TD 4

1 Pointeurs et variantes

1. Combien valent `!a`, `!b`, `!c` après exécution du code Caml suivant ?

```
let a = ref 2;;
let b = ref (!a);;
let c = a;;
a:= 9;;
```

2. Même question avec `a^`, `b^`, `c^` et le code Pascal suivant :

```
var a, b, c : integer^;
new a;  a^ := 2;
new b;  b^ := a^;
c := a;
a^ := 9;
```

3. Pareil avec `*a`, `*b`, `*c` et le code C suivant :

```
int *a, *b, *c;
a = malloc (sizeof (int));  *a = 2;
b = malloc (sizeof (int));  *b = *a;
c = a;
*a = 9;
```

4. Pareil avec `a.value`, `b.value`, `c.value` et le code Java suivant :

```
public class MutableInteger{
  private int value;
  MutableInteger (int v) { value = v; }
  MutableInteger (MutableInteger a) { value = a.value; }
  public int getValue() { return value; }
  public void setValue(int v) { value = v; }
}
...
  MutableInteger a, b, c;
  a = new MutableInteger(2);
  b = new MutableInteger(a);
  c = a;
  a.setValue(9);
```

(Question subsidiaire : à quoi sert cette manie des programmeurs Java de programmer des accesseurs `getValue()` et `setValue()` plutôt que d'accéder aux champs eux-mêmes ?)

5. Pareil avec `a.get()`, `b.get()`, `c.get()` en Python :

```
class ref:
    def __init__(self, obj): self.obj = obj
    def get(self):    return self.obj
    def copy(self):   return ref(self.obj)
    def set(self, obj):      self.obj = obj
a = ref 2
b = a.copy()
c = a
a.set(9)
```

2 Copie profonde

On considère l'extrait de code Java suivant :

```
public class Node{
  private int value;
  private Node left, right;
  public Node(v, l, r){
    value = v; left = l; right = r;
  }
  public Node(Node n){
    value = n.value; left = n.left; right = n.right;
  }

  // accesseurs, mutateurs... omis
}
...
  Node a = new Node(1, new Node(2, null, null),
                       new Node(3,
                           new Node(4, null, null), null));
```

6. Dessiner un diagramme boîtes-flèches décrivant la mémoire après l'exécution du code ci-dessus.
7. Combien vaut `a.left.value` après exécution ? `a.right.left.value` ?
8. Combien vaut `a.left.value` après exécution du code suivant, exécuté à la suite du code ci-dessus ?

```
Node b = new Node (5, new Node (a), null);
b.left.left = new Node (6, null, null);
```

9. Pareil mais avec le code suivant :

```
Node b = new Node (7, new Node (a), null);
b.left.left.value = 8;
```

10. Écrire une méthode `deepCopy()` qui effectue une *copie profonde*, c'est-à-dire qui copie tous les nœuds de l'arbre, et pas seulement la racine.
11. Que se passe-t-il si l'on lance le code suivant ?

```
a.left = a;
Node b = a.deepCopy();
```

12. Comment pourrait-on corriger le problème ? Le jeu en vaut-il la chandelle ?

3 Structures en C, pointeurs sur structures

On définit les structures suivantes en C.

```
struct s1 { int i; };
struct s2 { struct s1 s; };
struct s3 { struct s1 *p; };
```

13. Écrire deux fragments de code C qui créent des structures de type `struct s2`, resp. `struct s3`, contenant une structure contenant la valeur 42.
14. Dessiner un diagramme boîtes-flèches de la mémoire après l'exécution du code suivant :

```
  struct s1 s1; // oui, on peut donner le meme nom a une
  struct s2 s2; // variable et a un type struct...
  struct s3 s3, ss3;
  s1.i = 42; s2.s = s1;
  s3.p = &s1;
  ss3.p = malloc (sizeof (struct s1)); ss3.p->i = 54;
```

On différenciera bien le tas de la pile.

15. On définit la fonction suivante :

```
struct s1 *f(void) { // (void) = ne prend pas d'argument
  struct s1 s;
  s.i = 42;
  return &s;
}
```

Que se passe-t-il lorsqu'on exécute cette fonction ? En profiter pour répondre à la question : à quoi sert `malloc()` ?

16. On définit la fonction suivante :

```
struct s3 *f(void) {
  struct s1 s;   s.i = 9;
  struct s3 *p3 = malloc (sizeof (struct s3));
  p3->p = &s;
  return p3;
}
```

Que se passe-t-il lorsqu'on exécute cette fonction ? Que doit-on faire pour corriger le problème ?

17. Quelle est la différence entre les deux déclarations de structures suivantes ? La question ne porte pas tant sur l'absence ou la présence d'un $\star$, mais sur la façon dont sont représentés les objets des deux types en mémoire.

```
struct info1 { int value; struct s1 s1; };
struct info2 { int value; struct s1 *p1; };
```

18. L'une des deux déclarations suivantes sera rejetée par le compilateur C :

```
struct tree1 { int value; struct tree1 left, right; };
struct tree2 { int value; struct tree2 *left, *right; };
```

Laquelle ? Pourquoi ?

19. En Java, la seule déclaration autorisée d'un type équivalent sera celle qui ne mentionne pas « $\star$ », et pour cause : $\star$ n'existe pas en Java. La voici :

```
public class Tree {
  int value;
  Tree left, right;
  // methodes omises
}
```

De laquelle des définitions de la question précédente se rapproche-t-elle, en termes de représentation en mémoire ?

20. Pourquoi les deux formes d'« inclusions » de structures (voir la question 17) sont-elles autorisées en C ? (Seule la premiàre est effectivement appelée inclusion, au passage.)

21. Pourquoi Java et Caml n'autorisent-ils qu'une seule des formes d'« inclusion » ?

4 Blocs mémoires et chaînes

22. Écrire la fonction :

```
void *memchr(const void *s, int c, size_t n);
```

dont voici un extrait de la page man :

```
DESCRIPTION
     The memchr() function locates the first occurrence
     of c (converted to an unsigned char) in string s.

RETURN VALUES
     The memchr() function returns a pointer to the byte
     located, or NULL if no such byte exists within n bytes.
```

Le type `size_t` est un type entier, réservé aux specifications de longueurs. Le type `void *` est le type des pointeurs vers n'importe quel type. (Ce n'est *pas* le type des pointeurs vers les objets de type `void`... il n'y a pas d'objet de type `void`.) Le modificateur `const` promet au compilateur que la fonction ne modifiera rien de ce qui est pointé par `s` (mais la signification de `const` est, à mon avis, des plus obscures).

23. Écrire la fonction :

```
char *strchr (const char *s, int c);
```

dont la page man dit :

```
DESCRIPTION
     The strchr() function locates the first occurrence of c
 (converted to a char) in the string pointed to by s.
 The terminating null character is considered to be part
 of the string; therefore if c is '\0', the function locates
 the terminating '\0'.

RETURN VALUES
     The function strchr() returns a pointer to the located
 character, or NULL if the character does not appear in the
 string.
```

24. Écrire les fonctions :

```
char *strstr(const char *haystack, const char *needle);
void *memmem(const void *haystack, size_t haystacklen,
             const void *needle, size_t needlelen);
```

qui retournent la première position de `needle` dans `haystack`. On ne demande pas d'algorithme subtil (Boyer-Moore, Knuth-Morris-Pratt, automate de Simon, ou autre...).

25. Que pensez-vous de la façon d'implémenter les chaînes de caractères en C ? Qu'y aurait-il comme autre façon de faire ? Comment d'autres langages représentent-ils les chaînes de caractères, d'après vous ?