Année 2025-2026 | 1ère session – 19 Décembre 2025

TE5-PROG1

Programmation Impérative

1ère année Télécommunications

Responsable
Philippe Swartvagher

Intervenants
Fadwa Abakarim
Joachim Bruneau-Queyreix
Guillaume Mercier

Durée de l'examen : 1 h

Documents autorisés	☐	sans document	☒
Calculatrice autorisée	☐	non autorisée	☒

Nom et Prénom: _____

- **N'oubliez pas d'écrire votre nom ci-dessus !**
- **Il est impératif de répondre dans les espaces prévus à cet effet : tout ce qui dépasse ne sera pas pris en compte lors de la correction.**
- **Dans les cadres où du code est attendu, il est impératif d'écrire en respectant les numéros de ligne.**
- **L'orthographe, la grammaire, la conjugaison et la syntaxe seront pris en compte lors de la correction.**
- **Écrivez le code comme si vous écriviez dans un fichier source : la syntaxe, le style (indentation, noms de variables cohérents, ...) seront pris en compte lors de la correction. Il va sans dire que le code que vous écrivez doit compiler.**
- **N'oubliez pas d'écrire votre nom ci-dessus !**

Exercice 1 – QCM

Chaque question possède une et une seule bonne réponse.

Barème :

- 1 point par bonne réponse
- **-0,5 point par mauvaise réponse**
- 0 point si absence de réponse

On considérera que chaque extrait de code présenté est correctement intégré dans une fonction `main()` et que tous les `#include` nécessaires sont présents.

1. Que va afficher le code suivant, compilé sur et pour un processeur `x86_64` ?

```
if (sizeof(char*) == sizeof(long*))
{
  printf("Égal\n");
}
else
{
  printf("Pas égal\n");
}
```

   - ☐ `Égal`
   - ☐ `Pas égal`
   - ☐ Erreur à la compilation
   - ☐ Erreur à l'exécution

2. Comment peut-on connaître le nombre d'éléments dans n'importe quel tableau ?
   - ☐ On parcourt le tableau jusqu'à atteindre une valeur particulière
   - ☐ Avec l'opérateur `sizeof`
   - ☐ On ne peut pas
   - ☐ Avec la fonction `len()`
   - ☐ Avec la fonction `strlen()` qui indique la fin du tableau

3. Que va afficher le code suivant ?

```
char* s = "Vive le c";
s[8] = 'C';
printf("%s\n", s);
```

   - ☐ `Vive le c`
   - ☐ Erreur à l'exécution
   - ☐ `Vive le C`
   - ☐ Erreur à la compilation

4. Que va afficher le code suivant ?

```
float a = 0.3;
float b = 4.8;
if (a+b == 5.1)
{
  printf("%f\n", 3.9);
}
else
{
  printf("Pas égal\n");
}
```

   - ☐ `Pas égal`
   - ☐ `3.9`
   - ☐ `4`
   - ☐ On ne peut pas prédire ce qui sera affiché.

Exercice 2 – Questions de cours

1. Quel est le problème avec le code suivant ?

```
int tab[10];
for (int i = 0; i <= 10; i++)
{
  tab[i] = i;
}
```

.....

.....

.....

2. Soit le code suivant :

```
enum jour_e { LUNDI, MARDI, MERCREDI, JEUDI, VENDREDI, SAMEDI, DIMANCHE };
char* jours[] = {"Lun", "Mar", "Mer", "Jeu", "Ven", "Sam", "Dim"};
printf("%s\n", jours[MARDI]);
```

   (a) Que va afficher ce code ?

   .....

   (b) Pourquoi est-il possible d'écrire `jours[MARDI]` dans le code ?

   .....

   .....

   .....

3. Transformez le code suivant en une condition ternaire :

```
if (a == b)
  return a-1;
else
  return b+1;
```

.....

4. Soit le code suivant :

```
int* premier_pair(int* tab, int n)
{
  for (int i = 0; i < n; i++)
  {
    if (tab[i] % 2 == 0)
    {
      return tab[i];
    }
  }
  return NULL;
}
```

(a) Quel est le problème avec ce code ?

.....

.....

.....

.....

(b) Quelle correction proposez-vous ?

.....

.....

.....

.....

Exercice 3 – Interprétation d'une chaîne de caractères

Inspiré de `https://adventofcode.com/2025/day/1`

On souhaite écrire une fonction qui interprète les chaînes de caractères formatées d'une certaine façon. Ces chaînes de caractères commencent par la lettre `L` ou `R`, lettre immédiatement suivi d'un nombre entier positif. Par exemple : `"L32"` ou `"R4"`.
On souhaite récupérer le nombre contenu dans cette chaîne de caractères. Si la première lettre est `L`, alors on prend l'opposé de ce nombre, mais si c'est `R`, alors on prend directement le nombre. Exemple :

```
printf("%d\n", extract("L32"));
printf("%d\n", extract("R4"));
```

affichera :

```
-32
4
```

Écrivez le code de la fonction `extract()`. On fera les hypothèses suivantes :

- la chaîne de caractères passée en paramètre est toujours formatée comme attendue (la lettre `L` ou `R` suivie d'un nombre) ;
- le nombre à récupérer peut avoir n'importe quel nombre de chiffres.

```
int extract(char* s)
{
}
```

Exercice 5 – Implémentation d'une pile

Dans cet exercice, il est question d'implémenter une pile (*stack* en anglais), structure de données où il est possible d'ajouter autant d'éléments qu'on le souhaite, mais il n'est possible de retirer que l'élément en haut de la pile (*i.e.*, le dernier élément ajouté).
La pile sera représentée par la structure suivante :

```
struct stack_s {
    int capacity; // combien d'éléments peuvent être stockés
    int occupancy; // combien d'éléments sont déjà stockés
    int* content; // tableau contenant les éléments
};
```

Notre implémentation stockera des éléments de type `int` dans un tableau. L'utilisation des fonctions de notre interface permettra de cacher qu'un tableau est utilisé. La fin de cet exercice contient un exemple d'utilisation.

1. Compléter la fonction suivante qui crée une pile vide de capacité passée en paramètre. Cette fonction alloue notamment le tableau qui va servir à stocker les éléments.

```
1  struct stack_s* stack_create(int capacity)
2  {
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22 }
```

2. Compléter la fonction suivante qui permet de libérer les ressources allouées par la fonction `stack_create()`.

```
1 void stack_free(struct stack_s* stack)
2 {
3
4
5
6 }
```

3. Compléter la fonction suivante qui permet d'ajouter un élément à la pile en le plaçant dans la première place disponible dans le tableau. Cette fonction renvoie 1 si l'ajout a réussi, 0 sinon.

```
int stack_push(struct stack_s* stack, int v)
{
}
```

4. Compléter la fonction suivante qui permet de *consulter* l'élément au sommet de la pile : l'appelant de la fonction récupère l'élément au sommet de la pile, *mais l'élément reste dans la pile*. L'élément à récupérer est placé à l'adresse contenue dans le paramètre `v`. Cette fonction renvoie 1 en cas de succès, 0 en cas d'échec.

```
int stack_peek(struct stack_s* stack, int* v)
{
}
```

5. Compléter la fonction suivante qui permet de *d'obtenir et enlever* l'élément au sommet de la pile : l'appelant de la fonction récupère l'élément au sommet de la pile, *et l'élément est retiré de la pile.* Cette fonction renvoie 1 en cas de succès, 0 en cas d'échec.
   Indice : il est possible d'utiliser les autres fonctions de l'interface.

```
int stack_pop(struct stack_s* stack, int* v)
{
}
```

6. Compléter la fonction suivante qui permet d'obtenir le nombre d'éléments dans la pile. Cette fonction renvoie -1 en cas d'échec.

```
int stack_occupancy(struct stack_s* stack)
{
}
```

7. Compléter la fonction suivante qui permet d'ajouter un élément à la pile *et augmente la capacité de la pile si c'est nécessaire.* S'il est nécessaire d'augmenter la capacité de la pile, la capacité de la pile sera doublée. Cette fonction renvoie 1 si l'ajout a réussi, 0 sinon.

   La fonction `realloc()` pourra vous être utile :

```
void* realloc(void* ptr, size_t size);
```

   Elle redimensionne à `size` octets la zone mémoire pointée par `ptr` allouée dynamiquement auparavant et renvoie un pointeur vers la zone mémoire redimensionnée, qui peut être différent de la zone initiale.

```c
int stack_push_expand(struct stack_s* stack, int v)
{
}
```

8. *Bonus.* Telle que définie, notre pile ne peut stocker que des éléments de type `int`. Quel(s) mécanisme(s) du langage C est-il possible d'utiliser pour définir des piles qui stockent des éléments de types différents (tous les éléments au sein d'une même pile sont de même type, mais deux piles peuvent stocker des éléments de types différentes) sans avoir à dupliquer le code de l'implémentation pour chaque type ciblé ?

.....

.....

.....

.....

.....

.....

.....

Voici un exemple illustrant l'utilisation de la pile :

```
int main()
{
    // TODO: vérifier les codes de retour de toutes les fonctions

    struct stack_s* s = stack_create(5);
    // s = { cap = 5, occ = 0, cont = 0x1876 } 0x1876 = { , , , , }

    stack_push(s, 3);
    // s = { cap = 5, occ = 1, cont = 0x1876 } 0x1876 = { 3, , , , }

    stack_push(s, 7);
    // s = { cap = 5, occ = 2, cont = 0x1876 } 0x1876 = { 3, 7, , , }

    stack_push(s, 1);
    // s = { cap = 5, occ = 3, cont = 0x1876 } 0x1876 = { 3, 7, 1, , }

    int x;
    stack_peek(s, &x);
    // s = { cap = 5, occ = 3, cont = 0x1876 } 0x1876 = { 3, 7, 1, , } x = 1

    int y;
    stack_pop(s, &y);
    // s = { cap = 5, occ = 2, cont = 0x1876 } 0x1876 = { 3, 7,  , , } y = 1

    stack_push(s, 4);
    stack_push(s, 5);
    stack_push(s, 2);
    // s = { cap = 5, occ = 5, cont = 0x1876 } 0x1876 = { 3, 7, 4, 5, 2}
    stack_push(s, 12); // échec : renvoie 0

    stack_push_expand(s, 12); // ok
    // s = { cap = 10, occ = 6, cont = 0x3014 } 0x3014 = { 3, 7, 4, 5, 2, 12, , , , }

    stack_free(s);

    return 0;
}
```