TD1: Pointeurs, fonctions, tableaux, complexité

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 – Nombre aléatoire et saisie clavier

Q 2.1 Dans un fichier appelé `main.c`, écrivez un programme permettant d'afficher un entier tiré aléatoirement. Donner les commandes permettant de compiler et lancer ce programme.

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.

Q 2.2 Modifier le code pour tirer aléatoirement un nombre réel compris entre 0 et 1 (inclus). Donner les commandes permettant de lancer le code ainsi modifié.

Q 2.3 Modifier le code de la question 2.1 pour qu'il retourne un entier entre 0 et N (inclus), où N est un entier saisie au clavier.

Exercice 3 – Complexité temporelle

Q 3.1 Quelle est la complexité-temps pire cas des algorithmes suivants ? On vous demande de répondre à cette question en utilisant la notation de Landau O, en comptant le nombre d'instructions simples des programmes (on ignore les structures de contrôle).

```c
int comparer_tab (int* tab1, int* tab2, int n){
    int i;
    for (i=0;i<n;i++){
        if (tab1[i] != tab[2]){
            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 produit_ou_somme (int n, int m, double p){
    if (p<0.5){
        return n*m;
    } else {
        return n+m;
    }
}
```

```c
void compare_mat (int** mat1, int** mat2, int n, int m){
    int i, max, rest;
    i=0;
    max=n;
    while (i<max){
        res = compare_tab(mat1[i], mat2[i], m);
        if (res == 0){
            max = 0;
        }
        i = i+1;
    }
    printf("Ligne de la premiere difference : \d", i);
}
```

Exercice 4 – 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`.
- `&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 (commande `malloc`). L'erreur la plus fréquente des pointeurs est l'oubli de suppression des valeurs en

mémoire créées dynamiquement (commande `free`). L'exemple ci-dessous permet d'illustrer ces différentes notions :

```
1  #include <stdio.h>
2  #include <malloc.h>
3
4  void main(void){
5      int *p;  /* p est un pointeur sur un entier */
6      int i = 1;
7
8      /* r'ecup'eration de l'adresse d'un int pr'e-existant */
9      p = &i;
10     printf("%d\n", *p);
11
12     /* cr'eation dynamique d'un entier */
13     p = (int *) malloc(sizeof(int));
14     printf("%d\n", *p);  /* valeur ? */
15
16     /* nettoyage */
17     free(p);
18 }
```

Q 4.1 Que pensez-vous des lignes 13-14 ?

Q 4.2 Comment faire une fonction qui modifie la valeur de son argument, en sachant que les fonctions C passent leurs arguments *par valeur* (recopie de la valeur dans une nouvelle zone mémoire) ? 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 4.3 Pour quelles raisons peut on vouloir passer un pointeur en argument d'une fonction plutôt qu'une valeur ?

Exercice 5 – Pointeurs et tableaux

Un tableau est un pointeur sur une zone de mémoire statiquement allouée. L'arithmétique des pointeurs permet de retrouver facilement un élément à partir du pointeur initial plus un déplacement :

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

Q 5.1 Comment définir un tel tableau dynamiquement ? Comment accéder à ses éléments ?

Q 5.2 L'instruction `float t[5];` permet de définir un tableau en allouant une zone mémoire de 5 `float`, et de définir un pointeur `t` qui contient l'adresse de cette zone. Mais où se trouve l'information sur la taille de cette zone ? Est-ce un problème ? Quelle solution proposée ?

Exercice 6 – Testez vos connaissances sur les pointeurs

Q 6.1 Pour une variable `p` de type pointeur sur un entier (`int*`), que représentent `&p, p et *p` et quel est leur type ?

Q 6.2 Donner les affichages réalisées par le programme suivant.

```c
#include <stdio.h>
#include <malloc.h>

int echange_pointeur(int *p, int *q){
    printf("Adresse de p dans la fonction = %p\n", &p);
    printf("Adresse vers laquelle pointe p dans la fonction = %p\n", p);

    printf("Adresse de q dans la fonction = %p\n", &q);
    printf("Adresse vers laquelle pointe q dans la fonction = %p\n", q);

    p = q;

    printf("Nouvelle adresse de p dans la fonction = %p\n", &p);
    printf("Nouvelle adresse vers laquelle pointe p dans la fonction = %p\n", p);
    printf("Valeur vers laquelle pointe p dans la fonction = %d\n", *p);
}

int main(void){
    int *p;
    int *q;

    printf("Adresse de p dans main = %p\n", &p);
    printf("Adresse de q dans main = %p\n", &q);

    int i = 1;
    int j = 2;

    printf("Adresse de i = %p\n", &i);
    printf("Adresse de j = %p\n", &j);

    p = &i;
    q = &j;

    printf("Adresse vers laquelle pointe p dans main avant echange = %p\n", p);
    printf("Valeur vers laquelle pointe p avant echange = %d\n", *p);

    echange_pointeur(p,q);

    printf("Adresse de p dans main apres echange = %p\n", &p);
    printf("Adresse vers laquelle pointe p dans main apres echange = %p\n", p);
    printf("Valeur vers laquelle pointe p dans main apres echange = %d\n", *p);
    printf("Adresse de q dans main apres echange = %p\n", q);
    printf("Adresse vers laquelle pointe q dans main apres echange = %p\n", q);

    return 0;
}
```

Q 6.3 Même question avec le programme suivant.

```c
#include <stdio.h>
#include <malloc.h>

int echange_pointeur(int **ad_p, int *q){
    printf("Adresse de p dans la fonction = %p\n", ad_p);
    printf("Adresse vers laquelle pointe p dans la fonction = %p\n", *ad_p);

    printf("Adresse de q dans la fonction = %p\n", &q);
```

```
9     printf("Adresse vers laquelle pointe q dans la fonction = %p\n", q);
10
11    (*ad_p = q);
12
13    printf("Nouvelle adresse de p dans la fonction = %p\n", ad_p);
14    printf("Nouvelle adresse vers laquelle pointe p dans la fonction = %p\n", *ad_p);
15    printf("Valeur vers laquelle pointe p dans la fonction = %d\n", **ad_p);
16 }
17
18 int main(void){
19    int *p;
20    int *q;
21
22    printf("Adresse de p dans main = %p\n", &p);
23    printf("Adresse de q dans main = %p\n", &q);
24
25    int i = 1;
26    int j = 2;
27
28    printf("Adresse de i = %p\n", &i);
29    printf("Adresse de j = %p\n", &j);
30
31    p = &i;
32    q = &j;
33
34    printf("Adresse vers laquelle pointe p dans main avant echange = %p\n", p);
35    printf("Valeur vers laquelle pointe p avant echange = %d\n", *p);
36
37    echange_pointeur(&p,q);
38
39    printf("Adresse de p dans main apres echange = %p\n", &p);
40    printf("Adresse vers laquelle pointe p dans main apres echange = %p\n", p);
41    printf("Valeur vers laquelle pointe p dans main apres echange = %d\n", *p);
42    printf("Adresse de q dans main apres echange = %p\n", q);
43    printf("Adresse vers laquelle pointe q dans main apres echange = %p\n", q);
44
45    return 0;
46 }
```