Programmation orientée objets [C++ & Python]

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 30:

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).

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 casse 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 31:

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.

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();

       Rectangle *c= new Carre();
       c->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!

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

La gestion des exceptions

Généralités

- Quelques fois, même si le programme est correct, il peut y survenir des risques pouvant compromettre son bon exécution.
- En C++, avec sa bibliothèque standard, les erreurs sont fournies par des codes qui constituent des valeurs de retour des différentes fonctions. En examinant ces valeurs de retour en de nombreuses points du programme, il est alors possible de capturer les éventuelles anomalies (exceptions) et puis de procéder à leurs traitements.
- ***En production***, il est utile de séparer le code et la détection/correction des erreurs.

La gestion des exceptions

Généralités

- Une exception est une rupture de séquence qui peut être déclenchée par une instruction ***throw***, comportant une expression quelconque dont le type (**classe** ou non) sert à identifier l'exception en tant que telle.
- Il est recommandable, pour chaque exception donnée, de proposer une classe afin de mieux représenter l'anomalie concernée.
- Exemple d'erreur avec "division par zéro":

Erreur

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

int main() {
        int x,a=5,b=0;
        x = a/b;
        cout << "x= " << x << endl;

        return 0;
}
```

La gestion des exceptions

Lever des exceptions avec « throw »

- En C++, la gestion des exceptions repose essentiellement sur *trois mots-clés* : **try**, **catch** et **throw**. Avec **try** qui définit le bloc de code test, **catch** qui définit le bloc de code à exécuter en cas de détection d'erreur et **throw** qui permet de lever une exception en cas de problème (on peut créer une erreur personnalisée à cet effet).
- **Lever des exceptions avec « throw »**:
  en réutilisant l'exemple d'erreur précèdent, il est possible d'identifier la nature de l'erreur. **La syntaxe est la suivante**: **throw expression**;

```
#include <iostream>
#include <string>

using namespace std;

int test_division(int a, int b)
{
        string x = " erreur: division par zero";
        if (b==0) {
                throw x;
        }
        return a/b;
}
```

```
int main() {
        int x,a=5,b=0;

        x = test_division(a, b);
        cout << "x= " << x << endl;

        return 0;
}
// Si une exception est levée et n'est interceptée nulle part, le programme se termine anormalement.
```

La gestion des exceptions

Intercepter/traiter une exception avec « try/catch »

- **Intercepter/traiter une exception avec « try/catch »** :
  Dans catch, entre parenthèses, il est possible de spécifier le type d'exception à intercepter et puis de personnaliser le message d'erreur.
  - **La syntaxe est la suivante**:

```
try {
    // bloc d'instruction à protégé
} catch( NomDeException e ) {
    // bloc d'instruction pour la gestion de l'exception
}
```

  - Reprenons l'exemple précèdent:

```
#include <iostream>
#include <string>

using namespace std;

int test_division(int a, int b)
{
    string x = "erreur: division par zero";
    if (b==0) {
        throw x;
    }
    return a/b;
}
```

```
int main() {
    int x,a=5,b=0;
    try{
        x = test_division(a, b);
        cout << "x= " << x << endl;
    }catch(string s){
        cout << s << endl;
    }
    return 0;
}
```

La gestion des exceptions

Identification et interception d'une exception via une classe

- **Identification et interception d'une exception via une classe** :

  Il est possible de créer une classe pour caractériser un exception.

  - En utilisant le mot clé **throw** puis l'objet correspondant à l'exception, il est possible d'identifier le type d'erreur!: (throw objet;)
  - Puis de l'intercepter (**try**) et de le traiter (**catch**).

```
try{
    // bloc d'instruction à protégé
}catch (classe &o){
    // Traitement de l'exception associée à la classe
}
```

  - Exemple de classe d'exception:

```
class erreur {
        public:
            string cause;
            erreur(string s) : cause(s) { }
            // Le constructeur de copie (nécessaire pour le catch):
            erreur(const erreur &e) : cause(e.cause) { }
};
```

La gestion des exceptions

Identification et interception d'une exception via une classe

- **Utilisation dans un main**

```
#include <iostream>
#include <string>
using namespace std;
// class erreur

int test_division(int a, int b)
{
        erreur x ("division par zero !");
        if (b==0) {
                throw x;
        }
        return a/b;
}

int main(){
    int a,b
    cout<<"a=";
    cin>>a;
    cout<<"b=";
    cin>>b;
    // ...
    // suite

    try{
        cout << "resultat= " << test_division(a, b) << endl;
    }
    catch (erreur &e)
    {
        // récupération de la cause.
        cout << "Erreur, cause: " << e.cause << endl;
    }
    catch (...)
    {
        // le gestionnaire d'exception universel
        // pour toutes autres erreurs
        cout << "Erreur inattendue !" << endl;
    }

    return 0;
}
```

La gestion des exceptions
standard

- C++ nous fournit un certains nombres d'exceptions standard qui sont définies dans la classe ***exception*** qui est la classe de base de toutes les exceptions lancées par la bibliothèque standard.
- Elle est définit comme suite:

```
class exception
{
        public:
            //Constructeur.
            exception() throw(){ }
            //Destructeur.
            virtual ~exception() throw();
            // what envoie une chaîne contenant des infos sur l'erreur.
            virtual const char* what() const throw();
};
```

- **throw** indique que ces méthodes ne vont pas lancer d'exceptions

La gestion des exceptions
standard

- Il devient possible de créer sa propre classe d'exception en héritant de la classe **exception**.
- Réécrivons la classe erreur:

```
class erreur: public exception
{
        private:
            int num;
            string msg;

        public:
            erreur(int n=0, string const& m=" ") throw() : num(n),msg(m) { }
            virtual ~erreur() throw() { }
            virtual const char* what() const throw() {
                    return msg.c_str();
            }
};
```

La gestion des exceptions

standard

- **Utilisation dans un main**

```
#include <iostream>
#include <string>
#include <exception>
using namespace std;
// class erreur

int test_division(int a, int b)
{
        if (b==0) {
                throw erreur(1,"division par zero !");
        }
        return a/b;
}

int main(){
    int a,b
    cout<<"a=";
    cin>>a;
    cout<<"b=";
    cin>>b;
    // ...
    // suite

    try{
        cout << "resultat= " << test_division(a, b) << endl;
    }
    catch (exception const& e)
    {
        // récupération de la cause.
        cout << "Erreur, cause: " << e. what()<< endl;
    }
    catch (...)
    {
         cout << "Erreur inattendue !" << endl;
    }

    return 0;
}
```

La gestion des exceptions

standard

- La bibliothèque standard est capable de lancer 5 types d'exceptions que sont:
  - **std::bad_alloc** – erreur lancée par new (en mémoire).
  - **std::bad_cast** - erreur lancée lors d'un dynamic_cast
  - **std::bad_typeid** – erreur lancée lors d'un typeid
  - **std::bad_exception** – erreur lancée si aucun catch ne correspond à un objet lancé
  - **ios_base::failure** – erreur lancée en manipulant un flux
- Si on ne souhaite pas créer une classe pour gérer les exceptions, on pourra faire appel au fichier standard ***stdexcept*** contenant des classes d'exceptions pour les erreurs les plus courants.
- Il contient exactement 9 classes subdivisées en 2 catégories: les **logic errors** et les **runtime errors.**
- les **logic errors** sont: domain_error, invalid_argument, length_error, out_of_range et logic_error (toutes autres erreurs logiques);
- les **runtime errors** sont: range_error, overflow_error, underflow_error, runtime_error (toutes autres erreurs d'exécution).

La gestion des exceptions

exemple avec *domain_error*

- **exemple avec *domain_error***

```
#include <iostream>
#include <string>
#include <exception>
#include <stdexcept>
using namespace std;
int test_division(int a,int b) {
   if(b==0)
      throw domain_error("division par zero !");
   else
      return a/b;
}
int main(){
   int a=4, b=0;
    try{
        cout << "resultat= " << test_division(a, b) << endl;
    } catch (exception const& e) {
        // récupération de la cause.
        cout << "Erreur, cause: " << e. what()<< endl;
        throw; //relance de l'exeption reçue pour la traiter une deuxiéme fois, plus loin dans le code.
     }
    return 0;
}
```

FIN

pape.abdoulaye.barro@gmail.com