Informatique *TD*
Bonnes pratiques de programmation Ipesup

Exercice 1- Lisibilité en C

On considère le code C suivant :

```
#include <stdio.h>

void f(int *t, int n,int x){
int i;
for(i=0;i<n;i++){if(t[i]==x){printf("trouve\n");return;} }
printf("pas trouve\n);}

int main(){
int a[5]={1,4,2,9,4};
f(a,5,9);
}
```

1. Corriger les erreurs de compilation.
2. Réécrire cette fonction avec une indentation propre, des retours à la ligne raisonnables (pas de ligne trop longue) et des noms de variables plus explicites si nécessaire.
3. Ajouter une spécification sous forme de commentaire au-dessus de `f` décrivant précisément l'entrée et la sortie attendues.

Exercice 2- Lisibilité en OCaml

On considère le code OCaml suivant :

```
let rec s l=
match l with
|[]->0
|x::q->if x mod 2=0 then x+s q else s q
```

1. Réécrire ce code de façon lisible : indentation, espaces, noms plus clairs pour la fonction et les variables.
2. Ajouter un commentaire décrivant ce que calcule la fonction (entrée, sortie, effet).
3. Modifier la fonction pour qu'elle calcule la somme des éléments impairs au lieu des pairs, en conservant une bonne lisibilité.

Exercice 3- Débogage en C avec `gdb`

On considère le programme suivant :

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

struct node {
  int value;
  struct node *next;
};

void print_list(struct node *l) {
  while (l->next != NULL) {
    printf("%d\n", l->value);
    l = l->next;
  }
}

int main(void) {
  struct node *p = NULL;
  p->value = 42;
  p->next = NULL;
  print_list(p);
  return 0;
}
```

1. Compiler le programme avec les options `-Wall -g`.
2. Exécuter le programme : que se passe-t-il ?
3. Lancer `gdb`, utiliser la commande `run` puis `bt` pour déterminer la ligne exacte qui provoque le crash.
4. Expliquer la cause de la segmentation fault.
5. Corriger le programme pour qu'il fonctionne sans erreur.

Exercice 4- Débogage en OCaml avec `OCAMLRUNPARAM=b`

On donne le programme OCaml suivant dans `file.ml` :

```
let rec find_first_even l =
  match l with
  | [] -> List.hd l
  | x :: q ->
      if x mod 2 = 0 then x
      else find_first_even q

let () =
  let l = [1;3;5;7] in
  let x = find_first_even l in
  Printf.printf "Premier pair = %d\n" x
```

1. Compiler avec `ocamlc -g file.ml -o program`.

2. Exécuter `./program` : quelle exception est levée ?
3. Relancer en utilisant `OCAMLRUNPARAM=b ./program`. Recopier et interpréter la trace obtenue (fonctions et lignes concernées).
4. Expliquer l'erreur logique dans `find_first_even`.
5. Corriger la fonction pour qu'elle renvoie une option `int option` (soit `Some x` s'il existe un élément pair, soit `None` sinon).

Exercice 5- Programmation défensive en C avec `assert`

On considère une fonction qui accède à un tableau :

```
#include <assert.h>

int get_value(int *a, int n, int i) {
  assert(i <= n);
  return a[i];
}
```

1. Expliquer pourquoi l'assertion `assert(i <= n);` est incorrecte.
2. Proposer une version correcte de l'assertion (en tenant compte des indices valides).
3. Donner un exemple d'appel de `get_value` pour lequel l'assertion actuelle ne déclenche pas d'erreur, mais l'accès est pourtant hors bornes.
4. Discuter l'intérêt de garder ces assertions pendant le développement, puis de les désactiver en production (option `-DNDEBUG`).

Exercice 6- `assert false` en OCaml

On veut écrire une fonction qui renvoie le dernier élément d'une liste non vide :

```
let rec last l =
  match l with
  | [] -> (* à compléter *)
  | [x] -> x
  | _ :: q -> last q
```

1. Compléter la fonction en utilisant `assert false` dans le cas `[]`.
2. Expliquer pourquoi, dans l'usage prévu, ce cas ne devrait jamais se produire.
3. Discuter l'intérêt de `assert false` pour marquer un cas théoriquement impossible tout en gardant un typage correct.
4. Que se passe-t-il si on appelle `last []` malgré tout ? Pourquoi ce comportement peut-il aider au débogage ?

Exercice 7- Débogage par affichages

Le programme suivant ne termine pas, et l'on veut comprendre pourquoi :

```c
#include <stdio.h>

int f(int n) {
  int i = 0;
  while (i < n) {
    if (i * i > n)
      n = n - 1;
  }
  return n;
}

int main(void) {
  printf("%d\n", f(10));
  return 0;
}
```

1. Expliquer informellement pourquoi la boucle `while` peut ne jamais se terminer.
2. Ajouter des `printf` (valeurs de `i` et `n`) pour suivre l'évolution des variables et confirmer votre hypothèse.
3. Proposer une version corrigée de la fonction `f` (par exemple en mettant à jour correctement `i` ou en modifiant la condition).

Exercice 8- Tests en boîte noire en OCaml

On veut tester une fonction de tri `my_sort : int list -> int list`.

1. Écrire une fonction `test_once n m` qui :
   — construit une liste de `n` entiers aléatoires dans `[0, m-1]` ;
   — vérifie avec `assert` que `my_sort l = List.sort Stdlib.compare l`.
2. Écrire une fonction `run_tests ()` qui appelle `test_once` pour différentes valeurs de `n` et `m` (inclure les cas `n = 0` et `n = 1`).
3. Expliquer en quoi ces tests sont des tests en boîte noire.

Exercice 9- Mesurer le temps d'exécution en C

On souhaite mesurer le temps d'exécution d'une fonction `work(int n)`. Compléter le programme suivant :

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

void work(int n) {
  volatile long long s = 0;
  for (int i = 0; i < n; i++) {
    s += i;
  }
}
```

```c
int main(void) {
  int n = 100000000; /* 1e8 */

  /* TODO : mesurer le temps d'exécution de work(n) */

  return 0;
}
```

1. Utiliser `clock_t start = clock();` puis `clock_t stop = clock();` pour mesurer le temps de `work(n)`.
2. Afficher le temps en secondes en utilisant `CLOCKS_PER_SEC`.
3. Exécuter le programme plusieurs fois et commenter la stabilité de la mesure.

Exercice 10- Mesurer le temps d'exécution en OCaml)

On considère la fonction suivante :

```ocaml
let work n =
  let s = ref 0 in
  for i = 0 to n - 1 do
    s := !s + i
  done
```

1. Écrire un programme complet qui :
   — lit un entier `n` sur la ligne de commande (ou fixe une valeur de `n`),
   — mesure le temps d'exécution de `work n` à l'aide de `Sys.time ()`,
   — affiche ce temps.
2. Faire varier `n` (par exemple $10^5, 10^6, 10^7$) et observer l'évolution du temps.
3. Expliquer pourquoi il est utile de répéter la mesure plusieurs fois et de moyenner les résultats.

Exercice 1

Les programmes suivants contiennent des erreurs. Les identifier et proposer des solutions.

1. ```
   bool f(int tab[], int tai) {
       bool res = true;
       int ind = 0;
       while (ind < tai - 1) {
           if (tab[ind] > tab[ind + 1]) {
               res = false;
           }
           ind = ind + 1;
       }
       return res;
   }
   ```
2. ```
   void print(int a[], int n) {
       for (int i = 0; i < n; i++)
           printf("%d", a[i]);
       printf("\n");
   }
   ```
3. ```
   int sum(int a[], int n) {
       int s = 0;
       for (int i = 0; i < n-1; i += 2)
           s += a[i] + a[i+1];
       return s;
   }
   ```

Exercice 2

Les programmes suivants contiennent des erreurs. Les identifier et proposer des solutions.

1. ```
   let rec sum l = match l with
     | [] -> 0      (* la liste est vide *)
     | x :: l -> x + sum l   (* on ajoute le premier élément *)
                             (* à la somme du reste de la liste *)
   ```
2. ```
   let conjonction a b =
     if a = true then
       if b = true then true else false
     else
       false
   ```
3. ```
   let rec map_reduce map red acc first l = match l with
     | [] -> first acc
     | x :: l -> map_reduce map red (red (map x) acc) first l
   ```