1

Héritage

Nous allons à présent aborder l'apport fondamental du C++ (et des langages objets) par rapport aux langages procéduraux classiques (C), à savoir l'héritage. Cette notion va nous montrer une nouvelle fois comment réutiliser du code déjà écrit.

1.1 Une petite histoire

Considérons un programmeur (que nous appellerons Eric) dans une société de location de vélos et de voitures, à qui l'on a demandé d'écrire un programme pour gérer l'ensemble des véhicules. Eric commence par écrire une classe décrivant une voiture :

```
1  class Voiture
2  {
3      public :
4          Voiture(int etat = 1, fuel = 100,
5        couleur = 0, prix = 10000);
6          ~Voiture();
7
8          void setEtat(int etat);
9          int getEtat();
10         void setFuel(int fuel);
11         int getFuel();
12         void setCouleur(int couleur);
13         int getCouleur(int couleur);
14         void getPrix();
15         int setPrix();
16         void description(); //affiche "Une voiture qui coute x Euros""
17
18     private:
19         int m_etat; //neuf?
20         int m_fuel; // niveau d'essence
21         int m_couleur;
22         int m_prix;
23 };
```

Listing 1.1 – voiture1.h

et continue avec la classe vélo :

```
class Velo
{
    public :
        Velo(int etat = 1, couleur = 0, prix = 10000);
        ~Velo();

        void setEtat(int etat);
        int getEtat();
        void setFuel(int fuel);
        int getFuel();
        void setCouleur(int couleur);
        int getCouleur(int couleur);
        void getPrix();
        int setPrix();
        bool aUnAntivol();
        void aUnAntivol(bool presenceAntivol);
        void description(); //affiche "Un velo qui coute x Euros"

    private:
        int m_etat; //neuf?
        int m_couleur;
        int m_prix;
        bool m_antivol;
};
```

Listing 1.2 – velo1.h

Eric étant prudent, il utilise les mêmes noms de variables afin de ne pas perturber l'utilisateur. Il décide alors d'écrire une fonction affichant les descriptions des véhicules :

```
#include <iostream.h>

void afficheDescriptions(
    Velo *listeVelos,
    int nVelos,
    Voiture* listeVoiture,
    int nVoitures)
{
    int i;

    for(i = 0; i < nVelos; i++)
        listeVelos[i].description();
    for(i = 0; i < nVoitures; i++)
  listeVoiture[i].description();
}

int main()
{
  return 0;
}
```

Listing 1.3 – main_vehicules1.cpp

Eric fait alors les remarques suivantes :

- Les vélos et les voitures sont tous deux des véhicules et partagent de nombreuses caractéristiques (prix, couleur, état), mais ont malgré cela quelques différences.
- Si son entreprise se met a louer d'autres véhicules (des motos, par exemple), il va devoir créer une nouvelle classe presque identique aux précédentes. Il devra également modifier sa fonction `description`.

Dans la mesure où vélos et voitures sont tous deux des véhicules, ne serait-il pas possible de définir un objet `vehicule`, et de les spécialiser en n'expliquant que leurs différences en vélos et voitures ? Cela permettrait de simplifier et raccourcir le code, et d'augmenter sa lisibilité. Il serait également plus facilement extensible et maintenable. Ceci est possible au moyen de ce que l'on appelle *l'héritage*.

1.2 Héritage simple

L'idée est la suivante : nous allons définir une classe de véhicule, et la spécialiser (ce qu'on appelle *dériver* la classe *mère* - ici la classe `vehicule`- en des classes *filles*), `velo` et `voiture` ne contenant que les différences. On dira que les classes `velo` et `voiture` héritent de la classe mère `voiture`. L'héritage permet d'écrire une relation du type "est un", puisqu'un vélo et une voiture sont des véhicules.

D'un point de vue syntaxique, la déclaration d'une classe dérivée est très simple. Il y a trois possibilités :

```
/*premiere possibilite */
class Fille : public class Mere
{
    /*nouveaux membres*/
};
/*seconde possibilite */
class Fille : protected class Mere
{
    /*nouveaux membres*/
};
/*troisieme possibilite */
class Fille : private class Mere
{
    /*nouveaux membres*/
};
```

Le lecteur attentif aura remarqué l'emploi des mots public, protected, ou private. Dans un premier temps, nous n'utiliserons que la forme `public`, et reviendrons dans la section ?? sur leur signification.

En pratique, notre code devient le suivant :

```
#ifndef VEHICULE_H
#define VEHICULE_H

class Vehicule
{
    public :
        Vehicule(int etat = 1,
      int couleur = 0,
      int prix = 10000);
        ~Vehicule();

        void setEtat(int etat);
        int getEtat();
        void setCouleur(int couleur);
        int getCouleur(int couleur);
        int getPrix();
        void setPrix(int prix);

    private:
        int m_etat; //neuf?
        int m_couleur;
        int m_prix;

};

#endif
```

Listing 1.4 – vehicule2.h

Nous avons donc rangé tous les attributs communs d'un véhicule dans une classe habilement nommée `vehicule`. La fonction `description` est laissée à l'extérieur, comme un attribut non commun des véhicules, dans la mesure où suivant l'objet considéré (un `velo` ou une `voiture`), elle n'affichera pas la même chose. Que deviennent alors `Velo` et `voiture` ? Le résultat est visible ci-dessous :

```
#include "vehicule2.h"

class Voiture : public Vehicule
{
    public:
        Voiture(int etat = 1, int fuel = 100,
    int couleur = 0, int prix = 10000);
        ~Voiture();

        void description();
        void setFuel(int fuel);
        int getFuel();

    private:
        int m_fuel;
};
```

Listing 1.5 – voiture2.h

```
#include "vehicule2.h"

class Velo : public Vehicule
{
    public:
        Velo(int etat = 1,
    bool antivol = false,
    int couleur = 0,
    int prix = 10000);
        ~Velo();

        void description();

        bool aUnAntivol();
        void aUnAntivol(bool presenceAntivol);

    private:
        bool m_antivol;
};
```

Listing 1.6 – velo2.h

Le code de la classe `Voiture` (par exemple) s'écrit alors :

```
#include "voiture2.h"
#include <iostream>
using namespace std;

Voiture::Voiture(int etat, int fuel, int couleur, int prix)
:Vehicule(etat, couleur, prix), m_fuel(fuel)
{
}

Voiture::~Voiture()
{
}


void Voiture::setFuel(int fuel)
{
  m_fuel = fuel;
}

int Voiture::getFuel()
{
  return m_fuel;
}

void Voiture::description()
{
  cout << "Une voiture qui coute " << m_prix << endl;
}
```

Listing 1.7 – voiture2.cpp

Comme nous pouvons le constater, le code des nouvelles classes `Voiture` et Velo est nettement plus court. En effet, nous n'avons pas besoin de redéclarer la plupart des fonctions, car elles sont automatiquement *héritées* de la classe mère véhicule. En particulier, la fonction `description` fait appel au membre `m_prix` de la classe mère `Vehicule`. Cela signifie également que l'on peut appeler les méthodes de la classe `vehicule` depuis un objet `Velo` ou `Voiture`. Par exemple, le code-ci dessous fonctionnera correctement :

```
Velo v;

cout << v.getPrix();
```

Il y a cependant un léger problème, en particulier en ce qui concerne la méthode `description`. En effet, si l'on compile le code présenté, le compilateur va nous donner une erreur :

```
'm_prix' : cannot access private member declared in class 'Vehicule'
```

Accès	public	protected	private
Membres de la classe	Oui	Oui	Oui
Membres des classes dérivées	Oui	Oui	Non
Reste du monde	Oui	Non	Non

TABLE 1.1 – Encapsulation : différents degrés de visibilité

Que signifie cette erreur ? L'origine de notre problème est que nous avons supposé que nous avions accès dans nos classes dérivées aux membres privés de la classe mère. Ce n'est pas le cas, et nous allons passer en revue ce problème de droit d'accès dans la section suivante.

1.3 Encapsulation : le retour

Lors de notre introduction à l'encapsulation, nous avions vu deux mots clés : `private` et `public`. `private` servait à interdire au reste du monde d'accéder à certains membres, et `public` servait à autoriser le reste du monde à accéder à certains membres. Cependant, que se passe-t-il lorsque une classe hérite d'une autre ?

Nous avons, dans le cadre vu jusqu'à présent (héritage à l'aide de `public`), le comportement suivant :

- Les membres `private` de la classe mère ne sont *pas* accessibles à la classe dérivée.
- Les membres `public` de la classe mère sont accessibles à la classe dérivée.

Cependant, on peut souhaiter un niveau intermédiaire : accessible à la classe mère, à la classe dérivée, mais *pas* au reste du monde (ce qui était le but initial de `private`). C'est le but du mot clé `protected`.

L'ensemble de ces règles d'héritage est résumé tableau 1.1.

Pour que nos classes dérivées aient accès au membre `m_prix`, nous avons donc deux possibilités :

- Utiliser l'accesseur `getprix` ;
- Mettre `m_prix` en `protected`.

Nous allons retenir la seconde solution, et notre classe `Vehicule` s'écrira alors :

```
#ifndef VEHICULE_H
#define VEHICULE_H

#include <iostream>
using namespace std;

class Vehicule
{
    public :
        Vehicule(int etat = 1,
    int couleur = 0,
    int prix = 10000);
        ~Vehicule();

        void setEtat(int etat);
        int getEtat();
        void setCouleur(int couleur);
        int getCouleur(int couleur);
        int getPrix();
        void setPrix(int prix);

    protected:
        int m_prix;

    private:
        int m_etat; //neuf?
        int m_couleur;


};
#endif
```

Listing 1.8 – vehicule3.h

Le code des autres classes sera alors inchangé.

Eric a donc organisé son code de manière beaucoup plus rationnelle et donc plus maintenable. Cependant, il souhaiterait tirer parti du fait qu'il peut à présent manipuler des `Vehicule` de manière générique, puisque :

- Il a défini une classe `Vehicule`.
- `Velo` et `Voiture` ont tous deux une méthode `description`.

Il décide donc de modifier le code de sa fonction d'affichage des descriptions (voir listing **??**) et de le remplacer par le suivant :

```cpp
#include "velo3.h"
#include "vehicule3.h"

void afficheDescriptionVehicules(Vehicule *listeVehicules, int nVehicules
    )
{
    //Attention, ne fonctionne pas!
    for(int i = 0; i < nVehicules; i++)
        listeVehicules[i].description();
}

int main()
{
  Velo * liste_velos = new Velo[10];

  afficheDescriptionVehicules(liste_velos, 10);

  delete[] liste_velos;

  return 0;
}
```

Listing 1.9 – main_ vehicules3.cpp

Il se dit alors que si le directeur de la compagnie - qui est en plein développement - décide de louer des motos, ce code restera inchangé. Un nouveau problème surgit alors, lors de la compilation :

```
'description' is not a member of 'Vehicule'
```

En effet, si `Velo` et `Voiture` ont tous deux une méthode `description`, ce n'est pas le cas de `Vehicule`. Il faut donc la rajouter, ce qui nous donne le header suivant :

```
#ifndef VEHICULE_H
#define VEHICULE_H

#include <iostream>
using namespace std;

class Vehicule
{
    public :
        Vehicule(int etat = 1,
    int couleur = 0,
    int prix = 10000);
        ~Vehicule();

        void setEtat(int etat);
        int getEtat();
        void setCouleur(int couleur);
        int getCouleur(int couleur);
        int getPrix();
        void setPrix(int prix);
        void description();

    protected:
  int m_prix;

    private:
        int m_etat; //neuf?
        int m_couleur;

};
#endif
```

Listing 1.10 – vehicule4.h

Il est important de noter qu'au final, nous avons *surchargé* dans les classes dérivées la méthode `description` définie dans la classe mère `Vehicule`.

Cette modification faite, il tente de réutiliser le code du listing ??.Ce code ne fait malheureusement pas ce qui est attendu. Pour être précis, si on l'exécute, on obtient :

Ce problème sera résolu dans la section suivante, par l'emploi de fonctions virtuelles.

1.4 Fonctions virtuelles

Pour résoudre le problème du listing 1.9, il nous faut

– Comprendre l'origine du problème ;

– Trouver un moyen de le résoudre.

L'origine du problème est relativement simple. L'idée est la suivante : à la compilation, le compilateur C++ assigne une adresse fixe en mémoire aux méthodes de chaque objet. Dans notre cas, il assignera trois adresses, une

pour `Vehicule.description`, une pour `Voiture.description`, et une pour `Velo.description`. Cette opération est effectuée une et une seule fois, *à la compilation*. La conséquence directe est que la ligne

```
1   listeVehicules[i].description()
```

va toujours appeler la même fonction `Vehicule.description`, et ce, *quelque soit le type de l'objet qui est fourni*. Il nous faudrait donc un moyen d'indiquer au compilateur qu'il doit *à l'exécution* déterminer le type de l'objet, et appeler la fonction en conséquence. Le C++ rend cela possible, au moyen du mot clé `virtual` que l'on va associer au à une fonction. On appelera *fonction virtuelle* une fonction dont l'adresse est déterminée à l'exécution.

Une fonction virtuelle est déclarée de la manière suivante :

```
1   class MaClasse
2   {
3     virtual typeDeRetour nomFonction(parametres);
4   }
```

Listing 1.11 – fichier.h

Le code reste lui inchangé :

```
1   typeDeRetour MaClasse::nomFonction(parametres)
2   {
3
4     return x;
5   }
```

Listing 1.12 – fichier.cpp

Nos headers restent donc inchangés, à l'exception de l'ajout du mot clé `virtual` :

```
#ifndef VEHICULE_H
#define VEHICULE_H

#include <iostream>
using namespace std;

class Vehicule
{
    public :
        Vehicule(int etat = 1,
    int couleur = 0,
    int prix = 10000);
        ~Vehicule();

        void setEtat(int etat);
        int getEtat();
        void setCouleur(int couleur);
        int getCouleur(int couleur);
        int getPrix();
        void setPrix(int prix);
        virtual void description();

    protected:
  int m_prix;

    private:
        int m_etat; //neuf?
        int m_couleur;

};
#endif
```

Listing 1.13 – vehicule5.h

```
#include "vehicule5.h"
#include <iostream>
using namespace std;

class Velo : public Vehicule
{
    public:
        Velo(int etat = 1,
    bool antivol = false,
    int couleur = 0,
    int prix = 10000);
  ~Velo();

        virtual void description(); //affiche "Je suis un velo"

        bool aUnAntivol();
        void aUnAntivol(bool presenceAntivol);

    private:
        bool m_antivol;
};
```

Listing 1.14 – velo5.h

```
#include "vehicule5.h"

class Voiture : public Vehicule
{
    public:
        Voiture(int etat = 1, int fuel = 100,
    int couleur = 0, int prix = 10000);
        ~Voiture();

        virtual void description();
        void setFuel(int fuel);
        int getFuel();

    private:
        int m_fuel;
};
```

Listing 1.15 – voiture5.h

Au vu des problèmes posés par l'absence du mot clé `virtual` dans certains cas, on est en droit de se demander pourquoi sa présence n'est pas systématique. La raison principale est une question de performances. En effet, déterminer à l'exécution quelle fonction il faut appeler, et ce à chaque appel, n'est pas gratuit.

Ceci fait, il peut enfin utiliser sa fonction d'affichage générique :

```
#include "velo5.h"
#include "voiture5.h"
#include <iostream>
#include <vector>
using namespace std;

void afficheDescriptions(const vector<Vehicule *> & listeVehicules)
{
    for(int i = 0; i < listeVehicules.size(); i++)
      listeVehicules[i]->description();
}

int main()
{
  vector<Vehicule *> listeVehicules;

  for(int i= 0; i<5;i++)
  {
    listeVehicules.push_back(dynamic_cast<Vehicule *>(new Velo())) ;
    listeVehicules.push_back(dynamic_cast<Vehicule *> (new Voiture())) ;
  }

  afficheDescriptions(listeVehicules);

  return 0;
}
```

Listing 1.16 – main_vehicules5.cpp

1.5 Classes abstraites

Nous avons vu l'emploi des fonctions virtuelles, qui permettent aux "bonnes" fonctions des classes dérivées d'être appelées. L'idée importante à retenir est que l'on peut créer de nouvelles classes, qui, si elles se conforment à un modèle donné (celui donné par `vehicule`), c'est-à-dire à une *interface*, peuvent s'intégrer naturellement dans un ensemble existant. Pourquoi ne pas pousser le raisonnement un peu plus loin ?

Assez fréquemment, il arrive que l'on souhaite définir un objet de manière abstraite, générique, sans vouloir l'utiliser tel quel. Par exemple, la classe véhicule que l'on a défini précédemment, n'a pas beaucoup d'intérêt utilisée telle quelle. En effet,

- On veut toujours savoir quel type de véhicule on manipule.
- La méthode qui affiche le type de véhicule affiche "Un vehicule qui coute x Euros" ce qui n'a pas d'intérêt.

Idéalement, on voudrait pouvoir :

- interdire la création d'objets de type véhicule ;
- obliger l'utilisateur à créer des classes dérivées ;
- obliger l'utilisateur à surcharger certaines méthodes, par exemple la méthode `description`.

Tout ceci est possible, en transformant notre classe `vehicule` en une classe abstraite. Cela se fait très simplement : dans le fichier de déclaration de la classe (le header), il suffit d'ajouter `"=0;"` à la suite du nom d'une méthode. Alors :

- La méthode en question devient *virtuelle pure* et il est obligatoire de la surcharger. Si n méthodes sont *virtuelles pures*, il faut surcharger les n méthodes.
- L'utilisateur ne pourra pas créer d'objet de la classe `vehicule`, celle-ci devenant abstraite. Une seule méthode virtuelle pure suffit à rendre toute la classe abstraite.
- L'utilisateur devra obligatoire créer une classe dérivée, dans laquelle la méthode en question devra obligatoirement être surchargée.

En l'occurrence :

```
1  #ifndef VEHICULE_H
2  #define VEHICULE_H
3
4  #include <iostream>
5  using namespace std;
6
7  class Vehicule
8  {
9      public :
10         Vehicule(int etat = 1,
11     int couleur = 0,
12     int prix = 10000);
13         ~Vehicule();
14
15         void setEtat(int etat);
16         int getEtat();
17         void setCouleur(int couleur);
18         int getCouleur(int couleur);
19         int getPrix();
20         void setPrix(int prix);
21         virtual void description() = 0;
22
23     protected:
24   int m_prix;
25
26     private:
27         int m_etat; //neuf?
28         int m_couleur;
29
30 };
31 #endif
```

Listing 1.17 – vehicule6.h

```
#include "velo6.h"
#include "voiture6.h"
#include <iostream>
#include <vector>
using namespace std;

void afficheDescriptions(const vector<Vehicule *> & listeVehicules)
{
    for(int i = 0; i < listeVehicules.size(); i++)
      listeVehicules[i]->description();
}

int main()
{
  vector<Vehicule *> listeVehicules;

  for(int i= 0; i<5;i++)
  {
    listeVehicules.push_back(dynamic_cast<Vehicule *>(new Velo())) ;
    listeVehicules.push_back(dynamic_cast<Vehicule *> (new Voiture())) ;
  }

  afficheDescriptions(listeVehicules);

  return 0;
}
```

Listing 1.18 – main_vehicules6.cpp

Une des conséquences directes de l'emploi d'une classe abstraite est qu'un pasage par valeur d'une classe abstraite n'est pas valable :

```
void f(Vehicule v)
{
}
```

En effet, comme `Vehicule` ne peut pas être instanciée, il n'est en particulier pas possible d'en faire une copie. Le code en question ne compilera donc pas. Si l'on souhaite néanmoins faire ce genre de passage, il faut passer par un pointeur (et pas une référence, qui doit toujours pointer sur quelque chose d'existant) :

```
void f(Vehicule * v)
{
}
```

1.6 Surcharge

1.6.1 Construction et destruction

Que se passe-t-il lorsque l'on surcharge le constructeur d'une classe dérivée ? Le constructeur de la classe mère est d'abord appelé, suivi par le constructeur

de la classe fille. Pour le destructeur, c'est logiquement l'inverse : le destructeur de la classe fille est appelé d'abord, suivi par le destructeur de la classe mère.

Par exemple, considérons les deux classes suivantes, dont la seule capacite est d'afficher quelque chose lorsqu'elles sont instanciees :

```
class Mere
{
  public:
    Mere();
    ~Mere();
};

class Fille : public Mere
{
  public:
    Fille();
    ~Fille();
};
```

Listing 1.19 – Une classe mere et une classe fille

```
#include "surchargeConstructeur.h"
#include <iostream.h>

Mere::Mere()
{
  cout << "Constructeur Mere" << endl;
}

Mere::~Mere()
{
  cout << "Destructeur Mere" << endl;
}

Fille::Fille()
{
  cout << "Constructeur Fille" << endl;
}

Fille::~Fille()
{
  cout << "Destructeur Fille" << endl;
}
```

Listing 1.20 – Une classe mere et une classe fille

ainsi que le programme principal suivant :

	public	protected	private
public	public	protected	non visible
protected	protected	protected	Non visible
private	private	private	Non visible

TABLE 1.2 – Visibilité et type d'héritage

```
#include <iostream.h>
#include "surchargeConstructeur.h"

int main()
{
  Fille f;

  return 0;
}
```

Listing 1.21 – Programme principal

Conformément à l'ordre d'appel des constructeurs, le programme affichera :

1.6.2 Héritage et visibilité

Nous avions mentionné dans la section METTRE REF qu'il était possible de déclarer une classe dérivée au moyen de trois mots clés différents : `private`, `protected`, et `public`. Jusqu'à présent, nous n'avons employé que le mot clé `public`. Que signifie-t-il précisément ?

Chacun d'entre eux agit comme une sorte de filtre sur la visibilité des membres de la classe mère.

A AMEliorer

1.7 Héritage multiple

Nous avons vu dans la section précédente que l'héritage permet de décrire une relation "est un" entre deux objets. Cependant, comment faire dans le cas où un objet "est un" X *et* un Y ? L'héritage multiple permet de résoudre ce problème. Avant de poursuivre, il me paraît bon de préciser que l'héritage multiple peut être *dangereux*[1], pour des raisons qui deviendront claires par la suite.

1.7.1 Principe

L'idée est assez naturelle : nous allons faire hériter notre classe dérivée de *deux* classes mères. D'un point de vue syntaxique, on écrira :

```
  class Fille :
    public|protected|private class Mere1,
```

1. Il est considéré comme suffisamment dangereux pour être explicitement interdit dans certains langages, JAVA étant sans doute le plus célèbre exemple. Cela ne veut pas dire qu'il ne faut jamais s'en servir, mais simplement qu'il faut être conscient des difficultés associées.

```
    public|protected|private class Mere2,
    . . .
    public|protected|private class Meren
  {
  }
```

Par exemple, considérons les classes de véhicules précédentes, et ajoutons une nouvelle classe représentant un avion (ou tout autre objet volant), qui va également dériver de `vehicule`. Celui-ci aura une nouvelle propriété qui sera l'altitude maximum à laquelle le véhicule peut voler.

```
#include "vehicule4.h"
class VehiculeVolant : public virtual Vehicule
{
  public:
    VehiculeVolant();
    ~VehiculeVolant();

    void setMaxAltitude(int maxAltitude);
    int getMaxAltitude();
    void description();

  private:
    int m_MaxAltitude;
}
```

Listing 1.22 – Véhicule volant

Supposons à présent que l'on souhaite manipuler une voiture volante. Une telle voiture est à la fois un véhicule volant et une voiture. Nous pouvons donc avoir recours à l'héritage multiple et écrire :

```
#include "voiture4.h"
#include "avion4.h"

class VoitureVolante : public Voiture, public VehiculeVolant
{
  public:
    VoitureVolante();
    ~VoitureVolante();
}
```

Listing 1.23 – voiturevolante.h

Nous pouvons représenter l'ensemble des relations entre nos classes sur le diagramme de la figure **??**.

Jusqu'ici, les choses paraissent assez naturelles. Cependant, nous pouvons remarquer que `VoitureVolante` dérive de `Voiture` et `VehiculeVolant`, qui dérivent elles-même de la même classe de base `Vehicule`. Si l'on regarde le dessin formé par les boîtes décrivant les objets et les flèches les reliant (figure **??**), on constate que l'ensemble forme - au moins approximativement - un losange, que l'on appelle en anglais *diamond*. C'est là que se trouve la difficulté principale de

l'héritage multiple, que nous allons expliciter à présent.

1.7.2 Le problème du losange

Le problème est le suivant : la méthode `description` a été surchargée dans `VehiculeVolant` comme dans `Voiture`. Supposons maintenant que nous ajoutions une méthode à `VehiculeVolant` qui fasse appel `description` :

```
1 VehiculeVolant::methodeSupplementaire()
2 {
3   description();
4 }
```

À la compilation, cette méthode va provoquer une erreur. Pourquoi ? Le problème est que la méthode `description` qui a été définie dans les classes mères est héritée *deux fois*. L'appel à la méthode `description` est donc ambigu. Une fois de plus, le langage C++ nous fournit une technique permettant de lever cette ambiguité, technique qui porte le nom d'*héritage virtuel*.

1.7.3 Héritage virtuel

La technique consiste simplement à préciser au compilateur qu'il ne doit hériter des méthodes de la classe `Vehicule` qu'une seule fois, et non deux. Cela se fait au moyen du mot clé `virtual` que l'on écrira devant le nom de la classe dont on hérite :

```
1 class Fille : public virtual Mere
2 {
3 }
```

Les headers de nos classes `VehiculeVolant` et `Voiture` deviendront donc :

```
1  #include "vehicule4.h"
2  class VehiculeVolant : public virtual Vehicule
3  {
4    public:
5      VehiculeVolant();
6      ~VehiculeVolant();
7
8      void setMaxAltitude(int maxAltitude);
9      int getMaxAltitude();
10     void description();
11
12   private:
13     int m_MaxAltitude;
14 }
```

Listing 1.24 – vehiculeVolant4.h

```
#include "vehicule4.h"

class Voiture : public Vehicule
{
    public:
        Voiture(int etat = 1, int fuel = 100,
    int couleur = 0, int prix = 10000);
        ~Voiture();

        virtual void description();
        void setFuel(int fuel);
        int getFuel();

    private:
        int m_fuel;
};
```

Listing 1.25 – voiture4.h

Nous pouvons noter la présence du mot clé `virtual` devant `vehicule` dans la déclaration d'héritage.

1.8 Méthodes amies

Un cas un peu particulier peut parfois se poser. Supposons que l'on dispose de deux classes, chacune avec des membres privés et des membres publics. Le code peut se présenter comme suit :

```
class A
{
  public:
    A(int a);
    ~A();
  private:
    int m_private_a;
};
```

Listing 1.26 – A.h

```
A::A(int a)
{
  m_private_a = a;
}

A::~A()
{
}
```

Listing 1.27 – A.cpp

```
class B
{
  public:
    B(A a);
    ~B();
    int add_a_to_x(int x);

  private:
    A m_a_instance;
};
```

Listing 1.28 – B.h

```
B::B(A a)
{
  m_a_instance = a;
}

B::~B()
{
}

int B::add_a_to_x(int x)
{
  return m_a_instance.m_private_a + x;
}
```

Listing 1.29 – B.cpp

Nous constatons que dans la méthode `add_a_to_X` cherche à utiliser le membre privé `m_private_a` de la classe A. Comme ce membre est privé, , la méthode n'y a pas accès, et le code ne compilera pas.

Une solution simple serait d'utiliser des accesseurs - c'est-à-dire de rajouter des méthodes `get_a` et `set_a` à la classe A, puis de les appeler depuis B. Cependant, dans certains cas bien précis [2], on ne peut pas avoir recours à cette technique.

L'idéal serait de permettre un accès *séléctif* aux membres privés de A, c'est-à-dire de réserver cet accès à certaines classes seulement. Ceci est possible en C++, il suffit de la déclarer *amie*, au moyen du mot clé `friend`.

2. Il est malheureusement difficile de construire un exemple à la fois naturel et simple.

```
1  class A
2  {
3    /*B est une classe amie*/
4    friend B;
5
6    public:
7      A(int a);
8      ~A();
9    private:
10     int m_private_a;
11 };
```

Listing 1.30 – A_friend.h