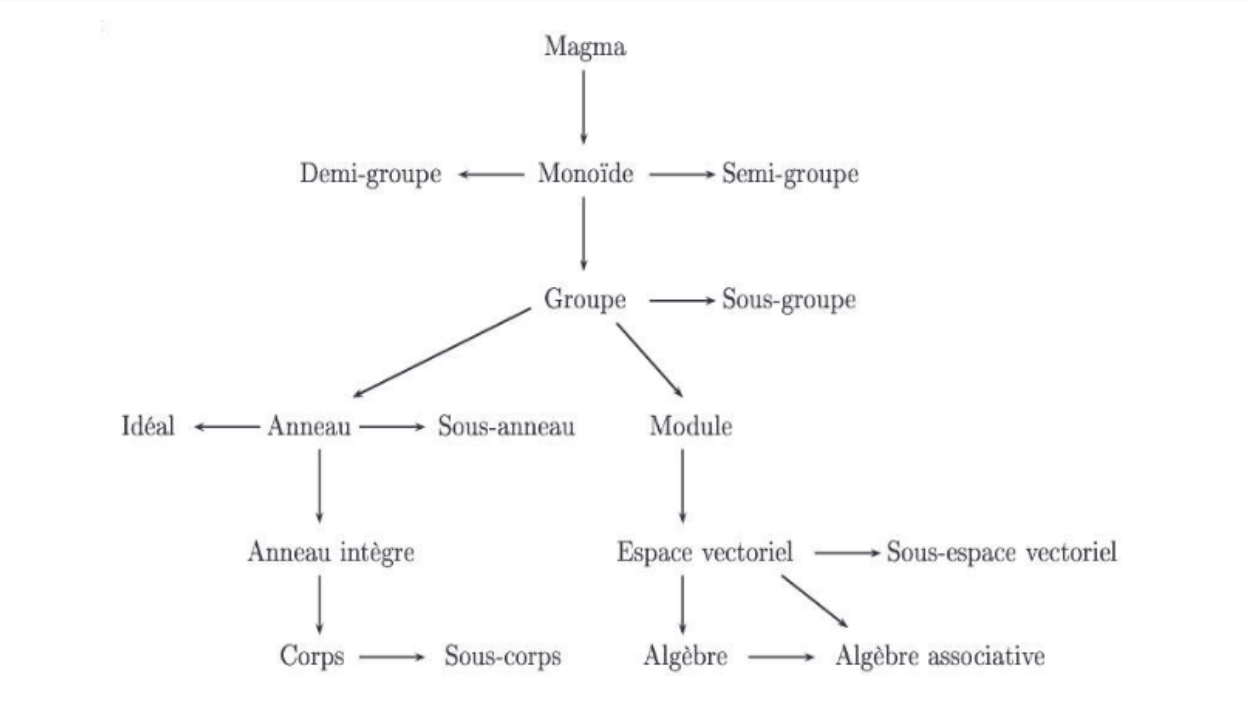
Programmation avancée en C++: Héritage multiple et polymorphisme II

Emmanuel Franck

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

Structure algébrique I



- L'ensemble des structures algébriques correspond à un exemple d'héritage multiple assez complet.

Objectif

On définit une arborescence de structure algébrique à l'aide des classes et fonctions virtuelles. Lorsque un utilisateur définit une classe comme une classe héritée d'une des structures, on peut lancer une vérification sur la classe à l'aide des classes mères.

Structure algébrique II

- Même exemple que précédemment, mais plus compliqué.
- On propose :
  - une classe groupe additif,
  - une classe espace métrique (em),
  - une classe espace vectoriel (ev), héritée de ”groupe additif”,
  - une classe espace vectoriel normé (evn), héritée de ”espace vectoriel” et ”espace métrique”.
- **Principe de vérification** un ”evn” vérifie qu’il est un ”em”, et un ”ev”, un ”ev” vérifie qui est un ”groupe” et les propriétés additionnelles, un ”groupe” vérifie ses propriétés. Vérification récursive.
- L’utilisateur lance juste la vérification de la classe de laquelle son exemple hérite.

Espace métrique abstrait

- L'espace métrique (E, d) est défini par les propriétés suivantes:
  - d application de $E \times E$ dans $\mathbb{R}$.
  - Symétrie: $d(x, y) = d(y, x), \quad \forall x, y \in E$
  - Séparation: $d(x, y) = 0 \rightarrow x = y, \quad \forall x, y \in E$
  - Inégalité triangulaire: $d(x, z) \leq d(x, y) + d(y, z), \quad \forall x, y, z \in E$

```
1  class em {
2  public:
3    em (){};
4    em (const em & x){};
5    ~em (){};
6    virtual double dis(em *,em *) {
7          double res; cout<<" distance em"<<endl;
8          return res};
9    bool certifie(em * e1,em * e2,em * e3);
10 };
```

- Fondamentalement la classe "em" ne contient pas de données, on ne souhaite pas déclarer d'objet avec (mais c'est possible). De plus on définit la fonction "dis" sans que ce soit nécessaire, elle sera définie par les classes filles.

Classe abstraite

- Il existe des notions pour décrire ce type de classe.
  - **Fonction virtuelle pure**: une fonction sans implémentation tant qu'elle est virtuelle.
  - Jusqu'à présent, on donnait une implémentation vide.
  - **Classe abstraite**: une classe dont on ne peut pas déclarer des objets de son type (mais on peut déclarer des pointeurs).
  - **Remarque**: ces notions correspondent à nos classes "em", "groupea" etc.
- **Fonction virtuelle pure**:

```
1   virtual double dis(em *,em *) =0;
```

- **Classe abstraite**: Toute classe contenant au moins une fonction virtuelle pure.
- **Avantages**:
  - Si une classe est abstraite, on ne peut pas déclarer des objets de cette classe.
  - Les fonctions virtuelles pures doivent être absolument implémentées dans les classes héritées.
  - Toute classe héritant de "em" doit implémenter de façon virtuelle ou réelle la

Espace métrique abstrait II

- On obtient donc la nouvelle version de la classe ”em:

```
1 class em {
2 public:
3   virtual double dis(em *,em *)=0;
4   bool certifie(em * e1,em * e2,em * e3);
5 };
```

- Classe abstraite ==> pas de déclaration d'objet possible donc: pas besoin de destructeur/constructeurs.
- Exemple si on déclare un ”em”:

```
user $ main.cpp:14:6: error: variable type 'em' is an abstract class
user $ em h;
user $ ./em.hpp:10: note: unimplemented pure virtual method 'dis' in 'em'
user $ virtual double dis(em *,em *)=0;
```

- La méthode virtuelle pure doit être redéfinie par une classe avec des données. Elle doit être appelée en fonction du type ”réel” de l'objet.
- La méthode classique est définie dans cette classe. Elle doit être appelée en fonction du type du pointeur et non du type réel.

Espace métrique abstrait III

- Méthode de vérification des propriétés de la distance:

```
1  bool em::certifie(em * e1,em * e2,em * e3){
2    bool res=false;
3    int c =0;
4    double dist1,dist2,dist3;
5
6    dist1 = e1->dis(e1,e2);
7    dist2 = e2->dis(e2,e1);
8    if( dist1== dist2){c++;}
9    dist1 = e1->dis(e1,e1);
10   dist2 = e2->dis(e2,e2);
11   if(dist1 +dist2 ==0.0){c++;}
12   dist1 = e1->dis(e1,e3);
13   dist2 = e2->dis(e1,e2);
14   dist3 = e3->dis(e2,e3);
15   if(dist1 <= dist2+dist3){c++;}
16   if(c==3){res = true;}
17   if(res ==false ){cout<<" ce n'est pas un espace métrique "<<endl;}
18   return res;
19 }
```

- A ce moment la il n'existe pas d'implémentation de la distance. Elle sera donc donnée plus tard par une classe héritée.

Groupe abstrait I

- Le groupe $(G, +)$ est définit par les propriétés suivantes:
  - Loi de composition interne: $+$ de $G \times G$ dans G.
  - Associativité: $x + (y + z) = (x + y) + z, \quad \forall x, y, z \in G$
  - Élément neutre: $\exists e \in E, \quad x + e = x, \quad \forall x \in E$
  - Opposé : $\exists x^{-1} \in E, \quad x + x^{-1} = e, \quad \forall x \in E$

```
1 class groupea {
2 public:
3   virtual void oppose(groupea *)=0;
4   virtual void addition(groupea *, groupea *)=0;//
5   virtual bool equal (groupea *)=0;//
6   virtual void zero()=0;//
7
8   bool certifie(groupea *, groupea *, groupea *);
9 };
```

- Contrairement à l'espace métrique où on utilise le $=$ de $\mathbb{R}$, ici, on a besoin de définir l'égalité dans E.

Groupe abstrait II

- Méthode de vérification des propriétés de groupe:

```
1  bool groupea::certifie(groupea * a, groupea * b, groupea * c){
2     bool resbool=false; int comp =0;
3     this->addition(a,b);  this->addition(this,c);
4     b->addition(b,c);  a->addition(a,b);
5     resbool = this->equal(a);
6     comp = comp+resbool;
7
8     resbool=false;
9     this->zero();  this->addition(a,this);
10    resbool = a->equal(this);
11    comp = comp+resbool;
12
13    resbool=false; c->oppose(a);
14    this->zero();  c->addition(a,c);
15    resbool = this->equal(c);
16    comp = comp+resbool;
17
18    resbool = false;
19    if (comp==3) {resbool=true; cout<<" c'est un groupe"<<endl;}
20    if(resbool ==false) {cout<<" ce n'est pas un groupe "<<endl; exit(0);}
21    return resbool;
22 }
```

Espace vectoriel abstrait I

- Le groupe $(E, +, *)$ est défini par les propriétés suivantes:
  - On a que $(E, +)$ est un groupe.
  - Distributivité: $\lambda * (x + y) = \lambda * x + \lambda * y, \quad \forall x, y \in E, \forall \lambda \in \mathbb{K}$
  - Distributivité: $(\lambda + \mu) * x = \lambda * x + \mu * x, \quad \forall x \in E, \forall \lambda, \mu \in \mathbb{K}$
  - Associativité: $(\lambda * \mu) * x = \lambda * (\mu * x), \quad \forall x \in E, \forall \lambda, \mu \in \mathbb{K}$
  - Element neutre : $\exists e_1 \in E, \quad e_1 * x = x, \quad \forall x \in E$

```
1  class ev : public groupea {
2  public:
3    virtual void oppose(groupea *)=0;
4    virtual void addition(groupea *, groupea *)=0;
5    virtual bool equal (groupea *)=0;
6    virtual void zero()=0;
7    virtual void produit_externe(double)=0;
8    virtual double un()=0;
9    bool certifie(ev*, ev*, ev*, double, double);
10 };
```

- Le ”un” est l'élément neutre de la loi de composition externe (multiplication).
- Puisque ”ev” hérite de ”groupea” les 4 premières fonctions doivent être redéclarées virtuelle pure ou avec une implémentation.

Espace vectoriel abstrait II

- On ne détaille pas la vérification d'espace vectoriel. Le principe est le même.
- Espace vectoriel normé (E,+,*,N):
  - on n'a que $(E, +, *)$ est un espace vectoriel,
  - on n'a que (E, d) est un espace métrique avec $d(x, y) = N(x - y)$ et N une norme.
- On a donc de l'héritage multiple.

```
1  class evn : public ev, public em {
2 public:
3   virtual void oppose(groupea *)=0;
4   virtual void addition(groupea *, groupea *)=0;
5   virtual bool equal (groupea *)=0;
6   virtual void zero()=0;
7   virtual void produit_externe(double)=0;
8   virtual double un()=0;
9   virtual double norm(evn*)=0;
10  virtual double dis(em *,em *)=0;
11  bool certifie(evn*, evn*, evn*, double, double);
12 };
```

- Ici la vérification consiste juste à appeler deux autres vérifications.

Espace vectoriel abstrait III

- Vérification:

```
1  bool evn::certifie(evn* e1, evn* e2, evn* e3, double l1, double l2){
2     bool resem=false, resev=false;
3     ev * g1, * g2, * g3, *g4;
4     em * em1, *em2, * em3, * em4;
5
6     em1 = e1;   em2 = e2;
7     em3 = e3;  em4 = this;
8     resem = em4->certifie(em1,em2,em3);
9     g1 = e1;   g2 = e2;
10    g3 = e3;   g4 = this;
11    resev = g4->certifie(g1,g2,g3,l1,l2);
12    if(resem+resev==2) {cout<<" c'est un evn "<<endl; return true;}
13    else {return false;}
14 };
```

- Les fonctions ”vérifie” ne sont pas des fonctions virtuelles (donc typage statique). Le choix de la fonction est fait en fonction du type des objets de paramètre et de l'objet qui appelle la méthode.
- **Ici**: un groupe et un espace métrique.

Espace matrice 2*2 I

- On implémente maintenant une classe on abstrait

```
1  class mat2 : public evn {
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 (const mat2 & x){
9      mat[0][0]=x.mat[0][0];  mat[0][1]=x.mat[0][1];
10     mat[1][0]=x.mat[1][0];  mat[1][1]=x.mat[1][1];  }
11   ~mat2 (){};
12 mat2 operator +(mat2);
13 mat2 operator -(mat2);
14 mat2 & operator =(const mat2 &);
15 void oppose(groupea *);
16 void addition(groupea *, groupea *);
17 bool equal (groupea *);
18 void zero();
19 void produit_externe(double);
20 double un();
21 double norm(evn *);
22 double dis(em *, em *);
23 };
```

Espace matrice 2*2 II

- **Essentiel**: une des 8 dernières fonctions n'est pas fournie, cela ne compile pas.
- **Remarque importante**: on implémente "+", "-", "==" etc en fonctions des "addition", "oppose" etc. En effet, ces fonctions seront certifiées donc construire les opérateurs avec, certifie aussi les opérateurs.

```
1  int main (){
2    cout<<" >>> verification mat2 <<<"<<endl;
3    mat2 * pma1, * pma2, * pma3,* pma4;
4
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
10   pma1 = &ma1; pma2 = &ma2;
11   pma3 = &ma3; pma4 = &ma4;
12
13   int c =0;  bool res;
14   double l1=1.2,l2=2.5;
15   res=pma4->certifie(pma1,pma2,pma3,l1,l2);
16   return 0;
17 };
```

Espace matrice 2*2 III

- **Utilisation**: en appelant certifie, on appelle toutes les certifications des classes abstraites. L'utilisateur n'a pas besoin de réfléchir à la certification.

Remarques importantes

- Les fonctions "addition, "zero" etc sont virtuelles donc l'appel dépend du type réel de l'objet.
- Les fonctions "certifie" sont non virtuelles donc depend du type déclaré des objets.
- Dans le code, au-dessus, le "certifie" est celui d"evn" car "mat2" hérite directement d'"evn".
- Cet exemple utilise à la fois les fonctions virtuelles/classiques afin d'appeler les bonnes fonctions au bon moment.
- Les classes abstraites permettent de formaliser/obliger un certain nombre de classes, de comportements.

Espace matrice 2*2 IV

Fonctions virtuelles pures

Les fonctions virtuelles pures doivent être implémentées dans UNE des classes dérivées.

- Exemple: on n'implémente pas zéro dans "mat2"

```
user $ ./evn.hpp:28:16: note: unimplemented pure virtual method 'zero' in
     'mat2'
user $  virtual void zero()=0;
```

- Exemple: on n'implémente pas zéro dans "ev" ou "evn". Puisqu'elle l'est dans '"mat2" ça pose pas de problème.
- Il ne faut pas à chaque classe l'implémenter ou la déclarer virtuelle. Il suffit de l'implémenter dans une classe.

Différence virtuel/virtuel pure et classe abstraite

- Cas: une classe mère, une fille
- Méthodes classiques/virtuelles/virtuelles pures:
  - Une méthode classique peut être implémentée dans les classes filles ou/et mère. La fonction appelée dépend du type de l'objet ou du pointeur.
  - Une fonction virtuelle doit être implémenter au moins dans la classe mère (même implémentation vide). Elle peu être re-implemter chez les filles. La fonction appelée dépend du type réel de l'objet (cas pointeur)
  - Une fonction virtuelle pure, n'a pas d'implémentation dans la classe mère mais forcément une dans la classe fille (dernière fille si il n'y a plus de classes). Appelée par un objet ou un pointeur sur un objet de type fille.
- Classe classique/abstraite:
  - Une classe classique : peut contenir des méthodes classiques, virtuelles peut contenir ou non des données et on peut déclarer un objet de cette classe.
  - Une classe abstraite : peut contenir des méthodes classiques, virtuelles et au moins une méthode virtuelle pure, peut contenir ou non des données et on ne peut pas déclarer un objet de cette classe.

Conclusion

Fonctions virtuelles pures et classes abstraites

Elles permettent de normer et de flécher le comportement des classes dérivées puisqu'il faut fournir les implémentations des fonctions virtuelles. Elles ne sont là que formater les classes dérivées donc pas de déclaration d'objet possible.

Polymorphisme

Permet d'utiliser les classes abstraites. Permet d'utiliser de façon dynamique les objets d'une hiérarchie de classes. En effet, l'utilisation ce fait en fonction du type réel de l'objet et n'est pas décidée lors de la compilation.

Conclusion

Les deux outils sont très utiles pour obtenir des hiérarchies de classes décrivant des problèmes abstraits ou complexes.