KOOC – 3ième Partie

Sommaire

1) Héritage et polymorphisme.

a) Problème de la spécialisation.

Avons-nous résolu le problème de spécialisation de nos librairies avec **@class** ?

La réponse est non, car certes nous sommes capables de créer de nouveaux types de données de manière plus souple qu'en C mais nos types abstraits sont toujours aussi longs à écrire.

Pour réduire considérablement l'ajout de nouveau code et accroître ainsi la réutilisabilité générale des classes précédemment écrites, il nous faut définir deux concepts : similarité et héritage.

Deux fonctions sont dites similaires, si elles ont la même structure logique (condition, expression et boucle) et appellent des fonctions de même signature.

Pour deux interfaces de module logiciel connu (par exemple: liste et pile), il existe un sous-ensemble commun aux deux interfaces dont les fonctions sont similaires.

Cette troisième interface pourrait être extraite. Écrite une fois pour toute et réutilisée afin de définir ces deux interfaces premières.

Ainsi nous n'avons qu'une seule occurrence d'un code qui était précédemment redondant dans chaque interface.

Inversement, on peut dire que les deux interfaces héritent d'une troisième un ensemble de fonctions membres.

Exigence #8 Héritage

Nous devons faire évoluer notre conception de @class pour qu'elle puisse supporter cette manière de réutiliser le code.

Chaque nouveau type doit disposer de ses propres données membres. Il doit pouvoir hériter des données membres d'un autre type.

Pour mettre en place ces propriétés nous allons pratiquer l'agrégation d'interfaces (aussi appelé héritage par extension).

b) Agrégation d'interface

L'agrégation d'interface vient du fait que tous les types B dérivé d'un type A présente une égalité structurelle pour toutes les propriétés qu'elles héritent de A. (à un alignement de structure près) Pour illustrer ceci, prenons 3 structures A, B, C.

```
struct A
{
      int   ai;
      char  *ac;
      double      ad;
};

struct B
{
      int   bi;
      char  *bc;
      double      bd;
      char  bc[10];
      float bf;
};

struct C
{
      int   ci;
      char  *cc;
      double      cd;
      char  bc[10];
      float bf;
      int   cx;
      int   cy;
      int   cz;
};
```

Malgré des noms de champs différents, ces 3 structures ont des parties communes. Nous pouvons les réécrire sous la forme :

```
struct A
{
      int   ai;
      char  *ac;
      double      ad;
};

struct B
{
      struct A    parent;
      char        bc[10];
      float       bf;
};

struct C
{
      struct B    parent;
      int         cx;
      int         cy;
      int         cz;
};
```

Grâce à cela nous avons les égalités suivantes :

```
{
      struct      A     *a, *ta;
      struct B    *b, *tb;
      struct C    *c;

      a = (struct A*) calloc(1, sizeof (struct A));
      b = (struct B*) calloc(1, sizeof (struct B));
      c = (struct C*) calloc(1, sizeof (struct C));
      /* acces au champs ac */
      a->ac;
      b->parent.ac;
      c->parent.parent.ac;
      /* cast possible */
      ta = (struct A*) b;
      ta->ac == b->parent.ac;
      ta = (struct A*) c;
      ta->ac == c->parent.parent.ac;
      /* acces au champs bc */
      b->bc;
      c->parent.bc;
      /* cast possible */
      tb = (struct B*) c;
      tb->bc == c->parent.bc;
}
```

Ce type de structure présente assez bien l'agrégation d'interface au niveau des variables membres.

Exigence #9 Polymorphisme

@class doit pouvoir hériter des fonctions membres d'un autre type. Les fonctions membres du type parent peuvent être réimplémentées.

En ce qui concerne ces fonctions membres nous pouvons imaginer quelque chose de semblable à l'héritage.

Toutefois au lieu de stocker des variables membres, nous allons stocker les pointeurs de fonctions membres. Réécrire la valeur du pointeur de fonction permet de changer le comportement de la classe pour cette fonction. Ici toutefois, nous accédons dorénavant aux fonctions à travers l'instance de la classe.

Nous devons donc lier données et fonctions au sein de l'instance par le mécanisme de vtable.

```
typedef struct vtable1
{
        void      (*func_1)();
        int (*func_2)(int);
}       vtable1_t;

typedef struct vtable2
{
        void      (*func_1)();
        int (*func_2)(int);
        int (*func_3)(char *, ...);
}       vtable2_t;

/* 2 ensemble different de fonction membre */
vtable1_t       list_vtable1 = {myfunc1, myfunc2};

vtable2_t       list_vtable2 = {redef_of_myfunc1, redef_of_otherfunc2,
otherfunc3};

typedef struct
{
      void  *vt;
      int         data;
} A;

typedef struct
{
      A     parent;
      int   data;
} B;

A     *creeA()
{
```

```
      A *res = malloc(sizeof(A));
      res->vt = (void *) list_vtable1;
      return res;
}

B     *creeB()
{
      B *res = malloc(sizeof(B));
      res->vt = (void *) list_vtable2;
      return res;
}

void  function_utilisatrice()
{
      A     *a = creeA();
      A     *b = creeB();//B herite de A
/*
 appele la fonction 1 sur A et B
*/
      ((vtable1_t*) a->vt)->func_1();// appel myfunc1
      ((vtable1_t*) b->vt)->func_1();// appel redef_of_myfunc1
}
```

Cette façon d'écrire les structures n'est cependant pas totalement conforme à ce que nous voulons mettre en place pour notre pile.

Que se passe-t-il en cas de collision de signature (variables membre ou fonctions membres) d'une interface parente et enfante ?

Tout en gardant le principe d'agrégation d'interfaces qui permet de garder la trace de l'ensemble de la liste d'héritage, la résolution à la compilation d'une signature sur une variable membre ou une fonction membre doit se faire de manière différente. Une signature de fonction est unique quelles que soient les interfaces dérivées. Une collision peut être considérée comme une réimplémentation de la fonction (il nous faut un nouveau mot-clef). KOOC va supporter le polymorphisme au niveau des fonctions lors de l'emploi du mot-clef **@virtual**. C'est-à-dire que pour une morphologie similaire entre deux objets (même forme donc même ensemble de signatures) le comportement peut changer en fonction du type réel de l'objet.

Par contre, le polymorphisme pour une variable n'a pas de sens. Les variables membres encapsulées dans la structure de l'instance sont considérées comme appartenant à un espace de noms qui est l'instance elle-même. On doit pouvoir accéder à toutes les variables membres de l'héritage. On peut se limiter à l'agrégation de structures pour les variables.

2) LE CHOIX DE C++

Pour répondre, aux 2 nouvelles exigences C++ altère la syntaxe de **struct** et introduit le mot-clef **virtual**.

a) Exigences logicielles :

Exigence #8 Héritage : les structures sont agrégées par la syntaxe suivante :

```
struct vide1 { int f();};// sizeof(vide1) == 0
struct A1 : vide1 {int a; int f();};// sizeof(A1) == 4
struct B1 : A1 {int b; int f();};// sizeof(B1) == 8
```

f est définit en tant que membre pour chaque classe. Mais il est différent à chaque fois.

```
vide1 v;
v.f(); // appel de vide1::f()
A1 a;
a.f(); // appel de A1::f()
B1 b;
b.f(); // appel de B1::f()
A1 *p = &b;
p->f(); // appel de A1::f()
```

Exigence #9 Polymorphisme : le mot clef **virtual** s'utilise de la manière suivante :

```
struct vide2 {virtual int f();};// sizeof(vide2) == 4
struct A2 : vide2 {int a; int f();};// sizeof(A2) == 8
struct B2 : A2 {int b; int f();};// sizeof(B2) == 12
```

par contre ici :

```
B2 b;
A2 *p = &b;
p->f(); // appel de B2::f()
```

Le seul moyen est de passer par la vtable expliquée au début de ce cours. La présence de la vtable peut être identifiée en C++ de cette manière:

```
struct magic
{
void  f() {cout<< «cool» << endl;}
virtual g() {cout<< «pas cool» << endl;}
};

magic &r = *((magic*)0);
r.f();// marche.
r.g();// segfault.
```

3) NOTRE CHOIX

Ici nous présentons succinctement les différents nouveaux mot-clefs introduits dans KOOC par rapport aux exigences, le reste est vu en détails dans les TPs.

a) Exigences logicielles :

Exigence #8 Héritage : nous pouvons faire de l'agrégation avec **@class**.

```
@class A
{
@member int a;
} // sizeof(A) == 4

@class B : A
{
@member int b;
} // sizeof (B) == 8
```

Exigence #9 Polymorphisme : nous inscrivons dans la vtable une fonction en la préfixant du mot clef **@virtual**. Une fonction **@virtual** est forcément membre, il n'est donc pas nécessaire de la marquer comme membre.

```
@class C
{
      @virtual void tutu();
      @virtual {
            int grumf();
            double gromf();
      }
      @member
      {
            int a;
            @virtual void g();
      }
}
```