Langage C

> En C, tout doit être déclaré avant d'être utilisé.

1. Organisation générale d'un programme C

Un fichier C se compose toujours des **mêmes grandes parties** :

```c
#include <stdio.h>      // 1. Bibliothèques (includes)
#define PI 3.14159      // 2. Constantes symboliques (facultatif)
int compteurGlobal = 0; // 3. Variables globales (facultatif)

int main(void)          // 4. Fonction principale
{
    // 4.1 Déclaration des variables locales
    int a, b;
    float moyenne;

    // 4.2 Instructions (calculs, saisies, affichages)
    printf("Entrez deux entiers : ");
    scanf("%d %d", &a, &b);

    moyenne = (a + b) / 2.0;

    printf("La moyenne de %d et %d est %.2f\n", a, b, moyenne);

    // 4.3 Retour de la fonction main
    return 0;
}
```

2. Explication des parties du code

2.1. Les *includes*

- Importent des **bibliothèques standard** du C (ou créées par l'utilisateur).
- Exemple :

```c
#include <stdio.h>   // pour printf, scanf
#include <math.h>    // pour sqrt, pow, etc.
```

2.2. Les *constantes et variables globales*

- Accessibles **dans tout le programme**.

- À utiliser avec précaution (elles consomment de la mémoire pendant toute l'exécution).

```
#define TAXE 0.2
int compteur = 0;
```

2.3. La fonction `main()`

C'est **le point d'entrée du programme**.

- Toujours de forme :

  ```
  int main(void) { ... }
  ```

- Peut contenir :
  - Déclaration de variables locales
  - Appels de fonctions
  - Calculs
  - Affichages (`printf`)
  - Saisies clavier (`scanf`)
  - Une valeur de retour (`return 0;`)

2.4. La déclaration des variables

En C, **toutes les variables doivent être déclarées avant utilisation**.

Cela signifie :

- Avant tout calcul ou affichage.
- Au début du bloc `{ ... }` où elles seront utilisées.

Exemple :

```
int a, b;      // déclaration
a = 5;         // affectation
b = a + 2;     // utilisation
```

Exemple incorrect :

```
a = 5;         // ❌ erreur : a non déclaré
int a;
```

---

3. Exemple complet

```c
#include <stdio.h>      // pour printf et scanf
#define TVA 0.20        // constante

// variable globale (accessible partout)
float totalGeneral = 0;

int main(void) {
    // Déclaration des variables locales
    float prixHT, prixTTC;

    // Saisie de la valeur
    printf("Entrez le prix HT : ");
    scanf("%f", &prixHT);

    // Calcul
    prixTTC = prixHT * (1 + TVA);
    totalGeneral = prixTTC;  // on modifie la variable globale

    // Affichage
    printf("Le prix TTC est : %.2f €\n", prixTTC);

    return 0; // fin du programme
}
```

---

4. Résumé

Un programme C minimal ressemble à ceci :

```c
#include <stdio.h>

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

- `#include <stdio.h>` : inclusion d'une bibliothèque standard (ici, pour `printf`).
- `int main(void)` : point d'entrée du programme (toujours présent).
- `return 0;` : indique que le programme s'est terminé correctement.

Élément	Rôle	Exemple
`#include`	Charger des bibliothèques	`#include <stdio.h>`
Constantes	Valeurs fixes	`#define PI 3.14`
Variables globales	Utilisables partout	`int compteur;`

Élément	Rôle	Exemple
`main()`	Fonction principale	`int main(void) { ... }`
Déclaration	Avant utilisation	`int a; float b;`
`return 0;`	Fin normale du programme	—

Exercice :

> Écrire un programme C qui affiche :
>
> ```
> Bonjour, je révise le langage C !
> ```
>
> On doit respecter la structure : includes, main avec return 0.

Variables et types de base

En C, une variable est **déclarée avec un type** :

```c
int age = 20;
float temperature = 36.6;
char lettre = 'A';
```

Types fondamentaux :

Type	Taille (approx.)	Exemple	Description
`int`	4 octets	`int x = 5;`	Nombre entier
`float`	4 octets	`float y = 2.5;`	Nombre réel simple
`double`	8 octets	`double z = 3.14159;`	Nombre réel précis
`char`	1 octet	`char c = 'a';`	Caractère unique

Exercice :

> Déclarer et initialiser :
>
> - une variable entière `a = 5`
> - une variable flottante `b = 2.5`
> - affiche leur somme avec `printf`.
>
> Exemple attendu :
>
> ```
> a = 5, b = 2.5, somme = 7.5
> ```

Lecture au clavier

On utilise `scanf` :

```c
int age;
printf("Quel est ton âge ? ");
scanf("%d", &age);  // %d → int
printf("Tu as %d ans\n", age);
```

Spécificateurs courants :

Type	Spécificateur	Exemple
int	%d	scanf("%d", &x);
float	%f	scanf("%f", &x);
char	%c	scanf(" %c", &x);
double	%lf	scanf("%lf", &x);

Exercice :

> Écrire un programme qui demande à l'utilisateur :
>
> - son prénom (un mot)
> - son âge
> - puis affiche :
>
> ```
> Bonjour <prénom>, tu as <âge> ans !
> ```

Structure d'un programme complet

```c
#include <stdio.h>

int main(void) {
    int a, b, somme;

    printf("Entrez deux entiers : ");
    scanf("%d %d", &a, &b);

    somme = a + b;
    printf("La somme de %d et %d est %d.\n", a, b, somme);
```

```
    return 0;
}
```

Les constantes

```
#define PI 3.14159
const int MAX = 100;
```

> Elles servent à rendre le code plus lisible et modifiable sans erreur.

Conditions en C

La structure `if` / `else`

- Permet d'exécuter un bloc de code **si une condition est vraie**.
- La condition doit toujours être **entre parenthèses** et renvoyer un booléen (`0 = faux`, `!=0 = vrai`).

```
#include <stdio.h>

int main(void) {
    int note;
    printf("Entrez votre note : ");
    scanf("%d", &note);

    if (note >= 10) {
        printf("Vous avez réussi !\n");
    } else {
        printf("Vous avez échoué.\n");
    }

    return 0;
}
```

`else if` pour plusieurs cas

```
if (note >= 16) {
    printf("Excellent !\n");
} else if (note >= 12) {
    printf("Bien !\n");
} else if (note >= 10) {
    printf("Passable.\n");
} else {
```

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

Remarque : Les conditions sont évaluées **dans l'ordre** ; le premier bloc vrai s'exécute et le reste est ignoré.

Boucles en C

La boucle `while` (tant que)

- Répète un bloc **tant qu'une condition est vraie**.

```c
int i = 1;
while (i <= 5) {
    printf("%d\n", i);
    i++;  // n'oubliez pas d'incrémenter sinon boucle infinie !
}
```

La boucle `for` (pour)

- Très pratique quand on connaît **le nombre d'itérations** à l'avance.

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

Syntaxe : `for(initialisation; condition; mise à jour)`

La boucle `do ... while`

- **Exécute au moins une fois**, puis teste la condition.

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

Résumé

1. **Conditions**
    - `if` → si vrai

- `else if` → sinon si vrai
- `else` → sinon

2. **Boucles**
   - `while` : condition testée avant chaque tour
   - `do ... while` : condition testée après chaque tour
   - `for` : pratique pour compteur connu
3. Toujours éviter les **boucles infinies** : vérifier l'incrémentation et la condition.
4. Indentation claire = lisibilité + moins d'erreurs.

Exercices

1. Écrire un programme qui demande un nombre à l'utilisateur et indique s'il est **positif, négatif ou nul**.
2. Écrire un programme qui affiche les nombres de 1 à 10 avec une **boucle `for`**.
3. Écrire un programme qui demande un mot de passe et **répète la saisie tant qu'il est incorrect** (boucle `do...while`).

Concept	Syntaxe	Exemple	Remarques
Si / sinon	`if (condition) { ... } else { ... }`	`c if (note >= 10) { printf("Réussi"); } else { printf("Échoué"); }`	Évalue la condition. `else` optionnel.
Sinon si	`else if (condition)`	`c if (note >=16) {...} else if (note>=12){...} else {...}`	Plusieurs conditions testées dans l'ordre.
while (tant que)	`while(condition) { ... }`	`c int i=1; while(i<=5) {printf("%d\n",i); i++;}`	Condition testée **avant** chaque itération.
do ... while	`do { ... } while(condition);`	`c int i=1; do { printf("%d\n",i); i++; } while(i<=5);`	Boucle exécutée **au moins une fois**.
for (pour)	`for(initialisation; condition; mise-à-jour) { ... }`	`c for(int i=1;i<=5;i++) {printf("%d\n",i);}`	Très pratique pour un nombre d'itérations connu.
Remarques	–	–	1. Attention aux boucles infinies 2. Indentation = lisibilité 3. Conditions évaluées dans l'ordre

Tableaux (Arrays)

- Un **tableau** permet de stocker plusieurs valeurs du même type sous **un seul nom**.
- Les **indices commencent à 0** : le premier élément est `tab[0]`.

Déclaration :

```c
int notes[5];       // tableau de 5 entiers non initialisés
int ages[3] = {12, 15, 14}; // tableau initialisé
```

- **Accès et modification d'un élément** :

```c
notes[0] = 10;      // premier élément
printf("%d\n", notes[2]);  // affiche le 3e élément
```

- **Boucle pour parcourir un tableau** :

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

> Attention : ne jamais dépasser la taille (`i < taille`).

Exercice :

> - Créer un tableau `entiers[5]` et demander à l'utilisateur de saisir les 5 valeurs.
> - Afficher ensuite la somme des éléments.

Corrigé

```c
#include <stdio.h>  // Inclusion de la bibliothèque standard pour les entrées/sorties

int main() {
    int entiers[5];   // Déclaration d'un tableau de 5 entiers
    int somme = 0;    // Variable pour stocker la somme des valeurs

    // Boucle pour demander à l'utilisateur de saisir les 5 valeurs
    for (int i = 0; i < 5; i++) {
        printf("Entrez un entier #%d : ", i + 1);  // Affiche le numéro de l'entrée
        scanf("%d", &entiers[i]);                  // Lit la valeur saisie par l'utilisateur
        somme += entiers[i];                       // Ajoute la valeur à la somme
```

```
    }

    // Affiche le résultat final
    printf("La somme des éléments est : %d\n", somme);

    return 0;  // Indique que le programme s'est terminé correctement
}
```

Fonctions

- Une fonction **regroupe un bloc d'instructions réutilisable**.
- **Prototype / déclaration** (avant `main` si la définition vient après) :

```
int somme(int a, int b);
```

- **Définition** :

```
int somme(int a, int b){
    return a + b;
}

void afficherBonjour(){
    printf("Bonjour !\n");
}
```

- **Appel d'une fonction** :

```
int resultat = somme(5, 7);
afficherBonjour();
```

- Les **paramètres sont passés par valeur**, sauf les tableaux (qui sont passés par référence).
- Une fonction peut être de type `void` si elle **ne retourne rien**.

Représentation synthétique :

1. Sans prototype (fonction AVANT main)

```
#include <stdio.h>

// Définition de la fonction
type_retour nom_fonction(paramètres) {
    // corps de la fonction
}
```

```
int main() {
    // corps du main
    // appel à nom_fonction()
}
```

2. Avec prototype (fonction APRÈS main)

```
#include <stdio.h>

// Prototype de la fonction
type_retour nom_fonction(paramètres);

int main() {
    // corps du main
    // appel à nom_fonction()
}

// Définition de la fonction après main
type_retour nom_fonction(paramètres) {
    // corps de la fonction
}
```

- Dans la version sans prototype, la fonction doit obligatoirement être écrite avant le `main` pour être connue lors de son appel.
- Dans la version avec prototype, seule la “signature” (le prototype) est placée avant le `main`; ensuite, la définition complète de la fonction peut venir après le `main`, ce qui rend le code plus lisible et modulaire.

Exercice — Illustration des fonctions avec et sans prototype

Objectif

Comprendre la différence entre une fonction **définie sans prototype** et une fonction **déclarée avec prototype** en C. Les deux approches donnent le même résultat, mais que la version avec prototype est **plus sûre et plus propre**.

Consignes

On souhaite écrire un programme qui calcule la somme de deux entiers à l’aide d’une fonction `addition()`. On va écrire **deux versions** du programme :

1. **Version 1 : sans prototype**
2. **Version 2 : avec prototype**
3. Écrire un programme en C qui :

- Demande à l'utilisateur deux entiers ;
- Calcule leur somme à l'aide d'une fonction `addition()` ;
- Affiche le résultat à l'écran.

4. Réaliser **deux versions** :

   - **Version 1 :** sans prototype (la fonction est définie avant `main()` et non déclarée avant).
   - **Version 2 :** avec prototype (la fonction est déclarée avant `main()`).

5. Comparer les deux programmes :

   - Le résultat est-il le même ?
   - Quelle version est la plus correcte selon vous ? Pourquoi ?

Version 1 — Sans prototype

Dans ce premier programme, nous définissons et utilisons la fonction `addition` sans la déclarer au préalable (pas de prototype avant `main()`).

```c
#include <stdio.h>

// Définition directe de la fonction (pas de prototype avant main)
int addition(int x, int y) {
    return x + y;
}

int main() {
    int a, b, resultat;

    printf("Entrez deux entiers : ");
    scanf("%d %d", &a, &b);

    resultat = addition(a, b);  // Appel de la fonction sans prototype
    printf("La somme de %d et %d est : %d\n", a, b, resultat);

    return 0;
}
```

Explications :

- La fonction `addition()` n'a pas été **déclarée** avant `main()` mais elle est définie avant `main()`.
- **Comportement** : Le programme demande à l'utilisateur de saisir deux entiers, puis il affiche leur somme en appelant `addition()`.
- Cela fonctionne, mais c'est une **mauvaise pratique** : → risque d'erreurs dans des programmes plus complexes.

Version 2 — Avec prototype

Dans ce deuxième programme, nous définissons la fonction `addition` avec un **prototype avant** `main()`.

```c
#include <stdio.h>

// Déclaration (prototype) de la fonction
int addition(int x, int y);

int main() {
    int a, b, resultat;

    printf("Entrez deux entiers : ");
    scanf("%d %d", &a, &b);

    resultat = addition(a, b);  // Appel de la fonction avec prototype
    printf("La somme de %d et %d est : %d\n", a, b, resultat);

    return 0;
}

// Définition de la fonction après main()
int addition(int x, int y) {
    return x + y;
}
```

Explications :

- Le **prototype** (`int addition(int x, int y);`) est une **déclaration anticipée** de la fonction.
- Il informe le compilateur du type de retour et des paramètres attendus.
- Cela permet au compilateur de **vérifier la cohérence** entre l'appel et la définition.
- **Comportement** : Le programme est similaire au premier, mais ici, la fonction est déclarée avant `main()`, permettant de mieux organiser le code, surtout dans des programmes plus complexes.
- C'est la **bonne pratique** à adopter dans tout projet.

Conclusion :

Les deux programmes fonctionnent et donnent le même résultat, **mais seule la version avec prototype est correcte selon les standards du langage C.**

Toujours déclarer les prototypes de vos fonctions avant `main()`, surtout dans les projets comportant plusieurs fichiers (`.h` et `.c`).

Exercice

Écrire un programme en langage C qui :

- Demande à l'utilisateur de saisir un nombre entier.
- Utilise une fonction nommée `estImpair` pour déterminer si ce nombre est impair.
- Affiche à l'utilisateur si le nombre est pair ou impair.

Vous écrirez deux versions de ce programme :

1. Une version où la fonction `estImpair` est écrite avant la fonction `main` (sans prototype).
2. Une version où vous déclarez le prototype de `estImpair` avant le `main`, puis écrivez la définition de la fonction après le `main`.

Corrigé 1 : Sans prototype (fonction avant main)

```c
#include <stdio.h>

// Définition de la fonction AVANT main
int estImpair(int nb) {
    return nb % 2;
}

int main() {
    int n;
    printf("Entrez un nombre entier : ");
    scanf("%d", &n);

    if (estImpair(n))
        printf("Le nombre est impair.\n");
    else
        printf("Le nombre est pair.\n");

    return 0;
}
```

Dans cet exemple, comme la fonction est écrite avant le main, il n'y a pas besoin de prototype.

Corrigé 2 : Avec prototype (fonction après main)

```c
#include <stdio.h>

// Prototype de la fonction
int estImpair(int nb);

int main() {
    int n;
    printf("Entrez un nombre entier : ");
    scanf("%d", &n);

    if (estImpair(n))
        printf("Le nombre est impair.\n");
    else
        printf("Le nombre est pair.\n");

    return 0;
```

```c
}

// Définition de la fonction APRES main
int estImpair(int nb) {
    return nb % 2;
}
```

Dans cette seconde version, on déclare le prototype avant main et on écrit la fonction après main. Le compilateur connaît alors la signature de la fonction lors de son appel.

Points essentiels

- Un prototype permet d'utiliser une fonction même si sa définition se trouve après le main et de guider le compilateur.
- Pour des petits programmes, ce n'est pas obligatoire si la fonction est définie avant le main, mais c'est une bonne pratique pour des projets plus grands.

Résumé : 1. Tableaux (Arrays)

Un **tableau** permet de stocker plusieurs valeurs du même type.

Concept	Syntaxe	Exemple	Remarques
Déclaration	`type nom[n];`	`int notes[5];`	`n` = taille du tableau (constante ou variable).
Initialisation	`int notes[5] = {12, 15, 10, 14, 18};`	–	Les valeurs non précisées = 0 par défaut.
Accès à un élément	`nom[i]`	`printf("%d", notes[2]); // 3e élément`	Les indices commencent **à 0** !
Boucle sur tableau	`for(int i=0;i<5;i++){ ... }`	`for(int i=0;i<5;i++) printf("%d\n",notes[i]);`	Très pratique pour afficher ou modifier le tableau.

Remarque : toujours vérifier les **indices** pour éviter les erreurs ("out of bounds").

Résumé : 2. Fonctions

Une **fonction** permet de découper le programme en **blocs réutilisables**.

Concept	Syntaxe	Exemple	Remarques
Déclaration / prototype	`type nom(paramètres);`	`int somme(int a, int b);`	À mettre **avant main** si définition après main.

Concept	Syntaxe	Exemple	Remarques
Définition	`type nom(paramètres){ ... }`	`int somme(int a,int b){ return a+b; }`	Code exécuté lorsque la fonction est appelée.
Appel	`nom(arguments);`	`int s = somme(5,7);`	Peut retourner une valeur ou être `void` (rien à retourner).
Exemple avec `void`	`void afficherMessage() { printf("Hello\n"); }`	`afficherMessage();`	Fonction sans valeur de retour.

Notes :

- Les **paramètres** sont des **copies** (passage par valeur).
- Les **tableaux** passés en paramètre deviennent **références**, donc modifiables dans la fonction.

Structures (`struct`)

- Une **structure** permet de **regrouper plusieurs variables de types différents** sous un seul nom.
- C'est utile pour représenter un **objet ou une entité** (ex : un élève, un point, une voiture).

Déclaration d'une structure :

```c
struct Point {
    int x;
    int y;
};
```

- Ici, `Point` a **deux champs** : `x` et `y`.

Déclaration d'une variable structure :

```c
struct Point p1;   // déclaration d'une variable p1 de type struct Point
p1.x = 10;
p1.y = 20;
```

- On peut aussi **initialiser directement** :

```c
struct Point p2 = {5, 8};
```

- Pour **tableaux de structures** :

```c
struct Point points[3]; // tableau de 3 points
points[0].x = 1;
```

```
points[0].y = 2;
```

Exemples d'utilisation :

- Afficher les coordonnées d'un point :

```
printf("Point : (%d, %d)\n", p1.x, p1.y);
```

- Passer une structure à une fonction :

```
void afficherPoint(struct Point p){
    printf("(%d, %d)\n", p.x, p.y);
}
```

> Remarque : en C, la structure **est copiée** lorsqu'elle est passée à une fonction, donc toute modification dans la fonction ne change pas la variable originale sauf si on passe un **pointeur**.

Exercice :

> - Créer une structure `Eleve` avec les champs :
>
> > `nom` (chaîne de caractères), `age` (int), `note` (float)
> >
> > - Déclarer un élève et initialiser ses champs.
> > - Afficher ensuite les informations de l'élève.

Résumé

1️⃣ Structure générale d'un programme C

```
#include <stdio.h>   // inclusion des bibliothèques

// déclaration de constantes ou variables globales
#define MAX 100

int main(void) {      // point d'entrée
    // déclaration de variables locales
    int x = 0;

    printf("Bonjour C !\n");
    return 0;          // fin du programme
}
```

- `#include` : inclusion des bibliothèques
- `#define` ou `const` : constantes
- `main()` : point d'entrée obligatoire
- `return 0` : programme terminé correctement

2️⃣ Variables et types de base

Type	Exemple	Description
`int`	`int a = 5;`	entier
`float`	`float b = 2.5;`	réel simple
`double`	`double c = 3.14;`	réel précis
`char`	`char d = 'A';`	caractère

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

3️⃣ Lecture et affichage

```
int age;
printf("Quel âge as-tu ? ");
scanf("%d", &age);    // lire un entier
printf("Tu as %d ans\n", age);
```

Spécificateurs courants :

- `%d` → int
- `%f` → float
- `%lf` → double
- `%c` → char
- `%s` → chaîne de caractères

4️⃣ Conditions

```
if (age >= 18) {
    printf("Majeur\n");
} else {
    printf("Mineur\n");
}
```

Note : on peut enchaîner avec `else if` pour plusieurs cas.

5️⃣ Boucles

- `for` : répétition avec compteur

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

- `while` : tant que la condition est vraie

```
int i = 0;
while (i < 5) {
    printf("%d\n", i);
    i++;
}
```

- `do...while` : au moins une fois

```
int i = 0;
do {
    printf("%d\n", i);
    i++;
} while (i < 5);
```

6️⃣ Tableaux

```
int notes[5];          // tableau de 5 entiers
notes[0] = 12;
notes[1] = 15;
```

- Tableaux de caractères : chaîne de caractères

```
char nom[20];
scanf("%s", nom);
```

- Boucle sur un tableau

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

7️⃣ Fonctions

- Déclaration et définition

```c
int somme(int a, int b){
    return a + b;
}

int main(void){
    int res = somme(3, 4);
    printf("Résultat = %d\n", res);
    return 0;
}
```

- Paramètres passés **par valeur** (copie)
- Utiliser **pointeurs** pour passer par référence

```c
void incrementer(int *x){
    (*x)++;
}
```

8️⃣ Structures (`struct`)

```c
struct Point {
    int x;
    int y;
};

struct Point p1 = {3, 4};
printf("Point : (%d, %d)\n", p1.x, p1.y);
```

- Tableaux de structures

```c
struct Point tab[3];
tab[0].x = 1;
tab[0].y = 2;
```

- Passer une structure à une fonction

```
void afficherPoint(struct Point p){
    printf("(%d, %d)\n", p.x, p.y);
}
```

9️⃣ TP récapitulatif

Objectif : combiner variables, conditions, boucles, tableaux et structures.

- Déclarer un tableau de 3 élèves (`struct Eleve { char nom[20]; int age; float note; }`)
- Demander les informations via `scanf`
- Calculer la moyenne des notes
- Afficher la liste des élèves et la moyenne

```
#include <stdio.h>

struct Eleve {
    char nom[20];
    int age;
    float note;
};

int main(void){
    struct Eleve classe[3];
    float somme = 0;

    for(int i=0; i<3; i++){
        printf("Nom de l'élève %d : ", i+1);
        scanf("%s", classe[i].nom);
        printf("Age : ");
        scanf("%d", &classe[i].age);
        printf("Note : ");
        scanf("%f", &classe[i].note);
        somme += classe[i].note;
    }

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

    printf("Moyenne de la classe : %.2f\n", somme/3);
    return 0;
}
```