Programmation C (E. Lazard)
Examen du 29 janvier 2010

CORRECTION

(durée 2h)

I. Expressions

1. Donnez les résultats affichés par le programme suivant :

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

int valeurs[3] = {2, 4, 12};

main() {
    int *p = valeurs;
    printf("%d\n", ++(*(p++)));
    printf("%d\n", ++(*p));
    printf("%d\n", *(++p));
    printf("%d\n", (*p)++);
}
```

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

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

3. Peut-on supprimer la variable `p` et la remplacer directement dans les `printf()` par `valeurs` ?

Corrigé :

1. – `++(*(p++))` incrémente la case mémoire pointée par `p++` PUIS la renvoie ; `p++` incrémente simplement le pointeur (qui pointe maintenant sur `valeurs[1]`) après avoir renvoyé sa valeur.
   – `++(*p)` incrémente d'abord la case mémoire pointée par `p` PUIS la renvoie.
   – `*(++p)` incrémente d'abord `p` (qui pointe sur `valeurs[2]`) puis renvoie la valeur pointée.
   – `(*p)++` renvoie la valeur pointée par `p` PUIS incrémente la case mémoire pointée (qui passe à 13).

   Ce programme affiche donc :

```
3
5
12
12
```

2. Non car l'ordre d'évaluation des quatre expressions n'est pas de gauche à droite. Le compilateur a loisir de les calculer dans l'ordre qu'il souhaite. On ne peut donc pas savoir sur quoi pointe `p` à chaque calcul d'expression.
3. Non car `valeurs` étant un pointeur de tableau, il ne peut pas être modifié, ce qu'essaie de faire `++valeurs` et `valeurs++`. Il y aura donc une erreur à la compilation.

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 ET on pourra utiliser, dans une fonction, une fonction définie antérieurement (ainsi, si on le souhaite, on pourra utiliser `myStrchr()` *et* `myReplaceChar()` *dans le code de la fonction* `isPalindrome()`*).*

1. Écrivez une fonction

   ```
   char *myStrchr(char *str, char c);
   ```

   qui renvoie un pointeur sur la première occurrence du caractère `c` dans la chaîne `str` ou `NULL` si le caractère ne se trouve pas dans la chaîne.

   Par exemple `myStrchr("Unix is great", 'i');` renvoie l'adresse du premier `'i'` dans la chaîne.

2. Écrivez une fonction

   ```
   char *myReplaceChar(char *str, char ch1, char ch2);
   ```

   qui renvoie une **nouvelle** chaîne, copie de `str`, dans laquelle toutes les occurrences du caractère `ch1` ont été remplacées par le caractère `ch2`.

   Par exemple `myReplaceChar("Unix is great", 'i', 'z');` renvoie `"Unzx zs great"`.

3. Écrivez une fonction

   ```
   int isPalindrome(char *str);
   ```

   qui renvoie 1 si la chaîne passée en argument est un palindrome (lue à l'envers, ses caractères sont identiques) et 0 sinon.

   Par exemple `isPalindrome("ici");` renvoie 1 tandis que `isPalindrome("baba");` renvoie 0.

4. Écrivez une fonction

   ```
   char *myExtractSubstring(char *str, int debut, int fin);
   ```

   qui renvoie la sous-chaîne de `str` commençant à l'indice `debut` et dont le dernier élément est à l'indice `fin-1` dans la chaîne initiale.

   Par exemple `myExtractSubstring("Unix is great", 2, 6);` renvoie `"ix i"`.

5. Écrivez une fonction

   ```
   char *myStrtok(char *str, char delim);
   ```

   qui découpe la chaîne `str` en *tokens*, c'est-à-dire en mots, séparés par le caractère `delim`. Le premier appel à `myStrtok()` se fait avec une chaîne `str` non-nulle et renvoie le premier token ; les appels successifs se font avec une chaîne nulle passée en paramètre (pour ne pas réinitialiser la découpe) et renvoient à chaque fois le token suivant ; une fois le dernier token renvoyé, les appels suivants renvoient `NULL`.

   Par exemple la suite d'appels :

   ```
   myStrtok("Unix is great.", ' ');
   myStrtok(NULL, ' ');
   myStrtok(NULL, ' ');
   myStrtok(NULL, ' ');
   ```

   vont renvoyer `"Unix"`, `"is"`, `"great."` puis `NULL`.

   Indication : vous pouvez utiliser une variable déclarée `static`*, locale à cette fonction, qui gardera la copie de* `str` *entre chaque appel.*

Corrigé :

1. ```
   char *myStrchr(char *str, char c) {
       if (str == NULL)
           return NULL;
       while (*str && *str != c)
           str++;
       return (*str ? str : NULL);
   }
   ```
2. ```
   char *myReplaceChar(char *str, char ch1, char ch2) {
       if (str == NULL)
           return NULL;
       char *dest = malloc(strlen(str)+1);
       char *save = dest;
       while (*str) {
           *(dest++) = ((*str == ch1) ? ch2 : *str);
           str++;
       }
       *dest = '\0';
       return save;
   }
   ```

3. ```
   int isPalindrome(char *str) {
       if (str == NULL)
           return 1;
       char *end = str + strlen(str);
       while (str < end) {
           if (*(str++) != *(--end))
               return 0;
       }
       return 1;
   }
   ```
4. ```
   char *myExtractSubstring(char *str, int debut, int fin) {
       int i;
       char *dest;

       if (str == NULL)
           return NULL;
       if (debut < 0)
           debut = 0;
       if (fin > strlen(str))
           fin = strlen(str);
       if (fin <= debut)
           return "";
       /* attention à bien faire le test après les deux ci-dessus */
       dest = malloc(fin-debut+1);
       for (i = debut; i < fin; i++)
           dest[i-debut] = str[i];
   ```

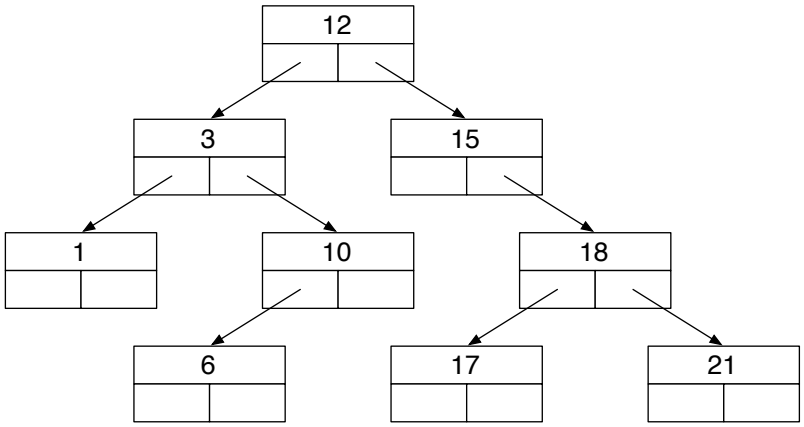
```
        dest[fin-debut] = '\0';
        return dest;
    }
```

5.
```
    char *myStrtok(char *str, char delim) {
        static char *pos = NULL;
        char *next, *tmp;

        if (str != NULL) {
            /* On ne peut pas modifier une chaîne constante.
             * On en fait donc une copie au cas où str le serait.
             * C'est l'équivalent du strdup(), ici interdit.
             */
            tmp = pos = malloc(strlen(str)+1);
            while (*str)
                *(tmp++) = *(str++);
            *tmp = '\0';
            /* On aurait aussi pu faire :
             * pos = myExtractSubstring(str, 0, strlen(str));
             */
        }
        if (pos == NULL)
            return NULL;
        next = myStrchr(pos, delim);
        if (next == NULL) { /* dernier token */
            tmp = pos;
            pos = NULL;
            return tmp;
        } else {
            *next = 0;
            tmp = pos;
            pos = next+1;
            return tmp;
        }
    }
```

III. Arbres binaires

On souhaite gérer un arbre binaire de recherche (ABR) défini de la façon suivante. Chaque nœud possède une valeur (ici entière) et deux liens vers les fils droit et gauche. La caractéristique d'un ABR est que si un nœud a pour valeur x, tous les nœuds de son sous-arbre gauche (resp. droit) ont des valeurs inférieures (resp. supérieures) à x. Pour simplifier, on supposera que toutes les valeurs sont strictement positives et toutes différentes.



1. Définissez la structure d'un nœud `struct noeud`.
2. Écrivez le code d'une fonction

   ```
   int recherche(struct noeud *root, int value);
   ```

   qui renvoie 1 s'il existe un nœud ayant la valeur indiquée dans l'arbre de racine `root` (passée en paramètre) et 0 sinon.
3. Écrivez le code de deux fonctions

   ```
   int plusGrand(struct noeud *root);
   int plusPetit(struct noeud *root);
   ```

   qui renvoient la plus grande et la plus petite valeur d'un arbre de racine `root` (et 0 si l'arbre est vide).
4. Écrivez le code d'une fonction

   ```
   struct noeud *insertion(struct noeud *root, int value);
   ```

   qui, dans un arbre de racine indiquée, insère à son emplacement correct un nouveau nœud (de la valeur indiquée) et renvoie la nouvelle racine (il y a une nouvelle racine si l'arbre de départ est vide). Pour simplifier, on supposera que la valeur à insérer n'existe pas précédemment dans l'arbre.

   Indication : on utilisera un algorithme récursif ; si l'arbre est vide, on crée un nœud qui en devient la racine, sinon on insère le nœud dans le sous-arbre (droit ou gauche) correspondant.
5. Écrivez le code d'une fonction

   ```
   void affiche(struct noeud *root);
   ```

   qui affiche les valeurs de l'arbre de racine `root`, de la plus petite à la plus grande. *(Indication : pensez là-encore à un algorithme récursif.)*

Corrigé :

```
struct noeud {
    int value;
    struct noeud *gauche;
    struct noeud *droit;
};

int recherche(struct noeud *root, int value) {
    if (root == NULL)
        return 0;
    if (root->value == value)
```

```
        return 1;
    else {
        if (root->value > value)
            return recherche(root->gauche, value);
        else
            return recherche(root->droit,  value);
    }
}

int plusGrand(struct noeud *root) {
    if (root == NULL)
        return 0;
    if (root->droit != NULL)
        return plusGrand(root->droit);
    else
        return root->value;
}

int plusPetit(struct noeud *root) {
    if (root == NULL)
        return 0;
    if (root->gauche != NULL)
        return plusPetit(root->gauche);
    else
        return root->value;
}

struct noeud *insertion(struct noeud *root, int value) {
    if (root == NULL) {
        root = malloc(sizeof(struct noeud));
        root->value = value;
        root->gauche = root->droit = NULL;
        return root;
    } else {
        if (root->value >= value)
            root->gauche = insertion(root->gauche, value);
        else
            root->droit = insertion(root->droit, value);
        return root;
    }
}

void affiche(struct noeud *root) {
    if (root == NULL)
        return;
    if (root->gauche != NULL)
        affiche(root->gauche);
```

```
    printf("%d ", root->value);
    if (root->droit != NULL)
        affiche(root->droit);
}
```

IV. Tableaux

1. *Un tableau à deux dimensions étant stocké ligne par ligne, on ne peut accéder à un élément à l'aide de ses coordonnées que si la taille des lignes est connue. Ce n'est pas le cas lorsque le tableau est passé en paramètre d'une fonction ; il faut alors également passer la taille des lignes en paramètre.*
   Écrivez une fonction
   `int acces(int *A, int taille, int ordonnee, int abscisse);`
   qui renvoie l'élément d'abscisse et d'ordonnée indiquées dans une matrice carrée *taille* × *taille* d'entiers (il s'agit donc de l'élément `A[ordonnee][abscisse]`).
2. Écrivez un programme qui effectue la multiplication de deux matrices carrées *taille* × *taille* d'entiers. La signature de la fonction est la suivante :
   `void mult(int *A, int *B, int *C, int taille);`
   et elle calcule $C = A \times B$ par la formule $c_{ij} = \sum_{k=1}^{taille} a_{ik} b_{kj}$. Les trois matrices sont stockées sous forme de tableau ligne par ligne et la place de la matrice `C` est déjà allouée.

Corrigé :

1. La seule difficulté est qu'on ne peut pas faire référence à un élément de la matrice par `C[i][j]` car le compilateur ne connaît pas la taille des lignes. Il faut donc calculer soi-même l'indice de l'élément concerné.
   ```
   int acces(int *A, int taille, int ordonnee, int abscisse) {
       return A[taille*ordonnee+abscisse];
   }
   ```
2. ```
   void mult(int *A, int *B, int *C, int taille) {
       int i, j, k;
       int somme;

       for (i=0; i<taille; i++) {
           for (j=0; j<taille; j++) {
               somme = 0;
               for (k=0; k<taille; k++)
                   somme += A[i*taille+k] * B[k*taille+j];
                   /* somme += acces(A, taille, i, k) * acces(B, taille, k, j); */
               C[i*taille+j] = somme;
           }
       }
   }
   ```
   Si la taille de la matrice est connue à la compilation (par exemple par un `#define N 10`), il est possible de modifier la signature en
   `void mult(int A[][N], int B[][N], int C[][N]);`
   et ensuite de faire référence à `C[i][j]`.
   ```