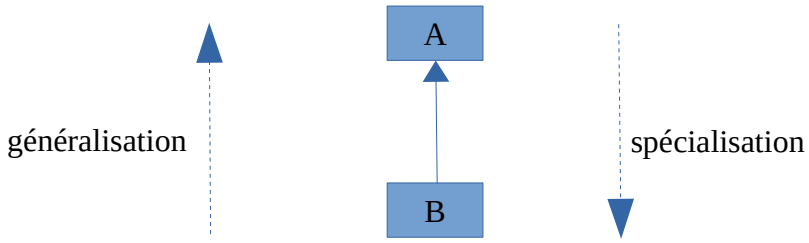
UE D – Introduction au C++ : Héritage

1. Introduction

1.1. Notion d'héritage

L'héritage est un concept fondamental en programmation orientée objet. Ce principe est fondé sur des classes «dérivés» qui héritent des caractéristiques des classes «parentes». L'héritage permet d'ajouter des propriétés à une classe existante pour en obtenir une nouvelle plus précise. Il permet donc la spécialisation ou la dérivation de types.



En programmation, l'héritage permet de regrouper des **caractéristiques communes** dans une classe de base (super-classe) dont héritent des classes dérivées (sous-classes) qui en sont des versions enrichies, étendues. Ce concept permet d'établir une relation « est-un » : si B hérite de A, alors ont dit que « B est un A », c'est-à-dire que toute instance de B est aussi une instance de A.

On dit aussi que B dérive de A ; A est une généralisation de B et B est une spécialisation de A.

De ce fait, l'ensemble des attributs et méthodes de A (sauf le constructeur et destructeur) est disponible dans B. B est enrichie par des attributs et méthodes supplémentaires et spécialisée si elle redéfinit des méthodes héritées de A.

1.2. Héritage des propriétés

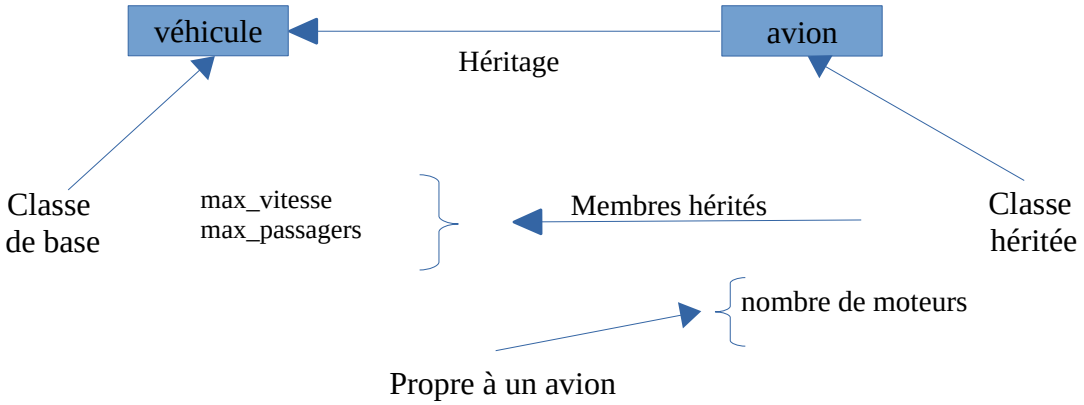
Une classe dérivée modélise un cas particulier de la classe de base. En utilisant l'héritage, il est possible :

- d'ajouter de nouvelles données membres;
- d'utiliser les données membres héritées ;
- de redéfinir les comportements de méthodes héritées.

La classe dérivée hérite des membres de la classe de base.

Pour illustrer cette notion, on peut imaginer vouloir représenter les moyens de transport en définissant par exemple une classe de base **`Vehicule`** ayant les caractéristiques communes à l'ensemble des moyens de transport comme attributs, par exemple `max_vitesse`, `max_passagers`. Les classes

dérivées **`Avion`**, **`Voiture`** et **`Moto`** sont des versions spécifiques de la classe **`Vehicule`**, comme le montre la figure suivante :



1.3. Définition d'une sous-classe

Pour faire hériter une classe dérivée d'une classe de base, on emploie la syntaxe suivante lors de sa déclaration :

```
class Classe_Dérivée : protection Classe_de_base {

// Déclaration des attributs et méthodes spécifiques à la sous-classe

};
```

```
#include <iostream>

using namespace std;

class Vehicule {
   private :
   double max_vitesse;
   int max_passagers;

   public:
   // fonction remplace le constructeur
       void init_vehicule(double v,int np) {
        max_vitesse = v;
        max_passagers = np;
   }
   // Pour afficher les membres private
   void affiche() {
       cout << "vitesse: " << max_vitesse <<
       " ; nbre_passagers: " << max_passagers << endl;
   }
} ;
```

Les types de protections sont : « public », « protected » et « private ».

```
class Avion : public Vehicule {
   int nbre_moteurs;
   public:
   // fonction remplace le constructeur
   void init_avion(int nm) {
       nbre_moteurs = nm;
   }
};


class Moto : public vehicule {
   double cylindree;
   public:
   // fonction remplace le constructeur
   void init_moto(double cy) {
       cylindree = cy;
   }
};
```

1.4. Accès aux membres de la classe de base

Lorsqu'une classe hérite d'une autre, elle reçoit toutes les propriétés de celle-ci. Cela signifie que les membres de la classe de base peuvent donc être utilisés syntaxiquement tout comme les membres de la classe dérivée définis directement dans celle-ci.

Les méthodes d'une classe peuvent manipuler les membres de la classe de base directement par leur nom. De même, l'accès à un objet s'effectue directement grâce aux noms des membres.

```
class base1 {
   private :
   int b1_i;
   int zone;
} ;

class base2 : public base1
{
   int b2_i;
   int zone;
} ;

class base3 : public base2
{
   int b3_i;
} ;


int main(int argc, char **argv)
{
   base3 x;
   x.b2_i = 0;            // accès à la variable b2_i de base2
   x.zone = 0;            // accès à la variable zone de bas2
   x.base1::zone;   // accès à la variable zone de base1
   return 0;
}
```

Dans l'exemple donné plus haut, on peut accéder ainsi aux membres de l'objet x. Il est aussi possible de spécifier directement la classe de base lors de l'accès à un membre, en utilisant **un opérateur de**

portée. Cette méthode est utilisée surtout pour les membres homonymes entre une classe de base et une classe dérivée.

Pour accéder à la variable zone de la classe base1, il faut obligatoirement expliciter sa classe, cela grâce à l'opérateur de résolution de portée.

1.5. Héritage et droits d'accès

Une classe dérivée ne peut jamais avoir plus de droits d'accès sur les membres de la classe de base que n'en prévoit la classe de base elle-même !!!

Si la classe dérivée **hérite publiquement** de la classe de base :

- les membres de la classe dérivée auront accès aux membres publics (champs et méthodes) de la classe de base,
- par contre, les membres privés ne sont disponibles que pour les méthodes de leur propre classe et non pour celles des classes dérivées.

L'exemple suivant illustre les accès possibles sur notre exemple de moyens de transport.

```
int main() {
   Avion boeing767;
   boeing767.init_avion(2);
   // Fonctions héritées de la classe de base.
   boeing767.init_vehicule(950,200);
   boeing767.affiche(); // affiche: 950, 200 mais pas 2.
   // Err: accès à des données privées de la classe de base.
   boeing767.max_vitesse=800;
   boeing767.max__passagers=188;
   // Err: accès à des données privées de la classe dérivée.
   boeing767.nbre_moteurs=4;
   return 0;
}
```

Un troisième type d'accès régit l'accès aux attributs/méthodes au sein d'une hiérarchie de classes :

- l'accès protégé assure la visibilité des membres d'une classe dans les classes de sa descendance. Le mot clé est «protected».

Le niveau d'accès protégé correspond à une extension du niveau privé permettant l'accès aux sous-classes… **mais uniquement dans leur portée** (de sous-classe), et non pas dans la portée de la super-classe.
L'exemple ci-dessous illustre l'accès possible d'une classe dérivée B à l'attribut protégé a de la classe de base A, mais uniquement dans sa portée. Un attribut privé de A n'est pas accessible en dehors de la classe de base. Au sein de B, on peut accéder au a protégé de l'instance courante ou d'une autre instance autre B de la même classe, mais pas à celui d'un objet d'une autre classe comme A : il s'agirait alors d'un accès externe à un attribut protégé, interdit.

1.6. Héritage et masquage (redéfinition)

```
class A {
// ...
   protected: int a;
   private:int prive;
};

class B: public A {
   public:
   // ...
   void f(B autreB, A autreA, int x) {
   a = x;                    // OK A::a est protected => accès possible
   prive = x;          // Erreur : A::prive est private
   a += autreB.prive; // Erreur (même raison)
   a += autreB.a;      // OK : dans la même portée (B::)
   a += autreA.a              // INTERDIT ! : this n'est pas de la même portée que autreA
};
}
```

Comme nous l'avons vu précédemment, avec l'héritage il est possible de redéfinir les comportements de méthodes héritées .

Dans notre exemple sur la modélisation des moyens de transports, la fonction `affiche()` est membre de la classe de base `Véhicule` :

- Elle n'affiche que les membres privés de cette classe.
- On ne peut pas donc afficher le nombre de moteurs.

Pour résoudre ce problème, il suffit de redéfinir dans la classe dérivée la fonction `affiche()`qui aura pour rôle d'afficher les données privées de la classe dérivée.

```
class Avion : public Vehicule {
   int nbre_moteurs;
   public:

   // fonction remplace le constructeur
   void init_avion(int nm) {
       nbre_moteurs = nm;
   }

   // Pour afficher les membres private
   void affiche() {
       Vehicule::afficher();
       cout << " nbre_moteurs: " << nbre_moteurs << endl;
   }

};
```

La nouvelle définition de `Avion::affiche()` cache l'ancienne définition (celle de `Véhicule::affiche`) accessible via héritage.

Toutefois, la classe `Avion` n'a pas le droit d'accéder aux membres privés de la classe de base. Se pose donc la question de comment afficher les données privées de la classe de base et celles de la classe dérivée, cela par l'utilisation d'une fonction dans la classe dérivée.

Puisque `afficher()` de la classe de base (ici `Véhicule`) est accessible, on fera donc appel à elle à partir de la fonction `afficher()` de la classe dérivée en utilisant l'opérateur de résolution de portée.

► Il ne faut pas confondre la **redéfinition** (overriding) et la **surdéfinition** ou surcharge (overloading) :

- Une **surdéfinition** (ou surcharge) permet d'utiliser plusieurs méthodes qui portent **le même nom au sein d'une même classe avec une signature différente**.
- Une **redéfinition** (overriding) permet de fournir une nouvelle définition d'une méthode d'une classe ascendante pour la remplacer. **Elle doit avoir une signature rigoureusement identique à la méthode parente**.

1.7. Héritage & constructeurs

L'instanciation d'une classe s'accompagne d'une initialisation des attributs. Lors de l'instanciation d'une classe dérivée, il faut initialiser :

- les attributs propres à la classe dérivée ;
- les attributs hérités des classes de base.

Cette tâche ne peut plus être intégralement prise en charge par le constructeur d'une classe dérivée : celle-ci hérite des attributs, parfois privés, d'une classe de base, qu'elle ne peut pas initialiser.

L'initialisation des attributs hérités doit se faire dans la classe où ils sont explicitement définis : chaque constructeur de la classe dérivée fait donc appel à un constructeur de la classe de base.

En C++, l'appel au constructeur de la classe de base depuis le constructeur de la classe dérivée se fait dans la liste d'initialisation :

```
Classe_dérivée (liste de paramètres)
              : Classe_de_base (Arguments),
                    Attribut1(valeur1),
                    ... {
              // Corps du constructeur
}
```

Le constructeur de la classe de base porte son nom et est placé au début de la section d'appel aux constructeurs des attributs.

Reprenant à nouveau l'exemple des moyens de transports. Les méthodes `init_avion()` et `init_vehicule()` des classes **`Vehicule`** et **`Avion`** peuvent être remplacées par des constructeurs comme suit :

```
#include <iostream>

using namespace std;

class Vehicule {
   private :
   double max_vitesse;
   int max_passagers;

   public:
   //Constructeur de la classe
       Vehicule(double v,int np): max_vitesse(v), max_passagers (np) { }

   //Déstructeur
   ~vehicule() {}

   void affiche() {
       cout << "vitesse: " << max_vitesse <<
       " ; nbre_passagers: " << max_passagers << endl;
   }
} ;
```

```
class Avion : public Vehicule {
   int nbre_moteurs;
   public:
   // Constructeur de la classe dérivée
   Avion (double v, int np, int nm) : Vehicule(v,np) {
        nbre_moteurs = nm;
   }
   //Déstructeur
   ~Avion() {}
   // Pour afficher les membres private
   void affiche() {
        Vehicule::afficher();
        cout << " nbre_moteurs: " << nbre_moteurs << endl;
   }
};
```

- Pour construire un avion , il faut construire d'abord un véhicule.
- Le constructeur de la classe de base (**`Véhicule`**) est donc appelé **avant** le constructeur de la classe dérivée (**`Avion`**).
- De façon symétrique, le destructeur de la classe de base (**`Véhicule`**) est appelé **après** le destructeur de la classe dérivée (**`Avion`**).
- Si la classe de base a un constructeur autre que celui par défaut, la classe dérivée doit avoir un constructeur associé à ce dernier, sinon il est impossible de créer un objet.
- Si dans l'appel du constructeur de la classe dérivée, le nom du constructeur de la classe de base n'est pas mentionné explicitement, le **constructeur par défaut de la classe de base sera pris en compte**.
- Si la classe de base ne possède pas de constructeur par défaut, l'invocation explicite d'un de ses constructeurs est obligatoire dans les constructeurs de la classe dérivée, sinon il y aura alors une erreur de compilation.

ORDRE D'APPEL DE CONSTRUCTEURS

Comme indiqué précédemment, dans une relation d'héritage, la construction d'une classe dérivée appelle d'abord le constructeur de la classe de base la plus générale puis, dans l'ordre, les constructeurs des super-classes qui en héritent, avant de terminer par l'initialisation de la partie spécifique de la classe instanciée.

Soit, par exemple, une classe **`base3`** , héritant d'une classe **`base2`** qui elle-même hérite d'une classe **`base1`**, relation illustrée par la figure ci-dessous. Le constructeur de **`base3`** reçoit les arguments 1, 2 et 3. Il commence par appeler le constructeur de **`base2`** avec les valeurs 2 et 3, qui appelle à son tour le constructeur de **`base1`** avec la valeur 3.

C'est d'abord un objet de la classe **`base1`** qui est crée, puis un objet de la classe **`base2`**. Le constructeur de la classe **`base3`** en dernier.

```
class base1 {
  int b1_i;
  public :
  base1(int i) : b1_i(i) {}
} ;

class base2 : public base1
{
  int b2_i;
  public :
  base2(int i, int j) : base1(j), b2_i(i) {}
} ;

class base3 : public base2
{
  int b3_i;
  public :
  base3(int i, int j, int k) : base2(j,k), b3_i(i) {}
} ;

int main(int argc, char **argv)
{
  base3 x(1,2,3);
  return 0;
}
```

1.8. Constructeur de copie

Une redéfinition du constructeur de copies d'une classe dérivée s'accompagne toujours d'un appel explicite au constructeur de copies de la classe de base ; sinon, son constructeur par défaut est appelé et la copie est souvent mal réalisée.

```
class B {
      /* classe de base */
};

class D:public class B {
      /* classe dérivée */
};
```

Deux cas peuvent se présenter:

1. La classe D n'a pas de constructeur de recopie:
   - Les appels des constructeurs se feront comme suit:
     - constructeur de recopie par défaut de la classe D,
     - constructeur de recopie explicite ou par défaut de la classe B.

2. La classe D a un constructeur de recopie:
   - Les appels des constructeurs se feront comme suit:
     - le compilateur appelle ce constructeur de recopie. C'est à ce constructeur de recopie d'appeler celui de la classe de base. Si l'on ne fait pas cet appel et, si la classe de base n'a pas de constructeur de recopie explicite, alors c'est le constructeur par défaut qui est appelé. Si la classe de base n'en possède pas un, c'est le constructeur avec arguments par défaut qui est appelé. S'il n'en existe pas un, il y aura erreur de compilation.

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

class Vehicule {
   private :
   double max_vitesse;
   int max_passagers;

   public:
   Vehicule(double v,int np);              // Constructeur de la classe
   Vehicule(const Vehicule& v2);     // constructeur de recopie.

   ~Vehicule();                            // destructeur.
   void afficher()                   // Pour afficher les membres private.
} ;

class Avion : public Vehicule {
   int nbre_moteurs;

   public:
   Avion(int,double,int);     // constructeur.
   Avion(const Avion&);              // constructeur de recopie.

   ~Avion();                  // destructeur.
   void affiche();            // Pour afficher les membres private.
};

// Constructeur de véhicule.
Vehicule::Vehicule(double v,int np): max_vitesse(v), max_passagers(np) {
   cout << "Constructeur.Veh.: " << this << endl;
}

// Destructeur de véhicule.
Vehicule::~Vehicule() {
   cout << "Destructeur.Veh.: " << this << endl;
}

// Constructeur de recopie.
vehicule::vehicule(const vehicule& v2): max_vitesse(v2.max_vitesse),
       max_passagers(v2.max_passagers) {
       cout << "Recopie.Veh.: " << this << endl;
}

// Constructeur d'avion
Avion::Avion(int nm, double v, int np): Vehicule(v,np), nbre_moteurs(nm) {
   cout << "Constructeur.Av.: " << this << endl;
}

// Constructeur de recopie d'avion
Avion::Avion(const avion& a2):Vehicule(a2) {
   nbre_moteurs = a2.nbre_moteurs;
   cout << "Recopie.Av.: " << this << endl;
}

// Destructeur de avion.
Avion::~Avion() {
   cout << "Destructeur.Av.: " << this << endl;
}

// Fonction pour afficher les données private
void Vehicule::affiche() { ... }
void Avion::affiche() { ...}
```

1.9. Copie profonde

Dans des situations où les attributs sont des pointeurs, le constructeur de copie doit être redéfini pour réaliser une copie profonde **qui ne copie pas les adresses mais duplique les zones pointées.** Les attributs d'un objet copié devraient pointer vers des variables à même valeur mais **stockées dans des zones mémoires distinctes de l'objet d'origine**. Cette indépendance implique que la manipulation de l'objet copié (et notamment, sa destruction) n'a plus aucune incidence sur l'objet d'origine.

Dans le constructeur de copie, il convient donc d'allouer de nouvelles zones mémoires, qui contiennent les valeurs pointées par l'objet à copier. Un exemple est proposé ci-dessous. Dans une relation d'héritage, il faudra explicitement faire appel au constructeur de la classe de base, comme vu précédemment.

```
class Rectangle {
   public:
   Rectangle(double l, double h)
        : largeur(new double(l)), hauteur(new double(h)) {}
   Rectangle(const Rectangle& obj);

   ~Rectangle();

   // Note: il faudrait aussi redefinir operator= !
   private:
   double* largeur;
   double* hauteur;
};
// constructeur de copie
Rectangle::Rectangle(const Rectangle& obj):
       largeur(new double(*(obj.largeur))),
       hauteur(new double(*(obj.hauteur))) {}
// destructeur
void Rectangle::~Rectangle() {
   delete largeur;
   delete hauteur;
}
```

Si une classe contient des pointeurs, penser à la copie profonde (au moins se poser la question) :

- constructeur de copie ;
- surcharge de l'opérateur = ;
- destructeur.

► Si l'on redéfinit le constructeur de copie d'une classe dérivée, penser à explicitement mettre l'appel au constructeur de copie de la classe de base (sinon c'est le constructeur par défaut de la classe de base qui est appelé ! !)

1.10. Opérateur d'affectation

Deux cas peuvent se présenter:

1. La classe dérivée n'a pas sur-défini l'opérateur d'affectation =
   - Dans ce cas, le compilateur appelle:
     - l'opérateur = de la classe de base (par défaut ou sur-défini).
   - Sinon:
     - l'opérateur = par défaut de la classe dérivée.
2. La classe dérivée a sur-défini l'opérateur d'affectation =
   - Dans ce cas, le compilateur appelle seulement cet opérateur. À lui d'appeler l'opérateur = de la classe de base s'il veut.

Pour l'exemple de moyens de transport, nous obtenons ce qui suit:

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

class Vehicule {
   private :
   double max_vitesse;
   int max_passagers;

   public:
   //etc
   Vehicule & operator=(const Vehicule&);
} ;

class Avion : public Vehicule {
   int nbre_moteurs;

   public:
   // etc
   Avion &operator=(const Avion&);
};

// Opérateur d'affectation de la classe de base.
Vehicule& Vehicule::operator=(const Vehicule& v2) {
   if (this != &v2) {
        max_vitesse = v2.max_vitesse;
        max_passagers = v2.max_passagers;
   }
   cout << "A.Veh.: " << this << endl;
   return *this;
}

// Opérateur d'affectation de la classe dérivée.
Avion& Avion::operator=(const Avion& a2) {
   if (this != &a2) {
        Vehicule::operator=(a2);
        nbre_moteurs = a2.nbre_moteurs;
   }
   cout << "A.av.: " << this << endl;
   return *this;
}
```

1.11. Transtypage ou conversion de type C++

Le transtypage (cast ou conversion de type) en C ++ permet la conversion d'un type vers un autre. Avec l'héritage, on distinguera :

- transtypage « ascendant » (upcast) : changer un type vers son type de base (cela ne pose pas de problème)
- transtypage « descendant » (downcast) : conversion d'un pointeur sur un objet d'une classe générale vers un objet d'une classe spécialisée.

Traiter un type dérivé comme s'il était son type de base est appelé **transtypage ascendant**. Cela ne pose donc aucun problème. Soit l'exemple suivant:

```
Vehicule v(300,4);
Avion a(800,350,3);
Vehicule* ptrv;
Avion* ptra;
ptrv = &v;
ptra = &a;
```

```
1) v = a ; // ok.
2) cout << v.vitesse << end; // ok.
3) cout << v.nbre_moteurs << endl; // erreur
4) a = v; // erreur
5) ptra = ptrv; // erreur
```

L'affectation (1) ne génère aucune erreur. Le compilateur fait une copie en ignorant les membres excédentaires (`nbre_moteurs`).

Supposons que tous les membres des classes base et dérivée ont été déclarés public, l'instruction (2) est correcte, alors que (3) va générer une erreur car véhicule n'a pas d'information sur le nombre de moteurs

La conversion d'un pointeur sur un objet d'une classe générale vers un objet d'une classe spécialisée est un transtypage « descendant » (downcast). Le compilateur ne l'accepte pas. Par exemple, l'affectation (4) va générer une erreur car un véhicule n'est pas forcément un avion. Idem pour l'affectation (5).

Pour que ça marche, Il faut forcer la conversion (forcer le changement de type) en utilisant l'opérateur de transtypage dynamic_cast : `ptra = dynamic_cast<Avion *> (a);`

2. Polymorphisme et méthodes virtuelles

La dernière notion fondamentale de la programmation orientée objet est le polymorphisme. Il représente la capacité du système à choisir dynamiquement la méthode qui correspond au type de l'objet en cours de manipulation. Les langages orientés objet utilisent le concept d'association tardive pour déterminer la méthode à invoquer pendant l'exécution, en fonction de la nature réelle des instances.

2.1. Exemple introductif

Pour illustrer le concept de polymorphisme considérant l'exemple permettant de représenter différentes formes géométriques à partir d'une classe **`Forme`**.

```
class Forme {
   public:
        Forme() { cout << "constructeur Forme <|- "; }
        void dessiner() { cout << "je dessine ... une forme ?\n"; }
};

class Cercle : public Forme {
   public:
        Cercle() { cout << "Cercle\n"; }
        void   dessiner() { cout << "je dessine un Cercle !\n"; }
};

class Triangle : public Forme {
   public:
        Triangle() { cout << "Triangle\n"; }
        void dessiner() { cout << "je dessine un Triangle !\n"; }
};
void faireQuelqueChose(Forme &f)
{
   f.dessiner(); // dessine une Forme
}

int main()
{
   Cercle c;
   Triangle t;
   faireQuelqueChose(c); // avec un cercle
   faireQuelqueChose(t); // avec un triangle
   return 0;
}
```

L'exécution du programme ci-dessus nous montre que nous n'obtenons pas un comportement polymorphe puisque c'est la méthode dessiner() de la classe Forme qui est appelée :

```
constructeur Forme <|- Cercle
constructeur Forme <|- Triangle
je dessine ... une forme ?
je dessine ... une forme ?
```

Pour obtenir un comportement polymorphe, il est nécessaire déclarer la méthode `dessiner()` comme **virtuelle**. Cela permet au compilateur de choisir la version de la méthode à utiliser qu'au moment de l'exécution. C'est le principe de la **résolution dynamique des liens**.

Pour avoir recours à la résolution dynamique des liens, il est nécessaire de réunir les deux conditions suivantes :

1. les méthodes concernées doivent être déclarées comme virtuelles ;
2. elles doivent s'exercer sur les instances réellement concernées grâce à des références ou des pointeurs.

2.2. Méthodes virtuelles

En C++, pour permettre la résolution dynamique des liens, il faut utiliser des méthodes virtuelles au travers de références ou de pointeurs. Ainsi, le choix de la méthode se fait en fonction du type réel de l'instance.

Une méthode à résoudre dynamiquement doit être déclarée comme *virtuelle*, en précédant son prototype par le mot-clé `virtual`. On effectue ce signalement dans la classe la plus générale qui admet cette méthode, par exemple dans la classe **`Forme`** pour la méthode `dessiner()`.

Toute spécialisation d'une méthode virtuelle, dans une classe dérivée, est aussi virtuelle par transitivité, même sans spécifier explicitement le mot-clé `virtual`.

Dans notre exemple, en déclarant virtuelle la méthode `dessiner()`, on obtient un comportement polymorphe.

```
constructeur Forme <|- Cercle
constructeur Forme <|- Triangle
je dessine un Cercle !
je dessine un Triangle !
```

► Si l'argument `f` de la fonction `dessiner()` était passé par valeur, le polymorphisme serait impossible : lors de l'appel, l'objet c du main serait copié dans une variable de type `Forme` en perdant la spécialisation de sa méthode `dessiner()`. Malgré la virtualisation de cette méthode, ce serait donc toujours la version générale qui serait invoquée par la fonction `faireQuelqueChose`.

L'exemple ci-dessous illustre l'emploi de pointeurs dans le cadre du polymorphisme. Une classe dérivée `Dauphin` y hérite d'une classe de base `Mammifere`. Le code de la fonction `main()` déclare tout d'abord un pointeur sur `Mammifere`, qui reçoit l'adresse d'un `Dauphin`. L'objet alloué

dynamiquement fait appel, dans l'ordre, au constructeur de sa classe de base, puis de sa classe spécialisée.

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

class Mammifere {
public:
     Mammifere() { cout << "Naissance d'un nouveau mammifère :  " << endl;}
     ~Mammifere(){ cout << "Disparition d'un mammifère ..." << endl;}
     void manger(){ cout << "C'est l'heure de manger ..." << endl;}
virtual  void avancer(){ cout << "Un grand pas pour l'humanité." << endl;}
} ;

class Dauphin : public Mammifere {
public:
     Dauphin() { cout << "Super ! c'est un petit dauphin" << endl;}
virtual ~Dauphin(){ cout << "Flipper, c'est fini ..." << endl;}
     void manger(){ cout << "Miam, un poisson !" << endl;}
     void avancer(){ cout << "Je nage." << endl;}
};

int main() {
   Mammifere* Flipper(new Dauphin());
   Flipper->avancer();
   Flipper->manger();
   delete Flipper;
   return 0;
}
```

Dans la seconde ligne du `main`, puisque la méthode `avancer` est virtuelle et qu'elle a accès à l'instance réelle par le biais du pointeur, la résolution dynamique des liens appelle la méthode `avancer` du `Dauphin`. Au contraire, une invocation de la méthode non virtuelle `manger` est résolue statiquement : c'est la méthode liée au type de la variable, `Mammifere` , qui s'exécute.

On a vu que le constructeur de la classe dérivée prenait intégralement en charge la construction de l'objet en appelant le constructeur de la classe de base. L'appel des destructeurs peut poser quelques problèmes :

- La destruction de l'objet de Flipper est résolue statiquement et invoque uniquement le destructeur de `Mammifere`. Ainsi, la part de l'objet de type `Dauphin` ne serait pas détruit car le destructeur de Dauphin n'est jamais invoqué. Cela met en évidence un problème de fuite mémoire.
- La solution : déclarer virtuel le destructeur de la classe `Dauphin`. Ainsi, le code appelle d'abord le destructeur spécifique à `Dauphin` , qui affiche « Flipper, c'est fini... », puis le destructeur de `Mammifere`, qui affiche «Disparition d'un mammifère ...».

3. Classes abstraites

Au niveau le plus élevé d'une hiérarchie de classe, il est parfois impossible de définir une méthode générale qui devra pourtant exister dans toutes les sous-classes. Par exemple, on ne sait pas programmer ce que doit faire la fonction `dessiner()` dans le contexte de **`Forme`**. On veut juste

s'assurer de sa présence dans toutes les classes dérivées, sans devoir la définir dans la classe de base.

3.1. Méthodes virtuelles pures

Une méthode virtuelle pure, ou méthode abstraite, est une méthode qui doit exister et être redéfinie dans chaque classe dérivée que l'on souhaite instancier, sans qu'il soit nécessaire de la définir dans la classe de base. Par conséquent, la classe de base ne donne souvent que son prototype sans la définir explicitement. On signale une méthode virtuelle pure en ajoutant = 0 à la fin de son prototype, c'est-à-dire avec la syntaxe donnée ci-dessous :

```
virtual Type nom_methode(liste de paramètres) = 0;
```

Dans notre exemple, il est ainsi possible d'appeler une telle méthode du niveau de la classe de base : la méthode `dessiner()` existe par exemple pour chaque `Forme` , même si l'on ne sait pas dessiner dans le contexte Forme.

```
class Forme {
   public:
       Forme() { }
      virtual void dessiner()  = 0;
};

class Cercle : public Forme {
   public:
       Cercle() { cout << "Cercle\n"; }
       void   dessiner() { cout << "je dessine un Cercle !\n"; }
};

class Triangle : public Forme {
   public:
       Triangle() { cout << "Triangle\n"; }
       void dessiner() { cout << "je dessine un Triangle !\n"; }
};
```

3.2. Classes abstraites et héritage

Une classe qui contient au moins une méthode virtuelle pure est qualifiée de **classe abstraite**. Toute classe dérivée héritant d'une classe abstraite l'est aussi tant qu'elle ne redéfinit pas toutes ses méthodes virtuelles pures héritées.

- On ne peut pas instancier d'objet à partir d'une classe abstraite. Mais on le peut à partir d'une classe dérivée à condition qu'elle définisse complètement la méthode virtuelle pure.

Supposons à présent l'existence d'une méthode dans la classe `Forme` permettant de calculer la surface d'une forme géométrique. Calculer la surface d'une forme géométrique quelconque se révèle difficile, tandis que pour des formes plus concrètes comme un `cercle` ou un `triangle`, on peut la mettre en œuvre. Une telle méthode `surface()` au niveau de la classe de base serait une méthode virtuelle pure.

```
class Forme {
   public:
       Forme() { }
      virtual void dessiner()  = 0;
      Virtual double surface() = 0;
};

class Cercle : public Forme {
   public:
       Cercle() { cout << "Cercle\n"; }
       void   dessiner() { cout << "je dessine un Cercle !\n"; }
       double surface() { return M_PI * rayon * rayon;}

   protected:
       double rayon;
};

class Triangle : public Forme {
   public:
       Triangle() { cout << "Triangle\n"; }
       void dessiner() { cout << "je dessine un Triangle !\n"; }
};
```

La figure ci-dessus illustre cette notion de classe abstraite: la classe `Cercle` n'est pas abstraite puisqu'elle redéfinit les méthodes virtuelles pures héritées de la classe `Forme`, alors que la classe `Triangle` le reste puisqu'elle ne donne pas de définition de `surface()`. On ne pourra donc pas déclarer d'instances de `Triangle`.

Les classes abstraites sont idéales pour construire la racine des arbres d'héritage. Grâce à l'outil des méthodes virtuelles pures, le polymorphisme complète l'abstraction. Elles permettent de définir des concepts génériques communs à toutes les sous-classes mais trop abstraits pour être codés à haut niveau.