■ Correction — Devoir Surveillé : Programmation en C

Durée : 2 heures — Correction complète et commentée

Partie 1 — Questions de cours (40 minutes)

1. Variables locales / globales / statiques
• **Locale** : définie dans une fonction, visible uniquement dans celle-ci.
• **Globale** : définie hors des fonctions, accessible dans tout le programme.
• **Statique** : conserve sa valeur entre appels (si dans une fonction) ou limite la visibilité au fichier (si globale).

2. Pointeur vs Référence
Le C n'a pas de vraies références comme le C++. Un **pointeur** contient une adresse mémoire et peut être réassigné. Une **référence** (en C++) est un alias non réaffectable d'une variable.

3. Oublier le caractère nul (`\0`)
Sans `\0`, une chaîne n'a pas de fin détectable. Le programme lit au-delà de la mémoire prévue → **comportement indéfini**.

4. Avantages et risques des pointeurs
■ Accès direct à la mémoire, passage par adresse, allocations dynamiques.
■ Erreurs possibles : pointeur non initialisé, fuite mémoire, double libération.

5. malloc / calloc / realloc
• `malloc(n)` : alloue n octets non initialisés.
• `calloc(n, size)` : alloue n×size octets initialisés à 0.
• `realloc(ptr, n)` : redimensionne la zone pointée par ptr.

6. Erreurs mémoire fréquentes
• Oublier `free()` après une allocation.
• Libérer deux fois la même zone (`double free`).
• Utiliser un pointeur non initialisé.

7. Fichier texte vs binaire
• **Texte** : lisible par l'humain, chaque caractère encodé.
• **Binaire** : données brutes, plus compact, illisible sans programme.

8. Vérification de `fopen()`
`fopen` renvoie `NULL` si l'ouverture échoue. Vérifier toujours avant lecture/écriture.

9. Rôle du Makefile
Automatise la compilation : détecte les fichiers modifiés, évite recompilation complète, simplifie les commandes.

10. Différence entre `=` et `==`
`=` : affectation
`==` : comparaison de valeur.

Partie 2 — Exercices d'application (1 heure)

Exercice 1 — Inversion de tableau

```c void inverser(int tab[], int n) { for (int i = 0; i < n / 2; i++) { int tmp = tab[i]; tab[i] = tab[n - 1 - i]; tab[n - 1 - i] = tmp; } } ```

Exercice 2 — Allocation dynamique

```c int* allouer_et_initialiser(int n) { int *t = malloc(n * sizeof(int)); if (t == NULL) return NULL; for (int i = 0; i < n; i++) t[i] = i; return t; } ```
Pensez à libérer la mémoire : `free(tab);`

Exercice 3 — Structure et passage par pointeur

```c typedef struct { char nom[20]; int note; int id; } Etudiant; void augmenter_note(Etudiant *e) { if (e->note < 20) e->note += 1; } ```

Exercice 4 — Fichiers texte

```c typedef struct { char nom[20]; int note; } Etudiant; int main() { FILE *f = fopen("notes.txt", "w"); Etudiant etu[5] = {{"Jean", 15}, {"Luc", 9}, {"Marie", 18}, {"Nina", 11}, {"Léo", 7}}; for (int i = 0; i < 5; i++) fprintf(f, "%s %d\n", etu[i].nom, etu[i].note); fclose(f); f = fopen("notes.txt", "r"); Etudiant e; while (fscanf(f, "%s %d", e.nom, &e.note;) == 2) if (e.note >= 10) printf("%s a %d\n", e.nom, e.note); fclose(f); } ```

Partie 3 — Mini-projet : Gestion de stock

```c typedef struct { char nom[20]; float prix; int quantite; } Article; float valeurTotale(Article *t, int n) { float total = 0; for (int i = 0; i < n; i++) total += t[i].prix * t[i].quantite; return total; } int main() { int n; printf("Combien d'articles ? "); scanf("%d", &n;); Article *tab = malloc(n * sizeof(Article)); for (int i = 0; i < n; i++) { printf("Nom, prix, quantite : "); scanf("%s %f %d", tab[i].nom, &tab;[i].prix, &tab;[i].quantite); } printf("Valeur totale du stock : %.2f\n", valeurTotale(tab, n)); FILE *f = fopen("stock.txt", "w"); for (int i = 0; i < n; i++) fprintf(f, "%s %.2f %d\n", tab[i].nom, tab[i].prix, tab[i].quantite); fclose(f); free(tab); } ```
■ Bonus : afficher l’article au **prix le plus élevé** ou à la **quantité maximale**.
```