université de TOURS

Développement objet C++

Dominique H. Li

dominique.li@univ-tours.fr

Licence Informatique - Blois

Université de Tours

université de TOURS

```cpp
#include <iostream>

int main() {
    std::cout << "Hello, World !" << std::endl;
    return 0;
}
```

Les Classes

Développement objet C++

Programmation orientée objet

- L'*abstraction* se concentre sur les caractéristiques importantes d'un objet selon le point de vu de l'observateur
- L'*encapsulation* permet d'occulter l'implémentation d'un objet
- La *hiérarchie* est un classement des abstractions représenté par le mécanismes d'héritage (simple ou multiple)

Programmation procédurale : les limites

- Comment manipuler les objets complexes ?
  - Initialisation
  - Affectation
  - Autres opérations courantes
- Programmation procédurale : par les fonctions d'assistance avec des pointers ou des références

Exemple : programmation procédurale

```
void date_initialize(date_t *, int, int, int);      // Pointeur
void date_increment(date_t *, int);                 // Pointeur
void date_set(date_t *, int, int, int);             // Pointeur
void date_input(date_t &, FILE *);                  // Référence
void date_output(const date_t &, FILE *);           // Référence
```

Les concepts d'objet en terms de POO

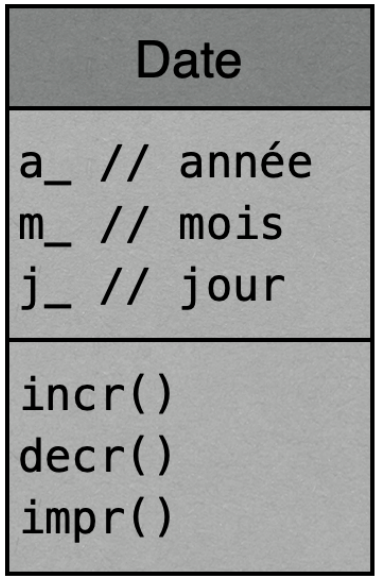
- Un objet est la modélisation d'une entité du monde réel
  - L'état est représenté par l'ensemble des attributs de l'objet
  - Le comportement est un ensemble d'opérations permettant de manipuler cet état
- Les opérations associées à un objet fournissent des services aux autres objets
  - Les objets communiquent entre eux par l'envoie de messages
  - L'objet receveur du message va invoquer, exécuter une méthode ou opération pour satisfaire la demande

Abstraction de données

- L'abstraction de données est le pas décisif pour la représentation des informations en vue de l'interface utilisateur
- En C++, l'abstraction se traduit en un type défini par l'utilisateur à l'aide d'une classe
  - Membres données
  - Membres fonctions ou méthodes
- Pour un type défini, le compilateur possède un modèle pour l'emplacement mémoire à réservé et les opérations applicables

Exemple : la modélisation des dates

- La modélisation des dates amène à construire une classe Date qui contiendra les différentes définitions
- Les instances de la classe Date seront des objets particuliers
  - Hier
  - Aujourd'hui
  - Noël
  - Le 21 décembre 2012



Attributs (état)

Méthodes (comportement)

Encapsulation de membres

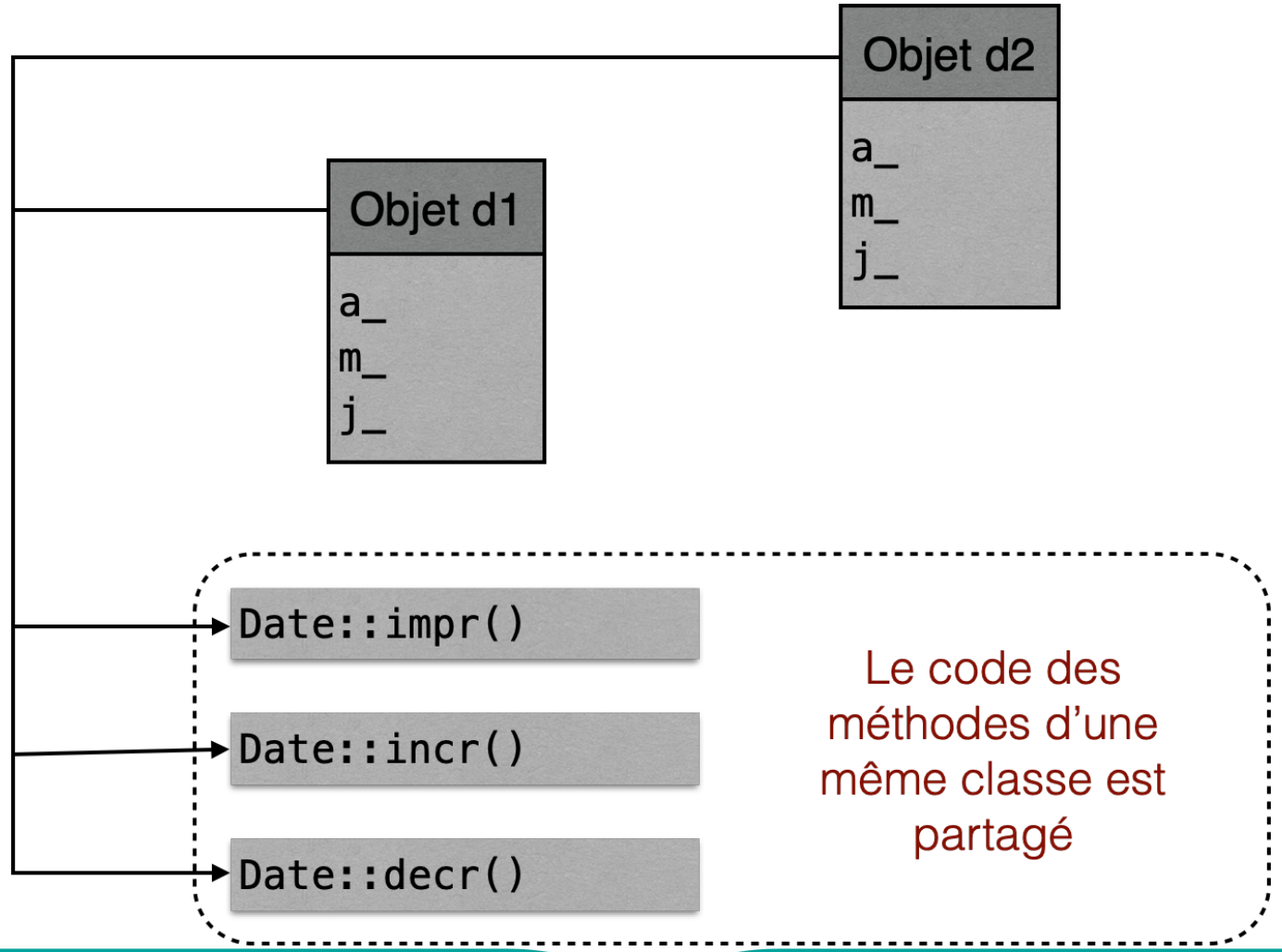
- La déclaration d'une classe permet de définir un type abstrait de données
- Les mots-clés **private**, **protected** et **public** règlent la visibilité des membres de la classe
- Les membres privés ne sont accessibles qu'à l'intérieur de la classe, ceci permet de cacher la structure interne de l'objet
- Le mécanisme qui empêche l'accès direct à des membres privés s'appelle l'*encapsulation*

Exemple : la déclaration de la classe Date

```
class Date {
public:                                  // Les membres publics
        Date();
        Date(const Date &);
        Date(int, int, int);
        int jour() const;
        void jour(int);
        void impr() const;               // Imprimer la date sur l'écran
        Date &incr();                    // Incrément de date en retournant l'objet
        Date &decr();                    // Décrément de date en retournant l'objet
private:                                 // Les membres privés
        int a_;                          // Année
        int m_;                          // Mois
        int j_;                          // Jour
};

// Cette classe se déclare dans Date.h
```

Mise en place des objets



Décomposition en fichiers de classe

- Tous les programmes complexes sont constitués de plusieurs unités compilées séparément appelés fichiers
- Dans la pratique, on sépare la déclaration d'une classe et sa définition (implémentation des fonctions) en deux fichiers
- La déclaration et l'implémentation d'une classe peuvent se délivrer séparément
  - Un fichier en-tête contenant la déclaration de la classe
  - Un fichier objet résultant de la compilation du fichier de définition de la classe (autrement dit, une bibliothèque)

Exemple : Date.h

```
class Date {
public:
        Date();
        Date(const Date &);
        Date(int, int, int);
        int jour() const;
        void jour(int);
        void impr() const;
        Date &incr();
        Date &decr();
private:
        int a_;
        int m_;
        int j_;
};
```

Exemple : Date.cpp

```
#include "Date.h”

Date::Date() {
        // ...
}

Date::Date(const Date &d) {
        // ...
}

void Date::impr() const {
        // ...
}

// ...
```

Le pointeur this

- Dans un objet d'une classe, il existe une déclaration implicite d'un pointeur constant this qui possède l'adresse de l'objet
- L'objet lui même est représenté par *this

Exemple

```
Date &Date::incr() {
        // Incrémenter la date,
        // puis retourner l'objet lui-même
        return *this;
}


int Date::jour() const {

        return this->j_; // return j_;

}
```

Les constructeurs

- Quand la représentation d'un type (notamment une classe) est cachée, des mécanismes doivent être fournis pour initialiser les attributs de ce types
- On appelle *constructeur* une fonction d'initialisation, appelée lors de la création de l'objet, portant le même nom que la classe
  - Une fonction membre de la classe sans type
  - Généralement publique mais peut être privée
- Une classe peut contenir plusieurs constructeurs

Exemple : le constructeur par défaut

```
class Date {
public:
        Date();
        // Autres membres publics et privés
};

Date::Date() {                                  // Style classique
        a_ = 2012;
        m_ = 12;
        j_ = 21;
}

// ou

Date::Date() : a_(2012), m_(12), j_(21) {}    // Style C++, plus efficace
```

Exemple : le constructeur de copie

```
class Date {
public:
        Date();
        Date(const Date &);
        // Autres membres publics et privés
};

Date::Date(const Date &d) {
        a_ = d.a_;
        m_ = d.m_;
        j_ = d.j_;
}

// ou

Date::Date(const Date &d) : a_(d.a_), m_(d.m_), j_(d.j_) {}
```

Exemple : les constructeurs avec paramètres

```
class Date {
public:
        // Autres constructeurs
        Date(int, int, int);
        // Autres membres publics et privés
};

Date::Date(int a, int m, int j) {
        a_ = a;
        m_ = m;
        j_ = j;
}

// ou

Date::Date(int a, int m, int j) : a_(a), m_(m), j_(j) {}
```

Le destructeur

- Lorsque la durée de vie d'un objet d'une classe se termine, le programme appelle automatiquement une fonction s'appelle destructeur qui détruit l'objet
- Le *destructeur* est identifié par le nom de la classe précédé de ~
- Un destructeur n'a ni type ni paramètre
- La définition d'un destructeur n'est pas obligatoire

Exemple : un tableau dynamique de dates

```
class DateArray {
public:
        DateArray(size_t n) : size_(n) {
                dates_ = new Date[n];                   // Créer un objet dynamique
        }
        ~DateArray() {
                delete[] dates_;                        // Supprimer l'objet dynamique
        }
        size_t size() const {
                return size_;
        }
private:
        Date *dates_;
        size_t size_;
};
```

Les fonctions et les classes amies

- Une *fonction amie* d'une classe est une fonction qui ne fait pas partie de cette classe mais qui a le droit d'accéder aux membres privés de celle-ci
- Le prototype d'une fonction amie doit être indiqué dans la classe dont elle est l'amie, précédé du mot-clé **friend**
- L'usage le plus fréquent d'une fonction amie se produit lorsqu'une fonction doit accéder aux éléments de deux classes différentes
- Il existe également la possibilité de déclarer une méthode d'une *classe amie* d'une autre classe

Exemple : le numéro de semaine d'une date (1)

```
///// Date.h /////

class Date {

        friend int semaine(const Date &);
public:
        // Autres membres publics
private:

        // Les membres privés
};
```

Exemple : le numéro de semaine d'une date (2)

```
///// semaine.cpp /////

int semaine(const Date &d) {
        int sem = 0;
        // Possible d'accéder à tout membre privé de la classe Date
        return sem;
}
```

Les membres statiques

- Un membre donnée déclaré avec l'attribut static est partagé par tous les objets de la même classe
- Un membre donnée statique est initialisé par défaut à 0, il doit être initialisé explicitement à l'extérieur de la classe
- Lorsqu'une fonction membre d'une classe a une action indépendante d'un quelconque objet de cette classe, on peut la déclarer **static**
- Une méthode statique ne peut pas accéder au pointeur **this**
- Exemple classique : le compteur d'instances

Exemple : incrémenter le compteur d'instances (1)

```
///// Date.h /////

class Date {
public:
        Date();
        // Autres membres publics
        static size_t count;
private:

        // Les membres privés
};
```

Exemple : incrémenter le compteur d'instances (2)

```
///// Date.cpp /////

size_t Date::count = 0;

Date::Date() {
        // Autres instructions
        ++count;
}
```

Les classes dérivées

- La technique d'héritage simplifie la création d'une nouvelle classe
- Une classe dérivée hérite toutes les fonctions et données membres de sa classe de base
- Une classe dérivée peut se différencier de sa classe de base par
  - l'ajout de membres données
  - l'ajout de fonctions membres
  - la redéfinition de fonctions membres héritées de la classe de base

Accès aux membres

- Les fonctions membres d'une classe dérivée n'ont pas accès aux membres privés de la classe de base, bien que ceux-ci aient été hérités
- Une instance d'une classe dérivée peut invoquer une méthode publique ou protégée de la classe de base

Exemple: accès aux membres

```
class A {
public:
        void set(char x);
};
```

```
class B: public A {
        // Sans méthode set()
};

B b;
b.set('B'); // A::set()
```

Redéfinition de membres

- Une classe dérivée peut redéfinir une fonction membre de la classe de base
- Si un objet d'une classe dérivée invoque une fonction membre redéfinie, c'est la fonction redéfinie dans cette classe dérivée qui sera utilisée
- Lorsque l'on désire appeler la version définie dans la classe de base, il suffit de faire précéder le nom de la fonction par le nom de la classe de base suivi de **::**

Exemple : redéfinition de membres

```
void B::set(char x) {
        if (x >= 'A' && x <= ‘z’) {
                A::set(x);
        }
        // Possible d’ajouter d’autres instructions
}
```

Les constructeurs dans les classes dérivées

- Le constructeur (par défaut) d'une classe de base doit être appelé par le constructeur de la classe dérivée (pourquoi ?)
- En cas général, la définition du constructeur de la classe dérivée sera suivi de : et d'un initialiseur de classe de base
- Le compilateur exécute d'abord le corps du constructeur de la classe de base puis celui de la classé dérivée

Exemple : les constructeurs dérivés

```
class A {
public:
        A(char x);
};
```

```
class B: public A {
public:
        B(char x) : A(x) {
                // ...
        }
};
```

Les destructeurs dans les classes dérivées

- Les destructeurs sont appelées dans l'ordre inverse des constructeurs
- Un destructeur de classe dérivée sera exécuté avant le destructeur de la classe de base

Exemple : les destructeurs dérivées

```
class B : public A {
public:
// ...
        virtual ~B() {
                delete z;
        }
// ...
};
```

```
class A {
public:
// ...
        virtual ~A() {
                delete y;
        }
// ...
};
```

Les fonctions virtuelles

- Une *fonction virtuelle* est une fonction déclarée **virtual** dans la classe de base et redéfinie dans les classe dérivées
- Le mot réservé **virtual** se place devant le prototype de la fonction, uniquement dans la déclaration de la classe de base
- Le mécanisme mis en place par les fonctions virtuelles permettra d'exécuter la version redéfinie d'une méthode correspondant au type d'objet

Destruction d'objets dynamique

- Si **delete** est appliqué à un pointeur sur la classe de base, le compilateur appelle le destructeur de la classe dérivée, même si le pointeur pointe sue une instance de la classe dérivée
  - Raison : la liaison statique
  - Solution : la liaison dynamique (fonction virtuelle)

Classe abstraite

- Une *fonction virtuelle pure* se déclare par l'ajout **= 0** à la fin du prototype d'un fonction virtuelle
- Une classe contenant une déclaration de fonction virtuelle pure est appelée *classe abstraite*
  - Il est possible de déclarer des pointeurs sur une classe abstraite
  - Il est fréquent d'écrire une hiérarchie de classes consistant en une ou plusieurs classes abstraites comme classes de base

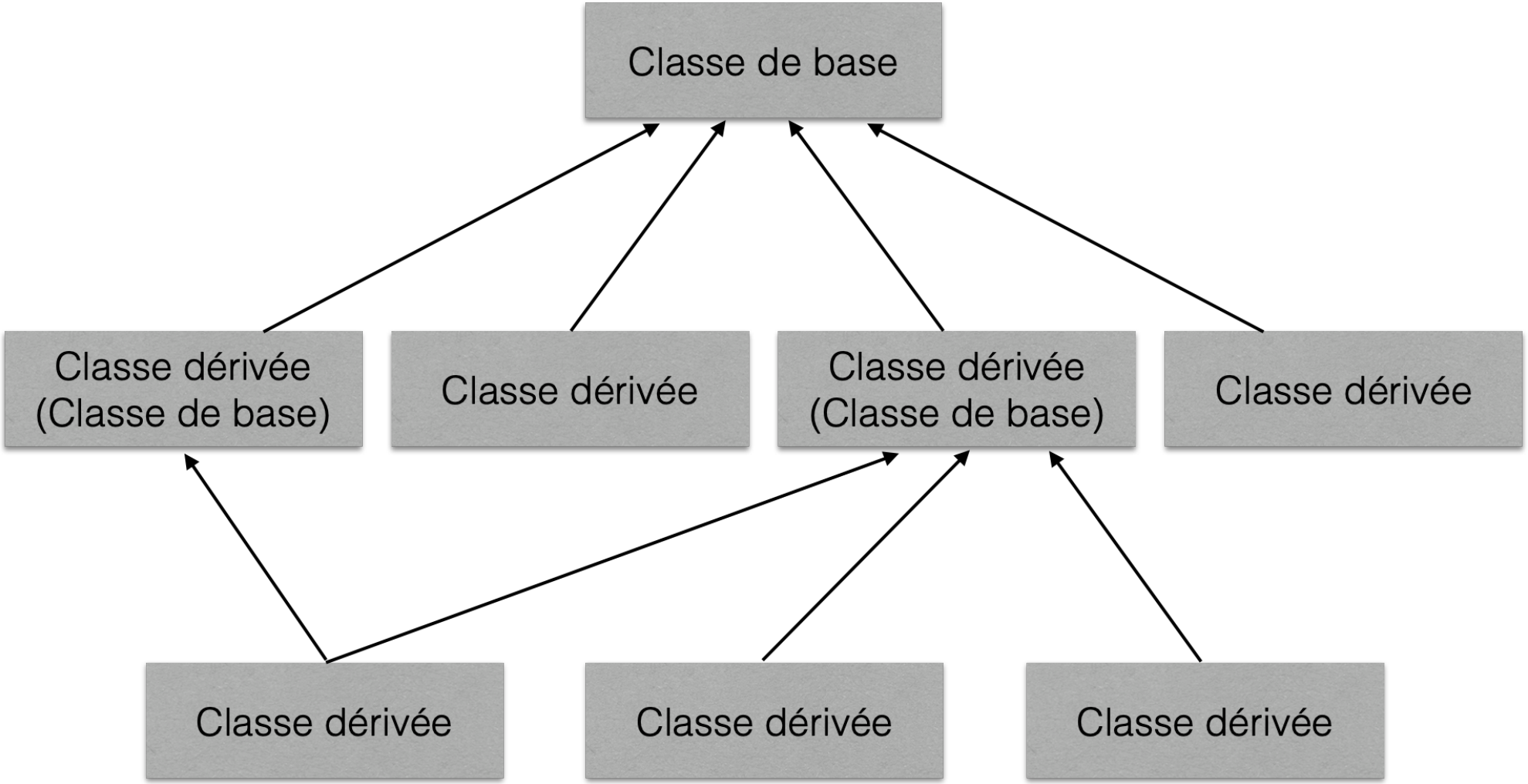
Exemple : classe interface en C++

```
class X {
public:
        virtual void set(char) = 0;
        virtual void unset() = 0;
};

class Y : public X {
public:
        void set(char x);
        void unset();
};

X xo;                   // Faux !
X *xp;
xp = new X;             // Faux !
xp = new Y;
```

Hiérarchie de classes



Héritage multiple

- C++ supporte de l'*héritage multiple* permettant de créer une classe dérivée à partir de plusieurs classes de base
- La classe hérite des membres de toutes ses classes de base
- Le constructeur de la classe dérivée doit être exécuté selon l'ordre de la déclaration d'héritage
- Une classe de base indirecte peut être incluse plus d'une fois et les membres de la classe de base indirecte seront ainsi inclus plusieurs fois

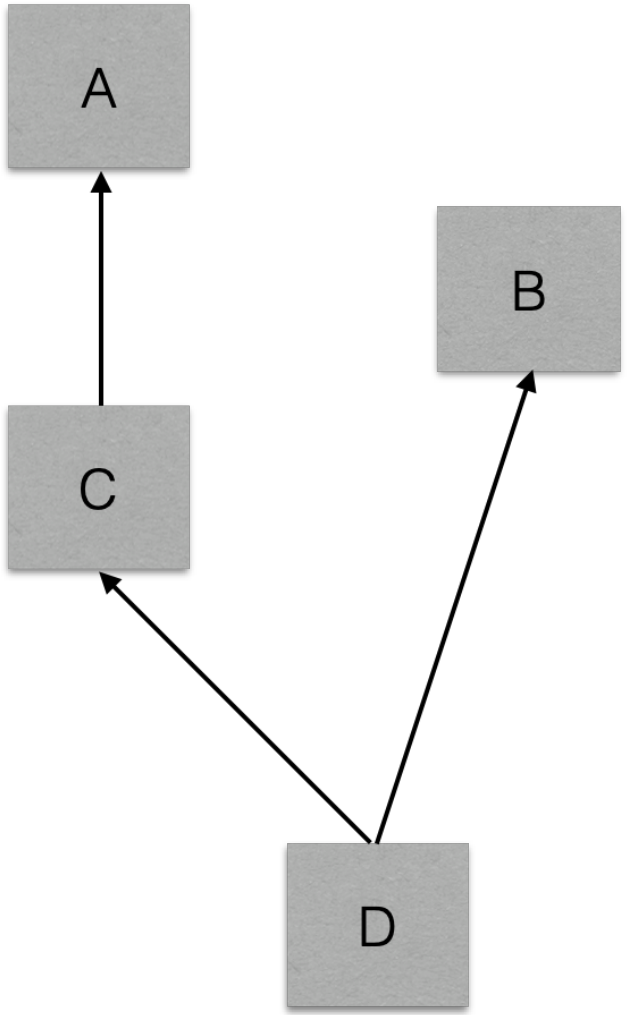
Déclaration d'héritage multiple

```
class A { // ...
};

class B { // ...
};

class C: public A { // ...
};

class D: public C, public B { // ...
};
```



Classe de base indirecte

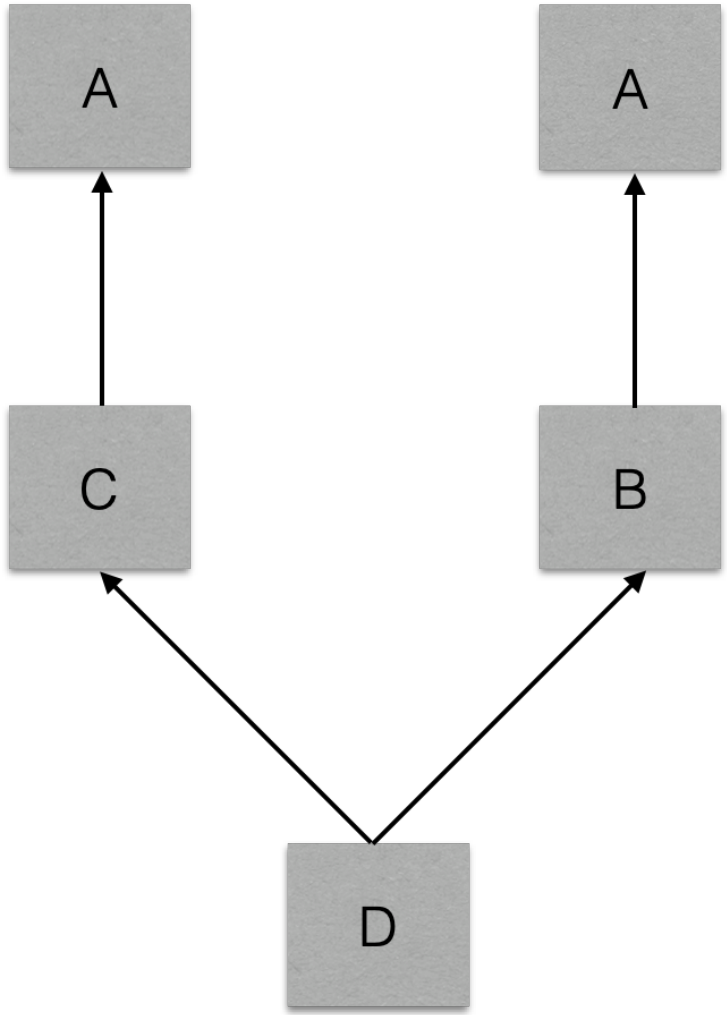
```
class A {
public:
        void set(char x);
};

class B: public A { // ...
};

class C: public A { // ...
};

class D: public B, public C { // ...
};

D d;
d.set('D');      // Erreur : ambigüe !
d::B.set('D');   // OK
```



Classe virtuelle

- Pour éviter la duplication des membres de la classe de base indirecte lors d'héritage multiple, on déclare cette classe de base **virtual** dans la déclaration d'héritage
- Une *classe de base virtuelle* fournit un moyen de partager de l'information
- La propriété de virtualité est une caractéristique de la dérivation, et non de la classe virtuelle
- Une classe dérivée peut avoir des classes de base à la fois virtuelles et non virtuelles

Classe de base virtuelle

```
class A {
public:
        void set(char x);
};

class B: virtual public A { // ...
};

class C: virtual public A { // ...
};

class D: public B, public C { // ...
};

D d;
d.set('D');      // OK
d::B.set('D');   // OK
```