Perfectionnement à la programmation en `C`

Licence 2 Informatique – Examen 2025–2026

Mardi 19 Mai 2026

Deux feuilles A4 de notes manuscrites personnelles autorisées, tout autre document interdit.

Voici quelques conseils :

- lire l'intégralité du sujet avant de commencer
- respecter les **conventions** étudiées (pré-assertions, gestion d'erreurs lorsque c'est pertinent). Une fonction qui « marche » peut être incorrecte
- lorsqu'il est demandé de définir une fonction, seuls les paramètres principaux sont précisés. Il est ainsi possible d'**ajouter d'autres paramètres**
- il est autorisé de déclarer des **types** et de définir des **fonctions intermédiaires** si besoin est
- les seules fonctions externes, types et macro-définitions qu'il est autorisé d'utiliser sont ceux apportés par les **en-têtes** `stdio.h`, `stdlib.h` et `assert.h` (qu'on supposera inclus) ainsi que la fonction `char *strdup(char *s)` qui duplique une chaine de caractères dans le tas et renvoie l'adresse de la nouvelle (ou `NULL` en cas d'échec).

Ce devoir est constitué de 18 exercices. Le barème est donné à titre indicatif par partie. Les questions des parties **Bases** et **Techniques avancées** sont indépendantes et peuvent être traitées dans n'importe quel ordre.

Analyse de code *(9 points)*

Voici le début d'un code permettant de manipuler un tableau de chaines de caractères.

```c
#include <stdlib.h>

typedef struct {
    char **chaines; // Tableau de chaines
    int taille;     // Nombre d'éléments dans le tableau
    int capacite;   // Capacité du tableau
} Vec;

// Allocation d'un nouveau tableau
Vec vec_nouveau(int n) {
    return (Vec) {
        .chaines = calloc(n, sizeof (char*)),
        .taille = 0,
        .capacite = n,
    };
}

// Ajoute une chaine au tableau, renvoie 1 en cas de succès et 0 sinon
int vec_ajouter(Vec *tab, char *mot) { /* TODO */ }

// Libère la mémoire associée au tableau et le met à zéro
void vec_detruire(Vec *tab) { /* TODO */ }
```

Exercice 1. **Écrire** les fonctions `vec_ajouter` et `vec_detruire`. On ne cherchera pas à augmenter la capacité du tableau.

Exercice 2. On dispose du `main` suivant utilisant le module de tableau de chaines de caractères.

```c
int main() {
  Vec tab = vec_nouveau(10);
  vec_ajouter(&tab, "Hello");
  vec_ajouter(&tab, "World");
  vec_detruire(&tab);
  return 0;
}
```

Suite aux deux ajouts, la variable `tab` représente le tableau `["Hello", "World"]`. **Indiquer** dans quelle(s) partie(s) de la mémoire se trouve `tab`, `tab.chaines`, `tab.chaines[0]` et les chaines « Hello » et « World » du tableau.

Exercice 3. On ajoute la ligne `vec_ajouter(&tab, strdup("Beautiful"));` entre les deux appels à la fonction `vec_ajouter`. **Expliquer** pourquoi le code précédent provoque une fuite mémoire et **indiquer** en justifiant à quel moment celle-ci devient irrécupérable (ex: en appelant `vec_detruire`, à la fin de l'appel à `vec_ajouter`, à la fin du `main`…).

Bases *(14 points)*

Exercice 4. **Écrire** une fonction `longueur_moyenne` paramétrée par une chaine de caractères et qui renvoie la longueur moyenne des mots de la chaine. On supposera que la chaine ne comporte que des lettres minuscules et des espaces.

Exercice 5. **Donner** le **prototype** d'une fonction `dict_definir` paramétrée par un dictionnaire, un mot et sa définition et qui ajoute une définition à un mot dans le dictionnaire et **écrire** sa documentation. Un dictionnaire est un objet de type `Dict`, non détaillé, dont le contenu est alloué dans le tas. Un mot et sa définition sont des chaines de caractères. La fonction devra gérer les éventuelles erreurs.

Donner un exemple d'utilisation de cette fonction.

Exercice 6. **Écrire** une fonction `fichier_lire_ligne` à gestion d'erreurs paramétrée par un nom de fichier, un numéro de ligne et un tableau d'octets dans lequel sera copié la ligne demandée. La fonction renverra le nombre d'octets recopiés en cas de succès. La fonction doit permettre à la fonction appelante de distinguer entre les erreurs rencontrées. *Note: On pourra lire le fichier plusieurs fois si nécessaire.*

Modularisation *(17 points)*

Pour réaliser le projet du jeu du serpent en `ncurses` vu en TP, on a réalisé les modules suivant : Main, Monde, Pomme, Serpent, Case, IGraphique.

Voici les types principaux :

```c
typedef struct {
  int ligne, colonne;
} Case;
typedef struct {
  Case c;
  CellCase *suivant;
} CellCase, *ListeCase;
typedef struct {
  ListeCase corps;
  Direction dir;
} Serpent;
```

```c
typedef enum { HAUT, BAS, GAUCHE,
 DROITE } Direction;
typedef struct {
  Case *tab;
  int taille;
  int capacite;
} Pommes;
typedef struct {
  int lignes, colonnes;
  Serpent serpent;
  Pommes pommes;
} Monde;
```

Exercice 7. **Placer** chaque type dans un des 6 modules du projet.

Exercice 8. Une des structures présente un problème de définition, **laquelle** ? (justifier) **Proposer** une correction.

Exercice 9. À quoi servent les 3 membres de la structure `Pommes` ?

Exercice 10. Pour chaque module, **proposer** le prototype d'une fonction qu'il pourrait contenir.

Exercice 11. On souhaite ajouter des pommes aux effets variés (score négatif, modification de la vitesse du serpent…). **Donner** les modifications de type nécessaires pour pouvoir implémenter cet ajout.

Exercice 12. **Proposer** un graphe de dépendances (étendues) du projet.

Exercice 13. À partir du graphe de dépendances, **écrire** un `Makefile` simple compilant votre projet avec `ncurses`. On utilisera une unique règle `%.o: %.c` avec les variables `$@` et `$<` pour compiler les fichiers objets et éviter la redondance.

Exercice 14. Après une compilation (qu'on supposera sans erreur), on modifie le fichier `Monde.h`. Sachant qu'on a conservé les fichiers objets, quels sont les fichiers à produire à nouveau à l'aide de `gcc/clang` ? **Justifier**.

Techniques avancées *(10 points)*

Exercice 15. On souhaite dans un programme représenter une salle de cours de l'université. Chaque salle a un identifiant unique d'au plus 12 caractères (comme `Ader 104` ou `Rabelais A1`). On distingue les salles de TD, les salles de TP et les amphis. Chaque salle a une capacité (en nombre de places assises). En plus de ces informations, on connait pour les salles de TP le nombre de machines disponibles, et pour les amphis, le nombre de chaises cassées et la présence ou non d'un microphone.

En utilisant une `union`, **définir** un type permettant de représenter les données d'une salle.

Exercice 16. En utilisant des opérateurs bits à bits, **écrire** une fonction `quotient_reste` qui prend en argument deux entiers positifs ou nuls n et k où k est un entier plus petit que 31, et renvoie le quotient et reste de la division euclidienne de n par 2^k.

Exercice 17. **Écrire** une fonction `melange` qui renvoie le mélange les bits de deux mots de 64 bits `a` et `b` selon les bits d'un 3e mot `m`. Ce mot doit avoir les bits de `a` aux positions où les bits de `m` sont à `0` et les bits de `b` aux positions où les bits de `m` sont à 1.

Par exemple sur **8 bits**, le mélange des mots `a=`**`01110010`** et `b=``01001110` suivant le mot `m=01100100` est **`0`**`1`**`01`**`0`**`11`**`0`.

Exercice 18. **Écrire** une fonction `grouper` paramétrée par un tableau d'entiers de type `int` et un pointeur sur une fonction `test` ayant un paramètre de type `int` et comme type de retour. La fonction modifie le tableau afin que les premiers éléments sont les éléments pour lesquels `test` renvoie `0`, et les derniers sont les autres éléments. La fonction renvoie le nombre d'éléments de valeur de test `0`.

Par exemple, si la fonction `test` renvoie `1` si son paramètre est supérieur à 5 et `0` sinon, la fonction `grouper` peut transformer le tableau `{4,9,8,7,2,1}` en `{4,1,2,8,7,9}` et renverra la valeur 3.

Écrire un court programme utilisant la fonction `grouper` précédente pour déplacer les éléments pairs au début d'un tableau.