Programmation avancée en C++: Héritage multiple et polymorphisme I

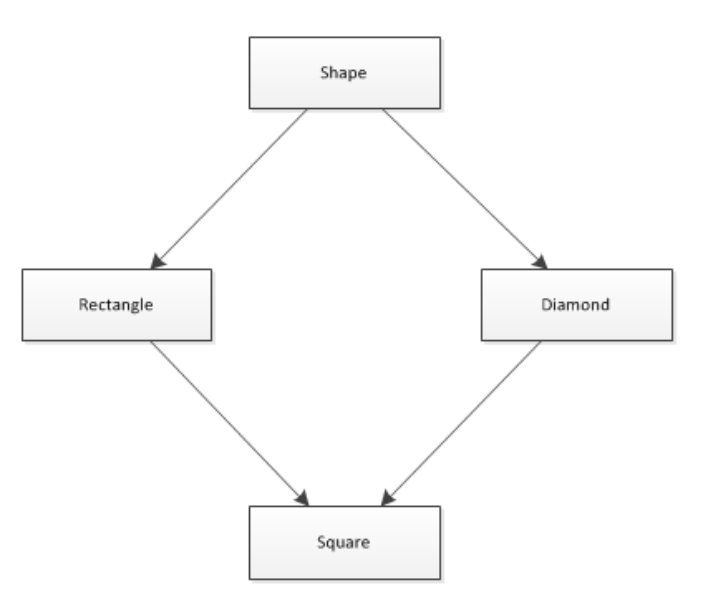
Emmanuel Franck

Bureau 225, UFR math info
mail: emmanuel.franck@inria.fr

1. Héritage multiple: cas simple et difficulté

Héritage multiple

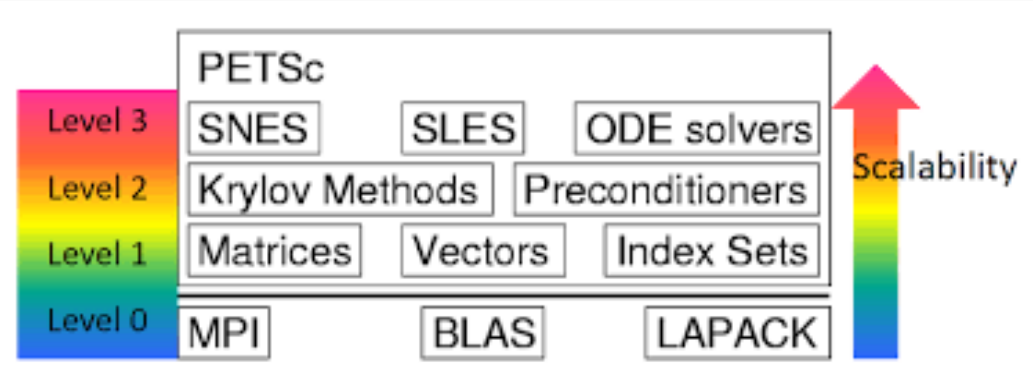
- Une classe peut hériter de plusieurs classes.
- Exemple :



- **Difficultés**: ordre et transmissions des appels des constructeurs et destructeurs, comment les conflits sont réglés etc.
- L'héritage multiple est compliqué. **A utiliser avec modération**.

Exemple simple I

- L'héritage multiple assez présent en algèbre linéaire.



- On définit une classe matrice 2x2, une classe vecteur de taille 2.
- On propose une classe dérivée des deux: une classe produit matrice-vecteur.

Exemple simple II

- **Classes**: mat2 et vec2.

```
1  class mat2 {
2  protected:
3   double mat[2][2];
4  public:
5    mat2 (){cout<< " constructeur par défaut mat2"<<endl;}
6    mat2 (double a11, double a12, double a21, double a22){
7      mat[0][0]=a11;    mat[0][1]=a12;
8      mat[1][0]=a21;    mat[1][1]=a22;
9      cout<< " constructeur  mat2"<<endl;
10   }
11   mat2 (const mat2 & x){
12     mat[0][0]=x.mat[0][0];      mat[0][1]=x.mat[0][1];
13     mat[1][0]=x.mat[1][0];   mat[1][1]=x.mat[1][1];
14     cout<< " constructeur par copie mat2"<<endl;
15   }
16   ~mat2 (){cout<< " destructeur mat2"<<endl;}
17   void multiplie(double);
18   friend ostream & operator << (ostream &,const mat2 &);
19 };
```

- On prend comme exemple de méthode, une méthode qui multiplie le vecteur et la matrice par un scalaire.

Exemple simple II

- **Classes**: mat2 et vec2.

```
1  class vec2 {
2  protected:
3   double vec[2];
4  public:
5    vec2 (){cout<< " constructeur par défaut vec2"<<endl;};
6    vec2 (double a1, double a2) {
7      vec[0]=a1;
8      vec[1]=a2;
9      cout<< " constructeur vec2"<<endl;
10   }
11   vec2 (const vec2 & x){
12     vec[0]=x.vec[0];
13     vec[1]=x.vec[1];
14     cout<< " constructeur par copie vec2"<<endl;
15   }
16   ~vec2 (){cout<<" destructeur vec2"<<endl;}
17   void multiplie(double);
18   friend ostream & operator << (ostream &,const vec2 &);
19 };
```

- On prend comme exemple de méthode, une méthode qui multiplie le vecteur et la matrice par un scalaire.

Exemple simple II

- **Classes**: mat2 et vec2.

```
1  class matvec2 : public mat2, public vec2 {
2  public:
3    matvec2 (){};
4    matvec2 (double a11, double a12, double a21, double a22, double b1, double b2)
5      cout<< " constructeur matvec2"<<endl;
6    }
7    matvec2 (const matvec2 & x) : mat2(x), vec2(x){
8      cout<< " constructeur par copie vec2"<<endl;
9    }
10   ~matvec2 (){cout<<" destructeur matvec2"<<endl;};
11
12   void multiplie(double);
13   friend ostream & operator << (ostream &,const matvec2 &);
14 };
```

- L'ordre dans lequel on écrit l'héritage est important. Ici d'abord mat2 puis vec2.

Exemple simple III

- Utilisation:

```
1  void mat2::multiplie(double x){
2    mat[0][0]=x*mat[0][0]; mat[0][1]=x*mat[0][1];
3    mat[1][0]=x*mat[1][0]; mat[1][1]=x*mat[1][1];
4    cout<<" multiplie mat2"<<endl;
5  }
6  void vec2::multiplie(double x){
7    vec[0]=x*vec[0];
8    vec[1]=x*vec[1];
9    cout<<" multiplie vec2"<<endl;
10 }
11 void matvec2::multiplie(double x){
12   mat2::multiplie(x);
13   vec2::multiplie(x);
14   cout<<" multiplie matvec2"<<endl;
15 }
```

- Si les membres et méthodes des deux classes ”mère” ont pas le même nom, on peut les appeler/utiliser directement.
- Si ils/elles ont le même nom on utilise *mere* :: *membre*.

Exemple simple IV

- Utilisation:

```
1 int main (){
2   matvec2 P;
3   matvec2 Q(1.0,0.0,0.0,1.0,2.0,2.0);
4   .....
5 };
```

```
user $ constructeur par défaut mat2
user $ constructeur par défaut vec2
user $ constructeur mat2
user $ constructeur vec2
user $ constructeur matvec2
```

Exemple simple IV

- Utilisation:

```
1  int main (){
2    .....
3    cout<<"multiplie le produit A b "<<endl;
4    Q.multiplie(5.0);
5    cout<<"multiplie la matrice A "<<endl;
6    Q.mat2::multiplie(1.2);
7    cout<<"multiplie le vecteur b "<<endl;
8    Q.vec2::multiplie(1.4);
9   . ....
10 };
```

```
user $ multiplie le produit A b
user $ multiplie mat2
user $ multiplie vec2
user $ multiplie matvec2
user $ multiplie la matrice A
user $ multiplie mat2
user $ multiplie le vecteur b
user $ multiplie vec2
```

Exemple simple IV

- Utilisation:

```
1 int main (){
2   ......
3
4   return 0;
5 };
```

```
user $ destructeur matvec2
user $ destructeur vec2
user $ destructeur mat2
user $ destructeur matvec2
user $ destructeur vec2
user $ destructeur mat2
```

- Les constructeurs sont appelés dans l'ordre ou on a donné l'héritage. Les destructeurs sont appelés dans l'ordre inverse.

Exemple: héritage en diamant I

- Matrice Pleine: stockage n^2 coefficients, inversion compliquée,
- Matrice symétrique: stockage $\approx n^2/2$ coefficients, inversion un peu plus simple,
- Matrice triangulaire: stockage $\approx n^2/2$ coefficients, inversion simple,
- Matrice diagonale: stockage n coefficients, inversion triviale.

- **Idée**: écrire une classe mère "pleine" qui contient la plupart des fonctions. Les autres dérivent du premier et re-implémente le nécessaire. On optimisera le calcul pas le stockage.

Héritage en diamant

La classe "symétrique" et la classe "triangulaire" hérite de la classe "matrice". La classe "diagonale" hérite de la classe "symétrique" et de la classe "triangulaire".

- On proposer un exemple très simple dans le cas 2*2.

Exemple: héritage en diamant II

- Les 3 premières classes (toutes amies de ”vec2”):

```
1  class mat2 {
2  protected:
3   double mat[2][2];
4  public:
5    mat2 (){cout<< " constructeur par défaut mat2"<<endl;}
6    .......
7    vec2 produitAb(vec2);
8    friend ostream & operator << (ostream &,const mat2 &);
9  };
10 class ts2: public mat2 {
11 public:
12   ts2 (){cout<< " constructeur par défaut ts2"<<endl;}
13   ......
14   vec2 produitAb(vec2);
15   friend ostream & operator << (ostream &,const ts2 &);
16 };
17 class sym2: public mat2 {
18 public:
19   sym2 (){cout<< " constructeur par défaut sym2"<<endl;}
20   ......
21   vec2 produitAb(vec2);
22   friend ostream & operator << (ostream &,const ti2 &);
23 };
```

Exemple: héritage en diamant II

- Exemple d'intérêt (limité dans ce cas):

```
1  vec2 ts2::produitAb(vec2 b){
2    vec2 x;
3
4    x.vec[0] = mat[0][0]*b.vec[0] + mat[0][1]*b.vec[1];
5    x.vec[1] = mat[1][1]*b.vec[1];
6    cout<< " produit ts2"<<endl;
7    return x;
8  }
9  vec2 sym2::produitAb(vec2 b){
10   vec2 x;
11
12   x.vec[0] = mat[0][0]*b.vec[0] + mat[0][1]*b.vec[1];
13   x.vec[1] = mat[1][0]*b.vec[0] + mat[1][1]*b.vec[1];
14   cout<< " produit sym2"<<endl;
15   return x;
16 }
```

- On peut optimiser l'implémentation pour chaque sous classe. En plus grande dimension cela peut être intéressant ou important.

Exemple: héritage en diamant III

- Maintenant on ajoute la classe ”diagonale”:

```
1  class d2: public ts2, public sym2 {
2  public:
3    d2 (){cout<< " constructeur par défaut d2"<<endl;}
4    d2 (double a11, double a22) : mat2(a11,0.0,0.0,a22) {
5      cout<< " constructeur d2"<<endl;
6    }
7    d2 (const d2 & x): mat2(x){
8      cout<< " constructeur par copie d2"<<endl;
9    }
10   ~d2 (){cout<< " destructeur d2"<<endl;}
11
12   vec2 produitAb(vec2);
13   friend ostream & operator << (ostream &,const d2 &);
14 };
```

```
user $ ./mat2_2.hpp:90:6: error: non-static member 'mat' found in multiple
     base-class subobjects of type 'mat2':
user $   class d2 -> class ts2 -> class mat2
user $   class d2 -> class sym2 -> class mat2
```

Exemple: héritage en diamant IV

Problème

- La classe ”ts2” hérite de la matrice ”mat[2][2]”, idem pour la classe ”sym2”.
- La classe ”d2” hérite de cette même matrice par la classe ”ts2” et une deuxième fois par la classe ”sym2”.
- La classe ”d2” a donc deux matrices ”mat[2][2]”.

- Cela peut générer des conflits donc le C++ l’interdit.
- Solution: déclarer les classes ”sym2” et ”ts2” comme héritant d’une classe virtuelle (ici ”mat2”).
- Cette déclaration ne change rien aux classes ”sym2” et ”ts2”. Cela dit que A sera inclue qu’une fois dans les descendants des deux classes.

Remarque importante

Puisque ”mat2” est virtuelle, les constructeurs de ”d2” peuvent avoir accès aux constructeurs de ”mat2” on n’est pas obligé de passer par ceux de ”sym2” et ”ts2”.

Exemple: héritage en diamant V

- Implémentation:

```
1  class mat2 {
2  protected:
3   double mat[2][2];
4  public:
5  ...
6  }
7  class sym2: public virtual mat2 {
8  public:
9  ...
10 }
11 class ts2: public virtual mat2 {
12 public:
13 ...
14 }
15 class d2: public ts2, public sym2 {
16 public:
17 ...
18 }
```

- On voit que la classe ”mat2” est virtuelle pour ”sym2” et ”ts2”.

Exemple: héritage en diamant VI

- Ordre d'appel

```
1 cout<<" matrice A"<<endl;
2   d2 A;
```

```
user $ matrice A
user $ constructeur par défaut mat2
user $ constructeur par défaut ts2
user $ constructeur par défaut sym2
user $ constructeur par défaut d2
user $ destructeur d2
user $ destructeur sym2
user $ destructeur ts2
user $ destructeur mat2
```

- Le constructeur de la classe virtuelle est toujours appelé avant les autres. Ici, on commence donc logiquement par "mat2". Cependant si "mat2" hérite d'une autre classe, son constructeur sera quand même appelé avant.

Exemple: héritage en diamant VIII

- Constructeur de la classe ”d2”

```
1 d2 (double a11, double a22) : mat2(a11,0.0,0.0,a22) {
2      cout<< " constructeur d2"<<endl;   }
```

```
user $ matrice B
user $ constructeur mat2
user $ constructeur par défaut ts2
user $ constructeur par défaut sym2
user $ constructeur d2
user $ 1 0 0 2
user $ destructeur d2
user $ destructeur sym2
user $ destructeur ts2
user $ destructeur mat2
```

- Cela marche bien. On voit que les constructeurs de ”ts2” et ”sym2” sont automatiquement appelés.

Exemple: héritage en diamant VIII

- Constructeur de la classe ”d2”

```
1 d2 (double a11, double a22) : sym2(a11,0.0,a22) {
2      cout<< " constructeur d2"<<endl; }
```

```
user $ constructeur par défaut mat2
user $ constructeur par défaut ts2
user $ constructeur  sym2
user $ constructeur d2
user $ 2.122e-314 2.24398e-314 0 0
user $ destructeur d2
user $ destructeur sym2
user $ destructeur ts2
user $ destructeur mat2
```

- Cela ne marche pas. Il pourrait avoir des ambiguïtés ça, on pourrait utiliser un constructeur de ”ts2” et un de ”sym2” qui appelle celui de ”mat2”. Il pourrait donc avoir plusieurs constructions. Pour éviter cela, le compilateur ignore les appels a partir de la classe ”d2” des constructeurs ”sym2” et ”ts2”.

Conclusion

- L'héritage multiple est solution intéressante proposée par le C++.
- Cependant cela peut être facilement compliqué à manier si la structure de l'héritage est complexe (exemple: structure en diamant).
- A utiliser avec parcimonie et bien réfléchir avant à la structure globale.

2. Polymorphisme, exemple simple

Groupe abstrait

- Le groupe $(E, +)$ est définit par les propriétés suivantes
  - Associativité: $x + (y + z) = (x + y) + z, \quad \forall x, y, z \in E$
  - Elément neutre "e": $x + e = x, \quad \forall x \in E$
  - Opposé "$x^{'}$": $x + x^{'} = e, \quad \forall x \in E$
- Ce sont des règles abstraite vérifiées par tous les groupes.
- **Application de l'héritage**: une classe groupe (sans donnée) contenant : "addition", "élément neutre" et "opposé" ainsi que la vérification que ces éléments forment un groupe. Puis une classe héritée pour un exemple de groupe. La vérification de la structure de groupe est faite par la classe mère.
- **Avantage**: La vérification de groupe est écrite une fois pour toutes, et toutes les classes héritées définies par l'utilisateur en hérite.
- Idem pour les structures algébriques.

Exemple

- On définit le groupe:

```
1  class groupea {
2  public:
3    groupea (){};
4    groupea (const groupea & x){};
5    ~groupea (){};
6  void oppose(groupea){ cout<< "oppose groupe "<<endl;};
7  void addition(groupea, groupea ){ cout<<" addition groupe "<<endl;};
8  void zero(){cout <<" zero groupe"<<endl;};
9  void groupea::verifie_groupe(groupea a , groupea b , groupea c){
10   //verification groupe //
11   a.zero(); b.zero();}
12 };
```

- On écrit pas la vraie fonction de vérification pour le moment, juste une fonction qui appelle une autre pour l'exemple.
- Les fonctions sont vides, elles seront redéfinies par le classe héritée qui contiendra des données.

Exemple II

- Exemple: Matrice de taille 2.

```
1  class mat2 : public groupea {
2  protected:
3   double mat[2][2];
4  public:
5    mat2 (){};
6    mat2 (double a11, double a12, double a21, double a22){
7      mat[0][0]=a11;   mat[0][1]=a12;
8      mat[1][0]=a21;   mat[1][1]=a22; }
9    mat2 (const mat2 & x){
10     mat[0][0]=x.mat[0][0];    mat[0][1]=x.mat[0][1];
11     mat[1][0]=x.mat[1][0];    mat[1][1]=x.mat[1][1];  }
12   ~mat2 (){};
13
14 mat2 operator +(mat2);
15 mat2 & operator =(const mat2 &);
16 void oppose(mat2);
17 void addition(groupea * , groupea* );
18 void zero();
19 friend ostream & operator << (ostream &,const mat2 &);
20 };
```

Exemple III

- Fin de l'exemple: Matrice de taille 2.
- On définit l'addition des matrices, l'opposée d'une matrice et la matrice nulle.
- On définit le + avec l'aide de l'addition.

```
1  mat2 m1(1.0,0.0,2.0,1.0);
2  mat2 m2(1.0,0.0,2.0,1.0);
3  mat2 m3;  m3 = m1 + m2;
4  m1.zero();
5  m1.verifie_groupe(m1,m2,m3);
```

```
>>>>> appel methode
zero du mat2
zero groupe
zero groupe
```

- On veut appeler la fonction "verifie_groupe" valable pour tout groupe additif.
- Marche pas: une fois dans la fonction "verifie_groupe" (classe mère), il a considéré les objets "mat2" en "groupea" (signature de "verifie_groupe") et ne peut plus appeler la fonction "zero()" de "mat2".

Exemple IV

- Fin de l'exemple : Matrice de taille 2.
- On définit l'addition des matrices, l'opposée d'une matrice et la matrice nulle.
- On définit le + avec l'aide de l'addition.

```
1  groupea g1,g2,g3;
2    groupea * pg1, * pg2, *pg3;
3    mat2 * pm1, * pm2, * pm3;
4
5    mat2 m1(1.0,0.0,2.0,1.0);
6    mat2 m2(1.0,0.0,2.0,1.0);
7    mat2 m3;  m3 = m1 + m2;
8    pm1 = &m1;  pg1 = pm1;
9    pm2 = &m2;   pg2 = pm2;
10   pm3 = &m3;  pg3 = pm3;
11 cout<< " >>>>> appel methode "<<endl;
12   pm1->zero();
13   pg1->zero();
```

```
>>>>> appel methode
zero du mat2
zero groupe
```

Exemple V

- l'idée c'était de
  - Construire une classe groupe avec les 3 fonctions et une vérification à l'aide de ces fonctions.
  - Les classes filles décrivent les données et ces fonctions.
  - On utilise un pointeur (classe mère) qui pointe sur un objet fille et qui nous permet d'appeler la fonction de vérification (mère) sur l'objet (fille).
- Si on fait cela, le pointeur (de type mère) appellera les fonctions de la classe mère même s'il pointe sur un objet fille.
- Exemple: la fonction "zéro()".

Important

un pointeur sur un objet peut contenir un objet d'une classe fille, mais lors d'un appel de méthode, il appelle la méthode liée au type du pointeur (classe mère) et non au type réel de l'objet.
Typage statique: le type de l'objet est défini à la compilation, il considère donc que le type de l'objet est celui du pointeur.

- **Solution** : polymorphisme.

Fonction virtuelle I

But

Typage dynamique: le compilateur ne décide pas l'avance du type. Avec "pg1– >zero()", il décide pas quel "zero()" appeler à la compilation, il le fera lors de l'exécution en fonction du type de l'objet pointé.

- Pour faire du typage dynamique, on définit une fonction dite virtuelle. L'implémentation de la fonction utilisée sera choisie à l'exécution.

```
1 class groupea {
2 public:
3   groupea (){};
4   groupea (const groupea & x){};
5   ~groupea (){};
6 virtual void oppose(groupea){ cout<< "oppose groupe "<<endl;};
7 virtual void addition(groupea, groupea ){ cout<<" addition groupe "<<endl;};
8 virtual void zero(){cout <<" zero groupe"<<endl;};
9 ....
```

- Un seul mot clé "virtual" suffit.

Fonction virtuelle II

- Utilisation:

```
1   groupea * pg1, * pg2, *pg3;
2   mat2 * pm1, * pm2, * pm3;
3
4   mat2 m1(1.0,0.0,2.0,1.0);
5   mat2 m2(1.0,0.0,2.0,1.0);
6   mat2 m3;  m3.oppose(m1);
7   pm1 = &m1;  pg1 = pm1;  pm2 = &m2;  pg2 = pm2;  pm3 = &m3; pg3 = pm3;
8   cout<< " type pointeur pm1: "<<typeid(pm1).name()\\
9   <<", type dans le pointeur: "<<typeid(*pm1).name()<<endl;
10  cout<< " type pointeur pg1: "<<typeid(pg1).name()\\
11  <<", type dans le pointeur: "<<typeid(*pg1).name()<<endl;
12  cout<< " >>>>> appel methode "<<endl;
13  pm1->zero();
14  pg1->zero();
```

```
user $ type pointeur pm1: P4mat2, type dans le pointeur: 4mat2
user $ type pointeur pg1: P7groupea, type dans le pointeur: 4mat2
user $ >>>>> appel methode
user $ zero du mat2
user $ zero du mat2
```

Fonction virtuelle III

- On voit qu'on appelle le "zero()" de la classe fille quelque soit le type du pointeur (classe mère ou fille) si le pointeur pointe sur un objet de la classe fille. **Objectif atteint**.
- Remarque:
  - La librairie "typeinfo" permet d'utiliser les fonctions pour identifier les types.
  - "typeid(pg1).name()" donne le type de l'objet pg1.
  - Ici un pointeur de type "groupea". En l'appliquant a "*pg1" on a le type de l'objet contenu dans le pointeur.
  - Pour l'utiliser on ajoute au début du fichier:

```
1 #include "groupea.hpp"
2 #include <typeinfo>
3 ...
4 using namespace std;
```

Fonction virtuelle IV

- Utilisation pour d'autres fonctions:

```
1  pm1 = &m1; pg1 = pm1;
2  pm3 = &m3; pg3 = pm3;
3  cout<< " >>>>> oppose 1"<<endl;
4  pm3->oppose(m1);
5  pg3->oppose(m1);
6  cout<< " >>>>> appel methode "<<endl;
7  pm1->zero();
8  pg1->zero();
```

```
user $ >>>>> oppose 1
user $ oppose mat2
user $ oppose groupe
user $ >>>>> appel methode
user $ zero du mat2
user $ zero du mat2
```

- Cela marche pour "zero()", pas pour "oppose()". **Pourquoi ?**

Fonction virtuelle V

- On propose plusieurs définitions pour ”oppose”:

```
1  virtual void oppose(groupea){ cout<< "oppose groupe "<<endl;};
2  virtual void oppose2(groupea &){ cout<< "oppose2 groupe "<<endl;};
3  virtual void oppose3(groupea *){ cout<< "oppose3 groupe "<<endl;};
4  virtual groupea *oppose4(){ groupea * res; cout<< "oppose4 groupe "<<endl;
5   return res;};
6  virtual void oppose5(groupea *){ cout<< "oppose5 groupe "<<endl;};
7  ....
8  void oppose(mat2);
9  void oppose2(mat2 &);
10 void oppose3(mat2 *);
11 mat2 * oppose4();
12 void oppose5(groupea *);
```

- Pour chacune de ses implémentations, on donne une implémentation dans la classe ”mat2”.
- Les trois première prennent l’opposition d’une matrice donnée et le stocke dans l’objet courant.
- La 4eme prend l’opposé de l’objet courant et renvoie son adresse dans un nouveau pointeur.
- La 5eme prend l’opposé de l’objet courant.

Fonction virtuelle VI

- Résultats:

```
user $ >>>>> oppose 1
user $ oppose mat2
user $ oppose groupe
user $ >>>>> oppose 2
user $ oppose2 mat2
user $ oppose2 groupe
user $ >>>>> oppose 3
user $ oppose3 mat2
user $ oppose3 groupe
user $ >>>>> oppose 4
user $ oppose4 mat2
user $ oppose4 mat2
user $ >>>>> oppose 5
user $ oppose5 mat2
user $ oppose5 mat2
user $ >>>>> appel methode
user $ zero du mat2
user $ zero du mat2
```

- Le polymorphisme ne s'applique pas a tout type de fonction.

Polymorphisme: condition

- **Méthodes virtuelles**:
  - Les paramètres des fonctions virtuelles doivent avoir exactement le même type dans les classes mère et fille, sinon le typage sera quand même statique (fonction 1 à 3).
  - Le type de retour doit avoir en général le même type de retour. Une exception si le type de retour est un pointeur sur une classe (fonction 4).
  - **Conditions non respectées**: le typa sera statique et non dynamique.
  - Conclusion: on ne peut pas virtualiser toutes les signatures/fonctions.
- Une fonction classique (non membre d'une classe) peut pas être virtuelle.
- **Constructeurs/destructeur**:
  - Les constructeurs ne peuvent être virtuel. Par définition il faut un type bien défini.
  - Le destructeur peut l'être (on en parlera pas).
  - L'opérateur d'affectation peut être virtuel en théorie, mais cela ne marche pas.

Important

Les fonctions virtuelles sont un objet puissant, mais avec des limites. Il faut donc bien réfléchir si on peut les utiliser dans notre cadre.

Exemple groupe abstrait I

- **Classe groupe** :

```
1  class groupea {
2  public:
3    groupea (){};
4    groupea (const groupea & x){};
5    ~groupea (){};
6  virtual void oppose(groupea *){ cout<< "oppose5 groupe "<<endl;};
7  virtual void addition(groupea *, groupea *){ cout<<" addition groupe "<<endl;};
8  virtual bool equal (groupea *){ cout<<" egalite du groupe "<<endl;};
9  virtual void zero(){cout <<" zero groupe"<<endl;};
10 bool certifie_associativite(groupea *, groupea *, groupea *);
11 bool certifie_neutre(groupea *);
12 bool certifie_oppose(groupea *, groupea *);
13 };
```

- Les fonctions virtuelles qui seront implémentées par les classes filles sont les propriétés du groupe + ”égalité”.
- Les fonctions de vérification du groupe, commune à tous les groupes, sont implémentées dans la classe mère et seront appelées par les objets des classes filles.

Exemple groupe abstrait II

- **Certification du groupe**: faite façon générique

```
1  bool groupea::certifie_associativite(groupea * a, groupea * b, groupea * c){
2     bool resbool=false;
3
4     this->addition(a,b);
5     this->addition(this,c);
6     b->addition(b,c);
7     a->addition(a,b);
8     resbool = this->equal(a);
9     return resbool;
10 }
11 bool groupea::certifie_neutre(groupea * a){
12    bool resbool=false;
13
14    a->zero();
15    this->addition(this,a);
16    resbool = this->equal(this);
17    return resbool;
18 }
```

- **Certification du groupe** : on évite que la fonction soit codée par l'utilisateur. Elle est fournie, il a juste à vérifier.

Exemple groupe abstrait III

- **Classe fille** : matrice 2x2

```
1  class mat2 : public groupea {
2  protected:
3   double mat[2][2];
4  public:
5    mat2 (){};
6    mat2 (double a11, double a12, double a21, double a22){
7    mat[0][0]=a11;   mat[0][1]=a12; mat[1][0]=a21;   mat[1][1]=a22;  }
8    ~mat2 (){};
9  mat2 operator +(mat2);
10 mat2 operator -(mat2);
11 mat2 & operator =(const mat2 &);
12 void oppose(groupea *);
13 void addition(groupea * , groupea* );
14 bool equal (groupea *);
15 void zero();
16 };
```

- Les fonctions ”oppose”, ”egal” et ”addition” prennent des objets ”groupea” en paramètre car si il faut la même signature que dans la classe mère pour avoir le typage dynamique (polymorphisme).

Exemple groupe abstrait IV

- **Membre de mat2** : comment les coder.
- Cas simple:

```
1 void mat2::zero(){
2     mat[0][0] = 0.0;     mat[1][0] = 0.0;
3     mat[0][1] = 0.0;     mat[1][1] = 0.0;
4     cout<<" zero du mat2"<<endl;
5 }
```

- Cas plus compliqué: ”opposé”. Ecriture naive:

```
1 void mat2::oppose(groupea * x){
2     mat[0][0] =-x->mat[0][0];  mat[0][1] =-x->mat[0][1];
3     mat[1][0] =-x->mat[1][0];  mat[1][1] =-x->mat[1][1];
4     cout<< "oppose5 mat2 "<<endl;
5 }
```

- On veut appeler la fonction que si il s’agit d’un objet ”mat2” pointé dans ”x”. Cependant, à la compilation, cela ne peut être su.
- En toute généralité, *x* ne contient pas forcément un objet de type ”mat2” donc ca compilera pas.

Exemple groupe abstrait V: cast dynamique

- **Solution**: cast dynamique de pointeur

```
1 void mat2::oppose(groupea * x){
2 .... }
```

- En général on souhaite appeler cette fonction quand *x* pointe vers un objet de mat2. On peut éviter le cas où cela n'est pas vrai par un test.
- Si "x" pointe vers un objet de type mat2, on peut construire un pointeur de type mat2 qui pointe vers le même objet et l'utiliser.

```
1 void mat2::oppose(groupea * x){
2     mat2 * p = dynamic_cast <mat2 *> (x);
3     mat[0][0] =-p->mat[0][0];
4     mat[0][1] =-p->mat[0][1];
5     mat[1][0] =-p->mat[1][0];
6     mat[1][1] =-p->mat[1][1];
7     cout<< "oppose mat2 "<<endl;
8 }
```

- Idem pour addition, etc. Si "x" contient pas de type "mat2" cela crash a l'exécution. Il faut mieux prevoir un test.

Exemple groupe abstrait VI

- **Certification du groupe**:

```
1  int main (){
2    cout<<" >>> verification groupe <<<"<<endl;
3    groupea * pga1, * pga2, *pga3, *pga4;
4    mat2 * pma1, * pma2, * pma3,* pma4;
5    mat2 ma1(1.0,0.0,2.0,1.0);
6    mat2 ma2(1.0,0.0,0.0,1.0);
7    mat2 ma3(-1.0,6.0,0.0,1.0);
8    mat2 ma4(-3.0,6.0,-2.0,1.0);
9    pma1 = &ma1; pma2 = &ma2;
10   pma3 = &ma3; pma4 = &ma4;
11   pga1 = pma1; pga2 = pma2;
12   pga3 = pma3; pga4 = pma4;
13
14   bool v1; nt c =0;
15   v1=pga4->certifie_associativite(pga1,pga2,pga3);
16   c = c+v1;
17   v1=pga3->certifie_neutre(pga1);
18   c = c+v1;
19   v1=pga3->certifie_oppose(pga1,pga2);
20   c = c+v1;
21   cout<<" tests conformité du groupe :"<<c<<"/3"<<endl;
22   return 0;
23 };
```

Exemple groupe abstrait VII

- **Avantage**: la certification est écrite une fois pour toutes (indépendante du groupe concret de l'utilisateur).
- On donne notre exemple de groupe avec ses opérations etc. La classe groupe permet de vérifier la structure de groupe facilement.
- Une fois les opérations certifiées, on construit le reste avec.

```
1  mat2 mat2::operator + (mat2 x){
2    mat2 res;
3    res.addition(this,&x);
4    return res;
5    cout<<"+ de mat2"<<endl;
6  }
7  mat2 mat2::operator - (mat2 x){
8    mat2 res,op;
9    op.oppose(&x);
10   res.addition(this,&op);
11   return res;
12   cout<<"- de mat2"<<endl;
13 }
```

- Idem pour le "==" et d'autre potentielles fonctions ou opérateurs.