Université Paris-Dauphine **L2 MI2E**
Programmation C **2018-2019**

Examen

18 janvier 2019

(durée 2h)

Répondez directement sur le sujet.
S'il vous manque de la place, vous pouvez écrire sur votre copie.

Numéro (**à reporter obligatoirement sur votre copie cachetée**) : _____

Exercice 1

Cocher la ou les bonne(s) réponse(s)

1. L'écriture :

```
while (1) {...}
```

est une boucle infinie :

- ☐ à ne jamais utiliser ;
- ☐ qui peut être interrompue avec un `continue` ;
- ■ **qui peut être interrompue avec un `break` ou un `return` ;**
- ☐ qui doit s'écrire `while (True) {...}`.

2. Les valeurs associées aux cas d'un **`switch()`** :

- ■ **peuvent être des caractères (`case 'A':`);**
- ☐ peuvent être des réels (`case 1.5:`) ;
- ☐ peuvent être des variables (`case i:`) ;
- ☐ sont forcément des nombres entiers positifs (`case 1:`).

3. Une fonction peut être définie sans rien entre () :

```
int fct() {...}
```

- ☐ dans ce cas il faut écrire `void fct {...}` ;
- ■ **c'est légal mais une mauvaise habitude ;**
- ☐ cela signifie forcément qu'il n'y a aucun paramètre à la fonction ;
- ☐ le compilateur affichera une erreur dans tous les cas.

4. Dans l'écriture :

```
int T[5];
```

`T` est du type :

- ☐ entier (`int`) ;
- ☐ tableau structuré (`[]`) ;
- ☐ pointeur sur tableau (`[] *`) ;
- ■ **pointeur sur entier (`int *`).**

Exercice 2

Soit le programme suivant :

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

int main() {
    char str[] = "abcde";
    char *p1 = str;
    char *p2 = str + strlen(str);

    while (*(++p1)) {
        p2--;
        *p1 = (*p2)++;
    }
    printf("%s\n", str);
    return 0;
}
```

Remarque : la fonction `strlen(str)` renvoie la longueur de la chaîne passée en paramètre (c'est-à-dire le nombre de caractères avant le premier `'\0'` rencontré).

1. Ce programme compile-t-il et s'exécute-t-il sans erreurs ?

   Solution :

   Oui

2. Si non, pourquoi ?

   Si oui, qu'est-il affiché à l'exécution du programme ?

   Solution :

   `afede`

Exercice 3

Soit le programme suivant :

```
#include <stdio.h>

int a = 0;
int b = 0;
int func(int i, int *pi, int **ppi) {
    a++;
    b = ++i;
    ++(*pi);
    return ++*(++(*ppi));
}

int main() {
    int T[2] = {6, 8};
    int *q = &T[0];
    int c = func(a, T, &q);
    printf("%d %d %d %d %d\n", a, b, c, T[0], T[1]);
    return 0;
}
```

L'exécution du programme affichera :

Solution :

`1 * 1 * 9 * 7 * 9`

Exercice 4

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.

Solution :

Listing 1. ***Suppression des espaces***

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

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. *(Rappelons que `'A'` est une constante numérique égale au code ASCII de `A` ; idem pour les autres lettres bien sûr.)*

Solution :

Listing 2. ***Passage en capitales***

```
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;
}
```

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.

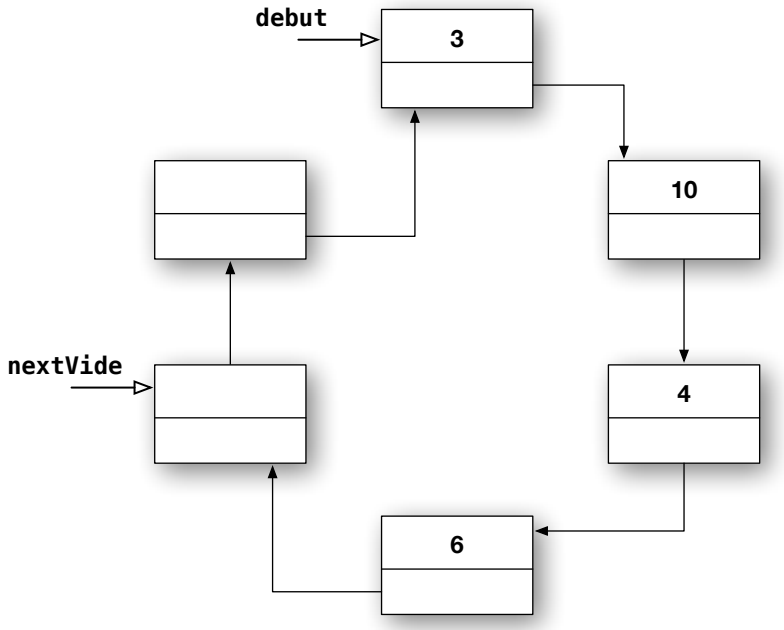
Solution :

Listing 3. ***Chiffrement***

```c
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; /* on reboucle sur A si nécessaire */
        *p++ = newC;
        if (*q == '\0') /* Est-on à la fin de la clef ?    */
            q = clef;   /* oui, retour au début de la clef */
    }
    *p = '\0';
    return newStr;
}
```

Exercice 5

On souhaite gérer une file d'attente FIFO de valeurs entières (pensez à des tickets d'attente portant un numéro dans une boutique). Avec cette file, on doit pouvoir récupérer le premier numéro de la file, et insérer un numéro en queue de file. Il est possible de gérer cette file d'attente avec une structure de liste chaînée mais cela oblige à supprimer une cellule lors du retrait d'un ticket et à en créer une lors de l'insertion, opérations qui peuvent être coûteuses en temps. On décide plutôt d'utiliser une liste circulaire où les cellules sont créées dès le départ et où deux pointeurs indiquent respectivement la première cellule occupée (le début de la file) et la première cellule vide (fin de la file) :




```
struct cell {
    int val;
    struct cell *next;
};

typedef struct cell Cell;

Cell *debut;
Cell *nextVide;
```

Récupérer le premier élément de la liste revient à prendre sa valeur puis à déplacer le pointeur `debut` ; insérer un élément revient à initialiser la cellule puis à déplacer le pointeur `nextVide`. L'avantage de cette structure est qu'elle ne nécessite aucune création de cellule mais simplement des déplacements de pointeurs.

- les deux pointeurs `debut` et `nextVide` seront gérés comme des variables globales, accessibles et modifiables directement depuis tout le code ;
- `debut` sera toujours initialisé, même si la file est vide ;
- si le pointeur `nextVide` est nul, cela signifie que la file est pleine.

1. Écrivez une fonction

```
void init(int taille);
```

qui crée une structure de file avec autant de cellules qu'indiqué et initialise les pointeurs globaux.

Solution :

Listing 4. ***Initialisation file***

```
struct cell {
    int val;
    struct cell *next;
};
typedef struct cell Cell;
Cell *debut = NULL;
Cell *nextVide = NULL;

void init(int taille) {
    Cell *p;
    int i;

    if (taille == 0) {
        debut = nextVide = NULL;
        return;
    } /* on crée une première cellule */
    p = nextVide = debut = malloc(sizeof(struct cell));
    /* puis toutes les autres */
    for (i = 1; i < taille; i++) {
        p->next = malloc(sizeof(struct cell));
        p = p->next;
    }
    /* et on termine en bouclant sur la première cellule */
    p->next = debut;
}
```

2. Écrivez une fonction

```
int estVide(void);
```

qui teste si la file est vide et renvoie 1 dans ce cas et 0 sinon.

Solution :

Listing 5. ***Test file vide***

```
int estVide(void) {
    return nextVide == debut;
}
```

3. Écrivez une fonction

```
int put(int val);
```

qui insère la valeur `val` en fin de file ; la fonction renvoie 1 si l'insertion a pu se faire et 0 sinon.

Solution :

Listing 6. ***Ajouter une valeur***

```
int put(int val) {
    if (nextVide == NULL)
        return 0; /* pas possible si la file est pleine */
    nextVide->val = val;
    nextVide = nextVide -> next;
    /* est-elle pleine maintenant ? */
    if (nextVide == debut)
        nextVide = NULL;
    return 1;
}
```

4. Écrivez une fonction

```
int getNext(int *val);
```

qui récupère la cellule en tête de file et met sa valeur dans la variable passée en paramètre par référence ; la fonction renvoie 1 si la valeur a correctement été récupérée et 0 sinon.

Solution :

Listing 7. ***Récupérer une valeur***

```
int getNext(int *val) {
    if (estVide())
        return 0;
    *val = debut->val;
    /* si la file était pleine, elle ne l'est plus */
    if (nextVide == NULL)
        nextVide = debut;
    /* on décale le pointeur */
    debut = debut->next;
    return 1;
}
```