Exercices Langage C

🧩 **BLOC 1 – Analyse, conception et implémentation**

Objectif du bloc :
Apprendre à poser un problème de manière structurée, à identifier les entrées / sorties / traitements, et à écrire l'implémentation en C.

Exercice 1 – Analyse d'un problème simple (signe du produit)

On souhaite écrire un programme qui demande **deux nombres réels** à l'utilisateur et indique ensuite si **leur produit est positif, négatif ou nul**.

1. Faire l'analyse du problème :
    - Quelles sont les **entrées** ?
    - Quelles sont les **sorties** ?
    - Quel est le **traitement** ?
2. Proposer une **liste d'objets principaux** (variables nécessaires).
3. Implémenter la solution en **langage C** avec des instructions conditionnelles (if / else).

Exercice 2 – Recherche du maximum dans un tableau

On dispose d'un **tableau d'entiers** de taille connue N.

1. Faire l'analyse du problème :
    - Entrée(s) : tableau, taille.
    - Sortie : plus grande valeur du tableau.
2. Lister les **objets principaux** (variables nécessaires : index, max, etc.).
3. Implémenter une **fonction en C** :

```
int max_tableau(const int t[], int n);
```

4. Implémenter la main permettant de tester la fonction max_tableau

---

Exercice 3 – Appartenance d'un mot dans un dictionnaire

On représente un **dictionnaire** sous la forme d'un **tableau de chaînes de caractères trié par ordre alphabétique**.

1. Faire l'analyse du problème :
    - Entrée : mot recherché, dictionnaire (tableau de chaînes), taille.
    - Sortie : booléen ou message "trouvé / non trouvé".
2. Lister les **objets principaux** : index, tableau, mot, booléen de résultat…
3. Implémenter la fonction de recherche en C.

🧩 **BLOC 2 – Sous-programmes et passage de paramètres**

Objectif du bloc :
Comprendre l'intérêt des sous-programmes, la différence **fonction / procédure**, et la notion de **passage par valeur / par adresse**.

Exercice 1 – Fonction et procédure pour f(x)

On considère la fonction :

$$f(x) = x^2 + x - 1$$

1. Écrire une **fonction** qui calcule f(x) et retourne un double :

   ```
   double f(double x);
   ```

2. Écrire une **procédure** (fonction void) qui calcule f(x) et l'écrit dans une variable passée **par adresse** :

   ```
   void f_procedure(double x, double *res);
   ```

3. Écrire un programme principal qui :

   1. demande une valeur de x à l'utilisateur,
   2. appelle les deux sous-programmes,
   3. affiche les résultats obtenus.

Exercice 2 – Passage par valeur / par adresse

On considère le code suivant :

```
void echange (int* x, int* y) {
    int temp;
    temp = *x;
    *x = *y;
    *y = temp;
```

```
}
void ajouteUn (int x) {
    x = x + 1;
}
void ajouteDeux (int* x) {
    *x = *x + 2;
}
void ajouteTrois (int x, int* y) {
    x = x + 3;
    *y = *y + 3;
}
```

Et le programme principal :

```
int main() {
    int a = 1, b = 2, c = 3, d = 4;
    echange(&a, &b); echange(&c, &d); /* 1 */
    ajouteUn(a); ajouteDeux(&b);      /* 2 */
    ajouteTrois(c, &d);               /* 3 */
    echange(&a, &d);                  /* 4 */
}
```

1. Compléter un tableau donnant les valeurs de a, b, c, d :
    - avant (1), après (1)
    - après (2)
    - après (3)
    - après (4)
2. Expliquer, pour chaque fonction, si les paramètres sont passés **par valeur** ou **par adresse**.
3. Commenter la différence de comportement entre ajouteUn et ajouteDeux.

Exercice 3 – Permutation de deux variables

1. Écrire un **sous-programme** qui permute le contenu de deux variables entières, avec un prototype initial de type :

   ```
   void permute(int a, int b);
   ```

puis écrire le programme principal qui appelle ce sous-programme et affiche les valeurs avant / après.

2. Tester : les valeurs affichées dans le programme principal sont-elles modifiées ? Pourquoi ?
3. Modifier le sous-programme pour que la permutation soit effective dans le programme principal en utilisant le passage **par adresse** :

   ```
   void permute(int *a, int *b);
   ```

🧩 **BLOC 3 – Instructions de contrôle**

Objectif du bloc :
Savoir choisir et utiliser les structures de contrôle (if, switch, for, while, do…while) pour résoudre un problème.

Exercice 1 – Table de multiplication

1. Écrire un programme qui :
    - demande un **nombre entier** à l'utilisateur,
    - affiche la **table de multiplication de ce nombre jusqu'à 10**.
2. Transformer cette partie en un **sous-programme** :

    ```
    void table(int n);
    ```

    puis écrire un main qui appelle ce sous-programme.

Exercice 2 – Moyenne de notes (saisie jusqu'à -1)

On souhaite calculer la **moyenne de notes** saisies par l'utilisateur.
L'utilisateur entre des notes (réelles) une par une, et termine la saisie en tapant -1.

1. Écrire un programme qui :
    - lit une suite de notes,
    - s'arrête quand la valeur -1 est saisie,
    - calcule et affiche la moyenne.
2. Écrire un **sous-programme** :

    ```
    double moyenne_notes(void);
    ```

    qui effectue ce travail et renvoie la moyenne, puis l'appeler dans main.

Exercice 3 – Machine à voter simple

On souhaite écrire un programme de **vote** entre deux candidats A et B.

1. L'utilisateur peut saisir des votes pour A ou pour B, tant qu'il le souhaite.
2. Après chaque vote, le programme demande s'il doit continuer.

3. À la fin, le programme affiche :
    - le **nombre de votes** pour chaque candidat,
    - le **pourcentage** de voix pour chacun,
    - le **vainqueur** (ou égalité).

On exige que le programme soit **robuste** : il ne doit pas accepter de saisie invalide (autre chose que A/B, oui/non…).

🧩 **BLOC 4 – Tableaux et chaînes de caractères**

Objectif du bloc :
Manipuler des tableaux (déclaration, remplissage, transformation) et des chaînes de caractères (saisie, analyse, transformations simples).

Exercice 1 – Miroir d'un tableau

1. Écrire un programme qui :
    - permet à l'utilisateur de remplir un **tableau d'entiers** de taille N (connue),
    - affiche ce tableau.
2. Ecrire une fonction qui **inverse l'ordre** des éléments **dans le même tableau**, sans tableau secondaire.
3. Écrire une fonction pour **afficher** un tableau.
4. Écrire un main qui teste le tout.

Exercice 2 – Fusion de deux tableaux triés

On suppose que l'on dispose de deux tableaux d'entiers **déjà triés dans l'ordre croissant**.

1. Écrire un programme qui fusionne ces deux tableaux en un **troisième tableau trié**.
2. La taille du tableau de fusion doit être adaptée (taille = n1 + n2).
3. Écrire une fonction de fusion :

```
void fusion(const int a[], int na, const int b[], int nb, int res[]);
```

Exercice 3 – Chaînes : miroir de phrase et nombre de mots

1. Écrire un programme qui lit une **phrase** entrée par l'utilisateur (via fgets) et l'affiche **à l'envers** (caractère par caractère).
2. Écrire un programme qui lit une phrase et **compte le nombre de mots** (séparés par des espaces).
3. Proposer un sous-programme :

   ```
   int compter_mots(const char *s);
   ```

   et le tester sur plusieurs phrases.

🧩 BLOC 5 – Types structurés (struct)

Objectif du bloc :
Apprendre à définir des types structurés (struct) et à les manipuler avec des tableaux et des sous-programmes.

Exercice 1 – Structure Produit

Un grossiste vend quatre types de produits : carte mère, processeur, barrette mémoire, carte graphique.

Chaque produit possède :

- un **code produit** (entier),
- une **référence** (entier),
- un **prix en euros**,
- une **quantité disponible**.

1. Définir une structure Produit adaptée.
2. Écrire un sous-programme qui permet de **saisir** un produit au clavier.
3. Écrire un sous-programme qui **affiche** les informations d'un produit.
4. Écrire un sous-programme qui, à partir d'un code produit et d'une quantité commandée, affiche :
    - les infos du produit,
    - le **montant total** de la commande.

Exercice 2 – Groupe d'utilisateurs

On veut stocker les informations d'utilisateurs d'un système :

- login
- mot de passe
- entité (AF, EADS, SAIPEM, CNRS, etc.)
- date de création (format AAAAMMJJ ou JJMMAA).

1. Déterminer la structure de données la plus appropriée (struct + tableau de structures).

2. Écrire une fonction qui permet de **saisir** une liste d'utilisateurs (en conservant l'ordre d'entrée ou en triant).
3. Écrire une fonction qui affiche :
    - tous les utilisateurs d'une **entité** donnée,
    - tous les utilisateurs créés à une **date donnée**.

Exercice 3 – Notes de TP

On souhaite stocker les notes de TP d'un groupe de **12 étudiants**.

1. Déterminer une structure pour stocker : nom de l'étudiant + note de TP.
2. Écrire un programme qui :
    - remplit un tableau de 12 étudiants,
    - affiche toutes les informations,
    - trouve et affiche l'étudiant qui a la **meilleure note**.

🧩 **BLOC 6 — Mémoire dynamique (malloc, free, realloc)**

Exercice 1 — Allocation d'un tableau dynamique

On souhaite lire une taille N au clavier et allouer un tableau dynamique d'entiers :

Objectifs :

1. Saisir N (contrôle : N > 0)
2. Allouer dynamiquement un tableau de N int
3. Remplir le tableau avec des valeurs entrées par l'utilisateur
4. Afficher les valeurs
5. Libérer correctement la mémoire

⚠️ Ajouter un test si malloc == NULL

Exercice 2 — Agrandir un tableau avec realloc()

Reprendre l'exercice précédent et rajouter la fonctionnalité suivante :

Objectifs :

1. Demander à l'utilisateur s'il veut **ajouter** d'autres valeurs
2. S'il répond oui :
    - saisir une nouvelle taille M > N
    - réallouer la mémoire avec realloc()
    - remplir les nouvelles cases du tableau
3. Afficher à nouveau le tableau entier

bonne pratique : utiliser un **pointeur temporaire** pour sécuriser la réallocation

---

Exercice 3 — Gestion d'une liste d'utilisateurs

Créer une structure :

```
typedef struct {
    char nom[30];
```

```
    int age;
} Utilisateur;
```

Objectifs :

1. Lire un nombre initial N
2. Allouer dynamiquement un tableau de N Utilisateur
3. Saisir les informations utilisateurs et les stocker
4. Ajouter **dynamiquement** un utilisateur supplémentaire (via realloc())
5. Afficher la liste entière
6. Libérer correctement la mémoire

🧩 BLOC 7 – Projet structuré : gestion d'une cohorte

Objectif du bloc :
Mettre en pratique struct, tableaux, fonctions et calculs de statistiques.

Exercice 1 – Informations par élève

On souhaite stocker les infos de **55 élèves** d'une cohorte :

- numéro d'étudiant,
- nom,
- 5 notes correspondant à 5 matières,
- les 5 ECTS correspondants (5, 4, 3, 6, 2),
- la **moyenne générale** de l'élève.

1. Définir un type structuré adapté.
2. Écrire un sous-programme qui :
    - demande à l'utilisateur de saisir pour un élève : numéro, nom, 5 notes,
    - calcule sa moyenne générale,
    - stocke le tout dans une structure.

Exercice 2 – Informations par matière

On souhaite maintenant stocker les infos concernant les **5 matières** :

- nom de la matière,
- note minimale,
- note maximale,
- moyenne de la cohorte pour cette matière,
- nom de l'élève **major** de cette matière.

1. Définir un type structuré Matiere.
2. Écrire un sous-programme qui :
    - calcule la **moyenne par matière** (sur l'ensemble des élèves).

3. Écrire un sous-programme qui détermine pour chaque matière :
    - la **note min**,
    - la **note max**.

Exercice 3 – Statistiques globales

1. Écrire un sous-programme qui détermine l'**élève major** pour chaque matière (nom et note).
2. Écrire un sous-programme qui calcule le **nombre d'étudiants en dessous de la moyenne générale**.