# Dossier de Conception
# KOOC

ELKAIM_R
AMSTUT_A
LEDARA_F
DOUZIE_L
BARAOU_C
SLAIM_L

Octobre 2015

# I. Sommaire

# II. Buts du document

Ce document concerne la réalisation d'une surcouche au langage C au moyen de l'outil Pyrser et Cnorm. Cette surcouche servira à augmenter les possibilités du langage:

1. Utilisation de modules
2. Ajout d'un système de classes
3. Héritage simple/virtuel
4. Notions de public/privé
5. Exceptions
6. ...

# III. Documents applicables et de référence

## *A) Documents applicables*

- Licence publique générale GNU: http://www.gnu.org/licenses/gpl-3.0.fr.html

## *B) Documents de référence*

- Sujet du KOOC: https://intra.epitech.eu/module/2015/B-PAV-475-1/PAR-5-1/acti-187352/project/file/14_KOOC_Sujet.pdf, Lionel Auroux

- Documentation de Pyrser: http://pythonhosted.org/pyrser/

- Documentation de Cnorm: http://pythonhosted.org/cnorm/

# *IV.* Contexte de la conception

## *A) Présentation de l'application*

L'application a pour but d'apporter une surcouche au langage C. Elle s'occupe d'une pré-compilation du programme. Ces améliorations visent à apporter certaines fonctionnalités des langages objets dans le langage C.

En effet, la programmation orientée objet apporte de multiples avantages. L'encapsulation permise par le système de classes permet d'architecturer un programme beaucoup plus simplement qu'en programmation impérative. De plus, le programme final permettra l'implémentation d'interfaces, qui servent à décrire un ensemble de fonctionnalités pour un composant logiciel.

A la fin de son exécution, notre programme renverra du code C compilable de façon classique

## *B) Principales exigences applicables*

### 1) Exigences techniques:

Le programme intervenant pendant la période de compilation, il doit pouvoir faire remonter les erreurs (syntaxe, domaine de définition, …) de façon claire à l'utilisateur.

Le fichier C renvoyé par le KOOC devra toujours être valide et ne pas avoir besoin d'être modifié par l'utilisateur avant d'être compilé en exécutable.

### 2) Exigences du langage:

Les ajouts faits au langage C doivent avoir une syntaxe la plus simple possible, tout en étant distinguable simplement par rapport au langage de base.

Les mots clés utilisés par le KOOC doivent être similaires à ceux utilisés dans la plupart des langages objets, pour qu'un utilisateur puisse rapidement s'adapter.

# V. Description des tâches

## *A) DECORATEUR DE SYMBOLES*

Le décorateur de symboles a pour but de créer un symbole unique pour chaque variable et fonction présentes dans un module.

Nous allons donc créer une classe «Mangler» qui s'en occupera.

Cette classe se base sur l'utilisation de la classe "Module" Décrite dans la partie D. suivant la convention suivante; elle se base sur le prototype de chaque fonction contenue dans un module ou une classe pour établir son nom une fois en C:

**["Func"/"Var"]'$'["Module"/"Class" '_' module/class name]?'$'OriginalName'$'[[P]* [U]?'$'ReturnType'_'[[P]* [U]?'$'argsN'_']["$$$variadic$$$"]?**

De cette manière, une fonction avec le prototype :

int toto(char *stuff, void *something);

Dans le module tata donnera :

Func$Module_tata$toto$$int_P$char_P$void

De même, une variable 'int foo' dans le module 'bar' donnera :

var$module_bar$foo$$int

À noter:

En cas de collision à l'issue de la décoration, un nombre sera ajouté à la fin, jusqu’à ne trouver aucune collision.

- Le nom de chaque type commence par un $.
- un P est ajouté au nom du type selon la profondeur du pointeur (un void ** correspondrait donc à PP$void).
- Un U correspond à un entier non signé et se retrouve après le(s) p symbolisant les pointeurs.

## *B) IMPORT*

Le but de la fonction import est de connaitre des types déclarés dans un fichier .kh.

Pendant le parsing, à chaque rencontre d'un mot clé "@import fichier", nous allons faire un appelle récursif à Kooc pour parser ce fichier et en récu-pérer l'ast.

Grâce à cet ast, nous allons pouvoir lister toutes les déclarations faites dans ce fichier pour les mémoriser. Ces déclarations sont alors stockées dans une liste appartenant au singleton "DeclKeeper". Cela nous permettra ensuite de vérifier si les fonctions, variables et type utilisé dans le fichier courant sont connu.

Nous ajoutons ensuite à l'ast courant un objet " Raw("#include fichier\n") " afin d'ajouter la directive "Include" dans le fichier C généré.

## *C) MODULE*

Le module va servir a rassembler des entités logiques du code. Il regroupe un ensemble de fonctions, variables et classes.

Pendant le parsing, chaque module rencontré via le catch de « @module » sera représenté sous la forme d'un objet « module ».

Cet objet module va stocker dans un dictionnaire chaque déclaration faite avec comme clef le symbole manglé. Il sera stocker dans le singleton "DeclKeeper" dans un dictionnaire avec comme clef le nom du module.

## *D) CLASSE*

La classe se comporte comme un module mais définit en plus un type abstrait ce qui veut dire qu'il est possible de l'instanciée et donc de lui donner des fonctions et variables via le mot clef « @member »

**@member :** @member est un préfixe sur les variables ou les fonctions dans une classe pour désigner ses membres.

A chaque rencontre de "@class" la classe rencontrée va être stockée dans un objet de la classe "classe" qui possèdera un dictionnaire de ses attributs membres et un autre de ses attributs non membres. A la création de cet objet un typedef de structure sera généré. Cet objet sera stocké dans le singleton "declkeeper".

## *E) IMPLEMENTATION*

L'implémentation est le code des fonctions qui se trouve dans les modules ou les classes.

Lors de la rencontre d'un "@implémentation" nous allons créer un objet implémentation qui va contenir un dictionnaire liant le symbole décoré d'une implémentation à son corps.

Un flag dans l'objet Kooc est utilisé pour savoir si nous sommes dans une récursion ou non pour ne pas récupérer les implémentations dans un fichier importé.

## *F) HERITAGE*

### 1) HERITAGE SIMPLE

L'héritage consiste à récupérer des fonctions et variables d'une classe mère dans une classe fille afin d'éviter la duplication de code.

Lorsqu'un héritage est détecté, nous allons stocker l'objet Class "mère" dans l'objet Class "fille" afin de remonter la chaîne de l'héritage pour résoudre les appels de fonction.

De base nous ne récupérons pas les prototypes des fonctions héritées dans la classe fille sauf celles qui sont ré implémentées. Cela nous permet de savoir dans quelle classe chaque fonction est implémentée.

Dans le code C résultant, une structure de la classe mère sera contenue dans la structure fille. Cela permet d'appeler une fonction membre de la classe mère avec une instance de la classe fille.

### 2) LA CLASSE "OBJECT"

Toutes les classes dérivent de la classe "Object" qui définit un ensemble de fonctions utilitaires pour toutes les classes:

- Name_of_interface: Retourne le nom de la classe.
- Alloc: Alloue la mémoire pour l'objet instancié. Cette fonction sera redéfinie pour chaque classe.
- New: Cette fonction va instancier l'objet. Elle va d'abord appeler la fonction "alloc" puis la fonction "init".
- Init: Cette fonction va remplir les champs de l'objet. Elle sera redéfinie pour chaque classe pour remplir leurs champs spécifiques.
- Clean: Libère les champs de l'objet. Cette fonction sera définie par l'utilisateur.

- Delete: Cette fonction va appeler la fonction "clean" de l'objet puis libérer la mémoire utilisée par celui-ci.

- IsKindOf: Cette fonction pourra prendre en paramètre soit un "Object *" ou un nom de classe. Elle vérifie si la classe est dans sa liste d'héritages ou si elle est du même type que l'objet.

- IsInstanceOf: Cette fonction pourra prendre en paramètre soit un "Object *" ou un nom de classe. Elle vérifie que l'objet reçu en paramètre est du même type que l'objet ou que les noms soient les mêmes.

Dans le code C généré, un typedef sera crée pour définir la structure "Object" qui contiendra le nom de la classe, sa liste d'héritages et sa "vtable". Un objet de cette structure sera contenu dans chaque classe héritant d'"Object".

### 3) VIRTUAL

Le mot clé "@virtual" s'utilise de la même manière que "@member", une fonction virtuelle est à fortiori une fonction membre. La différence est qu'une fonction virtuelle est appelée sur le type réel de l'objet à savoir que par exemple, une fonction virtuelle appelée sur un objet A contenu dans un objet B appellera la méthode de l'objet A bien qu'il soit considéré comme un objet B.

La "vtable" est une structure possédant des champs et qui permet d'appeler les fonctions avec le type réel de l'objet et pas celui dans lequel il est contenu. Par exemple :

"""

A nomVariable #le type A hérite du type B

B nomvariable2 = (B)nomVariable

B.méthode() #methode étant virtuelle c'est celle du type A qui est appelée.

"""

Chaque classe possèdera une structure de la forme "vt_className" où "className" est le nom de la classe. Les champs de cette structure seront les noms manglés des fonctions de la classe et des classe parentes. Ce manglage ne possèdera ni identifiant de variable car cela ne sera que des fonctions et ni référence au module car les fonctions pouvant être redéfinies, la classe à laquelle elles appartiennent n'a pas d'importance. Nous conservons également l'ordre d'héritage des fonctions.

## *G) TYPEUR*

Le typeur a pour but de déduire le type de certaines variables et fonctions (retour, paramètres), pour permettre la surcharge de variables et fonctions dans du code.

Après le parsing, le typeur va faire sa passe et il va ajouter une variable "Type" dans les Nodes de l'ast contenant un nom, un type de retour et la liste des paramètres pour une fonction.

Le typeur descend récursivement dans tous les Nodes et il évalue le type de l'expression. Par exemple: si on trouve un "@!(type)", le type de l'expression est le type en paramètre sinon on le déduit.

# VI. Déroulement du programme

## *A) Parsing*

- Création du module "Object"
    - Définition et implémentation des fonctions de base
    - Définition et déclaration de la structure "Object"
    - Définition et déclaration de la "vtable Object"

- Rencontre de "@class"
    - Récupération de l'héritage
    - Récupération des variables du parent
    - Sauvegarde du lien d'héritage
    - Sauvegarde des variables et fonctions membres
    - Sauvegarde des virtuelles
        - Déclaration de la "vtable"
        - Copie des champs de la "vtable" parent
        - Ajout des nouvelles fonctions virtuelles
    - Sauvegarde des variables et fonctions non membres
    - Décoration des symboles

- Rencontre "@module"
    - Sauvegarde des déclarations
    - Décoration des symboles

- Rencontre "@Import"
    - Parsing du fichier importé pour connaître les déclarations faites dedans

        - Ajout de l'include vers le fichier "h" correspondant au "kh" importé

- Rencontre "@Implémentation"
    - Récupération nom classe ou module
        - Module
            - Décoration des symboles
            - Ajout des fonctions dans l'AST
        - Classe
            - Décoration des symboles
            - Sauvegarde de la fonction
            - Vérification si la fonction est virtuelle (si oui, changement du lien dans la "vtable")

- Rencontre de « [] »
    - Sauvegarde de l’appel

## *B) TRANSFORMATION*

- Résolution des types (variables et fonctions) utilisés dans les appels Kooc
- Résolution des appels grâce aux types récupérés précédemment
- Ajout des déclarations des modules dans l'AST
- Ajout des implémentations (modules et classes) dans l'AST

# VII. Description des unités logicielles

## *A) CLASSE "MODULE"*

Cette classe va nous servir à stocker les variables et prototypes déclarés dans un module. Un objet "Module" sera instancié à chaque "@module" rencontré.

La classe contient :

- Nom du module (String)
- Déclarations (Dictionnaire : String -> Decl)

## *B) CLASSE "CLASS"*

Cette classe va nous servir à stocker les différents composants d'une classe. Un objet "Class" sera instancié à chaque "@class" rencontré.

La classe contient :

- Nom de la classe (String)
- Typedef de la "vtable" (Decl)
- Instance de la "vtable" (Decl)
- Membres (Dictionnaire : String -> Decl)
- Fonctions virtuelles (Dictionnaire : String -> Decl)
- Déclarations (Dictionnaire : String -> Decl)

## *C) CLASSE "IMPLEMENTATION"*

Cette classe a pour but de sauvegarder les différentes implémentation pour un module ou une classe. Un objet "Implémentation" sera instancié à chaque "@Implémentation" rencontré.

La classe contient :

- Nom de la classe ou du module (String)
- Implémentations (Dictionnaire : String -> Decl)

- Réassignation de pointeurs dans la "vtable"  (liste de Decl)

## *D) CLASSE "MANGLER"*

Cette classe a pour but de décorer les symboles qu'elle reçoit. Cette classe est un "singleton" car elle n'a besoin d'être instancie qu'une fois.

La classe contient trois méthodes utilitaires :

- muckFangle : cette fonction prend en paramètre un objet "Decl" et un nom de module/classe. Elle va décorér le nom de l'objet « Decl » selon les règles citéés ci-dessous.
- mimpleSangle : Cette fonction prends en paramètre un objet « Decl ». Elle va décorer le nom de l'objet « Decl » mais à la différence de « muckFangle », elle ne tiendra pas compte de :
  - Nom des classes ou modules
  - Variable ou fonction
  - Type du premier paramètre (paramètre « self »)

## *E)* **CLASSE** *"TYPEUR"*

Cette classe a pour but de résoudre les types des variables et fonctions dans les appels Kooc. Un objet «Typeur» sera instancié à la fin du parsing. Il va faire une boucle sur tous les membres de l'AST pour retrouver les objets « KoocCall ». Après avoir résolu les types, il appellera un objet « ResolveCall » qui lui renverra la « Decl » à ajouter dans l'AST.

## *F) CLASSE « KOOCCALL »*

Cette classe va sauvegarder tous les paramètres d'un appel Kooc. Elle contiendra donc :

- Nom du module (String)
- Nom du membre appelé (String)
- Paramètres (Liste)

## *G) CLASSE « RESOLVECALL »*

Cette classe va servir à résoudre les appels après avoir récupéré les informations de types. Cette classe va être un singleton. Elle appellera le « Mangler » pour récupérer les symboles décorés.

## *H) CLASSE « DECLKEEPER »*

Cette classe va servir à stocker toutes les déclarations et elle va créer les différents typedef de la « struct Object » et de la « vtable Object », elle contient :

- Identifiants rencontrés dans les fichiers importés (Liste)
- Modules déclarés (Dict : String -> Module)
- Implémentations (Dict : String -> Implémentation)
- Classes (Dict : String -> Class)
- Liens d'héritage (Dict : String -> String)
- Typedef de la « vtable Object » (Cnorm.nodes.Decl)
- Instanciation de la « vtable Object » (Cnorm.nodes.Decl)

## *I) CLASSE « KOOC »*

Cette classe va effectuer le parsing du fichier, elle contient :

- La grammaire (syntaxe à parser)
- Une liste de « hooks » qui sont appelés lors des différents appels Kooc rencontrés.

# VIII. Conclusion

## *A) CONTRAINTES DE CONCEPTION*

Les symboles générés par le KOOC doivent être lisibles par des humains pour un souci de clarté.

## *B) CONTRAINTES DE DEVELOPPEMENT*

Ce programme doit obligatoirement être écrit en langage Python puisqu'il repose sur l'utilisation de la librairie «Pyrser» et de l'outil «Cnorm».

# IX. Annexes

## *I) IMPORT*

Ce code « Kooc » :

```
@import "file.kh"
```

Donnera en sortie :

```
#include "file.kh"
```

## *II) MODULE*

Ce code « Kooc » :

```
@module A
{
  int   i = 0;

  void  f();
  int   f(char, float);
}
```

Donnera en sortie :

```
// Si le fichier est un ".kh"
#ifndef FICHIER_PARSE_H
#define FICHIER_PARSE_H

int     Var$A$i$$int = 0;
void    Func$A$f$$void();
int     Func$A$f$$int$$char$$float(char, float);

#endif
```

## *III) CLASSE*

Ce code « Kooc » :

```
@class A
{
  char c = '0';
  @member int i;
  @virtual void get_void();
}
```

Donnera en sortie :

```
i le fichier est un ".kh"
def FICHIER_PARSE_H
ine FICHIER_PARSE_H

def struct _kc_A    A;
def struct _kc_vt_A vt_A;
ct _kc_A

ject parent;
t Var$A$i$$int;

ct _kc_vt_A

 Champs de la Vtable parent (Object)
id (*clean$P$void)(Object *);
t (*isKindOf$P$int)(Object *, const char *);
t (*isKindOf2$P$int)(Object *, Object *);
t (*isInstanceOf$P$int)(Object *, const char *);
t (*isInstanceOf2$P$int)(Object *, Object *);

 Champs ajoutés
id (*get_void$P$void)(A*);


 Var$A$c$char = '0';
 vtable_A = { &Func$Object$clean$$void$P$Object, &Func$Object$isKindOf$$int$P$Object$P$char, &Func$Object$isKindOf$$int$P$Object$P$Object,
ct$isInstanceOf$$int$P$Object$P$char, &Func$Object$isInstanceOf$$int$P$Object$P$Object, &Func$A$get_void$$void$P$A };
```

## *IV) IMPLEMENTATION*

Cet exemple reprend la classe A exemple de l'annexe III.

Ce code « Kooc » :

```
@implementation A
{
  void get_void()
  {
  }
}
```

Donnera en sortie :

```
// Le paramètre self est ajouté comme get_void
// est une fonction virtuelle
void Func$A$get_void$$void$P$A(A *self)
{
}
```

## *V) HERITAGE*

Cet exemple reprend la classe A exemple de l'annexe III.

Ce code « Kooc » :

```
@class B : A
{
  @virtual void get_void();
}
```

Donnera en sortie :

```
i le fichier est un ".kh"
def FICHIER_PARSE_H
ine FICHIER_PARSE_H

def struct _kc_B    B;
def struct _kc_vt_B vt_B;

ct _kc_B

parent;


ct _kc_vt_B

 Champs de la Vtable parent (A)
id (*clean$P$void)(Object *);
t (*isKindOf$P$int)(Object *, const char *);
t (*isKindOf2$P$int)(Object *, Object *);
t (*isInstanceOf$P$int)(Object *, const char *);
t (*isInstanceOf2$P$int)(Object *, Object *);
id (*get_void$P$void)(A*);


 vtable_B = { &Func$Object$clean$$void$P$Object, &Func$Object$isKindOf$$int$P$Object$P$char, &Func$Object$isKindOf$$int$P$Object$P$Object, &Fun
ct$isInstanceOf$$int$P$Object$P$char, &Func$Object$isInstanceOf$$int$P$Object$P$Object, &Func$B$get_void$$void$P$B };

if
```

## *VI) DIAGRAMME DE CLASSES*

Le diagramme de classes du projet se trouve dans le fichier « diagramme_kooc.png ».