L'héritage

Introduction

L'héritage est un des piliers du C++ et de la Programmation Orientée Objets en général.

Il vise la **récupération du code** développé pour un type d'objet donné en vue d'en **créer un nouveau type** semblable au premier mais en le spécialisant davantage ou l'**étendant au niveau de ses données et de ses fonctionnalités**.

Exemple :

Les clients, mécaniciens, vendeurs, etc… d'un concessionnaire automobile ont des caractéristiques communes :

- Un nom
- Un prénom
- Une date de naissance
- Un sexe
- Une adresse, etc…

mais un client possède en plus

- Un numéro de client

qu'un mécanicien n'a pas. Au contraire, celui-ci possède

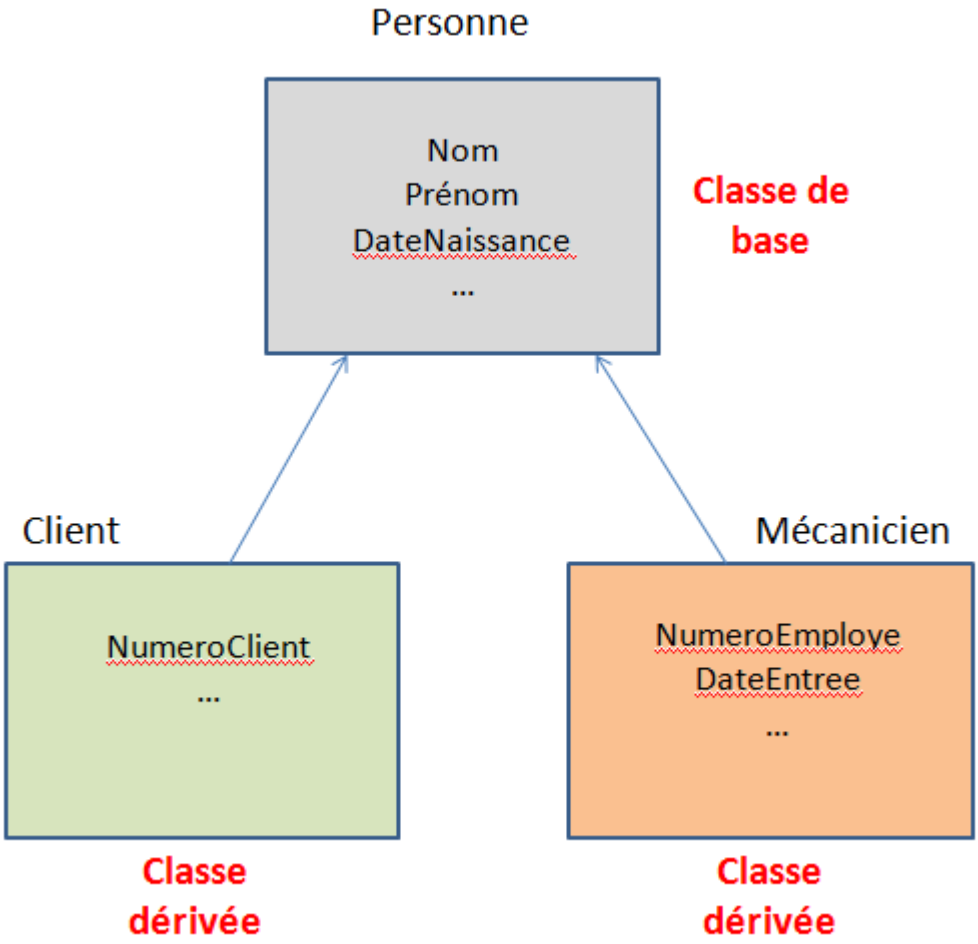
- Une date d'entrée en service
- Un numéro d'employé

que le client n'a pas.

En conclusion, on peut dire que ces deux entités ont des **propriétés communes** en tant que « personne » et des **propriétés différentes** selon leur rôle joué dans le fonctionnement du concessionnaire.

L'héritage permet de ne pas devoir écrire deux fois la même chose en ce qui concerne la partie commune des deux classes.

Diagramme de classes :



Une classe de base : la classe Personne

Imaginons une classe **Personne** reprenant quelques renseignements concernant toutes les personnes, quel que soit leur rôle dans la société :

Personne.h :

```
#ifndef PERSONNE_H
#define PERSONNE_H

#include <iostream>
#include <string>
using namespace std;

class Personne {
  private:
        string nom;
        string prenom;
        string dateDeNaissance;
        char sexe;

  public:
        Personne();
        Personne(string, string, string, char);
        Personne(const Personne&);
        ~Personne();

        void affiche() const;
};

#endif
```

Personne.cpp :

```
#include "Personne.h"

Personne::Personne() {
        nom = prenom = "---";
        dateDeNaissance = "--/--/----";
        sexe = 'M';
}

Personne::Personne(string nom, string prenom, string dateDeNaissance,
char sexe) {
        this->nom = nom;
        this->prenom = prenom;
        this->dateDeNaissance = dateDeNaissance;
        this->sexe = sexe;
}
```

```
Personne::Personne(const Personne& p) {
        this->nom = p.nom;
        this->prenom = p.prenom;
        this->dateDeNaissance = p.dateDeNaissance;
        this->sexe = p.sexe;
}

Personne::~Personne() {
}

void Personne::affiche() const {
        cout << "nom=" << this->nom
             << " prenom=" << this->prenom
             << " dateDeNaissance=" << this->dateDeNaissance
             << " sexe=" << this->sexe << endl;
}
```

Exemple de programme utilisant cette classe (**main1.cpp**) :

```
#include <iostream>
using namespace std;
#include "Personne.h"

int main() {
  Personne p1,
           p2("Wagner","Jean-Marc","07/02/1974",'M'),
           p3(p2);

  p1.affiche();
  p2.affiche();
  p3.affiche();

  Personne * personnes[] = { &p1, &p2, &p3 };

  cout << "***** Les personnes : *****" << endl;
  for (int i=0; i<3 ; i++)
    personnes[i]->affiche();

  return 0;
}
```

dont la compilation et l'exécution fournissent

```
[student@moon]$ g++ Personne.cpp -c
[student@moon]$ g++ main1.cpp Personne.o -o main1
[student@moon]$ main1
nom=--- prenom=--- dateDeNaissance=--/--/---- sexe=M
nom=Wagner prenom=Jean-Marc dateDeNaissance=07/02/1974 sexe=M
nom=Wagner prenom=Jean-Marc dateDeNaissance=07/02/1974 sexe=M
***** Les personnes : *****
```

```
nom=--- prenom=--- dateDeNaissance=--/--/---- sexe=M
nom=Wagner prenom=Jean-Marc dateDeNaissance=07/02/1974 sexe=M
nom=Wagner prenom=Jean-Marc dateDeNaissance=07/02/1974 sexe=M
[student@moon]$
```

Une classe dérivée : la classe Client

La classe développée au point précédent comprend les caractéristiques d'une personne.

Or, certaines personnes ont des rôles/métiers précis dans la société :

- Des clients
- Des mécaniciens
- etc…

A chacun de ces types va correspondre une classe d'objets. Cette classe héritera des propriétés communes à toutes les personnes en spécifiant en plus les caractéristiques supplémentaires propres au type d'objets qu'elle représente.

Dès lors, prenons l'exemple de la classe « **Client** » qui va hériter de la classe « **Personne** » → on dit qu'elle va **spécialiser** la classe **Personne**

Vocabulaire :

La classe Personne est **la classe de base** (ou la **classe mère**) de Client.

La classe Client est **une classe dérivée** (ou une **classe fille**) de la classe Personne.

Syntaxe :

```
class classe_derivee : public classe_de_base
```

Ce qui donnera dans notre exemple pour lequel un client est simplement une personne qui dispose en plus d'un numéro de client :

Client.h :

```
#ifndef CLIENT_H
#define CLIENT_H

#include <iostream>
#include <string>
using namespace std;
#include "Personne.h"
```

```
class Client : public Personne {
  private:
        int numClient;

  public:
        Client();
        Client(string, string, string, char, int);
        Client(const Client&);
        ~Client();

        void setNumClient(int);
};

#endif
```

Client.cpp :

```
#include "Client.h"

Client::Client() {
        nom = prenom = "---";
        dateDeNaissance = "--/--/----";
        sexe = 'M';
        numClient = 1;
}

Client::Client(string nom, string prenom, string dateDeNaissance, char
sexe, int numClient) {
        this->nom = nom;
        this->prenom = prenom;
        this->dateDeNaissance = dateDeNaissance;
        this->sexe = sexe;
        this->numClient = numClient;
}

Client::Client(const Client& c) {
        this->nom = c.nom;
        this->prenom = c.prenom;
        this->dateDeNaissance = c.dateDeNaissance;
        this->sexe = c.sexe;
        this->numClient = c.numClient;
}

Client::~Client() {
}

void Client::setNumClient(int numClient) {
        this->numClient = numClient;
}
```

La déclaration **« class Client : public Personne »** signifie qu'**un objet Client est un objet Personne avec des membres supplémentaires**.

Un objet instanciant **Client** possède donc :

- 5 variables membres (4 accessibles au sein de **Personne** et 1 propre à **Client**)
- Hormi les constructeurs, 2 fonctions membres : **affiche** et **setNumClient**

Cependant, le compilateur n'est pas d'accord ☹… Les fonctions membres de la classe Client ne peuvent accéder aux variables membres de la classe Personne…

```
[student@moon]$ g++ Client.cpp -c
Client.cpp: Dans le constructeur « Client::Client() »:
Client.cpp:4:2: error: « std::__cxx11::string Personne::nom » est privé
dans ce contexte
  nom = prenom = "---";
  ^~~
In file included from Client.h:4,
                 from Client.cpp:1:
Personne.h:7:11: note: déclaré privé ici
    string nom;
           ^~~
Client.cpp:4:8:  error:  « std::__cxx11::string  Personne::prenom »  est
privé dans ce contexte
  nom = prenom = "---";
…
[student@moon]$
```

Pourquoi ? Parce que toutes les variables membres de la classe **Personne** ont été déclarés comme **privés** à cette classe. Des membres privés ne sont accessibles qu'à partir de la classe à laquelle ils appartiennent.

Pour que les classes dérivées puissent y accéder, il faut les déclarer « **protected** ». Ainsi, les variables membres sont protégées du monde extérieur mais sont accessibles par les classes qui vont hériter de la classe de base.

Une variable ou une fonction membre protégée (**protected**) est **privée**, mais est **accessible à toutes les classes dérivées** (du moins dans un processus de dérivation public).

Récapitulatif sur les types d'accès aux membres d'une classe :

Type d'accès	Monde extérieur	Classes dérivées
public	OUI	OUI
private	NON	NON
protected	NON	OUI

Notre classe Personne devient (fichier Personne.h) :

```
#ifndef PERSONNE_H
#define PERSONNE_H

#include <iostream>
#include <string>
using namespace std;

class Personne {
  protected:
        string nom;
        string prenom;
        string dateDeNaissance;
        char sexe;

  public:
        Personne();
        Personne(string, string, string, char);
        Personne(const Personne&);
        ~Personne();

        void affiche() const;
};

#endif
```

Tandis que la classe Client est inchangée → le fichier **Client.cpp** compile à présent sans erreur.

Comment savoir si un objet doit être dérivé d'une autre ou s'il doit être un membre d'une autre ?

Une classe D **hérite** d'une classe B **si D est un B** avec des **éléments supplémentaires**.

Exemple : les classes **Personne** et **Client**. → un client est une personne

Une classe M est **un membre de** classe la classe B si **B contient un M**, entre autres…

Exemple : Une classe **Date** (de naissance ici) est un membre d'une classe **Personne** → une personne n'est pas une date, elle possède une date (de naissance)

Les constructeurs et le destructeur d'une classe dérivée

Puisqu'une classe D dérivée d'une classe B est un B avec des membres en plus, cette classe possède sa propre batterie de constructeur(s) et son propre destructeur.

Les constructeurs

Afin d'initialiser les membres définis dans la classe dérivée, on redéfinit une série de constructeurs au sein de la classe dérivée. Ils effectuent deux opérations :

- Initialiser les variables membres de la classe de base en appelant le constructeur ad hoc de celle-ci → **réutilisation du code**
- Initialiser les variables membres définis au sein de la classe dérivée.

Actuellement, dans le code ci-dessus, il n'y a aucune réutilisation du code ! → les constructeurs de la classe **Client** sont une recopie quasi conforme des constructeurs de la classe **Personne** → nous remédions à cela ci-dessous

La syntaxe est la suivante :

```
constructeur_classe_derivee : constructeur_classe_de_base
```

Dans notre exemple, le fichier **Client.h** restant inchangé, la classe **Client** devient (fichier **Client.cpp**) :

```
#include "Client.h"

Client::Client():Personne() {
        numClient = 1;
}

Client::Client(string nom,
                    string prenom,
                    string dateDeNaissance,
                    char sexe,
int numClient):Personne(nom,prenom,dateDeNaissance,sexe) {
        this->numClient = numClient;
}

Client::Client(const Client& c):Personne(c) {
        this->numClient = c.numClient;
}

Client::~Client() {
}

void Client::setNumClient(int numClient) {
        this->numClient = numClient;
}
```

où on observe que

- Le constructeur par défaut de la classe **Client** appelle le constructeur par défaut de la classe **Personne** avant d'entrer dans son corps de fonction
- Le constructeur d'initialisation de la classe **Client** appelle le constructeur d'initialisation de la classe **Personne** avant d'entrer dans son corps de fonction
- Le constructeur de copie de la classe **Client** appelle le constructeur de copie de la classe **Personne** avant d'entrer dans son corps de fonction
- Les contenus des constructeurs de la classe Client ne s'occupent que d'initialiser les variables supplémentaires !!! → ici la variable **numClient**

Remarque importante :

En l'absence de constructeur au sein de la classe dérivée, celle-ci hérite, implicitement, du **constructeur par défaut** et du **constructeur de copie** de la classe de base, **pas des constructeurs d'initialisation**.

Le destructeur

La gestion du **destructeur** s'effectue dans le même ordre d'idée.

Le destructeur de la classe dérivée fait appel au destructeur de la classe de base. Mais cet appel s'effectue après qu'il ait détruit les variables membres définies au sein de la classe dérivée.

Lors de la destruction d'un objet instanciant une classe dérivée, **les opérations s'effectuent dans l'ordre inverse** par rapport à la construction de l'objet.

Exemple d'utilisation de la classe Client (main2.cpp) :

```
#include <iostream>
using namespace std;
#include "Personne.h"
#include "Client.h"

int main() {
  Personne p1,
           p2("Wagner","Jean-Marc","07/02/1974",'M'),
           p3(p2);

  Client c1,
         c2("Smith","Sarah","25/03/1990",'F',112),
         c3(c2);

  p1.affiche();
  p2.affiche();
  p3.affiche();
  c1.affiche();
```

```
  c2.affiche();
  c3.affiche();

  return 0;
}
```

dont la compilation et l'exécution fournit :

```
[student@moon]$ g++ Personne.cpp -c
[student@moon]$ g++ Client.cpp -c
[student@moon]$ g++ main2.cpp Personne.o Client.o -o main2
[student@moon]$ main2
nom=--- prenom=--- dateDeNaissance=--/--/---- sexe=M
nom=Wagner prenom=Jean-Marc dateDeNaissance=07/02/1974 sexe=M
nom=Wagner prenom=Jean-Marc dateDeNaissance=07/02/1974 sexe=M
nom=--- prenom=--- dateDeNaissance=--/--/---- sexe=M
nom=Smith prenom=Sarah dateDeNaissance=25/03/1990 sexe=F
nom=Smith prenom=Sarah dateDeNaissance=25/03/1990 sexe=F
[student@moon]$
```

où on observe que

- La classe **Client** a hérité de la méthode **affiche()** de la classe **Personne** → c'est l'utilité principale de l'héritage !
- Cependant, les clients sont plus que des personnes et leur affichage par la fonction **affiche()** héritée les affichent en tant que personne ! → Quid ? → **redéfinition** (**« surcharge »** ou **« overriding »**) des méthodes héritées

La redéfinition des méthodes héritées

Dans l'état actuel des choses, les objets du type **Personne** et **Client** peuvent faire appel à la méthode **affiche()**. Celle-ci

- est définie au sein de la classe **Personne**
- affiche les caractéristiques d'une personne sans qualification particulière (c'est-à-dire non spécialisé).

Pour pouvoir afficher l'ensemble des caractéristiques d'un client, il faudrait spécialiser la méthode **affiche()** au sein de la classe **Client**.

Cela revient à dire qu'il faut redéfinir la méthode **affiche()** au sein de la classe **Client** → On parle d' ***overriding***.

Ce qui donnera (**Client.h**) :

```
#ifndef CLIENT_H
#define CLIENT_H

#include <iostream>
#include <string>
using namespace std;
#include "Personne.h"

class Client : public Personne {
  private:
        int numClient;

  public:
        Client();
        Client(string, string, string, char, int);
        Client(const Client&);
        ~Client();

        void setNumClient(int);
        void affiche() const; // redéfinition de la méthode affiche()
};

#endif
```

et (fichier **Client.cpp**) :

```
#include "Client.h"

Client::Client():Personne() {
        numClient = 1;
}

…

void Client::affiche() const {
        cout << "Client: ";
        cout << "nom=" << this->nom
             << " prenom=" << this->prenom
             << " dateDeNaissance=" << this->dateDeNaissance
             << " sexe=" << this->sexe << endl;
```

```
        cout << "avec NumClient=" << this->numClient << endl;
}
```

Notons que, pour respecter le principe de la réutilisation du code, on aurait pu écrire (fichier **Client.cpp**) :

```
#include "Client.h"

Client::Client():Personne() {
        numClient = 1;
}

…

void Client::affiche() const {
        cout << "Client: ";
        Personne::affiche(); // appel de la méthode affiche()
                             // de la classe de base
        cout << "avec NumClient=" << this->numClient << endl;
}
```

où **Personne::affiche()** permet d'appeler, dans la méthode **affiche()** de la classe **Client**, la méthode **affiche()** de la classe mère **Personne**.

La compilation et l'exécution de **main2.cpp** fournissent alors

```
[student@moon]$ g++ Client.cpp -c
[student@moon]$ g++ main2.cpp Personne.o Client.o -o main2
[student@moon]$ main2
nom=--- prenom=--- dateDeNaissance=--/--/---- sexe=M
nom=Wagner prenom=Jean-Marc dateDeNaissance=07/02/1974 sexe=M
nom=Wagner prenom=Jean-Marc dateDeNaissance=07/02/1974 sexe=M
Client: nom=--- prenom=--- dateDeNaissance=--/--/---- sexe=M
avec NumClient=1
Client: nom=Smith prenom=Sarah dateDeNaissance=25/03/1990 sexe=F
avec NumClient=112
Client: nom=Smith prenom=Sarah dateDeNaissance=25/03/1990 sexe=F
avec NumClient=112
[student@moon Heritage]$
```

La bonne méthode **affiche()** est appelée en fonction du type de l'objet.

Les méthodes virtuelles

Une méthode virtuelle est une méthode dont la résolution de l'appel s'effectue **à l'exécution** et **pas lors de la compilation** du programme.

A quoi cela sert-il ?

Dans les cas où le type des objets est « clairement » défini, comme dans le cas de **main2.cpp** ci-dessus (recopié ici) :

```
#include <iostream>
using namespace std;
#include "Personne.h"
#include "Client.h"

int main() {
  Personne p1,
           p2("Wagner","Jean-Marc","07/02/1974",'M'),
           p3(p2);

  Client c1,
         c2("Smith","Sarah","25/03/1990",'F',112),
         c3(c2);

  p1.affiche();
  p2.affiche();
  p3.affiche();
  c1.affiche();
  c2.affiche();
  c3.affiche();

  return 0;
}
```

→ cela ne sert à rien car on connaît, on est sûr de, la nature exacte des objets.

Par contre, quand on a un pointeur d'objets, il se peut qu'on ne puisse pas définir avec exactitude, lors de l'édition de liens, le type d'objet.

Exemple :

Considérons le fichier **main3.cpp** suivant

```
#include <iostream>
using namespace std;
#include "Personne.h"
#include "Client.h"

int main() {
  Personne p1,
           p2("Wagner","Jean-Marc","07/02/1974",'M'),
           p3(p2);

  Client c1,
         c2("Smith","Sarah","25/03/1990",'F',112),
         c3(c2);

  Personne * personnes[] = { &p1, &c2, &c3, &p2, &c1, &p3 };

  cout << "***** Appel de la methode affiche() *****" << endl;
  for (int i=0; i<6 ; i++)
    personnes[i]->affiche();

  return 0;
}
```

dont la compilation et l'exécution fournissent

```
[student@moon]$ g++ Personne.cpp -c
[student@moon]$ g++ Client.cpp -c
[student@moon]$ g++ main3.cpp Personne.o Client.o -o main3
[student@moon]$ main3
***** Appel de la methode affiche() *****
nom=--- prenom=--- dateDeNaissance=--/--/---- sexe=M
nom=Smith prenom=Sarah dateDeNaissance=25/03/1990 sexe=F
nom=Smith prenom=Sarah dateDeNaissance=25/03/1990 sexe=F
nom=Wagner prenom=Jean-Marc dateDeNaissance=07/02/1974 sexe=M
nom=--- prenom=--- dateDeNaissance=--/--/---- sexe=M
nom=Wagner prenom=Jean-Marc dateDeNaissance=07/02/1974 sexe=M
[student@moon]$
```

Dans l'état actuel des choses, le programme **affichera les caractéristiques de base** d'une personne pour toutes les personnes → appel de la méthode **affiche()** de la classe **Personne** → ce n'est bien sûr pas ce que l'on veut !

Ceci est dû au fait que le vecteur de personnes a été déclaré comme étant un vecteur de « Personne ». **Par défaut**, la **résolution de l'appel** de fonction s'effectue **lors de la compilation du programme** → « **c'est le type du pointeur et non la vraie nature de l'objet qui gagne** »

Pour que cela se passe correctement, il faut repousser la **résolution de l'appel à l'exécution**. C'est-à-dire que c'est au moment où on exécutera la ligne

```
personnes[i]->affiche() ;
```

que le programme devra se poser la question de savoir de quel type est exactement l'objet pointé par **personnes[i]**. C'est le **principe des méthodes virtuelles**.

Déclaration d'une méthode virtuelle

Pour déclarer une méthode comme étant virtuelle, il suffit de la faire précéder du mot réservé « **virtual** », et cela dans la classe mère et uniquement dans le fichier .h. Dans notre exemple, la classe **Personne** (fichier **Personne.h**) devient

```
#ifndef PERSONNE_H
#define PERSONNE_H

#include <iostream>
#include <string>
using namespace std;

class Personne {
  protected:
        string nom;
        string prenom;
        string dateDeNaissance;
        char sexe;

  public:
        Personne();
        Personne(string, string, string, char);
        Personne(const Personne&);
        ~Personne();

        virtual void affiche() const;
};
```

```
#endif
```

Depuis la version 11 du C++, il est d'usage d'ajouter, dans les classes héritées, le décorateur **« override »** à toute méthode surchargée et **virtual** de la classe de base → le compilateur peut ainsi réaliser une vérification supplémentaire et s'assurer que la signature de la méthode surchargée est identique à la signature de la méthode déclarée virtual dans la classe de base.

Dès lors, notre classe **Client** (fichier **Client.h**) devient

```
#ifndef CLIENT_H
#define CLIENT_H

#include <iostream>
#include <string>
using namespace std;
#include "Personne.h"

class Client : public Personne {
  private:
      int numClient;

  public:
      Client();
      Client(string, string, string, char, int);
      Client(const Client&);
      ~Client();

      void setNumClient(int);
      void affiche() const override;
};

#endif
```

les fichiers **Personne.cpp** et **Client.cpp** restant inchangés.

Remarque importante

Le décorateur **« override »** ne peut se placer que sur une méthode déclarée **virtual** dans la classe de base. De plus, il ne se place que dans le fichier **.h** et non **.cpp**

Une **méthode virtuelle** est donc une méthode

- qui fait l'objet de **surcharges** dans un processus d'héritage de classes.
- dont la **résolution** des appels s'effectue **à l'exécution** et non **pas à la compilation** → « **c'est la nature de l'objet et non le type du pointeur qui gagne** »
- qui n'intervient que lorsque le type de l'objet n'est pas prévisible à l'avance, c'est-à-dire lorsque les objets sont référencés par des **pointeurs** du type de la classe de base

Remarques :

- Une fois une méthode déclarée virtuelle dans une classe de base, il n'est plus nécessaire de la rappeler dans les classes dérivées. Toutes les fonctions identiques surchargées dans les classes dérivées feront appel au mécanisme des fonctions virtuelles.

- Les différentes versions d'une méthode virtuelle doivent avoir le **même prototype** → l'utilisation du mot réservé **override** permet d'en demander la vérification au compilateur

A présent, la compilation de nos deux classes ainsi que **main3.cpp** et l'exécution fournissent

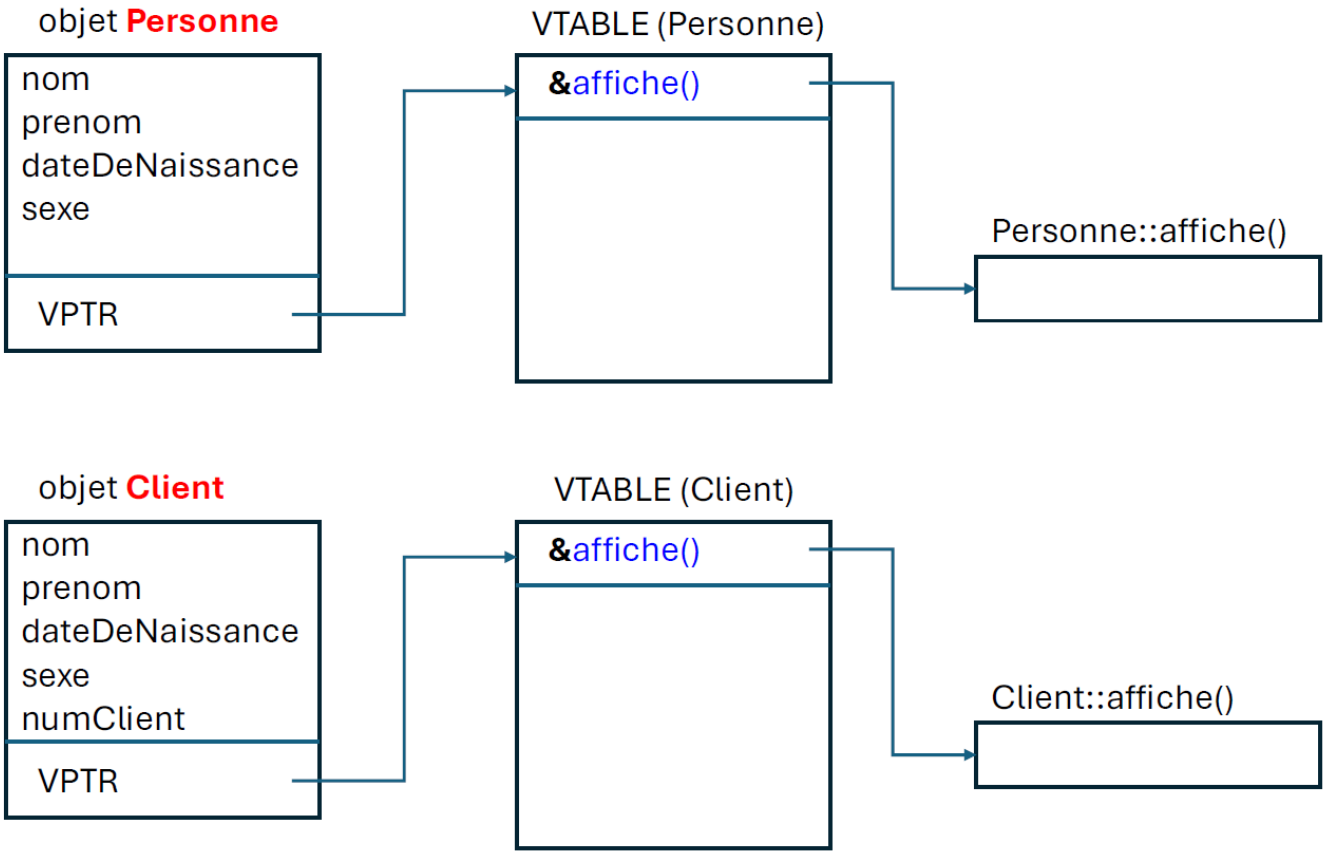
```
[student@moon Heritage]$ g++ Personne.cpp -c
[student@moon Heritage]$ g++ Client.cpp -c
[student@moon Heritage]$ g++ main3.cpp Personne.o Client.o -o main3
[student@moon Heritage]$ main3
***** Appel de la methode affiche() *****
nom=--- prenom=--- dateDeNaissance=--/--/---- sexe=M
Client: nom=Smith prenom=Sarah dateDeNaissance=25/03/1990 sexe=F
avec NumClient=112
Client: nom=Smith prenom=Sarah dateDeNaissance=25/03/1990 sexe=F
avec NumClient=112
nom=Wagner prenom=Jean-Marc dateDeNaissance=07/02/1974 sexe=M
Client: nom=--- prenom=--- dateDeNaissance=--/--/---- sexe=M
avec NumClient=1
nom=Wagner prenom=Jean-Marc dateDeNaissance=07/02/1974 sexe=M
[student@moon Heritage]$
```

où on observe que la « bonne méthode » **affiche()** est appelée pour le « bon objet »

Mécanisme des fonctions virtuelles

Comment la résolution de l'appel d'une fonction est-il reporté à l'exécution du programme ?

Le **compilateur ajoute**, **de manière implicite**, **une variable supplémentaire à la classe**. Il s'agit d'un pointeur **VPTR** qui pointe une entrée dans un tableau de pointeurs de fonctions, appelé **VTABLE**. **Il y a une VTABLE par classe**.



Lors de l'exécution de la ligne

```
personnes[i]->affiche() ;
```

1. Le pointeur **personnes[i]** indique où trouver l'objet à « traiter », lequel est soit un objet **Personne**, soit un objet **Client**.
2. Le pointeur **VPTR** de l'objet indique où trouver la **VTABLE** correspondante.
3. Le pointeur de fonction que l'on y trouve indique où trouver la méthode cherchée.

Remarques :

- Un **constructeur** ne peut être virtuel
- Le **destructeur** peut être virtuel

Upcasting et downcasting

C'est l'un des problèmes classiques du C++.

Cela revient à caster un objet comme étant un « objet » (→ plutôt un **pointeur**) d'une classe de base (**Upcasting**) ou d'une classe dérivée (**Downcasting**).

Soient la classe de base suivante :

```
class Base
{
  public:
    static int cptObjets;
    Base() {
      cout << ++cptObjets << ".* constructeur de Base" << endl;
    }

    virtual void f();
};

int base::cptObjets = 0;

void base::f() {
  cout << "fonction base" << endl;
}
```

et la classe dérivée suivante :

```
class Derivee : public Base
{
  public:
    Derivee() {
      cout << cptObjets << ".** constructeur de Derivee" << endl;
    }
    void f() override;
```

```
    void g() {
      cout << "fct add" << endl;
    }
};

void Derivee::f() {
  cout << "fonction derivee" << endl;
}
```

Caster vers le haut : Upcasting

Le programme suivant

```
int main()
{
  Derivee *pDerivee = new Derivee();
  Base *pBase;
  pBase = (Base*)pDerivee; // Upcasting

  pBase->f();

  delete pBase;
  return 0 ;
}
```

fournit l'exécution suivante

```
bash-3.00$ a.out
1.* constructeur de Base
1.** constructeur de Derivee
fonction derivee
bash-3.00$
```

On remarque que

```
pBase->f() ;
```

appelle la fonction de la classe dérivée car **la fonction f est virtuelle**.

On parle de « **Upcasting** » car un pointeur d'objet Derivee est casté en pointeur d'objet de base. Ce qui ne pose aucun problème dans la mesure où un objet « **Derivee** » est une « **Base** » avec des éléments en plus.

Remarque :

L'instruction

```
delete pBase ;
```

appellera le destructeur de la classe **Derivee** si celui-ci a été déclaré **virtuel** dans la classe **Base**.

Caster vers le bas : Downcasting

Le programme suivant

```
int main()
{
  Base *pBase = new Derivee();
  pBase->g(); // !!!!

  delete pBase;
  return 0;
}
```

ne compile pas ☹… En effet,

```
bash-3.00$ g++ Downcasting.cxx
Downcasting.cxx: In function `int main()':
Downcasting.cxx:45: error: 'class base' has no member named 'g'
bash-3.00$
```

Ceci est logique puisque d'un point de vue statique, **pBase** est censé être un pointeur vers un objet de type **Base**. Or, on sait qu'en fait il pointe vers un objet de type **Derivee**. On peut réaliser un **casting vers le bas** (**downcasting**) pour résoudre ce problème :

```
int main()
{
  Base *pBase = new Derivee();
  Derivee *pDerivee;

  pDerivee = (derivee*)pBase;  // Downcasting
  pDerivee->g();

  delete pBase;
  return 0;
}
```

dont l'exécution fournit

```
bash-3.00$ a.out
1.* constructeur de base
1.** constructeur de derivee
fct add
bash-3.00$
```

Mais il faut être extrêmement prudent avec les downcastings !!!

En effet, le compilateur nous fait confiance. Mais si on se trompe et qu'en réalité le pointeur **pBase** pointe réellement un objet de type **Base**, le programme se plantera ☹ !

Tester un downcasting

Il est possible de tester que le downcasting s'est bien déroulé avant d'appliquer à l'objet des méthodes propres à la classe dérivée.

Prenons l'exemple d'un tableau de pointeurs d'objets de type **Base** contenant en réalité un mélange de **Base** et de **Derivee** :

```
Base* pb[4] ;

for (int i=0 ; i<4 ; i++)
{
  int test = rand() ;
  if (test % 2) pb[i] = new Derivee;
  else pb[i] = new Base;
}
```

Comment savoir, lorsqu'on parcourt un tel vecteur, si l'objet sur lequel on pointe est un objet instanciant la classe **Base** ou la classe **Derivee** ?

Il existe en C++ un opérateur, faisant partie intégrante du langage, permettant d'effectuer un **casting de manière dynamique**. Sa syntaxe est la suivante :

```
dynamic_cast<classe *>(pointeur)
```

Exemple complet :

```
int main()
{
  Base * pb[4];
  for (int i=0; i<4; i++)
  {
    int test=rand(); cout << "-- test = " << test << endl;
    if (test%2) pb[i] = new Derivee;
    else pb[i] = new Base;
  }
```

```
  Derivee * pd;
  for (int i=0; i<4; i++)
  {
    cout << i << ". ----->" << endl;
    pd = dynamic_cast<Derivee *> (pb[i]);
    if (pd != nullptr)
    {
      pd->f(); // fct derivee !
      pd->g();
    }
    else
    {
      pb[i]->f(); // fct de base
      cout << "casting dynamique impossible" << endl;
    }
  }

  return 0;
}
```

dont un exemple d'exécution fournit

```
bash-3.00$ a.out
-- test = 16838
1.* constructeur de Base
-- test = 5758
2.* constructeur de Base
-- test = 10113
3.* constructeur de Base
3.** constructeur de Derivee
-- test = 17515
4.* constructeur de Base
4.** constructeur de Derivee
0. ----->
fonction base
casting dynamique impossible
1. ----->
fonction base
casting dynamique impossible
2. ----->
fonction derivee
fct add
3. ----->
fonction derivee
fct add
bash-3.00$
```

L'héritage des opérateurs

Les opérateurs étant des fonctions qui peuvent être des fonctions membres, elles peuvent donc être héritées comme toute fonction membre quelconque.

Mais cela peut poser problème !

Exemple : l'opérateur d'affectation

```
class mere
{
  friend ostream& operator<< (ostream&, const mere&);

  protected :
    float x;
    char *m;

  public :
    mere ()
    {
      x=4;
      m=new char[5];
      strcpy(m,"vilv");
      cout << "c. def. mere" << endl;
    }
    mere (float, char *);
    mere (mere&);
    mere& operator= (const mere&);
    mere& operator+ (const char);
    operator float () { return x; }
    int operator<(const mere& c) { return x < c.x; }
};

mere& mere::operator= (const mere& mm)
{
  x = mm.x;
  if (m) delete m;
  m = new char[strlen(mm.m)+1];
  strcpy(m,mm.m);
  return *this;
}

…
```

```
class fille : public mere
{
  private :
    char flag;

  public :
    fille () : mere() {};
    fille (char f) : mere() { flag = f; }
    char getFlag() { return flag; }
};

int main()
{
  mere m, mm(5.8,"bonsoir");
  fille f, fff('A');

  m = mm;
  cout << m << endl;

  f = fff;
  cout << f << endl;

  m = fff;
  cout << m << endl;

  f = mm; // !!!!
  cout << f << endl;

  return 0;
}
```

pose problème à la compilation pour « f = mm ». En effet,

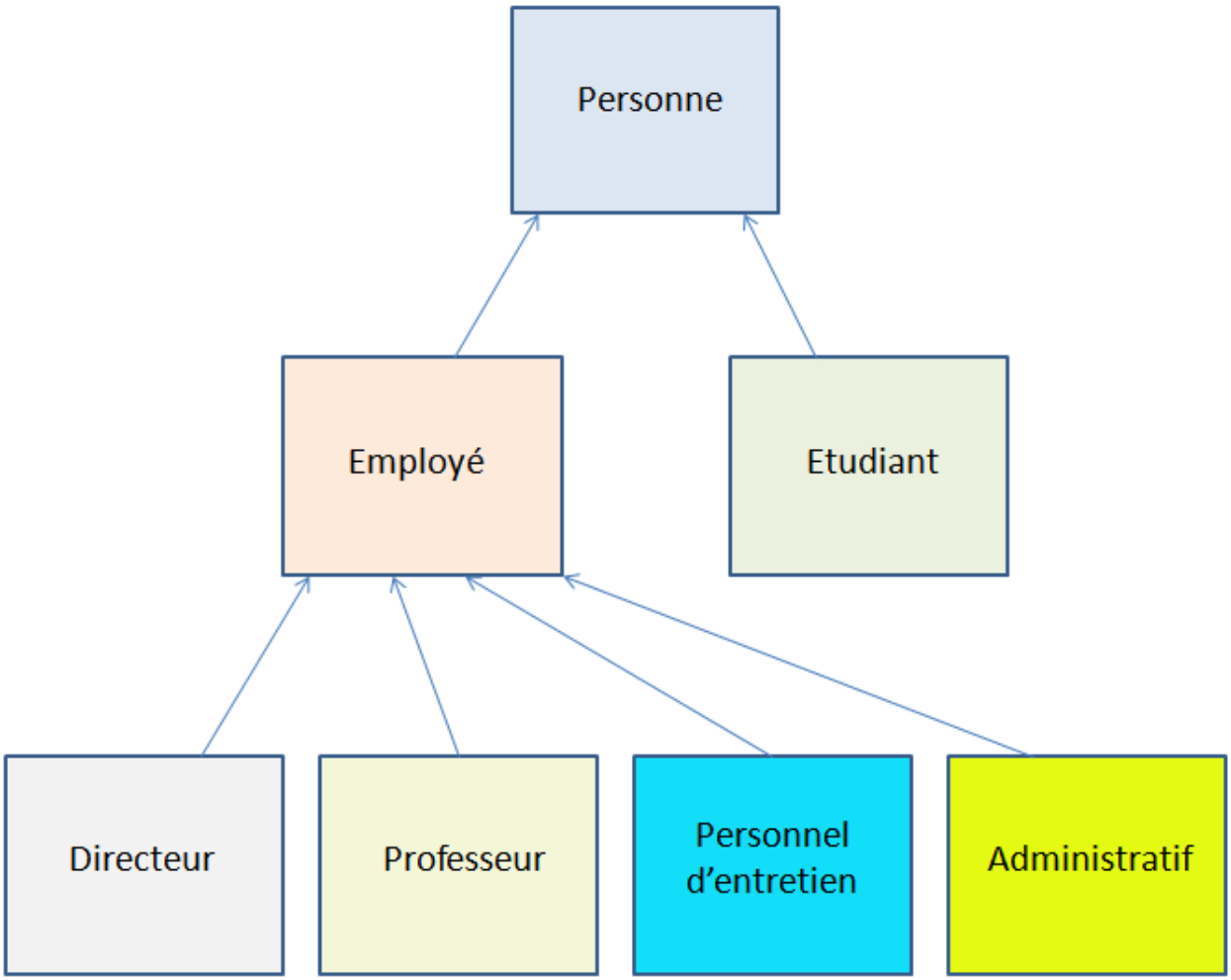
```
bash-3.00$ g++ HeritageOperateurs1.cpp
HeritageOperateurs1.cxx: In function `int main()':
HeritageOperateurs1.cxx:94: error: no match for 'operator=' in 'f = mm'
HeritageOperateurs1.cxx:88:     note:     candidates     are:     fille&
fille::operator=(const fille&)
bash-3.00$
```

En effet, il recherche un opérateur d'affectation dont le premier opérande est une fille et le second une mère et il ne le trouve pas !

→ il est donc plus que demandé de redéfinir les opérateurs (surtout **l'opérateur =**) dans les classes héritées sous peine de gros problème d'ambiguïté

Notion de hiérarchie

Exemple :



Vocabulaire :

Super-classe : la super-classe de « Employé » est « Personne ».

Sous-classe : une sous-classe de « Employé » est « Professeur ».

Les méthodes virtuelles pures : les classes abstraites

Il est courant de rencontrer dans une hiérarchie de classes des classes de base qui sont si peu spécialisées que l'on ne les instancie jamais.

En guise d'exemple sorti de la géométrie, considérons les classes

- **Rectangle**, **Carre**, **Cercle**, … qui ont leur propres variables membres spécifiques (coté, rayon, …) et qui héritent toutes de la classe
- **FormeGeometrique** qui possède une couleur

La hiérarchie est cohérente car toutes les formes géométriques possèdent une couleur et ont leurs propres caractéristiques.

Les classes **Rectangle**, **Carre**, **Cercle**, … possèderont toutes les méthodes

- **float aire()** qui retourne l'aire de la forme correspondante en tenant compte des caractéristiques propres de la forme
- **float perimetre()** qui retourne le périmètre de la forme correspondante en tenant compte des caractéristiques propres de la forme

Il n'y a aucun sens à définir les méthodes **aire()** et **périmètre()** dans la classe **FormeGeometrique** car celle-ci ne contient aucune information permettant le calcul de l'aire ou du périmètre. Néanmoins, nous savons tous qu'une forme géométrique possède une aire et un périmètre → on aimerait pouvoir déclarer les méthodes **aire()** et **perimetre()** dans la classe **FormeGeometrique** mais sans les définir vu que l'on a, dans cette classe, aucune information permettant leur calcul.

Nous sommes clairement ici dans un cas d'application où la classe **FormeGeometrique** serait une **classe abstraite**.

Une **classe abstraite** est une classe

- que l'on n'instanciera jamais (et que le compilateur nous empêchera d'instancier)
- qui sert de base à d'autres classe plus spécialisées en
  - leur fournissant un service « de base » (ici la gestion de la couleur)

- leur imposant une interface commune (ici on souhaite que toutes les classes qui héritent de **FormeGeometrique** disposent des méthodes **aire()** et **perimetre()**

Mais comment le compilateur C++ peut-il empêcher l'instanciation d'une classe abstraite ? En déclarant (dans le fichier .h) au moins une **méthode virtuelle pure** :

```
virtual float perimetre() = 0 ;
```

Une **méthode virtuelle pure** est une **méthode virtuelle**

- déclarée dans le fichier **.h** avec le décorateur « **= 0;** »
- qui n'a pas de corps → elle n'apparait pas dans le fichier **.cpp**

Le fait de déclarer une **méthode virtuelle pure** dans une classe la rend **abstraite** → on ne peut plus instancier cette classe → il suffit d'une seule méthode virtuelle pour rendre une classe abstraite.

Pour ne pas être abstraites, les classes héritées comme Rectangle ou Cercle devront définir toutes les **méthodes virtuelles pures** de la classe de base.

Voici un exemple de classe **FormeGeometrique** (fichier **FormeGeometrique.h**) :

```
#ifndef FORME_GEOMETRIQUE_H
#define FORME_GEOMETRIQUE_H

#include <iostream>
#include <string>
using namespace std;

class FormeGeometrique {
  protected:
    string couleur;

  public:
    FormeGeometrique();
    FormeGeometrique(const string&);
    FormeGeometrique(const FormeGeometrique&) = default;
    virtual ~FormeGeometrique() = default;

    virtual string toString() const;
    virtual float aire() const = 0;
```

```
    virtual float perimetre() const = 0;
};

#endif
```

où on observe que

- les méthodes **aire()** et **périmètre()** sont bien des **méthodes virtuelles pures** (« **= 0** »)
- la méthode **toString()** mais n'est pas virtuelle pure → son implémentation apparaîtra dans **FormeGeometrique.cpp**
- le décorateur « **= default** » a été ajouté au constructeur de copie et au destructeur afin de demander au compilateur de générer leur code de manière implicite (sans devoir les définir dans le fichier **FormeGeometrique.cpp**)
- le destructeur est lui-même virtuel

Voici le fichier **FormeGeometrique.cpp** :

```
#include "FormeGeometrique.h"
#include <sstream>
using namespace std;

FormeGeometrique::FormeGeometrique() {
	couleur = "bleu";
}

FormeGeometrique::FormeGeometrique(const string& couleur) {
	this->couleur = couleur;
}

string FormeGeometrique::toString() const {
	ostringstream oss;
	oss << "Forme geometrique de couleur " << this->couleur;
	return oss.str();
}
```

où on observe que

- les méthodes virtuelles pures **aire()** et **perimetre()** sont bel et bien absentes
- le constructeur de copie et le destructeur sont également absents car générés implicitement par le compilateur

Considérons à présent le main de test (fichier **main4.cpp**) suivant :

```
#include <iostream>
using namespace std;
#include "FormeGeometrique.h"

int main() {
  FormeGeometrique *ptr = nullptr;

  FormeGeometrique fg1("vert"),fg2;

  return 0;
}
```

Sa compilation et celle de la classe **FormeGeometrique** fournissent

```
[student@moon]$ g++ FormeGeometrique.cpp -c
[student@moon]$ g++ main4.cpp FormeGeometrique.o -o main4
main4.cpp: Dans la fonction « int main() »:
main4.cpp:8:20: error: ne peut déclarer la variable « fg1 » comme étant
de type abstrait « FormeGeometrique »
   FormeGeometrique fg1("vert"),fg2;
                    ^~~
In file included from main4.cpp:3:
FormeGeometrique.h:8:7: note:   parce que les fonctions virtuelles
suivantes sont pures dans « FormeGeometrique »:
 class FormeGeometrique {
       ^~~~~~~~~~~~~~~~
FormeGeometrique.h:19:19: note: « virtual float FormeGeometrique::aire()
const »
     virtual float aire() const = 0;
                   ^~~~
FormeGeometrique.h:20:19: note: « virtual float
FormeGeometrique::perimetre() const »
     virtual float perimetre() const = 0;
                   ^~~~~~~~~
main4.cpp:8:32: error: ne peut déclarer la variable « fg2 » comme étant
de type abstrait « FormeGeometrique »
   FormeGeometrique fg1("vert"),fg2;
                                ^~~
[student@moon]$
```

où on observe que

- la classe **FormeGeometrique** compile sans souci
- il est possible de déclarer un pointeur de type **FormeGeometrique*** → aucune erreur de compilation à ce sujet

- la ligne 8 tente d'instancier deux objets de la classe **FormeGeometrique** → le compilateur le rejette étant donné que la classe est abstraite

Considérons à présent les classes **Carre** et **Disque** héritées de **FormeGeometrique** et données par (fichier **Carre.h**) :

```
#ifndef CARRE_H
#define CARRE_H

#include <iostream>
#include <string>
using namespace std;
#include "FormeGeometrique.h"

class Carre : public FormeGeometrique {
  private:
      float cote;

  public:
      Carre();
      Carre(const string&,float);
      Carre(const Carre&) = default;
      ~Carre() = default;

     string toString() const override;
     float aire() const override;
     float perimetre() const override;
};

#endif
```

et (fichier **Carre.cpp**) :

```
#include "Carre.h"
#include <sstream>
using namespace std;

Carre::Carre():FormeGeometrique() {
      cote = 1.0;
}

Carre::Carre(const string& couleur, float cote):FormeGeometrique(couleur)
{
      this->cote = cote;
```

```
}

string Carre::toString() const {
      ostringstream oss;
      oss << "Carre " << this->couleur << " de cote " << this->cote;
      return oss.str();
}

float Carre::aire() const {
      return this->cote*this->cote;
}

float Carre::perimetre() const {
      return 4.0*this->cote;
}
```

Et (fichier **Disque.h**) :

```
#ifndef DISQUE_H
#define DISQUE_H

#include <iostream>
#include <string>
using namespace std;
#include "FormeGeometrique.h"

class Disque : public FormeGeometrique {
  private:
      float rayon;

  public:
      Disque();
      Disque(const string&,float);
      Disque(const Disque&) = default;
      ~Disque() = default;

     string toString() const override;
     float aire() const override;
     float perimetre() const override;
};

#endif
```

Et (fichier **Disque.cpp**) :

```
#include "Disque.h"
#include <sstream>
using namespace std;

Disque::Disque():FormeGeometrique() {
      rayon = 1.0;
}
```

```
Disque::Disque(const string& couleur, float
rayon):FormeGeometrique(couleur) {
	this->rayon = rayon;
}

string Disque::toString() const {
	ostringstream oss;
	oss << "Disque " << this->couleur << " de rayon " << this->rayon;
	return oss.str();
}

float Disque::aire() const {
	return 3.1416*this->rayon*this->rayon;
}

float Disque::perimetre() const {
	return 2.0*3.1416*this->rayon;
}
```

où on observe que

- le mot réservé **« virtual »** a disparu dans les classes héritées → ce n'est plus nécessaire → dès que c'est déclaré **virtual** dans une classe, les méthodes surchargées sont automatiquement virtual dans les classes filles
- dans les fichiers **.h**, toutes les méthodes virtuelles sont à présent décorées du mot réservé **« override »** précisant au compilateur que l'on a affaire à une méthode surchargée → cela permet au compilateur de faire une vérification de signature supplémentaire et ça clarifie le code
- ni le mot réservé **« virtual »** et ni le mot réservé **« override »** n'apparaissent dans les fichiers **.cpp**
- Ayant surchargé et défini les méthodes virtuelles pures **aire()** et **perimetre()** de la classe abstraite **FormeGeomettrique**, les classes **Carre** et **Disque** ne sont plus abstraites et peuvent être instanciées

Voici un main de test (fichier **main5.cpp**) de ces classes :

```
#include <iostream>
using namespace std;
#include "FormeGeometrique.h"
#include "Carre.h"
#include "Disque.h"

int main() {
```

```
  Carre c("rouge",3.0);
  Disque d("bleu",4.0);

  cout << c.toString() << endl;
  cout << "Aire = " << c.aire() << endl;
  cout << "Perimetre = " << c.perimetre() << endl << endl;

  cout << d.toString() << endl;
  cout << "Aire = " << d.aire() << endl;
  cout << "Perimetre = " << d.perimetre() << endl << endl;

  FormeGeometrique* vec[] = {&c, &d};
  for(int i=0 ; i<2 ; i++) {
    cout << vec[i]->toString() << endl;
    cout << "Aire = " << vec[i]->aire() << endl;
    cout << "Perimetre = " << vec[i]->perimetre() << endl << endl;
  }
  return 0;
}
```

dont la compilation et l'exécution fournissent

```
[student@moon]$ g++ Disque.cpp -c
[student@moon]$ g++ Carre.cpp -c
[student@moon]$ g++ main5.cpp Disque.o Carre.o FormeGeometrique.o -o
main5
[student@moon]$ main5
Carre rouge de cote 3
Aire = 9
Perimetre = 12

Disque bleu de rayon 4
Aire = 50.2656
Perimetre = 25.1328

Carre rouge de cote 3
Aire = 9
Perimetre = 12

Disque bleu de rayon 4
Aire = 50.2656
Perimetre = 25.1328

[student@moon]$
```

où on observe que

- nous n'avons pas instancié d'objet de la classe **FormeGeometrique** mais bien **Disque** et **Carre**

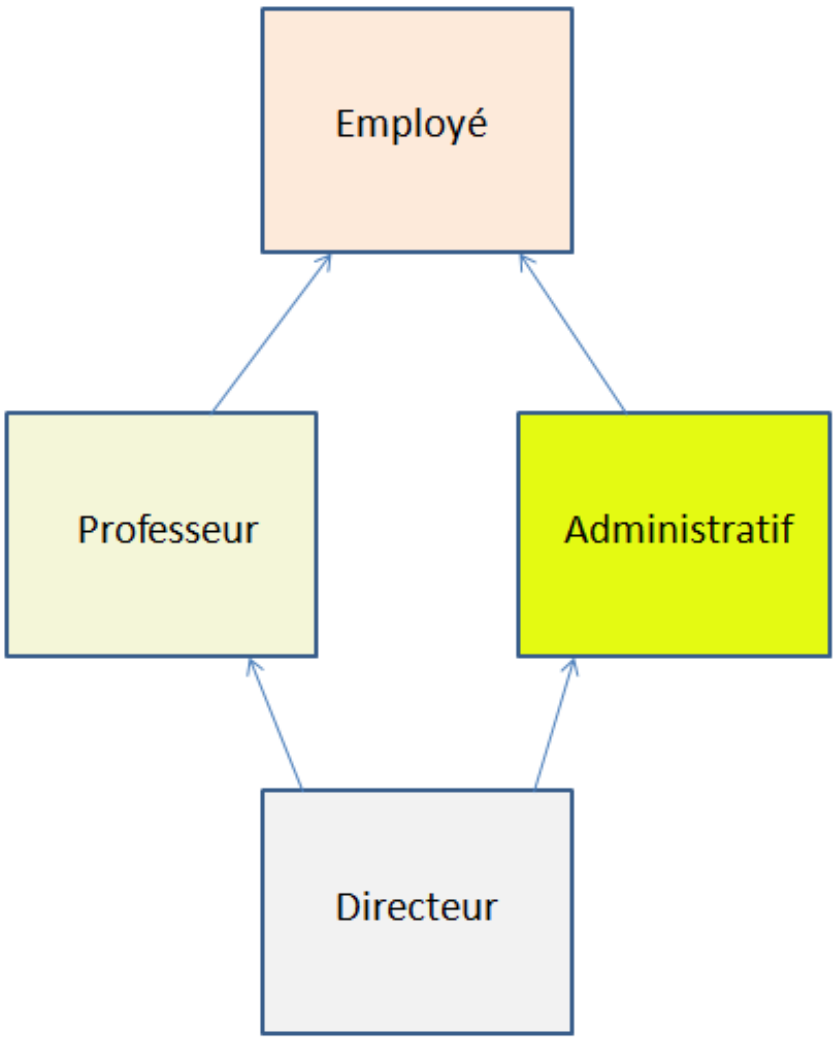
- nous avons déclaré un **vecteur de pointeurs** de type **FormeGeometrique** → aucun souci de compilation → ces pointeurs pointent vers des objets instanciant des classes filles → la virtualité est de mise !

L'héritage multiple

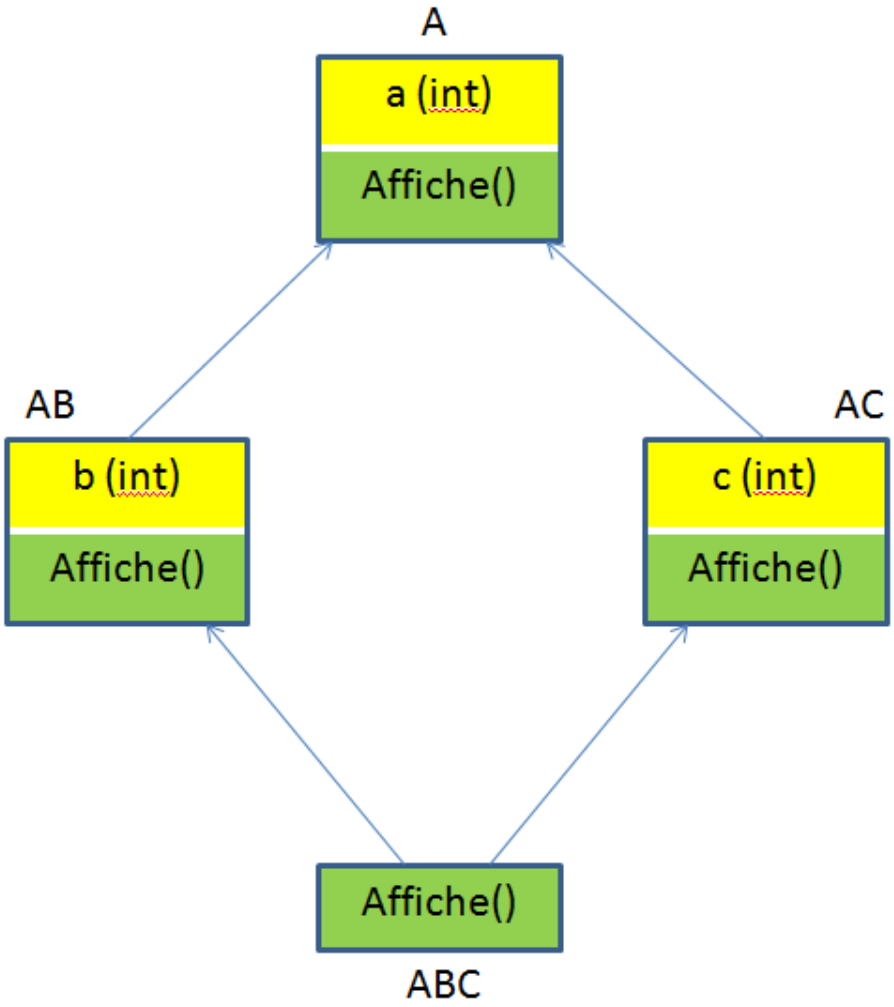
Le C++ permet à une classe d'hériter de plusieurs classes : on parle d'**héritage multiple**.

Si ses classes mères héritent d'une même classe de base, on parle d'**héritage en diamant**.

Exemple :



Autre exemple :



Comme nous allons le voir, l'héritage multiple peut provoquer des **problèmes d'ambiguïté**. En effet,

```
class A
{
  protected:
    int a;

  public:
    A() { cout << "Constructeur defaut A" << endl; a=0;}
    A(int x) { cout << "Constructeur init A" << endl; a=x;}
    ~A() { cout << "Destructeur A" << endl;}

    int getA() {return a;}
    void setA(int x) {a=x;}
    void Affiche() { cout << "A = " << a << endl; }
};

class AB : public A
```

```
{
  protected:
    int b;

  public:
    AB():A() { cout << "Constructeur defaut AB" << endl; b=0;}
    AB(int xa,int xb):A(xa)
      { cout << "Constructeur init AB" << endl; b=xb;}
    ~AB() { cout << "Destructeur AB" << endl;}

    int getB() {return b;}
    void setB(int x) {b=x;}
    void Affiche() {cout << "A = " << a << " # B = " << b << endl;}
};

class AC : public A
{
  protected:
    int c;

  public:
    AC():A() { cout << "Constructeur defaut AC" << endl; c=0;}
    AC(int xa,int xc):A(xa)
      { cout << "Constructeur init AC" << endl; c=xc;}
    ~AC() { cout << "Destructeur AC" << endl;}

    int getC() {return c;}
    void setC(int x) {c=x;}
    void Affiche() {cout << "A = " << a << " # C = " << c << endl;}
};

class ABC : public AB, public AC
{
  public:
    ABC():AB(),AC() { cout << "Constructeur defaut ABC" << endl; }
    ABC(int xa,int xb,int xc):AB(xa,xb),AC(xa,xc)
      { cout << "Constructeur init ABC" << endl; }
    ~ABC() { cout << "Destructeur ABC" << endl; }

    void Affiche()
    {cout << "A = " << AB::a << " # B = " << b << " # C = " << c <<
endl;}
    // on affiche le a de AB ou le a de AC !!!
};
```

```
int main()
{
  ABC abc(1,2,3);
  abc.Affiche();

  abc.setA(5);
  cout << "a = " << abc.getA() << endl;

  return 0;
}
```

Le compilateur refuse :

```
bash-3.00$ g++ HeritageMultiple1.cpp
HeritageMultiple1.cxx: In function `int main()':
HeritageMultiple1.cxx:66: error: request for member `setA' is ambiguous
HeritageMultiple1.cxx:15: error: candidates are: void A::setA(int)
HeritageMultiple1.cxx:15: error:                 void A::setA(int)
HeritageMultiple1.cxx:67: error: request for member `getA' is ambiguous
HeritageMultiple1.cxx:14: error: candidates are: int A::getA()
HeritageMultiple1.cxx:14: error:                 int A::getA()
bash-3.00$
```

En effet, le compilateur ne sait pas quelle fonction appeler, celle de **AB** ou celle de **AC**… Il faut donc lui préciser laquelle des deux appeler afin de lever ce problème d'ambiguïté :

```
int main()
{
  ABC abc(1,2,3);
  abc.Affiche();

  abc.AB::setA(5);
  abc.AC::setA(8);
  cout << "a = " << abc.AB::getA() << endl;
  cout << "a = " << abc.AC::getA() << endl;

  return 0;
}
```

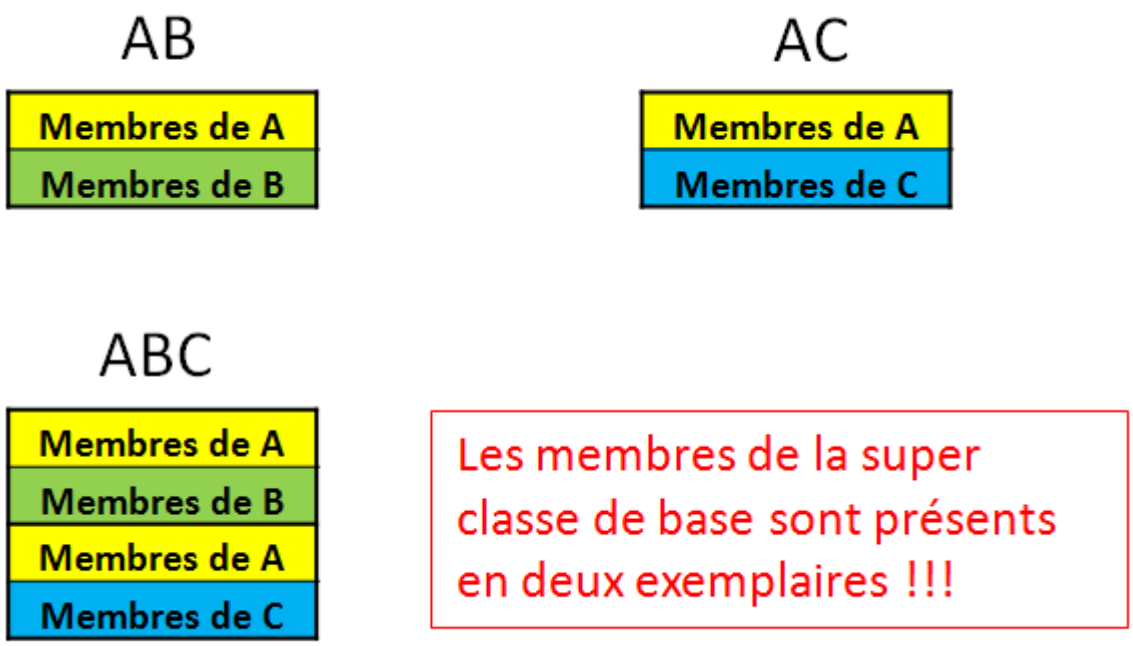
→ ce qui est une horreur absolue… et pas du tout ce que l'on veut !

dont l'exécution fournit

```
bash-3.00$ a.out
Constructeur init A
Constructeur init AB
Constructeur init A
Constructeur init AC
Constructeur init ABC
A = 1 # B = 2 # C = 3
a = 5
a = 8
Destructeur ABC
Destructeur AC
Destructeur A
Destructeur AB
Destructeur A
bash-3.00$
```

Cela signifie que **la variable a existe 2 fois en mémoire**, ainsi que **les fonctions getA() et setA()** ☹ !!!

Voici une représentation de ce qui se passe en mémoire :



L'héritage virtuel

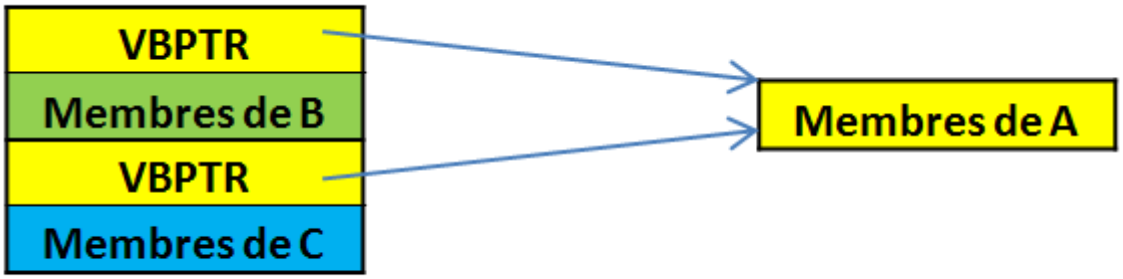
Pour lever ce problème d'ambiguïté, il est également possible de faire en sorte que **la classe dérivée** (ABC dans notre exemple) **ne comporte qu'une seule version de la classe de base**. On pratique dans ce cas un **héritage virtuel**.

Lorsqu'une classe hérite virtuellement d'une autre classe,

- les données membres de la classe de base ne sont pas recopiés dans la classe dérivée ;
- on incorpore à la place une **sorte de pointeur sur les données de la classe de base**. Ainsi, si deux classes dérivent d'une même classe de base, elles héritent toutes deux d'un pointeur sur les données de celle-ci.

Dès lors, il n'y a plus d'ambiguïté dans le cas d'un héritage multiple ☺ !

Représentation mémoire de l'héritage virtuel :



Au niveau du code, cela donne :

```
class A
{
  protected:
    int a;

  public:
    A() { cout << "Constructeur defaut A" << endl; a=0;}
    A(int x) { cout << "Constructeur init A" << endl; a=x;}
    ~A() { cout << "Destructeur A" << endl;}

    int getA() {return a;}
    void setA(int x) {a=x;}
    void Affiche() { cout << "A = " << a << endl; }
};

class AB : public virtual A
```

```
{
  protected:
    int b;

  public:
    AB():A() { cout << "Constructeur defaut AB" << endl; b=0;}
    AB(int xa,int xb):A(xa)
      { cout << "Constructeur init AB" << endl; b=xb;}
    ~AB() { cout << "Destructeur AB" << endl;}

    int getB() {return b;}
    void setB(int x) {b=x;}
    void Affiche() {cout << "A = " << a << " # B = " << b << endl;}
};

class AC : public virtual A
{
  protected:
    int c;

  public:
    AC():A() { cout << "Constructeur defaut AC" << endl; c=0;}
    AC(int xa,int xc):A(xa)
      { cout << "Constructeur init AC" << endl; c=xc;}
    ~AC() { cout << "Destructeur AC" << endl;}

    int getC() {return c;}
    void setC(int x) {c=x;}
    void Affiche() {cout << "A = " << a << " # C = " << c << endl;}
};

class ABC : public virtual AB, public virtual AC
{
  public:
    ABC():A(),AB(),AC()
      { cout << "Constructeur defaut ABC" << endl; }
    ABC(int xa,int xb,int xc):A(xa),AB(xa,xb),AC(xa,xc)
      { cout << "Constructeur init ABC" << endl; }
    ~ABC() { cout << "Destructeur ABC" << endl; }

    void Affiche() { cout << "A = " << a << " # B = " << b << " # C = "
<< c << endl;}
};
```

```
int main()
{
  ABC abc(1,2,3);
  abc.Affiche();

  abc.setA(5);
  abc.setA(8);
  cout << "a = " << abc.getA() << endl;

  abc.Affiche();

  return 0;
}
```

dont l'exécution fournit

```
bash-3.00$ a.out
Constructeur init A
Constructeur init AB
Constructeur init AC
Constructeur init ABC
A = 1 # B = 2 # C = 3
a = 8
A = 8 # B = 2 # C = 3
Destructeur ABC
Destructeur AC
Destructeur AB
Destructeur A
bash-3.00$
```

On remarque que :

- Le mécanisme d'héritage virtuel **impose d'appeler explicitement le constructeur de la super classe**.
- Dès qu'une classe de base est virtuelle, ceci est valable pour toutes les classes de la hiérarchie qui dépendent d'elle.