









Langage C

COURS COMPLET LANGAGE C

- |  PARTIE 1 : BASES
- |  PARTIE 2 : CONDITIONS & BOUCLES
- |  PARTIE 3 : TABLEAUX & CHAÎNES
- |  PARTIE 4 : POINTEURS
- |  PARTIE 5 : FONCTIONS
- |  PARTIE 6 : FICHIERS
- |  PARTIE 7 : STRUCTURES
- |  ANNEXES

Chapitre 1 — Introduction au langage C

◆ Qu'est-ce que le langage C ?

Le **langage C** est un langage de programmation **rapide, puissant et proche du matériel**.

Il est utilisé pour :

- Créer des systèmes d'exploitation (Windows, Linux...)
- Programmer des microcontrôleurs et des logiciels embarqués
- Écrire des jeux, compilateurs et applications performantes

💡 C est **la base de plusieurs autres langages** : C++, C#, Java, Python...

◆ Premier programme en C

Voici le programme le plus simple :

```
#include <stdio.h> // Bibliothèque pour printf

int main() {
    printf("Bonjour Aya !\n");
    return 0;
}
```

Explication :

- `#include <stdio.h>` → on importe une bibliothèque standard qui contient `printf`
- `int main()` → point d'entrée du programme (tout commence ici)
- `{ ... }` → contient les instructions à exécuter

- `printf("Bonjour Aya !\n");` → affiche du texte à l'écran
`\n` = retour à la ligne
- `return 0;` → indique que tout s'est bien passé

Chapitre 2 — Les variables et les types

◆ Définition

Une **variable** est un espace mémoire qui **stocke une valeur**.

Chaque variable a :

- un **nom**
- un **type** (int, float, char...)
- une **valeur**

◆ Les principaux types en C

Type	Taille approximative*	Valeurs possibles	Exemple de déclaration	Description courte
<code>int</code>	4 octets	-2 147 483 648 à 2 147 483 647	<code>int age = 25;</code>	Entier (positif ou négatif)
<code>short</code>	2 octets	-32 768 à 32 767	<code>short s = 100;</code>	Petit entier
<code>long</code>	4 ou 8 octets	±2 milliards ou plus (dépend du système)	<code>long population = 7000000;</code>	Entier long
<code>long long</code>	8 octets	±9 223 372 036 854 775 807	<code>long long bigNum = 1234567890123;</code>	Très grand entier
<code>float</code>	4 octets	~ ±3.4e-38 à ±3.4e38	<code>float pi = 3.14;</code>	Nombre décimal simple précision
<code>double</code>	8 octets	~ ±1.7e-308 à ±1.7e308	<code>double e = 2.718281828;</code>	Nombre décimal double précision
<code>long double</code>	8, 12 ou 16 octets	Plus grande précision que double	<code>long double x = 3.141592653589793238;</code>	Très haute précision
<code>char</code>	1 octet	-128 à 127 (ou 0 à 255 si <code>unsigned</code>)	<code>char lettre = 'A';</code>	Caractère simple
<code>unsigned int</code>	4 octets	0 à 4 294 967 295	<code>unsigned int n = 100;</code>	Entier positif uniquement
<code>_Bool</code> / <code>bool</code> (C99)	1 octet	0 ou 1	<code>#include <stdbool.h> bool ok = true;</code>	Valeur booléenne (vrai/faux)

- Les tailles peuvent varier selon l'architecture (32-bit ou 64-bit).

◆ Exemple de variables

```
#include <stdio.h>
```

```
int main() {
    int age = 20;
    float taille = 1.65;
    char initiale = 'A';

    printf("Age : %d ans\n", age);
    printf("Taille : %.2f m\n", taille);
    printf("Initiale : %c\n", initiale);

    return 0;
}
```

Remarques :

- `%d` pour afficher un `int`
- `%f` pour un `float`
- `%c` pour un `char`
- `%.2f` → afficher 2 chiffres après la virgule
-

Un **tableau complet des types de données en C avec leurs formats d'affichage pour `printf`** :

Type	Taille approximative	Exemple de valeur	Format <code>printf</code>	Exemple <code>printf</code>	Description courte
<code>int</code>	4 octets	42	<code>%d</code> ou <code>%i</code>	<code>printf("%d", age);</code>	Entier
<code>unsigned int</code>	4 octets	100	<code>%u</code>	<code>printf("%u", n);</code>	Entier positif uniquement
<code>short</code>	2 octets	300	<code>%hd</code>	<code>printf("%hd", s);</code>	Petit entier
<code>unsigned short</code>	2 octets	500	<code>%hu</code>	<code>printf("%hu", s);</code>	Petit entier positif
<code>long</code>	4 ou 8 octets	1234567890	<code>%ld</code>	<code>printf("%ld", l);</code>	Entier long
<code>unsigned long</code>	4 ou 8 octets	4000000000	<code>%lu</code>	<code>printf("%lu", l);</code>	Entier long positif
<code>long long</code>	8 octets	1234567890123	<code>%lld</code>	<code>printf("%lld", ll);</code>	Très grand entier
<code>unsigned long long</code>	8 octets	9000000000000	<code>%llu</code>	<code>printf("%llu", ll);</code>	Très grand entier positif
<code>float</code>	4 octets	3.14	<code>%f</code>	<code>printf("%f", pi);</code>	Nombre décimal simple précision

Type	Taille approximative	Exemple de valeur	Format <code>printf</code>	Exemple <code>printf</code>	Description courte
<code>double</code>	8 octets	2.718281828	<code>%lf</code> ou <code>%f</code>	<code>printf("%lf", e);</code>	Nombre décimal double précision
<code>long double</code>	8, 12 ou 16 octets	3.141592653589793238	<code>%Lf</code>	<code>printf("%Lf", x);</code>	Très haute précision
<code>char</code>	1 octet	'A'	<code>%c</code>	<code>printf("%c", lettre);</code>	Caractère
<code>char[]</code>	selon taille	"Bonjour"	<code>%s</code>	<code>printf("%s", str);</code>	Chaîne de caractères
<code>_Bool</code> / <code>bool</code>	1 octet	0 ou 1	<code>%d</code>	<code>printf("%d", ok);</code>	Valeur booléenne (0=false, 1=true)

💡 Remarques importantes :

- Pour `float`, `%f` est utilisé, mais en `scanf` il faut utiliser `%f` pour `float` et `%lf` pour `double`.
- Pour `long double`, il faut `%Lf`.
- Les chaînes de caractères (`char[]`) ne doivent pas dépasser la taille du tableau alloué.
- Les entiers peuvent être signés (`int`) ou non signés (`unsigned int`).

abc Chapitre 3 — Entrées et sorties (`scanf` et `printf`)

◆ `printf()` → afficher

Tu l'as déjà utilisé pour afficher du texte.

◆ `scanf()` → lire une valeur saisie par l'utilisateur

```
#include <stdio.h>

int main() {
    int age;
    printf("Entrez votre age : ");
    scanf("%d", &age); // & : adresse mémoire où stocker la valeur

    printf("Vous avez %d ans\n", age);
    return 0;
}
```

⚠ Le **symbole** `&` est très important :

il indique **où stocker la valeur entrée** (on y reviendra avec les pointeurs).

Résumé de la partie 1

Élément	Rôle
<code>#include <stdio.h></code>	permet d'utiliser <code>printf</code> et <code>scanf</code>
<code>int main()</code>	fonction principale
<code>printf()</code>	affiche du texte
<code>scanf()</code>	lit une valeur saisie
<code>%d</code> , <code>%f</code> , <code>%c</code> , <code>%s</code>	formats pour afficher/lire des variables
<code>\n</code>	retour à la ligne

Exercices pratiques

Exercice 1 :

Écris un programme qui demande à l'utilisateur :

- son nom
 - son âge
- et affiche :

| Bonjour [nom], vous avez [âge] ans.

Exemple attendu :

Entrez votre nom : Aya
Entrez votre age : 20
Bonjour Aya, vous avez 20 ans.


Exercice 2 :


Écris un programme qui demande deux nombres et affiche leur **somme**.

Exemple attendu :

Entrez le premier nombre : 5
Entrez le deuxième nombre : 8
La somme est : 13

Souhaites-tu que je te montre les **corrections des exercices** avant de passer à la **Partie 2 : Conditions, boucles et opérateurs** ?

Super Aya 

On passe maintenant à la  **Partie 2 : Conditions, boucles et opérateurs**

Tu vas apprendre ici **comment faire des choix et répéter des actions** dans un programme — c'est le cœur de toute logique en programmation 🧠

PARTIE 2 — Conditions, boucles et opérateurs


Objectifs de cette partie

- Comprendre les **conditions** (`if` , `else` , `switch`)
- Maîtriser les **boucles** (`for` , `while` , `do...while`)
- Connaître les **opérateurs** (arithmétiques, logiques, comparaisons)
- Créer des **programmes interactifs** (calcul, décision, répétition)

1. Les opérateurs

◆ Opérateurs arithmétiques

Opérateur	Signification	Exemple	Résultat
<code>+</code>	addition	<code>5 + 3</code>	<code>8</code>
<code>-</code>	soustraction	<code>5 - 3</code>	<code>2</code>
<code>*</code>	multiplication	<code>5 * 3</code>	<code>15</code>
<code>/</code>	division	<code>6 / 2</code>	<code>3</code>
<code>%</code>	reste de la division entière	<code>5 % 2</code>	<code>1</code>

 Exemple :

```
int a = 7, b = 3;
printf("a + b = %d\n", a + b);
printf("a %% b = %d\n", a % b);
```

◆ Opérateurs de comparaison

Opérateur	Signification	Exemple	Résultat
<code>==</code>	égal à	<code>5 == 5</code>	vrai
<code>!=</code>	différent de	<code>5 != 3</code>	vrai
<code>></code>	supérieur à	<code>7 > 4</code>	vrai
<code><</code>	inférieur à	<code>2 < 8</code>	vrai
<code>>=</code>	supérieur ou égal à	<code>5 >= 5</code>	vrai
<code><=</code>	inférieur ou égal à	<code>3 <= 4</code>	vrai

◆ Opérateurs logiques

Opérateur	Signification	Exemple	Résultat
&&	ET logique	(a > 0 && b > 0)	vrai si les 2 sont vrais
,		,	OU logique
!	NON logique	!(a > 0)	inverse vrai/faux

2. Les conditions **if** , **else** , **else if**

◆ Structure de base

```
if (condition) {  
    // instructions si vrai  
} else {  
    // instructions si faux  
}
```

◆ Exemple :

```
int age;  
printf("Entrez votre âge : ");  
scanf("%d", &age);  
  
if (age >= 18) {  
    printf("Vous êtes majeur.\n");  
} else {  
    printf("Vous êtes mineur.\n");  
}
```

◆ Chaîne de conditions (**else if**)

```
int note;  
printf("Entrez votre note : ");  
scanf("%d", &note);  
  
if (note >= 16) {  
    printf("Très bien !\n");  
} else if (note >= 10) {  
    printf("Moyen.\n");  
} else {
```

```
printf("Insuffisant.\n");  
}
```

Instruction **switch case**

◆ Définition

L'instruction `switch` permet de **tester une variable** contre plusieurs valeurs possibles de manière plus lisible que des `if/else if` multiples.

◆ Syntaxe

```
switch (variable) {  
    case valeur1:  
        // instructions  
        break;  
    case valeur2:  
        // instructions  
        break;  
    default:  
        // instructions si aucun cas  
}
```

◆ Exemple Simple

```
#include <stdio.h>  
  
int main() {  
    int jour = 3;  
  
    switch (jour) {  
        case 1: printf("Lundi\n"); break;  
        case 2: printf("Mardi\n"); break;  
        case 3: printf("Mercredi\n"); break;  
        default: printf("Jour invalide\n");  
    }  
    return 0;  
}
```

◆ Avec char


```
char grade = 'B';

switch (grade) {
    case 'A': printf("Excellent\n"); break;
    case 'B': printf("Très bien\n"); break;
    case 'C': printf("Bien\n"); break;
    default: printf("Grade invalide\n");
}
```

◆ Cas multiples sans break

```
int note = 15;

switch (note) {
    case 20: case 19: case 18:
        printf("Excellent\n");
        break;
    case 17: case 16: case 15:
        printf("Très bien\n");
        break;
    default:
        printf("Autre\n");
}
```

🧠 Points Importants

- `break` : essentiel pour sortir du switch
- `default` : exécuté si aucun cas ne correspond
- Sans `break` : exécution continue au cas suivant

📊 Quand utiliser switch vs if/else

Condition	Recommandation
Égalité simple	<code>switch</code>
Comparaisons	<code>if/else</code>
Tests complexes	<code>if/else</code>

✅ **Switch case ajouté !** La partie 2 est maintenant complète avec tous les types de conditions. 🎉

🧠 Résumé Mis à Jour de la Partie 2

Élément	Description	Exemple
if/else	Conditions simples ou complexes	if (age >= 18)
else if	Chaîne de conditions	else if (note >= 10)
switch	Tests d'égalité multiples	switch (jour)
case	Valeur possible dans switch	case 1:
break	Sort du switch	break;
default	Cas par défaut dans switch	default:

3. Les boucles (répétitions)

a) Boucle **while**

Répète une action tant que la condition est vraie.

```
int i = 1;
while (i <= 5) {
    printf("%d\n", i);
    i++; // incrémentation
}
```

Résultat :

```
1
2
3
4
5
```

b) Boucle **do...while**

Elle s'exécute au moins une fois, même si la condition est fausse.

```
int x = 1;
do {
    printf("%d\n", x);
    x++;
} while (x <= 5);
```

c) Boucle **for**

Très utilisée quand on connaît le nombre de répétitions.

```
for (int i = 1; i <= 5; i++) {  
    printf("%d\n", i);  
}
```

Exemple pratique complet :

◆ Programme : afficher les nombres pairs de 1 à 10

```
#include <stdio.h>  
  
int main() {  
    for (int i = 1; i <= 10; i++) {  
        if (i % 2 == 0) {  
            printf("%d est pair\n", i);  
        }  
    }  
    return 0;  
}
```

4. Exemple combiné (boucles + conditions)

◆ Exemple : calcul de la somme de 5 nombres

```
#include <stdio.h>  
  
int main() {  
    int nombre, somme = 0;  
  
    for (int i = 1; i <= 5; i++) {  
        printf("Entrez le nombre %d : ", i);  
        scanf("%d", &nombre);  
        somme += nombre;  
    }  
  
    printf("La somme totale est : %d\n", somme);  
    return 0;  
}
```

Résumé de la Partie 2

Élément	Description	Exemple
<code>if</code>	teste une condition	<code>if (a > b)</code>
<code>else</code>	sinon	<code>else { ... }</code>
<code>else if</code>	plusieurs tests	<code>else if (a == b)</code>
<code>for</code>	boucle comptée	<code>for(i=0;i<10;i++)</code>
<code>while</code>	tant que c'est vrai	<code>while(x>0)</code>
<code>do...while</code>	s'exécute au moins une fois	<code>do{...}while(x>0)</code>
<code>&&</code> , <code>!</code>		<code>,</code> <code>!</code>

Exercices pratiques

Exercice 1 :

Demande à l'utilisateur un nombre et indique s'il est **pair** ou **impair**.

Exercice 2 :

Demande à l'utilisateur un nombre `n` et affiche les **n premiers entiers** (avec une boucle `for`).

Exercice 3 :

Demande à l'utilisateur de saisir **5 notes**, puis affiche la **moyenne** et si l'élève est **admis** (moyenne \geq 10).

Souhaites-tu que je te montre les **corrections de ces exercices**, ou qu'on passe directement à la **Partie 3 : Tableaux et chaînes de caractères** ?

Parfait Aya 🙌 ! On continue alors avec la **Partie 3 : Tableaux et chaînes de caractères**.

Ici, tu vas apprendre à **stocker plusieurs données** dans une seule variable et à **manipuler du texte** — ce qui est essentiel pour gérer des listes, des mots ou même des phrases en C.

PARTIE 3 — Tableaux et chaînes de caractères

Objectifs de cette partie

- Comprendre les **tableaux (arrays)**
- Manipuler des **chaînes de caractères**
- Apprendre les **fonctions utiles pour les tableaux et chaînes**
- Créer des programmes capables de **stocker, trier et afficher des données**

1. Les tableaux (arrays)

◆ Définition

Un **tableau** est une variable qui peut contenir **plusieurs valeurs du même type**.

Syntaxe générale :

```
type nomTableau[taille];
```

- `type` : int, float, char...
- `nomTableau` : nom du tableau
- `taille` : nombre d'éléments que le tableau peut contenir

◆ Exemple : tableau d'entiers

```
#include <stdio.h>

int main() {
    int notes[5]; // tableau de 5 entiers

    // remplissage
    notes[0] = 12;
    notes[1] = 15;
    notes[2] = 9;
    notes[3] = 18;
    notes[4] = 14;

    // affichage
    for (int i = 0; i < 5; i++) {
        printf("Note %d : %d\n", i+1, notes[i]);
    }

    return 0;
}
```

📌 Résultat :

```
Note 1 : 12
Note 2 : 15
Note 3 : 9
Note 4 : 18
Note 5 : 14
```

◆ Tableau pré-rempli

```
int jours[7] = {1, 2, 3, 4, 5, 6, 7};
```

Tu peux aussi laisser le compilateur calculer la taille automatiquement :

```
int jours[] = {1, 2, 3, 4, 5, 6, 7}; // taille = 7
```

◆ Tableaux multidimensionnels

Un tableau **2D** est comme un tableau de tableaux.

Exemple : une **matrice 3×3**

```
int matrice[3][3] = {
    {1, 2, 3},
    {4, 5, 6},
    {7, 8, 9}
};

for (int i = 0; i < 3; i++) {
    for (int j = 0; j < 3; j++) {
        printf("%d ", matrice[i][j]);
    }
    printf("\n");
}
```

Résultat :

```
1 2 3
4 5 6
7 8 9
```

2. Les chaînes de caractères

En C, une **chaîne de caractères** est un **tableau de** `char` terminé par un caractère spécial `\0` (fin de chaîne).

◆ Déclaration et initialisation

```
char nom[20]; // tableau de 20 caractères
char prenom[] = "Aya"; // initialisation directe
```

📌 Important : toujours prévoir une case pour `\0` si tu initialises manuellement.

◆ Lecture et affichage

```
#include <stdio.h>

int main() {
    char nom[30];

    printf("Entrez votre nom : ");
    scanf("%s", nom); // lit jusqu'au premier espace

    printf("Bonjour %s !\n", nom);

    return 0;
}
```

📌 Résultat si on saisit "Aya" :

Bonjour Aya !

◆ Fonctions utiles pour les chaînes (`string.h`)

Fonction	Description
<code>strlen(str)</code>	longueur de la chaîne
<code>strcpy(dest, src)</code>	copie <code>src</code> dans <code>dest</code>
<code>strcat(dest, src)</code>	concatène <code>src</code> à la fin de <code>dest</code>
<code>strcmp(s1, s2)</code>	compare deux chaînes (<code>0</code> si identiques)
<code>strstr(s1, s2)</code>	cherche <code>s2</code> dans <code>s1</code>

📌 Exemple :

```
#include <stdio.h>
#include <string.h>

int main() {
    char nom[20] = "Aya";
    char prenom[20] = "Mojahid";

    printf("Nom : %s\n", nom);
    printf("Longueur : %lu\n", strlen(nom));

    // concaténation
    strcat(nom, " ");
    strcat(nom, prenom);
}
```

```
printf("Nom complet : %s\n", nom);

return 0;
}
```

Résultat :

```
Nom : Aya
Longueur : 3
Nom complet : Aya Mojahid
```

3. Tableaux de chaînes de caractères

Pour stocker plusieurs noms :

```
#include <stdio.h>

int main() {
    char noms[3][20] = {"Aya", "Omar", "Lina"};

    for (int i = 0; i < 3; i++) {
        printf("Nom %d : %s\n", i+1, noms[i]);
    }

    return 0;
}
```

Résultat :

```
Nom 1 : Aya
Nom 2 : Omar
Nom 3 : Lina
```

4. Exemple pratique combiné

◆ Programme : demander 5 noms et les afficher

```
#include <stdio.h>

int main() {
    char noms[5][30];
```



```

for (int i = 0; i < 5; i++) {
    printf("Entrez le nom %d : ", i+1);
    scanf("%s", noms[i]);
}

printf("\nListe des noms :\n");
for (int i = 0; i < 5; i++) {
    printf("%s\n", noms[i]);
}

return 0;
}

```

Résumé de la Partie 3

Élément	Description	Exemple
Tableau 1D	stocke plusieurs valeurs d'un type	<code>int notes[5];</code>
Tableau 2D	tableau de tableaux	<code>int mat[3][3];</code>
Chaîne de caractères	tableau de <code>char</code> avec <code>\0</code>	<code>char nom[20] = "Aya";</code>
<code>strlen()</code>	longueur de chaîne	<code>strlen(nom)</code>
<code>strcpy()</code>	copie de chaîne	<code>strcpy(dest, src)</code>
<code>strcat()</code>	concaténation	<code>strcat(nom, prenom)</code>
<code>strcmp()</code>	comparaison de chaînes	<code>strcmp(a, b)</code>

le fichier `<string.h>`

En C, le fichier `<string.h>` contient **une vingtaine de fonctions importantes** pour manipuler les chaînes de caractères. Je vais te donner une **liste complète des fonctions les plus utiles**, avec explications et exemples simples pour chacune.

Fonctions du `<string.h>`

1 `strlen()`

Description : Retourne la longueur d'une chaîne (sans compter le `\0`).

```

#include <stdio.h>
#include <string.h>

int main() {
    char mot[] = "Aya";

```

```
printf("Longueur : %lu\n", strlen(mot)); // 3
return 0;
}
```

2 strcpy(dest, src)

Description : Copie la chaîne `src` dans `dest` .

```
char source[] = "Bonjour";
char destination[20];
strcpy(destination, source);
printf("%s\n", destination); // Bonjour
```

⚠ Attention : `dest` doit être assez grand pour contenir `src` .

3 strncpy(dest, src, n)

Description : Copie **au plus** `n` caractères de `src` dans `dest` .

```
char src[] = "Bonjour";
char dest[20];
strncpy(dest, src, 3);
dest[3] = '\0'; // toujours terminer par \0
printf("%s\n", dest); // Bon
```

4 strcat(dest, src)

Description : Concatène `src` à la fin de `dest` .

```
char a[20] = "Aya";
char b[] = " Mojahid";
strcat(a, b);
printf("%s\n", a); // Aya Mojahid
```

5 strncat(dest, src, n)

Description : Concatène **au plus** `n` caractères de `src` à `dest` .

```
char a[20] = "Aya";
char b[] = " Mojahid";
strncat(a, b, 3);
printf("%s\n", a); // Aya Moj
```

6 strcmp(s1, s2)

Description : Compare deux chaînes.

- Retourne 0 si identiques
- < 0 si `s1 < s2`
- 0 si `s1 > s2`

```
char a[] = "Aya";
char b[] = "Omar";
int res = strcmp(a, b);
printf("%d\n", res); // négatif
```

7 strncmp(s1, s2, n)

Description : Compare les `n` premiers caractères de deux chaînes.

```
strncmp("Bonjour", "Bonheur", 3); // 0 car "Bon" = "Bon"
```

8 strchr(str, c)

Description : Cherche le premier caractère `c` dans `str`.

- Retourne un pointeur sur `c` ou `NULL` si non trouvé.

```
char mot[] = "Aya";
char *p = strchr(mot, 'y');
printf("%c\n", *p); // y
```

9 strrchr(str, c)

Description : Cherche le dernier caractère `c` dans `str`.

```
char mot[] = "Aya AyA";
char *p = strrchr(mot, 'A');
printf("%s\n", p); // A
```

10 strstr(haystack, needle)

Description : Cherche la sous-chaîne `needle` dans `haystack`.

```
char texte[] = "Bonjour Aya";
char *p = strstr(texte, "Aya");
```

```
printf("%s\n", p); // Aya
```

1 1 `strspn(str1, str2)`

Description : Retourne le **nombre de caractères consécutifs de** `str1` **présents dans** `str2` à partir du début.

```
strspn("123abc", "0123456789"); // 3 (les 3 premiers sont des chiffres)
```

1 2 `strcspn(str1, str2)`

Description : Retourne le **nombre de caractères consécutifs de** `str1` **absents de** `str2` au début.

```
strcspn("abc123", "0123456789"); // 3 (abc avant le premier chiffre)
```

1 3 `strpbrk(str1, str2)`

Description : Retourne un pointeur sur le **premier caractère de** `str1` **présent dans** `str2`.

```
char *p = strpbrk("abc123", "345"); // '3'
```

1 4 `strtok(str, delim)`

Description : Divise une chaîne en **tokens** (mots) selon un séparateur `delim`.

```
char str[] = "Bonjour, Aya";
char *token = strtok(str, ",");
while (token != NULL) {
    printf("%s\n", token);
    token = strtok(NULL, ",");
}
// Bonjour
// Aya
```

1 5 `memset()` , `memcpy()` , `memcmp()` , `memmove()`

Ces fonctions sont **similaires mais pour la mémoire brute**, souvent utilisées avec les chaînes :

Fonction	Description
<code>memset(ptr, val, n)</code>	Remplit <code>n</code> octets à partir de <code>ptr</code> avec <code>val</code>
<code>memcpy(dest, src, n)</code>	Copie <code>n</code> octets de <code>src</code> vers <code>dest</code>

Fonction	Description
<code>memcmp(ptr1, ptr2, n)</code>	Compare <code>n</code> octets
<code>memmove(dest, src, n)</code>	Copie <code>n</code> octets de <code>src</code> vers <code>dest</code> en gérant le chevauchement

16 `strcasecmp(s1, s2)`

Description :

Compare **deux chaînes sans tenir compte des majuscules / minuscules** (non standard mais très utilisée sur Linux/Unix).

- Retourne `0` si les deux chaînes sont identiques (même lettres, peu importe la casse)
- `< 0` si `s1 < s2`
- `0` si `s1 > s2`

 Nécessite d'inclure :

```
#include <strings.h> // (pas <string.h> !)
```

Exemple :

```
#include <stdio.h>
#include <strings.h> // pour strcasecmp()

int main() {
    char a[] = "AYA";
    char b[] = "aya";

    if (strcasecmp(a, b) == 0)
        printf("Les deux chaînes sont identiques (insensible à la casse)\n");
    else
        printf("Les chaînes sont différentes\n");

    return 0;
}
```

Remarques :

- `strcasecmp()` existe sur les systèmes POSIX (Linux, macOS).
- Sur Windows, la fonction équivalente est `_stricmp()` :

```
#include <string.h>
_stricmp(a, b);
```

Si tu veux, je peux te faire **un super tableau résumé complet avec toutes les fonctions** `string.h` + **exemple rapide pour chacune**, pour que tu aies un **guide ultra pratique à garder sous les yeux**.
Veux-tu que je fasse ça ?

Exercice 1

Demande à l'utilisateur **5 nombres** et affiche le **plus grand**.

Exercice 2

Demande à l'utilisateur **3 prénoms** et affiche-les **en ordre inverse**.

Exercice 3

Demande à l'utilisateur un **mot** et affiche le **nombre de lettres** (utiliser `strlen`).

Exercice 4

Concatène deux chaînes saisies par l'utilisateur et affiche le **résultat complet**.

PARTIE 4 — Pointeurs et tableaux

Objectifs de cette partie

- Comprendre ce qu'est un **pointeur**
 - Savoir déclarer et utiliser un pointeur
 - Apprendre les opérations sur pointeurs (, `&`)
 - Utiliser des pointeurs avec les **tableaux**
 - Maîtriser des exemples pratiques pour la **passation de tableaux aux fonctions**
-

1. Qu'est-ce qu'un pointeur ?

Un **pointeur** est une variable qui **contient l'adresse mémoire** d'une autre variable.

- Chaque variable en C est stockée à une **adresse mémoire**
 - Avec un pointeur, on peut accéder à cette variable **via son adresse**
-

Déclaration d'un pointeur

```
int a = 10; // variable normale
int *p;    // pointeur vers un entier
p = &a;    // p reçoit l'adresse de a
```

📌 Explications :

Symbole	Signification
*	déréférencement (accéder à la valeur pointée)
&	adresse de la variable

◆ Accéder à la valeur via le pointeur

```
#include <stdio.h>

int main() {
    int a = 10;
    int *p = &a;

    printf("Adresse de a : %p\n", p);    // adresse mémoire
    printf("Valeur de a : %d\n", *p);    // valeur pointée par p

    *p = 20; // modifier la valeur de a via le pointeur
    printf("Nouvelle valeur de a : %d\n", a);

    return 0;
}
```

Résultat :

```
Adresse de a : 0x7ffee2f1c8ac
Valeur de a : 10
Nouvelle valeur de a : 20
```

✅ Ici, modifier `*p` change directement `a`.

🧠 2. Pointeurs et tableaux

En C, un **tableau** est en réalité un pointeur vers son **premier élément**.

◆ Exemple simple

```
#include <stdio.h>

int main() {
    int tab[5] = {10, 20, 30, 40, 50};
    int *p = tab; // équivalent à &tab[0]

    printf("Premier élément : %d\n", *p); // 10
}
```

```
printf("Deuxième élément : %d\n", *(p + 1)); // 20
printf("Troisième élément : %d\n", *(p + 2)); // 30

return 0;
}
```

📌 Explications :

- `p = tab` → p pointe vers `tab[0]`
- `(p + i)` → accès à l'élément `tab[i]`

◆ Boucle avec pointeur

```
for (int i = 0; i < 5; i++) {
    printf("%d ", *(p + i));
}
```

Equivalent à :

```
for (int i = 0; i < 5; i++) {
    printf("%d ", tab[i]);
}
```

✅ Les pointeurs permettent donc de **parcourir un tableau sans utiliser directement l'indice**.

🧠 3. Passer un tableau à une fonction avec un pointeur

◆ Exemple : calcul de la somme d'un tableau

```
#include <stdio.h>

int somme(int *tab, int taille) {
    int s = 0;
    for (int i = 0; i < taille; i++) {
        s += *(tab + i); // ou tab[i]
    }
    return s;
}

int main() {
    int nombres[5] = {1, 2, 3, 4, 5};
    int total = somme(nombres, 5);

    printf("Somme : %d\n", total);
}
```



```
    return 0;
}
```

✅ Ici, le tableau est passé **par adresse**, donc la fonction peut **modifier le tableau si nécessaire**.

4. Pointeurs et chaînes de caractères

Une chaîne de caractères est en réalité un **tableau de char**, donc on peut utiliser des pointeurs.

◆ Exemple : parcourir une chaîne

```
#include <stdio.h>

int main() {
    char mot[] = "Aya";
    char *p = mot;

    while (*p != '\0') {
        printf("%c\n", *p);
        p++;
    }

    return 0;
}
```

Résultat :

```
A
y
a
```

✅ On peut donc **parcourir une chaîne comme un tableau**, mais avec la **flexibilité des pointeurs**.

5. Pointeurs et modification de tableau dans une fonction

◆ Exemple : doubler tous les éléments

```
#include <stdio.h>

void doubler(int *tab, int taille) {
    for (int i = 0; i < taille; i++) {
        *(tab + i) *= 2; // modifier directement le tableau
    }
}
```

```

}

int main() {
    int nombres[5] = {1, 2, 3, 4, 5};

    doubler(nombres, 5);

    for (int i = 0; i < 5; i++) {
        printf("%d ", nombres[i]);
    }

    return 0;
}

```

Résultat :

2 4 6 8 10

6. Résumé des pointeurs

Concept	Explication
Pointeur <code>*p</code>	contient l'adresse d'une variable
<code>*p</code>	valeur pointée par <code>p</code>
<code>&var</code>	adresse de <code>var</code>
Tableaux et pointeurs	<code>tab = &tab[0]</code>
Accès tableau avec pointeur	<code>*(tab + i) = tab[i]</code>
Passer tableau à une fonction	<code>fonction(tab, taille)</code> → modifie directement le tableau
Chaîne de caractères	tableau de <code>char</code> → peut utiliser pointeur pour parcourir

Exercices pratiques

Exercice 1

Créer un tableau de 5 entiers, utiliser un pointeur pour **afficher tous les éléments**.

Exercice 2

Écrire une fonction qui prend un tableau et **le transforme en tableau inversé** (dernier élément devient premier).

Exercice 3

Demander un mot à l'utilisateur et **afficher chaque lettre sur une ligne** en utilisant un pointeur.

Exercice 4

Écrire une fonction qui **double tous les nombres d'un tableau** (comme l'exemple ci-dessus) mais utiliser la notation `tab[i]` et la notation `*(tab + i)` pour comprendre les deux méthodes.



PARTIE 5 - FONCTIONS

FONCTIONS EN C — COURS COMPLET AVEC EXPLICATIONS



1. Définition d'une fonction

Une **fonction** est un **bloc de code réutilisable** qui effectue une tâche précise.

- Elle peut **recevoir des informations** (paramètres)
- Elle peut **retourner un résultat** (ou ne rien retourner avec `void`)

Avantages :

- Réutilisation du code
- Organisation claire
- Facilite la maintenance et le débogage

◆ 2. Fonctions avec retour et paramètres

Explication :

On veut une fonction qui **calcule quelque chose et renvoie le résultat**.

Ici, par exemple, on veut **additionner deux nombres** et récupérer la somme dans `main`.

```
#include <stdio.h>

// Déclaration de la fonction
int addition(int a, int b) {
    return a + b; // retourne la somme des deux nombres
}

int main() {
    int res = addition(5, 3); // appel de la fonction
    printf("Résultat : %d\n", res);
    return 0;
}
```

✓ Résultat : 8

🔗 Explication : `int` = type de retour, `a` et `b` = paramètres. `return` renvoie la valeur au programme appelant.

◆ 3. Fonctions sans retour (`void`)

Explication :

Parfois, on veut **effectuer une action** (comme afficher un message) **sans renvoyer de valeur**.

Dans ce cas, on utilise `void`.

```
#include <stdio.h>

void afficherMessage() {
    printf("Bonjour Aya !\n"); // action sans retour
}

int main() {
    afficherMessage(); // on appelle la fonction
    return 0;
}
```

Résultat :

```
Bonjour Aya !
```

🔗 Explication : `void` signifie que la fonction ne retourne rien. Elle fait juste quelque chose.

◆ 4. Fonction avec paramètres et sans retour

Explication :

On veut **passer des informations à la fonction** pour qu'elle fasse quelque chose avec ces informations, mais **sans renvoyer de résultat**.

Exemple : afficher la somme de deux nombres directement dans la fonction.

```
#include <stdio.h>

void afficherSomme(int a, int b) {
    printf("Somme : %d\n", a + b); // calcul + affichage
}

int main() {
    afficherSomme(5, 7); // appel avec arguments
}
```

```
    return 0;
}
```

Résultat :

Somme : 12

🔗 Explication : La fonction reçoit `a` et `b`, les utilise pour l'affichage, mais ne renvoie rien.

◆ 5. Fonction sans paramètre mais avec retour

Explication :

On veut une fonction qui **fournit une valeur fixe ou calculée**, mais **sans besoin de passer des données**.

```
#include <stdio.h>

int getNombre() {
    return 42; // valeur fixe renvoyée
}

int main() {
    printf("Nombre : %d\n", getNombre()); // récupère la valeur
    return 0;
}
```

Résultat :

Nombre : 42

🔗 Explication : Même sans paramètres, la fonction peut renvoyer une valeur avec `return`.

◆ 6. Passage de paramètres : valeur vs adresse

a) Passage par valeur

Explication :

On envoie **une copie** de la variable à la fonction. Toute modification **ne change pas** la variable originale.

```
#include <stdio.h>

void changerValeur(int x) {
```

```

    x = 100; // modification locale
}

int main() {
    int a = 50;
    changerValeur(a); // passe une copie
    printf("%d\n", a); // reste 50
    return 0;
}

```

Résultat :

50

b) Passage par adresse (pointeur)

Explication :

Pour **modifier la variable originale**, on envoie son **adresse**. La fonction utilise un pointeur.

```

#include <stdio.h>

void changerValeur(int *x) {
    *x = 100; // modification directe
}

int main() {
    int a = 50;
    changerValeur(&a); // passe l'adresse
    printf("%d\n", a); // devient 100
    return 0;
}

```

Résultat :

100

📌 Explication : `*x` = valeur pointée, `&a` = adresse de `a`.

◆ 7. Fonctions et tableaux

Explication :

Les tableaux sont toujours passés **par adresse**, donc une fonction peut **modifier directement les éléments**.

```
#include <stdio.h>

void doubler(int tab[], int taille) {
    for(int i = 0; i < taille; i++) {
        tab[i] *= 2; // double chaque élément
    }
}

int main() {
    int tab[5] = {1,2,3,4,5};
    doubler(tab, 5); // modification directe
    for(int i=0;i<5;i++)
        printf("%d ", tab[i]);
    return 0;
}
```

Résultat :

2 4 6 8 10

🔗 Explication : Pas besoin de `return` : le tableau est modifié en place.

◆ 8. Fonctions et chaînes de caractères

Explication :

Une chaîne est un **tableau de caractères**. La fonction peut **parcourir ou modifier** la chaîne.

```
#include <stdio.h>

void afficherLettres(char mot[]) {
    for(int i=0; mot[i]!='\0'; i++)
        printf("%c\n", mot[i]); // affiche chaque lettre
}

int main() {
    char nom[] = "Aya";
    afficherLettres(nom);
    return 0;
}
```

Résultat :

A
y
a

◆ 9. Fonction qui inverse un tableau

Explication :

On veut une fonction qui **inverse un tableau entier** (dernier devient premier) **en modifiant le tableau directement**.

```
#include <stdio.h>

void inverser(int tab[], int taille) {
    for(int i = 0; i < taille/2; i++) {
        int temp = tab[i];
        tab[i] = tab[taille-i-1];
        tab[taille-i-1] = temp;
    }
}

int main() {
    int nombres[5] = {1,2,3,4,5};
    inverser(nombres, 5); // inverse le tableau
    for(int i=0;i<5;i++)
        printf("%d ", nombres[i]);
    return 0;
}
```

Résultat :

5 4 3 2 1

📌 Explication : La fonction échange les éléments **du début vers la fin**, modifiant le tableau original.

◆ 10. Résumé complet

Type de fonction	Paramètres	Retour	Exemple d'utilisation
Avec paramètre et retour	Oui	Oui	<code>int addition(int a,int b)</code>
Avec paramètre sans retour	Oui	Non (<code>void</code>)	<code>void afficherSomme(int a,int b)</code>
Sans paramètre avec retour	Non	Oui	<code>int getNombre()</code>
Sans paramètre sans retour	Non	Non (<code>void</code>)	<code>void afficherBonjour()</code>

Exercices pratiques

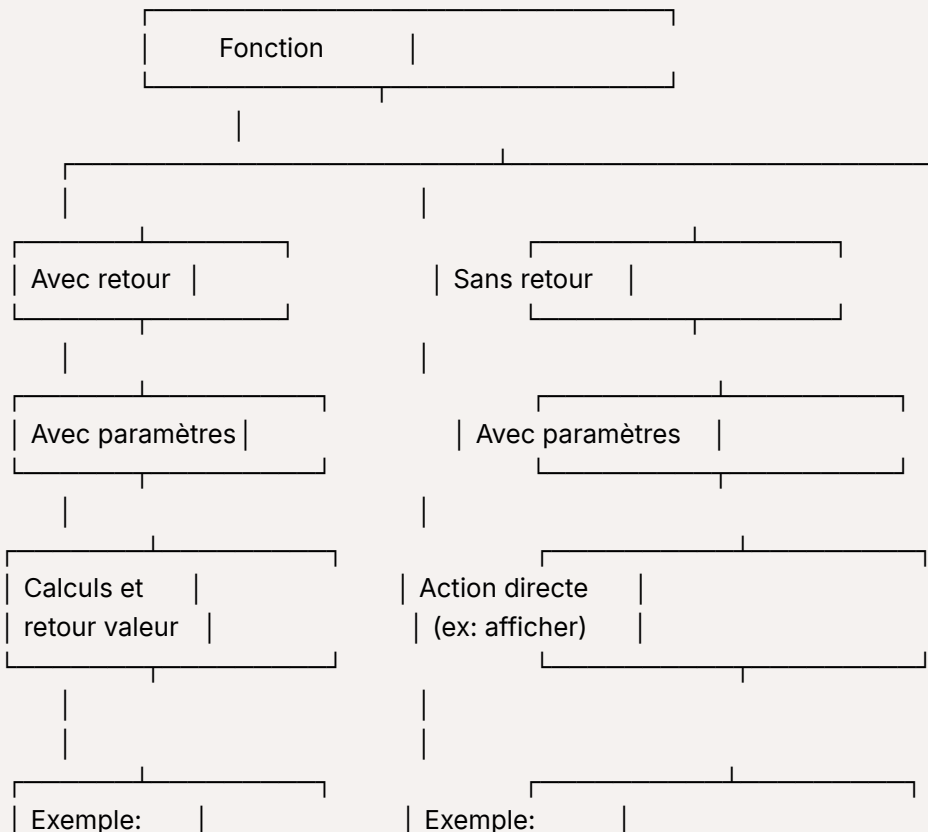
1. Écrire une fonction **qui affiche un tableau**.
2. Écrire une fonction **qui renvoie le maximum d'un tableau**.
3. Écrire une fonction **qui inverse un tableau**.
4. Écrire une fonction **qui double les éléments d'un tableau**, en utilisant `tab[i]` et `(tab+i)`.
5. Écrire une fonction **qui affiche chaque lettre d'un mot** saisi par l'utilisateur.

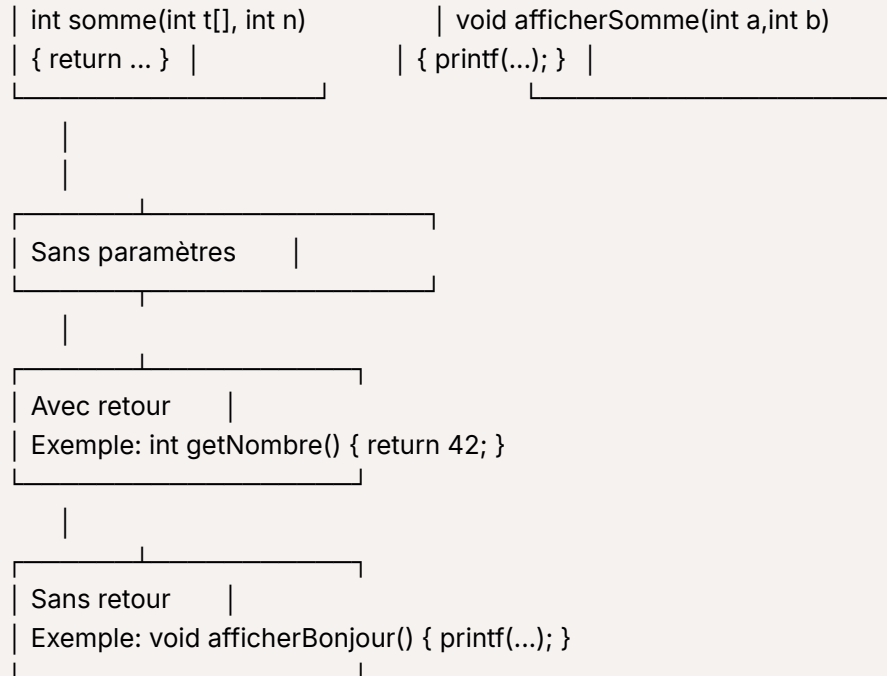
un **schéma visuel complet des fonctions en C** qui regroupe **tous les cas** :

- Fonctions avec ou sans retour
- Fonctions avec ou sans paramètres
- Passage par valeur et par adresse
- Tableaux et pointeurs

Voici le **schéma sous forme textuelle et facile à visualiser** (on peut le transformer en diagramme graphique après si tu veux) :

Schéma des fonctions en C





◆ Passage des paramètres et tableaux

Paramètres passés à la fonction

- Passage par valeur
 - Copie de la variable
 - Variable originale non modifiée
 - Exemple: void changerValeur(int x)
- Passage par adresse (pointeur)
 - Envoie l'adresse de la variable
 - Fonction peut modifier la variable originale
 - Exemple: void changerValeur(int *x)

Tableaux

- Toujours passés par adresse
- Fonction peut modifier directement le tableau
- Exemple: void doubler(int tab[], int taille)

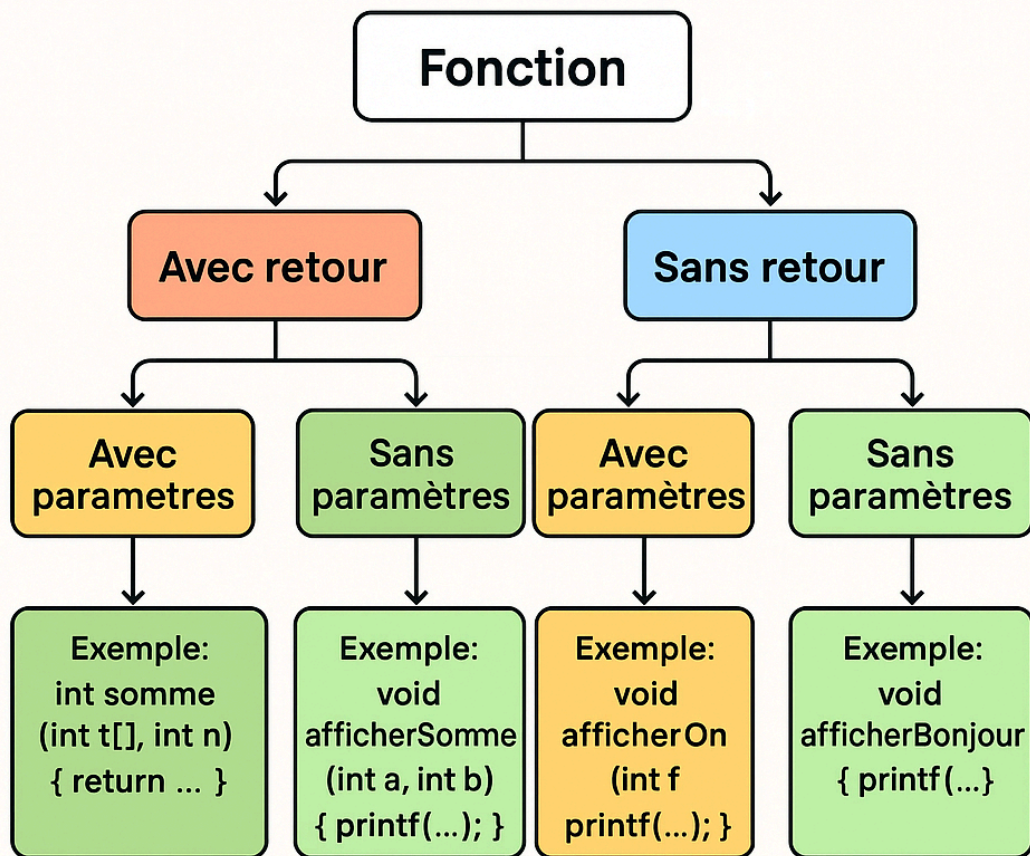
◆ Chaînes de caractères

Une chaîne = tableau de char

- |
- |— On peut passer la chaîne à une fonction
- |— On peut parcourir ou modifier la chaîne
- |— Exemple:
void afficherLettres(char mot[])

Résumé visuel global

- **Fonctions** : Avec/Sans retour, Avec/Sans paramètres
- **Paramètres** : Passage par valeur ou par adresse
- **Tableaux & chaînes** : Passés toujours par adresse → modifiables
- **Pointeurs** : Nécessaires pour modifier la variable originale ou parcourir un tableau



Paramètres passés à la fonction

- **Passage par valeur**
 - Copie de la variable
 - Variable originale non modifiée
 - Exemple: `void changerValeur(int x)`
- **Passage par adresse (pointeur)**
 - Envoie l'adresse de la variable
 - Fonction peut modifier la variable originale
 - Exemple: `void changerValeur(int *x)`
- **Tableaux**
 - Toujours passés par adresse
 - Fonction peut modifier directement le tableau
 - Exemple: `void doubler(int tab[], int taille)`
- **Chaines de caractères**
 - Une chaîne = tableau de char

Variable locale and variable globale:

Type	Où déclarée	Accessible où	Durée de vie	Exemple
Locale	Dans une fonction	Seulement dans cette fonction	Temporaire (disparaît à la fin de la fonction)	<code>int x = 10;</code>
Globale	En dehors des fonctions	Partout dans le code	Permanente (tant que le programme tourne)	<code>int y = 5;</code>

COURS COMPLET — Fonctions Mathématiques en C (`<math.h>`)

1. Introduction

En C, les fonctions mathématiques **ne sont pas incluses automatiquement**.

Elles se trouvent dans la **bibliothèque** `math.h`.

👉 Donc, pour les utiliser, il faut toujours écrire :

```
#include <math.h>
```

Et quand tu compiles avec gcc, tu dois souvent ajouter `-lm` :

```
gcc monProgramme.c -o monProgramme -lm
```

2. Quelques règles importantes

- Toutes ces fonctions travaillent avec le **type** `double` (valeurs à virgule).
- Si tu donnes un entier (`int`), il sera automatiquement converti en `double`.
- Résultat aussi : souvent en `double`.

3. Les principales fonctions de `math.h`

◆ 1. `sqrt(x)` — Racine carrée

Calcule la **racine carrée** d'un nombre.

```
#include <stdio.h>
#include <math.h>

int main() {
    double x = 25.0;
```

```
printf("Racine carrée de %.2f = %.2f\n", x, sqrt(x));  
return 0;  
}
```


 **Résultat :**

Racine carrée de 25.00 = 5.00

◆ 2. `pow(x, y)` — Puissance

Calcule (x^y) (x à la puissance y).

```
double a = 2.0, b = 3.0;  
printf("%.0f puissance %.0f = %.2f\n", a, b, pow(a, b));
```


 **Résultat :**

2 puissance 3 = 8.00

◆ 3. `fabs(x)` — Valeur absolue (pour double)

Renvoie la **valeur positive** d'un nombre, même s'il est négatif.

```
double n = -8.6;  
printf("Valeur absolue de %.2f = %.2f\n", n, fabs(n));
```


 **Résultat :**

Valeur absolue de -8.60 = 8.60

◆ 4. `ceil(x)` — Arrondir vers le haut

Arrondit à l'**entier supérieur**.

```
double n = 3.2;  
printf("ceil(%.2f) = %.2f\n", n, ceil(n));
```


 **Résultat :**

ceil(3.20) = 4.00

◆ 5. `floor(x)` — Arrondir vers le bas

Arrondit à l'entier inférieur.


```
double n = 3.8;
printf("floor(%.2f) = %.2f\n", n, floor(n));
```

 Résultat :

```
floor(3.80) = 3.00
```

◆ 6. `round(x)` — Arrondir à l'entier le plus proche

```
double n1 = 4.6, n2 = 4.4;
printf("round(%.2f) = %.2f\n", n1, round(n1));
printf("round(%.2f) = %.2f\n", n2, round(n2));
```


 Résultat :

```
round(4.60) = 5.00
round(4.40) = 4.00
```

◆ 7. `fmod(x, y)` — Reste de la division réelle

Donne le **reste** de la division `x / y`.

```
double a = 10.0, b = 3.0;
printf("fmod(%.0f, %.0f) = %.2f\n", a, b, fmod(a, b));
```


 Résultat :

```
fmod(10, 3) = 1.00
```

◆ 8. `sin(x)`, `cos(x)`, `tan(x)` — Trigonométrie

⚠ Ces fonctions utilisent **les radians**, pas les degrés.

```
double angle = 3.14159 / 2; // ≈ 90°
printf("sin(90°) = %.2f\n", sin(angle));
printf("cos(90°) = %.2f\n", cos(angle));
printf("tan(90°) = %.2f\n", tan(angle));
```

 Résultat :

```
sin(90°) = 1.00
cos(90°) = 0.00
tan(90°) = 16331239353195370.00 (≈ infini)
```

◆ 9. `log(x)` et `log10(x)`

- `log(x)` → logarithme **népérien (base e)**
- `log10(x)` → logarithme **base 10**

```
double x = 100.0;
printf("log(%.0f) = %.2f\n", x, log(x));
printf("log10(%.0f) = %.2f\n", x, log10(x));
```

🧠 Résultat :

```
log(100) = 4.61
log10(100) = 2.00
```

◆ 10. `exp(x)` — Exponentielle (e^x)

```
double x = 2.0;
printf("exp(%.2f) = %.2f\n", x, exp(x));
```

🧠 Résultat :

```
exp(2.00) = 7.39
```

■ 4. Exemple global (toutes les fonctions ensemble)

```
#include <stdio.h>
#include <math.h>

int main() {
    double x = 25.0;

    printf("sqrt(%.2f) = %.2f\n", x, sqrt(x));
    printf("pow(2, 3) = %.2f\n", pow(2, 3));
    printf("fabs(-8.6) = %.2f\n", fabs(-8.6));
    printf("ceil(3.2) = %.2f\n", ceil(3.2));
    printf("floor(3.8) = %.2f\n", floor(3.8));
    printf("round(4.5) = %.2f\n", round(4.5));
```



```

printf("fmod(10, 3) = %.2f\n", fmod(10, 3));
printf("sin(pi/2) = %.2f\n", sin(3.14159 / 2));
printf("log(100) = %.2f\n", log(100));
printf("log10(100) = %.2f\n", log10(100));
printf("exp(2) = %.2f\n", exp(2));

return 0;
}

```

Résultat (selon machine) :

```

sqrt(25.00) = 5.00
pow(2, 3) = 8.00
fabs(-8.6) = 8.60
ceil(3.2) = 4.00
floor(3.8) = 3.00
round(4.5) = 5.00
fmod(10, 3) = 1.00
sin(pi/2) = 1.00
log(100) = 4.61
log10(100) = 2.00
exp(2) = 7.39

```

Résumé visuel

Fonction	Rôle	Exemple	Résultat
<code>sqrt(x)</code>	Racine carrée	<code>sqrt(9)</code>	3
<code>pow(x,y)</code>	Puissance	<code>pow(2,3)</code>	8
<code>fabs(x)</code>	Valeur absolue	<code>fabs(-4.5)</code>	4.5
<code>ceil(x)</code>	Arrondi haut	<code>ceil(2.3)</code>	3
<code>floor(x)</code>	Arrondi bas	<code>floor(2.9)</code>	2
<code>round(x)</code>	Arrondi proche	<code>round(2.6)</code>	3
<code>fmod(x,y)</code>	Reste réel	<code>fmod(10,3)</code>	1
<code>sin(x)</code>	Sinus	<code>sin(pi/2)</code>	1
<code>log(x)</code>	Log base e	<code>log(100)</code>	4.61
<code>log10(x)</code>	Log base 10	<code>log10(100)</code>	2
<code>exp(x)</code>	Exponentielle	<code>exp(2)</code>	7.39

PARTIE 6 -LES FICHIERS EN C

1. Qu'est-ce qu'un fichier en C ?

Un **fichier** est une zone de mémoire sur le disque où sont stockées des données.

En C, pour manipuler un fichier, tu dois :

1. **L'ouvrir** (open)
2. **Lire / Écrire** dedans
3. **Le fermer**

👉 C'est comme lire un livre :

Tu dois l'ouvrir 📖, lire ou écrire ✍️, puis le refermer 🗄️.

2. La bibliothèque nécessaire

Pour travailler avec les fichiers, on doit inclure :

```
#include <stdio.h>
```

C'est cette bibliothèque qui contient toutes les fonctions comme `fopen` , `fprintf` , `fscanf` , etc.

3. Les étapes principales

◆ Étape 1 : Déclarer un pointeur vers un fichier

```
FILE *fichier;
```

Ici, `FILE` est un **type spécial** (défini dans `stdio.h`) qui contient des informations sur ton fichier.

◆ Étape 2 : Ouvrir le fichier

On utilise la fonction :

```
fopen("nom_du_fichier", "mode");
```

Exemples de modes :

Mode	Signification	Exemple
"r"	Lecture seule	ouvrir un fichier existant pour le lire
"w"	Écriture seule (écrase le contenu s'il existe)	créer ou remplacer un fichier
"a"	Ajout (append)	écrire à la fin d'un fichier existant
"r+"	Lecture et écriture	le fichier doit exister
"w+"	Lecture et écriture (efface tout avant)	crée un nouveau fichier
"a+"	Lecture et ajout	ajoute sans effacer

◆ Étape 3 : Vérifier si l'ouverture a réussi

```
if (fichier == NULL) {  
    printf("Erreur : impossible d'ouvrir le fichier.\n");  
    return 1;  
}
```

◆ Étape 4 : Lire ou écrire dans le fichier

Il existe plusieurs fonctions selon ce que tu veux faire 📌

📁 4. Écriture dans un fichier

✍ Exemple 1 — Écrire du texte simple

```
#include <stdio.h>  
  
int main() {  
    FILE *fichier = fopen("exemple.txt", "w"); // ouvrir en écriture  
  
    if (fichier == NULL) {  
        printf("Erreur d'ouverture du fichier.\n");  
        return 1;  
    }  
  
    fprintf(fichier, "Bonjour Aya !\n");  
    fprintf(fichier, "Ceci est un test d'écriture dans un fichier.\n");  
  
    fclose(fichier); // fermer le fichier  
    printf("Écriture terminée ✅\n");  
    return 0;  
}
```

🧠 Explication :

- `fprintf()` fonctionne comme `printf()`, mais écrit dans un fichier.
- `fclose()` ferme le fichier et libère la mémoire.

📖 5. Lecture d'un fichier

📖 Exemple 2 — Lire un fichier ligne par ligne

```
#include <stdio.h>

int main() {
    FILE *fichier = fopen("exemple.txt", "r"); // ouvrir en lecture
    char ligne[100]; // tableau pour stocker chaque ligne

    if (fichier == NULL) {
        printf("Erreur : impossible d'ouvrir le fichier.\n");
        return 1;
    }

    while (fgets(ligne, 100, fichier) != NULL) { // lire chaque ligne
        printf("%s", ligne); // afficher la ligne lue
    }

    fclose(fichier);
    return 0;
}
```

Explication :

- `fgets()` lit une ligne complète depuis le fichier.
- Elle s'arrête soit à la fin de la ligne, soit après 99 caractères (ici).
- Le `while` continue tant qu'il y a des lignes à lire.

6. Écriture et lecture de données formatées

Exemple 3 — Écrire puis relire des nombres

```
#include <stdio.h>

int main() {
    FILE *fichier = fopen("nombres.txt", "w");
    if (fichier == NULL) return 1;

    int a = 10, b = 20;
    fprintf(fichier, "%d %d\n", a, b);
    fclose(fichier);

    // Lire les valeurs
    fichier = fopen("nombres.txt", "r");
    if (fichier == NULL) return 1;
}
```

```

int x, y;
fscanf(fichier, "%d %d", &x, &y);
printf("Valeurs lues : %d et %d\n", x, y);

fclose(fichier);
return 0;
}

```

7. Ajouter à un fichier sans effacer

Exemple 4 — Mode append "a"

```

#include <stdio.h>

int main() {
    FILE *fichier = fopen("log.txt", "a");
    if (fichier == NULL) return 1;

    fprintf(fichier, "Nouvelle entrée ajoutée.\n");
    fclose(fichier);

    printf("Texte ajouté sans effacer l'ancien contenu.\n");
    return 0;
}

```

8. Lire caractère par caractère

```

#include <stdio.h>

int main() {
    FILE *fichier = fopen("exemple.txt", "r");
    if (fichier == NULL) return 1;

    char c;
    while ((c = fgetc(fichier)) != EOF) { // EOF = End Of File
        putchar(c);
    }

    fclose(fichier);
    return 0;
}

```

9. Résumé des fonctions principales

Fonction	Utilité
<code>fopen("nom", "mode")</code>	Ouvrir un fichier
<code>fclose(f)</code>	Fermer le fichier
<code>fprintf(f, "...")</code>	Écrire du texte
<code>fscanf(f, "...")</code>	Lire du texte formaté
<code>fgets(str, n, f)</code>	Lire une ligne
<code>fgetc(f)</code>	Lire un caractère
<code>fputc(c, f)</code>	Écrire un caractère
<code>feof(f)</code>	Tester la fin du fichier

10. Bonnes pratiques

- ✓ Toujours vérifier si `fopen()` a réussi.
- ✓ Toujours fermer le fichier avec `fclose()`.
- ✓ Utiliser des modes adaptés (`w`, `a`, `r`, etc.).
- ✓ Ne pas oublier de gérer les erreurs de lecture/écriture.

PARTIE 7 : STRUCTURES EN C

1. Définition et utilité

Une **structure** (mot-clé : `struct`) est un **type de donnée personnalisé** que tu peux créer pour regrouper plusieurs **informations de types différents** dans une seule entité.

Exemple simple

Tu veux stocker les informations d'un étudiant :

- nom → chaîne de caractères
- âge → entier
- note → réel

Tu pourrais créer 3 variables séparées :

```
char nom[30];
int age;
float note;
```

Mais si tu veux manipuler **plusieurs étudiants**, ce sera vite compliqué.

👉 La structure résout ce problème.

2. Définition d'une structure

Syntaxe générale :

```
struct NomStructure {  
    type1 membre1;  
    type2 membre2;  
    type3 membre3;  
    ...  
};
```

Exemple concret :

```
struct Etudiant {  
    char nom[30];  
    int age;  
    float note;  
};
```

Ici, `Etudiant` est un **type de structure** qui contient trois membres :

- `nom` (chaîne)
- `age` (entier)
- `note` (réel)

3. Créer et utiliser une structure

◆ Déclaration d'une variable de type struct :

```
struct Etudiant e1;
```

◆ Remplir les données :

```
strcpy(e1.nom, "Aya"); // strcpy vient de <string.h>  
e1.age = 20;  
e1.note = 16.5;
```

◆ Afficher les données :

```
printf("Nom: %s | Age: %d | Note: %.2f\n", e1.nom, e1.age, e1.note);
```

Exemple complet 1 — Structure simple

```
#include <stdio.h>
#include <string.h>

struct Etudiant {
    char nom[30];
    int age;
    float note;
};

int main() {
    struct Etudiant e1;

    strcpy(e1.nom, "Aya");
    e1.age = 21;
    e1.note = 17.25;

    printf("Nom: %s | Age: %d | Note: %.2f\n", e1.nom, e1.age, e1.note);
    return 0;
}
```

Explication :

- `struct Etudiant` crée un type structuré pour les étudiants.
- `strcpy()` sert à copier une chaîne dans le champ `nom`.
- Le point `.` permet d'accéder à chaque membre.

4. Initialiser directement une structure

Tu peux donner les valeurs dès la déclaration :

```
struct Etudiant e2 = {"Youssef", 22, 15.80};
```

5. Plusieurs structures — tableau de structures

Souvent, tu veux plusieurs objets du même type (plusieurs étudiants).

Tu peux créer un **tableau de structures** :

```
#include <stdio.h>

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



```

    int age;
    float note;
};

int main() {
    struct Etudiant classe[3] = {
        {"Aya", 21, 17.25},
        {"Rania", 22, 14.50},
        {"Youssef", 23, 15.10}
    };

    for (int i = 0; i < 3; i++) {
        printf("Nom: %s | Age: %d | Note: %.2f\n",
            classe[i].nom, classe[i].age, classe[i].note);
    }

    return 0;
}

```

🧠 Ici :

- `classe[i]` est le i-ème étudiant.
- `classe[i].nom` accède à son nom.
- C'est très utile pour gérer des listes d'objets (étudiants, produits, clients...).

6. Structure et fonction

Tu peux passer une structure à une fonction, comme un paramètre classique.

Exemple :

```

#include <stdio.h>
#include <string.h>

struct Etudiant {
    char nom[30];
    int age;
    float note;
};

// Fonction pour afficher un étudiant
void afficher(struct Etudiant e) {
    printf("Nom: %s | Age: %d | Note: %.2f\n", e.nom, e.age, e.note);
}

```

```
int main() {
    struct Etudiant e1 = {"Aya", 21, 17.25};
    afficher(e1);
    return 0;
}
```

7. Structure avec pointeur → opérateur


Quand tu passes une structure **par adresse**, tu dois utiliser  au lieu de .

```
#include <stdio.h>
#include <string.h>

struct Etudiant {
    char nom[30];
    int age;
    float note;
};

// Fonction qui modifie la note via pointeur
void modifier(struct Etudiant *e) {
    e->note = 18.5;
}

int main() {
    struct Etudiant e1 = {"Aya", 21, 16.0};
    modifier(&e1); // On passe l'adresse de e1
    printf("Nouvelle note de %s = %.2f\n", e1.nom, e1.note);
    return 0;
}
```

 **e->note** signifie "accéder au champ note via le pointeur e".

8. Structure dans une autre structure

Tu peux **imbriquer** des structures (comme une "structure à l'intérieur d'une autre").

Exemple :

```
#include <stdio.h>
#include <string.h>

struct Adresse {
```

```

    char ville[30];
    char pays[30];
};

struct Etudiant {
    char nom[30];
    int age;
    float note;
    struct Adresse adr; // structure imbriquée
};

int main() {
    struct Etudiant e1 = {"Aya", 21, 17.25, {"Rabat", "Maroc"}};

    printf("Nom: %s | Age: %d | Note: %.2f | Ville: %s | Pays: %s\n",
        e1.nom, e1.age, e1.note, e1.adr.ville, e1.adr.pays);
    return 0;
}

```

9. **typedef** pour simplifier le code

typedef te permet de **renommer** un type, donc tu n'as plus besoin d'écrire "struct" à chaque fois.


```

#include <stdio.h>
#include <string.h>

typedef struct {
    char nom[30];
    int age;
    float note;
} Etudiant; // alias pour "struct Etudiant"

int main() {
    Etudiant e1 = {"Aya", 21, 17.25};
    printf("Nom: %s | Age: %d | Note: %.2f\n", e1.nom, e1.age, e1.note);
    return 0;
}

```

 Maintenant tu peux écrire **Etudiant** directement, comme un type normal (comme **int**, **char**, etc.).

10. Structures et fichiers (bonus)

Tu peux combiner **struct + fichiers** pour sauvegarder des données.

Exemple simple :

```
#include <stdio.h>
#include <string.h>

typedef struct {
    char nom[30];
    int age;
    float note;
} Etudiant;

int main() {
    FILE *f = fopen("etudiants.txt", "w");
    Etudiant e1 = {"Aya", 21, 17.25};
    Etudiant e2 = {"Rania", 22, 15.80};

    fprintf(f, "%s %d %.2f\n", e1.nom, e1.age, e1.note);
    fprintf(f, "%s %d %.2f\n", e2.nom, e2.age, e2.note);

    fclose(f);
    printf("✅ Données enregistrées dans etudiants.txt\n");
    return 0;
}
```

11. Résumé du cours

Élément	Rôle
<code>struct</code>	Crée un type personnalisé
<code>.</code>	Accéder à un champ
<code>→</code>	Accéder via pointeur
<code>typedef</code>	Renommer le type
<code>struct dans struct</code>	Imbriquer des structures
<code>tableau de struct</code>	Gérer plusieurs objets
<code>fscanf / fprintf</code>	Lire ou écrire des structures dans un fichier

Exemple final résumé

```
#include <stdio.h>
#include <string.h>

typedef struct {
```

```

char nom[30];
int age;
float note;
} Etudiant;

int main() {
    Etudiant classe[3] = {
        {"Aya", 21, 17.25},
        {"Rania", 22, 14.50},
        {"Youssef", 23, 15.10}
    };

    printf("---- Liste des étudiants ----\n");
    for (int i = 0; i < 3; i++) {
        printf("%d) %s - %d ans - Note: %.2f\n",
            i + 1, classe[i].nom, classe[i].age, classe[i].note);
    }

    return 0;
}

```

Parfait Aya 🍌 ! Voici un **résumé complet et détaillé des structures en C** sous forme de **tableau**, pour que tu puisses **tout voir en un coup d'œil**.

Concept	Définition / Description	Exemple	Notes importantes
<code>struct</code>	Permet de créer un type personnalisé regroupant plusieurs variables de types différents.	<code>struct Etudiant { char nom[30]; int age; float note; };</code>	Chaque variable de la structure est appelée membre .
Déclaration variable	Crée une variable de type structure	<code>struct Etudiant e1;</code>	Permet de stocker les données d'un étudiant.
Accéder aux membres	Utiliser le point <code>.</code> pour accéder aux champs	<code>e1.age = 21; strcpy(e1.nom, "Aya");</code>	Le champ doit être déclaré dans la structure.
Initialisation directe	Définir les valeurs dès la déclaration	<code>struct Etudiant e2 = {"Youssef", 22, 15.8};</code>	Très pratique pour des tableaux ou variables uniques.
Tableau de structures	Créer plusieurs objets du même type	<code>struct Etudiant classe[3];</code>	Permet de gérer des listes (étudiants, produits...).
Boucle sur tableau	Accéder à chaque élément	<code>for(int i=0;i<3;i++) printf("%s", classe[i].nom);</code>	Permet d'afficher ou modifier tous les éléments.
Fonction avec structure	Passer la structure comme paramètre	<code>void afficher(struct Etudiant e) {...}</code>	La structure est copiée (passage par valeur).

Concept	Définition / Description	Exemple	Notes importantes
Pointeur vers structure	Passer par adresse pour modifier	<pre>void modifier(struct Etudiant *e) { e->note = 18.0; }</pre>	→ sert à accéder aux membres via pointeur.
Structure imbriquée	Une structure à l'intérieur d'une autre	<pre>struct Adresse { char ville[30]; char pays[30]; }; struct Etudiant { struct Adresse adr; };</pre>	Permet de mieux organiser les données complexes.
<code>typedef</code>	Renommer une structure pour simplifier le code	<pre>typedef struct { char nom[30]; int age; float note; } Etudiant;</pre>	Plus besoin d'écrire <code>struct</code> à chaque fois.
Fichier + structure	Lire/écrire les structures dans un fichier	<pre>fprintf(f, "%s %d %.2f\n", e.nom, e.age, e.note);</pre>	Combine struct et fichiers pour stocker des données persistantes.
Accès aux champs via pointeur	Utilisation de →	<pre>ptr->age = 20;</pre>	Équivalent à <code>(*ptr).age</code> mais plus lisible.

💡 Résumé en une phrase :

Les structures permettent de regrouper plusieurs informations liées dans un type unique, de les manipuler facilement (tableaux, fonctions, fichiers) et de rendre ton code plus organisé et puissant.