Programmation C (E. Lazard)
Examen du 25 janvier 2011

CORRECTION

(durée 2h)

I. Expressions

Donnez les résultats affichés par le programme suivant :

```
void main() {
    int a=9, b=7, c=0, d=3, e=-1, f=-2, y;
    a -= (b++); printf( "a = %d\n", a);
    b /= (d-1); printf( "b = %d\n", b);
    c = (e == f++); printf( "c = %d\n", c);
    y = (d << 2) & 5; printf( "y = %d\n", y);
}
```

Corrigé :

```
a = 2
b = 4
c = 0
y = 4
```

II. Chaînes de caractères

Le but de cet exercice est de réécrire un certain nombre de fonctions de traitement de chaînes de caractères semblables à celles disponibles dans la bibliothèque `string`. *On ne pourra donc pas utiliser de fonctions déclarées dans* `string.h` *à l'exception de* `strlen()`.

Toutes les questions sont indépendantes.

1. Écrivez une fonction

   `char *reverseStr(char *str);`

   qui renvoie une nouvelle chaîne de caractères égale à la chaîne passée en argument mais renversée. Par exemple, `reverseStr("Baobab");` renvoie `"baboaB"`.

2. Écrivez une fonction

   `int isSubChar(char *str, char *subChar);`

   qui renvoie 1 s'il y a au moins un des caractères de la seconde chaîne (`subChar`) qui se trouve dans la première et 0 sinon. Par exemple, `isSubChar("Baobab", "Toto");` renvoie 1 (le `'o'` est dans la première) alors que `isSubChar("Baobab", "Office");` renvoie 0.

3. Écrivez une fonction

   `int isSubstring(char *str, char *subStr);`

   qui teste si une chaîne donnée en second argument (`subStr`) est une sous-chaîne de celle donnée en premier argument (`str`). Elle renvoie 1 en cas de succès et 0 sinon. Si une des deux chaînes n'existe pas, la fonction renvoie 0 ; une chaîne vide est sous-chaîne de n'importe quelle autre chaîne. *L'algorithme peut s'écrire comme une double boucle : la boucle extérieure parcourt tous les caractères de la première chaîne, tandis que la boucle intérieure compare tous ceux de la seconde.*

Corrigé :

```
char *reverseStr(char *str) {
    int i;
    char *s, *tmp;

    if (str == NULL)
        return NULL;
    i = strlen(str);
    tmp = s = malloc(sizeof(char)*(i+1));
    str += i; /* aller à la fin */
    for (; i>0; i--)
        *tmp++ = *--str; /* recopie */
    *tmp = '\0';
    return s;
}

int isSubChar(char *str, char *subChar) {
    char *q;

    if ((str == NULL) || (subChar == NULL) || (strlen(subChar) == 0))
        return 0;
    while (*str) {
        q = subChar;
        while (*q) {
            if (*str == *q++)
                return 1;
        }
        str++;
    }
    return 0;
}

int isSubstring(char *str, char *subStr) {
    char *p, *q;

    if ((str == NULL) || (subStr == NULL))
        return 0;
    if (strlen(subStr) == 0)
        return 1;
    while (*str) {
        p = str;
        q = subStr;
        while (*q) {
            if (*p == *q) {
                p++; q++; /* on parcourt tant que les chars sont égaux */
            } else
                break;
        }
        if (!*q)
            return 1; /* fin 2e chaîne ? */
        str++;
    }
    return 0;
}
```

III. Polynômes

On souhaite gérer des polynômes réels de degré inférieur ou égal à 10 en utilisant la structure suivante :

```
typedef struct {
    double coeff[10];
} poly;
```

qui servira à représenter le polynôme $\Sigma_{i=0}^{9} a_i x^i$ en initialisant `coeff[i]` à a_i.

1. Écrivez une fonction

   `poly *saisie(void);`

   qui crée un polynôme et permet à l'utilisateur de saisir un par un ses dix coefficients. *On pourra utiliser le format* `"%lf"` *pour indiquer un* `double`.

2. Écrivez une fonction

   `void affiche(poly *P);`

   qui affiche à l'écran le polynôme sous forme de chaîne habituelle. On n'affichera pas les termes nuls. *On pourra utiliser le format* `"%+lf"` *pour forcer l'affichage du signe.* Voici un exemple d'affichage (le nombre de décimales de chaque coefficient n'est pas important) :

   `+1.0x^5+2.0x^3-1.0x^2+3.5x^1+2.0x^0`

3. Écrivez une fonction

   `double eval(poly *P, double x);`

   qui calcule la valeur du polynôme `P` au point `x`. On pourra se servir de la formule de Horner :

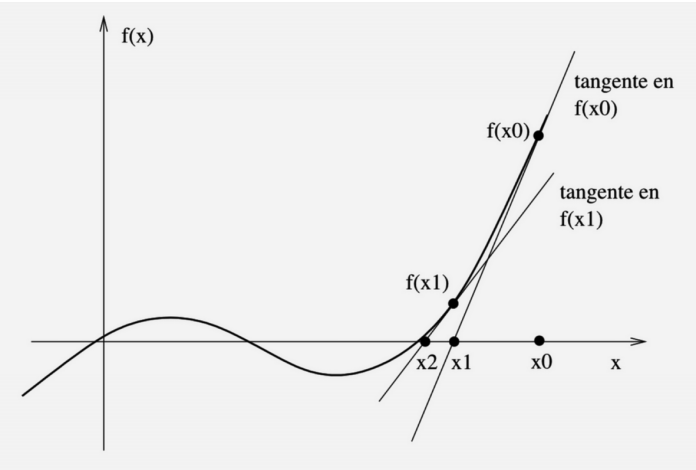
   $$P(x) = ((\ldots((a_9 x + a_8)x + a_7)x + \ldots)x + a_1)x + a_0$$

4. Écrivez une fonction

   `poly *derivee(poly *P);`

   qui renvoie le polynôme exprimant la dérivée du polynôme `P`.
   On rappelle que si $P(x) = \Sigma_{i=0} a_i x^i$, la dérivée de P est $P'(x) = \Sigma_{i=0}(i+1)a_{i+1}x^i$.

5. On souhaite trouver une racine d'un polynôme à l'aide de la méthode de Newton.

   Le principe est le suivant : en partant d'un point initial, on construit une suite de points qui se rapprochent de la racine en calculant à chaque étape un nouveau point à partir de la valeur du polynôme et de sa dérivée :

   $$x_{n+1} = x_n - \frac{P(x_n)}{P'(x_n)}$$

   Écrivez une fonction

   `double racine(poly *P);`

   qui :
   - demande à l'utilisateur un point initial x_0;
   - effectue une suite d'itérations calculant à chaque fois $x_n - \frac{P(x_n)}{P'(x_n)}$;
   - s'arrête lorsque $|P(x_n)| \leq 0.001$ (on utilisera la fonction `fabs(double)` pour la valeur absolue).

Pour simplifier, on pourra supposer que le calcul converge toujours vers une racine.

Corrigé :

```
typedef struct {
    double coeff[10];
} poly;

poly *saisie(void) {
    int i;
    poly *P = malloc(sizeof(poly));

    for (i=9; i>=0; i--) {
        printf("Coeff du degre %d ?", i);
        scanf("%lf", &P->coeff[i]);
    }
    return P;
}

void affiche(poly *P) {
    int i;

    if (P == NULL)
        return;
    for (i=9; i>=0; i--) {
        if (P->coeff[i] != 0)
            printf("%+1.1fx^%d", P->coeff[i], i);
    }
    printf("\n");
}

double eval(poly *P, double x) {
    double val;
    int i;

    if (P == NULL)
        return 0;
    val = P->coeff[9];
    for (i=8; i>=0; i--)
     val = val*x + P->coeff[i];
    return val;
}

poly *derivee(poly *P) {
    poly *DP;
    int i;

    if (P == NULL)
        return NULL;
    DP = malloc(sizeof(poly));
    DP->coeff[9] = 0;
    for (i=8; i>=0 ; i--)
        DP->coeff[i] = (i+1) * P->coeff[i+1];
    return DP;
}
```

```
double racine(poly *P) {
    double x, val;
    poly *DP;

    if (P == NULL)
        return 0;
    DP = derivee(P);

    printf("Entrez la valeur initiale : ");
    scanf("%lf", &x);
    while (fabs(val = eval(P, x)) > 0.0001)
        x -= val/eval(DP, x);
    return x;
}
```

IV. Tableau dynamique

On souhaite créer une structure de tableau dynamique d'entiers, semblable à une `ArrayList`*. Il s'agit de permettre au programmeur d'avoir un tableau d'entiers dont la taille peut s'accroître au fur et à mesure de l'utilisation. Pour ce faire, on définit une structure contenant la taille actuelle du tableau ainsi qu'un pointeur vers la zone mémoire de stockage :*

```
typedef struct {
    int taille;
    int *zone;
} tabDyn;
```

1. Écrivez une fonction

   ```
   tabDyn *initDynTab(int tailleInitiale);
   ```

   qui crée un tel tableau (et renvoie son adresse) ayant la taille initiale demandée en argument.
2. Écrivez une fonction

   ```
   int get(tabDyn *ptr, unsigned int index);
   ```

   permettant de récupérer la valeur de l'élément d'index indiqué. Si on essaie d'accéder à un élément inexistant, la fonction affiche un message d'erreur et renvoie 0.
3. Écrivez une fonction

   ```
   void put(tabDyn *ptr, unsigned int index, int valeur);
   ```

   qui initialise un élément du tableau. *Bien évidemment, c'est tout l'intérêt d'un tableau dynamique, le programmeur a la possibilité d'agrandir le tableau en indiquant un index plus grand que la taille actuelle du tableau (on supposera qu'il y a toujours assez de mémoire pour le faire).*
4. Écrivez une fonction

   ```
   void delete(tabDyn *ptr);
   ```

   supprimant un tableau dynamique.

Corrigé :

```
typedef struct {
    int taille;
    int *zone;
} tabDyn;

tabDyn *initDynTab(int tailleInitiale) {
    if (tailleInitiale <= 0)
        return NULL;
    tabDyn *p = malloc(sizeof(tabDyn));
    p->taille = tailleInitiale;
    p->zone = malloc(sizeof(int) * tailleInitiale);
    return p;
}

int get(tabDyn *ptr, unsigned int index) {
    if ((ptr != NULL) && (index < ptr->taille)) {
        return ptr->zone[index];
    } else {
        printf("Erreur d'acces\n");
        return 0;
    }
}

void put(tabDyn *ptr, unsigned int index, int valeur) {
    unsigned int i;

    if (ptr == NULL)
        return;
    if (index >= ptr->taille) {
        int *tmp = malloc((index+1) * sizeof(int));
        for (i=0; i<ptr->taille; i++)
            tmp[i] = ptr->zone[i]; /* recopie */
        free(ptr->zone);
        ptr->taille = index+1;
        ptr->zone = tmp;
    /* On peut aussi utiliser la fonction realloc(ptr)      */
    /* qui alloue une nouvelle zone en recopiant l'ancienne */
    }
    ptr->zone[index] = valeur;
}

void delete(tabDyn *ptr) {
    if (ptr != NULL) {
        free(ptr->zone);
        free(ptr);
    }
}
```