**Université Paris Dauphine** **L2 MI2E**
**Programmation C** **2016-2017**

# Examen

## 11 janvier 2017

(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**

*Questions de cours (cocher la ou les bonnes réponses)*

1. Dans la définition d'une fonction :

```
void fct(int var, double b) {...}
```

`var` est :

- ☐ un type ;
- ■ **une variable locale ;**
- ■ **un argument formel.**

2. L'instruction :

```
long *pl;
```

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

3. Le test :

```
if (x=y) {...}    /* x & y sont de type long */
```

- ☐ compare les valeurs de `x` et `y` ; le test est vrai en cas d'égalité ;
- ☐ affecte la valeur de `y` à `x` ; le test est toujours vrai ;
- ■ **affecte la valeur de `y` à `x` ; le test est vrai si `y` est différent de 0.**

4. Un prototype de fonction sert :

- ■ **au compilateur à vérifier la conformité des appels à cette fonction ;**
- ☐ à vérifier que les noms de la fonction et de ses paramètres se conforment aux règles de nommage définies par le programmeur ;
- ☐ à définir pour le compilateur dans quel fichier se trouve la fonction.

**Exercice 2**

Voici quelques exemples de code avec leurs spécifications. Chaque exemple est erroné. Expliquer pourquoi et comment le corriger. (On supposera toujours que les bons `#include` ont été faits.)

1. Cette fonction crée un pointeur sur un entier, initialise cet entier à la valeur 3, additionne 2 à cette valeur et l'affiche.

```
1  void test1(void) {
2      int *a = 3;
3      *a = *a + 2;
4      printf("%d\n",*a);
5  }
```

> **Solution :**
>
> La ligne 2 déclare bien un pointeur sur un entier mais c'est le pointeur qui est initialisé à 3, pas l'entier pointé (qui n'existe pas encore).
>
> Il faut initialiser le pointeur `int *a = malloc(sizeof(int));` puis l'entier `*a = 3;`.

2. Cette fonction crée deux pointeurs et initialise leur case pointée à respectivement la valeur 2 et la valeur 3.

```
1  void test2(void) {
2      int* a,b;
3      a = malloc(sizeof(int));
4      b = malloc(sizeof(int));
5
6      if (!(a && b)) {
7          printf("Memoire pleine\n");
8          exit(-1);
9      }
10     *a = 2;
11     *b = 3;
12 }
```

> **Solution :**
>
> La ligne 2 doit en fait se lire `int *a, b;` On a donc un pointeur `a` et un *entier* `b`. Il faut écrire `int *a, *b;`

3. Cette fonction alloue un tableau de 1 000 entiers et pour `i` allant de 0 à 999 initialise le `i`$^{e}$ élément à la valeur `i`.

```
1  void test3(void) {
2      int i, *a = malloc(1000);
3
4      if (!a) {
```

```
5           printf("Memoire pleine\n");
6           exit(-1);
7       }
8       for (i = 0; i < 1000; i++)
9           *(i+a)=i;
10  }
```

**Solution :**

Le `malloc()` alloue 1 000 octets et pas la place pour 1 000 entiers...

Il faut écrire `int i, *a = malloc(1000*sizeof(int));`

4. Cette fonction initialise la zone visée par le pointeur à une valeur entrée par l'utilisateur et affiche un message si cette valeur est nulle.

```
1   void test4(void) {
2       int *a = malloc(sizeof(int));
3       scanf("%d",a);
4       if (!a)
5           printf("Valeur nulle\n");
6   }
```

**Solution :**

À la ligne 4, c'est l'entier pointé qu'il faut tester, pas le pointeur !

`if (!*a) {...}` est la bonne écriture.

**Exercice 3**

Soit le programme suivant :

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

int main() {
    char str[] = "ab\0yz!!";
    char *p1 = str;
    char *p2 = str + strlen(str);

    while (*p1) {
        *(p2++) = *(p1++);
        *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-ce que l'exécution du programme affiche-t-elle ?

> **Solution :**
> `ayz!!`

**Exercice 4**

Écrivez une fonction

```
int hexa2int(char *str);
```

qui convertit une chaîne de caractères hexadécimaux en sa valeur numérique et la renvoie. On supposera que la chaîne ne contient que les caractères `0` à `9` et `A` à `F`. On supposera également que la valeur numérique tient toujours dans un entier.

On pourra utiliser la fonction `int isdigit(char)` qui renvoie 1 si le paramètre est un caractère numérique (entre `0` et `9`) et 0 sinon.

Exemples : `"23AB"` renvoie la valeur 9131, `"FF"` renvoie 255, `"A54B"`, renvoie 42315.

**Solution :**

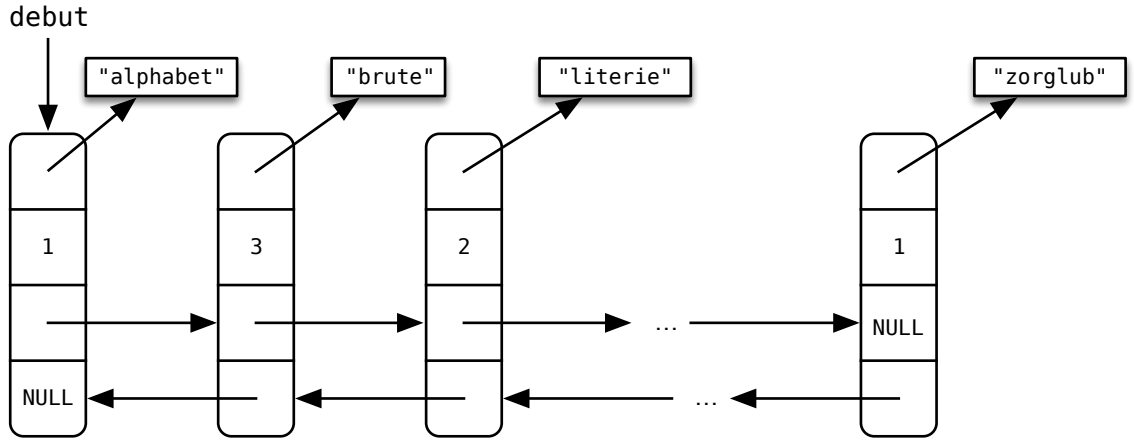
**Listing 1.** ***hexa2int***

```
int hexa2int(char *str) {
    int val = 0;
    int index = 0;
    char c;
    while (c = str[index++]) {
        if (isdigit(c))
            val = 16*val + (c - '0');
        else
            val = 16*val + (c - 'A' + 10);
    }
    return val;
}
```

**Exercice 5**

On souhaite gérer une liste doublement chaînée de chaînes de caractères de telle sorte que la liste soit toujours triée et que chaque élément contienne un compteur (qui comptera le nombre d'insertion d'une même chaîne). On gèrera la liste à partir d'un pointeur sur sa première cellule ; ce pointeur se trouvera dans une **variable globale** nommée `debut`.

Le dessin ci-dessus est un exemple après insertion de quelques chaînes (`"brute"` a été insérée trois fois, `"literie"` deux fois...).




On pourra utiliser les fonctions suivantes :

- `int strcmp(char *s1, char *s2);` permet de comparer des chaînes de caractères et renvoie 1 si `s1` >`s2`, 0 si `s1` = `s2` et −1 si `s1` <`s2` ;
- `char *strdup(char *);` permet de dupliquer une chaîne de caractères passée en argument en allouant la place mémoire nécessaire.

Dans toutes les fonctions suivantes, la liste peut être vide (et dans ce cas, le pointeur de tête `debut` vaut `NULL`). Il faudra également lorsque cela est nécessaire mettre à jour le pointeur global `debut`.

1. Définir la structure correspondante `struct cellule` ainsi que le pointeur `debut`.

**Solution :**

**Listing 2.** ***définition de la structure***

```
struct cellule {
    char *chaine ;
    int compteur;
    struct cellule *next ;
    struct cellule *pred ;
};

struct cellule *debut = NULL;
```

2. Écrire une fonction

```
struct cellule *findStr(char *str);
```

qui parcourt la liste et cherche une chaîne égale à `str`. Si une cellule ayant cette chaîne est trouvée, la fonction renvoie un pointeur sur la cellule correspondante ; dans le cas contraire, la fonction renvoie `NULL`.

**Solution :**

**Listing 3.** ***recherche***

```
struct cellule *findStr(char *str) {
    struct cellule *p = debut;
    while (p != NULL) {
        if (strcmp(str, p->chaine) == 0)
            break;
        else
            p = p->next;
    }
    return p;
}
```

3. Écrire une fonction

```
void insere(char *str);
```

qui :

- si on essaie d'insérer une chaîne existant dans la liste, incrémente simplement le compteur de la cellule correspondante *(on pourra utiliser la fonction précédente)* ;
- sinon insère une nouvelle chaîne au bon endroit dans la liste.

**Solution :**

**Listing 4.** ***insertion***

```
void insere(char *str) {
    struct cellule *p = debut;
    struct cellule *q = findStr(str);
    if (q != NULL)
        q->compteur++;
    else {
        q = malloc(sizeof(struct cellule));
        q->chaine = strdup(str);
        q->compteur = 1;
        q->pred = q->next = NULL;
        if (p == NULL)
            debut = q; /* liste vide */
        else if (strcmp(str, p->chaine) < 0) {
            p->pred = q;
            q->next = p;
            debut = q; /* ajout en tete */
        } else {
            while (p->next && strcmp(str, p->next->chaine) > 0)
                p = p->next;
            q->next = p->next;
            q->pred = p;
            p->next = q;
            if (q->next != NULL)
                q->next->pred = q; /* ajout au milieu */
        }
    }
}
```

**Solution :**