Héritage entre classes

1 Introduction

A ce stade, vous connaissez toutes les notions nécessaire pour programmer des classes. Cependant, afin de pouvoir revnediquer que vous comprenez les notions de la programmation orientée objet, vous devez également maîtriser la notion **d'héritage**. Celle-ci se trouve également dans d'autres langages comme Java par exemple.
Imaginons que nous souhaitions programmer une liste d'employés dans une entreprise. Un employé est avant tout, même si parfois on l'oublier, une personne. Toute personne a *un nom*, *un prénom*, *un age*. Cependant tous les employés n'ont pas exactement les mêmes caractéristiques: Salaire (fixe, par prime, par vente, par déplcament), horaires (plein temps, mi-temps, cadre...). Si je possède déjà une classe personne, dois-je absolument recréer complètement une classe secrétaire? Ne pourrais-je pas simplifier mon code simplement en indiquant au compilateur qu'une secrétaire est une personne et qu'elle devrait donc avoir les mêmes attributs?

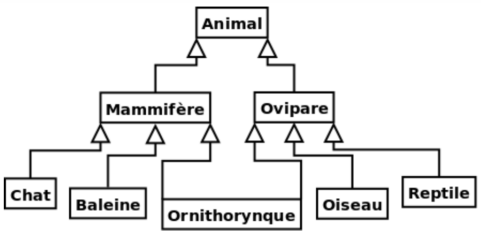
Et bien si, c'est possible. c'est exactement ca la notion d'héritage. De manière générale, lorsque l'on conçoit un code, et si on veut savoir si un héritage est nécessaire entre une classe B et une classe A, la question à se poser est: **B est il un A?**. Si la réponse est oui, alors l'héritage peut (et devrait) être utilisé. **Si B est un A, on dit que B hérite de A (attention pas le contraire) et que B est une classe fille de la classe mère A.**

Quelques exemples:

- *Carre* hérite de *Rectangle* qui lui-même hérite de *Quadrilatere*
- *Boss_De_Fin* hérite de *Monstre*
- *Voiture_Rouge* hérite de *Voiture* qui, comme *Moto* ou *Avion*, hérite de *Vehicule*

Ici vous voyez qu'un carré est un rectangle, alors que le contraire n'est pas vrai. C'est donc *Carre* qui hérite de *Rectangle*. De même, il n'y a aucun lien entre un Avion et un Quadrilatère. Donc il n'y a pas de raison qu'il y ait une notion d'héritage entre *Avion* et *Quadrilatere*.

Souvent, l'héritage entre classes est représenté graphiquement par une flèche de la classe fille pointant vers la classe mère. Cela aide largement, comme par le principe de l'UML à strcturer ses pensées lors de l'éllaboration d'un code. Voici un exemple trouvé sur le net:



Pour ce cours nous prendrons l'exemple de personnes et de plusieurs employés. Notez d'hors et déjà que j'ai utilisé tous les raccourcis de notation pour les constructeurs, que j'ai déclaré ces fonctions simples dans le header (souvent fait pour les getters et setters) afin d'éviter de longues déclarations de fonction dans un cpp. Enfin, j'ai ajouté un destructeur (qui n'est pas nécessaire ici) afin de bien comprendre comment marche la destruction en cas d'héritage.

Ainsi, soit la classe personne comme étant la suivante:

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

class Personne{
private:
    string nom;
    string prenom;
    int age;

public:
    Personne(): nom(" "),prenom(" " ),age(0){};
    Personne(string n, string p, int a): nom(n),prenom(p),age(a){};
    void Affiche(){cout<< "je m'appelle: "<< nom <<", "<<prenom<<"("<<age<<")"<<endl;};
    ~Personne(){cout<<"personne détruite"<<endl;};
};
```

2 L'héritage en mode public: la portée protected

Nous allons créer une classe employe qui hérite de personne, et qui en plus d'un nom et d'un prénom, à aussi un "metier" et un salaire fixe:

```
#include "Personne.h"

class Employe: public Personne{

private:
    string fonction;
    float Salaire_fixe;

public:

};
```

Ici, avant tout, on va oublier temporairement que cette classe n'a pas de constructeur. Simplement en regardant le code de *Employe*, que voyons nous? Afin de signaler au compilateur que *Employe* est une classe fille de *Personne*, nous indiquons la ligne `class Employe: public Personne{` (il ne faut bien sur pas oublier d'cinlure "Personne.h"). Cette ligne indique que *Employe* hérite publiquement de *Personne*. Dans le cadre de ce cours, seul l'héritage public doit être compris. Sachez qu'il est possible d'hériter différemment (nous y reviendrons plus tard).

Maintenant, voyons ce que nous pouvons faire avec *Employe*. Ici nous avons l'impression qu'un Employe n'a ni nom ni prenom ni age. OR ce n'est pas vrai. Puisqu'un *Employe* est une *Personne*, il dispose egalement de toutes les fonctions et données membre de Personne. C'est à dire qu'on peut tout à fait, dans le main faire appel à la fonction `E.Affiche();` (où E est un *Employe*). Un Employe dispose donc d'un nom d'un prenom d'un age et d'une fonction `Affiche()` **sans devoir les spécifier de nouveau**.

La portée Protected

Une fois que ceci est compris regardons-y de plus près. Nous pouvons faire `E.Affiche();` dans le main, car Affiche est déclarée en `public:` dans la classe Personne. Puisque cette fonction est publique, elle est accessible depuis lextérieur de la classe *Personne* et donc depuis la classe *Employe*.

Cependant nous ne pouvons PAS effectuer dans le main `E.nom;` puisque nom est une donnée privée de Personne et donc on ne peut pas l'utiliser à l'extérieur de la classe *Personne*. Puisque ces données sont privées dans *Personne*, cela veut il également dire qu'on ne peut pas les utiliser dans *Employe*? Imaginons que nous

voulions par exemple créer une fonction `set_nom()` dans *Employe* de la manière suivante:

```
class Employe: public Personne{

private:
    string fonction;
    float Salaire_fixe;

public:
    void set_nom(string n){nom=n;};

};
```

Cela fonctionne t'il? La réponse est NON. En effet le champs *nom* est privé dans *Personne*. Ceci signifie qu'il n'est pas utilisable à l'extérieur de la classe, et en particulier dans la classe *Employe*. Ceci signifie qu'un *Employe* possède des champs auquel il ne peut pas accéder. Ca peut devenir relativement fastidieux. Ne serait-il pas possible de créer des données qui soit inaccessibles en dehors de la classe SAUF dans les classes filles? Et bien si, c'est possible, grâce à une nouvelle portée (celles que nous connaissons sont les portées `private` et `public`): la portée `protected`.
Ainsi, nous pouvons modifier la classe mère *Personne* en autorisant les classes filles à accéder aux attributs *nom*, *prenom* et *age*:

```
class Personne{
protected:
    string nom;
    string prenom;
    int age;

public:
    Personne(): nom(" "),prenom(" " ),age(0){};
    Personne(string n, string p, int a): nom(n),prenom(p),age(a){};
    void Affiche(){cout<< "je m'appelle: "<< nom <<", "<<prenom<<"("<<age<<")"<<endl;};
    ~Personne(){cout<<"personne détruite"<<endl;};
};
```

Maintenant la fonction `set_nom` de *Employe* fonctionne. La portée protected est quasiement toujours employée dans les cas d'héritage, sauf conditions particulières où l'on souhaite ne pas pouvoir accéder à un champs depuis une classe fille.

Les Constructeurs

Jusqu'ici, nous n'avons pas parlé du constructeur des classes filles. Il doit y avoir un constructeur. **Les constructeurs et les destructeurs ne sont pas hérités.** En revanche, nous allons les écrire de manière un peu particulière. Un gros avantage de l'héritage est d'économiser du code, de ne pas toujours tout retaper. Puisque lorsque je créée une *Employe*, je créée une *Personne*, nous allons avant toute chose, appeler le constructeur de *Personne* dans le constructeur de *Employe*.

```
class Employe: public Personne{

protected:
    string fonction;
    float Salaire_fixe;

public:
    Employe():Personne(), fonction(" "), Salaire_fixe(0){};
    Employe(string n, string p, int a, string f, float s):Personne(n,p,a),fonction(f),Salaire_fixe(s){}
    void set_nom(string n){nom=n;};

};
```

Ici, pour chaque constructeur de *Employe*, je fais appel au constructeur de la classe mère *Personne*. De même si je faisais une classe *Secretaire*, fille de *Employe* je ferais:

```
class Secretaire: public Employe{
public:
    Secretaire(string n,string p, int a):Employe(n,p,a,"Secrétaire",1137.5){};
};
```

Nous constatons que le constructeur de *Secretaire* fera appel au constructeur de *Employe* qui fera lui-même appel au constructeur de *Personne*. Puisque j'ai défini un Secrétaire comme ayant toujours le même salaire fixe (1137.5 euros), je peux directement faire passer ces paramètres sans les demander à l'utilisateur. Pour créer un secretaire, il n'y aura qu'à taper dans le `main` Secretaire a("Jean-Paul","Lenom",43); pour le créer.

3 Le masquage de fonction

Pour en finir avec l'utilité de l'héritage, voyons le masquage de fonction. Imaginons que lorsque j'affiche un employe j'aimerais afficher "Nom, Prenom (Age) est un fonction et gagne un salaire de X euros". Cependant si `E` est un employe, `E.Affiche();` utilisera la fonction `Affiche()` de la classe mère *Personne*, puisque c'est la seule que le compilateur connaisse.

Comment faire? **Et bien, il est possible de réécrire cette fonction plus spécifique dans la classe fille. Elle masquera la fonction de la classe mère:** Si on veut afficher un *Employe*, le compulateur utilisera la fonction définie dans *Employe*, si on veut afficher une *Personne* on utilisera la fonction définie dans *Personne*. C'est un peu le même principe que la surcharge mais dont l'utilisation est conditionnée par l'objet y faisant appel et non par les paramètres de la fonction.

```
class Employe: public Personne{

protected:
    string fonction;
    float Salaire_fixe;

public:
    Employe():Personne(), fonction(" "), Salaire_fixe(0){};
    Employe(string n, string p, int a, string f, float s):Personne(n,p,a),fonction(f),Salaire_fixe(s){}
    void set_nom(string n){nom=n;};
    void Affiche();
};

void Employe::Affiche(){
    cout<< "je m'appelle: "<< nom <<", "<<prenom<<"("<<age<<")"<<endl;
    cout<< "Je suis un(e)" << fonction<< " et je gagne "<<Salaire_fixe<< " euros"<<endl;
}
```

Dans le main la ligne
E.Affiche(); affichera maintenant nom prenom age fonction et salaire.
Comment faire si je suis un peu tordu et que maintenant, dans ces conditions, j'aimerais tout de même aimé utiliser la fonction Affiche, mais de la classe Personne. Et bien c'est toujours possible grâce à la specification suivante:`E.Personne::Affiche();` Elle spécifie au compilateur que l'on souhaite utiliser la fonction Affiche, mais de la classe personne. D'ailleurs ce genre d'écriture est très utilisée pour éviter de retaper du code déjà tapé. Par exemple on peut redéfinir la fonction `Affiche` de *Employe* de la manière suivante:

```
void Employe::Affiche(){
    Personne::Affiche();
    cout<< "Je suis un(e)" << fonction<< " et je gagne "<<Salaire_fixe<< " euros"<<endl;
}
```

4 Destructeur

Que se passe t'il avec les destructeurs? Créons d'abord un destructeur pour la classe *Employe*:

```
class Employe: public Personne{

protected:
    string fonction;
    float Salaire_fixe;

public:
    Employe():Personne(), fonction(" "), Salaire_fixe(0){};
    Employe(string n, string p, int a, string f, float s):Personne(n,p,a),fonction(f),Salaire_fixe(s){}
    void set_nom(string n){nom=n;};
    void Affiche();
    ~Employe(){cout<<"Employé détruit"<<endl;}
};
```

Maintenant rappelons ce que nous avons au niveau des classes:

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

class Personne{
protected:
    string nom;
    string prenom;
    int age;

public:
    Personne(): nom(" "),prenom(" " ),age(0){};
    Personne(string n, string p, int a): nom(n),prenom(p),age(a){};
    void Affiche(){cout<< "je m'appelle: "<< nom <<", "<<prenom<<"("<<age<<")"<<endl;};
    ~Personne(){cout<<"personne détruite"<<endl;};
};


class Employe: public Personne{

protected:
    string fonction;
    float Salaire_fixe;

public:
    Employe():Personne(), fonction(" "), Salaire_fixe(0){};
    Employe(string n, string p, int a, string f, float s):Personne(n,p,a),fonction(f),Salaire_fixe(s){}
    void set_nom(string n){nom=n;};
    void Affiche();
    ~Employe(){cout<<"Employé détruit"<<endl;}
};

void Employe::Affiche(){
    Personne::Affiche();
    cout<< "Je suis un(e)" << fonction<< " et je gagne "<<Salaire_fixe<< " euros"<<endl;
}

class Secretaire: public Employe{
public:
    Secretaire(string n,string p, int a):Employe(n,p,a,"Secrétaire",1137.5){};
    ~Secretaire(){cout<<"Secrétaire détruit"<<endl;}
};
```

Regardons ce qui se passe si nous effectuons ce main:

```
int main(){
    Personne p("Paul","Jacques",24);
    Employe e("Paula","Franck",24,"Boss",6000);
    Secretaire s("Dylan","Milou",24);

    Personne p2;
    Personne *pptr;

    p.Affiche();
    e.Affiche();
    s.Affiche();

    p2=e;
    p2.Affiche();

    p2=s;
    p2.Affiche();

    pptr=&s;
    pptr->Affiche();

    return(0);
}
```

Alors tout d'abord ce code compile. Cela peut paraître étonnant surtout de par la ligne `p2=e;`, étant donné que p2 est une *Personne* et e est un *Employe*. **Cependant, il ne faut pas oublier que un Employe EST une Personne**. ATTENTION cette affectation ne peut se faire que dans ce sens: la ligne `e=p2;` ne compile pas car une *Personne* n'est pas forcément un *Employe*. C'est la même chose pour les pointeurs.

Ainsi, le code s'execute et écrit :

```
je m'appelle: Paul, Jacques(24)
je m'appelle: Paula, Franck(24)
Je suis un(e)Boss et je gagne 6000 euros
je m'appelle: Dylan, Milou(24)
Je suis un(e)Secrétaire et je gagne 1137.5 euros
je m'appelle: Paula, Franck(24)
je m'appelle: Dylan, Milou(24)
je m'appelle: Dylan, Milou(24)
personne détruite
Secrétaire détruit
Employé détruit
personne détruite
Employé détruit
personne détruite
personne détruite
Program ended with exit code: 0
```

Regardons en détails pour être sûr de bien comprendre:

- On demande à afficher p qui est une Personne et le programme affiche :
  je m'appelle: Paul, Jacques(24).

- On demande à afficher e qui est un Employe. Le compilateur utilise Affiche() définie dans Employe :
  je m'appelle: Paula, Franck(24)
  Je suis un(e)Boss et je gagne 6000 euros

- On demande à afficher s qui est un Secretaire. Le compilateur utilise Affiche() définie dans Employe, puisqu'elle n'est pas redéfinie dans secrétaire :
  je m'appelle: Dylan, Milou(24) (Affichage de s)
  Je suis un(e)Secrétaire et je gagne 1137.5 euros

- Maintenant, on demande d'afficher p2 qui est une Personne. Cependant, p2 est égal à e, un Employe. Quelle fonction Affiche() va t'il utiliser? Comme p2 est une Personne, le compilateur utilise la fonction définie dans Personne, et donc affiche:
  je m'appelle: Paula, Franck(24)

- De même lorsque p2 est un secrétaire:
  je m'appelle: Dylan, Milou(24)
- Et de même lorsqu'on passe par un pointeur pptr:
  je m'appelle: Dylan, Milou(24)
- Maintenant comment se passe la destruction? Vous pouvez constater que le compilateur utilise successivement les destructeurs des classes relatives à l'objet puis successivement ceux des classes mères:
  - Destruction de p:
    personne détruite
  - Destruction de s:
    Secrétaire détruit
    Employé détruit
    personne détruite
  - Dstruction de e:
    Employé détruit
    personne détruite
  - Destruction de p2:
    personne détruite

Ces notions sont importantes et nous verrons plus tard avec le polymorphisme que l'héritage réserve encore d'autres surprises

5 Pour aller plus loin

Je n'en parle pas dans ce cours, mais il est possible de faire hériter une classe autrement que publiquement. Afin de ne pas vous troubler je vous conseille d'aller voir par vous même les changements occasionnés avec un héritage protégé ou privé. Dans le cadre de ce cours, l'héritage public sera suffisant.

6 Conclusions

En résumé, voici ce qu'il faut retenir de ce cours:

- L'héritage permet de spécialiser une classe.
- Lorsqu'une classe hérite d'une autre classe, elle récupère toutes ses données et ses fonctions.
- On parle d'héritage entre une classe A et B, lorsqu'on peut dire que "B est un A".
- La classe de base est appelée classe mère et la classe qui en hérite est appelée classe fille.
- Utiliser les constructeurs des classes mère pour définir les constructeurs des classes filles.
- Une nouvelle portée, moins restrictive que `private`, et plus restrictive que `public` est utilisée: `protected`. Elle défini une portée `public` pour les classes filles mais `private` ailleurs.
- On peut masquer une fonction en utilisant la même déclaration dans une classe fille.