KOOC - 1ère Partie

Sommaire

1) Critique du C

Pour faire une critique honnête du C, il faut d'abord fixer des critères d'évaluation avec lesquels on peut jauger un langage. Les critères suivants sont souvent exposés comme étant des fonctionnalités des langages objets. Ceci est dû à l'évolution. Originellement, c'est en cherchant à combler ce genre de défaut que la POO (programmation orientée objet) a vue le jour.

En voici une liste non-exhaustive :

a) Modularité

Liberté de découper un programme en brique fonctionnelle (ensemble de fonctionnalités précises), en brique logique, etc.

b) Encapsulation

Besoin de masquer les détails et de gérer l'accès aux ressources d'un module : qui accède à quoi ? Partie publique (visible par des tiers) ou privée des modules. Visibilité des symboles au sein d'une unité de compilation.

c) Abstraction

Séparation entre ce qu'on appelle l'interface (ce qu'on doit fournir comme service) et l'implémentation (le komenkonfai) du module.

d) Réutilisabilité

N'avoir que le code spécifique à rajouter (spécialisation). Appliqué un algorithme unique à n'importe quel type de donnée (généricité).

e) Type utilisateur

Manipuler un type utilisateur comme un type primitif. Type primitif aka. les types de bases fournit par le langage (int, char, etc.)

f) Gestion des erreurs

Plus sensible. Que les erreurs d'exécution d'une fonction ne passent plus inaperçues! Plus de détails. Où l'erreur s'est-elle produite?

g) Gestion de la mémoire/ressources systèmes :

Comment gérer plus facilement mes ressources:

- mémoire,
- fichiers,
- sockets,
- connexions aux bases de données,
- lock sur variables ou zones de mémoire,
- etc.

de manière automatique ou à la main.

h) Aide au débogage

Obtenir des informations en temps réel sur les données que l'on manipule. Réflexivité.

2) Premiers problèmes

Attaquons notre refonte du C par le plus facile, les problèmes de modularités et d'encapsulation. La manière naturelle de créer un programme modulaire en C est de créer des bibliothèques de fonctions.

La création d'une librairie répond-elle aux besoins de modularité et d'encapsulation ?

Pour répondre, nous devons faire de petite révision de compilation en C.

Les headers

En C, un header liste toutes les fonctions et variables globales que met à disposition une bibliothèque. On peut dire que ce fichier décrit son interface.

Cependant, nous pouvons aussi créer des headers contenant des fonctions qui n'ont aucun intérêt pour l'utilisateur de cette bibliothèque.

Ces headers peuvent avoir un usage purement interne. Ce n'est pas une fonctionnalité du langage dont-il s'agit, c'est juste une pratique de programmation.

Il pourrait être utile de faire la différence entre un header à usage interne et un header listant les fonctions vraiment utilisables de l'extérieur.

De plus, en C aucun mécanisme n’assure de protection contre l'inclusion multiple.

Nous devons le faire à la main par un ennuyeux :

Ex :

```
#ifndef NOM_DE_MA_LIB
#define NOM_DE_MA_LIB
....
....le reste du .h
....
#endif
```

Exigence #1 Inclusion

Ne serait-il pas plus simple de disposer d'une fonction équivalente à #include mais plus évoluée et qui permettrait d'inclure un header et qui règlerait le problème de l'inclusion multiple automatiquement ?

3) Liberté de design

a) Petite révision sur la compilation en C

Qu'est ce que compiler ?

Tout programme en C est un fichier texte avec une extension qui par convention est '.c'. Quand on compile ce '.c' nous obtenons un substrat binaire '.o' contenant une partie du code assemblé (en langage machine) et une partie d'information pour l'éditeur de liens (aka linker, ld).

Quel est le rôle du linker ?

Quand nous écrivons un programme, nous décrivons des variables, des fonctions et leurs interactions. Vulgairement nous utilisons ces variables et ces fonctions. À la fin de la compilation, tout doit finir en langage machine. Le langage machine ne connaît que des opcodes (actions) qui manipulent de la mémoire et font des opérations mathématiques minimales. En toute fin du processus complexe appelé compilation, toute variable et toute fonction doit avoir une place en mémoire. Pour le compilateur la différence entre une variable et une fonction importe peu. Pour lui, ce ne sont que des zones de mémoire dont il doit connaître le nom et l'emplacement. Ce sont les symboles du fichier binaire.

Commande nm

Cette commande permet d'afficher tous les symboles d'un fichier binaire. L'information importante à retenir après lecture du man est qu'il n'y a que 2 types de symboles. Les symboles locaux et les symboles globaux.

Si les symboles sont connus à la génération du '.o' le compilateur n'aura aucun problème à leur assigner une adresse. Quid des symboles n'étant pas dans votre source? Le compilateur va faire un pari ! Il ne connaît pas leur adresse mais estime que ce sont des ressources qui vont exister une fois le programme lié, il va donc laisser des informations dans le '.o' pour indiquer au linker qu'il faut que celui-ci résolve l'adresse pour tel ou tel symbole. Un symbole dont l'adresse reste un mystère dans un '.o' doit forcement exister (être défini/instancié) dans un autre.

Variables et fonctions

Prenons l'exemple suivant.

Ex :

```
/*
      Fichier a.c
*/

int         gl_var1 = 42;
```

```
int         main(int ac, char **av)
{
      printf("%d\n", gl_var1);      /* affiche 42 */
      fex2();                       /* ?? voir plus bas */
      printf("%d\n", gl_var1);      /* ?? voir plus bas */
}
------------
/*
      Fichier b.c
*/
void  fex2();
{
      extern int gl_var1;

      printf("%d\n", ++gl_var1);    /* affiche 43 */
}
```

Si nous compilons chaque fichier indépendamment nous obtenons 'a.o' et 'b.o'. Inspectons ces fichiers avec 'nm', et nous verrons apparaître dans 'a.o' un 'D gl_var1' et dans 'b.o' un 'U gl_var1'. Si nous consultons le man de nm :

```
...
"D" Le symbole est dans la section des données initialisées.
...
"U" Le symbole n'est pas défini.
...
```

Le symbole est bien défini dans 'a.o' et non défini dans 'b.o'. La phase d'édition des liens va consister à résoudre tous les symboles U. À la fin, tous les symboles doivent avoir une adresse. Il ne peut y avoir 2 symboles de même nom. Quand cela se produit il y a 'collision' et une erreur s'affiche. L'utilisation du mot clé 'extern' suffit en C pour accéder à une variable d'une autre source. Ou plus exactement cela dit au compilateur "t'inquiète pas cela existe quelque part..." On peut aussi remarquer dans 'a.o' :

```
............
... U fex2
............
... U printf
............
```

Que cela soit une fonction ou une variable l'information n'est pas connue au niveau du fichier '.o' quand le symbole est indéfini. Attention, ceci n'est pas vrai quand le symbole est connu. Regardez attentivement les différences. Une fonction sera T alors qu'une variable sera principalement B ou D.

b) Conséquence de ceci...

2 fonctions ne peuvent pas avoir le même nom.

En effet, la définition d'une fonction est un symbole global. Et il ne peut y avoir 2 symboles globaux de même nom.

Ex :

```
int   myfunc(int a, char b) {...}
void  myfunc(char *str) {...}
```

provoque une erreur. Cela limite le nombre de symboles disponibles pour la création d'une bibliothèque de fonctions. Imaginez une Stack de pile d'entiers qui fournit 2 fonctions : push et pop.

```
void  push(int);
int   pop();
```

Par exemple, vous ne pourrez pas utiliser la bibliothèque List qui gère une liste et qui permet de manipuler cette liste comme une pile en définissant aussi 2 fonctions : push et pop.

2 variables globales ne peuvent pas avoir le même nom

Idem que pour une fonction. Une variable globale est un symbole global. De plus, la déclaration via 'extern' de la variable va permettre d'y accéder de n'importe quel autre fichier source. Ce qui n'est pas forcément voulu.

Rappel : toute variable déclarée dans un bloc n'est visible que dans ce bloc et dans les blocs fils.

Ex :

```
void fex1()
{
      int var1;
      int var2;

      var1 = 42;
      var2 = 24;
      {
            int var2;

            var2 = 104;
            printf("%d %d\n", var1, var2); /* affiche: 42 104 */
      }
      printf("%d %d\n", var1, var2); /* affiche 42 24 */
}
```

Cette tricherie est gérée par le compilateur et n'a aucun rapport avec les symboles, car comme tout le monde le sait (enfin j'espère !) les variables locales sont allouées

sur la pile... Comment ça ! vous ne savez pas !... Ne vous inquiétez pas vous aurez tôt fait de maîtriser ce sujet ! Mais les variables déclarées dans un bloc sont-elles toutes sur la pile ? ;)

Variables statiques et fonctions statiques

Ex :

```
static char var1 = 64;

static int *fex3()
{
      static int var1 = 42;
      return (&var1);
}

static int *fex4()
{
      static int var1 = 128;
      return (&var1);
}
```

Où est allouée var1 ? Ce type de symbole est-il géré par l'éditeur de liens ? Au chargement de votre programme, il se trouve que cette variable existe sans jamais avoir appelé fex3. Regardons avec nm :

```
............
... d var1
... d var1.0
... d var1.1
............
```

Plusieurs remarques :

- Le compilateur a trouvé une manière de tricher pour différencier les 3 symboles var1.

  L'ajout du .0 ou .1 est judicieux, car on ne peut pas écrire un identifiant en C qui contient un '.'. Ces symboles sont donc protégés par cette nomenclature.

  En C, vous ne pouvez accéder à une variable statique à partir d'une autre fonction ! C'est la 'cuisine' du compilateur qui vous en empêche. Techniquement, vous pouvez retourner l'adresse de var1 pour y accéder d'une autre fonction. Peut-être commencez-vous à comprendre le nom du projet : KOOC ?

- La lettre du type de symbole est en minuscule.

  Par rapport à la documentation de nm, cela nous informe que c'est un symbole local. C'est donc un symbole qui ne colisionnera pas pendant l'édition de liens.

Après toutes ces révisions, nous pouvons constater que le C de par sa nature nous restreint dans le choix du nom de nos fonctions.

On ne peut pas assurer qu'un symbole issu d'une librairie ne colisionnera pas avec le symbole d'une autre librairie.

Pour pallier ce problème, bon nombre de librairies préfixent tous les symboles par le nom de la librairie.

C'est une solution intéressante, mais souvent fastidieuse à mettre en oeuvre surtout quand nous voulons publier une librairie qui est à l'origine à usage interne.

Exigence #2 Scope de décoration de symbole

Nous devons disposer d'une nouvelle structure syntaxique capable de définir des modules logiciels au sein desquels les collisions entre symboles sont évitées.

Un autre fait marquant, est l'accès aux variables globales. Il est pratique d'utiliser telle ou telle variable globale à des fins internes ou externes. Toutefois, nous ne pouvons restreindre l'accès qui y sera fait, car ce sont des symboles globaux. De plus, toute variable globale doit être initialisée dans un fichier '.c' du module alors qu'elle sera déclarée comme 'extern' dans le '.h' du module. Nous n'avons pas connaissance de cette initialisation et potentiellement de son absence et de la valeur par défaut. Cela peut être une source d'erreur.

Exigence #3 Initialisations globales

Nous devons disposer d'un moyen unique de déclaration de variable globale au sein d'un module logiciel et contrôler l'accès interne ou externe de cette ressource.

4) Premières solutions

Pour répondre à nos besoins (modularité et encapsulation) nous allons mettre en place une technique particulière appelée '**décoration de symbole**' (ou **symbol mangling** en anglais).

Cette technique est utilisée par nombre de langages objet (C++, java, objectiveC). Elle est très simple (comme quoi les solutions simples sont souvent les meilleures). C'est une ruse qui va consister à masquer par un encodage le nom réel du symbole. Ceci va nous permettre d'éviter toute collision de symbole sans obliger les développeurs à suivre une règle de programmation.

Avant de mettre en place cette solution, il faut faire attention aux points suivants:

Exigence #4 Compatibilité avec symbole C

Nous devons garder une certaine compatibilité avec le C.

Si nous ne pouvons pas utiliser des fonctions écrites en C, KOOC devient très peu utilisable.

Exigence #5 Règle de décoration des symboles

Notre technique d'encodage ne doit pas perturber le compilateur. Les symboles encodés sont des identifiants valides.

Notre technique de dissimulation est plus une ruse pour rajouter ensuite de nouvelles fonctionnalités qu'une règle stricte qui pourra être gênante plus tard.

Exigence #6 Préservation sémantique du C

Nous voulons aussi pouvoir différencier simplement du code C et du code KOOC. Rappelez-vous KOOC est un pré-processeur intelligent, pas un vrai compilateur.

Concept de signature

Outre les noms qui sont identiques. Quelle est la différence entre ces déclarations ?

```
int   toto;
int   toto();
void  toto();
void  toto(int);
```

Malgré le nom identique, cela décrit bien 4 choses totalement différentes. La différence est dans la nature de ce que ces éléments décrivent :

- Variable ou fonctions
- Type de variable
- Type de retour
- Paramètres (liste de type)

La signature d'un symbole est cet ensemble d'informations. Ceci rend le symbole unique. Cependant, nous ne pouvons définir un symbole sous forme de déclaration en C. Il va falloir trouver un encodage pour garder trace de toutes ces informations. Le 'symbol mangling' (système de décoration de symbole) est propre à chaque implémentation de langage. Par exemple, le compilateur C++ GNU utilise un 'symbol mangling' différent de Visual C++. Le java a un 'symbol mangling' très différent de C++, mais assez proche d'objective C.

5) LE CHOIX DE C++

Pour répondre, aux 6 premières exigences C++ introduit par rapport au C le mot clef **namespace**, rajoute l'opérateur **::**, et autorise un sucre syntaxique sur le mot clef extern : le **extern "C"**.

Exigences logicielles :

Exigence #1 Inclusion

Non traité. On fait comme en C.

Exigence #2 Scope de décoration de symboles

Un compilateur C++ décore tous les symboles par défaut sauf main.

Exemple :

```
---------- t1.cpp
int f1(int a)
{
  return 42;
}

int f2(double b)
{
  return 666;
}

static int f3(int galaxy, double express)
{
  return 999;
}

int gloupier = 100;

static double shadok = 3.4;

int main()
{
  static int a = 12;
  return 0;
}
-----------
$> g++ -c t1.cpp
$> nm t1.o
00000000 D gloupier
00000024 T main
00000000 T _Z2f1i
0000000a T _Z2f2d
00000004 d _ZZ4mainE1a
$> nm -C t1.o
00000000 D gloupier
00000024 T main
```

```
00000000 T f1(int)
0000000a T f2(double)
00000004 d main::a
```

NB : les fonctions **static** n'exportent plus de symbole, pas comme le C.

Ensuite, C++ introduit un nouveau mot clef **namespace** qui permet de créer des espaces de nom dans lesquels les symboles ont moins de chance de collisionner.

Exemples :

```
---------- t2.cpp
int   stevostin(int ork)
{
  return 1;
}

namespace GRUU
{
  int stevostin(int ork)
  {
    return 2;
  }
}

int   gobelin(void)
{
  return 3;
}

int   gobelin(int)
{
  return 4;
}

int   gobelin(double)
{
  return 5;
}
------------------
$> g++ -c t2.cpp
$> nm t2.o
00000028 T _Z7gobelind
0000001e T _Z7gobelini
00000014 T _Z7gobelinv
00000000 T _Z9stevostini
0000000a T _ZN4GRUU9stevostinEi
$> nm -C t2.o
00000028 T gobelin(double)
0000001e T gobelin(int)
00000014 T gobelin()
00000000 T stevostin(int)
0000000a T GRUU::stevostin(int)
```

A vous de déduire le comportement de C++ sur les signatures des fonctions, sa gestion de la surcharge de fonctions de ces exemples.

Exigence #3 Initialisations globales
Non traité

Exigence #4 Compatibilité avec symbole C
Comme un compilateur C++ décore systématiquement tous les symboles, c'est l'inverse qui est fait. On doit indiquer quand on veut bloquer la décoration de symbole sur un bloc en utilisant **extern "C"** devant une déclaration ou **extern "C" {...}** autour de N déclarations (voir macro __BEGIN_DECLS dans sys/cdefs.h).

Exigence #5 Règle de décoration des symboles
Pour un exemple voir C++ ABI

Exigence #6 Préservation sémantique du C
Altéré mais similaire, le mot clef **static** précédent n'est plus géré de la même façon en C++ qu'en C mais le principe fonctionnel général est compatible.

6) NOTRE CHOIX
ICI nous présentons succinctement les différents nouveaux mots clefs introduits dans KOOC par rapport aux exigences, le reste est vu en détails dans les TPs.

Exigences logicielles :

Exigence #1 Inclusion
Nous rajoutons un mot clef **@import**

Exigence #2 Scope de décoration de symbole
Nous introduisons les mot-clefs **@module** et **@implementation** qui permettent d'encadrer des blocs de déclarations et de définitions. Seuls ces blocs sont décorés. Nous rajoutons une nouvelle syntaxe d'expression **[Module function :p1 :p2 :p3 ...]** pour accéder aux éléments de ces modules. Le reste, c'est du C.
Exemple :

```
@module A
{
      int   a;
      double      a;
      char  *a;

      int   function(void);
      double      function(void);
      int   function(int);
}
```

```
@implementation A
{
      int   function(void)
      {
            return 5;
      }

      double      function(void)
      {
            return 4.2;
      }

      int   function(int a)
      {
            return a * [A function];
      }
}
```

Exigence #3 Initialisations globales

Dans la partie **@module** nous pouvons définir la valeur de nos variables de module. Au compilateur de faire que cela marche car la partie **@module** a pour vocation de se faire inclure via **@import**. Exemple:

```
@module B
{
      int   a = 42;
}
```

Exigence #4 Compatibilité avec symbole C

Tout ce qui n'est pas dans les nouveaux mot-clefs n'est pas décoré.

Exigence #5 Règle de décoration des symboles

Pour le projet KOOC vous aurez à mettre en place votre propre 'symbole mangling'. Vous pouvez vous inspirer de ce qui existe mais vous pouvez aussi INNOVER.

Un mangling naïf consisterait à concaténer à l'aide du caractère '_' toutes les informations utiles pour une signature.

- Les espaces de noms
- Le nom de votre symbole
- Le type de symboles (données, fonctions)
- Type de retour
- Liste de paramètres

Toutefois pour simplifier encore les choses, il faudrait restreindre l'utilisation du caractère '_' dans la constitution d'un identifiant. Cela éviterait le mélange des genres entre C et KOOC. Pensez à l'utilisation de extern "C" en C++.

Exigence #6 Préservation sémantique du C

Préservé strictement.

Jamais nous ne remettrons en cause ou altérerons un comportement d'une structure du langage C. Nous rajouterons seulement de nouvelles structures pour traiter de nouveaux concepts. Il y aura toutefois un pont entre ces nouvelles structures et le C. Il y aura toujours une méthode pour accéder aux nouvelles structures à partir du C. Par exemple, pour la décoration rien n'empêchera un développeur C qui connaît les règles de mangling de faire :

```
@module C
{
      int   f1(int a);
}
@implementation C
{
      int   f1(int a);
      {
            return a * 77;
      }
}
int   main()
{
      return _1C_2f1_3int_3int(14);// exemple de symbole decorer pour [C f1]
}
```

Ce projet est à but pédagogique, il est donc complètement ouvert.

NB : A titre informatif, l'exemple précédent avec les symboles peut être aussi fait en C++. Pensez à **extern "C"**. Avec une utilisation judicieuse de LD_PRELOAD cela permet de faire certaines choses amusantes. :)