Héritage & Polymorphisme

Où comment implanter plusieurs comportements avec la même interface

Un retour vers « le sous-classement »

Définition : le sous-classement consiste à substituer à une fonction ou à un module une autre fonction ou un autre module implantant au moins les mêmes fonctionnalités que la fonction initiale ou le module initial

Concept utilisé de manière générique dans les logiciels.

Exemple :

Handler d'exception en C

Fonction « Call-back » passée aux appels systèmes

Difficulté :

Peu intuitif à mettre en œuvre

Une classe dérivée n'est pas un sous-classement

Définition : *une sous-classe est une classe qui redéfinit les méthodes d'une classe déjà existante et qui ajoute à cette classe des méthodes et champs membres additionnels*

Une classe dérivée cache les méthodes déjà existantes, elle ne les redéfinit pas

Limite des classes « dérivées »

```
class BaseClass
{     …
   public:
   void
   printClassName()
   { std::cout <<
      "BaseClass" <<
      std::endl;
   }
   …
};
```

```
class DerivedClass:
   public BaseClass
{     …
   public:
   void
   printClassName()
   { std::cout <<
        " DerivedClass"
        << std::endl;
   }
   …
};
```

Limite des classes « dérivées »

```
void PrintClassName(
   BaseClass& anObject)
{
   std::cout <<
      "Nom de la classe
   : ";
   anObject.
      printClassName();
   std::cout <<
   std::endl;
}
```

Exécution de :

```
DerivedClass anObject;
PrintClassName(anObject);
```

donne :

```
Nom de la classe : BaseClass
```

Est ce le bon comportement ?

- Dans l'exemple précédent, que souhaitons nous avoir ?
  - Le nom de la classe effectivement « manipulée »
  - Le nom de la classe paramètre passée à la fonction ?
- Dans le cas de la surcharge :
  - La méthode `printClassName` active est :
    - Celle définit par le type classe du paramètre
  - La fonction `PrintClassName` affiche donc :
    - Le nom de la classe paramètre passée à la fonction

Comment faire pour avoir un autre comportement ?

Pour obtenir un comportement différent:

- Il faut que la méthode

  `DerivedClass::printClassName()`

- Remplace la méthode

  `BaseClass::printClassName()`

Conclusion :

- Il ne doit exister qu'une seule méthode `printClassName()` dans une classe dérivée de `BaseClass`
- Si une nouvelle méthode `printClassName()` est définie dans une classe dérivée, elle se substitue à la méthode `printClassName()`

Les méthodes virtuelles

```
class BaseClass
{     …
   public:
   virtual void
   printClassName()
   { std::cout <<
      "BaseClass" <<
      std::endl;
   }
   …
};
```

```
class DerivedClass:
   public BaseClass
{     …
   public:
   [virtual] void
   printClassName()
   { std::cout <<
        " DerivedClass"
        << std::endl;
   }
   …
};
```

TD – Partie 1

- Création de méthodes polymorphes en C++

Un comportement polymorphe

```
void
    PrintClassName(BaseClass&
    anObject)
{
    std::cout <<
        "Nom de la classe : ";
    anObject.printClassName()
    ;
    std::cout << std::endl;
}
```

Exécution de :

```
DerivedClass anObject;
PrintClassName(anObject
    );
```

donne :

```
Nom de la classe : DerivedClass
```

Que veux dire « polymorphisme » ?

- Traduction littérale
  - « Plusieurs formes » !
- En fait
  - « Plusieurs comportements » pour un même objet
- Mais une classe ne définit qu'un seul comportement ! Alors comment est-il possible de parler de « polymorphisme » ?

Pourquoi donc la classe BaseClass est dite polymorphe ?

1. Elle introduit une méthode virtuelle

2. Une classe dérivée peut remplacer une méthode virtuelle par une nouvelle méthode (définir un nouveau comportement)

3. Quand on manipule un objet ayant pour type « `BaseClass` », ce peut-être :

    A. Un objet instance de `BaseClass`

    B. Un objet instance d'une classe dérivée de `BaseClass` et qui surcharge une méthode « virtuelle » de `BaseClass`

4. Les objets ayant pour type « `BaseClass` » peuvent avoir donc des comportements différents.

La destruction d'un objet en C++

- En C++, lors de la destruction, on appelle le destructeur :

```
class BaseClass
{
  ~BaseClass()
   { std::cout <<
    "Delete BaseClass\n« ;
   }
};
```

```
class DerivedClass: BaseClass
{
  ~DerivedClass()
   { std::cout <<
    "Delete DerivedClass\n" ;
   }
};
```

- Mais lequel est appelé

```
BaseClass* base = new DerivedClass();
delete base;
```

La destruction d'un objet en C++

- Destructeur virtuel:

```
class BaseClass
{
  virtual ~BaseClass()
   { std::cout <<
    "Delete BaseClass\n« ;
   }
};
```

```
class DerivedClass: BaseClass
{
  ~DerivedClass()
   { std::cout <<
    "Delete DerivedClass\n" ;
   }
};
```

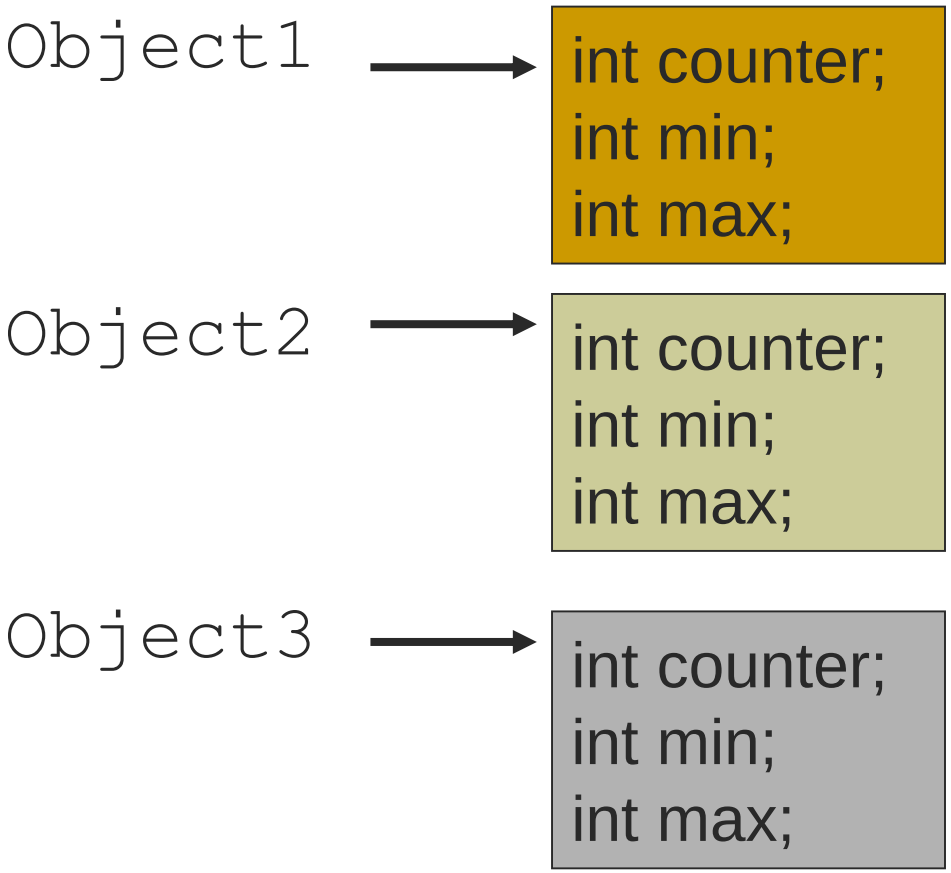
- Mais lequel est appelé

```
BaseClass* base = new DerivedClass();
delete base;
```

TD – Partie 2

- Destructeurs Polymorphe

Implantation des objets : les aspects techniques



- Un objet est une instance de classe
- Un objet `Object1` de type `MyCounter` et objet `Object2` de type `MyCounter` ne différe que par leurs valeurs
- Les méthodes sont communes à tous les objets
- Il suffit seulement de stocker les données pour chaque instance

Méthodes non-virtuelles : les aspects techniques

- Appel d'une méthode « standard » d'une classe BaseClass

```
int nomMethode(T1 param1, T2 param2);
```

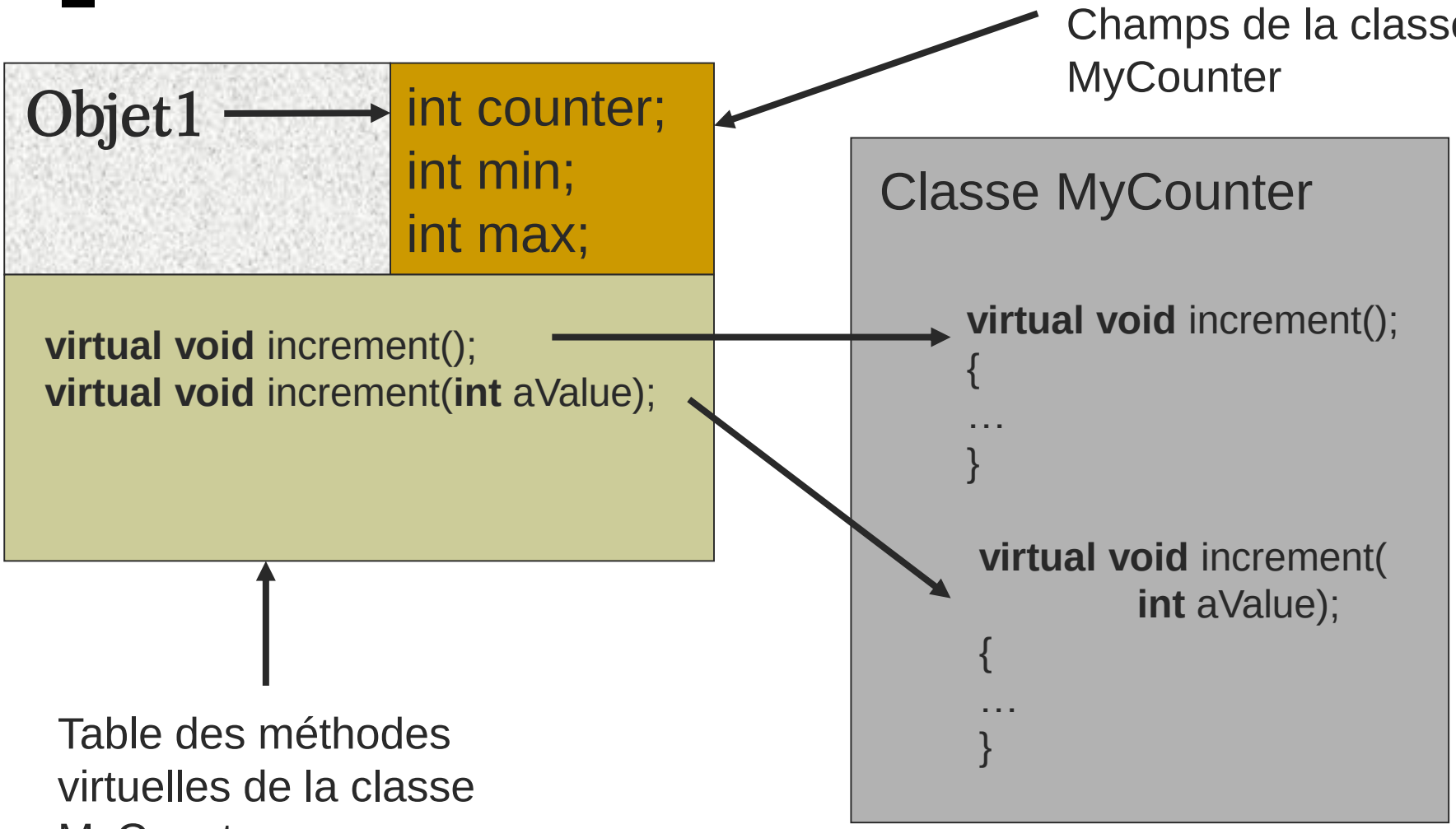
- Que ce passe-t-il lors de l'appel de la méthode ?
  - La méthode est une fonction qui prend un paramètre supplémentaire **this** qui est un pointeur sur les données
  - `monObject.nomMethode(p1, p2);` est transformé en :

```
BaseClass::nomMethode(monObject, p1, p2);
```

Méthodes virtuelles : les aspects techniques

- Les objets `Object1` et `Object2` sont tous les deux des objets instances de classes dérivant de BaseClass
  - `((BaseClass&)Object1).nomMethode(…)` appele la méthode `BaseClass::nomMethode(…)`
  - `((DerivedClass&)Object2).nomMethode(…)` appele la méthode `DerivedClass::nomMethode(…)`
- `Object1` et `Object2` doivent stocker la méthode `nomMethode()` qu'ils appelent

Méthodes virtuelles : les aspects techniques



Méthodes virtuelles : les aspects techniques

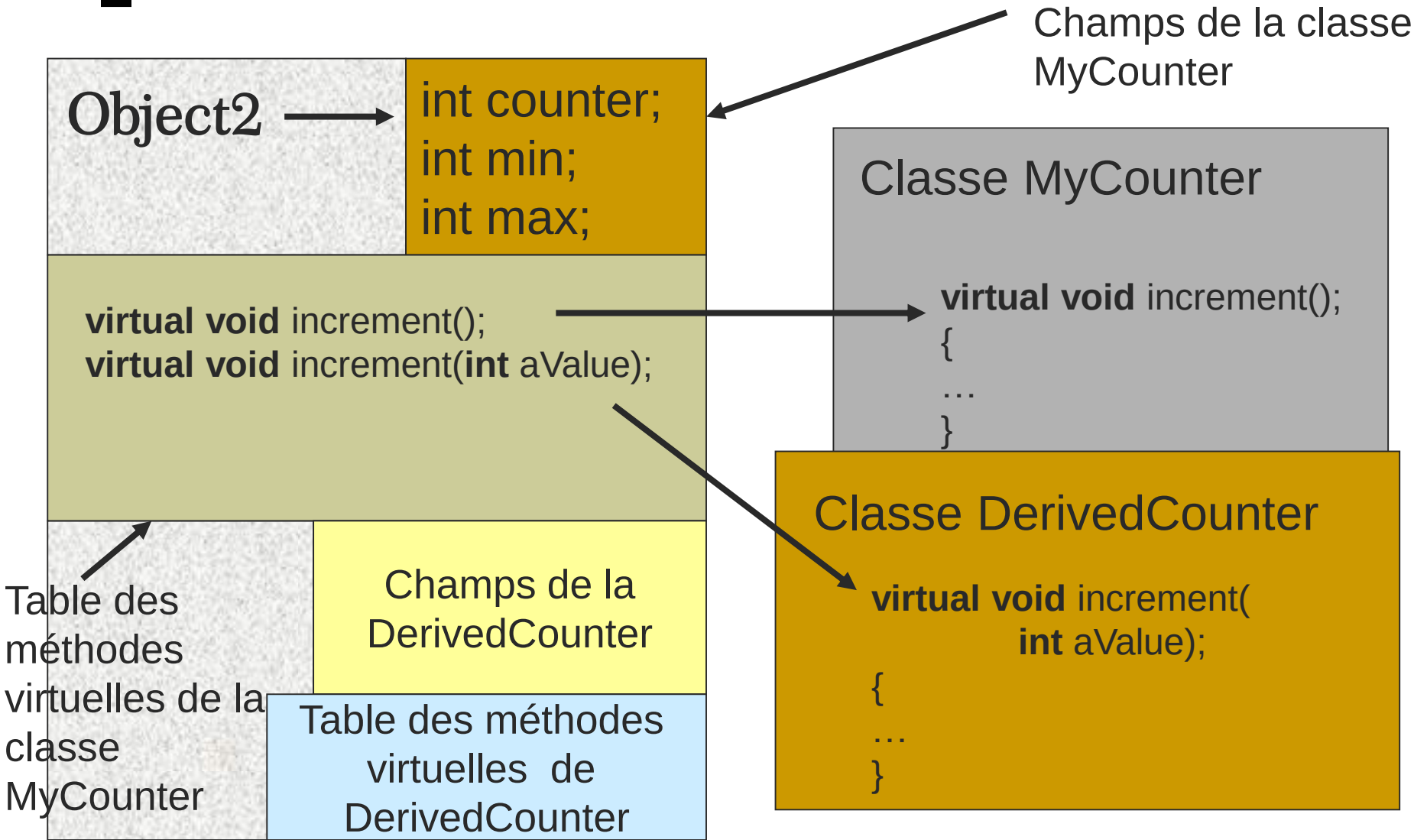
- Fonctionnement de l'appel

```
Object1.increment()
```

1. Cherche dans la table des méthodes virtuelles la méthode « `increment()` »
2. Se connecte à la méthode « `increment()` » stockée dans la table des méthodes virtuelles à la case « `increment()` » donc la méthode :

```
MyCounter::increment(Object1)
```

Méthodes virtuelles : les aspects techniques



Méthodes virtuelles : les aspects techniques

- Fonctionnement de l'appel

```
Object2.increment(int aValue)
```

1. Cherche dans la table des méthodes virtuelles la méthode « `increment(int aValue)` »
2. Se connecte à la méthode « `increment(int aValue)` » stockée dans la table des méthodes virtuelles à la case « `increment(int aValue)` » donc la méthode :

```
DerivedCounter::increment((DerivedCounter&)Obj
                          ect2, aValue)
```

Conclusion

- L’héritage autorise deux types de redéfinition de méthodes :
  - La surcharge des méthodes virtuelles
  - Le masquage des méthodes statiques
- Les méthodes virtuelles
  - Autorisent le polymorphisme « explicite »
- Le polymorphisme autorise
  - D’avoir plusieurs comportements pour des objets de même type
  - D’écrire des algorithmes « génériques » pour des objets ayant des comportements différents

TD – Partie 3

Polymorphisme et Algorithmes Génériques