utilise chaque type de classe ; ou dit autrement, quels sont les services rendus par chaque type de classe). 3. Par exemple, pas de conversion, comparaison, affectation pour la sémantique d'entité. 4. Par contre, de nouvelles possibilités : héritage, polymorphisme, fonctions virtuelles

---

L'héritage

L'héritage de classe permet de créer une classe à partir d'une autre classe (appelée classe parente), en utilisant dans la nouvelle classe (appelée classe enfant ou classe dérivée) les membres de la classe parent.

Conceptuellement, l'héritage est une notion simple à comprendre, puisque assez intuitive. Reprenant notre exemple sur les chaises. Une chaise possède des propriétés (ses dimensions, sa couleur, sa forme, etc) et rend des services (“déplacer la chaise”, “s’asseoir sur la chaise”, etc).

A partir de cette classe “chaise”, il sera possible de créer d'autres classes, qui possèdent au moins toutes les mêmes propriétés et rend au moins les mêmes services. Par exemple, il sera possible de créer des chaises à roulettes (qui permettent aussi de s’asseoir, mais propose en plus le service “rouler”) ou les chaises avec accoudoirs (qui permettent aussi de s’asseoir, mais également de poser les coudes).

On voit dans ces exemples que l'héritage n'a de sens que s'il respecte

une contrainte forte : la classe dérivée ne doit pas entrer en contradiction avec la classe parente. Cela n'aurait pas de sens d'avoir une chaise qui ne permet pas de s'asseoir, ça ne serait plus une chaise. Cette notion est exprimée dans le principe de substitution de Liskov, que nous verrons en détail par la suite. Pour résumer l'idée, il faut simplement que l'on puisse utiliser la classe enfant n'importe où l'on utilise la classe parente.

L'héritage d'une classe est définie lors de la déclaration d'une classe, juste après le nom de la classe.

```
class MyParentClass {};

class MyClass : MyParentClass {
};
```

Rappel sur la visibilité des membres

Pour rappel, les membres d'une classe peuvent avoir trois types de visibilité : publique, privé et protégé. Les visibilités publiques et privées ont déjà été utilisées dans la sémantique de valeur : les membres privés ne sont utilisables quand dans les autres fonctions membres de la même classe, les membres publics sont utilisables en dehors de la classe.

```
class MyClass {
public:
    int i {};
    int foo();
private:
    int j {};
    int bar();

    void une_fonction();
};

void MyClass::une_fonction() {
    std::cout << i << std::endl;       // ok, accès à un
membre publique depuis une fonction membre
    std::cout << j << std::endl;       // ok, accès à un
membre privé depuis une fonction membre
```

```
    std::cout << foo() << std::endl;   // ok, accès à un
membre publique depuis une fonction membre
    std::cout << bar() << std::endl;   // ok, accès à un
membre privé depuis une fonction membre
}

int main() {
    MyClass c;
    std::cout << c.i << std::endl;     // ok, accès à un
membre publique depuis l'extérieur
    std::cout << c.j << std::endl;     // erreur, accès à un
membre privé depuis l'extérieur
    std::cout << c.foo() << std::endl; // ok, accès à un
membre publique depuis l'extérieur
    std::cout << c.bar() << std::endl; // erreur, accès à un
membre privé depuis l'extérieur
}
```

Ces différentes visibilités permettent de contrôler ce qui appartient à l'interface d'une classe (ie ce que les utilisateurs d'une classe peuvent voir de cette classe) et ce qui appartient à son implémentation. Dans les cas où l'on souhaite qu'aucun membre privé ne soit visible dans le fichier d'en-tête, il est possible d'utiliser l'implémentation privée (*Pimpl*) vue dans un chapitre précédent.

Accès aux membres hérités

Avec la notion d'héritage, il faut ajouter un nouveau type d'accès aux membres d'une classe via les classes enfants. Il faut donc élargir la notion de visibilité dans ce cas :

- dans les membres des classes enfants ;
- pour les utilisateurs des classes enfants.

```
class MyParentClass {
public:
    int i {};
    void foo();
};
```

```
class MyClass : public MyParentClass {
public:
    void bar();
};

void MyClass::bar() {
    std::cout << i << std::endl;       // accès au membre i
de la classe parente
    std::cout << foo() << std::endl;   // accès au membre
foo() de la classe parente
}

int main() {
    MyClass c;
    std::cout << c.i << std::endl;     // accès au membre i
de la classe parente
    std::cout << c.foo() << std::endl; // accès au membre
foo() de la classe parente
}
```

On voit ici que l'on peut accéder directement aux membres de la classe parente et les manipuler comme si c'était des membres de la classe `MyClass`. On parle de relation “EST-UN” puisque la classe enfant est utilisable comme si elle était une classe de type `MyParentClass`.

Remarque : il est également possible d'utiliser `this` dans les fonctions membres, comme déjà vu dans les chapitres précédents pour accéder aux membres. Dans ce cas aussi, aucune distinction n'est faite entre les membres de la classe et les membres hérités.

```
void MyClass::bar() {
    std::cout << this->i << std::endl;       // accès au
membre i de la classe parente
    std::cout << this->foo() << std::endl;   // accès au
membre foo() de la classe parente
}
```

On va bien sûr pouvoir contrôler les accès aux fonctions membres obtenus via l'héritage.

Visibilité dans les membres de la classe dérivée

Concernant l'accès depuis une fonction membre de la classe enfant, l'accès aux membres de la classe parente est déterminée selon la visibilité de ces membres dans la classe parente. On connait déjà les visibilité publique et privée, qui autorise ou interdise l'accès aux membres depuis l'extérieur de la classe. Ces deux visibilités ont également un sens dans l'héritage :

- un membre publique d'une classe parente sera accessible dans les fonctions membres des classes dérivées ;
- un membre privé d'une classe parente ne sera pas accessible dans les fonctions membres des classes dérivées.

```
class MyParentClass {
public:
    int i {};
private:
    int j {};
};

class MyClass : public MyParentClass {
public:
    void bar();
};

void MyClass::bar() {
    std::cout << i << std::endl;       // ok, accès au
membre publique i de la classe parente
    std::cout << j << std::endl;       // erreur, accès au
membre privé j de la classe parente
}
```

En complément de ces deux visibilités, il faut donc un troisième type de visibilité, qui interdit l'accès en dehors de la classe, mais peut l'autoriser à l'intérieur des fonctions membres. C'est le rôle de la visibilité protégée, que l'on a déjà cité, mais pas expliqué.

```
class MyParentClass {
protected:
    int i {};
};

class MyClass : public MyParentClass {
public:
    void foo() const;
};

void MyClass::foo() const {
    std::cout << i << std::endl;     // ok, accès à un
membre protected dans une fonction membre
}

int main() {
    MyClass c;
    std::cout << c.i << std::endl;   // erreur, accès à un
membre protected depuis l'extérieur
}
```

Il n'existe pas de visibilité permettant d'avoir accès aux membres de la classe parente depuis l'extérieur de classe enfant, mais pas dans la classe enfant. Cela n'aurait pas trop de sens d'avoir une telle visibilité.
Donc pour résumé, dans les fonctions membres des classes enfants, on peut accéder aux membres publiques et protégés des classes parentes, pas aux membres privés.

Héritage publique et privé

Il faut également regarder l'accès aux membres de la classe parente depuis l'extérieur de la classe enfant. En plus de la visibilité publique, protégée et privée de chaque membre de la classe parente, il faut tenir compte de la visibilité de l'héritage.

Vous aurez surement remarqué que jusque maintenant, on utilisait le

mot-clé `public` dans la définition de l'héritage :

```
class MyClass : public MyParentClass {
};
```

Cela indique la visibilité globale des membres de la classe parente dans la classe enfant. La visibilité de chaque membre de la classe parente est déterminée par combinaison de la visibilité de la classe parente et de la visibilité de chaque membre. La règle est assez simple à retenir : on prend la visibilité la plus restrictive.

Par exemple, si on a un héritage publique et un membre publique dans la classe parente, le membre sera également publique dans la classe enfant (c'est-à-dire qu'il sera possible d'accéder à ce membre depuis l'extérieur de la classe enfant. Si l'héritage est privé et le membre publique dans la classe parente, le membre sera privé dans la classe enfant. Et ainsi de suite.

Le tableau suivant résume la situation :

		Visibilité dans la classe parente		
		`public`	**`protected`**	**`private`**
Visibilité	**`public`**	`public`	`protected`	`private`
dans la	**`protected`**	`protected`	`protected`	`private`
classe enfant	**`private`**	`private`	`private`	`private`

Chapitre précédent **Sommaire principal** **Chapitre suivant**
Cours, C++