# Contrôle Exemple de C (Correction à la fin)

Ce sujet est donné à but d'entrainement, les questions et exercices du contrôle peuvent être plus nombreux.

Une correction est fournie en fin de ce document.

*Pour toute question ou remarque relative aux cours/tps/sujets : pierre.bertin-johannet@orange.fr*

## Exercice 0: QCM

Plusieurs réponses possibles, pas de points négatifs.

**Une boucle for permet :**

1. D'éxecuter du code en paralelle
2. D'éxecuter du code plusieurs fois d'affilée
3. D'éxecuter du code une infinité de fois
4. D'éxecuter du code uniquement si une condition est vraie

**Pour accéder au deuxième élément du tableau T, j'ecris :**

1. t[1]
2. t[2]
3. t[3]
4. (*t)[2]

**Les variables en C**

1. Utilisent de la mémoire réservée uniquement pour la fonction dans laquelle elles sont déclarées
2. Sont enregistrées dans le **tas**
3. Existent pour une durée définie
4. Ont une valeur par défaut si elles ne sont pas initialisées

**L'instruction if permet :**

1. D'éxecuter du code en paralelle
2. D'éxecuter du code plusieurs fois d'affilée
3. D'éxecuter du code une infinité de fois
4. D'éxecuter du code uniquement si une condition est vraie

**Les arguments d'une fonction en C:**

1. Sont copiés dans la mémoire de la fonction appelée
2. Sont des pointeurs vers les variables de la fonction appelante
3. Sont stockés dans la **pile**
4. Sont déclarés en même temps que la fonction

**Une structure en C**

1. Peut servir à contenir plusieurs variables
2. Peut changer de taille lors de l'execution du programme
3. Peut etre créé avec le mot clé `struct`

**La chaine de charactères `"Bonjour"`**

1. Occupe 7 octets en mémoire
2. Occupe 8 octets en mémoire
3. Occupe 14 octets en mémoire
4. Occupe 15 octets en mémoire

**Une socket sous linux**

1. Correspond à un port physique sur la carte réseau
2. Est identifiée par un numéro

3. Permet de communiquer directement avec le driver
4. Peut être configurée pour utiliser un protocole particulier

**Pour allouer de la mémoire sur le tas je peux utiliser:**

1. malloc
2. dealloc
3. alloca
4. memalloc

**La taille mémoire d'un int est**

1. 4 octets
2. 2 octets
3. n'est pas fixe
4. 8 octets

**La taille mémoire d'un char est**

1. 1 octets
2. 2 octets
3. n'est pas fixe
4. 4 octets

**La taille d'une union**

1. Dépend de l'attribut le plus petit
2. Dépend de la somme de la taille des attributs
3. Dépend de l'attribut le plus grand
4. Dépend du nom de l'union

**Que dire d'un programme qui essaye d'accéder à l'élément 5 dans un tableau de taille 2:**

1. Le programme crash immédiatement
2. Le programme accède toujours à la mémoire, même si elle n'est pas réservée
3. Le programme contient une faille de type buffer-overflow

## Exercice 1

1. Qu'affichera le code suivant dans la console :

```c
int main(){
  int toFind = 12;
  int a = 0;
  int b = 0;
  int found = 0;
  while (!found && a < toFind/2){
      a = a+1;
      b = 0;
      while (!found && b < toFind/2){
          b = b + 1;
          if (a * b == toFind){
              found = 1;
          }
      }
  }
  printf("%d, %d\n", a, b);
}
```

2. Completez le code suivant afin que le programme affiche le nombre d'entiers pairs dans le tableau de taille n.

```c
int main(){
    int* tableau = /* initialisation du tableau */
    int n;
    /// completer ici
}
```

## Exercice 2

Complétez les types manquants dans ces fonctions.

```c
..... join(..... str1, ..... delim, ..... str2){
    ..... len1 = strlen(str1);
    ..... len2 = strlen(str2);
    ..... ret = malloc(sizeof(.....) * (len1 + len2 + 2))
    ..... count = 0;
    for (count = 0; count < len1; count++){
      ret[count] = str2[count];
    }
    ret[count] = delim;
    count++;
    for (..... i = 0; i < len2; i++){
      ret[count + i] = str2[i];
    }
    ret[count + len2] = 0;
    return ret;
}

..... raccourcir(..... tableau, ..... nouvelle_taille){
  ..... nouveau_tab = malloc(sizeof(float) * nouvelle_taille);
  for (..... i = 0; i < nouvelle_taille; i++){
    nouveau_tab[i] = (*tableau)[i];
  }
  free(*tableau);
  *tableau = nouveau_tab;
}
```

## Exercice 3

Que va afficher le code suivant dans la console ?

```c
void a(int n, int *b){
    n = *b;
}
void b(int* n){
    *n = 2;
}
void c(int n, int k){
    n = n + k;
}

int main(){
    int i = 1;
    int j = 2;
    int k = 3;
    a(k, &i);
    b(&i);
    c(j, k);
    printf("%d, %d, %d\n", i, j, k);
}
```

## Exerice 4.

On utilisera la structure suivante dans l'exercice :

```
struct User{
  int age;
  char* name;
}
```

1. Ecrivez le code d'une fonction qui demande dans la console les informations d'un utilisateur, les enregistre dans une struct User et les renvoie (On considère que l'utilisateur ne saisira jamais un nom de plus de 10 charactères).
2. Ecrivez le code d'une fonction qui accepte en argument un entier n, demande les informations de n utilisateurs et les enregistre dans un tableau avant de le renvoyer.
3. Ecrivez une fonction qui accepte un entier n ainsi qu'un tableau d'utilisateurs de taille n en argument et affiche les noms de tous les utilisateurs.
4. Créez une fonction main qui utilise les fonctions précédentes pour d'abord demander une liste d'utilisateurs et ensuite les afficher dans la console. Toute la mémoire allouée devra être libérée avant la fin de l'execution du programme.

## Exercice 5

1. Quels sont les deux problèmes de gestion mémoire dans la fonction ci-dessous ?

```
int somme_tab(int n, int* values){
  int somme = 0;
  for (int j = 0; j < n; j++){
    somme +=values[j];
  }
  return somme;
}

int calcul_somme(){
    for (int i = 0; i < 5; i++){
        printf("entrez un nombre");
        int n;
        scanf("%d\n", &n);
        int* values = malloc(n);
        for (int j = 0; j < n; j++){
          scanf("%d", values + j);
        }
        printf("somme %d \n", somme_tab(n, values));
    }
}
```

# Correction

## Exercice 0: QCM

**Une boucle for permet :**

1. D'éxecuter du code en paralelle
2. **D'éxecuter du code plusieurs fois d'affilée**
3. **D'éxecuter du code une infinité de fois**
4. **D'éxecuter du code uniquement si une condition est vraie**

**Pour accéder au deuxième élément du tableau T, j'ecris :**

1. **t[1]**
2. t[2]
3. t[3]
4. (*t)[2]

**Les variables en C**

1. **Utilisent de la mémoire réservée uniquement pour la fonction dans laquelle elles sont déclarées**
2. Sont enregistrées dans le **tas**
3. **Existent pour une durée définie**
4. Ont une valeur par défaut si elles ne sont pas initialisées

**L'instruction if permet :**

1. D'éxecuter du code en paralelle
2. D'éxecuter du code plusieurs fois d'affilée
3. D'éxecuter du code une infinité de fois
4. **D'éxecuter du code uniquement si une condition est vraie**

**Les arguments d'une fonction en C:**

1. **Sont copiés dans la mémoire de la fonction appelée**
2. Sont des pointeurs vers les variables de la fonction appelante
3. **Sont stockés dans la pile**
4. **Sont déclarés en même temps que la fonction**

**Une structure en C**

1. **Peut servir à contenir plusieurs variables**
2. Peut changer de taille lors de l'execution du programme
3. **Peut etre créé avec le mot clé `struct`**

**La chaine de charactères `"Bonjour"`**

1. Occupe 7 octets en mémoire
2. **Occupe 8 octets en mémoire**
3. Occupe 14 octets en mémoire
4. Occupe 15 octets en mémoire

**Une socket sous linux**

1. Correspond à un port physique sur la carte réseau
2. **Est identifiée par un numéro**
3. Permet de communiquer directement avec le driver
4. **Peut être configurée pour utiliser un protocole particulier**

**Pour allouer de la mémoire sur le tas je peux utiliser:**

1. **malloc**
2. dealloc
3. alloca
4. memalloc

**La taille mémoire d'un int est**

1. 4 octets
2. 2 octets
3. **n'est pas fixe**
4. 8 octets

**La taille mémoire d'un char est**

1. **1 octet**
2. 2 octets
3. n'est pas fixe
4. 4 octets

**La taille d'une union**

1. Dépend de l'attribut le plus petit
2. Dépend de la somme de la taille des attributs
3. **Dépend de l'attribut le plus grand**
4. Dépend du nom de l'union

**Que dire d'un programme qui essaye d'accéder à l'élément 5 dans un tableau de taille 2:**

1. Le programme crash immédiatement
2. Le programme accède toujours à la mémoire, même si elle n'est pas réservée
3. **Le programme contient une faille de type buffer-overflow**

## Exercice 1

1. Le code parcours toutes les combinaisons possibles de a et b et trouve la première telle que le produit des deux est 18. Il affiche 2, 6.
2. 
```c
int main(){
    int tableau[6] = {1, 2, 5, 3, 4, 9};
    int n = 6;
    int nb_pairs = 0;
    for (int i = 0; i < n; i++){
      nb_pairs += !(tableau[i] & 1);
    }
    printf("%d\n", nb_pairs);
}
```

## Exercice 2

```c
char* join2 (char* str1, char delim, char* str2){
    int len1 = strlen(str1);
    int len2 = strlen(str2);
    char* ret = malloc(sizeof(char) * (len1 + len2 + 2));
    int count = 0;
    for (count = 0; count < len1; count++){
      ret[count] = str2[count];
    }
    ret[count] = delim;
    count++;
    for (int i = 0; i < len2; i++){
      ret[count + i] = str2[i];
    }
    ret[count + len2] = 0;
    return ret;
}

void raccourcir(float** tableau, int nouvelle_taille){
```

```
  int nouveau_tab = malloc(sizeof(float) * nouvelle_taille);
  for (int i = 0; i < nouvelle_taille; i++){
    nouveau_tab[i] = (*tableau)[i];
  }
  free(*tableau);
  *tableau = nouveau_tab;
}
```

## Exercice 3

Les fonctions c et a ne modifient rien car les arguments sont copiés, on affiche donc 2, 2, 3

## Exerice 4.

```
struct User ask_user(){
  struct User user;
  user.name = malloc(11);
  scanf("%d", &(user.age));
  scanf("%s", user.name);
  scanf("%f", &(user.height));
  return user;
}

struct User* fill_users(int* n){
  printf("how many \n");
  scanf("%d", n);
  struct User* users = malloc(sizeof(struct User) * *n);
  for (int i = 0; i < *n; i++){
      users[i] = ask_user();
  }
  return users;
}

void print_users(struct User* users, int nb_users){
  for (int i = 0; i < nb_users; i++){
      printf("name : %s, height : %f, age: %d\n", users[i].name, users[i].height,
users[i].age);
  }
}

int main(){
  int n;
  struct User* users = fill_users(&n);
  print_users(users, n);
  free(users);
}
```

## Exercice 5

1. La fonction calcul_somme ne libère pas la mémoire après l'avoir alloué, cela cause une fuite mémoire.
2. L'appel à la fonction malloc alloue seulement n octets pour n `int`, la taille d'un int peut être superieure à un octet, il faudrait prendre en compte la taille d'un int ainsi: `malloc(n*sizeof(int))`.