Programmation C (E. Lazard)
Examen du 29 janvier 2015

CORRECTION

(durée 2h)

I. Expressions

1. Donnez les résultats affichés par le programme suivant :

Listing 1. ***Expressions***

```
#include <stdio.h>

int tab[3] = {7, 5, 3};

int main() {
    int *i = tab;
    printf("%d\n", ++(*i));
    printf("%d\n", (*(++i))++);
    printf("%d\n", ++(*(i++)));
    printf("%d\n", *(i++));
    printf("%d, %d, %d\n", tab[0], tab[1], tab[2]);
    return 0;
}
```

2. Les résultats sont-ils identiques si on remplace les quatre premiers `printf()` par la seule ligne ci-dessous ? *(Justifiez votre réponse)*

```
printf("%d\n%d\n%d\n%d\n", ++(*i), (*(++i))++, ++(*(i++)), *(i++));
```

3. Peut-on supprimer la variable `i` et la remplacer directement dans les premiers `printf()` par `tab` ? *(Justifiez votre réponse)*

Corrigé :

1. – `++(*i)` incrémente la case pointée (`tab[0]`) puis renvoie la valeur pointée par `i` (`tab[0]` qui vaut donc maintenant 8).
   – `(*(++i))++` incrémente d'abord `i` puis renvoie la valeur pointée par `i` (donc `tab[1]` qui vaut 5) puis incrémente cette case pointée (qui passe à 6).
   – `++(*(i++))` l'expression interne indique la case pointée qui est incrémentée puis renvoyée (On a donc la valeur 7 qui est dans `tab[1]`) avant d'incrémenter `i` (qui pointe sur `tab[2]`).
   – `*(i++)` renvoie la valeur pointée par `i` (donc 3) puis incrémente `i`.

   Ce programme affiche donc :

```
8
5
7
3
8, 7, 3
```

2. Non, car l'ordre d'évaluation des 4 expressions n'est pas de gauche à droite. Le compilateur peut les calculer dans l'ordre qu'il souhaite. On ne peut pas savoir sur quoi pointe `i` à chaque calcul d'expression.

3. Non, car `tab` étant un pointeur de tableau, il ne peut pas être modifié, ce que fait `++tab` et `tab++`. Il y aura donc une erreur à la compilation.

II. Calculs flottants

On souhaite écrire une fonction ayant pour but de calculer un zéro d'une fonction donnée (c'est-à-dire un x tel que $f(x) = 0$) à partir d'un encadrement de ce nombre.

Soit

```
float f(float);
```

une fonction réelle, continue, dont le code est déjà écrit et disponible. Écrire une fonction

```
float zero(float xMin, float xMax, float epsilon);
```

qui trouve un zéro de la fonction `f` donnée situé dans l'intervalle $[xMin, xMax]$ avec une précision `epsilon`. On supposera que $f(xMin)$ et $f(xMax)$ sont de signes contraires et différents de zéro (la fonction étant continue, cela garantit l'existence d'un zéro dans l'intervalle).

L'algorithme fonctionnera par dichotomie : tant que la largeur de l'intervalle $[xMin, xMax]$ est plus grand qu'epsilon, on calculera la valeur de la fonction au milieu de l'intervalle (par un appel à la fonction `f()` supposée définie ailleurs dans le code). Suivant le signe du résultat, on gardera la borne inférieure ou la borne supérieure et le point milieu deviendra l'autre borne. Pour finir, la fonction renverra une approximation du zéro de la fonction à epsilon près.

Corrigé :

Listing 2. ***zéro***

```
float zero(float xMin, float xMax, float epsilon) {
    float pointMilieu, valeurMilieu;
    while ((xMax - xMin) > epsilon) {
        pointMilieu = (xMax + xMin) / 2;
        valeurMilieu = f(pointMilieu);
        if (valeurMilieu == 0)
            return pointMilieu;
        if ((f(xMin) * valeurMilieu) < 0)
            xMax = pointMilieu;
        else
            xMin = pointMilieu;
    }
    return xMin;
}
```

III. Chaînes

On souhaite écrire quelques fonctions permettant le chiffrement par substitution d'une chaîne de caractères. On supposera par la suite que la chaîne de départ, constituant le message que l'on veut chiffrer, n'est composée que de lettres (majuscules et minuscules), d'espaces et de sauts de ligne (caractère `'\n'`). Le principe du chiffrement est le suivant :

- la chaîne est tout d'abord transformée en y supprimant tous les espaces et sauts de ligne (par exemple `"Mon cher Watson\n"` devient `"MoncherWatson"`) ;
- tous les caractères de la chaîne sont ensuite basculés en majuscule (par exemple `"MoncherWatson"` devient `"MONCHERWATSON"`) ;
- le chiffrement se fait à l'aide d'une clef (qui est une chaîne uniquement composée de lettres majuscules). On parcourt simultanément les caractères du message et de la clef. Chaque caractère du message est alors décalé dans l'alphabet de la valeur correspondant au caractère courant de la clef (`'A'` vaut 0, `'B'` vaut 1...).

  Si le décalage imposé à un caractère du message dépasse `Z`, on reboucle sur `A`. Lorsqu'on arrive à la fin de la clef, on reboucle au début.

  Ainsi, le message `"DAUPHINEXYZ"` codé à l'aide de la clef `"FAC"` devient `"IAWUHKSEZDZ"` : `F` décale la première lettre (`'D'`) de 5 places, `A` ne décale rien, `C` décale de 2 (donc `'U'` donne `'W'`) puis on recommence avec `F`... Notez que le `'Y'` de la fin se décale de 5 positions (il tombe en face du `F` de la clef) et reboucle pour donner `'D'`.

On ne pourra pas utiliser de fonctions déclarées dans `string.h` à l'exception de `strlen()`.

1. Écrire une fonction

   ```
   char *removeSpaces(char *str);
   ```

   qui renvoie une **nouvelle** chaîne, copie de `str` dans laquelle les espaces et les sauts de ligne ont été supprimés. On pourra supposer que `str` pointe bien sur une chaîne.
2. Écrire une fonction

   ```
   char *myToUpper(char *str);
   ```

   qui renvoie une **nouvelle** chaîne, copie de `str` dans laquelle toutes les lettres minuscules auront été converties en majuscule. On pourra supposer que `str` pointe bien sur une chaîne.
3. Écrire une fonction

   ```
   char *crypto(char *msg, char *clef);
   ```

   qui chiffre la chaîne `msg` avec la clef passée en paramètre suivant l'algorithme donné plus haut et renvoie une **nouvelle** chaîne composant le message chiffré. On pourra supposer que les deux arguments sont au bon format : pointant sur une chaîne non-vide en majuscule et sans espace ni sauts de ligne.
4. Ajouter à ces trois fonctions une fonction `main()` permettant à l'utilisateur d'entrer un message et une clef. Cette fonction doit ensuite les convertir au bon format pour le chiffrement et afficher le message d'origine suivi en-dessous du message chiffré. On pourra supposer que les deux chaînes entrées par l'utilisateur ne font pas plus de 256 caractères, sont non-vides et ne sont composées que de lettres (majuscules et minuscules), d'espaces et de sauts de ligne.

Corrigé :

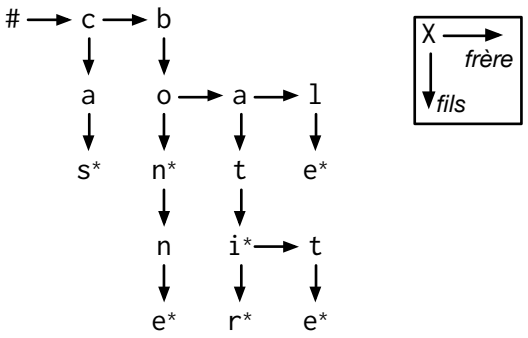
Listing 3. ***Chiffrement***

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

```c
#define TAILLE_MAX 256

char *removeSpaces(char *str) {
    char c;
    char *newStr = malloc(strlen(str)+1);
    char *p = newStr; /* pointeur pour parcourir la nouvelle chaîne */
    do {
        c = *str++;
        if (c != ' ' && c != '\n')
            *p++ = c;
    } while (c != '\0'); /* Le zéro final aura été recopié */
    return newStr;
}

char *myToUpper(char *str) {
    char c;
    char *newStr = malloc(strlen(str)+1);
    char *p = newStr;
    do {
        c = *str++;
        *p++ = (('a' <= c) && (c <= 'z')) ? c -'a' + 'A' : c;
    } while (c != '\0');
    return newStr;
}

char *crypto(char *msg, char *clef) {
    char newC;
    char *newStr = malloc(strlen(msg)+1);
    char *p = newStr;
    char *q = clef;
    while (*msg) {
        newC = (*msg++) + (*q++ - 'A'); /* On décale */
        if (newC > 'Z')
            newC -= 26; /* et on reboucle sur A si nécessaire */
        *p++ = newC;
        if (*q == '\0') /* Est-on arrivé à la fin de la clef ? */
            q = clef;   /* Si oui, on recommence au début de la clef */
    }
    *p = '\0';
    return newStr;
}

int main() {
    char msg[TAILLE_MAX];
    char clef[TAILLE_MAX];
    printf("Message ?");
    fgets(msg, TAILLE_MAX, stdin);
    printf("Clef ?");
    fgets(clef, TAILLE_MAX, stdin);

    char *correctKey = myToUpper(removeSpaces(clef));
    char *correctMsg = myToUpper(removeSpaces(msg));
    printf("%s%s\n", msg, crypto(correctMsg, correctKey));
    return 0;
}
```

IV. Arbre dictionnaire *(Nettement plus difficile !)*

Le but est d'implémenter un dictionnaire (c'est-à-dire une liste de mots) sous forme d'arbre. Chaque nœud de l'arbre représente une lettre. De chaque nœud part un lien `fils` qui correspond à la lettre suivante dans un mot et un lien `frere` qui correspond à une autre possibilité de lettre au même niveau que le nœud courant. La racine de l'arbre sera représentée par une lettre n'existant pas dans les mots, par exemple `'#'`. Voici par exemple l'arbre stockant les mots `cas`, `bon`, `bati`, `batir`, `batte`, `ble`, `bonne`.



Chaque nœud est représenté par la cellule suivante :

```
struct cell {
    char lettre;
    int final;
    struct cell *frere;
    struct cell *fils;
};
```

dans laquelle le champ final indique que la lettre finit un mot (il vaut 1 dans ce cas, représenté par un astérisque dans la figure ci-dessus, et 0 sinon).

Indication : pour toutes les fonctions demandées, on pourra supposer que le dictionnaire général existe déjà avec au moins un nœud, la racine `'#'`. Les algorithmes à envisager s'écrivent assez bien de manière récursive ; pour `ajouterfils()`, la version non-récursive est aussi facilement exprimable.

1. Écrire une fonction

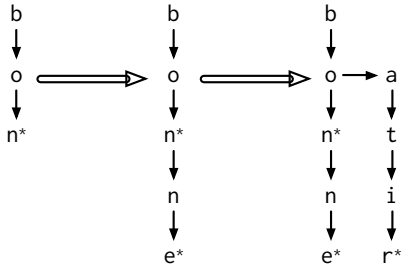
   ```
   int cherche(struct cell *racine, char *mot);
   ```

   qui cherche le mot indiquée par la chaîne `mot` dans le dictionnaire pointé par la racine passée en argument. Elle renvoie 1 si le mot est trouvé et 0 sinon.

2. Écrire une fonction

   ```
   void ajouterFils(struct cell *noeud, char *mot);
   ```

   qui ajoute, comme *nœud fils*, la suite de lettres donnée par `mot` au noeud indiqué ; on supposera que celui-ci n'a pas encore de fils. Ainsi, l'ajout du mot `"ne"` comme fils du nœud `'n'` terminant le mot `"bon"` donne la flèche de gauche de la figure ci-dessous.



3. Écrire une fonction

   ```
   void ajouterFrere(struct cell *noeud, char *mot);
   ```

qui ajoute, comme *nœud frère*, la suite de lettres donnée par `mot` au noeud indiqué ; on supposera que celui-ci n'a pas encore de frère. Ainsi, l'ajout du mot `"atir"` comme frère du nœud `'o'` donne la flèche de droite de la figure ci-dessus.

4. Écrire une fonction

   ```
   void ajouter(struct cell *racine, char *mot);
   ```

   qui ajoute un mot indiqué dans le dictionnaire de racine indiquée.

Corrigé :

Listing 4. ***arbre dictionnaire***

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

struct cell {
    char lettre;
    int final;
    struct cell *frere;
    struct cell *fils;
} root = {'#', 0, NULL, NULL};

int cherche(struct cell *racine, char *mot) {
    if (racine == NULL)
        return 0;
    if (racine->lettre == mot[0]) {
        if (mot[1] != '\0')
            return cherche(racine->fils, &mot[1]);
        else
            return racine->final;
    } else {
        if (racine->frere != NULL)
            return cherche(racine->frere, mot);
        else
            return 0;
    }
}

void ajouterFils(struct cell *noeud, char *mot) {
    int i = 0;
    while (mot[i] != '\0') {
        struct cell *p = malloc(sizeof(struct cell));
        p->lettre = mot[i++];
        p->final = 0;
        p->frere = p->fils = NULL;
        noeud->fils = p;
        noeud = p;
    }
    noeud->final = 1;
}

/* Ou la version récursive */
void ajouterFils(struct cell *noeud, char *mot) {
    if (mot[0] == '\0') {
        noeud->final = 1;
        return;
    }
    struct cell *p = malloc(sizeof(struct cell));
```

```
    p->lettre = mot[0];
    p->final = 0;
    p->frere = p->fils = NULL;
    noeud->fils = p;
    ajouterFils(p, &mot[1]);
}

void ajouterFrere(struct cell *noeud, char *mot) {
    /* On ajoute un noeud frère */
    struct cell *p = malloc(sizeof(struct cell));
    p->lettre = mot[0];
    p->final = 0;
    p->frere = p->fils = NULL;
    noeud->frere = p;
    /* puis tous les fils */
    ajouterFils(p, &mot[1]);
}

void ajouter(struct cell *racine, char *mot) {
    if (racine->lettre == mot[0]) {
        if (racine->fils != NULL)
            ajouter(racine->fils, &mot[1]);
        else
            ajouterFils(racine, &mot[1]);
    } else {
        if (racine->frere != NULL)
            ajouter(racine->frere, mot);
        else
            ajouterFrere(racine, mot);
    }
}
```
