TP 2 : Mandelbrot

1 Manipulation de pointeurs

1.1 Exercice 1

On considère le code suivant :

```c
#include <stdio.h>

int n = 3;
int p;

int f(int n) {
    int *p = &n;
    int x = n + *p;
    return x + 1;
}

int g(int x, int y) {
    int z = f(x);
    return z + f(y);
}

int main() {
    p = 4;
    int result = g(n, p);
    printf("result = %d\n", result);
    return 0;
}
```

Aller sur le site `https://pythontutor.com/c.html` (vous pouvez taper *C tutor* sur Google, ça devrait être le premier résultat) et visualiser l'exécution pas à pas du programme. Essayer de bien comprendre ce qui se passe.

1.2 Exercice 2

Dans chacun des cas suivants :
- Déterminer si le programme est « faux » (lecture d'une variable non initialisée, déréférencement d'un pointeur invalide, erreur de type...) ;
- Écrire sur papier (sans exécuter le programme) ce qu'on obtiendrait sur *C Tutor* (évolution du schéma mémoire et affichage produit) ;
- Copier le code sur *C Tutor* et vérifier.

1. Code 1

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

void print_bool(bool b) {
    if (b) {
        printf("true\n");
    } else {
        printf("false\n");
    }
}

int main() {
    double pi = 3.14;
    double e;
    double *p = NULL;
    p = &e;
    *p = pi;
    print_bool(e == pi);
    pi = 4.5;
    print_bool(e == pi);
}
```

2. Code 2

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

int x = 7;
int y = 12;
int *p;

int f(int x) {
    printf("x = %d\n", x);
    printf("y = %d\n", y);
    printf("*p = %d\n", *p);
    int y = 1;
    printf("y = %d\n", y);
    x = x + y;
    return x;

}

int main() {
    int z;
    p = &x;
    z = f(x + 1);
    printf("z = %d\n", z);
    return 0;
}
```

1.3 Exercice 3

Écrire une fonction de prototype :

```c
void extrema(int t[], int n, int *min, int *max)
```

Les préconditions sont :
— La longueur de `t` vaut `n`, et elle est strictement positive ;
— `min` et `max` sont des pointeurs valides.
La fonction affectera le minimum de `t` à l'objet pointé par `min`, et le maximum à celui pointé par `max`.

1.4 Exercice 4

On considère la fonction suivante :

```
void mystere(int *x, int *y) {
    *x = *x - *y;
    *y = *x + *y;
    *x = *y - *x;
}
```

1. Quel affichage obtiendrait-on avec le code suivant ?

   ```
   int x = 3;
   int y = 4;
   mystere(&x, &y);
   printf("x = %d\n", x);
   printf("y = %d\n", y);
   ```

2. De manière générale, quel est l'effet de la fonction `mystere` ? On justifiera.
3. Quel affichage obtiendrait-on avec le code suivant ?

   ```
   int x2 = 3;
   int y2 = 3;
   mystere(&x2, &y2);
   printf("x2 = %d\n", x2);
   printf("y2 = %d\n", y2);
   ```

4. Quel affichage obtiendrait-on avec le code suivant ?

   ```
   int x3 = 3;
   mystere(&x3, &x3);
   printf("x3 = %d\n", x3);
   ```

   Quel est le problème dans la démonstration faite plus haut ?

1.5 Exercice 5

On considère le code suivant :

```
#include <stdio.h>

void f(int n, int *nmax) {
    printf("Début de l'appel de f(%d, _)\n", n); // debut
    printf("n     = %d\n", n);
    printf("&n    = %p\n", (void *)&n);
    printf("nmax  = %p\n", (void *)nmax);
    printf("*nmax = %d\n", *nmax);
    printf("&nmax = %p\n", (void *)&nmax);
    if (n < *nmax) f(n + 1, nmax);
    printf("Fin de l'appel de f(%d, _)\n", n);  // fin
}

int main() {
    int n = 2;
    f(0, &n);
    return 0;
}
```

1. En oubliant les `printf` situés ailleurs qu'aux lignes `debut` et `fin`, prévoir l'affichage produit par la fonction.
2. Parmi les autres lignes provoquant un affichage, lesquelles :
   (a) Affichent toujours la même chose ?

(b) Affichent des choses différentes, mais que l'on peut prévoir parfaitement en regardant le code ?
(c) Affichent des choses différentes et partiellement imprévisibles ?

3. Prévoir le plus complètement possible l'affichage produit par le programme.
4. Exécuter ce programme (dans le terminal et éventuellement avec *C Tutor*) et observer l'affichage produit.
5. Quelle est la taille (en octets) du bloc d'activation de `f` ?

1.6 Exercice 6

1. Écrire une fonction `incremente` qui prend en entrée un pointeur vers un entier et incrémente la valeur de cet entier de 1 (cette fonction ne renverra rien).
2. Dans tous les exemples suivants, on souhaite qu'un appel `f(px, py)` incrémente celle des valeurs pointées par `px` et `py` qui est la plus petite (ou celle pointée par `px` en cas d'égalité). Par exemple :

```c
#include <stdio.h>

int main() {
    int x = 4;
    int y = 3;
    printf("x = %d, y = %d\n", x, y);
    f(&x, &y);
    printf("x = %d, y = %d\n", x, y);
    f(&x, &y);
    printf("x = %d, y = %d\n", x, y);
    return 0;
}
```

doit donner l'affichage suivant :

```
x = 4, y = 3
x = 4, y = 4
x = 5, y = 4
```

Dans chaque cas, dire si :
— Il y a un problème de type (et si oui, où) ;
— Les fonctions ont bien le comportement attendu (uniquement dans les cas où il n'y a pas de problème de type). Si ce n'est pas le cas, on expliquera d'où vient le problème.

(a) Code 1

```c
int plus_petit(int x, int y) {
    if (x <= y) return x;
    return y;
}

void f(int *px, int *py){
    incremente(plus_petit(*px, *py));
}
```

(b) Code 2

```c
int *plus_petit(int x, int y) {
    if (x <= y) return &x;
    return &y;
}

void f(int *px, int *py) {
    incremente(plus_petit(*px, *py));
}
```

(c) Code 3

```
int *plus_petit(int *x, int *y){
    if (*x <= *y) return x;
    return y;
}

void f(int *px, int *py){
    incremente(plus_petit(px, py));
}
```

(d) Code 4

```
int *plus_petit(int *x, int *y){
    int a = *x;
    int b = *y;
    if (a <= b) return &a;
    return &b;
}

void f(int *px, int *py){
    incremente(plus_petit(px, py));
}
```