Exercice 1

Que fait ce programme ?

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

int main(int argc, char *argv[]) {
    for (int i = 0; i < argc; ++i) {
        char buffer[100];
        if (strlen(argv[i]) > 100) {
            printf("Argument trop long: ignore\n");
            continue;
        }
        strncpy(buffer, argv[i], 100);
        for (int j = 0; j < strlen(buffer); ++j)
            if (buffer[j] >= 'a' && buffer[j] <= 'z')
                buffer[j] -= 'a' - 'A';
        char *p = buffer;
        while (*p != '\0') {
            printf("%c", *p);
            p++;
        }
        puts("");
    }
}
```

Ce programme affiche les arguments du programme en majuscules. Il itère sur les arguments, copie chaque argument dans un buffer de taille 100, transforme les minuscules en majuscules, puis affiche le contenu du buffer.
Voici une version commentée du programme :

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

int main(int argc, char *argv[]) {
    for (int i = 0; i < argc; ++i) { // Itère sur les arguments
        char buffer[100]; // Crée un espace de travail modifiable
        // Si l'argument dépasse la taille du buffer on l'ignore
        if (strlen(argv[i]) > 100) {
            printf("Argument trop long: ignore\n");
            continue;
        }
        // Copie l'argument dans le buffer
        strncpy(buffer, argv[i], 100);

        // Transforme les minuscules en majuscules
        for (int j = 0; j < strlen(buffer); ++j)
            if (buffer[j] >= 'a' && buffer[j] <= 'z')
                buffer[j] -= 'a' - 'A';

        // Affiche le contenu du buffer
        char *p = buffer;
        while (*p != '\0') {
            printf("%c", *p);
            p++;
        }
        puts("");
    }
}
```

Exercice 2 : Tableaux

Considérez les déclarations suivantes :

```c
int a[10];
int b[10][10];
int c[] = {1, 2, 3, 4, 5};
char d[] = "Hello";
int *e[10];
int (*f)[10];
```

(a) Quelle est la taille en mémoire de chaque déclaration ?

```c
int a[10]; // 10 * sizeof(int) = 40
int b[10][10]; // 10 * 10 * sizeof(int) = 400
int c[] = {1, 2, 3, 4, 5}; // 5 * sizeof(int) = 20
char d[] = "Hello"; // (5 + 1) * sizeof(char) = 6

// Déclare e comme un tableau de 10 pointeurs vers int
int *e[10]; // 10 * sizeof(int *) = 80

// Déclare f comme un pointeur vers un tableau de 10 entiers
int (*f)[10]; // sizeof(int *) = 8
```

(b) Quelle est la différence entre `sizeof(a)`, `sizeof(*a)` et `sizeof(a[1])` ?

```
sizeof(a) = 40 // Taille du tableau, car sa taille est connue
sizeof(*a) = sizeof(int) = 4 // Taille du premier élément du tableau
sizeof(a[1]) = sizeof(int) = 4 // Taille du deuxième élément du tableau
```

(c) Quelle est la différence entre `sizeof(d)`, `strlen(d)` ? Et comment est-ce que `strlen` fonctionne ?

```
sizeof(d) = 6 // Taille du tableau en mémoire (5 octets plus la sentinelle)
strlen(d) = 5 // Taille de la chaîne de caractères, sans la sentinelle
```

La fonction ‘strlen‘ itère sur la chaîne de caractères jusqu'à trouver la sentinelle ‘
0‘.

(d) Pouvez-vous implémenter le comportement de `strlen` ?

```
size_t strlen(char *str) {
    size_t size = 0;
    while(str[size] != '\0') {
        size++;
    }
    return size;
}
```

(e) Comment afficher le contenu de `b` sous forme de matrice ligne colonne ?

```
for (int i = 0; i < 10; i++) {
    for (int j = 0; j < 10; j++)
        printf("%d ", b[i][j]);
    printf("\n");
}
```

Exercice 3

Pointeurs et fonctions

(a) Quelle est la différence entre `int *p` et `int* p` ?

Il n'y en a pas, les deux sont équivalents. Rappelez-vous que le C ne considère pas les espaces comme significatifs.

(b) Quelle est la différence entre `int *p` et `int p[10]` ?

`int *p` est un pointeur vers un entier, la variable prend en mémoire la taille d'une adresse soit 64 bits sur une architecture 64 bits. Elle ne définit pas un tableau mais contient uniquement une adresse.
`int p[10]` déclare un tableau non initialisé de 10 entiers. La différence est que `p` peut être modifié pour pointer vers un autre entier, alors que `p[10]` est une constante qui ne peut pas être modifiée, seul le contenu du tableau peut être modifié.

(c) Pour la déclaration `int a[10]`, si j'écrit `a[10] = 42`, est-ce que le compilateur va générer une erreur ? Et que se passe-t-il ?

Le compilateur ne génère pas d'erreur, mais le comportement est indéfini. Cela veut dire que le compilateur peut générer un code qui fonctionne, ou qui ne fonctionne pas, ou qui fonctionne parfois et pas d'autres fois. C'est une erreur de programmation car on accède à un élément qui n'existe pas.
En effet `a[10]` est un raccourci pour `*(a + 10)`, et `a + 10` est une adresse qui n'est pas dans le

> tableau car on accède au onzième élément alors que le tableau n'en contient que dix. On appelle ça du jardinage mémoire et le risque est une erreur de segmentation.

Exercice 4

Passage par adresse et par valeur

(a) Que signifie ce prototype de fonction `void f(int *p)` ?

> La fonction `f` reçoit un pointeur vers un entier en paramètre. C'est à dire qu'elle reçoit l'adresse d'une variable de type entier et peut donc modifier cette variable à la source. Si la variable p était passée directement `void f(int p)`, la fonction recevrait une copie de la variable et ne pourrait pas la modifier. Enfin, elle pourrait modifier la valeur sur le *stack* mais pas la valeur de la variable originale.

(b) Que signifie ce prototype de fonction `void f(int a[])`, comparez avec la question précédente.

> C'est la même chose. L'écriture `int a[]` est un raccourci pour `int *a`.

(c) Considérant le code suivant, est-ce que le code est fonctionnel ? Que voyez-vous à l'écran si vous l'exécutez ?

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

size_t f(int a[]) {
    return sizeof(a);
}

int main() {
    int a[10];
    printf("%ld\n", sizeof(a));
    printf("%ld\n", f(a));
}
```

> Le code compile mais n'est pas correct. Il affichera :
>
> ```
> 40
> 8
> ```
>
> Il affiche la taille en bytes du tableau. Cependant comme seul le pointeur est reçu dans la fonction 'f', la taille du tableau n'est pas connue. Le compilateur considère ainsi 'a' comme un pointeur simple et retourne la taille d'un pointeur.
> Par ailleurs une alerte de compilation apparaît :
>
> ```
> ptr.c: In function 'f':
> ptr.c:5:18: warning: 'sizeof' on array function parameter 'a' will return
> ↪  size of 'int *' [-Wsizeof-array-argument]
>    5 |     return sizeof(a);
>      |                  ^
> ```

(d) Si le code est modifié avec ce prototype de fonction `void f(int a[10])`, comparez avec la questions précédente.

> Le comportement est strictement identique car `int a[10]` est un raccourci pour `int *a`. Le passage de la taille du tableau est ignoré par le compilateur.

(e) Comment convenablement écrire la fonction `f` pour qu'elle s'adapte à n'importe quelle taille de tableau ? (indice, il faut deux paramètres)

Il faut passer le tableau ainsi que sa taille en paramètres. Par exemple :

```
size_t f(int a[], size_t size) {
    for (int i = 0; i < size; i++)
        printf("%d\n", a[i]);
    return size;
}
```

La fonction retourne la taille du tableau et affiche sur la sortie standard les éléments du tableau.

(f) Comment modifier le prototype de `f` pour que le tableau puisse être parcouru mais qu'il ne puisse pas être modifié ?

Il faut passer le tableau en lecture seule. Par exemple :

```
size_t f(const int a[], size_t size) {
    for (int i = 0; i < size; i++)
        printf("%d\n", a[i]);
    return size;
}
```