# KOOC - 2ième Partie

## Sommaire



## 1) Passage à l'objet.

Nous avons partiellement résolu les problèmes de modularité. Le module n'est pas un mot-clef récursif. Nous ne pouvons pas faire de module inclus dans un autre. Toutefois, cela reste fonctionnel, mais...

### a) @module est-il suffisant?

Nous avons maintenant à disposition un nouvel outil d'abstraction 'le module'. Il répond aux mêmes fonctionnalités qu'une bibliothèque de fonctions. Il est toutefois plus pratique et plus puissant à utiliser car il garantit l'unicité des symboles.

Voyons maintenant si cela répond à nos exigences.

Il y a 2 grands types de bibliothèques :

- Les bibliothèques qui permettent de définir de nouvelles structures de données.
- Les bibliothèques qui sont un patchwork d'algorithmes s'appliquant à un domaine particulier plus qu'à un type particulier.

Nous allons étudier dans un premier temps les bibliothèques qui tentent de définir des nouveaux types de données.

Prenons le cas d'un module permettant de gérer une pile. Nous pouvons le définir comme suit :

```
------- fichier Stack.kh
@module Stack
{
#define MAX_SIZE  4096
      static int        stack_int[MAX_SIZE];
      static double     stack_double[MAX_SIZE];
      static char       *stack_char[MAX_SIZE];
      static int        nbitem;

      int               nbitem();
      void              push(int);
      void              push(double);
      void              push(char *);
      int               pop();
      double            pop();
      char              *pop();
}
```

Voici donc un module de pile. L'accès à la pile se fait par 2 types de fonctions '**push**' et '**pop**'. Grâce à la surcharge de fonctions, nous pouvons gérer notre pile dans tous les types que nous souhaitons.

Concrètement la pile est un tableau déclaré en '**static**' pour en empêcher l'accès de l'extérieur. Comme indiqué dans le sujet, les variables '**static**' ne sont pas manglées, donc elles peuvent collisionner. Nous sommes encore obligés de rajouter un suffixe.

Cependant, le module souffre encore d'un défaut majeur, les variables qu'il héberge sont uniques. Nous ne pouvons pas encore créer de pile à volonté. En programmation classique, ce genre de situation se règle en utilisant une structure de données allouées dynamiquement et qui héberge les données spécifiques à notre pile.

```
------- fichier Stack2.kh
@module Stack2
{
      typedef enum stack_type_e
      {
      INT,
      STRING,
      DOUBLE
      }           stack_type_t;

      int   nbstack = 0; /* pour compter le nombre de pile presente
```

```
                dans le système */

      typedef struct stack_s
      {
            stack_type_t type; /* type de pile */
            int nbitem;
            union
            {
                  int   *stack_int;
                  double *stack_double;
                  char  **stack_char;
            }     data;
      }           stack_t;

      stack_t     *alloc();
      stack_t     *init(stack_t *, int, stack_type_t);
      stack_t     *new(int, stack_type_t);
      void            clean(stack_t *);
          void              delete(stack_t *);

      int   nbitem(stack_t *);
      void  push(stack_t *, int);
      void  push(stack_t *, double);
      void  push(stack_t *, char *);
      int   pop(stack_t *);
      double      pop(stack_t *);
      char  *pop(stack_t *);
}

------ fichier Stack2.kc
/** @implementation Stack2 ****/
stack_t           *alloc()
{
      stack_t     *stack;

      stack = (stack_t *) calloc(1, sizeof (stack_t));
      if (!stack)
      {
            fprintf(stdout, "can't alloc\n");
            exit(1);
      }
      return (stack);
}

stack_t           *init(stack_t     *stack, int size, stack_type_t type)
{
      stack->type = type;
      stack->nbitem = 0;
      switch(stack->type)
      {
      case INT:
      stack->data.stack_int = (int *) calloc(size, sizeof (int));
            break;
      case STRING:
      stack->data.stack_char = (char**) calloc(size, sizeof (char*));
```

```
            break;
      case DOUBLE:
      stack->data.stack_double = (double *) calloc(size, sizeof (double));
            break;
      }
      return (stack);
}

stack_t           *new(int size, stack_type_t type)
{
      stack_t     *stack;
      int   nbst;

      stack = [Stack2 alloc];
      stack = [Stack2 init :stack :size :type];
      nbst = [Stack2.nbstack];
      nbst += 1;
      [Stack2.nbstack] = nbst;
      return (stack);
}

void        clean(stack_t *stack)
{
      free(stack->data.stack_int);
          stack->data.stack_int = 0;
}

void        delete(stack_t *stack)
{
          if (stack->data.stack_int)
                  clean(stack);
          free(stack);
}
...
int         nbitem(stack_t *stack)
{
      return (stack->nbitem);
}
...
void        push(stack_t *stack, double d)
{
      if (stack->type != DOUBLE)
      {
            fprintf(stdout, "wrong type\n");
            exit(1);
      }
      stack->data.stack_double[stack->nbitem++] = d;
}
...
```

Vous avez déjà beaucoup  codé ce type de bibliothèque. Pour KOOC, nous allons généraliser le principe. Nous pouvons noter que :

- Nous pouvons classer les données d'une librairie en 2 grandes familles :
    - Les données qui participent au nouveau type de données. Typiquement celles qui sont créées dynamiquement. Elles représentent l'état du nouveau type de données. Ce sont les "variables membres" d'un échantillon du type (une instance du type, cf. synonyme).

      ex: la structure **stack_t**

    - Les données indépendantes du nouveau type de données créés. Ces données sont rattachées au module en général. Ce sont les "variables non-membres". Souvent utiles à des fins de gestions internes.

      ex: **nbstack**

- Nous pouvons classer les fonctions d'une librairie de ce type en 2 grandes familles :
    - Les fonctions qui s'appliquent à la structure de données définissant le type. Ces fonctions prennent une instance du type de données en paramètre et agissent dessus ou à travers elle. Ces fonctions sont communément appelées "fonctions membres".

      ex: **push**, **pop**, **init**, **clean**, **delete** et **nbitem**.

    - Les fonctions qui ne rentrent pas dans l'ensemble précédent. Ces fonctions sont indépendantes de l'état de tel ou telle instance. Ce sont les "fonctions non-membres".

      ex: **new**, **alloc**

Note : Dans le type pile présenté ici, nous dissocions la partie allocation de mémoire de la partie initialisation des différents champs de la structure. En effet, l'allocation est un morceau de code quasi générique qui ne dépend que du '**sizeof**' de la structure à allouer. Alors que l'initialisation de l'objet alloué dépend des champs de l'objet et du rôle de la fonction **init**.

## b) Problème de la spécialisation

Nous sommes bien capables grâce à @module d'écrire des bibliothèques qui définissent de nouveaux types de données. Cependant, si nous reprenons le cas de notre pile, nous sommes obligés de modifier notre définition et notre implémentation à chaque fois que l'on voudra ajouter un nouveau type de pile.

De plus, notre type **enum** va naturellement grossir à chaque ajout.

Un autre problème vient du fait que je peux appeler une fonction push ou pop sur le mauvais type de données.

N'y a-t-il pas une technique qui nous permettrait de ne plus avoir à gérer le type à la main ?

Nous pourrions créer un module par type de pile. Par contre pour chaque module, nous devrions recoder la fonction 'nbitem'.

Pour cela, il nous faut un outil plus puissant que @module, capable d'associer un type de données à un ensemble de fonctions, mais aussi capable de gérer le cycle de vie (allocation, désallocation) de la structure associée.

### Exigence #7 Type abstrait

**Nous arrivons donc à la notion de "Type abstrait". Un "Type abstrait" est un type de données utilisateur se comportant comme un type primitif (int, char, double). Nous commençons notre étude des types abstraits en rajoutant le mot clef @class.**

# 2) LE CHOIX DE C++

Pour répondre, à cette nouvelle exigence, C++ modifie le comportement initial de **struct**, introduit le mot-clef **class**, invente l'opérateur **new** et **delete** et enfin la variable **this**. On verra finalement que pour implémenter une notion complète d'un type abstrait, C++ autorise la surcharge d'opérateur.

## a) Exigences logicielles

### Exigence #7 Type abstrait

Comme indiqué en définition, un type abstrait se comporte comme un type primitif. C++ modifie donc la sémantique du mot-clef **struct**.

Pour simplifier les choses disons que **class** est similaire à **struct.** Elle tient juste compte de l'exigence #2 concernant l'encapsulation.

La modification sémantique de **struct** permet dans un premier temps de contrôler le cycle de vie du type abstrait, de son allocation à sa destruction.

En C, une **struct** se comporte presque comme un type primitif (int), pour les allocations sur la pile. Pas sur le tas. Le tas est une notion système (implémentation de la libc de malloc). En C++, c'est une notion du langage grâce à **new** et **delete**. De plus, en C, il y a une structure lexicale appelée '**initializer_list**' qui permet d'initialiser les données sur la pile, non sur le tas.

Le C++, au contraire, introduit la notion de constructeur et destructeur qui uniformise l'initialisation des champs de la structure sur le tas ou sur la pile.

Exemple:

```
// en C
struct A {int a; double f;};
struct B {const char *p1; void *p2;};

struct A var = {42, 4.2}; // initializer list
void *v = malloc(...);
struct B v2 = {"tutu", v};
// en C++
struct A {
      int   _a;
      double      _f;
      A(int a, double f) : _a(a), _f(f) {}
};
A     var(42, 4.2);

struct B {
      const char  *_p1;
      void        *_p2;
      B(const char *p1, void *p2) : _p1(p1), _p2(p2) {}
      ~B() {if (_p2) delete _p2;}
};
void  *v = (void *) new ...;
B     v2("tutu", v);
```

On peut en C++, surcharger les opérateurs associés au type abstrait. Un exemple de base est de voir la définition de la forme de Coplien. La forme de Coplien est un bon moyen pour être sûr que la nouvelle **struct** (ou **class**) se comporte comme un type primitif.

```
struct coplien
{
      coplien();//constructeur par defaut
      coplien(const coplien &);//constructeur par copie. exemple:
      //    coplien a; coplien b(a);
      coplien &operator=(const coplien &);//operateur d'affectation. exemple:
      //    coplien a, b; b = a;
      ~coplien();//destructeur
};
```

Finalement on peut ajouter de nouveaux traitements au type abstrait. Les méthodes sont associées au type abstrait. On peut à l'intérieur d'une méthode, accéder aux données membres par l'utilisation du pointeur **this**. L'appel de la méthode se fait uniquement grâce au type de l'objet. Même si on pense accéder à une méthode grâce au dé-référencement de **this**, en fait l'adresse de la méthode est toujours connue. **this** est juste automatiquement son premier paramètre.

```
struct C
{
      int   _a;

      C(int a) : _a(a) {}

      void printThis()
      {
            std::cout << "THIS:" << this << std::endl;
      }

      void printA()
      {
            std::cout << "A:" << this->_a << std::endl;
      }
};

C     &rC = *((C*) 0); // on fabrique une reference null
rC.printThis(); // appel C::printThis(); affiche un THIS: null
rC.printA(); // appel A::printA(); derefence un this null, segfault.
```

## 3) NOTRE CHOIX

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

## a) Exigences logicielles:

### Exigence #7 Type abstrait

Nous ajoutons un mot-clef **@class**. Celui-ci introduit le début de la définition d'un type abstrait. **@class** est une évolution du **@module**, et permet d'utiliser **@member** pour délimiter les fonctions et variables membres. A l'inverse de C++ qui modifie la sémantique de **static** pour signifier les parties non-membres, KOOC procède différemment.

Les parties non-membres existent par défaut, **@member** est un cas particulier pour les parties membres.

La sémantique du C étant conservée, la somme des parties membres constitue une **struct** qui peut être allouée automatiquement sur la pile par ce genre d'écriture.

```
@class A
{
      @member {int a; double f;}
      void  init(A *);
      @member void      init(int a, double f);
}

A     d;// d allouer sur la pile pour contenir un A
```

Cependant, la phase d'initialisation est faite par les fonctions membre '**init**'. Les fonctions membres sont des fonctions de modules qui prennent en premier paramètre un pointeur sur la structure des membres.

```
A     c;
A     d;
[A    init :&c];
[A    init :&d :42 :4.2];
```

On ajoute une facilité d'écriture similaire au C++. On peut passer par un pointeur du type du **@class** correspondant pour appeler la méthode avec la gestion automatique du premier paramètre.

C'est un appel qui génère le même code C que précédemment, mais qui s'écrit de manière plus concise.

```
A     c;
A     d;
[&c   init];
[&d   init :42 :4.2];
```

Au lieu de rajouter de nouveaux opérateurs pour l'instanciation sur le tas, nous créons automatiquement des fonctions non-membres pour allouer l'objet. Nous créons automatiquement une fonction non-membre **new** par fonction **init** définie.

```
@class A
{
      @member
      {
            int a; double f;
            void  init();
      }
      @member void      init(int a, double f);
}
A     *pc;
A     *pd;
pc = [A     new];// appel @!(void)[A init :(A*)];
pd = [A new :42 :4.2];// appel @!(void)[A init :(A*) :(int) :(double)];
```

Vis à vis du C++, il manque cependant la surcharge globale des opérateurs pour que cela soit un type abstrait complet. Cela pourrait être défini et spécifié mais cela ne constitue pas une partie obligatoire pour le projet.