Informatique et Langage C

1re année GPSE

Thibaud Sadé

ENSICAEN

15 septembre 2026

Table des matières (1/2)

Présentation du langage C

Outils du développeur

Constituants du langage

Codage des lettres et standard

Fonctions et bibliothèques

Portée des variables

Structures de contrôle

Structures répétitives

Tableaux et chaînes de caractères

Entrées / sorties formatées

Pointeurs et mémoire

Allocation dynamique et mémoire

Table des matières (2/2)

Structures et types personnalisés

Enumérations et unions

Directives de préprocesseur et constantes

Code source sur plusieurs fichiers

Manipulation des fichiers

Interactions avec le système

Fonctions récursives

Cinq classes d'allocation

Listes chaînées

Usage de l'IA pour le langage C en 2026

Conclusion et fin

Présentation du langage C

Historique

C a été conçu afin de réécrire de façon portable et en langage évolué UNIX.

- *1972* : *Dennis Ritchie* (Laboratoires Bell, AT&T) crée le langage C.
- *1973* : UNIX est réécrit en C (D. Ritchie et K. Thompson). C s'impose comme le langage de programmation sous UNIX.
- *1978* : Kernighan et Ritchie publient une définition du langage C (le « K&R »).
- *1989* : fin du travail de normalisation du langage par l'ANSI (standard C89).
- *1999 / 2011 / 2023* : évolutions successives du standard ISO C (C99, C11, C23).

C++, développé par *B. Stroustrup* (AT&T), tend à prendre le relais de C tout en restant presque compatible avec C ANSI.

Caractéristiques

Atouts

- Langage à usage général
- Présent sur pratiquement toutes les machines et tous les systèmes
- Facilités d'interfaçage avec les primitives système
- Bibliothèque de fonctions et grand nombre d'opérateurs
- Programmation structurée et modulaire
- Code généré rapide et compact
- Portabilité du code si bien écrit

Faiblesses

- Langage relativement dépouillé
- Sémantiquement près du matériel
- Très permissif (faiblement typé)
- Un code très concis nuit à la clarté
- Conventions d'écriture et de nommage non imposées par le langage
- Le programmeur doit être rigoureux et s'imposer une convention d'écriture et de nommage.

Bibliographie

- **Le langage C**, 2e édition
  B. Kernighan & D. Ritchie — Masson, 1990
  Première référence du C par le père du langage, réédité pour le C ANSI.
- **Langage C — Manuel de référence**
  S.P. Harbison & G.L. Steele Jr. — Masson, 1990
  Référence technique très pointue, indispensable pour réaliser un compilateur C.
- **Langage C — Les finesses d'un langage redoutable**
  J. Charbonnel — Armand Colin, Collection U, 1992
  Guide pédagogique exposant les principaux pièges du langage.

Exemple de programme

Voici le code source d'un programme simple qui affiche le texte « Hello World ! » dans la console :

```c
#include <stdio.h>

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

Ce code source est enregistré dans un fichier texte avec l'extension `.c` pour le différencier d'un fichier `.txt` classique. Il peut être compilé avec la commande shell :

```
gcc hello.c      # produit l'executable a.out par defaut
```

La chaîne de compilation en langage C

Le passage du code source humain au fichier binaire exécutable par le processeur n'est pas instantané. Il s'exécute en quatre étapes distinctes.

Fichier d'entrée		Outil / Étape		Fichier généré
Code source (`.c`)	$\longrightarrow$	**1. Préprocesseur**	$\longrightarrow$	Code étendu (`.i`)
Code étendu (`.i`)	$\longrightarrow$	**2. Compilateur**	$\longrightarrow$	Code assembleur (`.s`)
Code assembleur (`.s`)	$\longrightarrow$	**3. Assembleur**	$\longrightarrow$	Code objet (`.o` ou `.obj`)
Code objet (`.o`) + *libc*	$\longrightarrow$	**4. Éditeur de liens**	$\longrightarrow$	**Exécutable** (`.exe` ou natif)

1. Préprocesseur : Traite les directives (`#include`, `#define`). Remplace les macros par du texte brut.

2. Compilateur : Traduit la logique C en langage assembleur textuel spécifique à l'architecture (ISA).

3. Assembleur : Traduit l'assembleur textuel en instructions binaires brutes (code objet non finalisé).

4. Éditeur de liens (*Linker*) : Assemble les fichiers objets et injecte le code des bibliothèques (*libc*).

Les standards du langage C : Le rôle historique de l'ANSI

Le C d'origine (K&R C) :
Dans les années 1970, le C évolue sans charte officielle, basé sur le livre de ses créateurs. Chaque constructeur le modifie à sa guise, menaçant la portabilité.

La normalisation ANSI C :
En 1983, l'institut américain (**ANSI**) forme un comité pour graver les règles officielles. Ce standard sort en **1989** sous le nom de **C89** ou **ANSI C**.

Les apports majeurs de l'ANSI C (C89) :

- **Bibliothèque standard** : Intégration officielle de la *libc* (`stdio.h`, `stdlib.h`...).
- **Prototypes de fonctions** : Obligation de déclarer la signature d'une fonction pour que le compilateur vérifie la cohérence du code.
- **Mots-clés rigoureux** : Introduction des qualificateurs fondamentaux `const` et `volatile`.

L'évolution des standards : De C89 à C23

Dès 1990, l'organisme international "ISO" adopte le standard de l'ANSI. Depuis, c'est l'ISO qui fait évoluer le langage C à travers des révisions régulières appelées les "standards ISO C".

Standard	Année	Évolution / Caractéristiques majeures
C89 / C90	1989	**ANSI C** → Standardisation universelle de base.
C99	1999	Commentaires `//`, variables déclarées n'importe où, type `bool`.
C11	2011	Gestion native du multi-threading, alignement mémoire, sécurité accru.
C17	2018	Version de maintenance (corrections de bugs, aucune nouveauté).
C23	2024	Intégration des mots-clés `true`/`false`, `nullptr`, type `constexpr`.

Compatibilité ascendante : Une force du langage C est que presque tout code écrit en pur ANSI C (C89) compile et fonctionne sans modification sur un compilateur moderne C23.

Comment forcer un standard avec le compilateur GCC ?

Le comportement par défaut :

Par défaut, chaque version de GCC utilise un standard de référence récent (souvent le C17 avec des extensions GNU).

L'option de rigueur `-std` :

Il est fortement conseillé de spécifier manuellement le standard requis pour garantir la portabilité stricte de votre code sur d'autres systèmes.

Exemples de compilation GCC avec restrictions de standards :

```
# Compiler en mode strict ANSI C (C89 / C90)
gcc main.c -o programme -std=c89 -pedantic

# Compiler avec le standard largement repandu C99
gcc main.c -o programme -std=c99

# Compiler avec le standard moderne C23
gcc main.c -o programme -std=c23
```

Pédagogie : Le flag `-pedantic` force GCC à rejeter impitoyablement toutes les extensions non conformes au standard choisi.

Programmation Bare-Metal vs OS Application

Le mode Bare-Metal (Brut) :

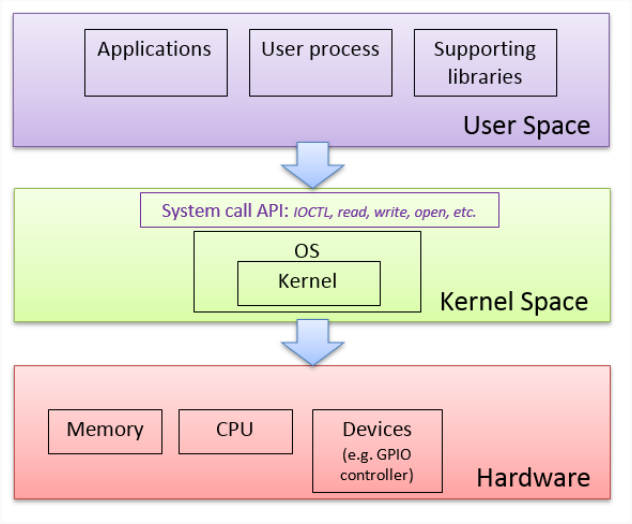
Le code s'exécute directement sur le microprocesseur, sans aucun système d'exploitation. Le programme a le contrôle absolu et exclusif de tout le matériel.

Le mode Applicatif (Sous OS) :

Le programme s'exécute sous le contrôle d'un système d'exploitation (Linux, Windows). L'OS sert d'arbitre et protège le matériel contre les abus.

	Programmation Bare-Metal	**Application sous OS (Linux)**
Accès matériel	Direct (écriture directe dans les registres)	Indirect via des *System Calls* (ex : `open`)
Gestion mémoire	Adresse physique brute (RAM complète)	Adressage virtuel (Pile/Tas gérés par l'OS)
Multitâche	Monotâche ou boucle infinie (*superloop*)	Géré par l'ordonnanceur (*scheduler*) de l'OS
Sécurité	Aucun pare-feu (erreur = crash matériel)	Erreur isolée (*Segmentation Fault*) par l'OS
Cibles types	Microcontrôleurs (Arduino, STM32, IoT)	Ordinateurs (PC), Serveurs, Raspberry Pi

Représentation en couches de l'OS



Outils du développeur

Les outils du développeur : Qu'est-ce qu'un IDE ?

Définition générale :

Un IDE (*Integrated Development Environment*) est un logiciel tout-en-un qui rassemble dans une seule interface graphique tous les outils nécessaires au développement.

L'intérêt pratique :

Il évite au développeur d'avoir à manipuler manuellement plusieurs outils complexes en ligne de commande (comme appeler `gcc` ou `gdb` séparément).

Les composants fondamentaux intégrés dans un IDE :

- **L'éditeur de code avancé** : Un traitement de texte spécialisé qui gère la coloration syntaxique, l'auto-complétion et repère les erreurs de syntaxe en temps réel.
- **Le compilateur / chaîne de build** : Intégré de manière invisible. Un simple bouton *"Run"* ou *"Build"* déclenche automatiquement la compilation du code C en binaire.
- **Le débogueur (*Debugger*)** : Permet d'exécuter le programme ligne par ligne, de mettre le code en pause (*breakpoints*) et d'espionner la valeur des variables en RAM.
- **Exemples courants** : *VS Code*, *CLion*, *Code ::Blocks*, *Geany*, *Eclipse* ou encore *Xcode*.

L'open source : Les grandes plateformes publiques

Le partage public :
Des serveurs web centralisés permettent aux développeurs de publier leur code source et de le rendre accessible à tous, gratuitement.

Un catalogue géant :
Ces espaces servent d'annuaires pour chercher du code, analyser des projets existants et télécharger des bibliothèques logicielles.

Les principales plateformes publiques de partage :

- **GitHub** : Le leader mondial incontournable (Microsoft), hébergeant la majorité des projets Open Source de la planète.
- **SourceForge** : L'un des pionniers historiques, toujours très utilisé comme répertoire public de logiciels libres.
- **Bitbucket** : Une plateforme majeure (Atlassian) accueillant de nombreux projets communautaires et professionnels.
- **Codeberg** : Une alternative publique européenne, indépendante et gérée par une association à but non lucratif.

Les outils du développeur : Le manuel système `man`

La documentation intégrée :

Sous Linux, il n'est pas nécessaire de chercher sur Internet pour connaître les arguments d'une fonction C. La commande `man` donne accès aux manuels officiels.

La section 3 (Bibliothèque) :

Le manuel est divisé en sections. La section 3 est spécifiquement dédiée aux fonctions de la bibliothèque standard du C (la *libc*).

Exemples de consultation dans le terminal :

```
# Consulter la documentation de printf (indique l'en-tete àinclure et les arguments)
man 3 printf

# Consulter le manuel de la fonction de saisie scanf
man 3 scanf

# Consulter le manuel de gestion de memoire malloc
man 3 malloc
```

Astuce orale : Si vous tapez simplement `man printf`, Linux affichera par défaut la section 1 (la commande de terminal `printf`). Préciser le chiffre **3** force le Shell à afficher la documentation du langage C.

Constituants du langage

Composants élémentaires d'un programme C

Un programme en langage C est constitué des groupes de composants élémentaires suivants :

- les commentaires
- les identificateurs
- les mots-clefs
- les constantes
- les chaînes de caractères
- les opérateurs
- les signes de ponctuation

Les commentaires

Les commentaires ne sont utiles que pour le développeur, afin d'améliorer la lisibilité et la compréhension du code source. Le compilateur ignore les commentaires.

Il existe deux façons d'écrire des commentaires :

```
1 /* utiliser slash-etoile pour ouvrir un commentaire
2   multilignes puis etoile-slash pour le fermer */
3
4 // utiliser slash slash pour les commentaires sur une seule ligne
```

Il n'est pas possible d'imbriquer des commentaires multilignes :

```
1 /* ceci /* ne fonctionnera pas */ lors de la compilation: */
```

Les identificateurs

Dans un programme C, les variables, fonctions, type définis, etc. sont désignés par un nom appelé *identificateur* :

- Composé de lettres, chiffres, tiret « - » ou underscore « _ ».
- Commence obligatoirement par une lettre ou un underscore _.
- Éviter les lettres accentuées : certains compilateurs les acceptent, d'autres non.
- Le C est *case sensitive* : il fait la différence entre minuscules et majuscules.

```c
int _compteur; // compteur est l'identificateur de cette variable
printf("hello"); // printf est l'dentificateur de cette fonction
```

Conventions d'écriture des identificateurs

Voici quelques conventions d'écriture courantes pour les identificateurs :

Nom de la convention
`camelCase`
`PascalCase`
`kebab-case`
`snake_case`
`SCREAMING_SNAKE_CASE`
`Train-Case`
`UPPERCASE`

Les mots clés

- Liste des 32 mots clés réservés du langage C (C89).
- Ils ne peuvent pas être utilisés comme identificateurs.

`auto`	`break`	`case`	`char`	`const`
`continue`	`default`	`do`	`double`	`else`
`enum`	`extern`	`float`	`for`	`goto`
`if`	`int`	`long`	`register`	`return`
`short`	`signed`	`sizeof`	`static`	`struct`
`switch`	`typedef`	`union`	`unsigned`	`void`
`volatile`	`while`			

Les types primitifs

Types	Contient	Octets
`char`	Caractère	1
`int`	Entier	4
`short`	Entier	2
`long int`	Entier	8
`float`	Réel	4
`double`	Réel	8
`void`	*	*

- Le C est un langage typé.
- Un type définit ce que représentent les bits stockés dans la variable (entier, réel, signé, non signé, caractère, etc.), une plage de valeurs et les opérations applicables.
- Le type *void* est un cas particulier, expliqué plus loin. Il ne doit jamais être utilisé pour déclarer une variable simple.

Les types primitifs : cas particuliers

Booléen (vrai ou faux)

- Il n'existe pas de type booléen natif en C89.
- Dans une structure de test `if(){} else{}`, un entier non nul est interprété comme « vrai », un entier nul comme « faux ».
- Une macro peut être utilisée par commodité : `#define true 1`

Chaîne de caractères

- Il n'existe pas de type « string » comme dans d'autres langages.
- On utilise à la place des tableaux de caractères (voir la section sur les pointeurs).

Entiers non signés

- Tout type entier peut devenir non signé : `char`, `short`, `int`, `long`.
- Cela double la valeur positive maximale stockable.
- Un `char` contient un nombre dans la plage $[-128; 127]$.
- Un `unsigned char` contient un nombre dans la plage $[0; 255]$.

Les types primitifs : taille et valeurs

Type	Taille	Valeur Min	Valeur Max
`char`	1 octet	-128	127
`unsigned char`	1 octet	0	255
`short`	2 octets	-32 768	32 767
`unsigned short`	2 octets	0	65 535
`int`	4 octets	-2 147 483 648	2 147 483 647
`unsigned int`	4 octets	0	4 294 967 295
`long long`	8 octets	$\approx -9{,}22 \times 10^{18}$	$\approx 9{,}22 \times 10^{18}$
`float`	4 octets	$\approx \pm 1{,}2 \times 10^{-38}$	$\approx \pm 3{,}4 \times 10^{38}$
`double`	8 octets	$\approx \pm 2{,}3 \times 10^{-308}$	$\approx \pm 1{,}7 \times 10^{308}$

La taille en mémoire des types dérivés `long`

Une taille variable :
Le standard du langage C garantit uniquement des tailles minimales (`long` $\geq$ `int`). La taille réelle dépend de l'architecture et du système d'exploitation.

Le piège du 64 bits :
Sous Linux/macOS (modèle *LP64*), un `long` passe à 8 octets. Sous Windows (modèle *LLP64*), il reste figé à 4 octets pour préserver la rétrocompatibilité.

Type en langage C	Taille en 32 bits	Taille en 64 bits (Linux/macOS)	Taille en 64 bits (Windows)
`long int`	4 octets (32 bits)	8 octets (64 bits)	4 octets (32 bits)
`long long int`	8 octets (64 bits)	8 octets (64 bits)	8 octets (64 bits)
`long double`	12 octets (96 bits)	16 octets (128 bits)	8 octets (64 bits)

Astuce pédagogique : Pour manipuler des entiers à taille fixe garantie sur toutes les plateformes, utilisez l'en-tête `<stdint.h>` (`int32_t`, `int64_t`).

Déclaration et initialisation des variables

```
type  nom_variable  =  valeur_initiale;
```

Concepts clés

1. **La déclaration**
   Réserve une zone dédiée dans la mémoire RAM.

2. **L'initialisation**
   Attribue une première valeur sécurisée dès la création.

Exemples de code en C

```c
// Declaration brute
int compteur;

// Declaration + Init
double score = 10.5;

// Declarations en serie
int x = 0, y = 1;
```

Sans initialisation, la valeur de la variable est **imprévisible** (déchet mémoire).

L'affectation : le sens de l'opérateur =

L'affectation se fait **toujours** de la **droite** vers la **gauche**.

Le mécanisme

1. L'ordinateur évalue d'abord l'expression à **droite** du signe =.
2. Il stocke ensuite le résultat dans la variable située à **gauche**.

Exemples en C

```c
int x; // Variable existante

x = 10;   // x <- 10
x = 5 + 3; // x <- 8

// Le cas classique :
x = x + 1; // x <- (8 + 1)
```

En C, le symbole = n'est pas une égalité mathématique, c'est une **action** (une copie).

Constantes littérales : les entiers

Suffixe / Préfixe	Signification
Aucun	Base 10 (Décimal)
Préf. `0`	Base 8 (Octal)
Préf. `0x` ou `0X`	Base 16 (Hexadécimal)
Préf. `0b` ou `0B`	Base 2 (Binaire, C23)
Suff. `U` ou `u`	Type `unsigned int`
Suff. `L` ou `l`	Type `long`
Suff. `LL` ou `ll`	Type `long long`

Code source exemple C

```c
int a = 42;
int b = 001; // = 1 base 10
int c = 0x2A; // = 42 base 10
int d = 0b101; // = 5 base 10

unsigned int e = 42U;
long f       = 42L;
long long g  = 42LL;
```

Constantes littérales : les réels

Suffixe / Notation	Signification
Aucun	Type `double` (par défaut)
Suff. `f` ou `F`	Type `float`
Suff. `L` ou `l`	Type `long double`
Not. exponentielle	$1{,}232 \times 10^{1} = 12{,}32$
Not. exponentielle	$1232 \times 10^{-2} = 12{,}32$

Code source exemple C

```c
double val1 = 12.32;
float val2 = 12.32f;
long double v3 = 12.32L;

double sci1 = 1.232e1;
double sci2 = 1232e-2;
```

Les suffixes des littéraux réels et le piège syntaxique

Le type par défaut :

En C, tout nombre à virgule écrit brut (ex : `3.0`) est automatiquement interprété comme un **`double`** (64 bits).

Forcer le type au compilateur :

Les suffixes indiquent la taille en mémoire :

- `f` / `F` : Force en `float` (32 bits).
- `l` / `L` : Force en `long double`.

Note : Le suffixe `lf` n'existe pas pour les littéraux, il ne sert que dans `scanf`.

Ecrire `3f` n'est pas autorisé en langage C :

Le chiffre 3 est syntaxiquement un **entier**. Le suffixe `f` étant réservé aux réels, le compilateur rejette cette association. Pour corriger, il faut transformer l'entier en réel avec un point décimal : `3.0f` ou `3.f`.

Syntaxe des littéraux en mémoire :

```c
float a = 3.0f; // Correct : stocké sur 32 bits (float)
double b = 3.0; // Correct : stocké sur 64 bits (double)
float c = 3.f; // Correct : le 0 après le point est implicite

// float d = 3f; // ERREUR : suffixe invalide sur un entier
// float e = 3.0lf; // ERREUR : le suffixe "lf" n'existe pas en C
```

Le stockage des réels : la norme IEEE 754

La norme universelle

Les nombres à virgule flottante (`float` et `double`) sont stockés en mémoire selon la norme internationale **IEEE 754**, sous forme de notation scientifique binaire.

Le découpage sur 32 bits

Pour un type `float` (4 octets = 32 bits), la configuration binaire est découpée en trois composants distincts.

Formule de décodage : $V = (-1)^{\mathbf{S}} \times 1,\mathbf{M} \times 2^{\mathbf{E}-127}$

- **Le Signe (S)** [1 bit] : `0` pour positif, `1` pour négatif.
- **L'Exposant (E)** [8 bits] : code la puissance de 2, stocké avec un décalage de $+127$ (exposant biaisé) pour coder des puissances négatives.
- **La Mantisse (M)** [23 bits] : code la partie fractionnaire. Le `1,` initial est implicite (non stocké) pour économiser un bit.

Codage IEEE 754 : exemple concret

Prenons un exemple simple à analyser : le nombre réel $-\mathbf{6,5}$.

- Étape 1 : passage en binaire pur $\rightarrow -6{,}5 = -(4+2+0{,}5) = -110{,}1_2$
- Étape 2 : normalisation scientifique $\rightarrow -1{,}101_2 \times 2^2$

Représentation finale dans les 32 bits de la RAM

- **Signe (S)** : le nombre est négatif $\rightarrow$ 1
- **Exposant (E)** : $2 + 127 = 129 \rightarrow$ 10000001
- **Mantisse (M)** : partie après le 1, de $1{,}101 \rightarrow$ 101 puis des 0 : 10100000000000000000000

Signe (1 bit)	Exposant (8 bits)	Mantisse (23 bits)
1	10000001	10100000000000000000000

Décodage IEEE 754 : exemple concret

Prenons la configuration de 32 bits suivante stockée en mémoire RAM :

Signe (1 bit)	Exposant (8 bits)	Mantisse (23 bits)
0	01111110	10000000000000000000000

Extraction et Reconstruction de la valeur

- **Étape 1 : Signe (S)** $\rightarrow$ le bit vaut 0, le nombre est donc **positif** $(+)$.
- **Étape 2 : Exposant (E)** $\rightarrow 01111110_2 = 126$.
  On retire le biais : Puissance $= 126 - 127 = -\mathbf{1} \rightarrow 2^{-1}$
- **Étape 3 : Mantisse (M)** $\rightarrow$ on extrait les bits utiles et le 1, implicite : $\mathbf{1{,}1_2}$

Calcul en binaire : $1{,}1_2 \times 2^{-1} = \mathbf{0{,}11_2}$ (décalage de la virgule à gauche)

Conversion finale en base 10 : $0{,}11_2 = (1 \times 2^{-1}) + (1 \times 2^{-2}) = 0{,}5 + 0{,}25 = +\mathbf{0{,}75}$

Complément à 2 : Algorithme d'encodage

Pour stocker un entier négatif en mémoire RAM, le processeur n'utilise pas un simple bit de signe. Il applique l'algorithme du complément à 2 afin de pouvoir réaliser des soustractions avec des circuits d'addition standard.

Exemple concret : Encodage du nombre -5 (sur 8 bits)

1. **Étape 1 : Écrire la valeur absolue en binaire pur**
   On prend la valeur positive $(+5) \rightarrow 00000101_2$
2. **Étape 2 : Inverser tous les bits (Complément à 1)**
   Chaque 0 devient 1 et chaque 1 devient 0 $\rightarrow 11111010_2$
3. **Étape 3 : Ajouter 1 au résultat binaire**
   On effectue l'addition : $11111010_2 + 1 = \mathbf{11111011_2}$

Bilan : Le nombre entier -5 est physiquement stocké sous la forme 11111011 en RAM. Le bit le plus à gauche (1) confirme que le nombre est négatif.

Complément à 2 : Algorithme de décodage

Quand le processeur lit un entier signé en mémoire, il regarde le premier bit à gauche (*MSB*). Si ce bit vaut 1, le nombre est négatif : il faut inverser l'algorithme pour retrouver sa valeur décimale.

Exemple concret : Décodage de la valeur `11111011`

1. **Étape 1 : Inverser tous les bits (Complément à 1)**
   On permute les 0 et les 1 $\rightarrow 00000100_2$
2. **Étape 2 : Ajouter 1 au résultat binaire**
   On effectue l'addition $\rightarrow 00000100_2 + 1 = \mathbf{00000101_2}$
3. **Étape 3 : Convertir en base 10 et appliquer le signe moins**
   Le bloc 00000101_2 vaut 5 en décimal. On rétablit le signe $\rightarrow \mathbf{-5}$

Méthode mathématique alternative rapide :
On applique un poids négatif au bit de signe :
$(-1 \times 2^7) + 2^6 + 2^5 + 2^4 + 2^3 + 2^1 + 2^0 = -128 + 64 + 32 + 16 + 8 + 2 + 1 = \mathbf{-5}$

Les opérateurs arithmétiques

Opérateur	Opération
+	Addition
-	Soustraction
*	Multiplication
/	Division
%	Modulo

Exemples en C

```c
int a = 10;
int b = 3;
int res;

res = a + b; // res vaut 13
res = a * b; // res vaut 30
res = a / b; // Division entiere : 3
res = a % b; // Reste : 1
```

Les opérateurs relationnels (comparaison)

Opérateur	Signification
==	Égal à
!=	Différent de
<	Strictement inférieur à
>	Strictement supérieur à
<=	Inférieur ou égal à
>=	Supérieur ou égal à

Exemples en C

```c
int a = 10, b = 5;
int res;

res = (a == b); // res vaut 0 (Faux)
res = (a != b); // res vaut 1 (Vrai)
res = (a >= 10); // res vaut 1 (Vrai
    )
```

Note de cours
En C standard (avant le C23), le type booléen pur n'existe pas nativement : **0** représente le faux, et toute valeur **différente de 0** (généralement 1) représente le vrai.

Raccourci syntaxique : L'opérateur conditionnel (? :)

Un opérateur ternaire :

C'est le seul opérateur du langage C qui prend **trois opérandes**. Il permet d'évaluer une condition logique directement au sein d'une expression.

Structure de l'expression :

La syntaxe suit une règle simple : `condition ? valeur_si_vrai : valeur_si_faux`

Il remplace avantageusement un bloc `if...else` court et lourd.

Utilisation de l'opérateur ternaire dans un jeu vidéo :

```c
#include <stdio.h>

int main(void) {
    int pv = 45; // Points de vie d'un joueur

    // Raccourci de : if (pv > 0) etat = "Vivant"; else etat = "Game Over";
    char *etat = (pv > 0) ? "Vivant" : "Game Over";

    printf("Etat du heros : %s\n", etat); // Affiche : Vivant

    // Exemple 2 : Appliquer un bonus de bouclier sans depasser le maximum
    int bouclier_actuel = 90, bonus = 20, max_bouclier = 100;
    int nouveau_bouclier = (bouclier_actuel + bonus > max_bouclier) ? max_bouclier : bouclier_actuel + bonus;

    return 0;
}
```

Les opérateurs logiques

Opérateur	Opération logique
`&&`	ET (AND)
`||`	OU (OR)
`!`	NON (NOT)

Évaluation court-circuit
Si le résultat final est certain dès le premier opérande, le compilateur n'évalue pas le second (ex : `faux && ...` s'arrête immédiatement).

Exemples en C

```c
int age = 20;
int permis = 1; // 1 = Vrai
int valide;

// Condition ET
valide = (age >= 18 && permis);
// valide vaut 1

// Condition NON
int mineur = !(age >= 18);
// mineur vaut 0
```

Incrémentation et décrémentation

Opérateur	Action
`x++`	Post-incrémentation
`++x`	Pré-incrémentation
`x--`	Post-décrémentation
`--x`	Pré-décrémentation

Différence clé

- **Pré** : augmente la valeur *avant* d'utiliser la variable.
- **Post** : utilise la valeur actuelle *avant* de l'augmenter.

Exemples en C

```c
int n = 5;
int res;

n++; // n vaut desormais 6

// Exemple de la difference :
int x = 5;
res = ++x; // x=6, res=6

int y = 5;
res = y++; // y=6, res=5
```

Les expressions en mémoire : `lvalue` vs `rvalue`

`lvalue` (Left value) :

Possède une **adresse fixe** en mémoire RAM (ex : une variable). On peut modifier sa valeur.

`rvalue` (Right value) :

Une valeur **temporaire** ou un littéral brut. N'a pas d'adresse accessible.

Pourquoi le code `--x++` provoque-t-il une erreur de compilation ?

- `x++` est prioritaire : il incrémente `x` mais retourne son ancienne valeur sous forme de **`rvalue`** éphémère.
- L'expression devient virtuellement : `--(10)`
- **Le crash :** L'opérateur `--` exige une **`lvalue`** (une case mémoire). Modifier une constante brute est impossible.

```c
int main(void) {
    int x = 10;

    // int res = --x++; // ERREUR : "lvalue required as decrement operand"
    return 0;
}
```

Les opérateurs d'affectation composée

Syntaxe abrégée	Forme équivalente
`a += b`	`a = a + b`
`a -= b`	`a = a - b`
`a *= b`	`a = a * b`
`a /= b`	`a = a / b`
`a %= b`	`a = a % b`

Utilité

Ces opérateurs simplifient l'écriture et évitent les répétitions lorsqu'on souhaite modifier la valeur d'une variable.

Exemples en C

```c
int a = 10;

a += 5; // Equivalent a a = 10 + 5
        // a vaut 15

a *= 2; // Equivalent a a = 15 * 2
        // a vaut 30

a %= 4; // Equivalent a a = 30 % 4
        // a vaut 2
```

Les opérateurs bit à bit : logique binaire

Définition

Ces opérateurs manipulent individuellement chaque bit (0 et 1) de la représentation binaire des nombres entiers.

Opérateur	Rôle logique
`&`	ET (*AND*)
`|`	OU (*OR*)
`^`	OU Exclusif (*XOR*)
`~`	NON (*NOT* / Complément)

Exemples d'opérations logiques en C

```c
unsigned char a = 5; // En binaire : 00000101
unsigned char b = 3; // En binaire : 00000011
unsigned char res;

res = a & b; // ET -> 00000001 (valeur : 1)
res = a | b; // OU -> 00000111 (valeur : 7)
res = a ^ b; // XOR -> 00000110 (valeur : 6)
res = ~a;   // NOT -> 11111010
```

Les opérateurs bit à bit : décalages

Décalage à gauche (<<)

Déplace les bits vers la gauche et remplit les vides par des 0. Cela équivaut à multiplier par 2^n.

Décalage à droite (>>)

Déplace les bits vers la droite. Pour un type non-signé, cela équivaut à diviser par 2^n (division entière).

Exemples de décalages en C

```c
unsigned char x = 4; // En binaire : 00000100
unsigned char res;

// Décalage àgauche (Multiplication)
res = x << 1; // Décale de 1 bit -> 00001000 (valeur : 8)
res = x << 2; // Décale de 2 bits -> 00010000 (valeur : 16)

// Décalage àdroite (Division)
res = x >> 1; // Décale de 1 bit -> 00000010 (valeur : 2)
```

La priorité des opérateurs : ordre d'évaluation

Priorité	Famille d'opérateurs	Exemples
1 (Max)	Accès / Parenthèses / Suffixes	`( ) [ ] . -> ++ --` (post)
2	Unaires / Préfixes / Logique	`++ --` (pré) `! ~ - + * &` **sizeof**
3	Conversions de type (*cast*)	`(type)`
4	Multiplicatifs	`* / %`
5	Additifs	`+ -`
6	Décalages de bits	`<< >>`
7	Comparaisons strictes	`< <= > >=`
8	Égalités et inégalités	`== !=`
9	ET bit à bit	`&`
10	XOR bit à bit	`^`
11	OU bit à bit	`|`
12	ET logique	`&&`
13	OU logique	`||`
14	Expression conditionnelle (ternaire)	`? :`
15	Affectations et raccourcis	`= += -= *= /= &= |= <<= >>=`
16 (Min)	Évaluation séquentielle	`,` (virgule)

Priorité des opérateurs : cas pratiques

Arithmétique et décalages

```c
int res;

// 1. Multiplication prioritaire
res = 5 + 3 * 2;
// Evaluation : 5 + 6
// res vaut 11

// 2. Parentheses absolues
res = (5 + 3) * 2;
// Evaluation : 8 * 2
// res vaut 16

// 3. Addition avant decalage
res = 1 + 1 << 2;
// Evaluation : (1 + 1) puis << 2
// res vaut 8
```

Logique et bit à bit

```c
int x = 5, y = 10, res;

// 4. Comparaison avant bit a bit
res = x > 3 & y < 20;
// Evaluation : (1) & (1)
// res vaut 1

// 5. Comparaison avant ET logique
res = x > 3 && y < 20;
// Evaluation : Vrai && Vrai
// res vaut 1

// 6. Piege classique : ET avant OU
res = 1 || 0 && 0;
// && est evalue en premier : (0&&0)->0
// Evaluation finale : 1 || 0
// res vaut 1
```

L'associativité

Qu'est-ce que c'est ?

L'associativité définit le sens de lecture (gauche-à-droite ou droite-à-gauche) lorsque plusieurs opérateurs ont exactement la **même priorité** sur une ligne.

Sens Gauche $\rightarrow$ Droite

Concerne la majorité des opérateurs (+, -, *, /).

Sens Droite $\rightarrow$ Gauche

Concerne principalement les affectations (=, +=) et les opérateurs unaires.

Exemples en C

```c
int res;

// Meme priorite (* et /) :
// Lecture de gauche a droite
res = 10 * 3 / 2;
// (10 * 3) = 30 -> 30 / 2 = 15

// Meme priorite (=) :
// Lecture de droite a gauche
int a, b;
a = b = 5;
// b = 5 en premier,
// puis a = b. Les deux valent 5.
```

Opérateurs unaires : ! vs ~

Logique vs binaire

- ! considère la variable comme un bloc unique (vrai/faux) et produit **toujours** 0 ou 1.
- ~ descend en mémoire et inverse chaque bit individuellement.

Opérateur	Rôle
!	NON logique (booléen)
~	NON bit à bit (complément)

Exemple comparatif en C

```c
unsigned char x = 5; // 00000101
int res;

// 1. Inversion logique
res = !x;
// 5 est Vrai -> res vaut 0

// 2. Inversion bit a bit
res = ~x;
// 00000101 -> 11111010
// res vaut 250
```

L'opérateur `sizeof`

Ce n'est **pas une fonction** mais un opérateur évalué à la compilation.

- Il mesure la taille en octets d'un type ou d'une variable.
- Les parenthèses sont obligatoires uniquement pour les types.

Exemples en C

```c
int entier;
int taille;

// Mesurer via un type
// (parentheses obligatoires)
taille = sizeof(int); // 4

// Mesurer via une variable
// (parentheses facultatives)
taille = sizeof entier; // 4
```

La conversion explicite de type : le cast

Définition

Le *cast* permet de forcer temporairement la conversion d'une valeur ou d'une variable vers un autre type pour une opération précise.

Syntaxe

Placer le type souhaité entre parenthèses devant l'élément à convertir.

```
(nouveau_type) valeur;
```

Exemples d'application

```c
int x = 10;
int y = 3;
float division;
int grand_entier = 300;
char petit_entier;

// 1. Eviter la division entiere
division = (float)x / y;
// x est traite comme 10.0f
// la division vaut alors 3.333

// 2. Forcer une troncature
petit_entier = (char)grand_entier;
// Force le stockage dans 1 octet
```

Le mécanisme du cast : dépassement de capacité

La troncature binaire

Lorsqu'un type plus grand (`int`, 4 octets) est converti dans un type plus petit (`char`, 1 octet), le compilateur ne garde que l'octet de poids faible.

Calcul mathématique

Le résultat équivaut à conserver le reste de la division par le nombre de combinaisons de l'octet (soit 256).

$$300 \pmod{256} = 44$$

Explication visuelle

```
1 int grand = 300;
2 // En binaire sur 32 bits :
3 // ... 00000001 00101100
4
5 char petit = (char)grand;
6 // On ne garde que 8 bits :
7 // 00101100
8 // petit vaut desormais 44
```

Le mécanisme de la division en C

Règle du compilateur

Le type du résultat dépend uniquement du type des deux opérandes.

- **int / int** = résultat **int** (tronqué).
- **float / int** = résultat **float** (réel).

Le piège classique

Le type de la variable de destination (à gauche du =) n'influence jamais le calcul à droite.

Exemples concrets

```c
int a = 10, b = 3;
float res;

// 1. Division entiere
res = a / b; // Calcul : 10 / 3 = 3
             // res stocke 3.0

// 2. Division decimale (Cast)
res = (float)a / b;
// Calcul : 10.0 / 3 = 3.333
// res stocke 3.333

// 3. Constante reelle
res = 10 / 3.0; // res stocke 3.333
```

Codage des lettres et standard

Le stockage des lettres : La Table ASCII et le type `char`

Le type `char` en mémoire :

Un processeur ne manipule que des bits. Pour stocker une lettre, on lui associe un index numérique. En C, le type `char` occupe strictement **1 octet** (8 bits) en mémoire RAM.

Le standard ASCII (1963) :

L'ASCII définit une table de correspondance officielle codée sur **7 bits** (128 caractères valides), regroupant les lettres anglaises, les chiffres et la ponctuation de base.

Dualité du type char (Lettre vs Entier)

```c
char lettre = 'A'; // Stocke la configuration binaire de 'A'

// 1. Affichage sous forme de caractère
printf("Caractere : %c\n", lettre); // Affiche : A

// 2. Affichage sous forme d'entier (Table ASCII)
printf("Valeur ASCII : %d\n", lettre); // Affiche : 65
```

La table ASCII standard (Caractères imprimables)

Dec	Char	Dec	Char	Dec	Char	Dec	Char
32	*Espace*	56	8	80	P	104	h
33	!	57	9	81	Q	105	i
34	”	58	:	82	R	106	j
35	#	59	;	83	S	107	k
36	$	60	<	84	T	108	l
37	%	61	=	85	U	109	m
38	&	62	>	86	V	110	n
39	'	63	?	87	W	111	o
40	(	64	@	88	X	112	p
41	)	65	**A**	89	Y	113	q
42	*	66	B	90	Z	114	r
43	+	67	C	91	[	115	s
44	,	68	D	92	\	116	t
45	-	69	E	93	]	117	u
46	.	70	F	94	^	118	v
47	/	71	G	95	_	119	w
48	0	72	H	96	‘	120	x
49	1	73	I	97	**a**	121	y
50	2	74	J	98	b	122	z
51	3	75	K	99	c	123	{
52	4	76	L	100	d	124	|
53	5	77	M	101	e	125	}
54	6	78	N	102	f	126	~
55	7	79	O	103	g	127	*Suppr*

L'évolution : Les tables étendues et le problème régional

L'utilisation du 8e bit :

L'ASCII d'origine (7 bits) ne possède aucun caractère accentué (*é, à, ç*) ni alphabet non latin. Les constructeurs ont utilisé le 8e bit disponible d'un octet pour étendre la table à 256 caractères.

La fragmentation (ISO-8859) :

Comme 256 cases ne suffisent pas pour toutes les langues du monde, des dizaines de tables régionales incompatibles sont nées (les encodages ISO-8859).

Standard d'encodage	Cible linguistique	Exemple d'interprétation du code `0xE9`
ISO-8859-1 (Latin-1)	Europe de l'Ouest	Interprété comme la lettre : **é**
ISO-8859-5	Alphabet Cyrillique	Interprété comme la lettre :
ISO-8859-7	Alphabet Grec	Interprété comme la lettre : **é** (Omega)

Le piège de l'affichage : Si un fichier textuel est enregistré en Latin-1 et lu avec une table Cyrillique, le texte devient totalement corrompu et illisible (*mojibake*).

La solution moderne : Le standard Unicode et l'UTF-8

Le standard Unicode :

Une immense table universelle qui associe un index unique (*Code Point*) à chaque caractère de chaque langue humaine, incluant les émojis (plus de 150 000 entrées).

L'encodage intelligent UTF-8 :

Encodage à taille variable (de 1 à 4 octets par lettre) conçu pour optimiser l'espace tout en restant compatible avec l'écosystème existant.

Le fonctionnement dynamique de l'UTF-8 :

- **Compatibilité ASCII totale** [1 octet] : Les caractères anglais standard conservent leur code ASCII natif. Un fichier pur ASCII est un fichier UTF-8 valide.
- **Caractères européens** [2 octets] : Les lettres accentuées (*é, à*) ou alphabets grecs basculent automatiquement sur deux octets.
- **Alphabets asiatiques / Émojis** [3 à 4 octets] : Les idéogrammes ou symboles complexes consomment plus d'espace à la demande.

Bilan en C : Une chaîne contenant des accents en UTF-8 occupera **plus d'octets en RAM** que son nombre de lettres (`strlen()` mesurant les octets, le résultat sera supérieur au nombre visuel de caractères).

Exemple d'encodage UTF-8 : Le caractère 'é'

L'index Unicode :

Le caractère **'é'** possède l'index universel `U+00E9`. Sa valeur binaire est `11101001`. Elle dépasse les 7 bits de l'ASCII standard.

Le codage sur 2 octets :

L'UTF-8 détecte que le code nécessite 2 octets. Il utilise des bits de contrôle structurels (110 et 10) pour baliser le flux binaire.

Décomposition binaire en RAM :

Structure UTF-8	**Premier octet**	**Second octet**
Bits de contrôle	`110x xxxx`	`10xx xxxx`
Bits de la lettre 'é'	`11000011`	`10101001`
Valeur brute en RAM	`0xC3` (195 en décimal)	`A9` (169 en décimal)

Bilan en C : En UTF-8, le mot `"été"` consomme physiquement **5 octets** en mémoire RAM (`0xC3 0xA9 0x74 0xC3 0xA9`), plus le caractère de fin `'\0'`.

Fonctions et bibliothèques

Introduction aux fonctions en langage C

Définition d'une fonction

Bloc de code nommé qui regroupe un ensemble d'instructions pour accomplir une tâche précise, éviter les répétitions et structurer le programme.

Fonctionnement général

- Des données d'entrée (paramètres)
- Un traitement interne (instructions)
- Une donnée de sortie (valeur de retour)

Déclaration et appel

```c
// Declaration (Prototype)
int calculer(int, int);

int main(void) {
    int a=3, b=4, result=0;
    result = calculer(a, b); // Appel
    return 0;
}
```

Définition (corps)

```c
// Definition de la fonction
int calculer(int a, int b) {
    return a+b;
}
```

Appel à des bibliothèques : `#include`

Le principe de l'inclusion

Le langage C est modulaire. Pour utiliser des fonctions prêtes à l'emploi (mathématiques, affichage, etc.), il faut inclure leur fichier d'en-tête (`.h`).

La directive `#include <...>` ordonne au préprocesseur de copier-coller les déclarations de la bibliothèque dans votre fichier avant la compilation.

Exemple complet avec `math.h`

```c
#include <stdio.h>
#include <math.h> // Pour sqrt()

int main(void) {
    double nombre = 16.0;
    double racine;

    // Appel de la fonction sqrt
    racine = sqrt(nombre);

    printf("Racine= %.1f\n", racine);

    return 0;
}
```

Comment le compilateur localise-t-il les `.h` ?

1. Recherche automatique

Le compilateur possède une liste interne de dossiers système (comme `/usr/include`) fouillée automatiquement pour les chevrons `<...>`.

2. Variables d'environnement

Le compilateur consulte des variables système pour ajouter des dossiers (ex : `CPATH` sous Linux, `INCLUDE` sous Windows).

3. Indication manuelle

Si un en-tête se trouve dans un dossier personnalisé, on indique ce chemin au compilateur avec l'option `-I`.

Commande Shell (Terminal)

```
gcc main.c -I /chemin/mon_dossier
```

Localisation des fichiers d'en-tête (`.h`)

Système d'exploitation	Chemins d'installation possibles
Linux	`/usr/include/` `/usr/local/include/`
macOS	`/Library/Developer/CommandLineTools/SDKs/.../usr/include/` `/Applications/Xcode.app/Contents/Developer/Platforms/...`
UNIX (FreeBSD, etc.)	`/usr/include/` `/usr/local/include/`
Windows (MSVC)	`C:\Program Files (x86)\Windows Kits\10\Include\`

Inclusion : <...> vs "..."

Syntaxe officielle : `#include <exemple.h>`

Cherche le fichier uniquement dans les **dossiers système par défaut** de la bibliothèque standard.

Usage : bibliothèques standards fournies avec la libc ou l'OS.

Syntaxe locale : `#include "exemple.h"`

Cherche d'abord dans le **dossier courant** (où se trouve votre fichier source `.c`). S'il n'y est pas, bascule vers la recherche dans les dossiers système.

Usage : vos propres fichiers d'en-tête ou modules locaux.

Le langage C et l'OS : la bibliothèque standard

Le langage C

Le langage C est uniquement une **syntaxe**. Il définit :

- Les types primitifs (`int`, `char`...)
- Les structures de contrôle (`if`, `while`...)
- Les opérateurs (+, =, `sizeof`...)

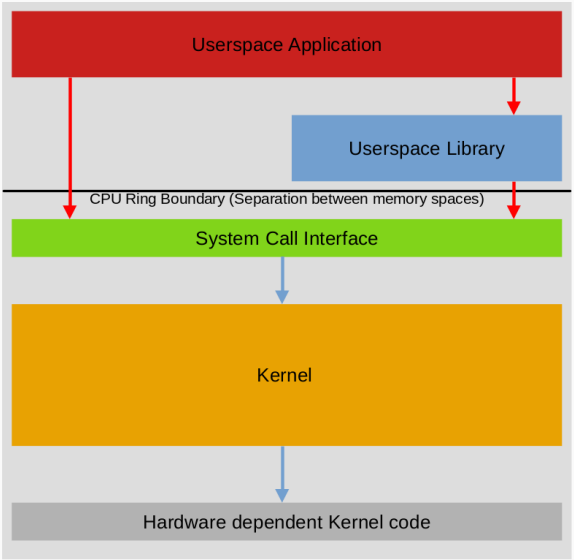
Le langage seul est **incapable** d'interagir avec les entrées/sorties (écran, clavier, fichiers). Il passe par la bibliothèque standard.

La bibliothèque standard (libc)

La *libc* est la passerelle obligatoire pour interagir avec le système d'exploitation (**OS**). Elle fournit les fonctions de communication :

- **Flux standards** : `printf()`, `scanf()`, `putchar()`
- **Fichiers** : `fopen()`, `fclose()`, `fread()`
- **Mémoire** : `malloc()`, `calloc()`, `free()`
- **Processus / réseau** : `system()`, `getenv()`

Appel système par l'application



La libc : un standard, plusieurs implémentations

Le principe de la norme

La norme ISO C définit précisément les prototypes et le comportement attendu des fonctions de la bibliothèque standard. Chaque OS fournit une **implémentation** spécifique adaptée à son architecture.

Système	Implémentations de la libc
Linux	`glibc` (GNU), `musl`, `uClibc`
Windows	`MSVCRT` / `UCRT` (Microsoft)
macOS	`BSD libc` (dans `libSystem`)
UNIX	`FreeBSD libc`, `Solaris libc`

Extensions des implémentations et portabilité

Au-delà de la norme

Les implémentations ne se limitent pas à la norme ISO C : elles ajoutent des fonctions spécifiques non standard pour simplifier le développement ou optimiser les performances.

L'exemple de la glibc

La `glibc` (GNU) est particulièrement riche : `asprintf()`, `strchrnul()`, fonctions de gestion des threads GNU, etc.

Le piège de la portabilité

L'utilisation de ces extensions lie le programme à l'écosystème d'origine. Un code s'appuyant sur la `glibc` refusera de compiler sur un système utilisant une autre *libc* (`musl`, libc de Windows...).

Portée des variables

Gestion des variables : La portée (*Scope*)

Variables Locales :

Déclarées à l'intérieur d'un bloc de code { }. Elles ne sont visibles et accessibles qu'au sein de ce bloc spécifique et sont détruites à sa sortie.

Variables Globales :

Déclarées en dehors de toute fonction. Elles sont visibles et modifiables par tout le code du fichier. Leur usage massif est fortement déconseillé (risques d'effets de bord).

Illustration de la visibilité des variables :

```c
#include <stdio.h>

int globale = 100; // Accessible partout

void fonction(void) {
    int locale = 5; // Uniquement accessible ici
    printf("%d %d\n", locale, globale);
}

int main(void) {
    fonction();
    // printf("%d", locale); // ERREUR : 'locale' n'existe pas ici
    return 0;
}
```

Structures de contrôle

Structures conditionnelles : `if` / `else if` / `else`

Fonctionnement

Exécute un bloc d'instructions uniquement si une condition (expression logique) est évaluée à **vrai** (différente de 0).

Règle des accolades

Les accolades `{ }` sont obligatoires si le bloc contient plusieurs instructions ; facultatives s'il n'y en a qu'une.

Exemple de structure alternative

```c
int note = 14;

if (note >= 16) {
    printf("Tres bien\n");
} else if (note >= 12) {
    printf("Assez bien\n"); // Ce bloc sera execute
} else {
    printf("A ameliorer\n");
}
```

Structures conditionnelles : `switch` / `case` / `default`

Le choix multiple

Le `switch` teste la valeur d'une variable entière ou caractère par rapport à une liste de constantes.

L'instruction `break`

Indispensable à la fin de chaque `case`. Sans elle, le programme continue dans les cas suivants sans tester la condition.

Exemple d'aiguillage avec switch

```c
char choix = 'B';

switch (choix) {
    case 'A':
        printf("Option A choisie\n");
        break;
    case 'B':
        printf("Option B choisie\n"); // Ce bloc sera execute
        break;
    default:
        printf("Option inconnue\n");
}
```

Structures répétitives

Structures répétitives : la boucle `while`

Principe

Répète un bloc **tant que** la condition reste vraie. La condition est évaluée **avant** chaque tour.

Nombre de répétitions

Inconnu à l'avance. Si la condition est fausse dès le départ, le bloc n'est **jamais** exécuté (0 fois).

Exemple de boucle tant que

```c
int compteur = 0;

while (compteur < 3) {
    printf("Compteur = %d\n", compteur);
    compteur++; // Variable de controle modifiee a l'interieur
}
// Affiche successivement : 0, 1, puis 2
```

Structures répétitives : la boucle `do ... while`

Principe

Le bloc est exécuté, puis la condition est évaluée à la fin du tour. La boucle se répète tant que la condition reste vraie.

Exécution minimale

Contrairement à `while`, le bloc est exécuté **au moins une fois**, même si la condition est fausse dès le début.

Exemple de boucle do-while

```c
int valeur = 10;

do {
    printf("Ce message s'affiche au moins 1 fois.\n");
} while (valeur < 5); // Faux des le depart
```

Structures répétitives : la boucle `for`

Principe

Idéale lorsque le nombre de répétitions est **connu à l'avance**. Toute la gestion de la boucle tient en une seule ligne.

Les 3 étapes de la syntaxe

`for (initialisation; condition; incrementation)`

Exemple de boucle pour

```c
// i est initialise, teste, puis incremente a chaque fin de tour
for (int i = 0; i < 3; i++) {
    printf("Tour numero %d\n", i);
}
// Affiche successivement les tours 0, 1 et 2
```

Rupture de boucle : `break` et `continue`

L'instruction `break`

Interrompt immédiatement la boucle en cours et force la sortie du bloc répétitif.

L'instruction `continue`

Interrompt le tour actuel et force le passage immédiat au tour suivant (au test de condition).

Exemples de rupture en C

```c
for (int i = 1; i <= 5; i++) {
    if (i == 2) {
        continue; // Saute le reste du tour pour i = 2
    }
    if (i == 4) {
        break;   // Arrete definitivement la boucle a i = 4
    }
    printf("i = %d\n", i);
}
// Affichage obtenu : "i = 1" puis "i = 3"
```

Tableaux et chaînes de caractères

Tableaux à une dimension : déclaration

Définition

Un tableau regroupe un nombre fixe d'éléments de **même type**, stockés de manière contiguë en mémoire RAM.

Syntaxe générale

`type nom_tableau[taille];`

La taille doit obligatoirement être une valeur entière connue à la compilation.

Exemples de déclarations et initialisations

```c
// 1. Declaration brute (valeurs initiales imprevisibles)
int poids[5];

// 2. Declaration avec initialisation complete
int poids_optimums[5] = {12, 15, 8, 19, 14};

// 3. Initialisation a zero de toutes les cases
int compteurs[100] = {0};
```

Tableaux à une dimension : accès et parcours

Le système d'indices

En C, les cases d'un tableau sont numérotées à partir de **0**. Pour un tableau de taille N, les indices valides vont de **0 à** $N-1$.

Attention au débordement

Le compilateur ne vérifie pas si l'indice dépasse la taille. Accéder à une case interdite provoque un comportement indéterminé (*Segmentation Fault*).

Accès individuel et parcours complet

```c
int tab[3] = {10, 20, 30};

// Modification d'une case precise
tab[1] = 25; // Le tableau devient {10, 25, 30}

// Parcours automatique a l'aide d'une boucle for
for (int i = 0; i < 3; i++) {
    printf("Case %d = %d\n", i, tab[i]);
}
// Indices parcourus : 0, 1 et 2
```

Les chaînes de caractères : définition et stockage

Définition

En C, il n'existe pas de type « string » natif. Une chaîne est un simple tableau de `char` consécutifs.

Le caractère nul `'\0'`

Toute chaîne doit se terminer par le caractère `'\0'` (valeur ASCII 0). Il sert de balise d'arrêt indiquant la fin du texte en mémoire.

Déclarations de chaînes de caractères

```c
// 1. Initialisation automatique (le \0 est ajoute en cache)
char message[] = "Hello";
// Taille reelle en memoire = 6 octets ('H','e','l','l','o','\0')

// 2. Initialisation manuelle case par case
char mot[] = {'C', '\0'};

// 3. Reservation d'espace sans initialisation complete
char texte[50];
```

Les chaînes de caractères : manipulation

Affichage

On utilise le spécificateur de format %s avec `printf()` pour afficher le texte jusqu'au `'\0'`.

La bibliothèque `string.h`

Indispensable pour mesurer, copier ou comparer des chaînes de manière sécurisée.

Utilisation des fonctions standard

```c
#include <stdio.h>
#include <string.h> // Indispensable pour strlen

int main(void) {
    char source[] = "C23"; int longueur;

    // Calcul de la taille utile (exclut le \0)
    longueur = strlen(source); // longueur vaut 3

    printf("Chaine : %s (%d lettres)\n", source, longueur);
    return 0;
}
```

Entrées / sorties formatées

Affichage formaté : le rôle de `printf()`

La chaîne de contrôle

`printf()` (de `<stdio.h>`) prend en premier argument du texte contenant des balises appelées **spécificateurs de format**.

La substitution

À l'exécution, chaque spécificateur (commençant par %) est remplacé par la valeur de la variable correspondante passée en argument.

Exemple de substitution de format

```c
#include <stdio.h>

int main(void) {
    int age = 25;
    double taille = 1.82;

    // Les balises %d et %.2f sont remplacees dans l'ordre
    printf("Age : %d ans, Taille : %.2f m\n", age, taille);

    return 0; }
```

Forcer l'interprétation des bits

La réalité de la RAM

En mémoire, il n'y a ni lettres ni nombres à virgule : il n'y a que des **suites de bits (0 et 1)**. Le type n'est qu'une convention logicielle.

Le pouvoir du spécificateur

Le spécificateur de format impose au processeur une règle de lecture. Il **force l'interprétation** de la configuration binaire brute, quel que soit le type d'origine.

Même suite de bits, deux lectures différentes

```c
char lettre = 'A'; // En RAM, 'A' est stocke comme 01000001

// 1. Lecture standard sous forme de texte
printf("Format %%c : %c\n", lettre); // Affiche : A

// 2. Lecture forcee sous forme d'entier numerique (Table ASCII)
printf("Format %%d : %d\n", lettre); // Affiche : 65
```

Liste des spécificateurs de format usuels

Spécificateur	Type attendu	Interprétation / Affichage
`%d` ou `%i`	`int`	Entier relatif (base 10, signé)
`%u`	`unsigned int`	Entier positif pur (non signé)
`%x` ou `%X`	`int` / `char`	Nombre en base 16 (Hexadécimal)
`%o`	`int`	Nombre en base 8 (Octal)
`%f`	`float` / `double`	Réel à virgule fixe (ex : `12.34`)
`%e` ou `%E`	`float` / `double`	Notation scientifique
`%c`	`char`	Un seul caractère
`%s`	`char[]` (chaîne)	Chaîne (jusqu'au `'\0'`)
`%p`	Pointeur (`void *`)	Adresse mémoire brute (Hexadécimal)
`%%`	*Aucun*	Affiche le symbole %

Saisie de données : la fonction `scanf()`

Le rôle de `scanf()`

`scanf()` (de `<stdio.h>`) met le programme en pause pour attendre une saisie clavier de l'utilisateur.

L'obligation de l'adresse (`&`)

Pour que `scanf()` puisse modifier la variable d'origine, il faut lui transmettre son **adresse mémoire** via l'opérateur `&`.

Exemple de saisie utilisateur

```c
#include <stdio.h>

int main(void) {
    int age = 0;
    printf("Veuillez entrer votre age : ");

    scanf("%d", &age); // %d indique le format, &age fournit l'adresse

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

Saisie de texte : `scanf` vs `fgets`

Le problème de `scanf()`

`scanf("%s", ...)` s'arrête au premier **espace ou retour à la ligne**. Impossible de saisir une phrase complète (ex : « Jean Dupont »).

La solution `fgets()`

Lit une ligne entière, **espaces inclus**, jusqu'au retour à la ligne. Elle impose une taille maximale pour éviter les débordements de mémoire.

Saisie d'une chaîne avec espaces

```c
#include <stdio.h>

int main(void) {
    char nom[50];

    printf("Entrez votre nom complet : ");

    // Lit max 50 caracteres depuis l'entree standard (stdin)
    fgets(nom, sizeof(nom), stdin);

    // Remarque : contrairement a scanf, l'esperluette &
    // n'est pas requise car 'nom' est deja un pointeur.

```

Pointeurs et mémoire

Les pointeurs : adresses mémoire et déclaration

Un pointeur est une variable particulière dont le rôle est de stocker **l'adresse** d'un espace mémoire. Cela a de nombreuses applications en langage C. Le pointeur est d'ailleurs l'élément le plus fondamental dans la compréhension de ce langage. Il est indispensable de le maîtriser parfaitement.

Déclaration et affectation d'un pointeur

```c
int nombre = 42;
int *ptr; // Le symbole * indique que ptr est un pointeur sur int

// Stockage de l'adresse de 'nombre' dans 'ptr'
ptr = &nombre;

// Remarque : ptr contient une adresse (ex: 0x7ffeefbff56c)
printf("%p", nombre); // 42
printf("%p", &nombre); // 0x7ffeefbff56c
printf("%p", ptr);   // 0x7ffeefbff56c
printf("%p", *ptr);  // 42
```

Les pointeurs : l'opérateur de déréférencement

Le déréférencement (*)

Placé devant un pointeur existant, l'opérateur * permet d'accéder directement à la **valeur** située à l'adresse pointée.

Double rôle du symbole *

- À la déclaration : indique le type pointeur.
- Dans le code : permet de lire ou modifier la valeur pointée.

Manipulation de valeur via un pointeur

```c
int nombre = 42;
int *ptr = &nombre;

// 1. Lecture indirecte
printf("Valeur = %d\n", *ptr); // Affiche 42

// 2. Modification indirecte
*ptr = 100;
// La variable 'nombre' vaut desormais 100 !
```

Passage de paramètres : le problème de la copie

Le passage par valeur

Par défaut, lorsqu'on passe une variable à une fonction, le compilateur crée une **copie locale** de sa valeur en RAM.

L'isolement des fonctions

Toute modification du paramètre à l'intérieur de la fonction n'agit que sur la copie. La variable d'origine reste inchangée.

Exemple : Échec de la modification

```c
void tenterModifier(int x) {
    x = 99; // Modifie uniquement la copie locale
}

int main(void) {
    int nombre = 10;
    tenterModifier(nombre);
    printf("Nombre = %d\n", nombre); // nombre vaut toujours 10 !
    return 0;
}
```

Passage de paramètres : la solution par adresse

Le principe

Au lieu de copier la valeur, on transmet **l'adresse mémoire** de la variable d'origine. Le paramètre devient un **pointeur**.

La modification directe

En déréférençant le pointeur, on accède et on modifie directement la case mémoire de la variable d'origine.

Exemple : Modification réussie via un pointeur

```c
void modifierPourDeBon(int *ptr) {
    *ptr = 99; // Accède directement a la variable d'origine
}

int main(void) {
    int nombre = 10;
    modifierPourDeBon(&nombre); // On transmet l'adresse
    printf("Nombre = %d\n", nombre); // nombre vaut desormais 99 !
    return 0;
}
```

Application : la permutation de deux valeurs

L'objectif

Échanger le contenu de deux variables. Pour que l'échange persiste après l'appel, il est obligatoire de passer par leurs **adresses mémoire**.

Le besoin d'une variable tierce

Comme pour vider deux verres l'un dans l'autre, une variable **temporaire** est indispensable pour sauvegarder la première valeur avant de l'écraser.

Implémentation de la fonction permuter

```c
void permuter(int *a, int *b) {
    int temp; // Variable de sauvegarde locale
    temp = *a; // Etape 1 : Sauvegarde la valeur pointee par a
    *a = *b;  // Etape 2 : Copie la valeur de b dans la case de a
    *b = temp; // Etape 3 : Restitue la sauvegarde dans la case de b
}
int main(void) {
    int x = 5, y = 10;
    permuter(&x, &y); // x vaut 10 et y vaut 5
    return 0; }
```

Pointeurs et tableaux : l'équivalence fondamentale

La nature cachée du tableau

Le nom d'un tableau seul est un raccourci représentant **l'adresse mémoire de sa toute première case**.

L'équivalence mathématique

Écrire `tab` est strictement équivalent à écrire `&tab[0]`. C'est un pointeur constant.

Preuve de l'équivalence des adresses

```c
int tab[] = {10, 20, 30};
int *ptr;

// Les deux affectations suivantes sont identiques :
ptr = tab;    // On affecte l'adresse de base
ptr = &tab[0]; // Meme adresse explicite

// Déréférencer le nom du tableau donne la premiere case :
printf("Premiere case = %d\n", *tab); // Affiche 10
```

Pointeurs et tableaux : l'arithmétique des pointeurs

L'arithmétique des adresses

Ajouter 1 à un pointeur ne l'augmente pas d'un octet, mais de la **taille du type pointé** pour sauter à la case suivante.

La dualité syntaxique

L'écriture `tab[i]` est interprétée par le compilateur comme `*(tab + i)`.

Parcours de tableau avec l'arithmétique

```c
int tab[] = {10, 20, 30};

// Accès a la deuxieme case (index 1) :
int val1 = tab[1];    // Syntaxe classique : 20
int val2 = *(tab + 1); // Syntaxe pointeur : 20

// Deplacement d'un pointeur independant :
int *ptr = tab;       // Pointe sur 10
ptr++;                // Avance de 4 octets, pointe sur 20
```

Fonctions : Tableaux multidimensionnels en paramètre

La règle des dimensions :
En C, pour passer un tableau à plusieurs dimensions en paramètre, il est obligatoire de spécifier la taille de toutes les dimensions, sauf la première.

La réalité de la mémoire RAM :
Même à 3 dimensions, le tableau reste une ligne continue en RAM. Le CPU doit connaître la taille des sous-blocs pour calculer où sauter.

Comment le processeur calcule-t-il l'adresse ?

Pour accéder à la case `tab[i][j][k]` d'un tableau défini en `[N][2][8]`, le compilateur traduit l'écriture par un calcul d'adresse brute :

$$\text{Adresse} = \text{Base} + (i \times (\mathbf{2} \times \mathbf{8}) + j \times \mathbf{8} + k) \times \text{sizeof(int)}$$

On constate que la taille de la première dimension (N) n'intervient jamais dans le calcul. En revanche, les tailles des dimensions suivantes ($\mathbf{2}$ et $\mathbf{8}$) sont indispensables.

Fonctions : Exemple de tableau multidimensionnel

Syntaxe de déclaration et d'appel en C :

```c
#include <stdio.h>

// La premiere dimension [] peut rester vide
void manipulerTableau(int tab[][2][8], int taille_dim1) {
    // Calcul d'adresse possible grace aux valeurs 2 et 8
    tab[0][1][4] = 42;
}

int main(void) {
    // Declaration d'un tableau (4 blocs de 2x8 entiers)
    int mon_tableau[4][2][8] = {0};

    // On passe uniquement le nom du tableau (son adresse)
    manipulerTableau(mon_tableau, 4);

    printf("Valeur modifiee : %d\n", mon_tableau[0][1][4]);
    return 0;
}
```

Allocation dynamique et mémoire

Allocation statique vs dynamique

L'allocation statique

La taille et l'espace sont réservés et figés **avant** l'exécution, lors de la compilation.

La mémoire est libérée automatiquement dès la sortie de la fonction.

L'allocation dynamique

La taille et la réservation s'effectuent à la demande **pendant** l'exécution, selon les besoins réels (via `malloc`).

La mémoire reste allouée tant que le développeur ne la libère pas manuellement.

Comparaison dans le code C

```c
int tab_statique[10]; // Allocation statique (taille figee a 10)

int taille = 50; // Allocation dynamique (taille ajustable)

int *tab_dynamique = malloc(taille * sizeof(int));

free(tab_dynamique); // Liberation manuelle obligatoire
```

L'allocation dynamique : la fonction `malloc`

La mémoire Tas (*Heap*)

Contrairement aux variables classiques (allouées sur la Pile), l'allocation dynamique permet de réserver de la mémoire pendant l'exécution, à une taille ajustée dynamiquement.

Le rôle de `malloc()`

`malloc()` (de `<stdlib.h>`) réserve un bloc d'octets consécutifs dans le Tas et renvoie un pointeur générique (`void *`) vers son adresse de base.

Réservation dynamique d'un tableau

```c
#include <stdlib.h>
int main(void) {
  int taille = 5, *tableau;
  tableau = malloc(taille * sizeof(int)); // Allocation de 5 entiers

  if (tableau == NULL) { // Verification du pointeur
     return 1; // Echec de l'allocation (memoire RAM saturee)
  }
  return 0; }
```

L'allocation dynamique : la fonction `free`

La libération manuelle

La mémoire allouée dans le Tas n'est jamais détruite automatiquement à la fin d'une fonction. Il faut la libérer explicitement avec `free()`.

La fuite de mémoire

Oublier `free()` engendre une **fuite de mémoire** (*memory leak*). Le programme accumule de la RAM inutilement jusqu'à saturation.

Libération et bonne pratique

```
1 // ... allocation et utilisation du tableau ...
2 tableau[0] = 42; // S'utilise comme un tableau classique
3
4 // Restitution de la memoire au systeme
5 free(tableau);
6
7 // Bonne pratique : eviter le pointeur fou (dangling pointer)
8 tableau = NULL;
```

Les zones de la mémoire RAM : Pile et Tas

La Pile (*Stack*)

Zone ordonnée, rapide et automatique. Stocke les variables locales et les arguments des fonctions.

Sa taille est limitée : un trop grand tableau statique provoque un plantage par saturation (*Stack Overflow*).

Le Tas (*Heap*)

Zone massive, flexible mais plus lente. Dédiée à l'allocation dynamique (`malloc`).

Espace « en vrac » géré par l'OS, où le développeur est responsable de l'organisation et du nettoyage (`free`).

Pile et Tas travaillent ensemble

```c
void maFonction(void) {
    int x = 10, *ptr = NULL; // x et ptr sont alloues sur la PILE

    ptr = malloc(5 * sizeof(int)); // Bloc mémoire réservé sur le TAS

    free(ptr); // ptr (sur la Pile) pointe vers le bloc memoire (sur le Tas)
}
```

La taille de la pile (Stack) par défaut

Une limite logicielle

La taille maximale de la pile n'est pas définie par le langage C, mais par le **système d'exploitation** lors du lancement de l'application.

Le risque d'*overflow*

Si les variables locales ou les appels récursifs dépassent cette taille, le programme s'arrête net avec une erreur de type *Stack Overflow*.

Système d'exploitation	Taille de la pile par défaut
Linux (thread principal)	8 Mo (8192 Ko)
macOS (thread principal)	8 Mo
Windows (MSVC / MinGW)	1 Mo (1024 Ko)
Linux (threads pthreads)	2 Mo

Comment redéfinir la taille de la pile ?

Sous Linux (Shell)

La modification est temporaire et s'applique à toutes les applications lancées depuis le terminal courant.

Configuration sous Linux (Bash)

```
# Definir la pile a 16 Mo
ulimit -s 16384

# Definir en illimite
ulimit -s unlimited
```

Sous Windows (compilateur)

La taille est gravée dans l'en-tête du fichier binaire exécutable (`.exe`) lors de la compilation.

Compilation sous Windows (GCC)

```
# Option en octets (16 Mo)
gcc main.c -o app -Wl,--stack,16777216
```

Dépassement de la pile : le Stack Overflow

La réaction de l'OS

Lorsque le programme tente d'allouer au-delà de la limite physique de la pile, il essaie d'écrire dans une zone mémoire non autorisée.

L'arrêt brutal

L'OS bloque immédiatement le processus pour protéger le système. Le programme plante instantanément.

Causes fréquentes en C

```c
// 1. Tableau statique trop grand
// (Ex: 10 Mo requis pour 8 Mo max)
double tab[1250000];

// 2. Recursivite infinie
void fonction_infinie(void) {
    int x = 5; // Alloue en boucle
    fonction_infinie();
}
```

Symptômes et messages

```
# Sous Linux / macOS :
Erreur de segmentation
Segmentation fault (core dumped)

# Sous Windows :
Process terminated
with status -1073741571
(0xC00000FD = Stack Overflow)
```

Variables et RAM : du nom à l'adresse mémoire

Pourquoi donner un nom ?

Il est impossible de retenir des adresses numériques brutes (ex : `0x7ffcc3`). Le nom d'une variable est une étiquette humaine simplifiée.

L'opérateur &

Il lève l'abstraction : placé devant une variable, il révèle l'adresse physique réelle de la case en mémoire RAM.

Affichage de la valeur et de son adresse

```c
#include <stdio.h>

int main(void) {
    int age = 20; // Etiquette humaine pour la valeur 20

    printf("Valeur de age : %d\n", age); // Affichera 20

    printf("Adresse en RAM : %p\n", (void*)&age); // 0x7ffeefbff56c

    return 0; }
```

Tableau de pointeurs : L'allocation dynamique

Le type double pointeur :

Puisque le tableau principal stocke des adresses mémoires vers des entiers (`int *`), sa propre adresse de base doit être manipulée via un type double pointeur (`int  `).

L'allocation en deux temps :

On réserve d'abord un espace pour y stocker le tableau de pointeurs, puis on parcourt ce tableau avec une boucle pour allouer l'espace de chaque ligne.

Étape 1 et 2 : Allocation du tronc puis des lignes :

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

int main(void) {
    int nb_lignes = 5;
    int tableau = NULL; // on veut un tableau de 5 lignes et 3 colonnes

    // 1. ALLOCATION DU TABLEAU PRINCIPAL (contient des adresses 'int *')
    tableau = malloc(nb_lignes * sizeof(int *));
    if (tableau == NULL) return 1;

    // 2. ALLOCATION DES DIVERSES LIGNES INDIVIDUELLEMENT (contient 3 'int')
    for (int i = 0; i < nb_lignes; i++) {
        tableau[i] = malloc(3 * sizeof(int));
        if (tableau[i] == NULL) return 1;
    }
```

Tableau de pointeurs : La désallocation

L'ordre inverse obligatoire :

Pour libérer proprement la mémoire, il faut impérativement désallouer les lignes *avant* de libérer le tableau principal.

Le piège de la fuite absolue :

Si vous faites `free(tableau)` en premier, vous perdez les adresses des lignes. Elles restent bloquées en RAM (fuite de mémoire).

Étape 3 et 4 : Libération des branches puis du tronc :

```
    // ... suite du code précédent ...

    // 3. DÉSALLOCATION DES LIGNES UNE PAR UNE
    for (int i = 0; i < nb_lignes; i++) {
        free(tableau[i]);
        tableau[i] = NULL; // Sécurisation du pointeur
    }

    // 4. DÉSALLOCATION DU TABLEAU PRINCIPAL
    free(tableau);
    tableau = NULL; // Évite le pointeur fou (dangling pointer)

    return 0;
}
```

Structures et types personnalisés

Les structures : définition et syntaxe

Définition

Une structure (`struct`) crée un nouveau type de données regroupant plusieurs variables (les **membres** ou **champs**) de types potentiellement différents sous un seul nom.

L'opérateur point (.)

Pour accéder aux membres d'une structure ou les modifier, on utilise l'opérateur de sélection directe représenté par un simple point.

Définition et utilisation d'une structure

```c
struct Echantillon { // 1. Definition du type Echantillon
    int id;
    double poids;
};
int main(void) { // 2. Declaration et initialisation d'une variable
    struct Echantillon e1 = {101, 14.5};
    e1.poids = 15.0; // 3. Accès et modification via le point (.)
    printf("Echantillon %d : poids = %.1f\n", e1.id, e1.note);
    return 0; }
```

Structures et pointeurs : l'opérateur ->

Pointeur vers une structure

Lorsqu'on manipule une structure via son adresse (par exemple lors d'un passage par adresse), on utilise un pointeur sur structure.

L'opérateur flèche (->)

La syntaxe `(*ptr).membre` est lourde.
Le C fournit un raccourci : `ptr->membre`.

Manipulation par pointeur

```c
struct Echantillon { int id; double note; };

void modifierPoids(struct Echantillon *ptr) { ptr->poids = 18.5; }

int main(void) {
    struct Echantillon e = {202, 11.0};
    modifierPoids(&e); // Passage par adresse
    return 0;
}
```

Simplification de la syntaxe : `typedef`

Le rôle de `typedef`

Le mot-clé `typedef` crée un **alias** (un nouveau nom raccourci) pour un type de données existant, primitif ou complexe.

Application aux structures

Principalement utilisé pour éviter de réécrire le mot-clé `struct` à chaque déclaration, rendant le code plus lisible.

Définition et utilisation d'un alias de structure

```c
typedef struct { // Definition structure avec alias combine
    int id;
    double poids;
} Echantillon;

int main(void) {
    Echantillon e1 = {303, 16.5}; // L'ecriture est simplifiee
    return 0;
}
```

Enumérations et unions

Les types personnalisés : Les Énumérations (enum)

Le rôle de enum :

Une énumération permet de créer un nouveau type dont les valeurs possibles sont limitées à une liste de constantes nommées (des identifiants textuels).

La réalité numérique :

Pour le processeur, ces mots-clés ne sont que des entiers relatifs. Par défaut, le compilateur attribue la valeur 0 au premier élément, puis 1, 2, etc.

Définition et utilisation d'une énumération en C :

```c
#include <stdio.h>

// Définition de l'énumération combinée avec un alias typedef
typedef enum {
    LUNDI,   // Vaut automatiquement 0
    MARDI,   // Vaut 1
    MERCREDI // Vaut 2
} Jour;

int main(void) {
    Jour aujourdhui = MARDI;

    // Le test logique est beaucoup plus lisible qu'avec des chiffres bruts
    if (aujourdhui == MARDI) {
        printf("Nous sommes le jour d'index : %d\n", aujourdhui); // Affiche 1
    }
    return 0; }
```

Les types personnalisés : Les Unions (`union`)

La superposition en RAM :

Une `union` ressemble syntaxiquement à une structure, mais ses membres partagent le même et unique espace mémoire. Modifier un membre écrase la valeur des autres.

L'intérêt (Économie de RAM) :

La taille d'une union est égale à la taille de son plus grand membre. Elle peut être utile en embarqué pour interpréter un même bloc d'octets sous plusieurs angles.

Exemple de superposition mémoire avec une union :

```c
#include <stdio.h>

typedef union {
    int entier;   // Occupe 4 octets
    char caractere; // Occupe 1 octet (superposé sur le premier octet de l'entier)
} Donnee;         // Taille totale en RAM = 4 octets (le maximum)

int main(void) {
    Donnee d;

    d.entier = 65; // On écrit l'entier complet

    // On lit le même espace mémoire sous forme de caractère (Table ASCII : 65 = 'A')
    printf("Lecture caractere : %c\n", d.caractere); // Affiche : A

    return 0;
}
```

Directives de préprocesseur et constantes

Le Préprocesseur : Les constantes avec `#define`

Le rôle du préprocesseur :

Il intervient avant la compilation réelle du code. Il effectue des transformations purement textuelles (copier-coller) sans analyser la syntaxe C.

La substitution textuelle :

La directive `#define` associe une étiquette à une valeur. Le préprocesseur remplace aveuglément l'étiquette partout dans le fichier.

Définition et substitution de constantes :

```c
#include <stdio.h>

#define PI 3.14159 // Attention : pas de signe '=' ni de ';' a la fin !
#define RAYON 5

int main(void) {
    // Le preprocesseur transformera cette ligne en : 2 * 3.14159 * 5
    double perimetre = 2 * PI * RAYON;

    printf("Perimetre = %f\n", perimetre);
    return 0;
}
```

Le Préprocesseur C et la directive `#define`

Qu'est-ce que le préprocesseur ?

C'est un outil qui analyse le code source **avant** la compilation réelle. Il effectue des manipulations purement textuelles (copier-coller automatique).

Éliminer les "nombres magiques" :

La directive `#define` associe un nom (généralement en MAJUSCULES) à une valeur. Lors de la pré-compilation, chaque nom est remplacé par sa valeur brute.

Définition et substitution de constantes de configuration :

```c
#include <stdio.h>

// Le preprocesseur remplacera textuellement ces mots partout dans le fichier
#define LARGEUR_CARTE 800
#define NB_VIES_MAX 3

int main(void) {
    int v_restantes = NB_VIES_MAX; // Devient en coulisses : int v_restantes = 3;

    printf("Configuration du jeu : %d px de large.\n", LARGEUR_CARTE);
    printf("Nombre de tentatives autorisees : %d\n", v_restantes);

    return 0;
}
```

Le Préprocesseur : Les macros avec paramètres

Des fonctions textuelles :

Une macro peut accepter des paramètres. Elle permet d'injecter un morceau de calcul rapide sans le coût matériel de l'appel d'une vraie fonction.

Le piège des parenthèses :

Puisqu'il s'agit d'un simple remplacement de texte, il faut impérativement entourer les paramètres de parenthèses pour éviter les erreurs de priorité mathématique.

Danger des priorités d'évaluation dans les macros :

```c
// MAUVAISE PRATIQUE : #define CARRE(x) x * x -> CARRE(1+1) devient 1 + 1 * 1 + 1 = 3 !
// BONNE PRATIQUE : parenthèses systématiques
#define CARRE(x) ((x) * (x))

int main(void) {
    int resultat = CARRE(3); // Devient : ((3) * (3)) = 9
    int piege = CARRE(1 + 1); // Devient : ((1 + 1) * (1 + 1)) = 4

    return 0;
}
```

Le Préprocesseur : La compilation conditionnelle

Sélectionner le code :

Les directives `#ifdef` (*if defined*) et `#ifndef` (*if not defined*) permettent d'inclure ou d'exclure des lignes de code avant la compilation.

Usages courants :

Idéal pour insérer du code de débogage temporaire, gérer du code multi-plateforme (Linux/Windows) ou concevoir des protections d'en-tête (*include guards*).

Activation dynamique d'un bloc de débogage :

```c
#include <stdio.h>

#define DEBUG_MODE // Si on commente cette ligne, le bloc de logs disparait du binaire

int main(void) {
    int x = 42;

#ifdef DEBUG_MODE
    // Ce code ne sera compile que si DEBUG_MODE est defini
    printf("[LOG] La variable x vaut %d\n", x);
#endif

    printf("Fin du programme.\n");
    return 0;
}
```

Le Préprocesseur : Les protections d'en-tête

Le problème du doublon :

Dans un grand projet, un fichier `.h` peut être inclus plusieurs fois indirectement. Le préprocesseur duplique alors le code, provoquant des erreurs de double déclaration.

La solution d'isolation :

Les **Include Guards** utilisent le préprocesseur pour verrouiller le fichier. Ils garantissent que le contenu d'un en-tête n'est copié qu'une seule et unique fois, peu importe le nombre d'appels.

Structure robuste d'un fichier d'en-tête (`outils.h`) :

```
#ifndef OUTILS_H // Étape 1 : Si l'étiquette n'est pas encore définie...
#define OUTILS_H // Étape 2 : On la définit immédiatement

// --- TOUTES VOS DÉCLARATIONS DE FONCTIONS ET STRUCTURES ICI ---
void maFonctionUtile(int a);

#endif // Étape 3 : Fin du bloc conditionnel. Le fichier est sécurisé.
```

Alternative moderne : La plupart des compilateurs actuels supportent également la directive unique `#pragma once` placée tout en haut du fichier pour jouer exactement le même rôle.

Code source sur plusieurs fichiers

La compilation séparée : Fichiers .h et .c

Pourquoi diviser le code ?

Écrire tout son code dans un unique fichier `main.c` devient vite ingérable. Diviser le projet en plusieurs modules permet de clarifier la structure et de réutiliser le code.

La répartition des rôles :

Chaque module métier du programme est scindé en deux entités distinctes :

- Le fichier d'en-tête (`.h`)
- Le fichier de code (`.c`)

Extension	Contenu standard	Rôle pédagogique
Fichier `.h`	Prototypes de fonctions, structures, macros	**La vitrine** (Ce que le module sait faire
Fichier `.c`	Corps et logique interne des fonctions	**L'usine** (Comment le module le fait)

Exemple de module et protection des en-têtes

Le piège des doubles inclusions :

Si un fichier inclut plusieurs fois le même `.h`, le compilateur verra des déclarations en double et plantera. On utilise obligatoirement des **gardes d'inclusion** (`#ifndef`) pour sécuriser le fichier.

L'en-tête : `joueur.h`

```
1  #ifndef JOUEUR_H
2  #define JOUEUR_H
3
4  // Prototype (Vitrine)
5  void soigner_joueur(int *pv);
6
7  #endif
```

Le code : `joueur.c`

```
1  #include "joueur.h"
2  #include <stdio.h>
3
4  // Definition (Usine)
5  void soigner_joueur(int *pv) {
6      *pv += 20;
7      printf("Soin applique !\n");
8  }
```

Note syntaxique : On utilise des `"guillemets"` pour nos fichiers locaux, et des `<chevrons>` pour les bibliothèques standards système.

Manipulation des fichiers

Manipulation des fichiers : ouverture et fermeture

Le type `FILE *`

En C, un fichier est manipulé à travers un pointeur sur une structure `FILE`, définie dans `<stdio.h>`.

Les modes principaux :

- `"r"` : Lecture seule (*read*)
- `"w"` : Écriture (*write*, écrase le fichier)
- `"a"` : Ajout (*append*, écrit à la fin)

Sécurité et restitution

L'ouverture peut échouer (fichier inexistant, droits insuffisants). Il faut **toujours** vérifier que le pointeur n'est pas `NULL`.

Chaque fichier ouvert doit être refermé avec `fclose()` pour vider les tampons et libérer les ressources système.

Manipulation des fichiers : ouverture sécurisée

Ouverture et fermeture sécurisée

```c
#include <stdio.h>
int main(void) {
    FILE *fichier = NULL;

    fichier = fopen("donnees.txt", "w"); // Mode ecriture

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

    // ... operations d'ecriture ici ...

    fclose(fichier); // Fermeture obligatoire
    return 0;
}
```

Manipulation des fichiers : écriture et lecture

La fonction `fprintf()`

Fonctionne comme `printf()`, mais prend en premier argument le pointeur du fichier pour y écrire du texte formaté.

La fonction `fscanf()`

Fonctionne comme `scanf()`. Lit les données depuis le fichier selon un format précis. Attention à bien passer les **adresses** des variables de réception.

Exemple d'écriture et lecture formatée

```c
FILE *f_ecriture = fopen("score.txt", "w"); // 1. ECRITURE
if (f_ecriture != NULL) {
    fprintf(f_ecriture, "%d %.2f", 101, 14.5);
    fclose(f_ecriture);
}

FILE *f_lecture = fopen("score.txt", "r"); // 2. LECTURE
if (f_lecture != NULL) {
    int id; double note;
    fscanf(f_lecture, "%d %lf", &id, &note);
    printf("Lu : ID=%d, Note=%.1f\n", id, note);
    fclose(f_lecture);
}
```

Lecture de fichiers .txt multi-encodages

La locale du système : L'en-tête `<locale.h>` indique au programme l'encodage qui sera utilisé lors de l'exécution :

- `setlocale(LC_ALL, "");` → S'adapte à l'OS de l'utilisateur.
- `setlocale(LC_ALL, "fr_FR.UTF-8");` → Force l'UTF-8 en français.
- `setlocale(LC_ALL, "fr_FR.ISO-8859-1");` → Force le Latin-1.

Le problème des flux bruts :

Les fonctions comme `fgets()` lisent des octets bruts sans se soucier du format. Si vous lisez un fichier codé en UTF-8 avec un programme configuré en Latin-1, l'affichage sera corrompu.

La solution des "Wide Characters" :

Pour manipuler sereinement des fichiers aux encodages variés, le C standard propose les caractères larges (`wchar_t`) et les fonctions associées dans `<wchar.h>`.

Lecture de fichiers .txt multi-encodages

Exemple de lecture sécurisée avec gestion de la locale (UTF-8/Unicode) :

```c
#include <stdio.h>
#include <locale.h>
#include <wchar.h>

int main(void) {
    // Synchronise le programme avec l'encodage du systeme (ex: UTF-8)
    setlocale(LC_ALL, "");

    // Lecture d'un fichier avec des caracteres larges (w = wide)
    FILE *fichier = fopen("texte_utf8.txt", "r");
    if (fichier != NULL) {
        wchar_t ligne[100];
        // fgetws lit proprement la ligne en conservant le codage multi-octets
        while (fgetws(ligne, 100, fichier) != NULL) {
            wprintf(L"Ligne lue : %ls\n", ligne);
        }
        fclose(fichier);
    }
    return 0;
}
```

Lecture de fichiers .txt au format ISO 8859-1

Si le fichier source externe utilise un encodage mono-octet hérité comme l'ISO 8859-1 (Latin-1), il faut forcer explicitement la locale du programme lors de l'initialisation pour que la conversion en caractères larges s'opère correctement.

Exemple de lecture d'un fichier encodé en Latin-1 (ISO 8859-1) :

```c
#include <stdio.h>
#include <locale.h>
#include <wchar.h>

int main(void) {
    // Force le programme a interpreter le flux d'entree en Latin-1
    // (Note sous Windows : utiliser "French_France.1252")
    setlocale(LC_ALL, "fr_FR.ISO-8859-1");

    // Ouverture et lecture avec conversion automatique en wchar_t
    FILE *fichier = fopen("texte_latin1.txt", "r");
    if (fichier != NULL) {
        wchar_t ligne[100];

        while (fgetws(ligne, 100, fichier) != NULL) {
            wprintf(L"Ligne lue : %ls\n", ligne);
        }
        fclose(fichier);
    }
    return 0;
}
```

Manipulation des fichiers au format binaire

Le mode binaire (`"wb"`, `"rb"`)

Contrairement au mode texte qui convertit les données en caractères lisibles, le mode binaire copie **exactement les bits de la RAM** dans le fichier, sans transformation.

Fonctions `fwrite` et `fread`

Elles écrivent ou lisent un bloc complet de mémoire (tableaux, structures) d'un seul coup en spécifiant l'adresse, la taille d'un élément et le nombre d'éléments.

Écriture brute en binaire

```c
int tab_out[3] = {10, 20, 30}, int tab_in[3];

FILE *f_out = fopen("data.bin", "wb");

if (f_out != NULL) {
    // Copie 12 octets de la RAM
    fwrite(tab_out, sizeof(int), 3, f_out);
    fclose(f_out);
}
```

Lecture brute en binaire

```c
if (f_in != NULL) {
    // Restitue les 12 octets en RAM
    fread(tab_in, sizeof(int), 3, f_in);
    fclose(f_in);
}
```

Fichiers : Le curseur de position et `fseek()`

Le curseur de lecture/écriture :
Lorsqu'un fichier est ouvert, l'OS maintient un curseur invisible indiquant la position de la prochaine action. Lire ou écrire fait avancer ce curseur automatiquement.

Le déplacement avec `fseek()` :
La fonction `fseek()` de `<stdio.h>` permet de déplacer manuellement ce curseur à un endroit précis du fichier sans avoir à le lire en entier.

Les 3 origines de positionnement possibles :

- `SEEK_SET` : Calcule le déplacement à partir du début du fichier.
- `SEEK_CUR` : Calcule le déplacement à partir de la position actuelle du curseur.
- `SEEK_END` : Calcule le déplacement à partir de la fin du fichier (idéal pour les décalages négatifs ou mesurer une taille).

Fichiers : Exemple de manipulation de curseur

Code C de déplacement et lecture ciblée :

```c
#include <stdio.h>

int main(void) {
    FILE *fichier = fopen("texte.txt", "r"); // Contenu supposé : "ABCDEF"
    if (fichier == NULL) return 1;

    char lettre;

    // Se déplacer àla 3ème lettre (index 2 depuis le début)
    fseek(fichier, 2, SEEK_SET);
    lettre = fgetc(fichier); // Lit et fait avancer le curseur
    printf("Lettre lue : %c\n", lettre); // Affiche : C

    // Avancer de 2 caractères àpartir de la position actuelle
    fseek(fichier, 2, SEEK_CUR);
    lettre = fgetc(fichier);
    printf("Lettre lue : %c\n", lettre); // Affiche : F

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

Fichiers : Connaître la position avec `ftell()`

Le rôle de `ftell()` :

La fonction `ftell()` de `<stdio.h>` renvoie un entier long (`long`) représentant la position actuelle du curseur (en octets) par rapport au début du fichier.

Mesurer la taille d'un fichier :

En combinant `fseek()` et `ftell()`, on peut calculer précisément la taille d'un fichier : on déplace le curseur tout à la fin, puis on demande sa position numérique.

Code C pour obtenir la taille d'un fichier en octets :

```c
#include <stdio.h>

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

    fseek(fichier, 0, SEEK_END); // 1. Envoyer le curseur àla fin du fichier

    long taille = ftell(fichier); // 2. Récupérer la position actuelle (= taille totale)

    printf("La taille du fichier est de %ld octets.\n", taille);
    fclose(fichier);
```

Fichiers et UTF-8 : Le piège *Octet* vs *Caractère*

La réalité de l'UTF-8 :

Dans les fichiers modernes encodés en **UTF-8**, la taille d'un caractère est variable. Un caractère de base (A-Z) fait 1 octet, mais un accent (é) en fait 2, et un émoji en fait 4.

La limite de `ftell()` :

La fonction `ftell()` mesure uniquement des octets. Dans un fichier UTF-8 contenant du texte accentué, la valeur renvoyée par `ftell()` sera supérieure au nombre réel de caractères.

Solution en C : Compter les octets de départ en UTF-8

```c
// En UTF-8, un octet qui ne commence PAS par les bits '10' est le debut d'un nouveau caractere.
// Les octets suivants d'un meme caractere commencent toujours par '10' (soit 0x80 en masque).
long compterCaracteresUTF8(FILE *fichier) {
    rewind(fichier);
    long nb_caracteres = 0;
    int octet;

    while ((octet = fgetc(fichier)) != EOF) {
        // Si l'octet n'est pas un octet de continuation (bits de poids fort != 10)
        if ((octet & 0xC0) != 0x80) {
            nb_caracteres++;
        }
    }
    return nb_caracteres;
}
```

Interactions avec le système

Les arguments de la ligne de commande : `main()`

L'interaction avec le Shell

Il est possible de transmettre des paramètres textuels à un programme lors de son lancement depuis le terminal de commande.

La signature complète

Pour intercepter ces paramètres, la fonction d'entrée utilise le prototype standardisé : `int main(int argc, char *argv[])`

- **`argc`** (*Argument Count*) : entier stockant le nombre total d'arguments détectés (au minimum 1).
- **`argv`** (*Argument Vector*) : tableau de chaînes de caractères contenant les arguments textuels.
  - `argv[0]` contient **toujours** le nom ou le chemin de l'exécutable lui-même.
  - `argv[1]` à `argv[argc-1]` contiennent les options entrées par l'utilisateur.

Utilisation pratique de `argc` et `argv`

Le programme ci-dessous vérifie la présence d'au moins un argument dans le terminal, puis boucle sur le tableau pour afficher tous les paramètres reçus.

Gestion des arguments en ligne de commande :

```c
#include <stdio.h>

int main(int argc, char *argv[]) {
    // argv[0] est le nom du programme
    if (argc < 2) {
        printf("Usage : %s <arg1> <arg2> ...\n", argv[0]);
        return 1;
    }

    printf("Nombre d'arguments recus : %d\n", argc - 1);
    for (int i = 1; i < argc; i++) {
        printf("Argument [%d] = %s\n", i, argv[i]);
    }

    return 0;
}
```

Communication avec l'OS : les signaux

Qu'est-ce qu'un signal ?

Un signal est une notification asynchrone envoyée par le système d'exploitation (ou un autre processus) pour avertir l'application qu'un événement matériel ou logiciel s'est produit.

Le comportement par défaut

Par défaut, la majorité des signaux système (comme l'interruption par l'utilisateur) provoquent l'arrêt brutal et immédiat du programme.

Détourner le signal (le handler)

Grâce à `<signal.h>`, une application peut intercepter ces signaux pour exécuter une fonction personnalisée (un *handler*) avant de réagir. Cela permet par exemple de sauvegarder des fichiers, nettoyer la mémoire ou ignorer l'ordre d'arrêt.

Les principaux signaux et raccourcis clavier

Signal	Raccourci	Action par défaut	Usage standard
`SIGINT`	`Ctrl + C`	Arrêt immédiat	Interruption volontaire
`SIGQUIT`	`Ctrl + \`	Arrêt avec *core dump*	Quitter brutalement
`SIGTSTP`	`Ctrl + Z`	Suspension du processus	Mettre en pause en arrière-plan
`SIGTERM`	*Aucun*	Arrêt propre	Demande de fin (commande `kill`)
`SIGKILL`	*Aucun*	Arrêt forcé absolu	Tuer le processus (**non interceptable**)

Le signal `SIGKILL` (envoyé via `kill -9`) contourne le programme. Il est impossible de l'intercepter ou de l'ignorer par le code.

Interception d'un signal avec `signal()`

Pour intercepter un signal (comme `SIGINT` déclenché par `Ctrl+C`), on associe le signal à une fonction de rappel personnalisée (*handler*) qui reçoit le numéro du signal en paramètre.

Capture du Ctrl+C avec signal()

```c
#include <stdio.h>
#include <signal.h>
#include <unistd.h> // Pour sleep()

// Fonction de rappel declenchee a la reception du signal
void gerer_sigint(int num_signal) {
    printf("\n[Signal %d] Interception de Ctrl+C ! Nettoyage en cours...\n",
           num_signal);
}

int main(void) {
    signal(SIGINT, gerer_sigint); // Cablage du handler sur le signal SIGINT

    printf("Programme en cours. Appuyez sur Ctrl+C pour tester...\n");
    while(1) sleep(1); // Boucle infinie simulant un travail
    return 0; }
```

Fonctions récursives

Les fonctions récursives : Le concept

Définition :
Une fonction est dite récursive lorsqu'elle s'appelle elle-même à l'intérieur de son propre corps de code pour résoudre un sous-problème.

Les deux éléments obligatoires :

- **Le cas de base** : La condition d'arrêt qui stoppe les appels (essentielle pour éviter le *Stack Overflow*).
- **Le cas récursif** : L'appel de la fonction sur une donnée plus petite.

Le mécanisme de la Pile d'exécution :
À chaque auto-appel, l'OS empile les variables locales et l'adresse de retour dans la Pile (Stack). Tant que le cas de base n'est pas atteint, aucune fonction ne se termine. Elles restent toutes en attente, suspendues en mémoire.

Récursivité : Exemple de la Factorielle ($n!$)

La factorielle d'un entier n se définit mathématiquement par récurrence :

- Cas de base : $0! = 1$
- Cas récursif : $n! = n \times (n-1)!$

Implémentation récursive de la factorielle en C :

```c
#include <stdio.h>

long factorielle(int n) {
    // 1. Le cas de base (Condition d'arret)
    if (n == 0) {
        return 1;
    }

    // 2. Le cas recursif
    return n * factorielle(n - 1);
}

int main(void) {
    // Calcule : 4 * factorielle(3) -> 4 * 3 * factorielle(2) ... -> 24
    printf("4! = %ld\n", factorielle(4));
    return 0;
}
```

Efficacité mémoire : Récursif vs Itératif

Le coût de la Récursivité :

Chaque appel récursif consomme de l'espace sur la **Pile (Stack)** pour stocker le contexte. Pour N étapes, le coût mémoire est proportionnel à N. L'exécution est plus lente en raison du traitement des appels système.

L'avantage de l'Itération :

Une boucle `while` ou `for` s'exécute au sein du même et unique cadre de pile. Le coût mémoire est constant (1 seule allocation), peu importe le nombre de tours N. L'exécution est plus rapide.

Comparaison sur le calcul d'une somme de 1 à N :

```c
// APPROCHE RECURSIVE : Consomme N cadres sur la Pile (Risque de Stack Overflow)
int sommeRecursive(int n) {
    if (n <= 0) return 0;
    return n + sommeRecursive(n - 1);
}

// APPROCHE ITERATIVE : Consomme 1 seul cadre sur la Pile (Efficace et stable)
int sommeIterative(int n) {
    int resultat = 0;
    while (n > 0) {
        resultat += n;
        n--;
    }
    return resultat;
}
```

Cinq classes d'allocation

Sécurisation du code : Le mot-clé `const`

Une variable en lecture seule :

Le mot-clé `const` modifie le comportement d'une variable pour la transformer en constante. Une fois initialisée, sa valeur ne peut plus être modifiée par le programme.

Sécurité des pointeurs :

Il est indispensable lors du passage d'arguments par adresse (pointeurs) à une fonction pour garantir que la fonction lira la donnée sans risquer de la modifier par erreur.

Protection des variables et des paramètres en C :

```c
#include <stdio.h>

// La fonction reçoit l'adresse mais s'interdit d'y écrire
void afficherValeur(const int *ptr) {
    // *ptr = 50; // ERREUR DE COMPILATION : Ecriture interdite !
    printf("Valeur : %d\n", *ptr);
}

int main(void) {
    const double pi = 3.14159; // Initialisation obligatoire
    // pi = 3.14; // ERREUR DE COMPILATION : Modification interdite !

    int x = 42;
    afficherValeur(&x);

    return 0; }
```

Classe de stockage : Le mot-clé `static`

Persistance locale :

Placé devant une variable locale, `static` lui permet de conserver sa valeur d'un appel à l'autre de la fonction. Elle n'est initialisée qu'une seule fois au début du programme.

Masquage global :

Placé devant une variable globale ou une fonction, `static` restreint sa visibilité au seul fichier source courant, la rendant invisible pour le reste du projet.

Exemple de compteur persistant avec static :

```c
#include <stdio.h>

void incrementer(void) {
    static int compteur = 0; // Initialise une seule fois, survit a la fin du bloc
    compteur++;
    printf("Compteur = %d\n", compteur);
}

int main(void) {
    incrementer(); // Affiche 1
    incrementer(); // Affiche 2 (la valeur a survecu)
    return 0;
}
```

Classe de stockage : Le mot-clé historique `auto`

Le stockage automatique :

Par défaut, toute variable déclarée dans un bloc de code sans mot-clé spécifique appartient à la classe de stockage `auto` (*automatic duration*).

La gestion par la Pile :

Sa mémoire est allouée automatiquement sur la Pile à l'entrée du bloc et libérée sans intervention du développeur dès la sortie du bloc.

Redondance du mot-clé auto en C standard :

```c
#include <stdio.h>

int main(void) {
    // Les deux declarations suivantes sont strictement identiques :
    int x = 10;
    auto int y = 20; // Le mot-cle 'auto' est implicite et facultatif

    // Remarque : En C++ moderne (depuis C++11), 'auto' a ete completement
    // redéfini pour l'inférence automatique de type. En C, il reste historique.

    return 0;
}
```

Classe de stockage : Le mot-clé `extern`

Déclaration vs Définition :

Le mot-clé `extern` indique au compilateur qu'une variable (ou fonction) globale existe, mais qu'elle est définie et allouée dans un autre fichier source du projet.

Le rôle du Linker :

Il permet d'éviter les erreurs de double définition lors de la compilation de projets multifichiers, en reportant la résolution de l'adresse au moment de l'édition de liens.

Utilisation de extern pour lier deux fichiers :

```c
// --- FICHIER 1 : variables.c ---
int score_maximum = 999; // Definition physique de la variable globale


// --- FICHIER 2 : main.c ---
#include <stdio.h>

// On informe le compilateur que la variable existe ailleurs (pas d'allocation RAM ici)
extern int score_maximum;

int main(void) {
    printf("Le score max est : %d\n", score_maximum);
    return 0;
}
```

Classe de stockage : Le mot-clé `register`

Optimisation de vitesse :

Le mot-clé `register` suggère poliment au compilateur de stocker la variable directement dans un registre ultra-rapide du CPU plutôt qu'en mémoire RAM, afin d'accélérer les calculs intensifs.

La restriction majeure :

Une variable résidant dans un registre du CPU n'a pas de case en mémoire vive. Il est donc strictement interdit d'utiliser l'opérateur d'adresse `&` sur une variable `register`.

Utilisation historique de register (ex : compteurs de boucles critiques) :

```c
#include <stdio.h>

int main(void) {
    // Suggestion pour une variable tres sollicitee
    register int i;
    int somme = 0;

    for (i = 0; i < 100000; i++) {
        somme += i;
    }

    // int *ptr = &i; // ERREUR DE COMPILATION : impossible d'obtenir l'adresse !

    return 0;
}
```

Listes chaînées

Structures de données : Le concept de Liste Chaînée

La limite des tableaux :
Un tableau classique a une taille fixe et nécessite un bloc de mémoire contigu. Insérer un élément au milieu oblige à décaler toutes les cases suivantes.

La flexibilité du chaînage :
Une liste chaînée est une suite d'éléments (appelés nœuds) éparpillés dans le Tas. Chaque nœud contient sa donnée et un pointeur vers le nœud suivant.

Caractéristiques clés :

- **Taille dynamique** : La liste s'agrandit ou se rétrécit à volonté pendant l'exécution, maillon par maillon.
- **Insertion rapide** : Pour ajouter un élément, il suffit de modifier des pointeurs (des flèches), sans déplacer les données en RAM.
- **La fin de liste** : Le tout dernier maillon pointe vers la constante `NULL` pour marquer la fin de la chaîne.

Listes Chaînées : Structure et Maillage en C

La structure auto-référencée :

Pour coder un nœud, on utilise une structure qui contient un pointeur vers un élément du même type que la structure elle-même.

Le maillage manuel :

On utilise `malloc()` pour réserver chaque maillon individuellement dans le Tas, puis l'opérateur flèche `->` pour lier les adresses entre elles.

Les étapes de création d'une liste :

- **Étape 1** : Définir la structure du nœud avec son pointeur interne.
- **Étape 2** : Allouer la tête de liste et lui affecter une valeur.
- **Étape 3** : Allouer le nœud suivant et lui donner l'adresse de fin `NULL`.
- **Étape 4** : Relier le pointeur du premier nœud vers l'adresse du second.

Listes Chaînées : Exemple de maillage en C

Code de création de deux maillons reliés dans le Tas :

```c
#include <stdlib.h>

// 1. Definition du type Noeud avec un alias typedef
typedef struct Element {
    int valeur;
    struct Element *suivant; // Pointeur auto-reference
} Node;

int main(void) {
    // 2. Allocation dynamique du premier maillon (la Tete)
    Node *tete = malloc(sizeof(Node));
    tete->valeur = 10;

    // 3. Allocation du second maillon
    Node *deuxieme = malloc(sizeof(Node));
    deuxieme->valeur = 20;
    deuxieme->suivant = NULL; // Dernier maillon, fin de liste

    // 4. Chainage physique des deux blocs en memoire Tas
    tete->suivant = deuxieme;
    return 0; }
```

Listes Chaînées : Le principe du parcours

Le pointeur mobile :
Pour traverser une liste chaînée, on utilise un pointeur temporaire (souvent nommé `courant`) qui va sauter de maillon en maillon.

Préserver la tête de liste :
Il ne faut **jamais** déplacer le pointeur `tete` d'origine pour faire un parcours, sous peine de perdre définitivement l'accès au début de la structure.

La condition d'arrêt :

- La boucle avance tant que le pointeur `courant` n'est pas égal à `NULL`.
- L'avancement se fait par la ligne mécanique : `courant = courant->suivant;`

Listes Chaînées : Code de la fonction d'affichage

Implémentation du parcours et de l'affichage en C :

```c
#include <stdio.h>

typedef struct Element {
    int valeur;
    struct Element *suivant;
} Node;

void afficherListe(Node *tete) {
    // Initialisation au debut de la liste
    Node *courant = tete;

    // Boucle jusqu'a la balise de fin NULL
    while (courant != NULL) {
        printf("%d -> ", courant->valeur);

        // Saut vers le maillon suivant
        courant = courant->suivant;
    }
    printf("NULL\n");
}
```

Listes Chaînées : Le principe de désallocation

Le piège de la suppression :
Si vous faites `free(courant)` directement, vous détruisez instantanément le maillon ainsi que le pointeur `suivant` qu'il contenait.

La sauvegarde temporaire :
Il faut impérativement copier l'adresse du maillon suivant dans une variable tampon avant de libérer le maillon actuel.

L'ordre logique à chaque tour de boucle :

1. Étape 1 : Sauvegarder le pointeur vers le maillon suivant.
2. Étape 2 : Libérer la mémoire du maillon actuel avec `free()`.
3. Étape 3 : Déplacer le pointeur principal sur le maillon sauvegardé.

Listes Chaînées : Code de la désallocation

Implémentation de la libération propre du Tas :

```c
#include <stdlib.h>

typedef struct Element {
    int valeur;
    struct Element *suivant;
} Node;

void libererListe(Node *tete) {
    Node *courant = tete;
    Node *suivantTemporaire = NULL;

    while (courant != NULL) {
        // 1. Sauvegarde essentielle
        suivantTemporaire = courant->suivant;

        // 2. Liberation du bloc actuel
        free(courant);

        // 3. Bascule sur le maillon sauvegarde
        courant = suivantTemporaire;
    }
```

Usage de l'IA pour le langage C en 2026

Pratique moderne : L'usage de l'IA pour coder en C

Un assistant puissant :
Les IA actuelles (générateurs de code, LLM) excellent pour expliquer des concepts complexes, traduire un algorithme en C ou générer des structures de base rapidement.

Le piège de la confiance :
Le langage C n'a aucun filet de sécurité. Une IA peut générer du code syntaxiquement correct, mais contenant des failles de sécurité invisibles à première vue.

Les 3 règles d'or face à une IA en programmation C :

- **Ne jamais copier-coller sans comprendre** : Vous devez être capable de justifier et d'expliquer chaque ligne, chaque pointeur et chaque allocation de mémoire.
- **Traquer les comportements indéterminés** : Les IA oublient parfois de vérifier si `malloc` renvoie `NULL` ou si un index de tableau déborde. Soyez plus rigoureux qu'elles.
- **Utiliser l'IA pour apprendre** : Demandez-leur de relire votre code C pour y déceler des fuites de mémoire ou pour vous expliquer un message d'erreur du compilateur GCC.

L'IA face à la dualité Applicatif vs Embarqué

Le Dev Applicatif (Sous OS) :
Sur les environnements standards (PC, Linux, serveurs), l'IA est d'une rigueur redoutable. Elle gère souvent mieux la mémoire, les formats et la sécurité (`malloc`, fichiers) qu'un humain débutant.

Le Dev Embarqué (Bare-Metal) :
Ici, l'IA se heurte au mur du matériel. Elle ne connaît pas le microcontrôleur, les adresses des registres bruts, ni les contraintes de temps réel à la microseconde.

La posture du développeur moderne :

- **L'IA pilote, vous êtes l'arbitre** : En applicatif, l'IA produit un code excellent, mais vous devez savoir le relire pour valider son intégration.
- **En embarqué, vous êtes le traducteur** : Vous devez guider l'IA en lui fournissant la documentation technique matérielle (*Datasheet*), car elle ne peut pas deviner l'architecture physique de votre circuit imprimé.

Apprentissage : Le piège de l'illusion de maîtrise

L'illusion de la fluidité :
Lire un code C bien structuré ou généré par une IA produit une sensation d'évidence. Comprendre à la lecture n'est pas savoir faire. C'est une illusion cognitive.

Lire n'est pas écrire :
Reconnaître la syntaxe d'une liste chaînée est passif. Concevoir l'algorithme, manipuler les pointeurs et gérer la mémoire RAM sans aide est un effort actif radicalement différent.

Comment surmonter ce piège face à l'IA ?

- **Le test de la page blanche** : Si vous comprenez une solution générée par une IA, fermez l'onglet et tentez de la réécrire seul, de mémoire. C'est là que l'apprentissage commence.
- **Le prix de l'erreur** : C'est en luttant contre un *Segmentation Fault* et en lisant les messages de GCC que le cerveau câble les connexions logiques. L'IA ne doit pas vous priver de cette lutte nécessaire.
- **L'analogie sportive** : Regarder un champion de natation à la télévision est limpide. Sauter dans l'eau et reproduire le même mouvement requiert un entraînement musculaire et pratique.

Conclusion et fin

Conclusion et pour aller plus loin

- L'informatique repose sur une chaîne d'abstractions : du transistor au langage évolué, chaque couche masque la complexité de la précédente.
- Le langage C étant **proche du matériel** : il donne accès à la mémoire, aux adresses et aux bits, ce qui le rend à la fois puissant et exigeant.
- Maîtriser le C, c'est comprendre ce que fait *réellement* la machine derrière chaque instruction.

Pour approfondir :

- *Le langage C* — Kernighan & Ritchie (la bible).
- *Modern C* — Jens Gustedt (gratuit, C11/C17).
- La documentation de référence de votre compilateur (GCC, Clang).

Merci !

Questions ?