KOOC - TP 2ième Partie

Sommaire

1) Etape 2

Vous devez rajouter les mots-clefs suivants à votre partie 1 de KOOC. Ces ajouts ne devront pas modifier le comportement spécifié précédemment.

2) @class

Synopsis

```
@class Nom_class
{
      @member
      {
      // variables membres
            int a;
      // fonctions membre
            void myMemberFunction();
      }
// variable non-membre
int nonmember;
//fonction non-membre
void myNonMemberFunction();
// variable membre
@member int member;
//fonction membre
@member void myOtherMemberFunction();
//autre ecriture de fonction membre
void myOtherMemberFunction2(Nom_class *);
}
```

Description

@class fournit le support des types abstraits manquant aux modules KOOC. Le mot-clé **@implementation** sert aussi pour implémenter une classe.

Cependant les fonctions, variables membres sont introduites par **@member**.

Par convention les fonctions membres prennent en premier paramètre l'instance sur laquelle porte la fonction.

@member devant une fonction simplifie l'écriture des fonctions membres. Cependant, comme tout dépend de la signature, si la fonction prend en premier paramètre un pointeur sur le type de la classe, ça sera une fonction membre.

Voyez **myOtherMemberFunction2**! Nous utilisons 'self' comme mot-clef par défaut pour designer ce pointeur.

Comme les variables membres sont manglées dans le pointeur de structure self. L'objet est totalement opaque. Nous sommes donc obligés de passer par une écriture du type **[self.nom_variable_membre]** pour accéder aux membres de l'instance.
Cependant comme nous pouvons passer par l'équivalence d'écriture entre le module et la classe, nous pouvons lui donner un autre nom. Voyez l'exemple de **@implementation** de la fonction clean.

Exemple

```
---- StackInt.kh

@class StackInt
{
      @member
      {
            int size;
            int nbitem;

            void  init(int);
            void  clean();
            int   nbitem();
      }

      @member int *data;
      @member void      push(int);
            int   pop(StackInt *);//pop fonction membre
/* pour compter le nombre de pile presente dans le systeme */
      int   nbstack = 0;
}
---- StackInt.kc
@import "StackInt.kh"
@implementation StackInt
      @member void      init(int size)
      {
            int   *buf;

            [self.nbitem] = 0;
            [self.size] = size;
            buf = (int *) calloc(size, sizeof(int));
            [self.data] = buf;
      }

    @member
    {
      int   nbitem()
      {
            int n;

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

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

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

```
    }

      int   pop(StackInt *self)
      {
            int *buf;
            int pos;
            int r;

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

      void  clean(StackInt *this)
      {
            int *buf;

            buf = [this.data];
            free(buf);
      }
}// fin @class
```

Quelques précisions:

Appel ambigu? Même si l'appel KOOC a évolué, vous devez rester compatible avec KOOC 1.

```
v = [a.b];// acces a un champs 'b' du module 'a'
v = [a b];//soit a est un module soit c'est une variable dont le type est
objet
      // b est alors une fonction membre
```

Appel sur objet

```
@class Toto
{
      ...
}
// a partir d'ici un nouveau type 'Toto' est connu
//nous  pouvons donc faire des siouxeries

typedef Toto mumu; // alias de type


...// quelque part dans un block de code
{
      Toto        t;//variable sur la pile
      Toto        *tp;
      void        *t2;

      [Toto init :&t]; // initialisation d'un objet sur la pile
```

```
      tp = &t;// cast implicite
      [Toto init :tp];//ok
      [tp init];//ok
      t2 = tp;
      [t2 clean];//erreur
      [tp clean];//ok
}
```

3) Allocateur, Initialisateur & Destructeur

Synopsis

```
[* alloc]
[* new ]
[* delete]
```

Description

Le script KOOC ajoute automatiquement pour chaque classe créée :

- Une fonction non-membre d'allocation de l'instance de l'objet nommée '**alloc**'. Elle alloue l'espace mémoire nécessaire pour stocker toutes les variables membres sur le tas (heap).
- Une fonction membre de destruction nommée **'delete'** qui désalloue la mémoire allouée précédemment par '**alloc**' et qui tente d'appeler une fonction membre nomme '**clean**' si celle-ci est définie.
- Une fonction non-membre de construction nommée '**new**' par surcharge de la fonction **'init'** présente dans le code et qui appelle '**alloc**' puis la surcharge de la fonction **'init'** correspondante. Le type de retour de la fonction **new** est un pointeur sur une structure portant le nom de la classe. Il y a une fonction **new** par fonction **'init'** différente.

Exemple

```
--- main.kc
@import "StackInt.kh"

int main(int ac, char **av)
{
      StackInt    *my_stack;

      my_stack = [StackInt new :4242];
      [my_stack push :10];
      [StackInt push :my_stack :11];//appel compatible module
      [my_stack push :12];
      [my_stack push :14];
      [my_stack delete];

      StackInt    local_stack;
      [&local_stack init :4242];
```

```
      [&local_stack push :33];
}
```

Note: l'appel " [StackInt push :my_stack :11]; " est équivalent à " [my_stack push : 11]; " par définition.

4) [];

Synopsis en BNF

A ajoutez en tant que *primary_expression* au Cnorm

```
primary_expression ::=
... /* les regles existantes  */ ...
| appel_KOOC
;

appel_KOOC ::= '[' identifiant_module_instance
      [identifiant_fonction list_parametre]?
      ']'
;

list_parametre ::= [':' assignment_expression ]*
;

identifiant_module_instance ::= identifier ['.' identifier]?
;

identifiant_fonction ::= identifier
;

identifiant_variable ::= identifier
;
```

Description

Modification d'un appel KOOC pour tenir compte de l'instanciation. Dans un bloc d'implémentation d'une interface le mot-clef '**self**' a une signification particulière. **'self'** désigne l'instance courante.

Exemples

Cf. @class.

5) Tips

Variables membres

Par convention le mot-clef **@class** génère un type de structure du même nom. C'est -à-dire que le code généré contient un **typedef** et une structure.

Par exemple :

```
@class A {}

/* genere */

typedef struct _kc_interface_A
/* vous pouvez donner un nom interne a la
      structure suivant la convention qui vous convient.
      Ce nom interne vous permets de gerer les cas comme :
      @class List
      {     @member {List     *next;}
      }
*/
{
      // ici les membres
}     A; /* le nom de A est important */
```

ATTENTION TOUTEFOIS A BIEN INSTRUMENTER L'ARBRE POUR QU'IL SOIT CAPABLE DE RECONNAITRE LE TYPE.

Pour ce qui est des variables membres, elles correspondent aux différents champs de la structure dans l'ordre de définition (ordre des différents appels à **@member**). Leur nom est cependant manglé.

Attention dans l'exemple le '~~' avant et après le symbole signifie qu'il est manglé. '~~A~~' signifie que le symbole est manglé dans le contexte A.

```
@class A
{
      @member     int   a;
      @member     int   b;
}

/* donne */

typedef struct _kc_interface_A
{
int   ~~A~~a~~;
int   ~~A~~b~~;
}     A;
```

Pour le mangling de la classe, c'est exactement le même principe que pour un module. Doivent apparaître la signature complète du symbole (variable ou fonction) et le contexte auquel il appartient (nom du module/classe)