Le polymorphisme

1 Introduction

Imaginons un garage contenant plusieurs types de **Véhicules**, chacun avec des caracteristiques différentes. Tout **Véhicule** par exemple a un prix, une couleur, peut être vendu, acheté, ... Certains types de **Véhicules** ont des caractéristiques particulières, comme un nombre de porte pour une **Voiture**, ou le fait d'être routiere ounon pour une **Moto**. Comme le garage contient beaucoup de **Véhicules**, un garage est en fait un Tableau de **Véhicules**. Ce qu'on aimerait bien, c'est que chaque objet de ce tableau soit géré de manière similaire. Par exemple, si on affiche un véhicule, on aimerait avoir ses caractéristiques de base comme la couleur et le prix, plus le fait de savoir si il s'agit d'une moto ou d'une voiture avec ses caractéristiques spéciales.

2 Un exemple

Dans un but pédagogique, nous allons prendre l'exemple suivant simplifié. Notez que j'ai ajouté des destructeurs même si ils ne sont pas nécéssaires: Je souhaiterais savoir comment se comporte un objet lorsqu'il est détruit.

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

class Vehicule{

protected:
    int nb_places;
public:
    Vehicule(): nb_places(0){};
    ~Vehicule(){cout <<"Vehicule detruit"<<endl;};
    void Affiche() const{cout<<"je suis un Vehicule"<<endl;};


};

class Moto: public Vehicule{
public:
    Moto():Vehicule(){nb_places=2;};
    ~Moto(){cout <<"Moto detruite"<<endl;};
    void Affiche() const{cout<<" je suis une Moto"<<endl;};


};

class Voiture: public Vehicule{
public:
    Voiture():Vehicule(){nb_places=4;};
    ~Voiture(){cout <<"Voiture detruite"<<endl;};
    void Affiche() const{cout<<" je suis une Voiture"<<endl;};


};
```

Selon vous, que va t'il être affiché sur la console lorsque j'effectuerai le main suivant?

```
#include <iostream>
#include "polymorphisme.hpp"
int main(int argc, const char * argv[]) {


    Vehicule Ve;
    Moto m;
    Voiture v;

    Ve=m;
    Ve.Affiche();
    Ve=v;
    Ve.Affiche();

    return 0;
}
```

Tout d'abord il est à noté que le code compile sans problème, que l'on peut écrire `Ve=m` ou `Ve=v` puisque je le rappelle, une **moto** ou une **voiture** SONT des **véhicules**. Ensuite on espère que le code va afficher :

```
je suis une Moto
je suis une Voiture
```

Cependant, ce n'est pas ce qui va se passer voici ce que l'execution va afficher :

```
je suis un Vehicule
je suis un Vehicule
Voiture detruite
Vehicule detruit
Moto detruite
Vehicule detruit
Vehicule detruit
```

Etonnant non? Le code affiche tout de même : "Je suis un Vehicule". Puisque `Ve` est un **véhicule**, il n'y a pas de raison de ne pas appeler la fonction `Affiche` de `Vehicule`. De plus, il est interessant de noter la chose suivante : les destructeurs sont appelés à la chaine à la fin du programme.

Lorsqu'on détruit une Voiture, on appelle d'abord le destructeur de `Voiture`, puis celui de la classe mère, `Vehicule`. De même pour la Moto. Comment dès lors, faire afficher "Je suis une moto" lorsque Ve correspond à une moto, et "Je suis une voiture", lorsque Ve correspond à une voiture? C'est tout le but du **polymorphisme**.

3 Le polymorphisme

Tout d'abord afin que le polymorphisme fonctionne, il faut des **pointeurs d'objet**. Quoique nous fassions, si nous utilisons le premier `main`, cela ne peut pas fonctionner. Modifions le de façon à avoir des pointeurs:

```
#include "polymorphisme.hpp"
int main(int argc, const char * argv[]) {

    Vehicule *Ve;
    Moto *m=new Moto;
    Voiture *v=new Voiture;

    Ve=m;
    Ve->Affiche();
    Ve=v;
    Ve->Affiche();

    Ve=v;
    delete Ve;
    Ve=m;
    delete Ve;


    return 0;
}
```

Voici ce que l'execution de ce code affiche: `je suis un Vehicule`

```
je suis un Vehicule
Vehicule detruit
Vehicule detruit
```

C'est encore pire. Non seulement, le code affiche toujours des véhicules, mais en plus, il ne fait plus appel dutout au destructeur de `Moto` ni de `Voiture`. Ceci est normal. Dans la version précédente,

le destructeur était appelé naturellement pour `m` et `v` à la fin du code. Puisque maintenant le destructeur est appelé par `delete` nous voyons ici que nous demandons à détruire un `Vehicule` et c'est ce qui se produit.
Comment y remédier?

3.1 Le mot clef : virtual

Nous allons introduire le mot clef `virtual`. Définir une fonction comme virtuelle signifie au compilateur que, si un pointeur de la classe mère, pointe sur un objet de la classe fille, la fonction à utiliser est celle de la classe fille. **Puisque les destructeurs ne sont naturellement pas hérités, il faudra toujours dans le cas du polymorphisme, utiliser des destructeurs virtuels**. Le mot clef virtual est à insérer dans le Header. Il n'a cependant pas besoin d'être rappelé dans le cpp. Ainsi nous pouvons modifier le Header de la façon suivante:

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

class Vehicule{

protected:
    int nb_places;
public:
    Vehicule(): nb_places(0){};
   virtual ~Vehicule(){cout <<"Vehicule detruit"<<endl;};
   virtual void Affiche() const{cout<<"je suis un Vehicule"<<endl;};


};

class Moto: public Vehicule{
public:
    Moto():Vehicule(){nb_places=2;};
  virtual ~Moto(){cout <<"Moto detruite"<<endl;};
  virtual void Affiche() const{cout<<" je suis une Moto"<<endl;};


};

class Voiture: public Vehicule{
public:
    Voiture():Vehicule(){nb_places=4;};
   virtual ~Voiture(){cout <<"Voiture detruite"<<endl;};
   virtual void Affiche() const{cout<<" je suis une Voiture"<<endl;};


};
```

Notez l'introduction du mot-clef virtual. A l'exécution, ce code produit:

```
je suis une Moto
je suis une Voiture
Voiture detruite
Vehicule detruit
Moto detruite
Vehicule detruit
```

Vous pouvez constater que maintenant, la fonction `Affiche` appelée dépend de l'objet sur lequel pointe le pointeur. De même lors de la destruction des objets, le destructeur appelé dépend de l'objet sur lequel `Ve` pointe. C'est cela le polymorphisme. On peut utiliser la même fonction `Affiche` quelque soit le véhicule, avec des effets différents. Cela simplifie grandement les codes.

En conclusion, pour fonctionner, le polymorphisme a besoin de pointeurs, combiné à des fonctions virtuelles.

4 Classes abstraites

Pour revenir à l'exemple donné en introduction, il se peut que l'on puisse spécifier un nombre de places par véhicules. A ce moment, si je souhaite créer une fonction `Affiche_nb_Places()` qui affiche le nombre de places, une moto devrait afficher 2, une voiture devrait afficher 4, mais un véhicule? Un véhicule ne peut rien afficher puisque le nombre de places dépend du véhicule concerné.

Et bien il est possible de définir une fonction `Affiche_nb_Places()` dans la classe `Vehicule` sans pour autant la spécifier. Pour cela il faut que la fonction soit **virtuelle** et on ajoute à la fin de la déclaration `=0`. **C'est ce qu'on appelle une fonction virtuelle pure**.
`virtual Affiche_nb_Places()=0;`

ATTENTION: Toute classe possédant une fonction virtuelle pure devient une "Classe Abstraite". Une classe abstraite se comporte de manière particulière. **On peut créer un pointeur vers un tel Objet, mais on ne peut pas créer un Objet en soit.** Par exemple si nous modifions nos classes de la façon suivante:

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

class Vehicule{

protected:
    int nb_places;
public:
    Vehicule(): nb_places(0){};
    virtual ~Vehicule(){cout <<"Vehicule detruit"<<endl;};
    virtual void Affiche() const{cout<<"je suis un Vehicule"<<endl;};
    virtual void Affiche_nb_places() const=0;

};

class Moto: public Vehicule{
public:
    Moto():Vehicule(){nb_places=2;};
    virtual ~Moto(){cout <<"Moto detruite"<<endl;};
    virtual void Affiche() const{cout<<" je suis une Moto"<<endl;};
    virtual void Affiche_nb_places() const{cout<<"j'ai"<<nb_places<<"places"<<endl;} ;


};

class Voiture: public Vehicule{
public:
    Voiture():Vehicule(){nb_places=4;};
    virtual ~Voiture(){cout <<"Voiture detruite"<<endl;};
    void Affiche() const{cout<<" je suis une Voiture"<<endl;};
    virtual void Affiche_nb_places() const{cout<<"j'ai"<<nb_places<<"places"<<endl;} ;


};
```

x

Et bien la classe Vehicule devient une classe abstraite. Ainsi, dans le main, il sera possible d'écrire `Vehicule *Ve;` mais on ne pourra plus écrire `Vehicule Ve;`. Cela parait handicapant, mais ce qu'on y gagne, c'est de pouvoir utiliser la fonction `Affiche_nb_Places()` de manière équivalente pour tout type de véhicule.

Voici par exemple un exemple d'utilisation du code suivant:

```cpp
#include <iostream>
#include "polymorphisme.hpp"
int main(int argc, const char * argv[]) {


     Vehicule *Ve;
     Moto *m=new Moto;
     Voiture *v=new Voiture;

     Ve=m;
     Ve->Affiche();
     Ve->Affiche_nb_places();
     Ve=v;
     Ve->Affiche();
     Ve->Affiche_nb_places();


     Ve=v;
     delete Ve;
     Ve=m;
     delete Ve;


    return 0;
}
```

---

produira l'affichage suivant: `Je suis une Moto`
`J'ai 2 places`
`Je suis une Voiture`
`J'ai 4 places`
`Voiture detruite`
`Vehicule detruit`
`Moto detruite`
`Vehicule detruit`

Nous avons le bon affichage et les bons destructeurs, tout utilsant toujours la fonction sur un pointeur. Afin de pouvoir constater le gain de temps voici un code qui utilise de manière éfficace le code.

5 L'exemple du garage

La où le polymorphisme prend tout son sens est dans l'utilisation de tableaux de pointeurs. Voici une simulation d'un garage simple. Constatez à quel point il aurait été fastidieux de déterminer, pour les 100 véhicules du garage s'il s'agissait d'une moto ou d'une voiture. Ici on ne fait pas de différence.

```
#include <iostream>
#include "polymorphisme.hpp"
int main(int argc, const char * argv[]) {


    Vehicule **Garage =new Vehicule* [100];
    Moto *m;
    Voiture *v;

    for (int i=0;i<50;i++)
    {
        m=new Moto;
        *(Garage+i)=m;
    }

    for (int i=50;i<100;i++)
    {
        v=new Voiture;
        *(Garage+i)=v;
    }


    for (int i=0;i<100;i++)
    {
        (*(Garage+i))->Affiche();
        (*(Garage+i))->Affiche_nb_places();
    }

    for (int i=0;i<100;i++)
    {
       delete (*(Garage+i));
    }

    delete [] Garage;

    return 0;
}
```

---

6 Exercice

Si vous avez compris, déterminez ce que va afficher le code suivant:

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

class A {
    public :
    void M1 () { cout << " A"; }
    virtual void M2 () { cout << " A"; }
    virtual void M3 () { cout << " A"; }
};
class B : public A {
    public :
    virtual void M1 () { cout << " B"; }
    virtual void M2 () { cout << " B"; }
    void M3 () { cout << " B"; }
};
class C : public B {
    public :
    void M1 () { cout << " C"; }
    void M2 () { cout << " C"; }
    void M3 () const { cout << " C"; }
};

int main () {
    A a; B b; C c; A* pa = &b; B* pb = &c; A* pc = &c;
    pa->M1(); pa->M2(); pa->M3(); cout << "\n";
    pb->M1(); pb->M2(); pb->M3(); cout << "\n";
    pc->M1(); pc->M2(); pc->M3(); cout << "\n";
    b = c;
    b.M1(); b.M2(); b.M3(); cout << "\n";
}
```
