Université Paris-Dauphine **L2 MI2E**
Programmation C **2017-2018**

Examen

19 janvier 2018

(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. Dans la définition d'une fonction :

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

`void` signifie :

- ■ **que la fonction ne renvoie rien ;**
- ☐ qu'on ne connait pas le type renvoyé ;
- ☐ que la fonction n'a pas d'argument ;
- ☐ que le compilateur ne doit pas vérifier les appels.

2. L'instruction :

```
double *p = malloc(sizeof(double));
```

- ☐ définit un pointeur `p` toujours initialisé à `NULL` ;
- ☐ définit un double `p` ;
- ■ **définit un pointeur `p` et un espace de stockage pour un double ;**
- ☐ définit un pointeur `p` non initialisé.

3. Les deux expressions :

`i++` et **`++i`**

- ☐ sont identiques à tous points de vue ;
- ■ **renvoient deux valeurs différentes ;**
- ☐ renvoient deux valeurs identiques ;
- ■ **effectuent la même opération sur `i`.**

4. Si une fonction est définie sans type de retour :

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

- ■ **Le compilateur peut afficher un avertissement ;**
- ☐ Le compilateur affichera une erreur dans tous les cas ;
- ☐ Le compilateur tiendra pour acquis que la fonction ne renvoie rien ;
- ☐ Le compilateur ne dira jamais rien ;
- ■ **Le compilateur tiendra pour acquis que la fonction renvoie un entier.**

Exercice 2

Soit le programme suivant :

```c
#include <stdio.h>

int a = 1;
int func(int *X, int *Y, int Z) {
    a = (*X)++;
    a++;
    Z++;
    *Y = *(++X);
    return Z;
}

int main() {
    int b = 2, c = 0;
    int d[2] = {3, 6};
    int e = func(d, &b, c);
    printf("%d %d %d %d %d %d\n", a, b, c, d[0], d[1], e);
    return 0;
}
```

L'exécution du programme affichera :

Solution :

```
4 * 6 * 0 * 4 * 6 * 1
```

Exercice 3

Soit le programme suivant :

```c
#include <stdio.h>

int main() {
    char str[] = "abcdef";
    char *p1 = str;
    char *p2 = str + 1;
    do {
        *(p1++) = *(++p2);
    } while (*(p2));
    printf("%s\n", str);
    return 0;
}
```

L'exécution du programme affichera :

Solution :

```
cdef
```

Exercice 4

(Les deux questions sont indépendantes)

1. On désire implémenter un algorithme de tri par sélection **récursif** sur un tableau d'entiers de taille N, le tableau et sa taille étant passés en argument. L'algorithme de tri est le suivant :
   - on cherche le plus grand des N éléments ;
   - on l'échange avec celui situé en dernière place dans le tableau ; le plus grand élément est maintenant à sa place finale ;
   - on recommence en rappelant la fonction récursivement pour trier le reste du tableau.

   Écrire une fonction

   ```
   int max(int T[], int taille);
   ```

   qui renvoie l'indice, dans le tableau, de l'élément le plus grand.

Solution :

Listing 1. ***indice de l'élément max***

```
int max(int T[], int taille) {
    int i, indice = 0, m = T[0];
    for (i = 1; i < taille; i++)
        if (T[i] > m) {
            indice = i;
            m = T[i];
        }
    return indice;
}
```

Écrire une fonction

```
void triRec(int T[], int taille);
```

qui trie par l'algorithme récursif précédent le tableau T dont la taille est passée en paramètre.

Solution :

Listing 2. ***tri récursif d'un tableau d'entiers***

```
void triRec(int T[], int taille) {
    int indice, tmp;
    if (taille == 0)
        return;
    indice = max(T, taille);
    tmp = T[taille-1];
    T[taille-1] = T[indice];
    T[indice] = tmp;
    triRec(T, taille-1);
}
```

2. On désire implémenter un algorithme de tri à bulles sur un tableau de chaînes de caractères, le tableau et sa taille N étant passés en argument. Le tri se fera par ordre lexicographique croissant (celui du dictionnaire). L'algorithme de tri est le suivant :
   - pour chaque élément d'indice i (i variant de 0 à $N-1$)
     - on le compare avec l'élément d'indice $i+1$;
     - si le premier est plus grand que le second (et donc les deux sont inversés pour le tri), on les permute ;
   - on recommence la boucle de 0 à $N-1$ tant qu'on a effectué au moins une permutation à la boucle précédente.

   Écrire une fonction

   ```
   void tri(char *T[], int taille);
   ```

   qui trie par l'algorithme précédent le tableau T dont la taille est passée en paramètre.

   On pourra utiliser la fonction `int strcmp(char *s1, char *s2);` qui permet de comparer des chaînes de caractères et renvoie 1 si `s1`>`s2`, 0 si `s1`=`s2`, et -1 si `s1`<`s2`.

   Solution :

   Listing 3. ***tri d'un tableau de chaînes***

   ```
   void tri(char *T[], int taille) {
       int i, encore;
       char *tmp;
       do {
           encore = 0;
           for (i = 0; i < taille-1; i++) {
               if (strcmp(T[i],T[i+1]) > 0) {
                   tmp = T[i];
                   T[i] = T[i+1];
                   T[i+1] = tmp;
                   encore = 1;
               }
           }
       } while (encore);
   }
   ```

Exercice 5

On souhaite gérer une liste doublement chaînée de réels (de type `double`) de telle sorte que la liste soit **toujours triée** par ordre croissant. On gèrera la liste à partir d'un pointeur sur sa première cellule (pointeur valant `NULL` si la liste est vide) ; ce pointeur sera passé en argument à chaque fonction et s'il peut être modifié par la fonction, cette dernière renverra la nouvelle adresse du début de la liste.

1. Définir la structure correspondante `struct cellule` ainsi qu'un `typedef Cellule` équivalent.

Solution :

Listing 4. ***définition de la structure***

```c
struct cellule {
    double val;
    struct cellule *pred;
    struct cellule *next;
};

typedef struct cellule Cellule;
```

2. Écrire une fonction

```c
double total(Cellule *debut);
```

qui renvoie la somme des valeurs de la liste.

Solution :

Listing 5. ***somme des valeurs***

```c
double total(Cellule *debut) {
    double total = 0;
    while (debut) {
        total += debut->val;
        debut = debut->next;
    }
    return total;
}
```

3. Écrire une fonction

```
Cellule *suppNeg(Cellule *debut);
```

qui supprime de la liste toutes les valeurs strictement négatives (et libère la mémoire).

Solution :

Comme la liste est triée, il suffit de décaler le début de la liste au premier élément positif ou nul.

Listing 6. ***suppression des négatifs***

```
Cellule *suppNeg(Cellule *debut) {
    Cellule *p;
    while (debut && debut->val < 0) {
        p = debut->next;
        free(debut);
        debut = p;
    }
    if (debut)
        debut->pred = NULL;
    return debut;
}
```

4. Écrire une fonction

```
Cellule *insere(Cellule *debut, double x);
```

qui insère une valeur `x` dans une nouvelle cellule au bon endroit de la liste.

Solution :

Listing 7. ***insertion***

```
Cellule *insere(Cellule *debut, double x) {
    Cellule *sauvegarde = debut;
    Cellule *q = malloc(sizeof(Cellule));
    q->val = x;
    q->pred = q->next = NULL;
    if (debut == NULL) {
        return q; /* liste vide */
    }
    else if (x < debut->val) {
        debut->pred = q;
        q->next = debut;
        return q; /* ajout en tete */
    } else {
        while (debut->next && debut->next->val < x)
            debut = debut->next;
        q->next = debut->next;
        q->pred = debut;
        debut->next = q;
        if (q->next != NULL)
            q->next->pred = q; /* ajout au milieu */
    }
    return sauvegarde;
}
```