KOOC – TP 3ième Partie

Sommaire

1) Etape 3

Vous devez rajouter les mot-clefs suivants à votre partie 2 de KOOC. Ces ajouts ne devront pas modifier le comportement spécifier précédemment.

2) @class

Synopsis

```
@class Nom_Classe ou @class Nom_Classe : Nom_Classe_parente
{
@member{
// variables membres
// s'il n'y a pas de variable membre on peut se passer des accolades
}
// variable non-membre
// fonctions membre et fonction non-membre
}
```

Description

@class fournit maintenant le support de l'héritage. Nous pouvons utilisez **[super.membre**] pour accéder au membre de la classe parente, ou **[super fonction]** pour appeler une fonction de la classe parente. **super** existe uniquement dans les fonctions membres d'une classe ayant un parent.

Exemple

```
----- Stack.kh
@class Stack
{
      @member  {
            int size;
            int nbitem;
      }
/* pour compter le nombre de pile presente dans le systeme */
      int         nbstack = 0;
      @member     void  init(int);
      @memeber    int   nbitem();
}
----- Stack.kc
@implementation Stack
{
  @member {
      void  init(int size)
      {
            [super init];
            [self.size] = size;
      }

      int   nbitem()
      {
            int n;

            n = [self.nbitem];
            return (n);
      }
  }
}
```

```
---- StackInt.kh
@import "Stack.kh"
@class StackInt : Stack
{
      @member int *data;
/* pour compter le nombre de pile presente dans le systeme */
      int   nbstack = 0;
      // fonctions membres
      @member void init(int);
      @member void clean();
      // ou
      @member {
      int   nbitem();
      void  push(int);
      int   pop();
      }
}
---- StackInt.kc
@import "StackInt.kh"
@implementation StackInt
{

@member void      init(int size)
{
      int   *buf;

      [super init :size];
      buf = (int *) calloc(size, sizeof(int));
      [self.data] = buf;
}

@member void      push(int i)
{
      int pos;
      int *buf;

      pos = [self.nbitem];
      buf = [self.data];
      buf[pos++] = i;
      [self.nbitem] = pos;
}

@member int pop()
{
      int *buf;
      int pos;
      int r;

      pos = [self.nbitem];
      buf = [self.data];
      r = buf[pos--];
      [self.nbitem] = pos;
      return (r);
}
```

```
@member void      clean()
{
      int *buf;

      buf = [self.data];
      free(buf);
}
}
```

Quelques précisions:

Que se passe-t-il lorsqu'il y a collision entre une variable membre et une variable non-membre?

Les données ne sont pas stockées au même endroit, donc il n'y aura pas d'ambiguïté au moment de l'appel.

Que se passe-t-il pour les instances sur la pile?

En effet pour les données présentes sur la pile, il va manquer quelques initialisations nécessaires pour en faire de vrais objets.

On va donc considérer qu'un objet non initialisé n'est pas un objet mais juste un espace mémoire. Il faudra donc toujours initialiser les structures sur la pile pour en faire un objet, même avec l'initialiseur par défaut.

```
A* oneObjectOnTheHeap;

[oneObjectOnTheHeap Function :params];
// erreur, oneObjectOnTheHeap n'est pas un objet
oneObjectOnTheHeap = [A new];
[oneObjectOnTheHeap Function :params];
// ok, oneObjectOnTheHeap est un objet

A oneObjectOnTheStack;
[&oneObjectOnTheStack Function :params];
// erreur, oneObjectOnTheStack n'est pas un objet
memset(&oneObjectOnTheStack, 0, sizeof([A.null]));
[A init :&oneObjectOnTheStack];
[&oneObjectOnTheStack Function :params];
// ok, oneObjectOnTheStack est un objet
```

Par défaut on considère que si l'utilisateur réimplémente le **init** par défaut, ou code un autre **init** il faudra appeler le **init** du parent ou le **init** par défaut de Object dans la fonction **init** implémentée via "**[super init]**;" (le constructor chaining se fait à la main).

2) @virtual, Upcasting/Downcasting, Héritage et polymorphisme

Description

On rajoute à la manière de @member, le mot-clef @virtual qui démarque les fonctions polymorphes. Les fonctions **@virtual** appelées travaillent avec le type réel. Il n'y a pas grand chose à faire pour le upcast (cast vers le type parent) car il n'y a pas de réelle conversion. Le cast vers le type réel (downcast ou dynamic cast) est pour l'instant non demandé bien qu'il soit faisable de manière non propre.

Exemple

```
--- test_poly.kh

@class A
{
      @member {
            int     value;
            void  init();
            @virtual void     print();
      }
}

@class B : A
{
      @member     int     value;
      @member     void  init();
      // @virtual peut etre  ecris hors de @member, la fonction est
necessairement member ET virtual
      @virtual    void  print();
}

@class C : B
{
        @member int     value;

      @member  void     init();
      @virtual void     print();
}
--- test_poly.kc
@import "test_poly.kh"
@implementation A
{
@member void    init()
{
  [super init];
  [self.value] = 42;
}

@virtual void     print()
{
  int value;
```

```
  char      *name;

  name = [self name_of_interface];
  value = [self.value];
  print("%d %s\n", value, name);
}
}

@implementation B
{
@member void    init()
{
  [super init];
  [self.value] = 124;
}

@virtual void     print()
{
  int value;
  char      *name;

  name = [self.name_of_interface];
  value = [self.value];
  print("%d %s\n", value, name);
}
}

@implementation C
{
@member {
      void    init()
      {
        [super init];
        [self.value] = 666;
      }

      @virtual void     print()
      {
        int value;
        char      *name;

        name = [self.name_of_interface];
        value = [self.value];
        print("%d %s\n", value, name);
      }
}
}
----- main.kc
@import "test_poly.kh"
int   main(int ac, char **av)
{
  A   *a;

  a = [A new];
  [a print];      // affiche "42 A\n"
```

```
  a = [B new];
  [a print];      //affiche "124 B\n"
  a = [C new];
  [a print];      //affiche "666 C\n"
  [A print :a];   //affiche "42 C\n"
}
```

3) Librairie de runtime et type 'Object*'

Description

On considère que toutes les classes dérivent d'un même type nommé 'Object'. Voici la description (virtuelle) de sa classe. Concrètement vous ne pourrez pas compiler cette classe. Elle est juste donnée pour exemple. Vous devrez l'écrire avec **@module**. Il se peut aussi que pour votre implémentation vous deviez ajouter des éléments à cette classe. Faites donc ! Tout n'est pas donné.

```
@class Object
{
      @member const char      *name_of_interface();//nom reel de l'interface

      Object            *alloc(); //allocateur
      Object            *new();//constructeur par defaut
      @member {
      void        init();//initialisateur par defaut
      @virtual void           clean();//nettoyeur par defaut
      void        delete();//destructeur par defaut
      @virtual int            isKindOf(const char *);
      @virtual int            isKindOf(Object *);
      @virtual int            isInstanceOf(const char *);
      @virtual int            isInstanceOf(Object *);
      }
}
```

isInstanceOf

isInstanceOf est une fonction qui teste si l'instance est de la même classe que le nom d'une classe passée en paramètre ou si elle dérive de la même classe que l'objet passé en paramètre. Elle retourne une valeur > 0 si oui, sinon 0.

isKindOf

isKindOf est une fonction qui cherche dans la liste d'héritage si l'instance dérive d'une classe dont le nom est passé en paramètre ou dont un exemplaire est passé en paramètre. Elle retourne une valeur > 0 si oui, sinon 0.

```
----- main.kc
@import "test_poly.kh"
/*
       Toujours avec l'exemple precedent.
*/
int   main(int ac, char **av)
```

```
{
  A   *c;
  int res;

  c = [C new];
  res = [c isInstanceOf :"A"];
  /* Affiche No*/
  if (res)
      printf("Yes\n");
  else
      printf("No\n");
  res = [c isKindOf :"A"];
  /* Affiche Yes*/
  if (res)
      printf("Yes\n");
  else
      printf("No\n");
  B* b = [B new];
  res = [c isKindOf :b];
  /* Affiche Yes*/
  if (res)
      printf("Yes\n");
  else
      printf("No\n");
}
```

4) Tips

Le problème le plus épineux pour l'ajout de ces fonctionnalités est de trouver le 'modèle objet'. Pour cela il faut utiliser le principe de l'agrégation de structure pour simuler les objets. Et surtout, il faut traiter séparément les fonctions membres des variables membres. Pour ce qui est des variables et fonctions non-membres il faut les gérer exactement comme des variables et fonctions de modules.

Héritage

L'héritage pour les variables se gère facilement avec l'agrégation de structure. Comme il n'y a pas de polymorphisme pour les variables chaque variable de chaque classe est présente dans l'arbre d'héritage ! Le plus simple est de directement copier/coller la définition de la classe parente dans la classe enfante.

Exemple:

```
@class C
{
@member {
  int a;
  float a;
}
}

@class D : C
```

```
{
@member {
  int a;
  float a;
}
}

/* Va generer quelque chose comme ceci. */

typedef struct _kc_interface_C
{
int   ~~~C~~~int~~~a~~~;
float ~~~C~~~float~~~a~~~;
}     C;

typedef struct _kc_interface_D
{
int   ~~~C~~~int~~~a~~~;
float ~~~C~~~float~~~a~~~;
int   ~~~D~~~int~~~a~~~;
float ~~~D~~~float~~~a~~~;
}     D;
```

Les 2 structures ont une partie commune et le plus important c'est qu'il n'y aura pas d'indirection compliquée pour accéder à tous les champs de l'arbre d'héritage. Ainsi une structure D est castable en C (attention au alignement, à vérifier suivant le compilo quel pragma utiliser).

```
/* n'importe ou dans le code */

D     *d;

d = [D new];
[d.a] = 42;

/* va donner le code suivant */

D   *d;

d = ~~~D~~~new~~~();
d->~~~D~~~int~~~a~~~ = 42;

/* Dans une fonction membre de D */
@member int   FonctionMembre()
{
[super.a] = 42;
....
}

/* va donner le code suivant.
Attention: comme indiquer, une fonction membre == une fonction
non-membre qui prend en premier paramètre un pointeur sur l'instance de
l'objet. Il faut donc pour chaque fonction membre faire comme s'il sagit
```

```
d'une fonction non-membre et rajouter le paramètre en plus.
Pour simplifier et gérer en même temps le mot clef [self...] autant appèle
ce paramètre en plus 'self'.
 */
int   ~~~D~~~FonctionMembre~~~(D *self)
{
self->~~~C~~~int~~~a = 42;
....
}
```

Fonction membres

Pour la gestion des fonctions membres polymorphe (**@virtual**), il faut voir que tous les objets descendent d'un ancêtre commun appelé 'Object'.

Les exemples du cours précédent présentent l'utilisation de structure de pointeur de fonction stockant l'interface de programmation. L'interface est identique pour les fonctions héritées. Ensuite l'interface est étendue des nouvelles fonctions de la classe dérivée. Comme l'ordre des fonctions va être capital pour la gestion du polymorphisme, il faut classer les fonctions membres dans une structure à part.

Finalement, on doit pouvoir accéder à ces informations à travers n'importe quelle instance d'objet. Il faut donc rajouter une variable membre dédiée à cette tache dans la classe Object. Ainsi :

```
/* avec le fichier .kh suivant */
@class A
{
      @virtual void     print();
}

/* fichier .h générer */
/* En tenant compte que A dérive de Object et suivant le sujet.
*/

typedef struct _kc_interface_A
{
/* pointeur générique sur la liste de fonction de l'interface */
void        *~~~Object~~~vtable~~~;
/* Fin des variables hérite de Object */
/* variable membre de A */
...
}           A;

/* C'est cette partie qui est important à générer */
struct            _kc_implementation_A
{
/* Comme pour les variables on recopie mangler la définition des fonctions
de l'interface parente et surtout dans l'ordre de déclaration de l'interface.
*/
void        (*~~~Object~~~clean~~~)(Object *);
void        (*~~~Object~~~isKindOf~~~)(Object *,const char *);
void        (*~~~Object~~~isKindOf~~~)(Object *,Object *);
```

```
void        (*~~~Object~~~isInstanceOf~~~)(Object *,const char *);
void        (*~~~Object~~~isInstanceOf~~~)(Object *,Object *);
/* fonction membre de A */
void        (*~~~A~~~print~~~)(A *);
};
```

Ceci est généré dans les fichiers '.h', il faut ensuite associer cette structure avec le champ supplémentaire 'vtable' de Object dans la partie implémentation.

```
@import "A.kh"

@implementation A {}
/* fichier .meta.c générer */

/* instanciation des fonctions pour l'interface A
   avec les fonctions par défaut fournie par l'ancêtre object.
*/
_kc_implementation_A    _kc_vtable_A =
{
      /* Cette partie est hérité de Object */
      ~~~A~~~clean~~~,
      ~~~Object~~~isKindOf~~~,
      ~~~Object~~~isKindOf~~~,
      ~~~Object~~~isInstanceOf~~~,
      ~~~Object~~~isInstanceOf~~~,
      /* Cette partie est généré automatiquement pour A */
      ~~~A~~~print~~~
};

/* Le plus important est d'ensuite de générer (ou compléter) la fonction init
par défaut qui associe cette structure a vtable. */

void  ~~~A~~~init~~~(A *self)
{
      ...
      self->~~~Object~~~vtable~~~ = (void*)&_kc_vtable_A;
      ...
}
```

Grâce à toutes ces opérations, il est désormais possible d'appeler une fonction membre à travers son instance (pointeur sur la structure correspondant à l'interface). Il faut toutefois connaître l'indice (index) de la fonction dans la classe au moment de l'appel. Pour plus de détail, étudions le cas de B tel que B dérive de A.

```
@interface A
{
@member void      print();
}

@interface B : A
{
@member void      print(); /* redefinition de print */
}
```

```
/* Comme precedement on generere pour B */
/* Dans le fichier .h */
struct           _kc_implementation_B
{
/* Comme pour les variables on recopie mangler la definition des fonctions
de l'interface parente et surtout dans l'ordre de declaration de l'interface.
*/
void       (*~~~Object~~~clean~~~)(Object *);
void       (*~~~Object~~~isKindOf~~~)(Object *,const char *);
void       (*~~~Object~~~isKindOf~~~)(Object *,Object *);
void       (*~~~Object~~~isInstanceOf~~~)(Object *,const char *);
void       (*~~~Object~~~isInstanceOf~~~)(Object *,Object *);/* fonction
membre de A */
void       (*~~~A~~~print~~~)(A *);
/* fonction membre de B */
/* --- vide --- "print" est definie dans A */
};

/* Dans le fichier .meta.c */
_kc_implementation_B    _kc_vtable_B =
{
     /* Cette partie est herite de Object */
     ~~~B~~~clean~~~,
     ~~~Object~~~isKindOf~~~,
     ~~~Object~~~isKindOf~~~,
     ~~~Object~~~isInstanceOf~~~,
     ~~~Object~~~isInstanceOf~~~,
     /* Cette partie est herite de A */
     ~~~B~~~print~~~
     /* Cette partie est generer automatiquement pour B */

};

/* Mais aussi l'initialisateur */

void  ~~~B~~~init~~~(B* self)
{
     ...
     self->~~~Object~~~vtable~~~ = (void*)&_kc_vtable_B;
     ...
}
```

Notre pointeur 'vtable' pointe, quelle que soit les instances, sur un ensemble de fonction conforme au type de l'objet. Si pour un objet B nous castons le champ vtable en _kc_implementation_A* est que nous accédons à la fonction 'print', seul est accessible la fonction print de A :

```
B     *b;
b = ~~~B~~~new~~~;
((_kc_implementation_A*)b->vtable)->~~~A~~~print~~~(b);
```

Ici nous appelons l'implémentation de la classe B. Ce qui est voulu. La définition du polymorphisme veut que nous appelions la fonction membre associé au type réel de

l'instance. Donc la fonction print de l'interface B, car c'est un B qui est instancié. Tout cela se gère au niveau de _kc_vtable_B. Pour faciliter les recherches dans _kc_vtable_B Il est intéressant de dissocier 2 choses dans le mangling :

- Le contexte: Module Y ou classe Z
- La signature: void print()

Maintenant à vous de vous approprier ce modèle et de le compléter.