COLLEGE DE PARIS SUPÉRIEUR

DAKARTECH

Programmation orientée objets [C++]

Dr. Pape Abdoulaye BARRO

La POO

- Notion de Classe
- Fonctions membres
- Constructeurs, destructeurs et initialisation d'objet
- Héritage simple
- Héritage multiple
- **Fonctions virtuelles et polymorphisme**
- La gestion des exceptions
- Généralités sur la STL

Fonctions virtuelles et polymorphisme

polymorphisme et classes abstraites

Considérons le programme suivant ou Carré dérive de rectangle:

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

class  Rectangle{
        public:
           void toString(){
                cout <<"je suis un rectangle"<<endl;
            }
};

class Carre : public Rectangle {
        public:
           void toString(){
                cout <<"je suis un carre"<<endl;
            }
};
```

```
void main()
{
    Rectangle rect;
    Carre car;

    Rectangle *r =&rect;
    r->toString();
    r= &car;
    r->toString();

    return 0;
}
```

- La situation en est que lorsqu'on appel la fonction toString() de Carre via l'objet pointé, c'est la fonction toString() de la classe Rectangle qui est appelé et non celle réellement pointé.
- Le problème en est que le compilateur ne connait pas le type de l'objet réellement pointé, et se base uniquement sur le type du pointeur. On parle donc de typage statique.
- En C++, il est possible de faire face à cela, en permettant le typage dynamique de ces objets. Un tel mécanisme permettrait au compilateur de choisir à l'exécution, la fonction appropriée. Il s'agit de la notion de Polymorphisme.

Fonctions virtuelles et polymorphisme

polymorphisme et classes abstraites

Supposons que l'on souhaite créer une collection d'objets de type Rectangle, en demandant toString() pour chacun de ces objets. Ce sera automatiquement la version correspondant à chaque forme qui sera appelée et exécutée : *on dit que toString() est polymorphe*.

Ce choix de la version adéquate de toString() sera réalisé au moment de l'exécution.

- **Règle générale**: Toute fonction-membre de la classe parente devant être redéfinie (*surchargée*) dans une classe dérivée doit être précédée du mot-clé *virtual*.
- Exemple:

```
class Rectangle{
    public:
        // fonction destinée à être surchargée
        virtual void toString();
};
```

Fonctions virtuelles et polymorphisme

polymorphisme et classes abstraites

Exemple:

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

class Rectangle{
        public:
            // fonction destinée à être surchargée
            virtual void toString();
};
void Rectangle::toString(){
        cout << "Je suis un rectangle !" <<endl;
}

 class Carre : public Rectangle{
        public:
            void toString();
 };
 void Carre::toString(){
        cout <<"Je suis un carré!"<<endl;
 }
```

```
int main()
{
        Rectangle rect;
        Carre car;

        Rectangle *r= &rect;
        r->toString();
        r= &car;
        r->toString();

        return 0;
}
```

- Seule une fonction membre peut être virtuelle. Pas de fonction indépendante ou de fonction amie.
- Un constructeur ne peut pas être virtuel
- Un destructeur par contre, peut être virtuel

Fonctions virtuelles et polymorphisme

polymorphisme et classes abstraites

Reconsidérons la situation précédente en définissant que des constructeurs et destructeurs des classes concernées:

```
#include<iostream>
#include<string>
using namespace std;
 class  Rectangle{
          public:
             Rectangle(int, int);
             ~Rectangle () {cout <<"Fin Rect"<<endl;}
          protected:
             int longueur, largeur;
};
Rectangle::Rectangle(int lo, int la){
          longueur = lo;
          largeur = la;
}
 class Carre : public Rectangle {
          int c_cote;
          public:
             Carre(int cote);
              ~Carre() {cout <<"Fin Car"<<endl;}
};
Carre::Carre(int cote) : Rectangle(cote, cote) {
          c_cote=cote;
}
```

```
int main()
{
          Rectangle *r= new Carre(5);
          delete r;

          return 0;
}
```

- Dans ce scénario, c'est seulement le destructeur de Rectangle qui est appelé. Celui de Carre n'est pas appelé.
- Pour remédier à cela, le destructeur de Rectangle doit être déclaré comme virtuel :
  - virtual ~Rectangle ();

Fonctions virtuelles et polymorphisme

polymorphisme et classes abstraites

- Une fonction-membre virtuelle d'une classe est dite purement virtuelle lorsque sa déclaration est suivie de **= 0**.

Exemple:

```
class A {
    public:
        virtual void fonct() = 0;
};
```

  - Une fonction purement virtuelle n'a pas de définition dans la classe. Elle ne peut qu'être surchargée dans les classes dérivées.

- Une classe comportant au moins une fonction-membre purement virtuelle est appelée classe abstraite.
  - Aucune instance d'une classe abstraite ne peut être créée.
  - L'intérêt d'une classe abstraite est uniquement de servir de "**canevas**" à ses classes dérivées, en déclarant l'interface minimale commune à tous ses descendants.

Fonctions virtuelles et polymorphisme

polymorphisme et classes abstraites

- Aucune instance d'une classe abstraite ne peut être créée. Exemple:

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

class Rectangle{
        public:
            // fonction purement virtuelle destinée à être surchargée
            virtual void toString()=0;
};

class Carre : public Rectangle{
        public:
            void toString();
};

void Carre::toString(){
        cout <<"Je suis un carré!"<<endl;
}
```

```
void main()
{
      Rectangle *r= new Rectange();
       r->toString();


      return 0;
}
```

Error !

Fonctions virtuelles et polymorphisme

polymorphisme et classes abstraites

- Nous allons juste déclarer la fonctions toString() dans **Rectangle** et puis la redéfinir dans **Carre** . Exemple:

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

class Rectangle{
        public:
            // fonction purement virtuelle destinée à être surchargée
            virtual void toString()=0;
};

class Carre : public Rectangle{
        public:
            void toString();
};

void Carre::toString(){
        cout <<"Je suis un carré!"<<endl;
}
```

```
void main()
{
      Rectangle *c= new Carre();
       c->toString();

       return 0;
}
```

Correcte!

Fonctions virtuelles et polymorphisme

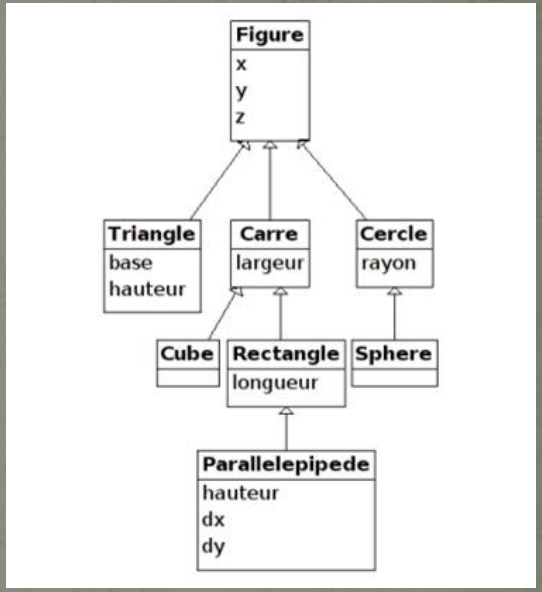
EXERCICES D'APPLICATIONS

- Application 36:

On désire réaliser un programme orienté objet en C++ qui fournit une hiérarchie de classe destinées à mémoriser ou manipuler les propriétés de différentes figures géométriques.
On propose de traiter au moins le cas des rectangles, cercles, triangles, carrés, sphères, parallélépipèdes rectangles, cubes, ... mais cette liste n'est pas limitative.
Chaque classe devra permettre de mémoriser les données qui permettent de définir une instance, par exemple la longueur des deux côtés d'un rectangle, le rayon de la sphère, etc. Chaque instance devra disposer quand cela a un sens :

- d'une méthode double perimetre() qui renvoie la valeur du périmètre de l'instance sur laquelle on l'appelle ;
- d'une méthode double aire() qui renvoie l'aire de la surface de l'objet concerné ;
- d'une méthode double volume() qui renvoie le volume de la forme concernée.

On considérera que lorsque ces trois notions ne sont pas définies mathématiquement, alors la valeur renvoyée sera nulle. Par exemple, le volume d'un carré sera nul, et le périmètre d'une sphère également.
Proposer un programme de test (les données étant fournies par l'utilisateur).



À suivre …

Feedback:

pape.abdoulaye.barro@gmail.com