TD n° 3 : Pointeurs, tableaux et tris élémentaires

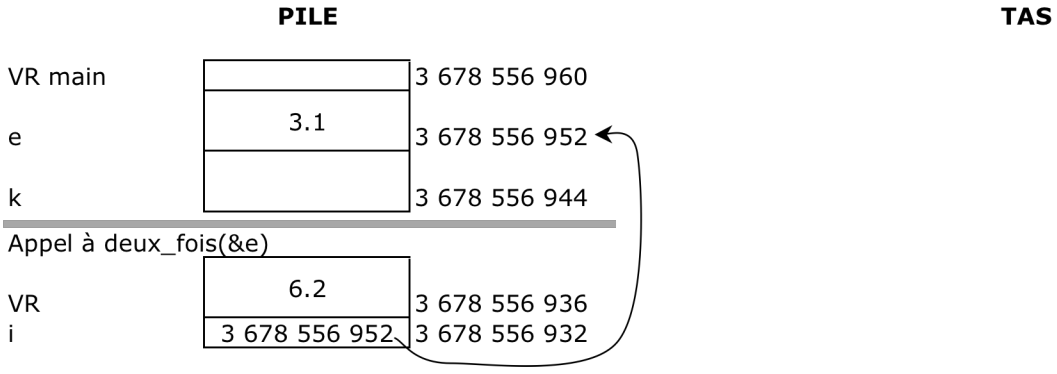
Exercice 1 : Pointeurs

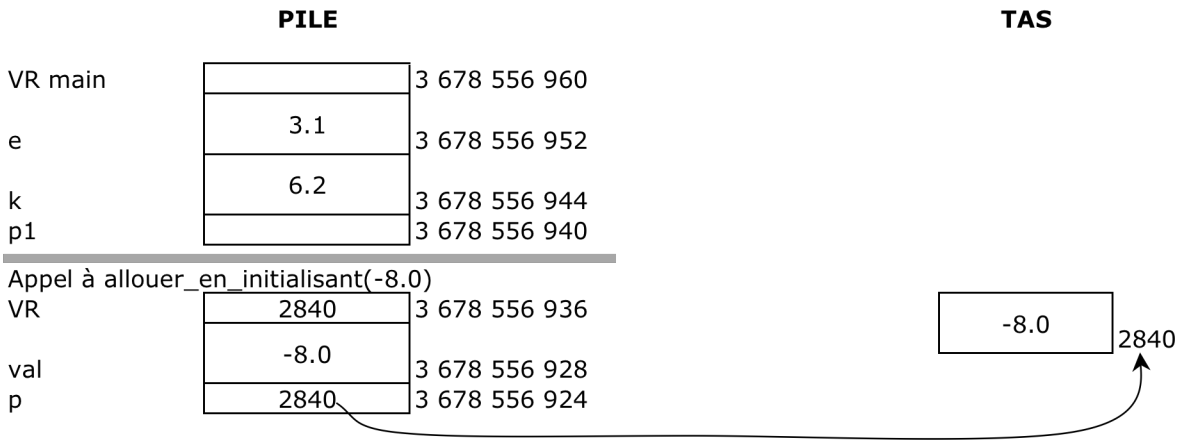
Soit le programme C suivant :

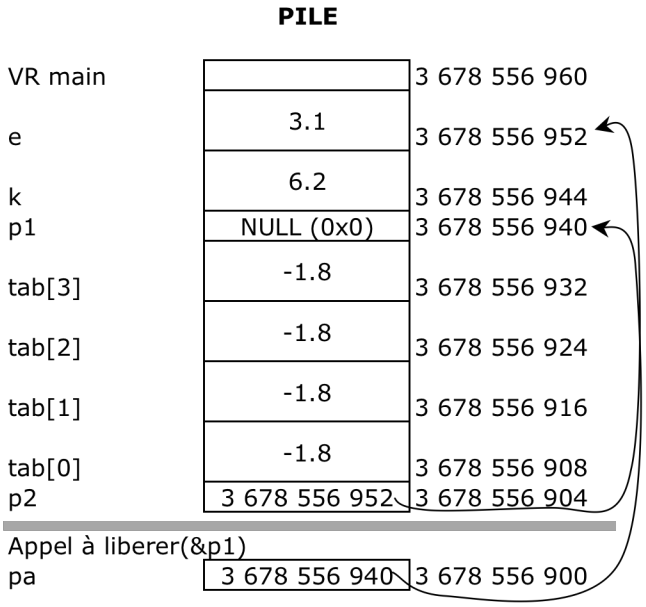
```
double deux_fois(const double * i) {return 2*(*i);}

double * allouer_en_initialisant(double val) {
   double *p = (double*) malloc(sizeof(double));
   *p = val; return p;
}

void liberer_en_mettant_a_zero(double ** pa) {free(*pa); *pa = 0x0;}

int main()
{
   double e = 3.1;
   double k = deux_fois(&e);

   double * p1 = allouer_en_initialisant(-8.0);

   double tab[4];
   double * p2;
   for(p2=tab; p2 < tab+4; p2++) { *p2 = (*p1) + k; }
   p2 = &e;
   liberer_en_mettant_a_zero(&p1);

   double **pp2 = &p2;
   return 0;
}
```

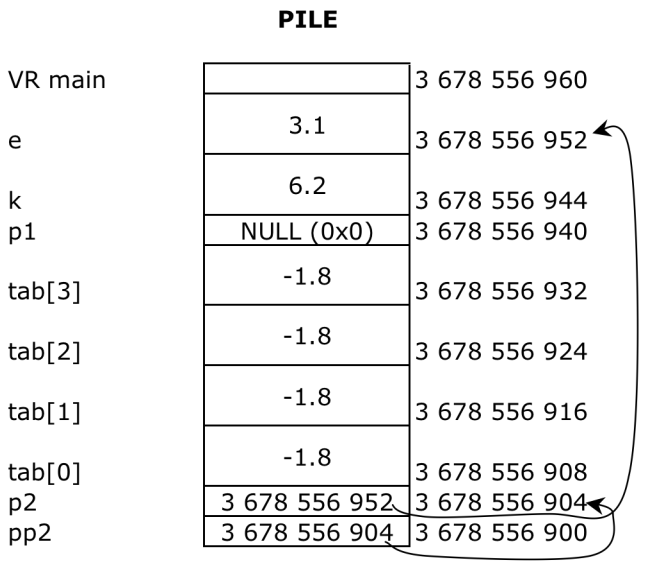
a) Pour chaque apparition du caractère '*', indiquez s'il s'agit d'une définition de variable de type pointeur, de l'opérateur de déréférencement d'une adresse, ou bien de l'opérateur de multiplication.

b) Faîtes la trace mémoire de ce programme, en supposant que la valeur de retour du main est située à l'adresse 3 678 556 960.




PILE
TAS
VR main
3 678 556 960
e
3.1
3 678 556 952
k
6.2
3 678 556 944
p1
3 678 556 940
Appel à allouer_en_initialisant(-8.0)
VR
2840
3 678 556 936
val
-8.0
3 678 556 928
p
2840
3 678 556 924
-8.0
2840


PILE
TAS
VR main
3 678 556 960
e
3.1
3 678 556 952
k
6.2
3 678 556 944
p1
NULL (0x0)
3 678 556 940
tab[3]
-1.8
3 678 556 932
tab[2]
-1.8
3 678 556 924
tab[1]
-1.8
3 678 556 916
tab[0]
-1.8
3 678 556 908
p2
3 678 556 952
3 678 556 904
Appel à liberer(&p1)
pa
3 678 556 940
3 678 556 900


PILE
TAS
VR main
3 678 556 960
e
3.1
3 678 556 952
k
6.2
3 678 556 944
p1
NULL (0x0)
3 678 556 940
tab[3]
-1.8
3 678 556 932
tab[2]
-1.8
3 678 556 924
tab[1]
-1.8
3 678 556 916
tab[0]
-1.8
3 678 556 908
p2
3 678 556 952
3 678 556 904
pp2
3 678 556 904
3 678 556 900

c) Indiquez pour chaque paramètre s'il est passé en mode donnée ou en mode donnée-résultat.

i : l'entier pointé par i est passé en mode donnée (passage par adresse mais mot-clé const)

val : le double est passé en mode donnée (passage par valeur)

pa : le pointeur pointé par pa est passé en mode donnée-résultat pour que l'adresse qu'il contient puisse être modifiée (on passe donc un pointeur sur un pointeur)

Exercice 2 : Cryptage d'une chaîne de caractères

Considérons un algorithme de cryptage de chaînes de caractères qui consiste à « additionner » les caractères du texte à crypter avec ceux d'une clé de chiffrement. Par exemple, le cryptage de la chaîne « Cherchez au pied de l'arbre » avec la clé « indice » peut s'illustrer ainsi :

- on place la clé en regard du texte à crypter, en répétant la clé autant de fois que nécessaire pour couvrir le texte, et en ignorant les caractères qui ne sont pas des lettres (ils seront laissés inchangés par l'algorithme de cryptage) :

C	h	e	r	c	h	e	z		a	u		p	i	e	d		d	e		l	'	a	r	b	r	e
i	n	d	i	c	e	i	n		d	i		c	e	i	n		d	i		c	'	e	i	n	d	i

- on remplace chaque lettre de la clé par sa position dans l'alphabet (0 pour 'a', 1 pour 'b'…),

C	h	e	r	c	h	e	z		a	u		p	i	e	d		d	e		l	'	a	r	b	r	e
8	13	3	8	2	4	8	13		3	8		2	4	8	13		3	8		2	'	4	8	13	3	8

- on remplace chaque lettre du texte à crypter par la lettre située *d* positions plus loin dans l'alphabet, où *d* est le nombre indiqué par la clé (si on dépasse z, on reboucle sur a, b etc) :

C	h	e	r	c	h	e	z		a	u		p	i	e	d		d	e		l	'	a	r	b	r	e
K	u	h	z	e	l	m	m		d	c		r	m	m	q		g	m		n	'	e	z	o	u	m

a) Si *x* est le code ASCII d'une lettre à crypter et *y* le code ASCII de la lettre de la clé située en regard, quelle formule permet de calculer le code ASCII résultant ? On supposera pour cette question que les deux lettres sont des minuscules. Indice : que vaut *y* – 'a' ?

((x – 'a' + y – 'a')%26) + 'a'

b) Ecrire en langage algorithmique la procédure de cryptage, dont voici l'entête :

Procédure crypter (texte : tableau de caractères, cle : tableau de caractères, result : tableau de caractères)
Préconditions : texte contient un ou plusieurs caractères suivis d'un '\0'. cle contient une ou plusieurs lettres minuscules suivies d'un '\0'.
Postcondition : result contient la version cryptée de texte jusqu'au '\0' exclu, puis un '\0'. Seules les lettres (majuscules ou minuscules) non accentuées sont cryptées, les autres caractères sont recopiés tels quels. Les éventuels caractères situés derrière le '\0' sont ignorés.
Paramètres en mode donnée : texte, cle
Paramètre en mode résultat : result

```
Procédure crypter (texte : tableau de caractères, cle : tableau de caractères, result : tableau de caractères)
Préconditions : cf énoncé
Postcondition : cf énoncé
Paramètres en mode donnée : texte, cle
Paramètre en mode résultat :  result
Variables locales : i, j ; entiers
Début
 i ← 1
 j ← 1

 Tant que (texte[i] ≠ '\0') Faire

    Si (texte[i] >= 'a') et (texte[i] <= 'z')  Alors   {lettre minuscule}
          result[i] ←  ((texte[i] + cle[j] - 2*'a') mod 26) + 'a'
          j ← j +1
          Si (cle[j] = '\0') Alors j ←  1 FinSi

    Sinon
        Si (texte[i] >= 'A') et (texte[i] <= 'Z') Alors  {lettre majuscule}
            result[i] ←  ((texte[i] - 'A' + cle[j] - 'a') mod 26) + 'A'
            j ← j + 1
            Si (cle[j] = '\0') Alors j ←  1 FinSi

        Sinon  {espace, lettre accentuée, ponctuation…}
            result[i] ←  texte[i];
         FinSi

    i ← i + 1
 FinTantQue

 result[i] ←  '\0'

Fin crypter
```

c) Quel serait le prototype de cette procédure en C ?

```
void crypter(const char * texte, const char * cle, char * result) ;
```

Remarque : Lors de l'appel, il faudra veiller à ce que l'espace nécessaire pour stocker tout le résultat soit alloué :
- soit la pile ne contient que les 4 octets du pointeur mais celui pointe sur un segment réservé dans le tas (= on a fait un malloc avant d'appeler crypter),
- soit la pile contient carrément tout le tableau (= on a déclaré un tableau statique).

Exercice 3 : Tri par insertion

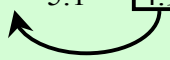
Le tri par insertion est l'algorithme utilisé par la plupart des joueurs lorsqu'ils trient leur « main » de cartes à jouer. Le principe consiste à prendre le premier élément du sous-tableau non trié et à l'insérer à sa place dans la partie triée du tableau.

a) Dérouler le tri par insertion du tableau {5.1, 2.4, 4.9, 6.8, 1.1, 3.0}.

5.1 **2.4** 4.9 6.8 1.1 3.0

On commence par essayer de placer l'élément 2, soit 2.4, dans le sous-tableau allant des cases 1 à 1. Comme 5.1 est supérieur à 2.4, on doit placer 2.4 avant 5.1. On passe ensuite à l'élément 3, c'est-à-dire 4.9.

2.4 5.1 **4.9** 6.8 1.1 3.0

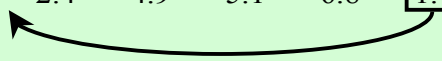


4.9 est inférieur à 5.1 mais supérieur à 2.4, on le place donc entre les deux. On passe ensuite à l'élément 4, soit 6.8.

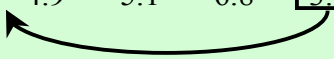
2.4 4.9 5.1 **6.8** 1.1 3.0

6.8 ne bouge pas car il est supérieur à 5.1. On passe à 1.1.

2.4 4.9 5.1 6.8 **1.1** 3.0



1.1 2.4 4.9 5.1 6.8 **3.0**



1.1 2.4 3.0 4.9 5.1 6.8 Le tableau est trié.

Le principe est simple à comprendre, mais la difficulté apparaît lorsqu'on essaie d'écrire l'algorithme avec des données structurées en tableau : il va parfois falloir déplacer plusieurs éléments pour en placer un, car il faut lui « faire de la place ».

b) Ecrire le corps de la procédure de tri par insertion, par ordre croissant, d'un tableau de réels :

Procédure tri_par_insertion (tab : tableau [1..n] de réels)
Précondition : tab[1], tab[2], … tab[n] initialisés
Postcondition : tab[1] ≤ tab[2] ≤ … ≤ tab[n]
Paramètres en mode donnée : aucun
Paramètre en mode donnée-résultat : tab

```
Procédure tri_par_insertion (tab : tableau [1..n] de réels)
Précondition : tab[1], tab[2], … tab[n] initialisés
Postcondition : tab[1] ≤ tab[2] ≤ … ≤ tab[n]
Paramètres en mode donnée : aucun
Paramètre en mode donnée-résultat : tab
Variables locales :
    i, j : entiers
    elt_a_placer : reel
Début
    Pour j allant de 2 à n par pas de 1 Faire
        elt_a_placer  ← tab[ j ]
        i ← j – 1
        Tant que (i > 0)  et (tab[i] > elt_a_placer) Faire
            tab[i+1] ← tab[i]   {on pousse l'élément i vers la droite}
```

```
            i ← i – 1
        FinTantQue
        tab[i+1] ← elt_a_placer
    FinPour
Fin tri_par_insertion
```

c) Donner l'invariant de boucle correspondant à cet algorithme, en démontrant qu'il vérifie bien les 3 propriétés d'un invariant de boucle : initialisation, conservation, terminaison.

Invariant de boucle : Juste avant l'itération j, le sous-tableau tab[1..(j-1)] se compose des éléments qui occupaient initialement les positions tab[1..(j-1)], mais qui sont maintenant triés.

Initialisation : Il faut démontrer que la propriété est vraie juste avant l'itération j=2. Il faut donc montrer que « le sous-tableau tab[1..1] se compose des éléments qui occupaient initialement les positions tab[1..1], mais qui sont maintenant triés. » Avant l'itération j=2, on n'a pas modifié le tableau, donc l'élément tab[1] est bien l'élément tab[1] originel. Par ailleurs, un sous-tableau qui ne contient qu'un seul élément est forcément trié. La propriété est donc vérifiée.

Conservation : Il faut démontrer que si la propriété est vraie juste avant l'itération j, alors elle reste vraie juste avant l'itération j+1. On suppose donc que « le sous-tableau tab[1..(j-1)] se compose des éléments qui occupaient initialement les positions tab[1..(j-1)], mais qui sont maintenant triés », et on doit montrer qu'après avoir exécuté le corps de la boucle Pour, « le sous-tableau tab[1..j] se compose des éléments qui occupaient initialement les positions tab[1..j], mais qui sont maintenant triés. » On sait que l'élément tab[j] va être inséré quelque part entre les positions 1 et j incluses, mais qu'il ne sera en aucun cas placé après la case j. Les éléments des cases 1 à j-1 peuvent être déplacés vers la droite, mais d'un cran seulement : aucun d'entre eux ne se retrouvera après la case j. Donc les éléments situés initialement dans les cases 1 à j restent bien dans ce bloc de cases. Par ailleurs, l'élément j va être bien être inséré de sorte à ce que le sous-tableau tab[1..j] reste trié. On a donc bien conservation de la propriété.

Terminaison : Il faut montrer qu'une fois la boucle terminée, l'invariant de boucle fournit une propriété utile pour montrer la validité de l'algorithme. Ici, la boucle prend fin quand j dépasse n, c'est-à-dire pour j=n+1. On sait alors que « le sous-tableau tab[1..n] se compose des éléments qui occupaient initialement les positions tab[1..n], mais qui sont maintenant triés. » Le tableau doit entier est donc trié, ce qui montre que l'algorithme est correct.

d) **Question à faire chez soi, pour le prochain TD** : Evaluer le nombre de comparaisons de réels et le nombre d'affectations de réels pour un tableau de taille *n*, dans le cas le plus défavorable (tableau trié dans l'ordre décroissant). Cet algorithme est-il meilleur que le tri par sélection (ou tri du minimum), vu en cours magistral ?

Affectations de réels : Dans le cas le plus défavorable, on fait (j-1) décalages avant d'insérer l'élément.

$$A = \sum_{j=2}^{n} \left(1 + \left(j-1\right) + 1\right)$$

$$A = \sum_{j=2}^{n} (1 + j)$$

$$A = n - 1 + \sum_{j=2}^{n} j$$

$$A = n - 1 + \frac{n(n+1)}{2} - 1$$

$$A = \frac{n^2}{2} + \frac{3n}{2} - 2$$

Le nombre d'affectations est donc $\mathcal{O}(n^2)$ alors qu'il était $\mathcal{O}(n)$ pour le tri du minimum.

Comparaisons de réels : Dans le cas le plus défavorable, on fait (j-1) comparaisons de réels avant d'insérer l'élément (on va de i==j-1 jusqu'à i==0, mais pour i==0 on ne fait pas la comparaison de réels, puisque i>0 est faux).

$$C = \sum_{j=2}^{n} (j - 1)$$

$$C = \sum_{j=2}^{n} (j) - (n - 1)$$

$$C = \frac{n(n+1)}{2} - 1 - (n - 1)$$

$$C = \frac{n^2}{2} - \frac{n}{2}$$

On retrouve le même nombre de comparaisons que pour le tri du minimum.