# Kind Of Objective C

Abstract : Le projet KOOC a pour objectif de concevoir une surcharge objet au langage C en créant un préprocesseur permettant de transformer ce langage, en C compilable et exécutable.
Laissez-vous pousser la barbe !

Victor Aubineau (Développeur)
Pierre Barrios (Développeur)
Jean Grizet (Développeur)
Guillaume Le-Monies-De-Sagazan (Développeur)
Alexandre Payen (Chef de projet)

EPITECH - Promotion 2016

# Historique des modifications :

| Date | Auteur | Résumé |
|---|---|---|
| 20/09/13 | grizet_j | Ajout de la partie inclusion. |
| 21/09/13 | grizet_j | Ajout de la partie module et implémentation |
| 24/09/13 | grizet_j | Ajout de la partie Appels KOOC |
| 27/09/13 | barrio_p | Mise en page et clarification |
| 27/09/13 | grizet_j | Ajout de la partie ADT |
| 04/10/13 | aubine_v | Complément sur l'introduction |
| 09/10/13 | payen_a | Ajout de l'algorithme de typage et conception. |
| 10/10/13 | grizet_j | Ajout de la partie Héritage simple. |
| 12/10/13 | barrio_p | Ajout de la partie Exception. |
| 12/10/13 | grizet_j | Ajout de la partie Interfaces. |
| 12/10/13 | payen_a | Ajout de la partie héritage multiple. |
| 13/10/13 | grizet_j | Précision sur la conception explications schématiques. |
| 13/10/13 | aubine_v | Correction orthographique/syntaxique et mise en page |
| 18/10/13 | aubine_v | Mise à jour exemple @import |
| 25/10/13 | aubine_v | Modification du fonctionnement @import |
| 10/11/13 | aubine_v | Mise à jour des schémas |

# Sommaire

# Introduction

Le KOOC (Kind Of Objective C) est un projet réalisé en 3e année. Celui-ci a pour but de créer un préprocesseur pour le langage KOOC devant générer du code C de façon à lui rajouter certaines fonctionnalités. Les problématiques abordées dans ce projet sont celles rencontrées par les créateurs des langages-objets.
Voici un schéma décrivant l'utilisation de KOOC, à partir du code KOOC (extension de fichier .kc et .kh) jusqu'à l'obtention d'un exécutable :


## Dépendance logicielle

Le projet étant à réaliser en python v3.3, voici les paquets mis à notre disposition:

- ***Pyrser*** :

Pyrser est un paquet qui permet de décrire rapidement une grammaire visant à analyser quelque chose. Pyrser utilise la syntaxe BNF pour décrire des règles de grammaire.

https://code.google.com/p/pyrser/

- ***Cnorm*** :

Cnorm est un paquet qui utilise Pyrser pour analyser la syntaxe du C. On notera la compatibilité avec c89, c99, gnuC, et Ms Visual C.

https://code.google.com/p/cnorm/

## Fonctionnement général

Nous allons surcharger tous les types CNorm connus (Declaration, Statment, Expression et Litteral) avec nos propres types.
Nous pourrons donc rajouter des règles BNF précisément dans la classe que nous souhaitons surcharger.
Par exemple si nous souhaitons surcharger la BNF de la classe Litteral nous allons écrire les règles dans KoocLit.
Nous allons ensuite créer des classes correspondant à nos types Kooc. Lors de l'analyse syntaxique, on stockera des références sur ces types dans une liste.
L'on fait ensuite un traitement sur cette liste, pour transformer la représentation du texte kooc vers une représentation transformable en C.
Ensuite, chacune de ces classi)es aura une méthode « to_c » qui sera appelée automatiquement lors de la génération du code C.

## Remarques diverses

Pour faciliter au maximum la compréhension, certains exemples dans ce document sont écrits sans décoration de symbole.

# 1 - Inclusion avancée

Cette nouvelle fonctionnalité a pour but de créer un mécanisme d'inclusion plus simple et plus intelligent, avec une gestion automatique de la double inclusion et des modules.

## A - Syntaxe

La syntaxe choisie pour ce nouveau mécanisme est l'utilisation du mot-clé « *@import* ».
Celui-ci peut être combiné à l'utilisation du mot-clé « *from* ».

Exemples **:**

```
@import « exemple.kh »                  /* ici tout le fichier header est inclus */
@import « module » from « exemple.kh »  /* ici seul un module est inclus depuis le fichier header */
@import « garbage » from « exemple.kh » /* ici seules les données hors des modules sont incluses*/
```

## B - Fonctionnement

En utilisant la compilation conditionnelle du C, il est possible de définir des blocs au sein d'un fichier d'en-tête. Ainsi chaque module dans un fichier « *kh* » sera transformé en un bloc dans le ficher « *h* » correspondant. Le code n'appartenant à aucun module sera rassemblé dans un bloc nommé systématiquement « garbage ».
L'inclusion de chaque bloc est soumise à une condition de la forme suivante:
#if defined(__FILE_NAME_[MODULE_*MODULE_NAME*/GARBAGE]__) || defined(__FILE_NAME_*FILE*__).
La 1ere partie valide l'inclusion du module en question (ou du garbage du fichier), la seconde permet l'inclusion totale du fichier.

La protection contre la double inclusion du C est automatiquement ajoutée à l'intérieur des blocs.

## C - Code généré

Prenons comme exemple un fichier « *exemple.kh* » avec un unique module « *moduleA* » et une déclaration de fonction. Le fichier « *exemple.h* » généré contiendra donc deux blocs, le bloc général (contenant le prototype de la fonction) et le bloc du module.

Fichier « *exemple.kh* » :

```
@module A
{
        int a;
        void    module_func(double, double);
}
void    function_sample();
```

Fichier « *exemple.h* » généré :

```
#if defined(__EXEMPLE_GARBAGE__) || defined(__EXEMPLE_FILE__)    /* Inclusion du garbage*/
        #ifndef __EXEMPLE_GARBAGE__    /* Protection du bloc contre la double inclusion*/
                #define __EXEMPLE_GARBAGE__
                /* Code n'appartenant a aucun module */
                void function_sample();         /* La déclaration de la fonction */
        #endif /* Fin de protection de la double inclusion */
#endif   /* Fin de l'inclusion du garbage */

#if defined(__EXEMPLE_MODULE_A__) || defined(__EXEMPLE_FILE__)       /* Inclusion du module A */
        #ifndef __EXEMPLE_MODULE_A_INC_  /* Protection du module contre la double inclusion */
                #define __EXEMPLE_MODULE_A_INC_
                /* Code du A */
                extern int a;
                void     module_func(double, double);
        #endif   /* Fin de protection de la double inclusion*/
#endif   /* Fin d'inclusion du module A */
```

Les variables de modules sont déclarées en tant qu'« *extern* » dans le fichier « *h* », car celles-ci sont instanciées dans le fichier « *c* » correspondant. Explications ci-après.

Un fichier de sources « *exemple.kc* » qui utilise « *@import* » sera converti en code C avec une définition du bloc à inclure et une inclusion standard en C.

| | **Inclusion de tout le fichier** | **Inclusion d'un seul module** |
|---|---|---|
| **Fichier « *exemple.kc* »** | *@import « exemple.kh »* | *@import « A » from « exemple.kh »* |
| **Fichier « *exemple.c* »** | *#ifndef __EXEMPLE_FILE__*<br>*#define __EXEMPLE_FILE__*<br>*#include « exemple.h »*<br>*#endif* | *#ifndef __EXEMPLE_MODULE_A__*<br>*#define __EXEMPLE_MODULE_A__*<br>*#include « exemple.h »*<br>*#endif* |

# 2 - Module

La syntaxe du KOOC permet de définir des modules, que l'on peut voir comme un espace de nom.

## A - Syntaxe

Dans les fichiers d'en-tête « *.kh* », la déclaration d'un module se fait grâce au mot-clé « *@module* » suivi du nom du module en question. La description du module doit impérativement se trouver juste après cette déclaration.

Exemple :

```
@module ModuleName
{
        ModuleContent
}
```

## B - Fonctionnement

Lors du pré-processing d'un fichier d'en-tête, les déclarations des modules sont décorées. Ainsi chaque déclaration devient unique et rattachée au module. De plus, ce mécanisme permet de définir un symbole plusieurs fois et de le transformer en fonction de sa signature.

L'algorithme de décoration se base sur le type naturel (dans le cas d'une fonction, le type de retour) et le contexte pour les symboles définis dans un module.
Dans le cas d'une fonction, on inclut également les paramètres de cette dernière.
En plus du type naturel, on utilise les qualificateurs suivant pour l'algorithme : const, pointer, unsigned et signed.
Les décorations respectent la règle suivante :

```
MANGLING ::= [ 'V' | 'F' ] '_' "_M"?
          '_' NAME '_'
          '_' TYPE_VALUE
          '_' NAME
          [ '_' TYPE_VALUE ]*
          '_' [ 'V' | 'F' ]
;
TYPE_VALUE ::= ATTRIBUTE* TYPE+
;
TYPE ::= [ INT | VOID | DOUBLE | FLOAT | LONG | CHAR ] | MANGLING
;
ATTRIBUTE ::= [ SIGNED| UNSIGNED | POINTER | CONST ]
;
NAME ::= ![ [ 'V' | 'F' ] '_' ]  Base.id ![ '_' [ 'V' | 'F' ] ]
;
```

Avec les symboles suivants:

| Type | Symbole |
|---|---|
| Char | c |
| Int | i |
| Short | s |
| Short int | si |
| Long | l |
| Long int | li |
| Long long | ll |
| Long long int | lli |
| Float | f |
| Double | d |
| Long double | ld |

| Attribut | Symbole |
|---|---|
| Signed | S |
| Unsigned | U |
| Const | K |
| Pointeur | P |

Ce mécanisme nous apporte la possibilité de surcharger variables et fonctions dans un module, mais ce n'est pas sans restriction. Si on regarde de plus près la règle *NAME*, on remarque que les noms commençant par « F_ » ou « V_ » et terminant par « _F » ou « _V » sont bannis pour des raisons de conflit avec l'algorithme de mangling.

## C - Code

Pour un module A défini comme :

```
@module      A
 {
        int a = 42;
        double a;
        char **str;
        void moduleA();
}
```

Le code généré avec les décorations est le suivant :

```
extern int V__A_i_a__V;
extern double V__A_d_a__V;
extern char ** V__A_ppc_str__V;
void  F__A__v_moduleA_F();
```

# 3 - Implémentation

Pour chaque module déclaré dans un fichier « *.kh* », on fournit une implémentation dans le fichier « *.kc* » correspondant.

## A - Syntaxe

De la même manière que le module, l'implémentation est définie par « *@implementation* » et est délimitée par des accolades.

## B - Fonctionnement

Dans un premier temps, les initialisations que l'on trouve dans le fichier d'en-tête sont générées. Chaque symbole est décoré en suivant les mêmes règles de mangling.

## C - Code généré

Pour le module A, une implémentation possible est :

```
@implementation      A
{
    void moduleA()   { printf(« Je suis le module A »); }
}
```

Le fichier c généré sera donc :

```
int V__A__i_a_V = 42;
double V__A__d_a_V;
char ** V__A__ppc_str_V;
void  F__A__v_moduleA_F() { printf(« Je suis le module A »); }
```

# 4 - Appels KOOC

## A - Syntaxe

Les appels KOOC sont définis entre crochets.
L'accès aux variables se fait par l'utilisation du nom du module, d'un point et du symbole de la variable.
L'accès aux fonctions se fait par l'utilisation du nom du module, d'un espace, du nom de la fonction et de la liste des paramètres ou chaque paramètre est précédé d'un double point.
Dans le cas d'une surcharge (de nom de variable ou de fonction), où plusieurs symboles pourraient convenir (dans le cas où il y aurait un conflit de type), on peut explicitement spécifier le type à utiliser en préfixant l'appel KOOC par « !@(type) ».

## B - Fonctionnement

L'on résout les polymorphismes et inférences de type suivant l'algorithme suivant sauf si spécifié explicitement par la syntaxe de type vu précédemment. Si la syntaxe « !@(type ) « est utilisée, on considère que le type donné est correct, et ne pratiquons pas de résolution sur le paramètre ou la variable.
Pour trouver le type d'une variable: nous allons inspecter le module et dresser un tableau de tous les types possibles pour ce nom de variable.
Pour trouver le type de retour d'une fonction: nous allons inspecter les différents paramètres de la fonction et dresser un tableau de tous les types de paramètres.
Si une fonction prend des variables de module, il faudra croiser le tableau des types de cette variable avec le tableau des types du module.
Par élimination successive, on résout le type de chaque expression, paramètre et type de retour.

Exemple:

```
@module A
{
   int a;
   double a;
   float b;
   int f(int, double);
   string f(int, float);
   double f(double, float);
};

[A.a] = [A f :A.a :A.b]
```

tab1 : int, double (gauche du signe '=')
tab2 : int, string, double

String n'est pas dans le tableau de gauche, nous l'excluons. Descente récursive:

tab1 : int, double
tab2 : float

En remontant et en croisant les types, la seule solution à cette équation est:
*[A.a](double) = (double)[A f :A.a(double) :A.b(float)]*

## C - Code généré

On remplace l'appel KOOC par le symbole décoré.

Soit un module A défini comme :

```
@module A
{
        int a;
        int b;
        char b;
}
```

Un exemple KOOC du type :

```
printf(« L'int vaut %d\n », [A.a]);
printf(« Le char vaut %c\n », !@(char)[A.b]);
```

Donnera une fois transformé :

```
printf(« L'int vaut %d\n », V__A__i_a_V);
printf(« Le char vaut %c\n », V__A__c_b_V);
```

# 5 - Abstract Data Type (ADT)

On ajoute la possibilité de créer un type de données abstrait, instanciable : l'Abstract Data Type. L'ADT se comporte comme un module avec la différence que les variables dites « *membres* » dépendent de l'instance et non plus du contexte global. De plus, ce type abstrait peut posséder des fonctions « *membres* » qui vont agir directement sur l'état interne de l'instance de cet ADT.

## A - Syntaxe

Pour définir un ADT on utilise le mot-clé « @class » suivi d'un bloc d'accolades.
Pour les déclarations appartenant à l'ADT on utilise le mot-clé « @member ». Ce dernier concerne une ligne unique sans accolades, ou un bloc complet limité par des accolades.
A l'intérieur d'une fonction membre, on accède aux variables internes de l'instance grâce à « self ».

## B - Fonctionnement

Pour obtenir des variables membres différentes dans chaque instance d'une ADT, une structure C contenant ces variables (décorées) est créée lors du pre-processing. On transforme également les fonctions membres en ajoutant en première position un pointeur sur la structure associée. Dans le contexte de la fonction, ce paramètre sera toujours nommé « this » et permet l'accès aux variables membres.

## C - Instanciation

Pour créer une instance d'ADT on introduit les syntaxes :
[ADT alloc], [ADT init], [ADT new], [ADT clean], [ADT delete]
On ajoute dans chaque module d'ADT les fonctions « alloc », « init », « new », « clean », « delete » :
- « alloc » est une fonction non membre qui alloue et retourne la structure contenant les variables membres.
- « init » est une fonction membre redéfinissable qui initialise les variables membres.
- « new » est une fonction non membre qui créer une nouvelle instance puis l'initialise avant de la retourner. (« alloc » puis « init »)
- « clean » est une fonction membre facultative pour libérer le tas si besoin est.
- « delete » est une fonction non membre qui tente d'appeler « clean » avant de libérer l'espace mémoire de l'instance.

Les fonctions « clean » et « delete » sont contenues dans une section spéciale appelée « live table » et ce pour éviter les conflits en prévision du mécanisme d'héritage.

## D - Appel

La syntaxe d'appel ressemble à celle du module avec quelques modifications :
- L'on peut appeler une fonction membre en utilisant une instance d'une ADT.
- L'on peut appeler une fonction membre en utilisant le nom de l'ADT et en passant en premier paramètre une instance.

## E - Code généré

L'implémentation suivante :

```
/* FILE.kh */
@class Foo
{
    @member
    {
        int a;
        void print();
    }
}
/* END FILE.kh */
/* FILE.kc*/
@implementation Foo
{
   @member void print(self)
    {
        printf(« A = %d\n », [self->a]);
    }
}
/* END FILE.kc */
```

Après pré-processing, la classe Foo sera représentée par :

```
/* FILE.h */
typedef struct s_Foo
{
    int a;
} t_Foo;
/* END FILE.h */
/* FILE.c */
void print(t_Foo *this)
{
    printf(« A = %d\n », this->a);
}

t_Foo  *alloc()
{
    return (malloc(sizeof(t_Foo)));
}
/* END FILE.c */
```

# 6 - Héritage simple

On ajoute à l'ADT la possibilité d'hériter d'un ADT existant. C'est-à-dire que cet ADT acquiert les variables membres et les fonctions de l'ADT duquel il hérite.

## A - Syntaxe

Pour ce faire, après la déclaration d'un ADT on peut ajouter une relation d'héritage en précisant le nom d'un ADT père.

Par exemple, un ADT « Son » héritant d'un ADT « Father » sera déclaré comme suit :

```
@class Son : Father
{
   /* Définition de l'ADT */
}
```

On autorise également la redéfinition des fonctions membres à travers l'héritage. C'est le polymorphisme d'héritage que l'on déclare en utilisant le mot-clé « @virtual ». Ce dernier appliqué à une fonction la rend automatiquement membre et redéfinissable. Si la fonction virtuelle n'est pas redéfinie, la fonction parente sera appelée avec son type réel, c'est-à-dire, le type parent.

Par exemple, dans la classe « Father » une fonction « foo » qui sera redéfinie dans « Son » sera déclarée :

```
@class Father
{
 @virtual void foo();
 /* OU */
@virtual
    {
       void foo();
    }
}
```

## B - Héritage des variables

Afin d'inclure les variables de la classe parente, on ajoute en premier dans la structure des variables membres de la classe fille la structure de la classe parente. Ce mécanisme s'appelle agrégation de structures.

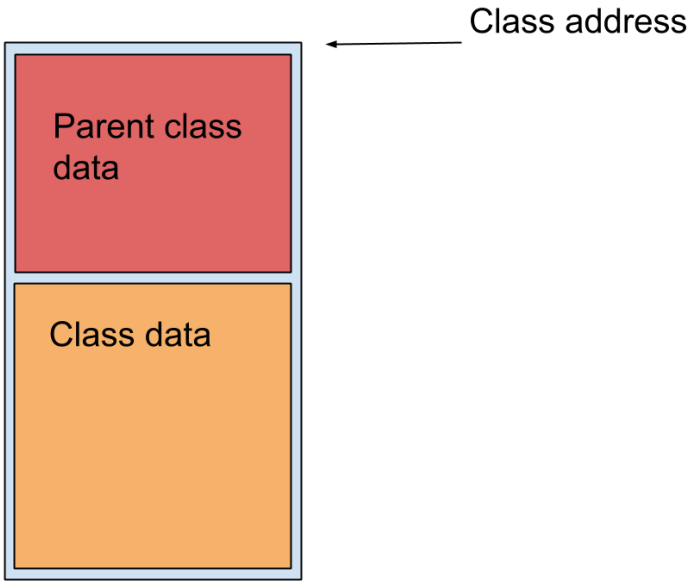


Figure 1 : Agrégation de structures

## C - Polymorphisme d'héritage

Pour assurer l'intégrité du cycle de vie de la classe fille et répondre aux attentes en matière de polymorphisme d'héritage, on ajoute, dans les ADT qui n'héritent pas, un pointeur. C'est un pointeur sur une structure contenant des pointeurs sur fonctions.
Ainsi, chaque ADT possède bien une sauvegarde des fonctions qu'il utilise. Si lors de l'exécution l'ADT est utilisé comme son type parent les fonctions redéfinies seront bien appelées.
On en profite pour ajouter automatiquement l'attribut @virtual à « delete », permettant de libérer la totalité de l'espace mémoire lors de l'appel. Ceci même si le type a été perdu.
On alloue et initialise ces structures au moment de l'allocation (voir syntaxe « New »), avec des pointeurs sur les fonctions redéfinies.
Cette table de pointeurs sur fonction est un mécanisme de « dispatch table » ou encore « vTable ».

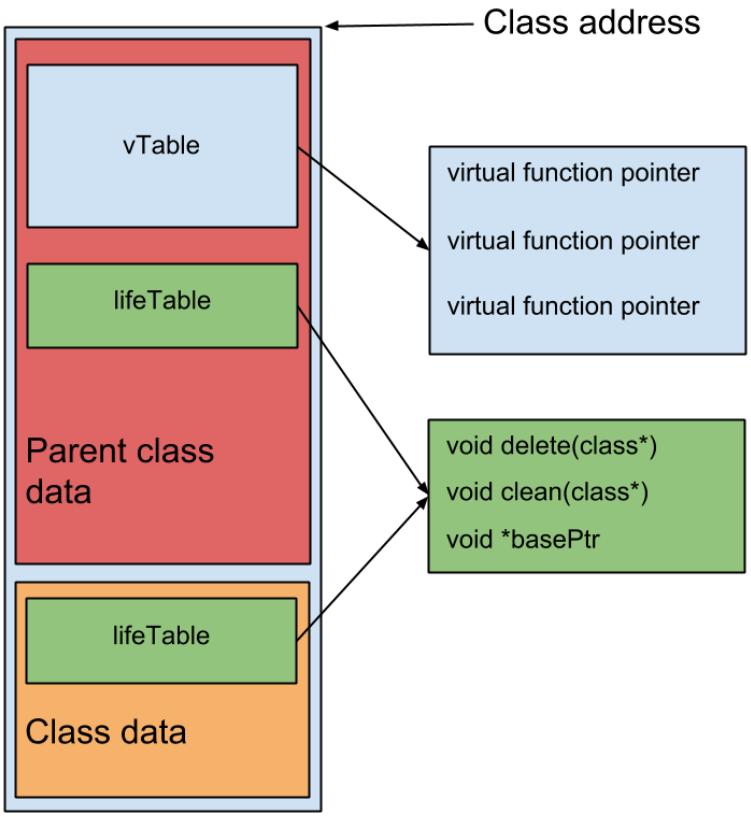


Figure 2 : Représentation de la dispatch table

## D - Code des données

Soit une classe « Foo » héritant d'une classe « Bar » :
La classe Bar étant représentée par la structure suivante :

```
typedef struct s_bar
{
   int __V_i_my_int_V;
} t_bar;
```

On fera l'agrégation de structure de la manière suivante :

```
typedef struct s_foo
{
    t_bar    parent;
   int        __V_i_my_second_int_V;
} t_foo;
```

## E - Code des fonctions virtuelles

On ajoute la sauvegarde des fonctions. Avec une fonction virtuelle « func » redéfinie dans « Foo », la représentation de « Bar » devient :

```
typedef struct s_bar
{
        void *func_table;
        void  func();
} t_bar;
```

Et on génère la sauvegarde de « Bar » :

```
typedef struct s_bar_table
{
   (void*) (delete(t_bar *));
   (void *)(func(void));
} t_bar_table;
```

Au moment d'allouer les structures de données, en fonction du niveau d'héritage on utilise le bon type de sauvegarde, initialisé avec les pointeurs sur fonctions qui correspondent à ce niveau.

Soit pour la classe « Bar » :

```
t_bar_table *bar_table = malloc(sizeof(bar_table));
bar_table->delete = &F__Bar_v_delete_v;    /* Fonction delete de Bar*/
bar_table->func = &F__Bar_v_func_v;    /* Fonction func de Bar*/
obj->func_table = (void *)bar_table;
```

Et pour la classe « Foo » :

```
t_foo_table *foo_table = malloc(sizeof(foo_table));
foo_table->delete = &F__Foo_v_delete_v;    /* Fonction delete de Foo */
foo_table->func = &F__Foo_v_func_v;     /* Fonction func de Foo */
obj->func_table = (void*)foo_table
```

# 7 - Exception

Le KOOC va implémenter son propre système d'exceptions. Ainsi, à la manière de beaucoup de langages comme C++, il sera possible de créer ses types d'exceptions et de les utiliser pour gérer les erreurs. Pour profiter pleinement de ce mécanisme, quelques prérequis seront nécessaires :

- Toutes les exceptions utilisateurs devront hériter directement de la classe KOOC '' *Exception* ''
- Les fichiers '' *Exception.kc* '' ainsi que '' *Exception.kh* '' devront être traités par KOOC et ajoutés aux futurs projets
- Le fichier implémentant la fonctions '' *main()* '' ( exemple '' *main.kc* '' ) devra impérativement être aussi traité par KOOC pour générer un fichier '' *main.c* ''

## A - Syntaxe

Les exceptions sont déclarées comme de simples classes héritant d' '' *Exception* ''. Celles-ci auront donc par cet héritage une variable membre de type pointeur sur « *char* » (représentant un message expliquant l'erreur de l'exception "lancée"), ainsi qu'une fonction membre d'initialisation « *init* », servant à initialiser le pointeur membre à la valeur de celui passé en paramètre. De plus, un identifiant non membre représentant l'id de l'exception (pour pouvoir y accéder simplement à partir du module) sera rajouté lors du traitement d'une exception utilisateur.

Exemple :

```
/* FILE MyException.kh */
@class MyException : Exception
{
   /*
   **    int id;  (un identifiant sera rajouté lors du traitement)
   */
}
/* END FILE MyException.kh */



/* FILE MyException.kc */
@implementation MyException
{

}
/* END FILE MyException.kc */
```

Le code C généré sera le même que celui d'une classe standard KOOC, à la différence que notre KOOC devra absolument assigner à chaque exception de l'utilisateur un id différent et positif non nul.

Ensuite, le code à tester devra être placé dans un bloc "*@try {}*", et les exceptions devront être attrapées (ou "catchées" dans le jargon) dans un des blocs "*@catch () {}*" suivant directement le bloc précédent, ou bien dans l'un des blocs parents.
Les exceptions sont envoyées avec le mot-clé "*@throw*" suivi du type de l'exception à envoyer ainsi que de son message.

La syntaxe est la suivante :

```
@try
{
   /* Code à tester */
   /* Une exception peut être envoyée */
   if (fatal_error == true)
      @throw (MyException *)("A fatal error occured !");

   if (non_fatal_error == true)
      @throw (MyOtherException *)("An error occured, but the program can continue !");

}
@catch (MyException *ex)
{
   printf("%s\n", !@(char *)[ex->msg]);
   exit(1);
}
@catch (MyOtherException *ex)
{
   printf("%s\n", !@(char *)[ex->msg]);
   return (42);
}
```

## B - Fonctionnement global

Pour parvenir à créer cette abstraction d'exceptions en C, nous allons utiliser les puissants outils que composent le couple de fonctions "*int setjmp(jmp_buf env)*" et "*void longjmp(jmp_buf env, int val)*".
La première permet de sauvegarder un contexte d'exécution, et la suivante permet de le restaurer.

En d'autres termes, lors de l'appel de la fonction "*longjmp()*", l'exécution du programme reprend au dernier appel de "*setjmp()*" avec le paramètre de type "*jmpbuf*" correspondant. La valeur de retour de la fonction "*setjmp()*" sera alors différente de 0, et correspondra à la valeur passée en second paramètre de la fonction "*longjmp()*". La pile ainsi que l'environnement seront restaurés.

Nous aurons besoin de deux variables globales. Celles-ci devront être incluses dans chaque fichier où les exceptions seront utilisées ( elle seront déclarés en '' *extern* '' dans '' *Exception.kh* '' ) :
- La première est "*jmp_buf kooc_global_env*" qui nous servira à sauvegarder le contexte précédent dans chaque bloc "*@try {}*".
- Le second est un pointeur sur l'instance d'une exception, "Exception *kooc_global_exception". Elle sera initialisée à NULL et pointera vers l'exception qui vient d'être lancée lors d'un "*@throw*".

## C - Préparation du main()

La fonction ''*main*'' est préparée pour accueillir le système d'exception. En premier lieu, des inclusions y sont rajoutées. Ensuite, les deux variables globales nécessaires sont instanciées. Et enfin, le contexte est sauvegardé une première fois de façon à pouvoir "*catcher*" une exception qui ne l'aurait pas été durant l'exécution, et ainsi pouvoir terminer le programme avec une erreur.

Le code du ''*main*'' sera modifié comme suit :

```
#include <stdlib.h>    /* Pour avoir accès à EXIT_SUCCESS et EXIT_FAILURE */
#include <stdio.h>     /* Pour avoir accès à la fonction fprintf() */
#include <setjmp.h>    /* Pour avoir accès aux fonctions setjmp() et longjmp() */

/* Les deux globales sont instanciées avant le main() */
jmp_buf    kooc_global_env;
Exception    *kooc_global_exception = NULL;

int    main(int ac, char **av)
{
   /* Le contexte est sauvegardé */
   if (!(setjmp(kooc_global_env)))
   {
      /* Le code original du "main" original est juste recopié dans ce bloc */
   }
   else
   {
      /*
      ** Si l'on rentre dans ce bloc, c'est qu'une exception n'a pas été "catchée"
      ** On sort du programme avec une erreur
      */
      fprintf(stderr, "Process terminated by an unknown exception\n");
      return (EXIT_FAILURE);
   }
   return (EXIT_SUCCESS);
}
```

## D - Le bloc try

Lors d'un bloc "*@try {}*", le contexte général contenu dans la variable global "*kooc_global_env*" est copié dans une variable temporaire locale à la fonction ( grâce à "*memcpy()*" ), et le contexte courant est alors sauvegardé dans la variable globale (grâce à la fonction "*setjmp()*" ).

Ainsi le code KOOC suivant :

```
@try
{
   /* Some code */
}
```

Donnera le code C suivant :

```
/* On crée une variable de contexte locale */
jmp_buf    kooc_local_env;

/* Le contexte global est copié dans la variable locale */
memcpy(&kooc_local_env, &kooc_global_env, sizeof(jmp_buf));

/* La variable retour de setjmp() est instanciée */
int    kooc_local_ret;

/* Le contexte général courant est sauvegardé dans la variable global */
if (!(kooc_local_ret = setjmp(kooc_global_env)))
{

   /* Some code */

}
```

## E - Le throw

Tout d'abord, lorsqu'un "*@throw*" est effectué, une instance correspondant à l'exception est créée, et le pointeur global d'exception "*kooc_global_exception*" doit alors pointer sur celle-ci.

Ensuite, un "*longjmp()*" sur le dernier contexte sauvegardé en C sera effectué (celui contenu dans la variable global "*kooc_global_env*" ). De cette manière, le cours de l'exécution du programme reprendra au dernier appel de la fonction "*setjmp()*" avec le contexte correspondant (à l'endroit du "@try"). Cette fois-ci, la fonction ne renverra pas 0 mais une valeur que nous définirons, passée en second paramètre de "*longjmp()*" et qui sera l'identifiant de l'exception "lancée" (accessible depuis le module).

Code KOOC :

```
@throw (MyException *)("A fatal error occured !");
```

Code C (avec les décorations) :

```
/* L'exception est instanciée avec le message */
kooc_global_exception = MyException_ new ("A fatal error occured !");

/* On restaure le contexte, et on envoie l'identifiant de l'exception */
longjmp(kooc_global_env, V__MyException_i_id__V);
```

## F - Le bloc catch

Lorsqu'un contexte est restauré (appel de "*longjmp()*"), l'exécution reprend à la fonction "*setjmp()*". Sa valeur de retour, désormais différente de zéro, correspond à l'identifiant d'un type utilisateur d'exception : elle doit donc être testée. Pour ce faire, le contexte précédent est alors restauré (il avait été sauvegardé dans une variable locale), et la valeur de retour est comparée à tous les identifiants des exceptions pour lesquels un bloc "*@catch () {}*" a été créé. De ce fait, si une correspondance est trouvée, cela veut dire que l'exception a été interceptée ("*catchée*"), sinon cela veut dire qu'elle ne l'a pas été. De ce fait, la fonction "longjmp()" sera donc de nouveau appelée avec le contexte courant, ce qui ramènera l'exécution du programme soit dans un bloc "*@try {}*" parent pour peut-être y "*catcher*" l'exception, ou bien dans le ''*main*'', à la première sauvegarde vue précédemment. Le programme se terminera donc avec une erreur.

Soit le code KOOC suivant :

```
@catch (MyException *ex)
{
    printf("%s\n", [ex->msg]);
    exit(1);
}
@catch (MyOtherException *ex)
{
    printf("%s\n", [ex->msg]);
    return (42);
}
```

Donnera le code C suivant :

```
/* Bloc if () {} précédent */
else
{
    /* On restitue le contexte précédent */
    memcpy(&kooc_global_env, &kooc_local_env, sizeof(jmp_buff));

    /* Les différents cas sont testés */
    if (kooc_local_ret == [MyException.id])
    {
        MyException *ex = kooc_global_exception;
        printf("%s\n", !@(char *)[ex->msg]);
        exit(1);
    }
   else if (kooc_local_ret == [MyOtherException.id])
    {
        MyOtherException *ex = kooc_global_exception;
        printf("%s\n", !@(char *)[ex->msg]);
        return (42);
    }
    else
    {
        /* L'exception n'a pas été "catchée" par l'utilisateur, on retourne dans le "main" */
        longjmp(kooc_global_env, kooc_local_ret);
    }
}
```

## G - La destruction des ADT

En programmation, lorsqu'une variable est créée dans un bloc, elle est empilée sur la stack ( le pile ). Lorsque l'exécution arrive à la fin de ce bloc, la pile temporaire est détruite, et toutes les variables s'y trouvant sont alors perdues. Les ADT étant des types personnalisés, la destruction de leur partie interne est donc plus complexe et requiert l'appel de leur destructeur.

Pour chaque ADT créé sur la pile, il faudra les déclarer avec l'attribut "*cleanup(&destruct)*" où "*void destruct(void *ptr)*" est la fonction destructrice de cet ADT.

Exemple :

```
MyADT    adt __attribute__((cleanup(&destruct)));
```

## H - La différenciation des exceptions

Pour différencier les exceptions, nous allons utiliser leur identifiant non membre, accessible directement à partir du module de l'exception. Un problème se pose alors, ces variables étant transformées en globales dans l'implémentation du module (cf. 3 – Implémentation), comment allons-nous pouvoir initialiser les différents identifiants de manière unique, sans avoir de valeurs identiques, sachant que les fichiers implémentant les exceptions peuvent être traités par KOOC séparément ?

Pour résoudre ce problème, nous allons utiliser l'attribut C « constructor ». Celui-ci, utilisé lors de la déclaration d'une fonction, permet de spécifier que celle-ci sera exécutée avant le « main() » principal du programme.

Exemple :

```
#include    <stdio.h>

void    func(void)    /* Déclaration de la function func() */
{
   Print("func()\n");
}

Void    func(void) __attribute__((constructor));    /* On assigne l'attribut à func() */

int    main(int ac, char **av)    /* Déclaration de la fonction "main" */
{
   printf("main()\n");
   return (0);
}
```

L'exécution de ce code affichera cette sortie :

```
func()
main()
```

Ensuite, nous allons créer une variable globale juste au-dessus de la fonction « main() », qui sera le compteur d'exceptions KOOC et qui sera instancié comme suit :

```
int    KOOC_EXCEPTION_COUNT = 1;
```

Enfin, pour chaque type d'exception utilisateur, nous allons créer une fonction d'initialisation d'identifiant, déclarée avec l'attribut « constructor ». Ces fonctions auront seulement pour but de copier la valeur du compteur global d'exception dans l'identifiant correspondant, puis, puis d'incrémenter ce compteur, qui servira pour le type d'exception suivante, et ainsi de suite.

**my_exception.h :**

```
extern int    KOOC_EXCEPTION_COUNT;

extern int    V__MY_EXCEPTION_i_id__V;

void    MY_EXCEPTION_INIT_ID(void) __attribute__((constructor));
```

**my_exception.c :**

```
int    V__MY_EXCEPTION_i_id__V;

void    MY_EXCEPTION_INIT_ID(void)
{
   V__MY_EXCEPTION_i_id__V = KOOC_EXCEPTION_COUNT;
   KOOC_EXCEPTION_COUNT++;
}
```

# 8 - Héritage multiple

L'héritage multiple reposera sur la même base que l'héritage simple, mais l'on décrétera qu'il y a dans les classes parentes, une classe principale qui sera la première de la liste et des classes secondaires.
La classe fille héritera de cette classe de la même manière qu'un héritage simple. L'on agrégera par la suite les classes parentes secondaires, ce qui donnera en mémoire cette représentation:

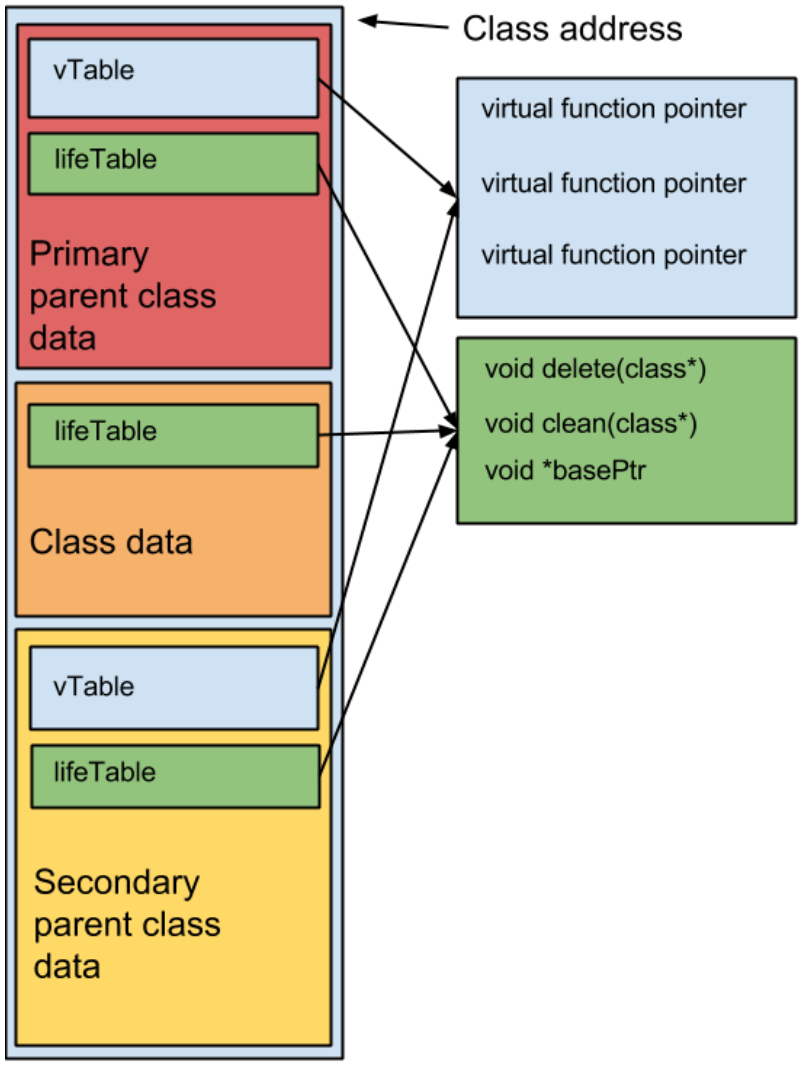


**Figure 3 : Agrégation héritage multiple**

## A - Syntaxe

La syntaxe de l'héritage multiple est la même que celle de l'héritage simple à ceci près que l'on sépare les différentes classes parentes par des virgules (',')

Exemple Kooc:

```
@class F1 { /* F1 class definition */ };
@class F2 { /* F2 class definition */ };
@class S : F1, F2 { /* S class definition */ }
```

Traduction C:

```
typedef struct  s_vtable
{
  /* virtual function pointer */
}          t_vtable;

typedef struct  s_F1
{
  t_vtable    *_vtable
  /* F1 class variables */
}          t_F1;

typedef struct  s_F2
{
  t_vtable    *_vtable
  /* F2 class variables */
}          t_F2;

typedef struct  s_S
{
  t_F1       _f1_vars;
  /* FS class variables */
  t_F2       _f2_vars;
}          t_S;
```

## B - Virtual Table (Dispatch Table)

La gestion de la vTable ne peut plus se faire de la même manière que dans le cas d'un héritage simple. Un problème réside dans la vTable des classes pères de la seconde branche, les index sont bons mais le pointeur sur l'instance n'est pas valable lui. Il va falloir décaler ce pointeur directement dans le registre sans faire de modification de code. Nous allons utiliser ce qui s'appelle des « Thunk Function ». Ces fonctions seront des petits bouts d'assembleur qui ne feront qu'incrémenter (ou décrémenter) le pointeur situé dans la pile et d'exécuter la bonne fonction.

Exemple:

```
[S f :p] == s->vtable[idx](p)
vtable[idx] == _thunk_f_
_thunk_f_ == asm(mov offset esp; jmp f;)
```

## C - Héritage en Diamant[1]

Lorsque deux classes parentes héritent de la même classe, on parle d'héritage en diamant. C'est un cas particulier de l'héritage multiple.
Cette classe n'est pas forcement instanciée de la même manière à chaque fois. Pour conserver une abstraction valable, l'on choisira de gérer cela de la manière suivante :
La classe commune sera instanciée deux fois, par ses classes filles. Pour interagir avec les variables et méthodes, on modifie la syntaxe de l'appel Kooc : on peut désormais préciser quelle classe parente possède l'attribut que l'on souhaite utiliser. Toutefois, cela reste optionnel, la branche primaire sera utilisée par défaut. L'appel prend désormais la forme suivante :

```
[Object.class.var]; /* .class est optionnel */
```

Exemple:

```
[Foo.bar] = 1;  /* Toutes les valeurs bar seront modifiées. */
[Foo.CLASS.bar] = 3; /* Seule bar de la classe sera modifiée. */
printf(« %d\n », [Foo.bar]);  /* Ici 1  car c'est la valeur primaire. Son pointeur se confond avec celui de Foo */
```

[1] Cette fonctionnalité n'a pas été implémentée par manque de temps

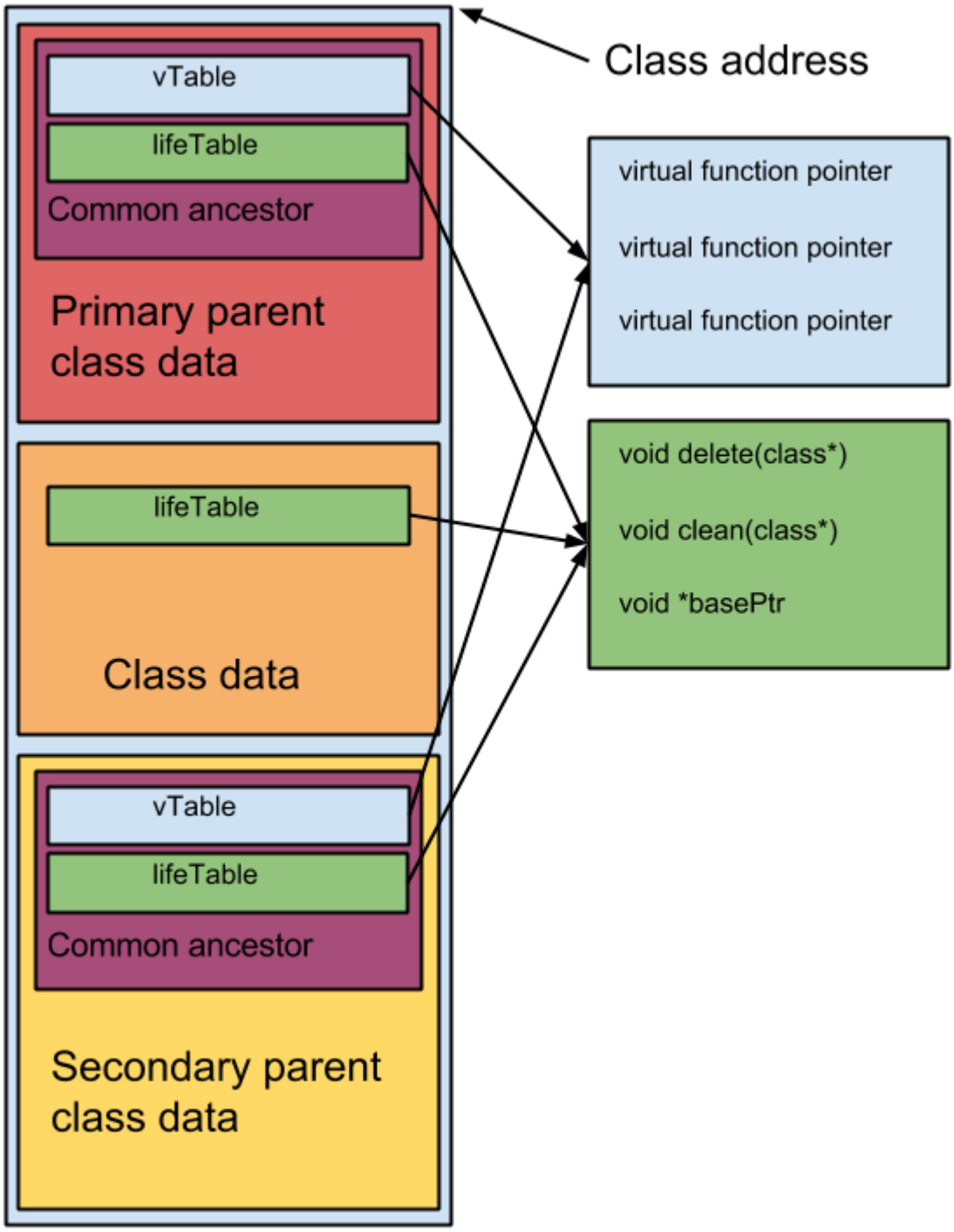


Figure 4 : Gestion de l'héritage en diamant

# 9 - Interface[2]

Pour obtenir un niveau d'abstraction supplémentaire, on implémente les interfaces qui permettent de donner une signature à une classe. Le but est de pouvoir manipuler l'interface implémentée par plusieurs classes de manière générique.

## A - Syntaxe

Pour déclarer une interface, on utilise le mot-clé *« @interface »* et des accolades pour le bloc de déclarations.
Une interface ne possède pas de variables, pas d'implémentation et par défaut toutes les fonctions seront virtuelles. C'est pourquoi le *« @virtual »* est optionnel.

Exemple :

```
@interface  interface_foo
{
    @virtual int my_function(int);
    int my_other_function(int);
}
```

Pour utiliser la signature d'une interface, une classe doit simplement en hériter.

## B - Fonctionnement

Lorsqu'une classe hérite d'une interface, on vérifie qu'elle complète bien la signature de l'interface.
L'interface ne possédant aucune variable, il n'y a pas agrégation de structures : l'extension de la vTable suffit. Puis on assigne l'implémentation de la signature dans la vTable.

[2] Cette fonctionnalité n'a pas été implémentée par manque de temps.

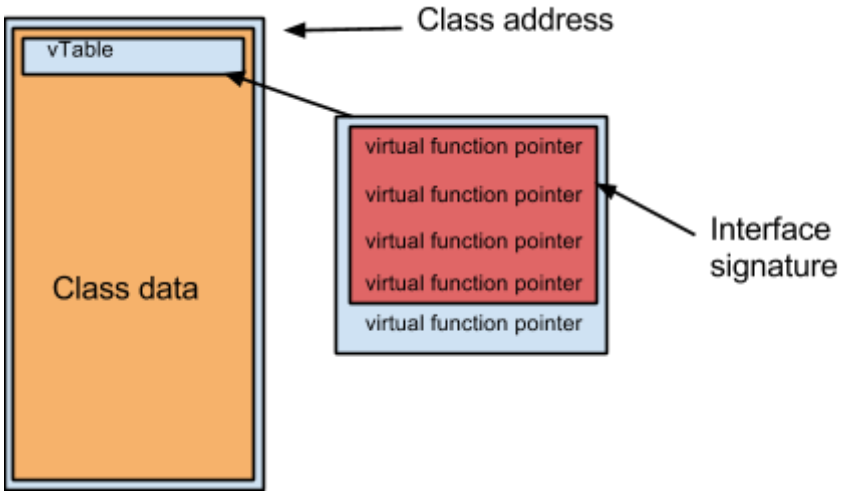


Figure 5 : Signature d'interface en mémoire

## C - Le délégué d'interface

Le délégué d'interface va servir à manipuler l'objet de manière générique en utilisant l'interface.
On ajoute la fonction membre qui sert à obtenir un délégué d'interface : *« createInterface() »*

On génère pour chaque interface un nouveau type, que l'on initialise depuis un objet. Il est composé de la partie de la vTable de l'objet qui correspond à l'interface et une référence sur l'objet de base.
Cela permet de manipuler l'objet via son interface.

Par exemple, le délégué d'interface issue d'un objet Bar qui implémente une interface Foo sera de la forme :

```
typedef struct s_interface_foo
{
   void *vt;
   void *obj;
}   interfaceFoo;
```

Avec les variables initialisées en fonction de l'objet Bar :

```
Bar          *barObj = [new Bar]        /* Creation d'un objet Bar, qui implémente Foo*/
interfaceFoo *iFoo = [barObj createInterface];      /* Recupération d'une déléguée  d'interface Foo */

/*  Initialisation dans createInterface */
iFoo->vt = bar->vt[index  of interface vtable];
iFoo -> obj = barObj;
/*  Fin d'initialisation   */
```