TP 5 : Langage C : manipulation de pointeurs et de tableaux

On utilisera la ligne de commande suivante pour compiler :

```
$ gcc -O0 -Wall -Wextra -Wvla -Werror -fsanitize=address,undefined
↪ votre_fichier.c -o votre_executable
```
Shell

On a ajouté des options au compilateur qui rajoutent des avertissements (`-Wall -Wextra -Wvla`) et les considèrent comme des erreurs (`-Werror`), et qui détecte les fuites de mémoire et les comportements non définis à l'exécution (`-fsanitize=address,undefined`).

Exercice 1 – Manipulation de pointeurs

Écrivez un programme effectuant les opérations suivantes :

1. D'abord, déclarer un entier `k` initialisé à `-25`,
2. Déclarer un pointeur de type `int*` initialisé à `NULL`,
3. Modifier ce pointeur pour qu'il indique l'adresse contenant `k`,
4. Modifier la valeur de `k`, d'abord comme d'habitude, puis sans passer par la variable `k`,
5. Afficher la valeur de `k`, d'abord en utilisant la variable `k` puis sans l'utiliser,
6. Afficher l'adresse de `k` : pour cela, on pourra utiliser le format `"%p"`.

Exercice 2 – Échange d'entiers

Un élève insouciant propose la fonction suivante pour échanger la valeur de deux entiers :

```c
void echange_entiers(int x, int y) {
  int temp = x;
  x = y;
  y = temp;
}
```
C

▶ **Question 1** Tester la fonction précédente et expliquer son effet éventuel.

★ **Question 2** En proposer une version corrigée.

Exercice 3 – Une fonction sur des pointeurs

```c
void fonction_mystere(int* x, int* y) {
  *x = *x - *y;
  *y = *x + *y;
  *x = *y - *x;
}
```
C

▶ **Question 1** Tester cette fonction et deviner son effet. Le montrer.

▶ **Question 2** Est-ce que cette fonction marche si les deux valeurs pointées par `x` et `y` sont égales ?

▶ **Question 3** Est-ce que cette fonction marche si `x == y` ?

Exercice 4 – Utilisation de la fonction `malloc`

Cet exercice vise à vous entraîner à utiliser la fonction `malloc`.

▶ **Question 1** Vérifier quel est le nombre d'octets utilisés pour stocker un `double` en C. *Remarque : la valeur de retour de* `sizeof` *est un* `long unsigned int`*, qui a besoin du format* `"%ld"` *pour être affiché dans un* `printf`.

★ **Question 2** Grâce à cette information, allouer suffisamment de mémoire pour stocker un `double` avec la fonction `malloc`.

▶ **Question 3** Compiler votre code. Que remarque-t-on ?

Remarque 1 – Transtypage de pointeurs

La valeur de sortie de la fonction `malloc` a pour type `void*`, c'est-à-dire un pointeur vers un octet sans préciser le type de ce qui y est stocké. C convertit automatiquement les pointeurs de type `void*` vers les autres types pointeurs, mais on peut l'expliciter avec la syntaxe suivante :

```c
double* p = (double*) malloc(...)
```
C

La syntaxe générale pour une conversion de type est la même : `(type) valeur` convertit la valeur dans ce type. Dans le cadre du programme, les transtypages sont imposés pour les valeurs de retour de `malloc` : dans la vraie vie, ça ne change rien tant qu'on n'essaye pas de compiler un code C avec un compilateur C++ (qui impose, lui, ces conversions).

▶ **Question 4** Écrire une fonction de prototype `double* creer_tableau(int n; double x)` qui alloue suffisamment de mémoire pour stocker n `double`, et initalise ces cases à x.
Puisque cette mémoire est allouée par `malloc`, elle n'est pas désallouée à la fin de l'exécution de `creer_tableau`, contrairement aux tableaux créés par la syntaxe `tab[10]` : la première est allouée sur le tas et doit être désallouée à la main par un appel à `free` sur le pointeur, alors que la deuxième est allouée sur la pile et est donc supprimée lors du rétrécissement de la pile d'appel à la fin de l'exécution de la fonction.

▶ **Question 5** Dans la fonction `main`, créer un tableau initialisé avec la fonction `creer_tableau` et afficher sa première valeur juste avant l'instruction `return 0`. Compiler et exécuter : que remarque-t-on ?

▶ **Question 6** Corriger votre programme pour éviter cette erreur.

Exercice 5 – Découverte des structures en C

Comme en OCaml, il est possible de définir des types construits au-delà des types prédéfinis. Ceux au programme de la MP2I sont similaires aux *enregistrements* en OCaml :

```c
struct S {int a; double b; char* c};
```

Cette instruction définit un nouveau type, nommé `struct S`. Par exemple, on pourra l'initialiser avec :

```c
struct S s = {.a = 25, .b = 3.25e19, .c = "Bonjour !"};
```

On remarque qu'une chaîne de caractères a pour type `char*`.
L'accès / la modification d'un champ d'une structure se fait par la syntaxe `s.a`. On peut aussi initialiser champ par champ la structure. Comme pour les autres types, on évitera de définir des variables non initalisées. Quand une structure est donnée en argument d'une fonction, elle est intégralement copiée, comme le serait un entier ou un flottant. Ainsi, il faut fournir en argument un *pointeur vers une structure* pour pouvoir faire modifier une structure par un appel de fonction.

▶ **Question 1** Proposer un type pour représenter des nombres complexes.

▶ **Question 2** Écrire des fonctions implémentant les opérations classiques sur les nombres complexes : conjugaison, somme, multiplication, partie réelle et partie imaginaire.

On pourra aussi utiliser le mot-clé `typedef` pour définir un alias de type et se passer du mot-clé `struct` partout. Par exemple, on pourra écrire :

```c
typedef struct complexe complexe_t;
```

`complexe_t` est un alias pour `struct complexe`. On pourra aussi combiner les deux syntaxes en écrivant `typedef struct { ... } complexe_t;`.
Alors, `complexe_t` est un alias pour le type sans avoir à écrire `struct` tout le temps.

★ **Question 3** Écrire une fonction conjuguant "en place" un complexe, de prototype `void conjugue_en_place(complexe* x)`. Vous pourrez utiliser la syntaxe `x->nom_champ` pour accéder et modifier un champ d'une structure pointée par `x`.
`x->nom_champ` est une syntaxe équivalente à `(*x).nom_champ` : on la préférera, mais on gardera conscience qu'elle s'utilise pour `x` un *pointeur* vers une structure, et non pas une structure elle-même.

Exercice 6 – Tableau décoré

Dans cet exercice, on va utiliser les structures pour créer un type de tableaux dont on peut connaître la taille. On va se limiter aux tableaux d'entiers :

```c
typedef struct {
  int longueur;
  int* donnees;
} tableau_entiers
```

▶ **Question 1** Définir une fonction de prototype `int get(tableau_entiers t, int k)` qui récupère le $k$ième élément du tableau. On pourra utiliser l'instruction `assert(cond_bool)` de la bibliothèque `assert.h` qui ne fait rien si `cond_bool` est évaluée à `true` et lève une erreur sinon. Faire de même pour la fonction de prototype `void set(tableau_entiers t, int indice, int nouvelle_valeur)`.
Comme pour les tableaux, pour créer une structure sur le tas, on peut utiliser la fonction `malloc(sizeof(tableau_entiers))`.

▶ **Question 2** Écrire une fonction de prototype `tableau_entiers* creer_tableau(int longueur, int valeur_initiale)` qui crée un tableau décoré initialisé à la valeur `donnees`.

▶ **Question 3** Écrire une fonction de prototype `void liberer_tableau_entiers(tableau_entiers* t)` qui libère *toute* la mémoire utilisée par `t`. Vérifier qu'elle libère bien toute la mémoire en la testant.