TD1: Rappels de C, complexité et compilation

Exercice 1 – Code, indentation

Le style de codage n'est qu'une convention. Toutefois, lorsqu'elle est connue et suivie, cette convention permet d'améliorer sensiblement la lisibilité d'un code pour tous (et particulièrement pour les correcteurs). Nous vous demanderons dans ce module de suivre (au mieux) la convention K&R [1] dont voici un exemple parlant :

```
1  /* Voici un commentaire de description d'une fonction */
2  int une_function(void){
3      int x, y;
4      if (x == y) {
5          /* voici un commentaire local, indent'e comme le code */
6          quelquechose1();
7          quelquechose2();
8      } else {
9          autrechose1();
10         autrechose2();
11     }
12     chosefinale();
13     return une_valeur_retournee;
14 }
```

Remarquez :

- accolade ouvrante sur la même ligne que l'instruction de contrôle (voir L4 et L8) [2].
- indentation incrémentée à chaque accolade ouvrante et décrémentée à chaque accolade fermante.
- commentaires indentés comme le code (voir L5).
- utilisation de caractères basiques. En particulier, pas d'accent ni dans le code ni dans les commentaires (voir 5).

Q 1.1 Faites quelque chose avec cela :

```
1 #include <stdio.h>
2 int xy(int n){ if (n) return n*/*warum nicht ? */xy(n-1); else return 1;} void
3 main(void) { printf("%d", xy(5));}
```

Exercice 2 – Pointeurs et fonctions

Un pointeur est une variable contenant l'adresse d'une zone mémoire. Cette zone mémoire est interprétée comme une valeur d'un certain type, dépendant du type du pointeur. La syntaxe des pointeurs est la suivante :

- `*p` : accéder à la zone mémoire pointée par le pointeur `p` et typée d'après le type de `p`.

1. *Le Langage C ANSI*, Brian Kernighan et Dennis Ritchie, 1988
2. Petite exception : dans la convention K&R, l'accolade de début de fonction est placée sur la ligne suivante, mais on ne vous demande pas de respecter cette exception dans le cadre de cette UE.

— `&i` : récupérer l'adresse de la zone mémoire représentant une variable `i`.

L'utilisation la plus fréquente des pointeurs est la création dynamique de valeurs en mémoire (fonction `malloc`). L'erreur la plus fréquente des pointeurs est l'oubli de suppression des valeurs en mémoire créées dynamiquement (fonction `free`). L'exemple ci-dessous permet d'illustrer ces différentes notions :

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  void main(void){
5      int *p;
6      int i = 1;
7
8      p = &i; // recuperation de l'adresse d'un entier existant
9      printf("%d\n", *p);
10
11     p = (int *) malloc(sizeof(int)); // creation dynamique d'un entier
12     printf("%d\n", *p);
13
14     free(p); // nettoyage
15 }
```

Q 2.1 Que pensez-vous des lignes 11-12 ?

Q 2.2 Sachant que les fonctions C passent leurs arguments *par valeur* (recopie de la valeur dans une nouvelle zone mémoire), comment faire une fonction qui modifie la valeur de son argument ? Donner le code d'une fonction `incrémenter` qui modifie un entier en l'incrémentant de 1. Écrivez une fonction `main` faisant appel à cette fonction.

Q 2.3 Plus généralement, pour quelles raisons peut on vouloir passer un pointeur en argument d'une fonction plutôt qu'une valeur ?

Exercice 3 – Pointeurs et tableaux

Un tableau est un pointeur sur une zone mémoire contiguë, statiquement ou dynamiquement alloué. L'arithmétique des pointeurs permet de retrouver facilement un élément à partir du pointeur initial plus un déplacement. Par exemple :

```
1      int t[5]; //allocation memoire d'un tableau d'entiers de taille 5
2      printf("%d",t[3]); //acces au 4eme entier
```

Q 3.1 Comment définir un tel tableau dynamiquement ? Comment accéder à ses éléments ?

Q 3.2 On souhaite connaître la taille d'un tableau qui a été alloué dynamiquement. Où peut-on trouver cette information ? Est-ce un problème ? Quelle solution proposée ?

Exercice 4 – Complexité temporelle

Q 4.1 Quelle est la complexité temporelle des algorithmes suivants ? On ne vous demande pas de donner le nombre exact d'opérations élémentaires, mais plutôt de répondre à cette question en donnant des ordres de grandeur avec les notation de Landau.

```c
#include <stdlib.h> //pour rand
#include <time.h> //pour time
#include <stdio.h> //pour printf

void produit_ou_somme (int n, int m){
    double nombre;
    srand(time(NULL));
    nombre = rand()*1.0/RAND_MAX; // on multiplie par 1.0 sinon le resultat est 0

    if (nombre<0.5){
      printf("Produit : %f\n", n*m);
    } else {
      printf("Somme : %f\n", n+m);
    }
}
```

```c
int equals_tab (int* tab1, int* tab2, int n){
    int i;
    for (i=0;i<n;i++){
        if (tab1[i] != tab2[i]){
            return 0; /*En C, Faux est repr'esent'e par la valeur 0 */
        }
    }
    return 1; /*En C, Vrai est repr'esent'e par tout sauf 0 */
}
```

```c
int equals_one_line(int** mat1, int** mat2, int n, int m){
    int i, equals;
    i=0;
    while (i<n){
        equals = equals_tab(mat1[i], mat2[i], m);
        if (equals == 1){
            return 1;
        }
        i = i+1;
    }
    return 0;
}
```

Exercice 5 – Compilation séparée

Séparer un programme en plusieurs fichiers est une bonne idée quand le code prend de l'ampleur. La création de bibliothèques de fonctions accroît fortement la maintenabilité d'un code C et permet de penser à la ré-utilisabilité de code écrit.

Considérons un projet composé de trois fichiers `A.c`, `B.c` et `prog.c`, tels que `A.c` et `B.c` sont des librairies de fonctions utilisées par `prog.c`. On peut demander à compiler l'ensemble dans un seul programme (de nom `prog`) par la commande suivante :

```
gcc A.c B.c prog.c -o prog
```

Pour que la compilation se passe bien, il est nécessaire d'inclure dans `prog.c` au moins une spécification (une signature) des fonctions que `A.c` et `B.c` apportent au projet. Pour ce faire, on utilise généralement des fichiers *header* (extension ".h") contenant toutes les spécifications nécessaires à l'utilisation de la bibliothèque de fonctions. Les fichiers *headers* pourront ensuite être inclus dans chaque fichier `*.c` qui

voudra utiliser les fonctions définies dans ces bibliothèques. Dans notre cas, nous allons donc inclure les fichiers `A.h` et `B.h` dans le fichier `prog.c`.

Par ailleurs, devoir compiler `A.c` et `B.c` à chaque fois que l'on fait une modification mineure dans le fichier `prog.c` n'est pas forcément utile. En effet, il s'avère que la compilation peut se décomposer en deux étapes bien différentes :

1. **La compilation** (proprement dite) : on passe d'un fichier `.c` à un fichier `.o`. Le fichier source est en fait compilé en langage machine (fichier objet). Mais le code est 'relogeable' : on ne connaît pas l'adresse des fonctions ni des variables. Plus exactement, les adresses sont encore 'translatables', données dans une mémoire virtuelle.
2. **L'édition de lien** : les mémoires virtuelles des différents fichiers objets sont unifiées et toutes les adresses sont résolues de manière univoque.

C'est uniquement cette dernière étape qui doit être faite d'un seul tenant, avec l'ensemble des fichiers-objets, résultats de compilations qui, elles, peuvent être séparées. Dans notre cas, nous pouvons donc exécuter les commandes suivantes :

```
gcc -c A.c
gcc -c B.c
gcc -c prog.c
gcc A.o B.o prog.o -o prog
```

puis exécuter seulement les deux dernières commandes en cas de modifications de `prog.c`.

Pour ne pas avoir à retenir quels fichiers ont été modifiés depuis la dernière compilation du projet, on peut utiliser l'outil `make` qui permet d'automatiser cette tâche à partir d'un `Makefile`. Un fichier `Makefile` suit un format permettant de déclarer les différentes dépendances entre fichiers d'un même projet, afin de trouver le nombre minimum d'opérations (compilations ou édition de lien) nécessaires afin de produire l'exécutable final. La commande `make` utilise le fichier `Makefile` pour effectuer ces opérations nécessaires.

Q 5.1 Reprenons nos trois fichiers `A.c`, `B.c` et `prog.c` et supposons que :
- seule une fonction `int f(int)` est définie dans le fichier source `A`,
- seule une fonction `void g(int)` est définie dans le fichier source `B` et qu'elle utilise la fonction `int f(int)`,
- le fichier `prog.c` contient seulement une fonction `main(void)` appelant `int f(int)` et `void g(int)`.

Proposer une organisation, le squelette de chaque fichier et le Makefile associé.