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 le polymorphisme
- La gestion des exceptions
- Généralités sur la STL

Héritage multiple

Il est possible de faire dériver une classe de plusieurs autres classes simultanément (**héritage multiple**). Exemple: une classe C peut hériter de la classe A et de la classe B. Une instance de la classe C possèdera alors à la fois les données et fonctions-membres de la classe A et celles de la classe B.

Exemple:

```
class A {
.....
};
```

```
class B {
.....
};
```

```
class C : public A, public B{
.....
};
```

- À la création de C, les constructeurs des classes parentes sont appelés: celui de A, puis celui de B.
- À la destruction, les destructeurs des classes parentes sont appelés, celui de B, puis celui de A.
- Il peut arriver que des données ou fonctions-membres des classes A et B aient le même nom. Dans ce cas, il faut utiliser l'opérateur de porté comme suit: A::var ou B::var pour faire la distinction;

Héritage multiple

En guise d'illustration, nous allons recréer la classe toubab, dérivant de la classe personne et de la classe couleur.

- Soit la classe personne et la classe couleur:

```
class personne {
    public:
        personne(){};
        ~personne(){cout<<"End personne!"<<endl;}
        personne(string p, string n){p_prenom = p; p_nom = n; }
        p_personne(string p, string n){p_prenom = p; p_nom = n; }
        void toString(){
            cout<<"Prenom"<<p_prenom<<"Nom"<<p_nom<<endl;
         }

    private:
        string p_nom; string p_prenom;
};
```

```
class couleur {
    public:
        c_couleur(string c){ clr = c; }
        ~couleur(){cout<<"End couleur!"<<endl;}
        void toString(){
            cout<<"teint"<<clr<<endl;
         }

    private:
        string clr;
};
```

- La classe toubab sera alors déclarer comme suit:

```
class toubab : public | private | protected personne, public | private | protected couleur
{
    // traitement
};
```

Héritage multiple

- Dans toubab, nous avons décidé de redéfinir la fonction toString() en se basant sur les fonctions toString() se trouvant successivement dans personne et dans couleur. ***Rappelons que Lorsque plusieurs fonctions membres portent le même nom dans différentes classes, on peut lever l'ambiguïté en employant l'opérateur de résolution de portée***. Ainsi, nous avons:

```
void toubab::toString (){
    personne::toString();
    couleur::toString();
};
```

- Classiquement, un objet de type **toubab** pourra faire appel aux fonctions membres des classes de base **personne** et **couleur** (on pourra se servir de l'opérateur de portée pour lever les éventuelles ambiguïtés).
  - Si on a: toubab p("Natacha", "Lacroix", "Blanc");
    - Alors on pourra faire appel à la fonction toString() se situant dans toubab en faisant p.toString(); à celle se situant dans personne par p.personne::toString(); et puis à celle se situant dans couleur par p.couleur::toString().

Héritage multiple

- La suite de exemple:

```
#include <iostream>
using namespace std ;
                ...
class toubab : public personne, public couleur
{
    public :
        toubab(string p, string n, string c){
                personne::ppersonne(p, n);
                couleur::ccouleur(c);
         }
         ~toubab(){ cout<< "End toubab"<<endl; }
         void toString ();
};

void toubab::toString (){
            personne::toString();
            couleur::toString();
    };
```

```
int main()
{
    toubab p("Natacha", "Lacroix", "Blanc");
    cout<< "toString dans toubab"<<endl;
    p.toString();
    cout<< "toString dans personne"<<endl;
    p.personne::toString();
    cout<< "toString dans couleur"<<endl;
    p.couleur::toString()
}
```

Héritage multiple

Problème de conflits

- Dans le cas d'un héritage multiple, le problème de doublon peut facilement se poser.
  - Supposons que nous avons une classe D qui hérite à la fois de la classe B et de la classe C et que ces derniers héritent toutes les deux de la classe A. leurs déclarations donnent la configuration suivante:

```
class A
{
    .....
} ;
class B : public A {.....} ;
class C : public A {.....} ;
class D : public B, public C
{
    .....
} ;
```

  - Il y'aura une redondance des membres données de A dans tous les objets de type D;
  - Mais maintenant, si l'on souhaite que cela arrive, on pourra toujours faire la distinction en utilisant l'opérateur de portée .

Héritage multiple

Problème de conflits

- Il y'a toujours un mécanisme permettant de travailler avec un seul motif de A dans la classe de D: il suffit de déclarer dans les classes de B et de C que la classe A est virtuelle avec le mot-clé *virtual*.

```
class B : public virtual  A {.....} ;
class C : public virtual A {.....} ;
```

- Déclarer la classe A comme virtuelle dans B et C n'a pas d'effet sur elles mais sur leur descendance (ici, D);

```
class D : public B, public C {.....} ;
```

- Si A est déclarée comme virtuelle dans B, alors A sera introduite qu'une seule fois dans les descendances de C.

Héritage multiple

Appels des constructeurs et des destructeurs

- Lorsque **A** a été déclarée virtuelle dans **B** et **C**, le choix des informations à fournir au constructeur de **A** a lieu dans **D** et non dans **B** ou **C**. On spécifie dans le constructeur de **D**, les informations destinées à **A**.

- Exemple:

D(string p, string n, string c) : B(string p, string n, string c), A(string p, string n)

- **A** doit absolument disposer d'un constructeur par défaut:
- Lorsqu'on crée un objet de type D, le constructeur de **A** est appeler en premier, puis celui de **B** ensuite celui de **C** et en fin celui de **D**.

Héritage multiple

Appels des constructeurs et des destructeurs - exemple

- Nous définissons d'abord trois classe: la classe personne, la classe couleur et la classe ethnie.

```
class personne {
                public:
                   personne(){};
                   personne(string p, string n){p_prenom = p; p_nom = n; }
                   void ppersonne(string p, string n){p_prenom = p; p_nom = n; }
                   ~personne(){cout<<"End personne!"<<endl;}
                   void toString(){
                      cout<<"Prenom: "<<p_prenom<<"Nom: "<<p_nom<<endl;
                   }

                private:
                   string p_nom;
                    string p_prenom;
};
```

```
class ethnie{
                public:
                    ethnie(){};
                    ~ethnie(){cout<<"End ethnie!"<<endl;}
                    ethnie(string e){e_nom = e; }
                    void p_ethnie(string e){e_nom = e; }
                    void toString(){
                        cout<<"Je suis de l'ethnie "<<e_nom<<endl;
                     }

                private:
                    string e_nom;
};
```

```
class couleur {
    public:
        couleur(){};
        couleur(string c){
                    clr = c;
        }
        void ccouleur(string c){
                    clr = c;
        }
        ~couleur(){
                    cout<<"End couleur!"<<endl;
        }
        void toString(){
                     cout<<" de teint "<<clr<<endl;
        }

    private:
         string clr;
};
```

Héritage multiple

Appels des constructeurs et des destructeurs - exemple

- Ensuite, on définie une classe **toubab** fille de personne (virtuelle) et de couleur et une autre classe **negro** fille de personne (virtuelle) et de ethnie

```
class toubab : public virtual personne, public couleur
{
   public :
      toubab(string p, string n, string c) : couleur(c){}
      ~toubab(){
            cout<< "End toubab"<<endl;
      }
      void toString (){
           personne::toString();
           couleur::toString();
       };
};
```

```
class negro : public virtual personne, public ethnie
{
   public :
      negro(string p, string n, string e) : ethnie(e){}
      ~negro(){
            cout<< "End negro !"<<endl;
       }
      void toString (){
              personne::toString();
              ethnie::toString();
       };
};
```

Héritage multiple

Appels des constructeurs et des destructeurs - exemple

- Et enfin une classe métis fille de toubab et de negro.

```
class metis : public toubab, public negro
{
    public:
        metis(string p, string n, string c, string e) : personne(p, n), toubab(p, n, c), negro(p, n, e){}
        void toString (){
                personne::toString();
                negro::toString();
                cout<<"d'une part, et ";
                toubab::toString();
                cout<<"d'autre part! "<<endl;
        };
};
```

```
iint main(){
    toubab p("Natacha", "Lacroix", "Blanc");
    cout<< "toString dans toubab"<<endl;
    p.toString();

    negro n("Soundiata", "Keita", "Manding");
    cout<< "toString dans Négro"<<endl;
    n.toString();

    metis m ("Natacha", "Keita", "Marron", "Manding");
    cout<< "toString dans metis"<<endl;
    m.toString();
}
```

Héritage multiple

EXERCICES D'APPLICATIONS

- Application 35:

On considère les classes personne et couleur suivants.

```
class personne {
                    public:
                       personne(){};
                       personne(string p, string n){p_prenom = p; p_nom = n; }
                       void ppersonne(string p, string n){p_prenom = p; p_nom = n; }
                       ~personne(){cout<<"End personne!"<<endl;}
                       void toString(){
                          cout<<"Prenom: "<<p_prenom<<"Nom: "<<p_nom<<endl;
                       }

                    private:
                       string p_nom;
                       string p_prenom;
};
```

```
class couleur {
    public:
        couleur(){};
        couleur(string c){
                        clr = c;
        }
        void ccouleur(string c){
                        clr = c;
        }
        ~couleur(){
                        cout<<"End couleur!"<<endl;
        }
        void toString(){
                         cout<<" de teint "<<clr<<endl;
        }

    private:
         string clr;
};
```

- Créer la classe toubab héritant de personne et de couleur. Dans toubab, vous allez redéfinir la fonction toString() en se basant sur les fonctions toString() se trouvant successivement dans personne et dans couleur.
- Dans votre programme principal, créer un objet de type toubab et afficher le contenu toString() des classes parentes.

À suivre …

Feedback:

pape.abdoulaye.barro@gmail.com