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 simple

Le principe est d'utiliser la déclaration d'une classe (appelée classe de base ou classe parente) comme base pour déclarer une seconde classe(appelée classe dérivée). La classe dérivée héritera de tous les membres (données et fonctions) publique de la classe parente.

- **Exemple de déclaration de la classe parente**:

```
class ClasseParente {
        public:
        void fonctionParente(string stringParente);
        protected:
        int valeurParente;
};
```

- **Exemple de déclaration de la classe dérivée de la classe parente**:

```
// héritage public
class ClasseDerivee : public ClasseParente {
        public:
        void fonctionDerivee(string stringDerivee);
        protected:
        int valeurDerivee;
};
```

Un objet de la classe Dérivée possède alors ses propres données et fonctions-membres, plus les données-membres et fonctions-membres héritées de la classe parente.

Héritage simple

Après avoir créer la classe dérivée, il est donc possible de déclarer des objets de type ClasseDerivee de manière usuelle:

ClasseDerivee c, d ;

Chaque objet aura accès :

- aux méthodes publiques de la classe dérivée;
- aux méthodes publiques de la classe de base.

Exemple:

```
#include <iostream>
#include "personne.hpp"
using namespace std ;

class toubab: public personne
{
    string couleur;
    public :
        void teint(string c="blanche") { couleur = c ; }
    } ;
```

```
int main()
{
        toubab t ;
        t.defn("Natacha","Lacroix") ;
        t.teint("Marron") ;
        t.toString () ;

}
```

Héritage simple

Utilisation des membres de la classe de base dans une classe dérivée

L'exemple précédent sur la classe toubab ne nous renseigne pas sur la couleur de peau de la personne après appel de la méthode toString().

Pour remédier à cela, deux méthodes sont possibles:

- La première consiste à écrire une nouvelle fonction membre publique dans toubab, qui est censée afficher à la fois le nom, le prenom ainsi que la couleur de peau.

```
void toubab::affiche()
{
    cout <<"Je m'appel"<<p_prenom<<" "<<p_nom<<endl;
    cout<< "je suis de teint " <<couleur <<endl ;
 }
```

- Mais cette méthode signifierai que affiche() a accès aux membres privés de personne (*ce qui est contraire au principe d'encapsulation*). Par conséquent: "***une méthode d'une classe dérivée n'a pas accès aux membres privés de sa classe de base***".

Héritage simple

Utilisation des membres de la classe de base dans une classe dérivée

- **Par contre**, comme une méthode de la classe dérivée a accès aux membres publics de sa classe de base, la fonction membre affiche() de la classe **toubab** pourra alors faire appel à la fonction membre toString() de la classe **personne**.

```
void toubab::affiche()
{
    toString();
    cout<< "je suis de teint " << couleur <<endl ;
 }
```

- **D'une manière générale**, on aurait pu définir une fonction init(<paramètres>) permettant d'initialiser les trois données membres de toubab:

```
void toubab::init(string prenom, string nom, string clr)
{
        defn(prenom, nom);
        couleur = clr;
}
```

Héritage simple

exemple

Exemple complet:

```
#include <iostream>
#include <string>
#include "personne.hpp"
using namespace std ;

class toubab: public personne
 {
      string couleur;
      public :
                void teint(string c="blanc") { couleur = c ; }
                void affiche();
                void init(string, string, string);
} ;

void toubab::affiche()
{
    toString();
    cout<< "je suis de teint " << couleur <<endl ;
 }

void toubab::init(string prenom, string nom, string clr)
{
          defn(prenom, nom);
          couleur = clr;
}
```

```
int main()
{
        toubab t ;
        t.init("Natacha","Lacroix", "Blanc") ;
        t.affiche();

}
```

Héritage simple

La redéfinition des membres d'une classe dérivée

- Dans la classe dérivée, nous avons définit une méthode affiche() qui fait pratiquement la même chose que la méthode toString() de la classe de base. On aurait pu seulement redéfinir la méthode toString() dans la classe dérivée, mais dans ce cas, on ne pourra plus appeler à l'intérieur de celle-ci, la méthode toString() de la classe parente comme on a l'habitude de le faire: on fera personne::toString() pour véritablement localiser la bonne méthode.

Exemple:

```
#include <iostream>
#include "personne.hpp"
using namespace std ;

class toubab: public personne
 {
      string couleur;
      public :
                  void teint(string c="blanc") { couleur = c ; }
                  void toString();
                  void init(string, string, string);
} ;

void toubab::toString()
{
    personne::toString();
    cout<< "je suis de teint " << couleur <<endl ;
 }
```

```
int main()
{
        toubab t ;
        t.init("Natacha","Lacroix", "Blanche") ;
        t.toString();  //  définit dans toubab
        t.personne::toString()  // définit dans  personne

}
```

Héritage simple

Constructeurs, destructeurs et l'héritage

Quand un objet est créé, si cet objet appartient à une classe dérivée, le constructeur de la classe parente est d'abord appelé. Quand un objet est détruit, si cet objet appartient à une classe dérivée, le destructeur de la classe parente est appelé après.

Voici un exemple:

```
#include<iostream.h>
class GrandPere
{       // données membres
        public:
                GrandPere();
                ~GrandPere();
};
class Pere : public GrandPere
{       // données membres
        public:
                Pere();
                ~Pere();
};
class Fils : public Pere
{       // données membres
        public:
                Fils();
                ~Fils();
};
```

```
void main()
{
        Fils *junior;
        junior = new Fils;
        // appels successifs des constructeurs
        de GrandPere, Pere et Fils
        ........
        delete junior;
        // appels successifs des destructeurs
        de Fils, Pere et GrandPere
}
```

Héritage simple

Constructeurs, destructeurs et l'héritage

- Il est possible d'utiliser un constructeur de la classe de base pour définir un constructeur de la clase dérivée (***mécanisme de transmission d'informations entre constructeurs***):

Exemple:

```
#include<iostream>
class  Rectangle{
        public:
            Rectangle(int lo, int la);
            void toString();
        protected:
            int longueur, largeur;
};
Rectangle::Rectangle(int lo, int la){
        longueur = lo;
        largeur = la;
}
void Rectangle::toString(){
        cout <<"surface= "<<longueur*largeur<<endl;
}
        -------
class Carre : public Rectangle {
        public:
            Carre(int cote);
};
Carre::Carre(int cote) : Rectangle(cote, cote) {
}
```

```
void main()
{
    Carre *monCarre;
    Rectangle *monRectangle;
    monCarre = new Carre(5);
    monCarre->toString();  // affiche 25
    …
    monRectangle = new Rectangle(5, 10);
    monRectangle->toString();   // affiche 50
}
```

Héritage

Le constructeur de recopie et l'héritage

Rappel: Un **constructeur de recopie** est généralement utilisé lorsqu'il s'agit d'initialiser un objet par un autre de même type ou lors de la transmission d'un objet en paramètre ou en retour à une fonction.

En supposant que la classe B dérive de la classe A, 2 situations s'offrent alors à nous:

- Soit B ne définit pas de constructeur de recopie. Dans ce cas, le constructeur de recopie de A sera appelé pour les membres données correspondants et le constructeur de recopie par défaut de B (on prévoira des informations pour le constructeur de A).
  - Exemple: B (B & b) : A (...)

- Soit B définit un constructeur de recopie (en supposant que A aussi à définit un constructeur de recopie). Dans ce cas, on pourra effectuer une conversion implicite de la classe B dans la classe A.
  - Exemple: B (B & b) : A (b){...}

Héritage

Le constructeur de recopie et l'héritage - Exemple

```
// inclure les bibliothèques requises
using namespace std ;

class personne {
    public:
        personne(string p, string n){
            p_prenom = p; p_nom = n;
            cout<<"Prenom"<<p_prenom<<"Nom"<<p_nom<<endl;
         }
        personne(personne & p){
           p_prenom = p.p_prenom; p_nom = p.p_nom;
           cout<<p_prenom<<" "<<p_nom<<endl;
         }
    private:
        string p_nom; string p_prenom;

};
class toubab : public personne {
  string couleur;
  public :
    toubab(string p, string n, string c):personne(p, n){
                couleur = c;
                cout<< "Couleur de peau:" << couleur <<endl ;
      }
    toubab(toubab & t):personne(t){
                couleur = t.couleur;
                cout<< "teint:" << couleur <<endl ;
      }
} ;
```

```
void fct (toubab t){
    cout << "Fin !"<<endl;
}
```

```
int main()
{
        void fct(toubab);
        toubab t("Natacha", "Lacroix", "Blanche");
        fct(t);

}
```

- Prenom Natacha Nom Lacroix
- Couleur de peau: Blanche
- Natacha Lacroix
- teint: Blanche
- Fin !

Héritage simple

EXERCICES D'APPLICATIONS

- Application 33:

On dispose d'un fichier point.h contenant la déclaration suivante de la classe point :

```
#include <iostream>
using namespace std ;
class point{
    float x, y ;
    public :
    point (float abs=0.0, float ord=0.0){
        x = abs ; y = ord ;
    }
    void affiche (){
        cout << "Coordonnées : " << x << " " << y << "\n" ;
    }
    void deplace (float dx, float dy) {
        x = x + dx ; y = y + dy ;
    }
} ;
```

a. Créer une classe pointcol, dérivée de point, comportant :

• un membre donnée supplémentaire cl, de type int, contenant la « couleur » d'un point ;

• les fonctions membre suivantes :

affiche (redéfinie), qui affiche les coordonnées et la couleur d'un objet de type pointcol ;

colore (int couleur), qui permet de définir la couleur d'un objet de type pointcol,

un constructeur permettant de définir la couleur et les coordonnées (on ne le définira pas en ligne).

Héritage simple

EXERCICES D'APPLICATIONS

- Application 34:

Ecrire une classe Rectangle permettant de construire un rectangle dotée de deux données membres longueur et largeur.

1. Créer une fonction membre Perimetre() permettant de calculer le périmètre du rectangle et une autre fonction membre Surface() permettant de calculer la surface du rectangle
2. Créer les getters et setters.
3. Créer une classe fille Parallélépipède héritant de la classe Rectangle et dotée en plus d'une donnée membre hauteur et d'une autre fonction membre Volume() permettant de calculer le volume du Parallélépipède.
4. Dans le programme principal, créer une instance de Rectangle et une instance de Parallélépipède.
    a. Afficher le périmètre du rectangle
    b. Afficher La surface du rectangle
    c. Afficher le volume du parallélépipède

À suivre …

Feedback:

pape.abdoulaye.barro@gmail.com