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 {
};
```

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 utilisée dans la sémantique de valeur : les membres privés ne sont utilisables quand dans les autres fonctions membres de la même classe, les membres publiques 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
```

```
}
```

Fonctions virtuelles

destructeur virtuel

Notion de programmation par contrat

Polymorphisme d'héritage et pointeurs

Chapitre précédent	Sommaire principal	Chapitre suivant

Cours, C++