Cour_Sur_Le_Language_C

Sommaire

Partie 1 : Les Fondamentaux du C

- Blocs d'instructions (`{}`)
- Instruction `if`, `else if`, `else`
- Instruction `switch`, `case`, `break`, `default` (Limitations)

7. -Structures de Contrôle : Boucles

- Boucle `while`
- Boucle `for` (Les 3 parties, Variantes)
- Boucle `do...while`
- Instructions `break` et `continue`
- Boucles imbriquées

8. -La Modularité : Les Fonctions

- Pourquoi utiliser des fonctions ?
- Déclaration (Prototype) et Définition
- Appel de fonction
- Paramètres et Arguments (Passage par valeur)
- **Arguments de la Ligne de Commande (`argc`, `argv`)**
- Valeur de Retour (`return`)
- Portée des Variables (Locales vs Globales)
- Variables Statiques (`static` locales)
- Récursivité (Introduction)

Partie 2 : Types Composites et Mémoire

9. -Les Tableaux

- Tableaux à une dimension (Déclaration, Initialisation, Accès, Limites)
- Tableaux et Boucles
- Tableaux Multidimensionnels (Matrices)

10. -Chaînes de Caractères

- Représentation (Tableaux de `char` terminés par `\0`)
- Initialisation (Littéraux de chaîne)
- **La bibliothèque `<string.h>` :**
  - Copie : `strcpy`, `strncpy` (Sécurité !)
  - Concaténation : `strcat`, `strncat` (Sécurité !)
  - Comparaison : `strcmp`, `strncmp`
  - Longueur : `strlen`
  - Recherche : `strchr`, `strstr` (Optionnel)
- **La bibliothèque `<ctype.h>` (`toupper`, `is...`)**
- **Formatage de chaînes (`sprintf`, `snprintf`)**
- Lecture de chaînes sécurisée (`fgets`) vs `scanf("%s")`

11. -Les Pointeurs

- Adresses mémoire
- Déclaration et Initialisation de pointeurs (`type *ptr`)
- Opérateurs `&` (adresse de) et `*` (déréférencement)
- Pointeurs `NULL` et vérifications
- Pointeurs et Tableaux (Relation étroite, Notation `ptr[i]`)
- **Passage par Adresse (Simulation du passage par référence)**
- Arithmétique des pointeurs (Mise à l'échelle `sizeof`)
- **Le qualificateur `const` avec les pointeurs (`const T*`, `T* const`, `const T* const`)**

12. -Allocation Dynamique de Mémoire (`<stdlib.h>`)

- La Pile vs Le Tas (Stack vs Heap)
- `malloc()` : Allocation brute (Vérification `NULL` !)
- `calloc()` : Allocation initialisée à zéro
- `free()` : Libération de la mémoire (Obligatoire !)
- `realloc()` : Redimensionnement (Utilisation sûre avec variable temporaire)
- Erreurs courantes : Fuites mémoire, Dangling Pointers, Double Free

13. -Les Structures (`struct`)

- Définition (`struct Tag { ... };`)
- Déclaration et Initialisation (Ordonnée, Désignée C99+)
- Accès aux membres (`.`)
- Pointeurs vers des structures et opérateur flèche (`->`)
- Structures et Allocation Dynamique
- `typedef` pour créer des alias de type structure
- Assignation et Copie de structures
- Structures imbriquées
- Tableaux de structures

14. -Unions et Énumérations

- Unions (`union`) : Partager la mémoire, Type Punning (avec précautions)
- Énumérations (`enum`) : Définir des constantes nommées

15. -Pointeurs Avancés et Pointeurs de Fonction

- Pointeurs vers pointeurs (`type **ptr`)
- Tableaux de pointeurs
- **Pointeurs de Fonction :**
  - Déclaration et Syntaxe
  - Assignation et Appel via pointeur
  - Passage en argument (Callbacks)
  - Exemple : `qsort()` de `<stdlib.h>`

Partie 3 : Gestion de Projets et Fichiers

16. -Le Préprocesseur C

- Rôle et Étapes de préprocessing
- Directive `#include` (`<...>` vs `"..."`)
- Directive `#define` (Constantes symboliques, Macros type fonction - Précautions !)
- Directive `#undef`
- Macros prédéfinies (`__FILE__`, `__LINE__`, ...)
- Compilation Conditionnelle (`#if`, `#ifdef`, `#ifndef`, `#else`, `#elif`, `#endif`, `defined()`)
- **Header Guards (`#ifndef`/`#define`/`#endif`)**
- Directives `#error`, `#warning`, `#pragma` (mention)

17. -Organisation du Code en Modules

- Séparer le code en plusieurs fichiers (`.c`) : Principes
- Utilisation des fichiers d'en-tête (`.h`) : Rôle d'interface
- Contenu typique d'un `.h` (Prototypes, Types, `extern`, `#define`)
- Contenu typique d'un `.c` (Implémentations, `static`)
- Notion de linkage (Interne `static` vs Externe)

18. -Compilation et Édition de Liens

- Processus : Préprocessing -> Compilation -> Assemblage -> Édition de liens
- Compilation séparée (`gcc -c`) : Fichiers objets (`.o`)
- Édition de liens (`gcc ... *.o -o prog`) : Résolution des symboles
- **Bibliothèques Statiques et Dynamiques (Concept)**
- **Utilisation des bibliothèques :**
  - Bibliothèque standard C (implicite)
  - Bibliothèque Mathématique (`-lm`)
  - **Bibliothèques externes (Ex: `-lSDL2`)**
- **Options du compilateur (`-I`, `-L`, `-D`, `-W...`, `-g`, `-O`)**

19. -Systèmes de Build : Make

- Pourquoi utiliser un système de build ?
- Introduction à `Make`
- Syntaxe de base d'un `Makefile` (Règles, Cibles, Dépendances, Commandes)
- Variables dans Make
- Gérer les dépendances et la compilation multi-fichiers avec Make

20. -Gestion des Fichiers (E/S Fichiers)

- Concept de Flux (`FILE *`)
- Ouverture (`fopen`, Modes texte/binaire, Vérification `NULL`)
- Fermeture (`fclose`, Importance du flush)
- **E/S Texte :** `fprintf`, `fscanf`, `fgets`, `fputs`, `fgetc`, `fputc`
- **E/S Binaire :** `fread`, `fwrite` (Écriture/Lecture de blocs, structures)
- Positionnement dans un fichier (`fseek`, `ftell`, `rewind`)
- Gestion des erreurs de fichiers (`feof`, `ferror`, `perror`, `errno`)

Partie 4 : Programmation Graphique avec SDL2

21. -Introduction à SDL2

- Qu'est-ce que SDL ? (Simple DirectMedia Layer)
- Installation de SDL2, SDL2_image, SDL2_ttf
- Configuration du projet (Includes et Linker flags)

22. -Initialisation et Fenêtrage

- Initialisation de SDL (`SDL_Init`) et des sous-systèmes
- Création d'une fenêtre (`SDL_CreateWindow`)
- Création d'un Rendu (`SDL_CreateRenderer`)
- Gestion des erreurs SDL (`SDL_GetError`)
- Fermeture propre (`SDL_Destroy...`, `SDL_Quit`)

23. -Affichage et Images

- Le Rendu (Renderer) : Nettoyage (`SDL_RenderClear`), Présentation (`SDL_RenderPresent`)
- Couleurs (`SDL_SetRenderDrawColor`, `SDL_Color`)
- Dessin de formes simples (`SDL_RenderDrawPoint`, `Line`, `Rect`)
- Chargement d'images avec `SDL_image` (`IMG_Load`, `IMG_Init`)
- Surfaces (`SDL_Surface`) et Textures (`SDL_Texture`)
- Création de textures depuis des surfaces (`SDL_CreateTextureFromSurface`)
- Affichage de textures (`SDL_RenderCopy`, `SDL_Rect`)
- Libération des surfaces et textures (`SDL_FreeSurface`, `SDL_DestroyTexture`)
- Transparence et Modes de fusion (`SDL_SetTextureBlendMode`)

24. -Gestion des Événements

- La boucle d'événements (`SDL_PollEvent`)
- Types d'événements (`SDL_Event`, `e.type`)
- Événement de fermeture (`SDL_QUIT`)
- Événements Clavier (`SDL_KEYDOWN`, `SDL_KEYUP`, `e.key.keysym.sym`)
- Événements Souris (`SDL_MOUSEBUTTONDOWN`, `UP`, `MOTION`, `WHEEL`, `e.button.x/y`, `e.wheel.y`)
- Gestion de la saisie de texte (`SDL_StartTextInput`, `SDL_StopTextInput`, `SDL_TEXTINPUT`)

25. -Affichage de Texte avec SDL_ttf

- Initialisation de `SDL_ttf` (`TTF_Init`)
- Chargement d'une police (`TTF_OpenFont`)
- Rendu de texte sur une surface (`TTF_RenderText_Solid`, `_Shaded`, `_Blended`)
- Création d'une texture depuis la surface de texte
- Affichage de la texture de texte
- Fermeture de la police et de `SDL_ttf` (`TTF_CloseFont`, `TTF_Quit`)

Partie 5 : Techniques Avancées et Parallélisme

26. -Programmation Parallèle avec OpenMP

- Introduction au parallélisme (Threads)
- Concepts d'OpenMP (Approche basée sur les directives)
- Configuration du compilateur (`-fopenmp`)
- Directive `#pragma omp parallel for` (Parallélisation de boucles)
- Clauses `private`, `shared`, `reduction` (Gestion des variables)
- Sections critiques (`#pragma omp critical`)
- (Optionnel) Autres directives (`parallel`, `sections`, `single`, `task`...)

27. -Techniques de Débogage et Analyse

- Utilisation avancée d'un débogueur (GDB/LLDB) : Points d'arrêt conditionnels, Watchpoints, Examen mémoire...
- **Débogage mémoire avec Valgrind** (Détection de fuites, accès invalides...)
- Analyseurs statiques (Ex: `clang --analyze`, `cppcheck`)
- Profiling (Mesurer les performances, ex: `gprof`) (Mention)

Partie 6 : Application et Conclusion

28. -Bonnes Pratiques et Style de Code

- Revue approfondie : Lisibilité, Modularité, Robustesse
- Gestion des erreurs avancée (Stratégies, Codes de retour vs `errno`)
- Documentation du code (Commentaires, Doxygen...)

29. -Conclusion et Prochaines Étapes

- Récapitulatif des compétences acquises
- Aller plus loin : Optimisation, Programmation réseau, Autres bibliothèques C, Contribution Open Source...

Cours C Complet (Focus Projet Échecs)

Bienvenue dans ce cours complet sur le langage C ! Ce langage, bien que plus ancien que beaucoup d'autres, reste extrêmement pertinent aujourd'hui, notamment dans la programmation système, les systèmes embarqués, les jeux vidéo et les domaines où la performance est critique. Ce cours vise à vous fournir les bases solides nécessaires pour comprendre et développer des applications C, y compris des projets complexes comme le jeu d'échecs que nous avons analysé.

---

1. Introduction au C

Qu'est-ce que le C ?

Le C est un langage de programmation **impératif**, **procédural** et de **bas niveau** (proche de la machine), développé dans les années 1970 par Dennis Ritchie aux Bell Labs. Il a été initialement conçu pour développer le système d'exploitation UNIX. Sa conception met l'accent sur l'efficacité, la portabilité et le contrôle direct de la mémoire et du matériel.

Pourquoi apprendre le C ?

- **Performance :** Le C permet de générer du code très rapide et efficace, car il est compilé directement en langage machine avec peu de surcoût ("overhead").
- **Contrôle :** Il offre un contrôle fin sur la gestion de la mémoire (pointeurs) et l'accès direct au matériel, ce qui est essentiel pour la programmation système ou embarquée.
- **Portabilité :** Un code C bien écrit, respectant les standards, peut souvent être compilé et exécuté sur différentes architectures et systèmes d'exploitation avec peu ou pas de modifications.
- **Fondation :** Comprendre le C aide à mieux saisir le fonctionnement interne d'un ordinateur (mémoire, processeur) et facilite grandement l'apprentissage d'autres langages influents qui en dérivent (C++, C#, Objective-C, Java, etc.).
- **Ubiquité :** Le C est utilisé partout :
  - Systèmes d'exploitation (Linux, Windows, macOS, *BSD...)
  - Compilateurs et interpréteurs pour d'autres langages
  - Bases de données
  - Moteurs de jeux vidéo
  - Systèmes embarqués (microcontrôleurs, automobile, avionique...)
  - Calcul scientifique et haute performance.

Installation de l'environnement (Compilateur)

Pour écrire, compiler et exécuter du code C, vous avez besoin principalement de deux choses :

1. **Un éditeur de texte ou un IDE (Environnement de Développement Intégré) :** Pour écrire votre code source (`.c` et `.h`).
   - *Éditeurs simples :* Notepad++, Sublime Text, VS Code (avec extensions C/C++), Vim, Emacs...
   - *IDE :* Code::Blocks, CLion, Visual Studio (avec le workload C++), Xcode (macOS)... Les IDE intègrent souvent un éditeur, un compilateur, un débogueur et des outils de gestion de projet.
2. **Un compilateur C :** Pour traduire votre code source en code exécutable par la machine.
   - **GCC (GNU Compiler Collection) :** Le compilateur C/C++ le plus répandu, gratuit et open-source. C'est souvent le choix par défaut.
   - **Clang :** Un autre compilateur C/C++/Objective-C moderne, open-source (projet LLVM), connu pour ses messages d'erreur clairs. Souvent utilisé sur macOS.
   - **MSVC (Microsoft Visual C++) :** Le compilateur de Microsoft, intégré à Visual Studio (peut compiler du C).

Installation du compilateur :

- **Linux (Debian/Ubuntu) :**

```
sudo apt update
sudo apt install build-essential gdb
```

(`build-essential` installe GCC, Make, et d'autres outils essentiels; `gdb` est le débogueur GNU).

- **Linux (Fedora/CentOS/RHEL) :**

```
sudo yum groupinstall "Development Tools"
sudo yum install gdb
```

- **macOS :** Le plus simple est d'installer les "Command Line Tools" d'Apple. Ouvrez le Terminal et tapez :

```
xcode-select --install
```

Cela installera Clang, Git, et d'autres outils. Vous pouvez aussi installer Xcode complet depuis l'App Store.

- **Windows :** Plusieurs options :
  - **MinGW-w64 :** Fournit GCC et les outils GNU pour Windows. Peut être installé via MSYS2 (recommandé : https://www.msys2.org/) ou d'autres installeurs. Après installation de MSYS2, ouvrez son terminal et installez GCC : `pacman -S mingw-w64-x86_64-gcc mingw-w64-x86_64-gdb make`.
  - **Visual Studio :** Installez Visual Studio Community (gratuit) avec le workload "Développement Desktop en C++". Il inclut le compilateur MSVC et un excellent IDE/débogueur.
  - **WSL (Windows Subsystem for Linux) :** Permet d'installer une distribution Linux (comme Ubuntu) directement sous Windows et d'utiliser GCC/GDB comme sous Linux.

Pour vérifier votre installation, ouvrez un terminal (ou invite de commande, ou terminal MSYS2) et tapez :

```
gcc --version
# ou clang --version si vous utilisez Clang
```

Cela devrait afficher la version de votre compilateur.

---

2. Les Bases du Langage

Votre Premier Programme : "Hello, World!"

C'est une tradition incontournable en programmation ! Voici le programme C le plus simple qui affiche "Hello, World!" à l'écran. Copiez ce code dans votre éditeur et enregistrez-le sous le nom `hello.c`.

```c
// Inclut les informations sur la bibliothèque standard d'Entrée/Sortie
#include <stdio.h>

// La fonction 'main' est le point d'entrée obligatoire de tout programme C exécutable.
// 'int' signifie qu'elle retourne une valeur entière au système d'exploitation.
int main() {
    // 'printf' est une fonction de la bibliothèque <stdio.h>
    // Elle permet d'afficher du texte (formaté) sur la sortie standard (l'écran).
```

```c
    // La chaîne "Hello, World!\n" est l'argument passé à printf.
    // '\n' est un caractère spécial qui représente un saut de ligne.
    printf("Hello, World!\n");

    // 'return 0;' indique au système que le programme s'est terminé avec
succès.
    // Une valeur autre que 0 signale généralement une erreur.
    return 0;
}
```

Structure d'un programme C

Analysons les composants de ce premier programme :

- `#include <stdio.h>` **:** C'est une **directive de préprocesseur**. Elle demande au préprocesseur (une étape avant la compilation) d'inclure le contenu du fichier d'en-tête `stdio.h` (Standard Input/Output Header). Ce fichier contient les déclarations des fonctions standard d'entrée/sortie, comme `printf`.
- `int main() { ... }` **:** C'est la définition de la fonction `main`.
  - Tout programme C exécutable doit avoir une fonction `main`. C'est le **point d'entrée**, là où l'exécution du programme commence.
  - `int` : Indique que la fonction `main` retourne une valeur entière au système d'exploitation à la fin de son exécution.
  - `main` : Le nom obligatoire de la fonction principale.
  - `()` : Les parenthèses indiquent que cette fonction `main` ne prend pas d'arguments (pour l'instant).
  - `{ ... }` : Les accolades délimitent le **bloc de code** (le corps) de la fonction `main`. Toutes les instructions de la fonction se trouvent entre ces accolades.
- `printf("Hello, World!\n");` **:** C'est une **instruction**.
  - Elle appelle la fonction `printf` (déclarée dans `stdio.h`).
  - `"Hello, World!\n"` est une **chaîne de caractères littérale** passée comme argument à `printf`.
  - `\n` est une **séquence d'échappement** qui représente un caractère de saut de ligne.
  - Le point-virgule `;` termine l'instruction. Presque toutes les instructions en C se terminent par un point-virgule.
- `return 0;` **:** C'est une autre **instruction**.
  - Elle termine la fonction `main` et retourne la valeur `0` au système d'exploitation. Par convention, retourner `0` signifie que le programme s'est exécuté avec succès. D'autres valeurs (non nulles) indiquent généralement qu'une erreur s'est produite.
- **Commentaires :** Les lignes commençant par `//` et le texte entre `/* ... */` sont des commentaires, ignorés par le compilateur mais utiles pour les humains.

Compilation et Exécution

Pour transformer votre code source `hello.c` en un programme exécutable :

1. **Ouvrez un terminal** (ou invite de commandes, ou terminal MSYS2/Git Bash sous Windows).

2. **Naviguez** jusqu'au répertoire où vous avez enregistré `hello.c` en utilisant la commande `cd` (Change Directory). Par exemple : `cd Documents/CoursC`.
3. **Compilez** le code en utilisant GCC (ou Clang) :

```
gcc hello.c -o hello
```

```
* `gcc` : Invoque le compilateur GCC.
* `hello.c` : Spécifie le fichier source à compiler.
* `-o hello` : L'option `-o` permet de spécifier le nom du fichier
exécutable de sortie. Ici, il s'appellera `hello` (ou `hello.exe` sous
Windows). Sans cette option, le nom par défaut est souvent `a.out`.
Cette commande effectue en réalité la compilation *et* l'édition de liens
en une seule étape.
```

4. **Exécutez** le programme nouvellement créé :
   - Sur Linux/macOS :

     ```
     ./hello
     ```

   - Sur Windows (Invite de Commandes classique) :

     ```
     hello.exe
     ```

     ou simplement

     ```
     hello
     ```

   - Sur Windows (PowerShell ou terminal moderne) :

     ```
     .\hello.exe
     ```

     ou

     ```
     .\hello
     ```

   Le préfixe `./` ou `.\` est nécessaire pour indiquer au système d'exécuter le programme situé dans le répertoire *courant*.

Vous devriez voir le texte `Hello, World!` s'afficher sur votre terminal.

Commentaires

Les commentaires sont cruciaux pour expliquer votre code, pour vous-même ou pour d'autres développeurs. Ils sont complètement ignorés par le compilateur.

- **Commentaire sur une seule ligne :** Commence par `//` et va jusqu'à la fin de la ligne. C'est le style le plus courant pour les commentaires courts.
- **Commentaire multi-lignes :** Commence par `/*` et se termine par `*/`. Peut s'étendre sur plusieurs lignes. Utile pour désactiver temporairement des blocs de code ou pour des explications plus longues.

```c
#include <stdio.h>

int main() { // Point d'entrée du programme

    // Affiche un message de bienvenue.
    // Ceci est une autre ligne de commentaire.
    printf("Bonjour !\n");

    /* Ceci est un commentaire
       sur plusieurs lignes.
       Il est utile pour des descriptions
       plus détaillées. */
    printf("Comment allez-vous ?\n");

    /* On peut aussi utiliser ce style pour commenter
       une partie d'une ligne, mais c'est moins courant :
    int x = 5; /* Initialisation de x */ // */

    // On peut aussi commenter temporairement du code :
    /*
    printf("Cette ligne ne sera pas compilée.\n");
    int temp = 10;
    */

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

Bonne pratique : Commentez pour clarifier l'intention ou la logique complexe, pas pour paraphraser le code évident. Des commentaires clairs et concis améliorent grandement la maintenabilité.

---

3. Variables et Types de Données

Qu'est-ce qu'une variable ? Mémoire et Adresses.

Imaginez la mémoire de votre ordinateur (RAM) comme une immense série de boîtes numérotées. Chaque boîte peut contenir une petite quantité d'information (un octet - 8 bits). Chaque boîte a un numéro unique, appelé son **adresse mémoire**.

Une **variable** en C est un **nom symbolique** que vous donnez à une ou plusieurs de ces boîtes mémoire pour y stocker une donnée. Lorsque vous déclarez une variable, vous dites au compilateur :

1. Quel **nom** vous allez utiliser pour faire référence à cette zone mémoire.
2. Quel **type** de donnée vous allez y stocker (ce qui détermine combien de boîtes/octets sont nécessaires et comment interpréter les bits contenus).

Le compilateur se charge alors de réserver l'espace mémoire nécessaire et d'associer le nom de la variable à l'adresse de début de cette zone mémoire. La valeur stockée dans cette zone peut généralement changer pendant l'exécution du programme (d'où le nom "variable").

Types de données de base

Le C fournit plusieurs types de données fondamentaux pour représenter différents genres d'informations :

- **`int` :** Pour les **nombres entiers** (positifs, négatifs ou zéro). Ex: `-10`, `0`, `42`. La taille exacte (et donc la plage de valeurs possibles) d'un `int` dépend du système et du compilateur (souvent 32 bits, soit 4 octets, allant d'environ -2 milliards à +2 milliards).
- **`float` :** Pour les **nombres à virgule flottante** (nombres réels) en **simple précision**. Utilise généralement 32 bits (4 octets) selon la norme IEEE 754. Offre environ 6 à 7 chiffres de précision décimale. Ex: `3.14f`, `-0.001f`, `1.2e5f` (le `f` final est recommandé pour indiquer un littéral `float`).
- **`double` :** Pour les nombres à virgule flottante en **double précision**. Utilise généralement 64 bits (8 octets) selon la norme IEEE 754. Offre environ 15 à 16 chiffres de précision décimale. C'est le type flottant le plus couramment utilisé pour les calculs, car il est plus précis que `float`. Ex: `3.1415926535`, `-2.718`, `6.022e23`.
- **`char` :** Pour représenter un **unique caractère**. En C, un `char` est en fait un petit type entier (généralement 8 bits, soit 1 octet), stockant la valeur numérique (souvent ASCII ou UTF-8) du caractère. Ex: `'A'`, `'b'`, `'!'`, `' '`, `'\n'` (caractère de nouvelle ligne). Notez les **apostrophes simples** (`' '`) pour les littéraux de type `char`.

Modificateurs de type (`short`, `long`, `signed`, `unsigned`)

Ces mots-clés peuvent être utilisés pour modifier les types entiers (`int`, `char`) et parfois `double` (`long double`) :

- **`short` et `long` :** Affectent la **taille** (et donc la plage de valeurs) des entiers.
  - `short int` (ou `short`) : Généralement plus petit ou égal à `int` (souvent 16 bits).
  - `long int` (ou `long`) : Généralement plus grand ou égal à `int` (souvent 32 ou 64 bits).
  - `long long int` (ou `long long`) (C99+) : Encore plus grand (généralement 64 bits).
  - `long double` : Peut offrir une précision supérieure à `double` (souvent 80 ou 128 bits).
- **`signed` et `unsigned` :** Affectent la manière dont le bit de poids fort est interprété pour les types entiers (`char`, `short`, `int`, `long`, `long long`).
  - `signed` (par défaut pour la plupart des types `int`) : La variable peut stocker des valeurs négatives, nulles ou positives. Le bit de poids fort est utilisé pour le signe.
  - `unsigned` : La variable ne peut stocker que des valeurs nulles ou positives. Le bit de signe est utilisé pour représenter des valeurs plus grandes, doublant ainsi la plage positive par rapport au type `signed` équivalent. Ex: `unsigned int`, `unsigned char`.

Note : La taille exacte de ces types (sauf `char` qui est toujours 1 octet par définition C) n'est pas fixée par le standard C mais dépend de l'implémentation (compilateur et architecture). Cependant, le standard garantit des relations de taille minimale (`sizeof(short) <= sizeof(int) <= sizeof(long) <= sizeof(long long)`).

Le type `bool` et `<stdbool.h>` (C99+)

Historiquement, le C n'avait pas de type booléen dédié. On utilisait des `int`, où `0` représentait "faux" et toute valeur non nulle représentait "vrai".

Depuis la norme **C99**, un vrai type booléen a été introduit. Pour l'utiliser :

1. Incluez l'en-tête `<stdbool.h>` : `#include <stdbool.h>`
2. Utilisez le type `bool`.
3. Utilisez les constantes `true` (qui vaut 1) et `false` (qui vaut 0).

C'est la manière moderne et recommandée de gérer les valeurs logiques en C.

```c
#include <stdio.h>
#include <stdbool.h> // Nécessaire pour bool, true, false

int main() {
    bool estActif = true;
    bool aReussi = false;
    int age = 20;

    if (estActif) {
        printf("Le système est actif.\n");
    }

    if (!aReussi) { // Utilisation de l'opérateur NON (!)
        printf("L'opération n'a pas réussi.\n");
    }

    // On peut toujours utiliser des entiers dans des contextes booléens
    if (age) { // age n'est pas 0, donc c'est vrai
        printf("L'âge est non nul.\n");
    }

    // Assignation du résultat d'une comparaison à un bool
    bool estMajeur = (age >= 18); // estMajeur vaudra true
    if (estMajeur) {
        printf("La personne est majeure.\n");
    }

    return 0;
}
```

Déclaration et Initialisation

Avant d'utiliser une variable, vous devez la **déclarer** pour spécifier son type et son nom. Vous pouvez (et devriez souvent) l'**initialiser** (lui donner une valeur de départ) en même temps.

Syntaxe de déclaration : `type nom_variable;` **Syntaxe de déclaration avec initialisation :** `type nom_variable = valeur_initiale;`

```c
#include <stdio.h>
#include <stdbool.h> // Pour bool

int main() {
    // Déclarations sans initialisation
    int compteur;          // La valeur de compteur est indéterminée
(locale)
    float temperature;

    // Déclarations AVEC initialisation
    int score = 0;
    double pi = 3.14159;
    char initiale = 'L';
    bool estPret = false;
    unsigned long population_mondiale = 8000000000UL; // UL pour Unsigned
Long

    // Assignation après déclaration
    compteur = 10;
    temperature = 25.5f; // 'f' pour float

    // Affichage
    printf("Compteur: %d\n", compteur);
    printf("Score: %d\n", score);
    printf("Température: %.1f C\n", temperature); // %.1f pour 1 décimale
    printf("Initiale: %c\n", initiale);
    printf("Pi: %f\n", pi); // %f fonctionne pour double aussi dans printf
    printf("Prêt ? %d\n", estPret); // Affiche 0 (false)
    printf("Population: %lu\n", population_mondiale); // %lu pour unsigned
long

    return 0;
}
```

Important : Les variables locales (déclarées à l'intérieur d'une fonction) non explicitement initialisées ont une valeur **indéterminée** ("garbage"). Utiliser une variable non initialisée conduit à un comportement indéfini. Les variables globales et statiques sont initialisées à zéro par défaut si aucune valeur n'est fournie. **Prenez l'habitude d'initialiser vos variables locales.**

Constantes (`const`)

Le mot-clé `const` peut être utilisé lors de la déclaration d'une variable pour indiquer que sa valeur **ne doit pas être modifiée** après son initialisation. Le compilateur essaiera d'empêcher toute tentative de modification ultérieure.

```c
#include <stdio.h>

int main() {
    const int SECONDES_PAR_MINUTE = 60;
    const float TAUX_TVA = 0.20f;
    const char LETTRE_DEBUT = 'A';

    printf("Secondes par minute: %d\n", SECONDES_PAR_MINUTE);
    printf("Taux TVA: %.2f\n", TAUX_TVA);

    // Tentatives de modification (généreront une erreur ou un
avertissement à la compilation) :
    // SECONDES_PAR_MINUTE = 61; // ERREUR
    // TAUX_TVA = 0.21f;         // ERREUR

    return 0;
}
```

Utiliser `const` améliore la lisibilité (on sait que la valeur ne changera pas) et la robustesse (le compilateur aide à prévenir les modifications accidentelles). Nous verrons plus tard comment `const` interagit avec les pointeurs.

---

4. Opérateurs

Les opérateurs sont des symboles spéciaux utilisés pour effectuer des opérations sur des variables et des valeurs (appelées **opérandes**). Le C offre une riche variété d'opérateurs pour les calculs, les comparaisons, les manipulations logiques et binaires, etc.

Opérateurs Arithmétiques

Ces opérateurs effectuent des calculs mathématiques courants :

- `+` : Addition
- `-` : Soustraction (peut aussi être unaire pour indiquer un nombre négatif, ex: `-5`)
- `*` : Multiplication
- `/` : Division
  - **Attention :** Si les *deux* opérandes sont des entiers, le résultat est une **division entière** (la partie décimale est tronquée). Ex: `7 / 2` donne `3`.
  - Pour obtenir une division flottante, au moins un des opérandes doit être de type `float` ou `double`. Ex: `7.0 / 2`, `7 / 2.0`, `(float)7 / 2` donnent tous `3.5`.
- `%` : Modulo (reste de la division entière). Ne fonctionne qu'avec des opérandes entiers. Ex: `7 % 2` donne `1`.

```c
#include <stdio.h>

int main() {
    int a = 10, b = 3;
```

```
    float x = 10.0f, y = 3.0f;

    printf("a + b = %d\n", a + b);       // 13
    printf("a - b = %d\n", a - b);       // 7
    printf("a * b = %d\n", a * b);       // 30
    printf("a / b = %d\n", a / b);       // 3 (Division entière)
    printf("a %% b = %d\n", a % b);      // 1 (Reste)

    printf("x / y = %f\n", x / y);       // 3.333333 (Division flottante)
    printf("x / b = %f\n", x / b);       // 3.333333 (Promotion de b en
float)
    printf("(float)a / b = %f\n", (float)a / b); // 3.333333 (Cast
explicite)

    // Opérateur unaire -
    int neg_a = -a;
    printf("Négation de a : %d\n", neg_a); // -10

    return 0;
}
```

Opérateurs Relationnels

Comparent deux valeurs et retournent un résultat booléen (en C, `1` pour vrai, `0` pour faux). Essentiels pour les conditions `if`, `while`, etc.

- `==` : Égal à
- `!=` : Différent de
- `<` : Inférieur à
- `>` : Supérieur à
- `<=` : Inférieur ou égal à
- `>=` : Supérieur ou égal à

```
#include <stdio.h>
#include <stdbool.h> // Pour utiliser bool, true, false

int main() {
    int note = 12;
    int moyenne = 10;

    bool est_egal = (note == moyenne); // false (0)
    bool est_diff = (note != moyenne); // true (1)
    bool est_sup = (note > moyenne);   // true (1)
    bool est_inf_egal = (note <= moyenne); // false (0)

    // On peut afficher les booléens comme des entiers avec %d
    printf("note == moyenne : %d\n", est_egal);
    printf("note != moyenne : %d\n", est_diff);
    printf("note > moyenne  : %d\n", est_sup);
    printf("note <= moyenne : %d\n", est_inf_egal);
```

```
    // Utilisation directe dans un if
    if (note >= moyenne) {
        printf("La note est supérieure ou égale à la moyenne.\n");
    }

    return 0;
}
```

Opérateurs Logiques

Combinent des expressions booléennes (ou des valeurs où 0 est faux et non-zéro est vrai) :

- `&&` : **ET logique** (vrai si les *deux* opérandes sont vrais/non-nuls).
- `||` : **OU logique** (vrai si *au moins un* des opérandes est vrai/non-nul).
- `!` : **NON logique** (unaire : inverse la valeur de vérité de l'opérande).

Évaluation court-circuit (Short-circuiting) :

- Pour `A && B`, si `A` est faux (0), `B` n'est **pas évalué** car le résultat est forcément faux.
- Pour `A || B`, si `A` est vrai (non-nul), `B` n'est **pas évalué** car le résultat est forcément vrai. C'est important si `B` a des effets de bord (comme `x++`).

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

int main() {
    int age = 25;
    bool a_le_permis = true;
    bool est_etudiant = false;

    // Peut conduire ET est majeur ?
    bool peut_conduire = a_le_permis && (age >= 18);
    printf("Peut conduire ? %d\n", peut_conduire); // 1 (true)

    // Est étudiant OU a moins de 20 ans ?
    bool tarif_reduit = est_etudiant || (age < 20);
    printf("Tarif réduit ? %d\n", tarif_reduit); // 0 (false)

    // N'est PAS étudiant ?
    bool non_etudiant = !est_etudiant;
    printf("N'est pas étudiant ? %d\n", non_etudiant); // 1 (true)

    // Exemple court-circuit
    int x = 5;
    printf("\nTest court-circuit :\nx = %d\n", x);
    // Si x est 0, x++ ne serait pas évalué grâce au court-circuit de &&
    bool condition_et = (x != 0) && (x++ > 0); // x++ est évalué ici
    printf("Condition ET: %d, x après évaluation: %d\n", condition_et, x);
// Affiche 1, 6
```

```
    x = 0;
    printf("x = %d\n", x);
    // Si x est 0, x++ n'est pas évalué
    condition_et = (x != 0) && (x++ > 0);
    printf("Condition ET: %d, x après évaluation: %d\n", condition_et, x);
// Affiche 0, 0

    x = 5;
    printf("x = %d\n", x);
    // Si x != 5 est vrai, x++ n'est pas évalué grâce au court-circuit de
||
    bool condition_ou = (x == 5) || (x++ > 10); // x++ n'est pas évalué ici
    printf("Condition OU: %d, x après évaluation: %d\n", condition_ou, x);
// Affiche 1, 5

    return 0;
}
```

Opérateurs Bitwise (sur les bits)

Ces opérateurs travaillent directement sur la représentation binaire des nombres entiers. Ils sont souvent utilisés en programmation système, embarquée, pour la manipulation de flags, les masques, ou pour des optimisations spécifiques.

- `&` : **ET bit-à-bit** (AND) : Met un bit à 1 dans le résultat si les bits correspondants dans les deux opérandes sont à 1.
- `|` : **OU bit-à-bit** (OR) : Met un bit à 1 si le bit correspondant dans *au moins un* des opérandes est à 1.
- `^` : **OU exclusif bit-à-bit** (XOR) : Met un bit à 1 si les bits correspondants sont *différents*.
- `~` : **Complément à un bit-à-bit** (NOT) (unaire) : Inverse chaque bit de l'opérande (0 devient 1, 1 devient 0).
- `<<` : **Décalage à gauche** : Décale les bits de l'opérande de gauche vers la gauche du nombre de positions spécifié par l'opérande de droite (remplit avec des zéros à droite). Équivaut à une multiplication par 2^n (où n est le nombre de positions).
- `>>` : **Décalage à droite** : Décale les bits vers la droite. Pour les nombres *non signés*, remplit avec des zéros à gauche (décalage logique). Pour les nombres *signés*, le comportement peut dépendre de l'implémentation (décalage arithmétique - copie du bit de signe, ou logique - remplissage par zéro). Équivaut à une division entière par 2^n.

```
#include <stdio.h>

int main() {
    unsigned char a = 5;  // Binaire: 00000101
    unsigned char b = 3;  // Binaire: 00000011

    printf("a = %u (0x%02X), b = %u (0x%02X)\n", a, a, b, b);

    unsigned char et_result = a & b; // 00000101 & 00000011 = 00000001 (1)
    printf("a & b = %u (0x%02X)\n", et_result, et_result);
```

```c
    unsigned char ou_result = a | b; // 00000101 | 00000011 = 00000111 (7)
    printf("a | b = %u (0x%02X)\n", ou_result, ou_result);

    unsigned char xor_result = a ^ b; // 00000101 ^ 00000011 = 00000110 (6)
    printf("a ^ b = %u (0x%02X)\n", xor_result, xor_result);

    unsigned char not_a = ~a; // ~00000101 = 11111010 (250 si unsigned char
8 bits)
    printf("~a    = %u (0x%02X)\n", not_a, not_a);

    unsigned char decal_gauche = a << 2; // 00000101 << 2 = 00010100 (20)
    printf("a << 2 = %u (0x%02X)\n", decal_gauche, decal_gauche);

    unsigned char decal_droite = a >> 1; // 00000101 >> 1 = 00000010 (2)
    printf("a >> 1 = %u (0x%02X)\n", decal_droite, decal_droite);

    // Exemple avec flags
    unsigned char READ_PERMISSION = 1 << 0; // 00000001
    unsigned char WRITE_PERMISSION = 1 << 1; // 00000010
    unsigned char EXEC_PERMISSION = 1 << 2; // 00000100

    unsigned char my_permissions = READ_PERMISSION | WRITE_PERMISSION; //
00000011
    printf("\nPermissions initiales: %u\n", my_permissions);

    // Vérifier si on a la permission de lecture
    if (my_permissions & READ_PERMISSION) {
        printf("Permission de lecture accordée.\n");
    }

    // Ajouter la permission d'exécution
    my_permissions |= EXEC_PERMISSION; // 00000111
    printf("Permissions après ajout EXEC: %u\n", my_permissions);

    // Enlever la permission d'écriture (en utilisant AND avec le
complément)
    my_permissions &= ~WRITE_PERMISSION; // 00000111 & 11111101 = 00000101
    printf("Permissions après retrait WRITE: %u\n", my_permissions);

    return 0;
}
```

Opérateurs d'Assignation

- `=` : Assignation simple : affecte la valeur de droite à la variable de gauche.
- Opérateurs d'assignation combinée : Combinaison d'une opération et d'une assignation.
  - Arithmétiques : `+=`, `-=`, `*=`, `/=`, `%=`
  - Bitwise : `&=`, `|=`, `^=`, `<<=`, `>>=`
  - Exemple : `x += 5;` est équivalent à `x = x + 5;`.
  - Exemple : `flags |= MASQUE;` est équivalent à `flags = flags | MASQUE;`.

```c
#include <stdio.h>

int main() {
    int solde = 1000;
    int retrait = 150;
    solde -= retrait; // Solde = 1000 - 150 = 850
    printf("Nouveau solde : %d\n", solde);

    int multi = 5;
    multi *= 3; // multi = 5 * 3 = 15
    printf("Multi : %d\n", multi);

    unsigned char options = 0b10101010;
    unsigned char masque = 0b00001111;
    options &= masque; // options = options & masque = 0b00001010 (10)
    printf("Options après masque ET : %u\n", options);

    int decal = 4;
    decal <<= 2; // decal = decal << 2 = 4 * 4 = 16
    printf("Decal après <<= 2 : %d\n", decal);

    return 0;
}
```

Opérateurs d'Incrémentation et de Décrémentation

Raccourcis pour ajouter ou soustraire 1.

- `++` : Incrémentation
- `--` : Décrémentation

Peuvent être utilisés en **préfixe** (`++variable`) ou **postfixe** (`variable++`).

- **Préfixe :** La variable est modifiée *avant* que sa valeur ne soit utilisée dans l'expression.
- **Postfixe :** La valeur *originale* de la variable est utilisée dans l'expression, *puis* la variable est modifiée.

```c
#include <stdio.h>

int main() {
    int compteur = 5;
    int valeur;

    // Post-incrémentation
    valeur = compteur++; // 1. valeur prend 5 (ancienne valeur de compteur)
                         // 2. compteur devient 6
    printf("Post-incrémentation: valeur = %d, compteur = %d\n", valeur,
compteur);

    compteur = 5; // Réinitialisation

    // Pré-incrémentation
```

```
    valeur = ++compteur; // 1. compteur devient 6
                         // 2. valeur prend 6 (nouvelle valeur de compteur)
    printf("Pré-incrémentation:  valeur = %d, compteur = %d\n", valeur,
compteur);

    // Idem pour --
    compteur = 5;
    valeur = compteur--; // valeur = 5, compteur = 4
    printf("Post-décrémentation: valeur = %d, compteur = %d\n", valeur,
compteur);

    compteur = 5;
    valeur = --compteur; // compteur = 4, valeur = 4
    printf("Pré-décrémentation:  valeur = %d, compteur = %d\n", valeur,
compteur);

    return 0;
}
```

Opérateur Ternaire (`?:`)

Une forme concise de `if...else` pour choisir entre deux expressions.

Syntaxe : `condition ? expression_si_vrai : expression_si_faux`

```
#include <stdio.h>

int main() {
    int a = 10, b = 20;
    int max = (a > b) ? a : b; // max prend la valeur de a si a > b, sinon
b
    printf("Le max est %d\n", max); // Affiche 20

    int score = 8;
    printf("Statut : %s\n", (score >= 10) ? "Réussi" : "Échoué"); //
Affiche Échoué

    return 0;
}
```

Opérateur `sizeof`

Retourne la taille en **octets** d'un type de données ou d'une variable. Très utile, notamment pour l'allocation de mémoire et la portabilité.

Le type du résultat est `size_t` (un type entier non signé défini dans `<stddef.h>`). Utilisez le spécificateur de format `%zu` (ou `%lu` sur certains systèmes plus anciens) pour l'afficher avec `printf`.

```c
#include <stdio.h>
#include <stddef.h> // Pour size_t

int main() {
    int i;
    double d;
    char tab[20];

    printf("Taille d'un int    : %zu octets\n", sizeof(int));
    printf("Taille d'un double : %zu octets\n", sizeof(double));
    printf("Taille de la var i : %zu octets\n", sizeof(i));
    printf("Taille de la var d : %zu octets\n", sizeof(d));
    printf("Taille du tableau tab[20] : %zu octets\n", sizeof(tab));
    printf("Taille de l'expression (i+d) : %zu octets\n", sizeof(i + d));
// Type promu en double

    // Calculer le nombre d'éléments d'un tableau
    size_t nb_elements = sizeof(tab) / sizeof(tab[0]); // Taille totale /
taille d'un élément
    printf("Nombre d'éléments dans tab : %zu\n", nb_elements);

    return 0;
}
```

Priorité et Associativité des opérateurs

Lorsque plusieurs opérateurs apparaissent dans une expression sans parenthèses, leur **priorité** détermine l'ordre d'évaluation (ex: `*` et `/` avant `+` et `-`). Si des opérateurs ont la même priorité, leur **associativité** (gauche à droite ou droite à gauche) détermine l'ordre.

Par exemple, `a = b = c;` est évalué de droite à gauche (`a = (b = c);`). Tandis que `x / y * z;` est évalué de gauche à droite (`(x / y) * z;`).

Consultez une table de priorité des opérateurs C si nécessaire.

Règle d'or : En cas de doute, ou pour rendre le code plus clair, **utilisez des parenthèses `( )`** pour forcer explicitement l'ordre d'évaluation souhaité.

---

5. Entrées et Sorties Console

Pour qu'un programme soit utile, il doit généralement interagir avec l'extérieur : afficher des résultats ou lire des données entrées par l'utilisateur. En C, ces opérations sont principalement gérées par les fonctions définies dans la bibliothèque standard d'Entrée/Sortie (`stdio.h`). Assurez-vous d'inclure cet en-tête au début de vos fichiers source qui utilisent ces fonctions : `#include <stdio.h>`.

Flux Standards (`stdin`, `stdout`, `stderr`)

Quand un programme C démarre, trois **flux** (streams) de communication textuels sont généralement ouverts automatiquement par le système d'exploitation :

- `stdin` **(Standard Input) :** Le flux d'entrée standard, par défaut connecté au **clavier**. C'est la source principale pour lire les données tapées par l'utilisateur (avec `scanf`, `getchar`, `fgets`...).
- `stdout` **(Standard Output) :** Le flux de sortie standard, par défaut connecté à l'**écran** (le terminal ou la console). C'est là que la plupart des résultats normaux du programme sont affichés (avec `printf`, `putchar`, `puts`...).
- `stderr` **(Standard Error) :** Le flux d'erreur standard, par défaut aussi connecté à l'**écran**. Il est spécifiquement destiné à l'affichage des messages d'erreur ou de diagnostic. L'avantage de séparer `stderr` de `stdout` est que l'on peut rediriger la sortie normale vers un fichier (`./mon_prog > sortie.txt`) tout en voyant les messages d'erreur s'afficher sur le terminal. On utilise souvent `fprintf(stderr, ...)` pour écrire sur ce flux.

Affichage formaté avec `printf()`

La fonction `printf` (qui signifie "print formatted") est la fonction la plus couramment utilisée pour afficher du texte et des valeurs de variables formatées sur `stdout`.

Syntaxe : `int printf(const char *format, ...);`

- `format`: La chaîne de caractères à afficher. Elle peut contenir du texte ordinaire et des **spécificateurs de format** (commençant par `%`).
- `...`: Une liste d'arguments (variables, constantes, expressions) dont les valeurs remplaceront les spécificateurs de format correspondants dans la chaîne `format`, dans l'ordre où ils apparaissent.
- **Valeur de retour :** Le nombre total de caractères écrits en cas de succès, ou une valeur négative en cas d'erreur. (On vérifie rarement cette valeur pour `printf`).

Sa forme la plus simple affiche une chaîne de caractères littérale :

```c
#include <stdio.h>

int main() {
    printf("Bonjour le monde !"); // Affiche le texte
    printf("Ceci est une autre ligne."); // S'affiche juste après, sans
retour à la ligne
    printf("\nUtilisation de \\n pour un\nsaut de ligne.\n"); // \n force
un retour à la ligne
    printf("Une\ttabulation ici.\n"); // \t insère une tabulation
    return 0;
}
```

La véritable puissance de `printf` réside dans sa capacité à insérer des valeurs de variables dans la chaîne affichée en utilisant des **spécificateurs de format**.

Format Specifiers (`%d`, `%f`, `%c`, `%s`, etc.)

Un spécificateur de format commence par `%` et indique le type de la donnée à afficher à cet endroit. Voici les plus courants :

- `%d` ou `%i` : Affiche un entier signé (`int`) en base 10 (décimal).
- `%u` : Affiche un entier non signé (`unsigned int`) en base 10.

- `%ld`, `%li` : Affiche un `long int`.
- `%lu` : Affiche un `unsigned long int`.
- `%lld`, `%lli` (C99+) : Affiche un `long long int`.
- `%llu` (C99+) : Affiche un `unsigned long long int`.
- `%f` : Affiche un nombre à virgule flottante (`float` ou `double`). Par défaut, avec 6 chiffres après la virgule.
- `%lf` : Bien qu'essentiel pour `scanf` avec `double`, pour `printf`, `%f` fonctionne aussi bien pour `float` que pour `double` (car les `float` sont automatiquement promus en `double` lorsqu'ils sont passés en argument variadique à `printf`). Utiliser `%lf` dans `printf` est accepté mais techniquement redondant avec `%f`.
- `%e` ou `%E` : Affiche un flottant en notation scientifique (ex: `3.14e+02`).
- `%g` ou `%G` : Affiche un flottant soit en notation standard (`%f`), soit en notation scientifique (`%e`), selon ce qui est le plus court. Très pratique.
- `%c` : Affiche un unique caractère (`char`).
- `%s` : Affiche une chaîne de caractères (attend un `char *` pointant vers une chaîne terminée par `\0`).
- `%p` : Affiche l'adresse mémoire d'un pointeur (la représentation exacte dépend du système, souvent hexadécimale). Il est recommandé de caster le pointeur en `(void*)`.
- `%%` : Affiche le caractère `%` lui-même.

Options de formatage (entre `%` et le type) :

- **Largeur :** Un nombre entier spécifie la largeur minimale du champ. L'affichage est complété par des espaces (par défaut à droite). Ex: `%10d` affiche un entier sur au moins 10 caractères. Un `0` devant indique de compléter avec des zéros : `%05d`.
- **Précision :** Un point `.` suivi d'un nombre entier. Pour les flottants (`%f`, `%e`, `%g`), spécifie le nombre de chiffres *après* la virgule. Pour les entiers, la précision minimale du nombre de chiffres (complété par des zéros). Pour les chaînes (`%s`), le nombre *maximum* de caractères à afficher. Ex: `%.2f`, `%.8s`.
- **Flags :** `-` (alignement à gauche), `+` (affiche toujours le signe +/-), (espace si positif, signe si négatif).

Exemple d'utilisation de `printf` avec des variables et du formatage :

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

int main() {
    int age = 30;
    float taille = 1.75f; // f pour float
    char initiale = 'G';
    double pi = 3.1415926535;
    char nom[] = "Gandalf";
    int *ptr_age = &age; // Pointeur vers age

    printf("--- Informations ---\n");
    printf("Nom: %s\n", nom);
    printf("Initiale: %c\n", initiale);
    printf("Age: %d ans\n", age);
    printf("Taille: %f metres\n", taille);
    printf("Taille (2 décimales): %.2f metres\n", taille);
    printf("Age sur 5 espaces: [%5d]\n", age);
```

```c
    printf("Age sur 5 zéros: [%05d]\n", age);
    printf("Age aligné à gauche: [%-5d]\n", age);

    printf("\n--- Valeurs diverses ---\n");
    printf("Pi (double, %%f): %f\n", pi);
    printf("Pi (double, %%lf): %lf\n", pi); // Fonctionne aussi
    printf("Pi (double, 4 déc.): %.4f\n", pi);
    printf("Pi (scientifique): %e\n", pi);
    printf("Pi (court): %g\n", pi);
    printf("Adresse de age: %p\n", (void*)ptr_age); // Cast en (void*) pour
%p
    printf("Afficher un %% : 50%%\n"); // %% pour afficher %

    return 0;
}
```

Lecture formatée avec `scanf()`

La fonction `scanf` (qui signifie "scan formatted") est l'équivalent de `printf` pour l'entrée depuis `stdin`. Elle tente de lire des caractères depuis l'entrée et de les convertir selon les spécificateurs de format, puis stocke les résultats aux adresses mémoire fournies.

Syntaxe : `int scanf(const char *format, ...);`

- `format`: Chaîne indiquant les types de données attendus (`%d`, `%f`, `%lf`, `%c`, `%s`, etc.).
- `...`: **Adresses** des variables (obtenues avec `&`) où stocker les valeurs lues. **C'est la source d'erreur la plus fréquente : oublier le `&` !** (Exception : pour lire une chaîne avec `%s` dans un tableau de `char`, on passe le nom du tableau qui est déjà une adresse).
- **Valeur de retour :** Le nombre d'éléments correctement lus et assignés. `0` si l'entrée ne correspondait pas dès le début. `EOF` si la fin de l'entrée est atteinte avant toute lecture ou en cas d'erreur. **Vérifier cette valeur est essentiel.**

Spécificateurs importants pour `scanf` :

- `%d`, `%i` : Lire un `int`.
- `%u` : Lire un `unsigned int`.
- `%f` : **Lire un `float`**.
- `%lf` : **Lire un `double`** (le `l` est **obligatoire** ici, contrairement à `printf`).
- `%c` : Lire un seul `char`. **Attention :** lit *n'importe quel* caractère, y compris les espaces et les `\n` laissés par des lectures précédentes. Pour ignorer les espaces blancs précédents, utilisez un espace dans le format : `" %c"`.
- `%s` : Lire une chaîne de caractères (jusqu'au prochain espace blanc : espace, tabulation, nouvelle ligne). **Très dangereux car ne limite pas la taille lue !** Peut facilement causer un **dépassement de tampon (buffer overflow)** si l'entrée est plus longue que le tableau prévu. **Préférez `fgets` pour lire des chaînes.**

Fonctionnement de `scanf` :

- Elle essaie de faire correspondre les spécificateurs de format avec l'entrée.

- Les espaces blancs dans la chaîne de format consomment zéro ou plusieurs espaces blancs dans l'entrée.
- Pour `%d`, `%f`, etc., elle saute les espaces blancs initiaux avant de lire la donnée.
- Pour `%s`, elle saute les espaces blancs initiaux, puis lit les caractères non blancs jusqu'au prochain espace blanc.
- Pour `%c`, elle lit le *prochain caractère disponible*, quel qu'il soit (même un espace ou `\n`).
- Elle s'arrête dès qu'une donnée ne correspond pas au format, ou à la fin de la chaîne de format, ou si une erreur de lecture se produit.

```c
#include <stdio.h>

int main() {
    int age;
    float taille;
    double poids;
    char initiale;
    char mot[20]; // Tableau pour stocker un mot (max 19 chars + \0)

    printf("Entrez votre âge (entier) : ");
    // Vérifier si scanf a réussi à lire 1 élément
    if (scanf("%d", &age) != 1) {
        printf("Erreur de saisie de l'âge.\n");
        return 1;
    }

    printf("Entrez votre taille (ex: 1.75) : ");
    // Utiliser %f pour lire un float
    if (scanf("%f", &taille) != 1) {
        printf("Erreur de saisie de la taille.\n");
        return 1;
    }

    printf("Entrez votre poids (ex: 68.5) : ");
    // Utiliser %lf pour lire un double
    if (scanf("%lf", &poids) != 1) {
        printf("Erreur de saisie du poids.\n");
        return 1;
    }

    // Vider le buffer d'entrée (absorber le \n laissé par scanf)
    while (getchar() != '\n');

    printf("Entrez la première lettre de votre prénom : ");
    // L'espace avant %c est important pour ignorer le \n précédent
    if (scanf(" %c", &initiale) != 1) { // Notez l'espace avant %c
        printf("Erreur de saisie de l'initiale.\n");
        return 1;
    }

    printf("Entrez un mot (sans espace) : ");
    // %19s lit au maximum 19 caractères pour éviter le débordement de
```

```
    'mot'
        if (scanf("%19s", mot) != 1) { // Pas de & car 'mot' est déjà une
    adresse (nom du tableau)
            printf("Erreur de saisie du mot.\n");
            return 1;
        }

        // Afficher les valeurs lues
        printf("\n--- Récapitulatif ---\n");
        printf("Age entré : %d\n", age);
        printf("Taille entrée : %.2f\n", taille);
        printf("Poids entré : %.2lf\n", poids);
        printf("Initiale entrée : %c\n", initiale);
        printf("Mot entré : %s\n", mot);

        return 0;
    }
```

Dangers et Alternatives à `scanf` :

- **Robustesse :** `scanf` est très fragile. Si l'utilisateur tape du texte quand un nombre est attendu, la lecture échoue et l'entrée invalide reste souvent dans le buffer, perturbant les lectures suivantes.
- **Buffer Overflows avec `%s` :** Comme mentionné, utiliser `%s` sans spécifier une largeur maximale (`%19s` pour un tableau de 20 `char`) est une faille de sécurité majeure.
- **Gestion du `\n` :** `scanf` laisse souvent le `\n` (quand on appuie sur Entrée) dans le buffer après avoir lu un nombre, ce qui est immédiatement lu par un `%c` ou un `fgets` suivant si on ne fait pas attention.

Alternative plus sûre : Lire une ligne entière avec `fgets`, puis analyser la chaîne lue avec `sscanf` (similaire à `scanf` mais lit depuis une chaîne) ou d'autres fonctions de conversion (`strtol`, `strtod`).

```
#include <stdio.h>

int main() {
    int c; // Utiliser int pour pouvoir stocker EOF

    printf("Entrez du texte (Ctrl+D ou Ctrl+Z pour terminer) :\n");

    // Lit et affiche chaque caractère jusqu'à la fin de l'entrée
    while ((c = getchar()) != EOF) {
        putchar(c); // Ré-affiche le caractère lu
    }

    printf("\nFin de la lecture.\n");

    return 0;
}
```

Ces fonctions sont simples mais fondamentales, souvent utilisées comme blocs de construction pour des fonctions d'E/S plus complexes.

6. Structures de Contrôle : Conditions

Les structures de contrôle conditionnelles permettent à votre programme de prendre des décisions et d'exécuter différentes portions de code en fonction de l'état de vos variables ou des résultats d'expressions. En C, une condition est considérée comme **vraie** si elle évalue à une valeur **non nulle** (par exemple 1, -5, 100), et **fausse** si elle évalue à **zéro** (0). Les opérateurs relationnels (`==`, `!=`, `<`, `>`, etc.) et logiques (`&&`, `||`, `!`) sont très souvent utilisés pour former ces conditions.

Blocs d'instructions (`{}`)

Avant de voir les conditions, rappelons qu'un **bloc** est un ensemble d'une ou plusieurs instructions C délimitées par des accolades `{ ... }`. Partout où la syntaxe C attend une seule instruction (comme après un `if`, `else`, `while`, `for`), vous pouvez utiliser un bloc pour regrouper plusieurs instructions qui seront exécutées ensemble.

`if`, `else if`, `else`

La structure `if` est la plus fondamentale pour l'exécution conditionnelle.

Syntaxe de base `if` : Exécute le bloc de code suivant *uniquement* si la `condition` est vraie (non nulle).

(Voir l'exemple de code `code_if_else_section6_v2` pour l'illustration)

Syntaxe `if...else` : Exécute le premier bloc si la `condition` est vraie, et le second bloc (après `else`) si la `condition` est fausse (nulle).

(Voir l'exemple de code `code_if_else_section6_v2` pour l'illustration)

Syntaxe `if...else if...else` : Permet de tester plusieurs conditions séquentiellement. Dès qu'une `condition` vraie est trouvée, son bloc de code est exécuté, et le reste de la structure (`else if` suivants, `else` final) est ignoré. Le bloc `else` final est optionnel et s'exécute si aucune des conditions précédentes (`if` ou `else if`) n'était vraie.

(Voir l'exemple de code `code_if_else_section6_v2` pour l'illustration)

Important : Utilisez **toujours** des accolades `{}` pour délimiter les blocs de code associés à `if`, `else if`, et `else`, même s'il n'y a qu'une seule instruction. Cela améliore grandement la lisibilité et évite des erreurs subtiles, notamment lors de modifications ultérieures ou avec des `if` imbriqués (problème du "dangling else").

```c
#include <stdio.h>

int main() {
    int note;

    printf("Entrez votre note (0-20) : ");
    // On vérifie que scanf a bien lu un entier
    if (scanf("%d", &note) != 1) {
        printf("Erreur de saisie.\n");
        // Vider le buffer d'entrée en cas d'erreur
        while (getchar() != '\n');
```

```
            return 1; // Quitte le programme avec un code d'erreur
        }

        // Valider la note et donner une appréciation
        if (note < 0 || note > 20) {
            printf("Note invalide.\n");
        } else if (note >= 18) {
            printf("Excellent !\n");
        } else if (note >= 14) {
            printf("Très bien.\n");
        } else if (note >= 12) {
            printf("Bien.\n");
        } else if (note >= 10) {
            printf("Assez bien (Admis).\n");
        } else { // Si note < 10 (et valide, donc >= 0)
            printf("Insuffisant (Recalé).\n");
        }

        // Exemple simple if/else
        int nombre = -5;
        printf("\nTest de signe pour le nombre %d :\n", nombre);
        if (nombre >= 0) {
            printf("Le nombre est positif ou nul.\n");
        } else {
            printf("Le nombre est négatif.\n");
        }

        return 0;
    }
```

Opérateur Ternaire (`?:`)

L'opérateur ternaire (ou opérateur conditionnel) est une forme très concise de l'instruction `if...else`. Il est souvent utilisé pour assigner une valeur à une variable ou pour choisir une valeur à passer à une fonction, en fonction d'une condition simple.

Syntaxe :

```
condition ? expression_si_vrai : expression_si_faux
```

Fonctionnement :

1. La `condition` est évaluée.
2. Si elle est vraie (non nulle), l'`expression_si_vrai` est évaluée, et sa valeur devient la valeur de toute l'expression ternaire.
3. Si elle est fausse (nulle), l'`expression_si_faux` est évaluée, et sa valeur devient la valeur de toute l'expression ternaire.

Important : Seule *une* des deux expressions (`expression_si_vrai` ou `expression_si_faux`) est évaluée, jamais les deux (similaire au court-circuit).

```c
#include <stdio.h>

int main() {
    int a = 10, b = 20;
    int max_val;

    // Utilisation de l'opérateur ternaire pour trouver le maximum
    max_val = (a > b) ? a : b;
    //         ^condition  ^si vrai ^si faux
    printf("Le maximum entre %d et %d est : %d\n", a, b, max_val);

    // Équivalent avec if...else :
    /*
    if (a > b) {
        max_val = a;
    } else {
        max_val = b;
    }
    */

    // Autre exemple : déterminer si un nombre est pair ou impair
    int nombre = 7;
    // L'expression ternaire retourne une chaîne de caractères (char*)
    char *parite = (nombre % 2 == 0) ? "pair" : "impair";
    printf("Le nombre %d est %s.\n", nombre, parite);

    // On peut l'utiliser directement dans printf
    int score = 15;
    printf("Résultat : %s\n", (score >= 10) ? "Admis" : "Recalé");

    return 0;
}
```

Opérateur Ternaire (`?:`)

L'opérateur ternaire (ou opérateur conditionnel) est une forme très concise de l'instruction `if...else`. Il est souvent utilisé pour assigner une valeur à une variable ou pour choisir une valeur à passer à une fonction, en fonction d'une condition simple.

Syntaxe :

```
condition ? expression_si_vrai : expression_si_faux
```

Fonctionnement :

1. La `condition` est évaluée.
2. Si elle est vraie (non nulle), l'`expression_si_vrai` est évaluée, et sa valeur devient la valeur de toute l'expression ternaire.
3. Si elle est fausse (nulle), l'`expression_si_faux` est évaluée, et sa valeur devient la valeur de toute l'expression ternaire.

Important : Seule *une* des deux expressions (`expression_si_vrai` ou `expression_si_faux`) est évaluée, jamais les deux (similaire au court-circuit).

(Voir l'exemple de code `code_operateur_ternaire_section6_v2` pour une illustration pratique).

Bien que compact, l'opérateur ternaire peut rendre le code difficile à lire s'il est utilisé pour des logiques complexes ou s'il est imbriqué. Pour la clarté, préférez `if...else` dans ces cas.

`switch`, `case`, `break`, `default`

La structure `switch` est une alternative à une longue chaîne de `if...else if...else` lorsque toutes les comparaisons se font sur la **même expression entière** (types `int`, `char`, ou `enum`) et testent son égalité avec différentes **valeurs constantes**.

Syntaxe :

(La syntaxe est montrée dans l'exemple de code `code_switch_case_section6_v2`)

Fonctionnement :

1. L'`expression_entiere` est évaluée **une seule fois**.
2. Sa valeur est comparée séquentiellement à chaque `constante` des `case` (qui doivent être des constantes littérales ou définies, pas des variables).
3. Si une correspondance est trouvée, l'exécution **saute** directement au code de ce `case`.
4. **`break;`** : L'instruction `break` est **cruciale**. Elle provoque la sortie immédiate de *toute* la structure `switch`. Sans `break`, l'exécution continue dans les `case` suivants !
5. **Fall-through :** Si vous omettez `break` à la fin d'un `case`, l'exécution "tombe" (fall-through) et continue dans le code du `case` suivant, **sans vérifier sa constante**. C'est parfois utile (comme pour regrouper plusieurs `case` menant aux mêmes instructions), mais le plus souvent, c'est une source d'erreur si l'oubli n'est pas intentionnel. Commentez explicitement si le fall-through est voulu.
6. **`default:`** : Le bloc `default` (optionnel) s'exécute si la valeur de l'expression ne correspond à aucune des constantes `case`. Il est généralement placé à la fin, mais peut être mis ailleurs (le `break` serait alors nécessaire s'il n'est pas à la fin).

```c
#include <stdio.h>

int main() {
    int jour; // 1 pour Lundi, ..., 7 pour Dimanche

    printf("Entrez un numéro de jour (1-7) : ");
    if (scanf("%d", &jour) != 1) {
        printf("Saisie invalide.\n");
        // Vider le buffer en cas d'erreur de scanf
        while(getchar() != '\n');
        return 1;
    }
     // Vider le buffer après une lecture réussie aussi
    while(getchar() != '\n');

    switch (jour) {
        case 1:
```

```
            printf("Lundi\n");
            break; // Sort du switch
        case 2:
            printf("Mardi\n");
            break;
        case 3:
            printf("Mercredi\n");
            break;
        case 4:
            printf("Jeudi\n");
            break;
        case 5:
            printf("Vendredi\n");
            break; // Fin des jours de semaine

        case 6: // Fall-through : si jour == 6...
        case 7: // ...ou si jour == 7
            printf("Week-end !\n");
            break; // Sort du switch

        default: // Si 'jour' n'est ni 1, 2, 3, 4, 5, 6, ni 7
            printf("Numéro de jour invalide.\n");
            // Pas besoin de break ici, car c'est la fin (mais bonne
pratique)
    }

    // Autre exemple avec char
    char option;
    printf("\nChoisissez une option (A, B, C) : ");
    scanf(" %c", &option); // Espace pour ignorer le \n précédent

    switch (option) {
        case 'a': // Fall-through intentionnel pour gérer
minuscules/majuscules
        case 'A':
            printf("Option A choisie.\n");
            break;
        case 'b':
        case 'B':
            printf("Option B choisie.\n");
            break;
        case 'c':
        case 'C':
            printf("Option C choisie.\n");
            break;
        default:
            printf("Option inconnue '%c'.\n", option);
    }

    return 0;
}
```

Le `switch` est moins flexible que `if`/`else if` (ne peut tester que l'égalité avec des constantes entières), mais il est souvent plus lisible et potentiellement plus efficace lorsque vous avez de nombreux cas basés sur la même valeur entière.

---

7. Structures de Contrôle : Boucles

Les boucles sont des structures fondamentales en programmation qui permettent de **répéter l'exécution d'un bloc de code** plusieurs fois. C'est essentiel pour automatiser des tâches répétitives, traiter des collections de données (comme les éléments d'un tableau), ou attendre qu'une certaine condition soit remplie pour continuer. Le C propose trois types principaux de boucles : `for`, `while`, et `do...while`.

Boucle `for`

La boucle `for` est généralement la plus adaptée lorsque :

- Vous savez (ou pouvez calculer facilement) à l'avance **combien de fois** vous voulez répéter une action.
- Vous avez besoin d'une **variable compteur** (ou itérateur) qui change de manière prévisible à chaque passage (par exemple, incrémenter de 1 à chaque tour).

Syntaxe :

La structure générale d'une boucle `for` est : `for ( initialisation ; condition ; mise_a_jour ) { /* bloc de code */ }`

Fonctionnement détaillé :

1. **`initialisation`** : Cette partie est exécutée **une seule fois**, tout au début, *avant* que la boucle ne commence réellement. Elle sert typiquement à déclarer et/ou initialiser une variable compteur (par exemple, `int i = 0`). La déclaration peut se faire ici (depuis C99).
2. **`condition`** : Cette expression est évaluée *avant* chaque itération potentielle (y compris la toute première).
    - Si la condition est vraie (non nulle), le bloc de code `{...}` de la boucle est exécuté.
    - Si la condition est fausse (nulle), la boucle se termine immédiatement, et le programme continue à l'instruction suivant le bloc `for`.
3. **`mise_a_jour`** : Cette partie est exécutée *après* chaque exécution complète du bloc de code de la boucle, juste avant de réévaluer la `condition` pour l'itération suivante. Elle est typiquement utilisée pour modifier la variable compteur (par exemple, `i++`, `compteur -= 2`).

Les trois parties (`initialisation`, `condition`, `mise_a_jour`) sont techniquement optionnelles, mais les points-virgules `;` doivent toujours être présents. Un `for(;;)` crée une boucle infinie, équivalente à `while(1)`.

```c
#include <stdio.h>

int main() {
    int i; // Déclaration du compteur possible avant (style C plus ancien)

    printf("Compter de 0 à 4 avec for :\n");
```

```c
    // i commence à 0; la boucle continue tant que i < 5; i est incrémenté
après chaque tour
    for (i = 0; i < 5; i++) {
        printf("  Itération i = %d\n", i);
    }
    // Après la boucle, i vaut 5 (la condition i < 5 est devenue fausse)
    printf("Après la boucle, i = %d\n", i);

    printf("\nTable de multiplication par 3 (jusqu'à 30) :\n");
    // Déclaration et initialisation dans la boucle (style C99+)
    for (int n = 1; n * 3 <= 30; n++) {
        printf("  3 x %d = %d\n", n, n * 3);
    }

    // Boucle décrémentant
    printf("\nCompte à rebours de 5 à 1 :\n");
    for (int compte = 5; compte > 0; compte--) {
         printf("  %d...\n", compte);
    }
    printf("  Décollage !\n");

    return 0;
}
```

Boucle `while`

La boucle `while` répète un bloc de code **tant qu'une condition spécifiée reste vraie**. Elle est idéale lorsque :

- Le nombre d'itérations n'est **pas connu à l'avance**.
- La répétition dépend d'un événement externe ou d'une condition qui peut changer de manière imprévisible à l'intérieur de la boucle.

Syntaxe :

La structure générale d'une boucle `while` est : `while ( condition ) { /* bloc de code */ }`

Fonctionnement :

1. La `condition` est évaluée *avant* chaque itération potentielle.
2. Si la condition est vraie (non nulle), le bloc de code `{...}` est exécuté.
3. Après l'exécution du bloc, le contrôle retourne à l'étape 1 pour réévaluer la `condition`.
4. Si la condition est fausse (nulle), la boucle se termine et le programme continue après le bloc `while`.

Important : Si la condition est fausse dès le départ, le bloc de code n'est jamais exécuté. Il est crucial que le code à l'intérieur de la boucle puisse potentiellement rendre la condition fausse, sinon vous créerez une boucle infinie.

```c
#include <stdio.h>

int main() {
```

```c
    int compteur = 5;

    printf("Compte à rebours (while) :\n");

    // Tant que compteur est strictement positif
    while (compteur > 0) {
        printf("  %d...\n", compteur);
        // Modification essentielle pour que la condition devienne fausse
        compteur--;
    }
    printf("  Décollage !\n");

    // Exemple: attendre une entrée spécifique
    char reponse = ' ';
    printf("\nTapez 'o' pour continuer : ");
    // Tant que la réponse n'est pas 'o' (et qu'on n'est pas en fin
d'entrée)
    while(reponse != 'o') {
        reponse = getchar(); // Lit un caractère
        // Ignorer les caractères restants sur la ligne (simplifié)
        if (reponse != '\n' && reponse != EOF) {
             while(getchar() != '\n');
        }
        if (reponse != 'o') {
            printf("Incorrect. Tapez 'o' : ");
        }
    }
    printf("Vous avez continué !\n");


    return 0;
}
```

Boucle `do...while`

La boucle `do...while` est très similaire à la boucle `while`, mais avec une différence clé : la condition est testée **après** l'exécution du bloc de code, et non avant. Cela garantit que le bloc de code est **toujours exécuté au moins une fois**, même si la condition est initialement fausse.

Syntaxe :

La structure générale d'une boucle `do...while` est : `do { /* bloc de code */ } while ( condition );`

Notez le point-virgule obligatoire après la parenthèse fermante du `while`.

Fonctionnement :

1. Le bloc de code `{...}` est exécuté une première fois.
2. Ensuite, la `condition` est évaluée.
3. Si la condition est vraie (non nulle), le contrôle retourne au début du bloc `do` pour une nouvelle itération.

4. Si la condition est fausse (nulle), la boucle se termine.

Cette boucle est particulièrement utile pour les menus où l'on veut afficher les options au moins une fois avant de demander le choix de l'utilisateur.

```c
#include <stdio.h>

int main() {
    int choix;

    // La boucle s'exécute au moins une fois pour afficher le menu
    do {
        // Afficher le menu
        printf("\n--- MENU ---\n");
        printf("1. Option 1\n");
        printf("2. Option 2\n");
        printf("0. Quitter\n");
        printf("Votre choix : ");

        // Lire le choix de l'utilisateur
        if (scanf("%d", &choix) != 1) {
            printf("Erreur de saisie. Veuillez entrer un nombre.\n");
            // Vider le buffer d'entrée
            while (getchar() != '\n');
            choix = -1; // Assigner une valeur invalide pour continuer la
boucle
            continue;   // Passer à l'itération suivante
        }
        // Vider le buffer après une lecture réussie
        while (getchar() != '\n');


        // Traiter le choix
        switch (choix) {
            case 1:
                printf("-> Vous avez choisi l'Option 1.\n");
                break;
            case 2:
                printf("-> Vous avez choisi l'Option 2.\n");
                break;
            case 0:
                printf("-> Au revoir !\n");
                break;
            default:
                printf("-> Choix invalide.\n");
        }

    } while (choix != 0); // Répéter tant que l'utilisateur n'a pas choisi
0

    printf("Programme terminé.\n");
```

```
    return 0;
}
```

Instructions `break` et `continue`

Ces deux instructions offrent un contrôle plus fin sur le déroulement des boucles (`for`, `while`, et `do...while`). `break` fonctionne aussi dans les `switch`.

- **`break;`** : Provoque la **sortie immédiate** de la boucle (ou du `switch`) la **plus interne** dans laquelle elle se trouve. Le programme continue son exécution à la première instruction *après* la boucle (ou le `switch`) terminée.
- **`continue;`** : **Saute le reste de l'itération actuelle** de la boucle la plus interne. Elle ne termine pas la boucle entière, mais passe directement à l'itération suivante :
    - Dans une boucle `while` ou `do...while`, l'exécution saute directement à l'évaluation de la `condition`.
    - Dans une boucle `for`, l'exécution saute à l'étape de `mise_a_jour`, puis évalue la `condition`.

```c
#include <stdio.h>

int main() {
    int i;

    printf("Boucle avec 'continue' (affiche seulement les nombres impairs < 10) :\n");
    for (i = 0; i < 10; i++) {
        if (i % 2 == 0) { // Si i est pair...
            continue; // ...on saute le reste de cette itération (le printf)
                      // et on passe directement à i++ puis au test i < 10
        }
        // Ce printf n'est exécuté que si i est impair
        printf("%d ", i);
    }
    printf("\n");


    printf("\nBoucle avec 'break' (cherche le premier multiple de 7 entre 1 et 20) :\n");
    int nombre_trouve = -1; // Initialisation à une valeur indiquant "non trouvé"
    for (i = 1; i <= 20; i++) {
        printf("Test de %d...\n", i);
        if (i % 7 == 0) { // Si i est un multiple de 7
            nombre_trouve = i;
            printf("Trouvé ! Arrêt de la boucle.\n");
            break; // Sortir immédiatement de la boucle for
        }
        // Si on n'a pas fait break, on continue normalement l'itération
        printf("Ce n'est pas un multiple de 7.\n");
```

```
    }

    // L'exécution reprend ici après le break (ou après la fin normale de
la boucle)
    if (nombre_trouve != -1) {
        printf("Le premier multiple de 7 trouvé est : %d\n",
nombre_trouve);
    } else {
        printf("Aucun multiple de 7 trouvé entre 1 et 20.\n");
    }

    return 0;
}
```

L'utilisation judicieuse de `break` et `continue` peut parfois simplifier la logique de certaines boucles (par exemple, sortir tôt si une condition spéciale est remplie, ou ignorer certaines valeurs sans imbriquer des `if` complexes). Cependant, une utilisation excessive peut rendre le code plus difficile à suivre et à déboguer ("programmation spaghetti").

Boucles imbriquées

Vous pouvez placer une boucle à l'intérieur d'une autre boucle. C'est ce qu'on appelle des **boucles imbriquées**. La boucle interne s'exécutera complètement pour *chaque* itération de la boucle externe.

C'est très courant pour travailler avec des structures de données bidimensionnelles (comme les tableaux 2D, les matrices, les images) ou pour générer des combinaisons.

```
#include <stdio.h>

int main() {
    int max_ligne = 3;
    int max_colonne = 4;

    printf("Affichage de coordonnées (ligne, colonne):\n");

    // Boucle externe pour les lignes
    for (int ligne = 0; ligne < max_ligne; ligne++) {
        printf("Ligne %d : ", ligne);

        // Boucle interne pour les colonnes
        // Cette boucle s'exécute entièrement pour chaque valeur de 'ligne'
        for (int colonne = 0; colonne < max_colonne; colonne++) {
            printf("(%d,%d) ", ligne, colonne);
        }

        printf("\n"); // Nouvelle ligne après avoir parcouru toutes les
colonnes pour une ligne
    }

    return 0;
}
```

---

8. La Modularité : Les Fonctions

Au fur et à mesure que les programmes deviennent plus complexes, il devient essentiel de les organiser. Écrire tout le code dans la fonction `main` devient rapidement ingérable, difficile à lire, à déboguer et à maintenir. Les **fonctions** sont le principal outil en C pour décomposer un programme en blocs de code plus petits, logiques et réutilisables, chacun responsable d'une tâche spécifiques.

Pourquoi utiliser des fonctions ?

- **Modularité et Organisation :** Permet de diviser un problème complexe en sous-problèmes plus simples à gérer. Chaque fonction encapsule une tâche précise (ex: `calculer_moyenne`, `afficher_menu`, `lire_entree_utilisateur`). Cela rend le programme global plus facile à comprendre et à structurer.
- **Réutilisabilité :** Une fois qu'une fonction est écrite et testée, elle peut être **appelée** (utilisée) depuis n'importe quelle autre partie du programme, autant de fois que nécessaire, sans avoir à réécrire le même code. C'est le principe **DRY** (Don't Repeat Yourself - Ne vous répétez pas).
- **Lisibilité :** La fonction `main` (ou d'autres fonctions de haut niveau) peut se concentrer sur l'enchaînement logique principal du programme, en appelant des fonctions aux noms descriptifs. Les détails d'implémentation de chaque tâche sont masqués à l'intérieur des fonctions appelées.
- **Maintenance et Débogage :** Il est beaucoup plus facile de tester, corriger une erreur (déboguer) ou mettre à jour une petite fonction isolée qui fait une seule chose, qu'un énorme bloc de code monolithique où tout est mélangé. Si une fonction spécifique cause un problème, on peut se concentrer sur celle-ci.

Déclaration (Prototype) et Définition

En C, pour pouvoir utiliser une fonction, le compilateur doit savoir "à quoi elle ressemble" *avant* de rencontrer le premier appel à cette fonction. Il a besoin de connaître :

- Le **nom** de la fonction.
- Le **type de la valeur** qu'elle retourne (si elle en retourne une).
- Le **nombre et le type** des informations (paramètres) qu'elle attend en entrée.

Cette information est fournie soit par la **définition** complète de la fonction, soit par une **déclaration** préalable, appelée **prototype**.

Définition de Fonction : C'est là que vous écrivez le code *réel* qui exécute la tâche de la fonction. La définition inclut l'en-tête de la fonction et son corps (le bloc de code).

L'en-tête spécifie le type de retour, le nom et les paramètres. Le corps contient les instructions entre accolades.

- `type_retour`: Le type de la valeur que la fonction renvoie (ex: `int`, `float`, `char*`, `void` si elle ne renvoie rien).
- `nom_fonction`: Un nom significatif décrivant ce que fait la fonction (suit les mêmes règles que les noms de variables).

- `paramètres (liste entre parenthèses)` : Déclarations de variables qui recevront les données d'entrée de la fonction lors de son appel. Chaque paramètre a un type et un nom. S'il n'y a pas de paramètres, on utilise `()` ou, de préférence pour indiquer explicitement l'absence de paramètres, `(void)`.
- `corps`: Le bloc de code `{ ... }` contenant les instructions qui réalisent la tâche de la fonction.

Déclaration de Fonction (Prototype) : Un prototype est une simple déclaration qui informe le compilateur de la "signature" de la fonction (type de retour, nom, types des paramètres) sans fournir son corps. C'est **nécessaire si vous appelez une fonction *avant* de l'avoir définie** dans le même fichier, ou si la fonction est définie dans un autre fichier source (`.c`). Les prototypes sont typiquement placés en haut du fichier `.c` ou, mieux, dans un fichier d'en-tête (`.h`) qui sera inclus.

La syntaxe est `type_retour nom_fonction(type1, type2, ...);` (Notez le point-virgule à la fin !). Les noms des paramètres sont optionnels dans le prototype, mais les types sont obligatoires.

```c
#include <stdio.h>

// Prototype (déclaration) de la fonction 'ajouter'
// Informe le compilateur qu'une fonction nommée 'ajouter' existe,
// prend deux 'int' en paramètres, et retourne un 'int'.
int ajouter(int a, int b);
// On aurait aussi pu écrire : int ajouter(int, int); mais c'est moins
clair.

// Prototype d'une fonction qui ne retourne rien (void)
void afficher_message(void); // (void) indique explicitement aucun
paramètre

int main() {
    int x = 15, y = 7;
    int somme;

    // Appel de la fonction 'ajouter'. Le compilateur connaît sa signature
grâce au prototype.
    somme = ajouter(x, y);

    printf("%d + %d = %d\n", x, y, somme);

    // Appel de la fonction 'afficher_message'
    afficher_message();

    return 0;
}

// Définition (implémentation) de la fonction 'ajouter'
// L'en-tête doit correspondre au prototype (types et ordre des paramètres)
int ajouter(int a, int b) {
    int resultat = a + b;
    return resultat; // Renvoie la somme calculée
}
```

```c
// Définition de la fonction 'afficher_message'
void afficher_message(void) {
    printf("Ceci est un message depuis une fonction void.\n");
    // Pas de 'return' avec une valeur nécessaire pour une fonction void.
    // On pourrait utiliser 'return;' pour sortir plus tôt si besoin.
}
```

Sans le prototype, si l'appel à `ajouter` ou `afficher_message` apparaissait avant leur définition, le compilateur générerait un avertissement ou une erreur (selon les options de compilation), car il ne saurait pas comment vérifier si l'appel est correct.

Appel de fonction

Pour exécuter le code contenu dans une fonction, on l'**appelle** en utilisant son nom suivi de parenthèses `()`. S'il y a des paramètres définis pour la fonction, on doit fournir des **arguments** (les valeurs concrètes) entre les parenthèses, dans le même ordre et de types compatibles.

Par exemple : `nom_fonction(argument1, argument2);`

Si la fonction renvoie une valeur (type de retour autre que `void`), on peut utiliser cette valeur de retour, par exemple en l'assignant à une variable ou en l'utilisant directement dans une expression :

```c
type_variable resultat = nom_fonction(arg1, arg2); printf("Le résultat est : %d\n", nom_fonction(val1, val2)); if (nom_fonction(x) > 10) { /* ... */ }
```

(Voir l'exemple de code `code_fonction_prototype_section8` pour une illustration pratique de l'appel).

Paramètres et Arguments (Passage par valeur)

Il est crucial de distinguer :

- **Paramètres (ou paramètres formels) :** Les variables déclarées dans la liste entre parenthèses de la **définition** de la fonction (ex: `int a, int b` dans `int ajouter(int a, int b)`). Ils agissent comme des variables locales à l'intérieur de la fonction.
- **Arguments (ou paramètres effectifs) :** Les valeurs ou variables réelles fournies lors de l'**appel** de la fonction (ex: `x, y` dans `ajouter(x, y)`).

En C, par défaut, les arguments sont passés **par valeur (pass by value)**. Cela signifie que :

1. Lorsque vous appelez une fonction, la **valeur** de chaque argument est évaluée et **copiée**.
2. Ces **copies** sont assignées aux paramètres correspondants de la fonction appelée.
3. La fonction travaille uniquement sur ces **copies locales**.
4. **Par conséquent, les modifications apportées aux paramètres à l'intérieur de la fonction n'affectent PAS les variables originales (arguments) dans la fonction appelante.**

```c
#include <stdio.h>

// Fonction qui tente de modifier son paramètre 'p'
void essayer_modifier(int p) {
    printf("  -> Entrée dans essayer_modifier: valeur de p (paramètre) =
```

```
%d\n", p);
    p = 99; // Modification de la COPIE locale 'p'
    printf("  -> Sortie de essayer_modifier: valeur de p (paramètre) =
%d\n", p);
}

int main() {
    int variable_originale = 10;

    printf("Dans main: variable_originale avant appel = %d\n",
variable_originale);

    // Appel de la fonction en passant la *valeur* de variable_originale
    essayer_modifier(variable_originale);

    // La variable originale dans main n'a PAS été modifiée
    printf("Dans main: variable_originale après appel = %d\n",
variable_originale);

    return 0;
}
```

Appel de fonction

Pour exécuter le code contenu dans une fonction, on l'**appelle** en utilisant son nom suivi de parenthèses `()`. S'il y a des paramètres définis pour la fonction, on doit fournir des **arguments** (les valeurs concrètes) entre les parenthèses, dans le même ordre et de types compatibles.

Par exemple : `nom_fonction(argument1, argument2);`

Si la fonction renvoie une valeur (type de retour autre que `void`), on peut utiliser cette valeur de retour, par exemple en l'assignant à une variable ou en l'utilisant directement dans une expression :

`type_variable resultat = nom_fonction(arg1, arg2); printf("Le résultat est : %d\n", nom_fonction(val1, val2)); if (nom_fonction(x) > 10) { /* ... */ }`

Paramètres et Arguments (Passage par valeur)

Il est crucial de distinguer :

- **Paramètres (ou paramètres formels) :** Les variables déclarées dans la liste entre parenthèses de la **définition** de la fonction (ex: `int a, int b` dans `int ajouter(int a, int b)`). Ils agissent comme des variables locales à l'intérieur de la fonction.
- **Arguments (ou paramètres effectifs) :** Les valeurs ou variables réelles fournies lors de l'**appel** de la fonction (ex: `x, y` dans `ajouter(x, y)`).

En C, par défaut, les arguments sont passés **par valeur (pass by value)**. Cela signifie que :

1. Lorsque vous appelez une fonction, la **valeur** de chaque argument est évaluée et **copiée**.
2. Ces **copies** sont assignées aux paramètres correspondants de la fonction appelée.
3. La fonction travaille uniquement sur ces **copies locales**.

4. **Par conséquent, les modifications apportées aux paramètres à l'intérieur de la fonction n'affectent PAS les variables originales (arguments) dans la fonction appelante.**

(Voir l'exemple de code `code_passage_valeur_section8` pour une illustration pratique).

Comme le montre l'exemple, la `variable_originale` dans `main` conserve sa valeur 10 après l'appel à `essayer_modifier`, car la fonction n'a travaillé que sur une copie de cette valeur. Nous verrons dans la section sur les pointeurs comment utiliser le **passage par adresse** pour permettre à une fonction de modifier les variables de l'appelant.

```c
#include <stdio.h>
#include <stdlib.h> // Pour atoi

// Utilise la signature complète de main pour accéder aux arguments
int main(int argc, char *argv[]) {
    int i;

    printf("Nombre total d'arguments reçus (argc) : %d\n", argc);
    printf("Nom du programme (argv[0]) : %s\n", argv[0]);

    // Vérifie s'il y a des arguments supplémentaires
    if (argc > 1) {
        printf("Arguments passés en ligne de commande :\n");
        // Boucle de 1 (pour ignorer argv[0]) jusqu'à argc-1
        for (i = 1; i < argc; i++) {
            printf("  Argument %d (argv[%d]) : \"%s\"\n", i, i, argv[i]);
        }

        // Exemple de conversion du premier argument en entier (si
possible)
        // Attention: atoi retourne 0 en cas d'erreur ou si la chaîne est
"0"
        // strtol offre une meilleure gestion des erreurs.
        int premier_arg_entier = atoi(argv[1]);
        printf("\nTentative de conversion du premier argument en entier :
%d\n", premier_arg_entier);

    } else {
        printf("Aucun argument supplémentaire n'a été passé.\n");
    }

    return 0; // Succès
}

/* Compilation et exécution possibles :
   1. Compilation :
      gcc exemple_args.c -o exemple_args

   2. Exécution sans arguments :
      ./exemple_args
      Sortie :
      Nombre total d'arguments reçus (argc) : 1
```

```
       Nom du programme (argv[0]) : ./exemple_args
       Aucun argument supplémentaire n'a été passé.

   3. Exécution avec arguments :
      ./exemple_args bonjour 123 "test avec espaces"
      Sortie :
      Nombre total d'arguments reçus (argc) : 4
      Nom du programme (argv[0]) : ./exemple_args
      Arguments passés en ligne de commande :
        Argument 1 (argv[1]) : "bonjour"
        Argument 2 (argv[2]) : "123"
        Argument 3 (argv[3]) : "test avec espaces"

      Tentative de conversion du premier argument en entier : 0
*/
```

Valeur de Retour (`return`)

L'instruction `return` a deux fonctions principales :

1. **Terminer l'exécution** de la fonction dans laquelle elle se trouve. Le contrôle retourne immédiatement à l'endroit où la fonction a été appelée (le point d'appel).
2. **Renvoyer une valeur** (si la fonction n'est pas de type `void`) à la fonction appelante. La valeur de l'`expression` qui suit `return` est renvoyée.

- Une fonction déclarée avec un type de retour autre que `void` (ex: `int`, `float*`, `struct Point`) **doit** contenir au moins une instruction `return expression;` sur chaque chemin d'exécution possible qui atteint la fin de la fonction. L'`expression` doit être d'un type compatible avec le type de retour déclaré.
- Une fonction déclarée `void` ne renvoie aucune valeur. Elle peut utiliser `return;` (sans expression) pour sortir prématurément. Si elle n'a pas de `return;` explicite, elle se termine simplement lorsque la dernière instruction de son corps est atteinte.
- La fonction `main` est un cas particulier : elle **doit** retourner un `int`. `return 0;` est la convention pour indiquer une exécution réussie. Une valeur non nulle (par exemple `return 1;` ou en utilisant les constantes `EXIT_FAILURE` de `<stdlib.h>`) signale une erreur au système d'exploitation ou au script qui a lancé le programme.

Portée des Variables (Locales vs Globales)

La **portée** (scope) d'une variable détermine la partie du code où cette variable est "visible" et peut être utilisée.

- **Variables Locales :**
  - Déclarées **à l'intérieur** d'une fonction ou d'un bloc `{}` (y compris les paramètres de fonction).
  - Leur portée est limitée au bloc `{...}` dans lequel elles sont déclarées. Elles ne sont pas accessibles depuis l'extérieur de ce bloc.
  - Elles ont une **durée de vie automatique** : elles sont créées lorsque l'exécution entre dans leur bloc (généralement sur la *pile d'exécution* ou *stack*) et sont détruites lorsque l'exécution quitte ce bloc. Leurs valeurs ne persistent pas entre les appels de fonction (sauf si elles sont déclarées `static`).

    - Si non initialisées explicitement, leur valeur initiale est indéterminée (contient des "déchets").

- **Variables Globales :**
    - Déclarées **à l'extérieur** de toute fonction (généralement en haut du fichier source, avant toute fonction).
    - Leur portée s'étend du point de leur déclaration jusqu'à la fin du fichier (`.c`). Elles peuvent être rendues accessibles à d'autres fichiers via le mot-clé `extern` (voir Section 17).
    - Elles ont une **durée de vie statique** : elles existent et conservent leur valeur pendant toute la durée d'exécution du programme.
    - Elles sont initialisées à zéro (ou `NULL` pour les pointeurs) par défaut si aucune valeur initiale n'est fournie.
    - **Abus à éviter :** L'utilisation excessive de variables globales rend le code difficile à comprendre (qui modifie quoi et quand ?), augmente le risque d'effets de bord imprévus entre différentes parties du programme, et rend les fonctions moins indépendantes et réutilisables. Privilégiez la communication via les paramètres et les valeurs de retour des fonctions.

```c
#include <stdio.h>

int variable_globale = 100; // Variable globale, visible dans tout ce
fichier

void fonction_exemple() {
    int variable_locale = 10; // Locale à fonction_exemple
                              // Créée à chaque appel, détruite à la
sortie

    printf("  Dans fonction_exemple: variable_locale = %d\n",
variable_locale);
    printf("  Dans fonction_exemple: variable_globale = %d\n",
variable_globale);

    variable_locale++;
    variable_globale++; // On peut modifier la globale depuis n'importe où

    printf("  Dans fonction_exemple (après modif): variable_locale = %d\n",
variable_locale);
    printf("  Dans fonction_exemple (après modif): variable_globale =
%d\n", variable_globale);
}

int main() {
    int variable_locale_main = 5; // Locale à main

    printf("Dans main (avant appels):\n");
    printf("  variable_locale_main = %d\n", variable_locale_main);
    printf("  variable_globale = %d\n", variable_globale);
    // printf("  variable_locale = %d\n", variable_locale); // ERREUR: Non
visible ici

    printf("\nAppel 1 de fonction_exemple:\n");
```

```
    fonction_exemple();

    printf("\nAppel 2 de fonction_exemple:\n");
    fonction_exemple(); // variable_locale est recréée et vaut 10 à nouveau

    printf("\nDans main (après appels):\n");
    printf("  variable_locale_main = %d\n", variable_locale_main);
    printf("  variable_globale = %d\n", variable_globale); // La globale a
été modifiée

    // Bloc imbriqué dans main
    if (variable_globale > 100) {
        int variable_locale_if = 99; // Locale uniquement à ce bloc if
        printf("\n  Dans le bloc if: variable_locale_if = %d\n",
variable_locale_if);
        printf("  Dans le bloc if: variable_locale_main = %d\n",
variable_locale_main); // Visible
    }
    // printf("variable_locale_if = %d\n", variable_locale_if); // ERREUR:
Non visible ici

    return 0;
}
```

Variables Statiques (`static` locales)

Le mot-clé `static` a plusieurs usages en C. Lorsqu'il est appliqué à une **variable locale** (déclarée à l'intérieur d'une fonction), il modifie sa **durée de vie** sans changer sa **portée**.

- **Durée de vie Statique :** Contrairement aux variables locales normales qui sont créées et détruites à chaque appel, une variable locale `static` existe pendant **toute la durée d'exécution du programme**, comme une variable globale. L'espace mémoire lui est alloué une seule fois.
- **Initialisation Unique :** Elle est initialisée **une seule fois**, la toute première fois que l'exécution passe par sa déclaration. Si elle n'est pas explicitement initialisée, elle est mise à 0 (ou `NULL` pour les pointeurs) par défaut. Lors des appels suivants à la fonction, l'étape d'initialisation est sautée.
- **Persistance de la valeur :** Puisqu'elle existe pendant toute l'exécution, elle **conserve sa valeur** entre les appels successifs de la fonction.
- **Portée Locale :** Malgré sa durée de vie étendue, elle reste une variable locale : elle n'est **accessible** (visible) qu'à l'intérieur de la fonction (ou du bloc) où elle est déclarée.

C'est utile pour maintenir un état ou un compteur à l'intérieur d'une fonction sans utiliser de variable globale.

```
#include <stdio.h>

void compteur_visites() {
    // Cette variable existe pendant toute l'exécution du programme.
    // Elle est initialisée à 0 uniquement avant le tout premier appel.
    static int nombre_visites = 0;

    nombre_visites++; // Incrémente la valeur persistante
```

```
    printf("Cette fonction a été visitée %d fois.\n", nombre_visites);
}

int main() {
    printf("Premier appel :\n");
    compteur_visites(); // nombre_visites devient 1

    printf("Deuxième appel :\n");
    compteur_visites(); // nombre_visites devient 2 (elle a conservé sa
valeur)

    printf("Troisième appel :\n");
    compteur_visites(); // nombre_visites devient 3

    // printf("%d", nombre_visites); // ERREUR: La variable n'est pas
accessible ici

    return 0;
}
```

Récursivité (Introduction)

Une fonction est dite **récursive** si elle s'appelle elle-même, soit directement (`func()` appelle `func()`), soit indirectement (`funcA()` appelle `funcB()`, qui appelle `funcA()`). La récursivité est une technique de programmation puissante et élégante pour résoudre des problèmes qui peuvent être définis en termes d'une version plus simple d'eux-mêmes. C'est souvent le cas pour des problèmes mathématiques (comme la factorielle, Fibonacci) ou pour parcourir des structures de données arborescentes (comme des arbres binaires).

Pour qu'une fonction récursive fonctionne correctement et ne boucle pas indéfiniment (ce qui mènerait à un **dépassement de la pile d'exécution** ou *stack overflow*), elle doit **impérativement** comporter deux éléments :

1. **Un (ou plusieurs) cas de base (Base Case) :** Une condition simple qui arrête la récursion. Lorsque le cas de base est atteint, la fonction retourne une valeur directement **sans faire d'appel récursif supplémentaire**.
2. **Une étape récursive (Recursive Step) :** La partie où la fonction s'appelle elle-même, mais en passant des arguments modifiés qui **convergent progressivement vers le cas de base**.

Exemple Classique : Factorielle (n!)

La factorielle d'un entier non négatif `n`, notée `n!`, est le produit de tous les entiers positifs de 1 à `n`.

- $n! = n \times (n-1) \times (n-2) \times \dots \times 1$
- Par convention, $0! = 1$.

On peut définir la factorielle de manière récursive :

- **Cas de base :** Si `n` est 0, `factorielle(n)` retourne 1.
- **Étape récursive :** Si `n` est supérieur à 0, `factorielle(n)` retourne `n * factorielle(n - 1)`. L'appel récursif se fait avec `n-1`, ce qui nous rapproche du cas de base `n=0`.

```c
#include <stdio.h>

// Calcule n! de manière récursive
// Utilise unsigned long long pour gérer de plus grands résultats
unsigned long long factorielle(int n) {

    // Cas de base 1: Gestion d'erreur pour les nombres négatifs
    if (n < 0) {
        fprintf(stderr, "Erreur: Factorielle non définie pour n < 0\n");
        // On pourrait retourner une valeur spéciale ou gérer l'erreur
autrement
        return 0; // Retourne 0 pour indiquer une erreur dans cet exemple
    }
    // Cas de base 2: Factorielle de 0 est 1
    else if (n == 0) {
        printf("  Cas de base atteint (n=0), retourne 1\n");
        return 1ULL; // 1 en tant que Unsigned Long Long
    }
    // Étape récursive: n! = n * (n-1)!
    else {
        printf("  Calcul de %d! : besoin de %d * factorielle(%d)\n", n, n,
n - 1);
        // Attention: risque de dépassement de capacité pour n > 20 environ
        unsigned long long resultat_precedent = factorielle(n - 1);
        unsigned long long resultat_actuel = (unsigned long long)n *
resultat_precedent;
        printf("  Retour de factorielle(%d) : %llu\n", n, resultat_actuel);
        return resultat_actuel;
    }
}

int main() {
    int num = 4;
    printf("Calcul de factorielle(%d) :\n", num);
    unsigned long long resultat = factorielle(num);

    if (resultat != 0 || num == 0) { // Vérifie si l'erreur n'a pas été
retournée (sauf pour 0!)
       printf("\nFactorielle de %d = %llu\n", num, resultat);
    }

    printf("\nCalcul de factorielle(0) :\n");
    resultat = factorielle(0);
    printf("\nFactorielle de 0 = %llu\n", resultat);

    // Test du cas d'erreur
    // factorielle(-2);

    return 0;
}
```

Avantages et Inconvénients de la Récursivité :

- **Avantages :**
  - Peut mener à un code plus court, plus élégant et plus facile à comprendre pour les problèmes naturellement récursifs.
  - Correspond souvent directement à la définition mathématique du problème.
- **Inconvénients :**
  - **Consommation mémoire :** Chaque appel récursif ajoute une nouvelle "frame" sur la pile d'exécution (pour stocker les paramètres, les variables locales, l'adresse de retour). Une récursion trop profonde peut épuiser la mémoire de la pile et causer un crash (*stack overflow*).
  - **Performance :** Les appels de fonction ont un certain coût (sauvegarde du contexte, saut d'adresse...). Pour certains problèmes (comme la factorielle ou Fibonacci simple), une solution itérative (avec une boucle) est souvent plus rapide et consomme moins de mémoire.
  - **Complexité :** Peut être plus difficile à déboguer ou à raisonner pour certains développeurs.

La récursivité est un outil puissant à avoir dans sa boîte à outils, mais il faut l'utiliser judicieusement et toujours s'assurer qu'il y a un cas de base clair pour éviter les boucles infinies.

---

9. Les Tableaux

Jusqu'à présent, nous avons manipulé des variables contenant une seule valeur à la fois (un entier, un flottant, un caractère). Les **tableaux** (Arrays en anglais) constituent une structure de données fondamentale en C qui nous permet de stocker une **collection d'éléments de même type** sous un seul nom.

Imaginez devoir gérer les notes de 30 étudiants, une série de températures relevées chaque heure, ou les noms des jours de la semaine : utiliser une variable distincte pour chaque valeur serait très fastidieux. Un tableau est la structure idéale pour représenter ces collections ordonnées de données homogènes.

Les éléments d'un tableau sont stockés de manière **contiguë** en mémoire, c'est-à-dire les uns à la suite des autres, ce qui permet un accès efficace à chaque élément.

Tableaux à une dimension

C'est la forme la plus simple et la plus courante de tableau : une séquence linéaire d'éléments du même type.

Déclaration, Initialisation, Accès aux éléments

Déclaration : Pour déclarer un tableau à une dimension, vous devez spécifier :

1. Le **type** des éléments que le tableau contiendra.
2. Le **nom** que vous donnerez au tableau (suit les règles des noms de variables).
3. La **taille** (le nombre total d'éléments) du tableau, indiquée entre crochets `[]`. Pour les tableaux standards dont la mémoire est allouée sur la pile ou statiquement, la taille doit être une **expression constante entière positive** connue à la compilation.

Syntaxe : `type nom_tableau[TAILLE];`

Exemples : `int notes[30];` // Déclare un tableau nommé 'notes' pouvant stocker 30 entiers. `float temperatures[24];` // Déclare un tableau pour 24 températures (flottants). `char nom_utilisateur[50];` // Déclare un tableau pour 50 caractères (souvent utilisé pour les chaînes).

Cette déclaration réserve un bloc de mémoire contigu suffisamment grand pour stocker le nombre spécifié d'éléments du type donné. Par exemple, si un `int` occupe 4 octets, `int notes[30];` réservera $30 \times 4 = 120$ octets consécutifs.

Initialisation : Vous pouvez donner des valeurs initiales aux éléments d'un tableau au moment de sa déclaration, en utilisant une liste de valeurs entre accolades `{}`.

- **Initialisation complète :** Fournir une valeur pour chaque élément. `int nombres[5] = {10, -2, 30, 8, 50};`
- **Initialisation partielle :** Si vous fournissez moins de valeurs que la taille déclarée, les éléments restants sont automatiquement initialisés à **zéro** (pour les types numériques) ou au caractère nul `\0` (pour les `char`). C'est vrai pour les variables globales, statiques, et souvent pour les locales dans les standards C modernes ou avec les options de compilateur courantes, mais il est plus sûr de ne pas compter dessus pour les variables locales automatiques des standards plus anciens. `float mesures[10] = {1.5f, 2.3f, 0.8f}; // mesures[3] à mesures[9] sont initialisés à 0.0f`
- **Initialisation sans taille explicite :** Si vous initialisez un tableau sans spécifier sa taille entre les crochets, le compilateur déterminera la taille automatiquement en fonction du nombre de valeurs fournies dans la liste. `int jours_par_mois[] = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31}; // Taille 12 déterminée par le compilateur`

Accès aux éléments : On accède à un élément individuel du tableau en utilisant son **indice** (index) entre crochets `[]` après le nom du tableau.

TRÈS IMPORTANT : Les indices en C (et dans beaucoup d'autres langages) commencent toujours à 0 !

Pour un tableau `tab` de taille `N`, les indices valides vont de **0** à **N-1**.

- `tab[0]` : Accède au **premier** élément.
- `tab[1]` : Accède au **deuxième** élément.
- ...
- `tab[N-1]` : Accède au **dernier** élément.

ATTENTION : Dépassement de Tampon (Buffer Overflow/Underflow) Le langage C **ne vérifie pas** automatiquement si l'indice que vous utilisez est valide (c'est-à-dire dans les limites `[0, N-1]`). Accéder à un élément avec un indice en dehors de cette plage (par exemple `tab[N]` ou `tab[-1]`) est une erreur grave appelée **dépassement (ou sous-dépassement) de tampon**. Cela conduit à un **comportement indéfini** : votre programme peut planter ("Segmentation fault"), donner des résultats incorrects en lisant ou écrivant dans une zone mémoire qui ne lui appartient pas, ou pire, créer une faille de sécurité exploitable. C'est **au programmeur** d'être vigilant et de s'assurer que les indices restent dans les limites autorisées.

```c
#include <stdio.h>

int main() {
```

```c
    // Déclaration et initialisation complète
    int scores[5] = {15, 18, 12, 14, 19};

    // Déclaration avec initialisation partielle (taille 4, 2 éléments
fournis)
    float valeurs[4] = {9.81f, -3.14f}; // valeurs[2] et valeurs[3] seront
0.0f

    // Accès en lecture (indices de 0 à 4 pour scores)
    printf("Le premier score est : %d\n", scores[0]); // Affiche 15
    printf("Le troisième score est : %d\n", scores[2]); // Affiche 12
    printf("La dernière valeur est : %.2f\n", valeurs[3]); // Affiche 0.00

    // Accès en écriture (modification d'un élément)
    printf("\nModification du score à l'indice 2...\n");
    scores[2] = 13; // Change le troisième score de 12 à 13
    printf("Le troisième score modifié est : %d\n", scores[2]); // Affiche
13

    // Parcourir le tableau avec une boucle for
    printf("\nAffichage de tous les scores :\n");
    // Calculer la taille du tableau de manière plus sûre
    // size_t est un type entier non signé approprié pour les tailles
    size_t nombre_scores = sizeof(scores) / sizeof(scores[0]);
    printf("(Taille calculée du tableau 'scores' : %zu éléments)\n",
nombre_scores);

    for (size_t i = 0; i < nombre_scores; i++) { // Boucle de 0 à 4
        printf("  Score[%zu] = %d\n", i, scores[i]);
    }

    // Tentative (dangereuse) d'accès hors limites
    // printf("Accès à scores[5] (hors limites !) : %d\n", scores[5]); //
COMPORTEMENT INDÉFINI ! NE PAS FAIRE !

    return 0;
}
```

Les Chaînes de Caractères (Strings) comme tableaux de `char`

En C, il n'existe pas de type de données "string" intégré comme dans d'autres langages (Java, C++, Python...). Une chaîne de caractères est conventionnellement représentée par un **tableau de `char`** qui se termine par un caractère spécial : le **caractère nul** (null terminator), représenté par `\0`.

Ce caractère `\0` a une valeur numérique de 0 et est crucial car il marque la fin de la chaîne pour toutes les fonctions standard qui manipulent les chaînes (comme `printf` avec `%s`, `strlen`, `strcpy`, etc.).

Conséquence importante : Lorsque vous déclarez un tableau de `char` pour stocker une chaîne de `N` caractères utiles, vous devez prévoir une taille d'au moins `N + 1` pour le tableau, afin d'avoir de la place pour le caractère nul `\0` final.

Initialisation avec des littéraux de chaîne : Le moyen le plus simple et le plus courant d'initialiser un tableau de `char` comme une chaîne est d'utiliser un **littéral de chaîne** (texte entre guillemets doubles `"`). Le compilateur C ajoute **automatiquement** le caractère nul `\0` à la fin de la chaîne dans la mémoire allouée pour le tableau.

Exemples : `char salutation[] = "Bonjour";` // Le compilateur crée un tableau de 8 chars : {'B','o','n','j','o','u','r','\0'} // La taille est automatiquement déterminée (7 caractères + 1 pour \0).

`char nom[30] = "Alice";` // Le compilateur crée un tableau de 30 chars. // Il copie {'A','l','i','c','e','\0'} au début. // Les 24 caractères restants sont initialisés à 0 (caractère nul).

`char vide[10] = "";` // Crée un tableau de 10 chars, contenant uniquement {'\0'} au début, // le reste étant initialisé à 0. C'est une chaîne vide valide.

```
#include <stdio.h>
#include <string.h> // Inclure pour les fonctions de manipulation de
chaînes comme strlen()

int main() {
    char prenom[] = "David"; // Taille 6 ('D','a','v','i','d','\0')
    char ville[20] = "Paris"; // Taille 20, contient
'P','a','r','i','s','\0', ...

    printf("Le prénom est : %s\n", prenom); // %s lit les caractères
jusqu'au '\0'
    printf("La ville est : %s\n", ville);

    // Accéder aux caractères individuels
    printf("La première lettre du prénom est : %c\n", prenom[0]); // 'D'
    printf("La troisième lettre de la ville est : %c\n", ville[2]); // 'r'

    // Modifier un caractère dans la chaîne
    ville[0] = 'L'; // Remplace 'P' par 'L'
    printf("La ville modifiée est : %s\n", ville); // Affiche "Laris"

    // Afficher le caractère nul (sa valeur numérique est 0)
    printf("Le caractère à l'index 5 de prenom est (valeur ASCII) : %d\n",
prenom[5]); // Affiche 0

    // Utilisation de strlen() pour obtenir la longueur (nombre de
caractères AVANT le '\0')
    size_t longueur_prenom = strlen(prenom);
    size_t longueur_ville = strlen(ville);
    printf("La longueur de la chaîne '%s' est : %zu\n", prenom,
longueur_prenom); // Affiche 5
    printf("La longueur de la chaîne '%s' est : %zu\n", ville,
longueur_ville); // Affiche 5

    // Taille totale du tableau vs longueur de la chaîne
    printf("Taille totale du tableau 'ville' en mémoire : %zu octets\n",
sizeof(ville)); // Affiche 20
```

```
    return 0;
}
```

Tableaux Multidimensionnels

Le C permet de créer des tableaux ayant plus d'une dimension. Le cas le plus fréquent est le **tableau à deux dimensions**, qui peut être vu comme un **tableau de tableaux**. Ils sont idéaux pour représenter des structures de données tabulaires comme des grilles, des matrices, des plateaux de jeu, ou des images bitmap simples.

Déclaration : Pour déclarer un tableau à deux dimensions, vous spécifiez le type des éléments, le nom du tableau, le nombre de **lignes** (première dimension) et le nombre de **colonnes** (deuxième dimension), chacun entre crochets `[]`.

Syntaxe : `type nom_tableau[NB_LIGNES][NB_COLONNES];`

Par exemple, `int matrice[3][4];` déclare une matrice de 3 lignes et 4 colonnes d'entiers. `char plateau_jeu[8][8];` déclare un plateau de jeu 8x8.

En mémoire, les éléments sont généralement stockés ligne par ligne (row-major order). Pour `int matrice[3][4]`, les éléments seraient stockés dans cet ordre : `matrice[0][0]`, `matrice[0][1]`, `matrice[0][2]`, `matrice[0][3]`, puis `matrice[1][0]`, `matrice[1][1]`, ..., `matrice[2][3]`.

Initialisation : On utilise des accolades imbriquées : une paire d'accolades externe pour le tableau entier, et une paire interne pour chaque "ligne" (chaque sous-tableau).

(Voir l'exemple de code `code_tableau_2d_section9` pour l'illustration de l'initialisation).

On peut omettre les accolades internes, mais c'est moins clair. On peut aussi omettre la *première* dimension si on initialise, mais pas les suivantes (le compilateur la déduit).

Accès aux éléments : On utilise **deux paires de crochets**, l'une pour l'indice de la **ligne**, l'autre pour l'indice de la **colonne** (toujours en commençant à 0).

Syntaxe : `nom_tableau[indice_ligne][indice_colonne]`

Par exemple, `tableau2D[0][1] = 10;` modifie l'élément à la ligne 0, colonne 1. `char piece = plateau_jeu[7][4];` lit l'élément à la ligne 7, colonne 4.

Le parcours d'un tableau 2D se fait typiquement avec deux boucles `for` imbriquées.

(Voir l'exemple de code `code_tableau_2d_section9` pour une illustration pratique de l'accès et du parcours).

```c
#include <stdio.h>

int main() {
    // Déclare et initialise une matrice 3x4
    int matrice[3][4] = {
        { 10, 11, 12, 13 }, // Ligne 0
        { 20, 21, 22, 23 }, // Ligne 1
```

```
        { 30, 31, 32, 33 }  // Ligne 2
    };

    int nb_lignes = 3;
    int nb_colonnes = 4;

    printf("Contenu de la matrice %dx%d :\n", nb_lignes, nb_colonnes);

    // Parcours avec des boucles imbriquées
    // Boucle externe pour les lignes
    for (int i = 0; i < nb_lignes; i++) {
        // Boucle interne pour les colonnes
        for (int j = 0; j < nb_colonnes; j++) {
            // Accès à l'élément [ligne i][colonne j]
            printf("%4d ", matrice[i][j]); // %4d pour aligner sur 4
espaces
        }
        printf("\n"); // Saut de ligne après chaque ligne de la matrice
    }

    // Accès direct à un élément
    printf("\nÉlément à la ligne 1, colonne 2 : %d\n", matrice[1][2]); //
Affiche 22

    return 0;
}
```

On peut déclarer des tableaux avec plus de deux dimensions (ex: `int cube[5][10][3];`), mais ils deviennent rapidement complexes à gérer et à visualiser.

Les tableaux sont une structure de données fondamentale et très efficace en C pour stocker des collections d'éléments homogènes. Leur gestion demande cependant de la rigueur, notamment pour éviter les erreurs d'accès hors limites qui sont une source fréquente de bugs et de vulnérabilités en C.

---

10. Chaînes de Caractères

Contrairement à de nombreux langages modernes, le C n'a pas de type de données "string" intégré. À la place, une **chaîne de caractères** (string) est traitée comme un **tableau de caractères (`char`)** qui possède une convention spéciale : elle doit se terminer par un **caractère nul** (null terminator), représenté par la séquence d'échappement `\0`.

Ce caractère `\0` a une valeur numérique de 0 et il est fondamental. Il sert de marqueur de fin pour toutes les fonctions standard de la bibliothèque C qui manipulent des chaînes (comme `printf` avec `%s`, `strlen`, `strcpy`, etc.). Ces fonctions lisent ou écrivent les caractères du tableau jusqu'à ce qu'elles rencontrent ce `\0`.

Conséquence importante : Lorsque vous définissez un tableau de `char` pour stocker une chaîne pouvant contenir jusqu'à `N` caractères *utiles*, la taille réelle du tableau doit être d'au moins `N + 1` pour pouvoir stocker le caractère nul `\0` final. Oublier cet espace supplémentaire est une source fréquente d'erreurs (dépassement de tampon).

Initialisation des Chaînes

La manière la plus simple d'initialiser un tableau de `char` en tant que chaîne est d'utiliser un **littéral de chaîne** (texte entre guillemets doubles `"`). Le compilateur C s'occupe d'allouer l'espace nécessaire (y compris pour le `\0`) et d'ajouter automatiquement le caractère nul à la fin.

- Si la taille du tableau n'est pas spécifiée, le compilateur la calcule : `char message[] = "Bonjour";` // Crée un tableau de 8 chars: {'B','o','n','j','o','u','r','\0'}
- Si la taille est spécifiée et est suffisante : `char ville[10] = "Lyon";` // Crée un tableau de 10 chars: {'L','y','o','n','\0', 0, 0, 0, 0, 0} // Les caractères après '\0' sont initialisés à 0.
- Si la taille spécifiée est *exactement* celle de la chaîne (sans compter le `\0`), le `\0` n'est **pas** ajouté, ce qui n'est **pas** une chaîne C valide ! `char erreur[5] = "Salut"; // DANGEREUX ! Pas de place pour \0`
- Initialisation caractère par caractère (plus rare, nécessite le `\0` manuel) : `char mot[4] = {'O', 'u', 'i', '\0'};`

```c
#include <stdio.h>

int main() {
    // Initialisation avec taille implicite
    char salutation[] = "Hello"; // Taille 6 (H, e, l, l, o, \0)

    // Initialisation avec taille explicite (suffisante)
    char nom[20] = "Alice"; // Taille 20, contient "Alice\0" au début

    // Initialisation caractère par caractère (avec \0 manuel)
    char reponse[4] = {'O', 'K', '!', '\0'};

    // Chaîne vide (contient juste \0)
    char vide[] = ""; // Taille 1 (contient juste \0)

    printf("Salutation: %s (taille tableau: %zu)\n", salutation,
sizeof(salutation));
    printf("Nom: %s (taille tableau: %zu)\n", nom, sizeof(nom));
    printf("Réponse: %s (taille tableau: %zu)\n", reponse,
sizeof(reponse));
    printf("Chaîne vide: \"%s\" (taille tableau: %zu)\n", vide,
sizeof(vide));

    // Accès au dernier caractère utile (avant \0)
    // strlen vient de <string.h> (voir ci-dessous)
    #include <string.h>
    if (strlen(salutation) > 0) {
        printf("Dernier caractère de salutation: %c\n",
salutation[strlen(salutation) - 1]);
    }
```

```
    return 0;
}
```

La bibliothèque `<string.h>`

Le C fournit une bibliothèque standard, `<string.h>`, contenant de nombreuses fonctions essentielles pour manipuler ces chaînes de caractères (tableaux de `char` terminés par `\0`). Il est crucial d'inclure cet en-tête (`#include <string.h>`) pour les utiliser.

Voici quelques-unes des fonctions les plus importantes :

- **`strlen(const char *s)` :** Calcule la **longueur** de la chaîne `s` (le nombre de caractères *avant* le `\0` terminal). Retourne un `size_t`.
- **`strcpy(char *destination, const char *source)` : Copie** la chaîne `source` (y compris le `\0`) dans le tableau `destination`. **DANGEREUX !** Ne vérifie pas si `destination` est assez grand, risque majeur de dépassement de tampon.
- **`strncpy(char *destination, const char *source, size_t n)` :** Copie au plus `n` caractères de `source` vers `destination`. **Plus sûr**, mais attention : si `source` a `n` caractères ou plus, `destination` ne sera **pas** terminée par `\0` automatiquement ! Il faut souvent ajouter le `\0` manuellement : `destination[n-1] = '\0';` (si `n > 0`).
- **`strcat(char *destination, const char *source)` : Concatène** (ajoute) la chaîne `source` à la fin de la chaîne `destination`. Le premier caractère de `source` écrase le `\0` de `destination`, et un nouveau `\0` est ajouté à la toute fin. **DANGEREUX !** Ne vérifie pas si `destination` a assez de place pour contenir les deux chaînes + le `\0` final.
- **`strncat(char *destination, const char *source, size_t n)` :** Concatène au plus `n` caractères de `source` à `destination`. Ajoute toujours un `\0`. **Plus sûr**, mais il faut s'assurer que la taille totale de `destination` est suffisante (`strlen(destination) + n + 1`).
- **`strcmp(const char *s1, const char *s2)` : Compare** lexicographiquement (ordre alphabétique/ASCII) les chaînes `s1` et `s2`.
  - Retourne `0` si `s1` et `s2` sont identiques.
  - Retourne une valeur `< 0` si `s1` est "inférieure" à `s2`.
  - Retourne une valeur `> 0` si `s1` est "supérieure" à `s2`.
- **`strncmp(const char *s1, const char *s2, size_t n)` :** Compare au plus les `n` premiers caractères de `s1` et `s2`.

Règle d'or : Soyez **extrêmement prudent** avec `strcpy` et `strcat`. Préférez **toujours** leurs équivalents `strncpy` et `strncat` (ou mieux, `snprintf` pour la concaténation formatée) en gérant soigneusement les tailles et la terminaison par `\0` pour éviter les dépassements de tampon.

```
#include <stdio.h>
#include <string.h> // Pour strlen, strcpy, strncpy, strcmp, strcat,
strncat

int main() {
    char chaine1[50] = "Bonjour"; // Assez d'espace
    char chaine2[] = " le monde";
    char chaine_copie[50];
    char chaine_limitee[6];
```

```c
    // strlen : Longueur
    printf("Longueur de chaine1 \"%s\" : %zu\n", chaine1, strlen(chaine1));
// 7
    printf("Longueur de chaine2 \"%s\" : %zu\n", chaine2, strlen(chaine2));
// 9

    // strcpy : Copie (Attention : potentiellement dangereux)
    strcpy(chaine_copie, chaine1);
    printf("chaine_copie après strcpy : \"%s\"\n", chaine_copie);

    // strncpy : Copie limitée (plus sûr)
    // Copie au max 5 caractères de chaine2 dans chaine_limitee (taille 6)
    strncpy(chaine_limitee, chaine2, 5);
    // IMPORTANT: strncpy ne garantit pas le \0 si la source est >= n
    chaine_limitee[5] = '\0'; // Assurer la terminaison manuellement
    printf("chaine_limitee après strncpy(..., 5) : \"%s\"\n",
chaine_limitee); // " le m"

    // strcat : Concaténation (Attention : potentiellement dangereux)
    // Assurez-vous que chaine_copie a assez de place !
    // Ici, 50 est assez grand pour "Bonjour" (7) + " le monde" (9) + \0
(1) = 17
    strcat(chaine_copie, chaine2);
    printf("chaine_copie après strcat : \"%s\"\n", chaine_copie); //
"Bonjour le monde"
    printf("Nouvelle longueur de chaine_copie : %zu\n",
strlen(chaine_copie)); // 16

    // strncat : Concaténation limitée
    char chaine_concat_limitee[15] = "Test";
    // Ajoute au max 6 caractères de " concat" à "Test"
    // Taille nécessaire: strlen("Test") + 6 + 1 = 4 + 6 + 1 = 11. Le
tableau de 15 est OK.
    strncat(chaine_concat_limitee, " concat", 6);
    printf("chaine_concat_limitee après strncat(..., 6) : \"%s\"\n",
chaine_concat_limitee); // "Test concat"

    // strcmp : Comparaison
    char s1[] = "abc";
    char s2[] = "abd";
    char s3[] = "abc";
    printf("\nComparaisons :\n");
    printf("strcmp(s1, s2) = %d\n", strcmp(s1, s2)); // < 0 (car 'c' < 'd')
    printf("strcmp(s2, s1) = %d\n", strcmp(s2, s1)); // > 0
    printf("strcmp(s1, s3) = %d\n", strcmp(s1, s3)); // 0 (identiques)

    // strncmp : Comparaison limitée
    printf("strncmp(s1, s2, 2) = %d\n", strncmp(s1, s2, 2)); // 0 (les 2
premiers chars "ab" sont égaux)
    printf("strncmp(s1, s2, 3) = %d\n", strncmp(s1, s2, 3)); // < 0 (le
3ème char diffère)
```

```
    return 0;
}
```

La bibliothèque `<ctype.h>`

Cette bibliothèque fournit des fonctions utiles pour tester et manipuler des **caractères individuels**. Elle est souvent utilisée lors de l'analyse de chaînes ou de la validation d'entrées. Incluez `#include <ctype.h>`.

Quelques fonctions courantes (`c` est un `int` contenant la valeur d'un caractère) :

- **Tests (retournent vrai/non-zéro ou faux/zéro) :**
  - `isdigit(c)` : Est-ce un chiffre ('0'-'9') ?
  - `isalpha(c)` : Est-ce une lettre alphabétique (a-z, A-Z) ?
  - `isalnum(c)` : Est-ce une lettre ou un chiffre ?
  - `islower(c)` : Est-ce une lettre minuscule ?
  - `isupper(c)` : Est-ce une lettre majuscule ?
  - `isspace(c)` : Est-ce un caractère d'espacement (espace, `\t`, `\n`, `\r`, `\f`, `\v`) ?
  - `ispunct(c)` : Est-ce un caractère de ponctuation ?
  - `isprint(c)` : Est-ce un caractère imprimable (y compris l'espace) ?
  - `iscntrl(c)` : Est-ce un caractère de contrôle (non imprimable, comme `\n`, `\t`) ?
- **Conversions (retournent la valeur `int` du caractère converti) :**
  - `tolower(c)` : Convertit `c` en minuscule (si c'était une majuscule, sinon retourne `c`).
  - `toupper(c)` : Convertit `c` en majuscule (si c'était une minuscule, sinon retourne `c`).

```c
#include <stdio.h>
#include <ctype.h> // Pour les fonctions de test/conversion de caractères

int main() {
    char test_char1 = 'a';
    char test_char2 = '7';
    char test_char3 = ' ';
    char test_char4 = '$';

    printf("Tests pour '%c':\n", test_char1);
    printf("  isalpha? %d\n", isalpha(test_char1)); // 1 (vrai)
    printf("  isdigit? %d\n", isdigit(test_char1)); // 0 (faux)
    printf("  islower? %d\n", islower(test_char1)); // 1 (vrai)
    printf("  toupper : %c\n", toupper(test_char1)); // 'A'

    printf("\nTests pour '%c':\n", test_char2);
    printf("  isalpha? %d\n", isalpha(test_char2)); // 0
    printf("  isdigit? %d\n", isdigit(test_char2)); // 1
    printf("  isalnum? %d\n", isalnum(test_char2)); // 1

    printf("\nTests pour '%c' (espace):\n", test_char3);
    printf("  isspace? %d\n", isspace(test_char3)); // 1
    printf("  isprint? %d\n", isprint(test_char3)); // 1

    printf("\nTests pour '%c':\n", test_char4);
    printf("  ispunct? %d\n", ispunct(test_char4)); // 1
```

```
    printf("  isalnum? %d\n", isalnum(test_char4)); // 0

    return 0;
}
```

Formatage de chaînes (`sprintf`, `snprintf`)

Parfois, au lieu d'afficher directement sur la console avec `printf`, vous voulez **créer une chaîne de caractères formatée** en mémoire, en combinant du texte et des valeurs de variables. Les fonctions `sprintf` et `snprintf` (de `<stdio.h>`) servent à cela.

- **`int sprintf(char *str, const char *format, ...);`**
    - Fonctionne comme `printf`, mais au lieu d'afficher sur `stdout`, elle écrit la chaîne formatée résultante dans le tableau de `char` pointé par `str`.
    - Ajoute automatiquement le caractère nul `\0` à la fin de la chaîne générée.
    - **EXTRÊMEMENT DANGEREUX !** `sprintf` ne sait pas quelle est la taille du buffer `str`. Si la chaîne formatée est plus longue que l'espace alloué pour `str`, un **dépassement de tampon** se produira. **Évitez `sprintf` !**
- **`int snprintf(char *str, size_t size, const char *format, ...);` (C99+)**
    - La version **sûre** de `sprintf`.
    - `str` : Le buffer de destination.
    - `size` : La **taille totale** du buffer `str` (y compris l'espace pour le `\0`). `snprintf` garantit de ne **jamais** écrire au-delà de `size` octets dans `str`.
    - `format` et `...` : Comme pour `printf`.
    - **Comportement :** Écrit la chaîne formatée dans `str`. Si la chaîne générée (y compris le `\0`) est plus courte ou égale à `size`, elle est écrite entièrement et terminée par `\0`. Si elle est plus longue que `size`, elle est **tronquée** pour tenir dans `size - 1` caractères, et un `\0` est ajouté à la fin (`str[size-1] = '\0';`).
    - **Valeur de retour :** Le nombre de caractères qui *auraient été écrits* si le buffer avait été assez grand (sans compter le `\0` final). Si la valeur retournée est `>= size`, cela signifie que la sortie a été tronquée.

Utilisez TOUJOURS `snprintf` au lieu de `sprintf` pour la sécurité.

```
#include <stdio.h>

int main() {
    char buffer[100]; // Un buffer pour stocker la chaîne formatée
    char buffer_petit[15];
    char nom[] = "Arthur";
    int age = 42;
    float score = 95.5f;

    // Créer une chaîne formatée dans 'buffer'
    int nb_caracteres_ecrits = snprintf(buffer, sizeof(buffer),
                                        "Utilisateur: %s, Age: %d, Score:
```

```
%.1f",
                                        nom, age, score);

    // sizeof(buffer) est 100 ici
    if (nb_caracteres_ecrits >= 0 && nb_caracteres_ecrits < sizeof(buffer))
{
        // Tout s'est bien passé, la chaîne complète est dans buffer
        printf("Chaîne générée (OK) : %s\n", buffer);
        printf("(%d caractères écrits)\n", nb_caracteres_ecrits);
    } else if (nb_caracteres_ecrits >= sizeof(buffer)) {
        // La sortie a été tronquée
        printf("Chaîne générée (TRONQUÉE) : %s\n", buffer);
        printf("(Aurait dû écrire %d caractères, buffer trop petit)\n",
nb_caracteres_ecrits);
    } else {
        // Une erreur d'encodage s'est produite
        printf("Erreur lors de snprintf.\n");
    }

    // Exemple avec buffer trop petit
    printf("\nTentative avec buffer_petit (taille %zu):\n",
sizeof(buffer_petit));
    nb_caracteres_ecrits = snprintf(buffer_petit, sizeof(buffer_petit),
                                    "Nom: %s, Age: %d", nom, age);

    if (nb_caracteres_ecrits >= sizeof(buffer_petit)) {
        printf("Chaîne générée (TRONQUÉE) : \"%s\"\n", buffer_petit);
        printf("(Aurait dû écrire %d caractères)\n", nb_caracteres_ecrits);
    } else if (nb_caracteres_ecrits >= 0) {
         printf("Chaîne générée (OK) : %s\n", buffer_petit);
         printf("(%d caractères écrits)\n", nb_caracteres_ecrits);
    } else {
         printf("Erreur snprintf.\n");
    }


    return 0;
}
```

Lecture de chaînes sécurisée (`fgets`)

Comme mentionné, `scanf("%s", ...)` est dangereux car il ne limite pas la quantité de données lues, risquant des dépassements de tampon.

La fonction **`fgets(char *str, int size, FILE *stream)`** est l'alternative **recommandée** pour lire des chaînes de manière sûre, en particulier depuis l'entrée utilisateur (`stdin`).

- `str` : Le tableau de `char` (buffer) où stocker la chaîne lue.
- `size` : La **taille maximale** du buffer `str` (y compris l'espace pour `\0`). `fgets` ne lira jamais plus de `size - 1` caractères.
- `stream` : Le flux d'entrée d'où lire (par exemple, `stdin` pour le clavier).
- **Comportement :**

    - Lit les caractères depuis `stream` jusqu'à rencontrer un caractère de nouvelle ligne (`\n`), ou jusqu'à ce que `size - 1` caractères aient été lus, ou que la fin du fichier (`EOF`) soit atteinte.
    - Stocke les caractères lus dans `str`.
    - Si un `\n` est lu, il **est stocké** dans `str` (contrairement à `scanf("%s")`).
    - Ajoute **toujours** un caractère nul `\0` à la fin de la chaîne stockée dans `str`.
- **Valeur de retour :**
    - Retourne `str` (le pointeur vers le buffer) en cas de succès.
    - Retourne `NULL` si la fin du fichier est atteinte *avant* d'avoir lu le moindre caractère, ou si une erreur de lecture se produit.

Traitement du `\n` final : Puisque `fgets` inclut le `\n` final s'il y a la place, il est souvent nécessaire de le supprimer manuellement si on ne le veut pas dans la chaîne résultante.

```c
#include <stdio.h>
#include <string.h> // Pour strlen et strcspn

int main() {
    char nom_complet[100]; // Buffer pour stocker le nom

    printf("Entrez votre nom complet : ");

    // Lire au maximum 99 caractères (+1 pour \0) depuis stdin
    if (fgets(nom_complet, sizeof(nom_complet), stdin) != NULL) {
        // fgets a réussi à lire quelque chose

        // Optionnel : Supprimer le '\n' final si présent
        // strcspn trouve l'index du premier caractère parmi "\n"
        nom_complet[strcspn(nom_complet, "\n")] = '\0';

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

    } else {
        // Erreur de lecture ou fin de fichier (ex: Ctrl+D)
        if (feof(stdin)) {
            printf("\nFin de l'entrée détectée.\n");
        } else {
            perror("Erreur lors de la lecture avec fgets");
        }
    }

    return 0;
}
```

La gestion correcte et sécurisée des chaînes de caractères est un aspect fondamental de la programmation en C. Comprendre la convention du caractère nul `\0` et utiliser les fonctions des bibliothèques `<string.h>` et `<ctype.h>` (en privilégiant les versions sûres comme `strncpy`, `strncat`, `snprintf`, et `fgets`) est essentiel pour éviter les bugs et les failles de sécurité.

---

11. Les Pointeurs

Les pointeurs sont l'une des caractéristiques les plus puissantes, distinctives, et parfois redoutées du langage C. Ils permettent une manipulation directe de la **mémoire** de l'ordinateur, ce qui est essentiel pour de nombreuses tâches avancées comme l'allocation dynamique de mémoire, la création de structures de données complexes (listes chaînées, arbres), le passage d'arguments par "référence" (simulé en C par le passage d'adresse), et l'optimisation des performances en évitant des copies de données volumineuses.

Cependant, cette puissance vient avec une responsabilité : une mauvaise utilisation des pointeurs est une source fréquente de bugs difficiles à tracer (plantages, corruption de données, failles de sécurité). Une compréhension précise de leur fonctionnement est donc cruciale.

Qu'est-ce qu'un pointeur ? Adresses mémoire.

Comme nous l'avons vu, chaque variable que vous déclarez est stockée quelque part dans la mémoire vive (RAM). Cet emplacement est identifié par une **adresse mémoire** unique (souvent représentée comme un nombre hexadécimal).

Un **pointeur** est simplement un type spécial de variable dont la **valeur** n'est pas une donnée ordinaire (comme un `int` ou un `float`), mais une **adresse mémoire**. Au lieu de contenir la donnée elle-même, un pointeur "pointe vers" l'endroit en mémoire où se trouve une *autre* variable (ou le début d'un bloc de mémoire).

Analogie :

- Une variable normale (`int age = 30;`) est comme une maison contenant directement l'information (le nombre 30).
- Un pointeur (`int *ptr_age = &age;`) est comme une note sur un papier contenant l'*adresse* de la maison. La note elle-même n'est pas la maison, mais elle vous indique où la trouver.

Opérateurs `&` (adresse de) et `*` (déréférencement)

Le C fournit deux opérateurs fondamentaux pour travailler avec les pointeurs et les adresses :

1. **L'opérateur `&` (Adresse de - Address-of) :**
    - C'est un opérateur *unaire* (il agit sur un seul opérande situé à sa droite).
    - Lorsqu'il est placé devant le nom d'une variable (ex: `&maVariable`), il **retourne l'adresse mémoire** où cette variable est stockée.
    - Le type du résultat est un *pointeur vers le type de la variable*. Si `age` est un `int`, alors `&age` est de type `int *` (prononcé "pointeur vers int" ou "int étoile").
2. **L'opérateur `*` (Déréférencement / Indirection - Dereference) :**
    - C'est aussi un opérateur *unaire* lorsqu'il est utilisé avec un **pointeur** (ne pas confondre avec l'opérateur de multiplication `*` qui est binaire).
    - Lorsqu'il est placé devant une variable de type pointeur (ex: `*monPointeur`), il **accède à la valeur qui est stockée *à l'adresse mémoire contenue dans le pointeur***. Il "suit" le pointeur pour lire ou modifier la donnée pointée. On dit qu'on "déréférence" le pointeur.

Exemple fondamental :

```c
#include <stdio.h>

int main() {
    int age = 30;          // Une variable entière ordinaire
    int *pointeurSurAge;   // Déclaration d'un pointeur vers un entier
('int *')

    // 1. Obtenir l'adresse de 'age' avec '&'
    //    et stocker cette adresse dans la variable pointeur
'pointeurSurAge'.
    pointeurSurAge = &age;

    printf("Valeur de la variable 'age'         : %d\n", age);
    // Utiliser %p pour afficher une adresse mémoire (pointeur).
    // Il est recommandé de caster en (void*) pour printf.
    printf("Adresse mémoire de 'age' (&age)     : %p\n", (void*)&age);
    printf("Valeur de 'pointeurSurAge' (adresse) : %p\n",
(void*)pointeurSurAge);
    printf("Adresse mémoire du pointeur lui-même: %p\n",
(void*)&pointeurSurAge);

    // 2. Accéder à la valeur pointée par 'pointeurSurAge' avec '*'
(déréférencement)
    int valeur_pointee = *pointeurSurAge; // Lit la valeur à l'adresse
stockée dans pointeurSurAge
    printf("\nValeur obtenue via *pointeurSurAge : %d\n", valeur_pointee);
// Doit afficher 30

    // 3. Modifier la valeur originale 'age' VIA le pointeur
    printf("\nModification de la valeur pointée...\n");
    *pointeurSurAge = 31; // Écrit la valeur 31 à l'adresse mémoire
contenue dans pointeurSurAge
                          // Cela modifie donc la variable 'age' originale
!

    printf("Nouvelle valeur de 'age' (direct)  : %d\n", age); // age vaut
maintenant 31 !
    printf("Nouvelle valeur via *pointeurSurAge: %d\n", *pointeurSurAge);
// Affiche aussi 31

    return 0;
}
```

Déclaration et Initialisation de pointeurs

Pour déclarer une variable pointeur, vous spécifiez le **type de données que le pointeur est censé pointer**, suivi d'un astérisque `*` (qui fait partie de la déclaration du pointeur, pas du type) et du nom de la variable pointeur.

Syntaxe : `type *nom_pointeur;`

Le `type` est crucial :

- Il indique au compilateur **combien d'octets lire ou écrire** lorsqu'on déréférence le pointeur (`*nom_pointeur`). Un `int *` lira/écrira typiquement 4 octets, tandis qu'un `char *` lira/écrira 1 octet.
- Il permet l'**arithmétique des pointeurs** correcte (voir plus loin).
- Il permet au compilateur de faire certaines vérifications de type.

Exemples de déclarations : `int *ptr_entier;` // Pointeur vers un entier (int) `float *ptr_flottant;` // Pointeur vers un flottant (float) `char *ptr_caractere;` // Pointeur vers un caractère (char) `double *ptr_double;` // Pointeur vers un double `struct Personne *ptr_personne;` // Pointeur vers une structure Personne (si définie)

Note sur la syntaxe : L'astérisque `*` peut être collé au type (`int* ptr;`) ou au nom (`int *ptr;`) ou être au milieu (`int * ptr;`). Le style `int *ptr;` est souvent préféré car il souligne que `ptr` est un pointeur vers `int`. Attention lors de déclarations multiples sur une même ligne : `int *p1, p2;` déclare `p1` comme un pointeur vers int, mais `p2` comme un **int ordinaire** ! Pour déclarer deux pointeurs, il faut répéter l'astérisque : `int *p1, *p2;`.

Initialisation :

Il est **crucial** d'initialiser un pointeur avant de le déréférencer (`*`). Un pointeur non initialisé contient une adresse mémoire aléatoire ("garbage") et tenter d'y lire ou d'y écrire mène à un comportement indéfini (souvent un crash).

Principales façons d'initialiser un pointeur :

1. **Avec l'adresse d'une variable existante** en utilisant l'opérateur `&` : `int score = 100; int *ptrScore = &score; // ptrScore contient maintenant l'adresse de score`
2. **Avec la valeur `NULL`** : `NULL` est une constante spéciale (définie dans `<stdio.h>`, `<stdlib.h>`, `<stddef.h>` et d'autres) qui représente une adresse mémoire **non valide** ou "nulle" (souvent, mais pas toujours, l'adresse 0). Elle indique que le pointeur ne pointe vers rien d'utile pour le moment. C'est la valeur d'initialisation sûre par défaut si vous n'avez pas encore d'adresse valide à lui assigner. `float *ptrTemp = NULL;`
3. **Avec l'adresse retournée par une fonction d'allocation mémoire** (comme `malloc`, voir Section 12).
4. **Avec l'adresse d'un autre pointeur compatible** (copie d'adresse). `int *ptrAutreScore = ptrScore; // ptrAutreScore pointe maintenant vers la même chose que ptrScore`

```
#include <stdio.h>
#include <stdlib.h> // Pour NULL

int main() {
    int score = 100;
    float temperature = 25.5f;
    char grade = 'A';

    // Déclaration et initialisation en une ligne avec l'adresse d'une
variable
```

```c
    int *ptrScore = &score;
    float *ptrTemp = &temperature;
    char *ptrGrade = &grade;

    // Déclaration puis initialisation
    int *ptrAutreScore; // Pointeur non initialisé (contient une adresse
aléatoire)
    // NE PAS FAIRE : printf("%d", *ptrAutreScore); // CRASH probable !
    ptrAutreScore = &score; // Maintenant il pointe vers 'score'

    // Pointeur initialisé à NULL (sûr par défaut)
    int *ptrInvalide = NULL;
    // NE PAS FAIRE : *ptrInvalide = 5; // CRASH garanti (accès à l'adresse
0)

    // Affichage des valeurs via les pointeurs valides
    printf("Score via ptrScore : %d\n", *ptrScore);
    printf("Temperature via ptrTemp : %.1f\n", *ptrTemp);
    printf("Grade via ptrGrade : %c\n", *ptrGrade);
    printf("Score via ptrAutreScore : %d\n", *ptrAutreScore);

    // Vérification avant d'utiliser un pointeur potentiellement NULL
    if (ptrInvalide != NULL) {
        printf("Valeur pointée par ptrInvalide : %d\n", *ptrInvalide);
    } else {
        printf("ptrInvalide est NULL, impossible de le déréférencer.\n");
    }

    // On peut réassigner un pointeur pour qu'il pointe ailleurs
    int autre_score = 200;
    ptrScore = &autre_score; // ptrScore pointe maintenant vers autre_score
    printf("\nAprès réassignation de ptrScore:\n");
    printf("Score pointé par ptrScore : %d\n", *ptrScore); // Affiche 200
    printf("Valeur de 'score' original : %d\n", score); // Toujours 100

    return 0;
}
```

Pointeurs `NULL`

Comme vu précédemment, `NULL` est une macro constante définie dans les en-têtes standards (comme `<stdlib.h>`, `<stdio.h>`) qui représente une adresse mémoire garantie comme étant **non valide** pour pointer vers un objet ou une fonction. C'est la valeur standard pour indiquer qu'un pointeur "ne pointe sur rien".

Utilisations courantes de `NULL` :

- Initialiser des pointeurs dont la cible n'est pas encore connue (c'est une bonne pratique pour éviter d'avoir des pointeurs avec des adresses aléatoires).
- Marquer la fin d'une structure de données (ex: le pointeur `suivant` du dernier nœud d'une liste chaînée).

- Comme valeur de retour pour des fonctions qui allouent de la mémoire (`malloc`, `calloc`, `realloc`) ou cherchent quelque chose, afin d'indiquer un échec ou l'absence de résultat.

ATTENTION : Déréférencer un pointeur `NULL` Tenter d'accéder à la mémoire via un pointeur `NULL` (c'est-à-dire faire `*ptr` ou `ptr->membre` lorsque `ptr` vaut `NULL`) est une erreur très grave qui conduit à un **comportement indéfini**. Sur la plupart des systèmes modernes, cela provoque un **plantage immédiat** du programme (souvent une "Erreur de segmentation" / "Segmentation fault" ou "Access violation").

Règle de sécurité fondamentale : Avant de déréférencer un pointeur, **vérifiez toujours s'il est différent de `NULL`**, à moins que vous ne soyez absolument certain (par la logique de votre code) qu'il pointe vers une adresse valide.

L'idiome courant est : `if (monPointeur != NULL) { /* Utiliser *monPointeur ici */ } else { /* Gérer le cas NULL */ }`

Appeler `free(NULL)` est sûr et défini par le standard comme ne faisant rien.

```c
#include <stdio.h>
#include <stdlib.h> // Pour NULL et malloc/free

int main() {
    int *ptr1 = NULL; // Pointeur initialisé à NULL
    int *ptr2 = NULL;
    int valeur = 50;

    // Tentative d'allocation mémoire (peut échouer et retourner NULL)
    // Allouer de la place pour 1 entier
    ptr2 = (int*)malloc(sizeof(int));

    // --- Vérification avant déréférencement ---

    // Pour ptr1 (qui est NULL)
    printf("Vérification de ptr1...\n");
    if (ptr1 != NULL) {
        printf("  ptr1 pointe vers : %d\n", *ptr1); // Cette ligne ne
s'exécutera pas
    } else {
        printf("  ptr1 est NULL. Impossible de déréférencer.\n");
    }

    // Pour ptr2 (qui vient de malloc)
    printf("\nVérification de ptr2 (après malloc)...\n");
    if (ptr2 != NULL) {
        // Allocation réussie, on peut utiliser le pointeur
        *ptr2 = 123; // Écrire dans la mémoire allouée
        printf("  ptr2 pointe vers : %d\n", *ptr2);
        free(ptr2); // Ne pas oublier de libérer la mémoire allouée !
        ptr2 = NULL; // Bonne pratique après free
    } else {
        // malloc a échoué
        printf("  Échec de l'allocation mémoire pour ptr2.\n");
    }
```

```c
    // Assignation d'une adresse valide
    ptr1 = &valeur;
    printf("\nVérification de ptr1 après assignation (&valeur)...\n");
     if (ptr1 != NULL) {
        printf("  ptr1 pointe maintenant vers : %d\n", *ptr1);
    } else {
        printf("  ptr1 est toujours NULL (ne devrait pas arriver).\n");
    }


    return 0;
}
```

Pointeurs et Tableaux

Il existe une relation très étroite et fondamentale entre les pointeurs et les tableaux en C. Comprendre cette relation est clé pour maîtriser le langage.

1. Le nom d'un tableau "se désintègre" (decays) en un pointeur : Dans la plupart des contextes où le nom d'un tableau est utilisé dans une expression (par exemple, lorsqu'il est assigné à un pointeur, passé à une fonction, ou utilisé avec l'opérateur `+`), le nom du tableau est automatiquement converti en un **pointeur constant vers le premier élément** du tableau.

Donc, si vous avez : `int monTableau[10]; int *ptr;`

L'assignation suivante est valide et fait pointer `ptr` vers le début du tableau : `ptr = monTableau;`

C'est strictement équivalent à : `ptr = &monTableau[0];`

Exceptions : Le nom du tableau ne se désintègre pas en pointeur lorsqu'il est l'opérande de :

- L'opérateur `sizeof` : `sizeof(monTableau)` donne la taille *totale* du tableau en octets (pas la taille d'un pointeur).
- L'opérateur `&` (adresse de) : `&monTableau` donne un pointeur vers le tableau *entier* (type `int (*)[10]`), pas seulement vers son premier élément.

2. Accès aux éléments via les pointeurs : Puisque le nom d'un tableau agit comme un pointeur vers son premier élément, et que les éléments sont contigus en mémoire, on peut utiliser l'**arithmétique des pointeurs** pour accéder aux éléments :

- `monTableau[0]` est équivalent à `*monTableau` (car `monTableau` pointe vers le premier élément).
- `monTableau[1]` est équivalent à `*(monTableau + 1)`.
- `monTableau[i]` est équivalent à `*(monTableau + i)`.

Inversement, si `ptr` pointe vers `monTableau[0]` (`ptr = monTableau;`), on peut utiliser la **notation crochet `[]` avec le pointeur** :

- `*ptr` est équivalent à `ptr[0]`.
- `*(ptr + 1)` est équivalent à `ptr[1]`.
- `*(ptr + i)` est équivalent à `ptr[i]`.

Cette équivalence entre `*(pointeur + indice)` et `pointeur[indice]` est une caractéristique centrale du C.

```c
#include <stdio.h>

int main() {
    int nombres[5] = {10, 20, 30, 40, 50};
    int *ptr_nombres; // Un pointeur vers un entier

    // Assignation : le nom du tableau se désintègre en pointeur vers le
1er élément
    ptr_nombres = nombres; // ptr_nombres pointe maintenant vers nombres[0]

    printf("Adresse du tableau (nombres)     : %p\n", (void*)nombres);
    printf("Adresse stockée dans ptr_nombres : %p\n", (void*)ptr_nombres);
    printf("Adresse du premier élément (&nombres[0]) : %p\n",
(void*)&nombres[0]);

    printf("\nAccès aux éléments de différentes manières :\n");

    // Accès au premier élément (indice 0)
    printf("nombres[0]   = %d\n", nombres[0]);
    printf("*nombres     = %d\n", *nombres); // Déréférencer le nom du
tableau
    printf("ptr_nombres[0]= %d\n", ptr_nombres[0]); // Notation crochet sur
pointeur
    printf("*ptr_nombres = %d\n", *ptr_nombres); // Déréférencer le
pointeur

    // Accès au troisième élément (indice 2)
    printf("\nAccès à l'élément d'indice 2 :\n");
    printf("nombres[2]        = %d\n", nombres[2]);
    printf("*(nombres + 2)    = %d\n", *(nombres + 2)); // Arithmétique sur
nom de tableau
    printf("ptr_nombres[2]    = %d\n", ptr_nombres[2]);
    printf("*(ptr_nombres + 2)= %d\n", *(ptr_nombres + 2)); // Arithmétique
sur pointeur

    // Parcourir le tableau avec un pointeur
    printf("\nParcours avec pointeur et arithmétique :\n");
    for (int i = 0; i < 5; i++) {
        printf("  Adresse %p contient la valeur %d\n",
               (void*)(ptr_nombres + i), *(ptr_nombres + i));
    }

     printf("\nParcours avec pointeur et notation crochet :\n");
    for (int i = 0; i < 5; i++) {
        printf("  ptr_nombres[%d] = %d\n", i, ptr_nombres[i]);
    }

    // Parcourir en modifiant le pointeur lui-même
    printf("\nParcours en modifiant le pointeur :\n");
```

```
    ptr_nombres = nombres; // Réinitialiser le pointeur au début
    for (int i = 0; i < 5; i++) {
        printf("  Valeur pointée : %d\n", *ptr_nombres);
        ptr_nombres++; // Fait pointer vers l'élément suivant !
    }
    // Attention: ptr_nombres pointe maintenant APRÈS la fin du tableau !

    return 0;
}
```

Passage par Adresse (Simulation du passage par référence)

Comme nous l'avons vu dans la section sur les fonctions, le C utilise par défaut le **passage par valeur**. Cela signifie qu'une fonction reçoit des *copies* des arguments et ne peut pas modifier les variables originales de la fonction appelante.

Les pointeurs permettent de contourner cette limitation en simulant ce qu'on appelle dans d'autres langages le **passage par référence**. L'idée est de passer non pas la valeur d'une variable, mais son **adresse mémoire** à la fonction.

1. **Dans la fonction appelante :** On utilise l'opérateur `&` pour obtenir l'adresse de la variable qu'on veut permettre de modifier. `maFonction(&maVariable);`
2. **Dans la définition de la fonction appelée :** Le paramètre correspondant est déclaré comme un **pointeur** du type approprié. `void maFonction(int *ptrVariable)`
3. **À l'intérieur de la fonction appelée :** Pour accéder ou modifier la *valeur originale* de la variable de l'appelant, on **déréférence** le pointeur reçu en paramètre. `*ptrVariable = nouvelleValeur; // Modifie la variable originale !`

Cette technique est essentielle lorsque :

- Une fonction doit modifier la valeur d'une variable de l'appelant (ex: fonction `swap` qui échange deux valeurs).
- Une fonction doit "retourner" plus d'une valeur (elle peut retourner une valeur via `return` et modifier d'autres valeurs via des pointeurs passés en arguments).
- On veut éviter de copier des structures de données volumineuses lors de l'appel de fonction (passer un pointeur vers la structure est beaucoup plus efficace que de copier toute la structure).

```
#include <stdio.h>

// Fonction qui échange les valeurs de deux entiers
// Elle reçoit les ADRESSES des variables à échanger
void echanger(int *adresse_a, int *adresse_b) {
    printf("  -> Entrée echanger: Adresse a=%p (*a=%d), Adresse b=%p
(*b=%d)\n",
           (void*)adresse_a, *adresse_a, (void*)adresse_b, *adresse_b);

    // On manipule les *valeurs pointées* par les adresses
    int temp = *adresse_a;  // 1. Stocke la valeur pointée par adresse_a
(valeur de x)
```

```c
    *adresse_a = *adresse_b; // 2. Met la valeur pointée par adresse_b
(valeur de y)
                             //    à l'adresse de a (modifie x !)
    *adresse_b = temp;      // 3. Met l'ancienne valeur de a (stockée dans
temp)
                             //    à l'adresse de b (modifie y !)

    printf("  -> Sortie echanger: Adresse a=%p (*a=%d), Adresse b=%p
(*b=%d)\n",
           (void*)adresse_a, *adresse_a, (void*)adresse_b, *adresse_b);
}

// Fonction qui essaie de retourner deux résultats via des pointeurs
void calculer_somme_et_produit(int x, int y, int *res_somme, int
*res_produit) {
    // Vérifier si les pointeurs de résultat sont valides avant de les
utiliser
    if (res_somme != NULL) {
        *res_somme = x + y; // Modifie la variable pointée par res_somme
dans main
    }
    if (res_produit != NULL) {
        *res_produit = x * y; // Modifie la variable pointée par
res_produit dans main
    }
}


int main() {
    int x = 10;
    int y = 20;

    printf("Avant échange : x = %d, y = %d\n", x, y);

    // On passe les ADRESSES de x et y en utilisant l'opérateur '&'
    echanger(&x, &y);

    printf("Après échange : x = %d, y = %d\n", x, y); // x et y ont bien
été échangées !

    printf("\nCalcul somme et produit :\n");
    int somme, produit;
    // On passe les adresses de 'somme' et 'produit' pour que la fonction
puisse les remplir
    calculer_somme_et_produit(5, 4, &somme, &produit);
    printf("Pour 5 et 4 : Somme = %d, Produit = %d\n", somme, produit);

    // On peut passer NULL si on ne veut pas d'un des résultats
    int somme_seule;
    calculer_somme_et_produit(7, 3, &somme_seule, NULL);
    printf("Pour 7 et 3 : Somme seule = %d\n", somme_seule);
```

```
    return 0;
}
```

Le qualificateur `const` avec les pointeurs

Le mot-clé `const` peut être utilisé de différentes manières avec les pointeurs pour indiquer ce qui ne peut pas être modifié. C'est très important pour écrire du code plus sûr et pour mieux exprimer les intentions (notamment dans les paramètres de fonction).

Il y a trois cas principaux à distinguer :

1. **Pointeur vers une constante (`const type *ptr` ou `type const *ptr`) :**
    - Le pointeur `ptr` peut être modifié pour pointer vers une autre adresse.
    - La **donnée pointée** par `ptr` **ne peut pas être modifiée** *via ce pointeur*.
    - Utilisation typique : Passer un pointeur à une fonction en garantissant que la fonction ne modifiera pas la donnée d'origine via ce pointeur.

```
const int x = 10;
const int y = 20;
const int *p = &x; // p pointe vers x (qui est const)
// *p = 15; // ERREUR: on ne peut pas modifier la valeur via p
p = &y;    // OK: on peut faire pointer p ailleurs
```

2. **Pointeur constant (`type * const ptr`) :**
    - Le pointeur `ptr` **doit être initialisé** lors de sa déclaration et **ne peut plus être modifié** pour pointer vers une autre adresse.
    - La **donnée pointée** par `ptr` **peut être modifiée** via ce pointeur (si la donnée elle-même n'est pas `const`).
    - Moins courant, utilisé quand l'adresse elle-même ne doit pas changer.

```
int x = 10;
int y = 20;
int * const p = &x; // p pointe vers x et ne pourra plus pointer
ailleurs
*p = 15; // OK: on peut modifier la valeur de x via p
// p = &y;    // ERREUR: on ne peut pas changer où p pointe
```

3. **Pointeur constant vers une constante (`const type * const ptr` ou `type const * const ptr`) :**
    - Combinaison des deux : ni le pointeur `ptr` ni la donnée pointée ne peuvent être modifiés via ce pointeur.
    - Le pointeur doit être initialisé.

```
```

```c
int x = 10;
int y = 20;
const int * const p = &x; // p pointe vers x et ne bougera plus
// *p = 15; // ERREUR: on ne peut pas modifier la valeur via p
// p = &y;    // ERREUR: on ne peut pas changer où p pointe
```

Règle mnémotechnique : Lire la déclaration de droite à gauche. "ptr is a const pointer to int" (`int * const ptr`), "ptr is a pointer to const int" (`const int * ptr`).

Utiliser `const` correctement avec les pointeurs, surtout pour les paramètres de fonction, est une excellente pratique pour améliorer la sécurité et la clarté du code.

```c
#include <stdio.h>

int main() {
    int val1 = 10;
    int val2 = 20;
    const int val_const = 100;

    // --- 1. Pointeur vers une constante ---
    const int *ptr_vers_const; // ou int const *ptr_vers_const;
    ptr_vers_const = &val1; // OK: Pointeur peut pointer vers une variable
non-const
    // *ptr_vers_const = 15; // ERREUR: Impossible de modifier la valeur
via ce pointeur
    printf("Valeur pointée par ptr_vers_const: %d\n", *ptr_vers_const);

    ptr_vers_const = &val_const; // OK: Peut pointer vers une variable
const
    printf("Valeur pointée par ptr_vers_const (après changement): %d\n",
*ptr_vers_const);

    ptr_vers_const = &val2; // OK: Le pointeur lui-même peut changer de
cible
    printf("Valeur pointée par ptr_vers_const (encore après changement):
%d\n", *ptr_vers_const);

    // --- 2. Pointeur constant ---
    int * const ptr_const = &val1; // DOIT être initialisé ici
    printf("\nValeur pointée par ptr_const: %d\n", *ptr_const);

    *ptr_const = 11; // OK: On peut modifier la valeur pointée
    printf("Nouvelle valeur pointée par ptr_const: %d\n", *ptr_const);
    printf("Nouvelle valeur de val1: %d\n", val1);

    // ptr_const = &val2; // ERREUR: Le pointeur lui-même est constant, ne
peut pas changer de cible

    // --- 3. Pointeur constant vers une constante ---
    const int * const ptr_const_vers_const = &val_const; // Doit être
```

```
initialisé
    // Ou: const int * const ptr_const_vers_const = &val1; // Peut pointer
vers non-const

    printf("\nValeur pointée par ptr_const_vers_const: %d\n",
*ptr_const_vers_const);

    // *ptr_const_vers_const = 150; // ERREUR: Impossible de modifier la
valeur pointée
    // ptr_const_vers_const = &val2; // ERREUR: Impossible de changer la
cible du pointeur

    return 0;
}
```

Arithmétique des pointeurs

Le C permet d'effectuer certaines opérations arithmétiques sur les pointeurs, ce qui est particulièrement utile pour parcourir des tableaux ou des blocs de mémoire alloués dynamiquement.

Opérations autorisées :

- **Addition/Soustraction d'un entier :** Ajouter ou soustraire un entier `n` à un pointeur `ptr` (`ptr + n`, `ptr - n`) déplace le pointeur pour qu'il pointe `n` *éléments* plus loin ou plus tôt en mémoire.
- **Incrémentation/Décrémentation :** `ptr++`, `++ptr`, `ptr--`, `--ptr` déplacent le pointeur vers l'élément suivant ou précédent.
- **Soustraction de deux pointeurs :** Si `ptr1` et `ptr2` pointent vers des éléments du **même tableau** (ou bloc alloué), `ptr2 - ptr1` donne le **nombre d'éléments** (pas d'octets !) entre les deux pointeurs (le résultat est de type `ptrdiff_t`, un type entier signé défini dans `<stddef.h>`).

Point crucial : Mise à l'échelle automatique (`sizeof`) L'arithmétique des pointeurs est **automatiquement mise à l'échelle** en fonction de la **taille du type de données pointé**.

- Si `ptr` est un `int *` et `sizeof(int)` vaut 4 octets :
  - `ptr + 1` ne signifie pas ajouter 1 à l'adresse brute, mais avancer l'adresse pour pointer vers l'**entier suivant**. L'adresse réelle augmentera de `1 * sizeof(int)` (soit 4 octets).
  - `ptr + 3` avancera l'adresse de `3 * sizeof(int)` (12 octets).
- Si `ptr` est un `char *` et `sizeof(char)` vaut 1 octet :
  - `ptr + 1` avancera l'adresse de `1 * sizeof(char)` (1 octet).
- Si `ptr` est un `struct Point *` et `sizeof(struct Point)` vaut 16 octets :
  - `ptr + 1` avancera l'adresse de `1 * sizeof(struct Point)` (16 octets).

Cette mise à l'échelle automatique rend l'arithmétique des pointeurs intuitive pour parcourir des tableaux de n'importe quel type.

```
#include <stdio.h>
#include <stddef.h> // Pour ptrdiff_t

int main() {
```

```c
    int nombres[] = {100, 200, 300, 400, 500};
    int *ptr1 = nombres; // Pointeur vers nombres[0]
    int *ptr2 = &nombres[3]; // Pointeur vers nombres[3] (valeur 400)

    printf("ptr1 pointe vers l'adresse %p (valeur %d)\n", (void*)ptr1,
*ptr1);
    printf("ptr2 pointe vers l'adresse %p (valeur %d)\n", (void*)ptr2,
*ptr2);

    // Addition d'un entier
    int *ptr_plus_2 = ptr1 + 2; // Pointe vers nombres[2] (valeur 300)
    printf("\nptr1 + 2 pointe vers l'adresse %p (valeur %d)\n",
(void*)ptr_plus_2, *ptr_plus_2);

    // Incrémentation
    ptr1++; // ptr1 pointe maintenant vers nombres[1] (valeur 200)
    printf("Après ptr1++, ptr1 pointe vers %p (valeur %d)\n", (void*)ptr1,
*ptr1);

    // Soustraction de pointeurs (doivent pointer dans le même
tableau/bloc)
    ptrdiff_t difference = ptr2 - ptr1; // ptr2 pointe vers indice 3, ptr1
vers indice 1
    printf("Différence (ptr2 - ptr1) : %td éléments\n", difference); //
Affiche 2 (%td pour ptrdiff_t)

    // Comparaison de pointeurs (utile pour les boucles)
    printf("\nParcours jusqu'à ptr2 inclus:\n");
    int *p_courant = nombres;
    while (p_courant <= ptr2) {
        printf("  Adresse %p, Valeur %d\n", (void*)p_courant, *p_courant);
        p_courant++;
    }

    return 0;
}
```

Attention : L'arithmétique des pointeurs n'est définie que si les pointeurs restent à l'intérieur des limites du tableau (ou du bloc alloué) ou pointent juste **un élément après** la fin. Tenter d'accéder à la mémoire via un pointeur qui a été déplacé en dehors de ces limites (sauf pour la comparaison avec le pointeur "un après la fin") conduit à un comportement indéfini.

Les pointeurs sont un concept riche et puissant en C. Leur maîtrise, y compris l'arithmétique et l'interaction avec `const`, ouvre la porte à des techniques de programmation efficaces et de bas niveau, mais demande de la pratique et une grande attention pour éviter les erreurs liées à la mémoire.

---

12. Allocation Dynamique de Mémoire

Jusqu'à présent, la taille de nos variables et tableaux était fixée au moment de la compilation (par exemple, `int tableau[50];`). Cette mémoire est gérée de deux manières principales :

- **Allocation statique :** Pour les variables globales et les variables locales déclarées avec `static`. La mémoire est réservée par le compilateur pour toute la durée d'exécution du programme.
- **Allocation automatique :** Pour les variables locales non-`static` (y compris les paramètres de fonction). La mémoire est allouée sur la **pile d'exécution (stack)** lorsque la fonction ou le bloc est entré, et automatiquement libérée lorsque l'exécution quitte la fonction ou le bloc.

Ces méthodes sont simples mais limitées : la taille doit être connue à la compilation. Que faire si la taille nécessaire dépend d'une entrée de l'utilisateur, de données lues dans un fichier, ou si elle doit changer pendant l'exécution ?

C'est là qu'intervient l'**allocation dynamique de mémoire**. Elle permet de demander de la mémoire au système *pendant que le programme s'exécute*. Cette mémoire est allouée depuis une zone différente de la pile, appelée le **tas (heap)**.

Points clés de l'allocation dynamique :

- **Flexibilité :** La taille de la mémoire peut être déterminée à l'exécution.
- **Contrôle manuel :** Le programmeur est **responsable** de demander explicitement la mémoire (`malloc`, `calloc`, `realloc`) et, surtout, de la **libérer explicitement** (`free`) lorsqu'elle n'est plus nécessaire.
- **Durée de vie :** La mémoire allouée sur le tas persiste jusqu'à ce qu'elle soit explicitement libérée avec `free()`. Sa durée de vie n'est pas liée à la portée de la fonction où elle a été allouée.

Les fonctions pour l'allocation dynamique se trouvent principalement dans l'en-tête `<stdlib.h>`.

Pourquoi l'allocation dynamique ?

- **Taille inconnue à la compilation :** Créer des tableaux ou structures dont la taille dépend d'informations disponibles uniquement à l'exécution.
- **Flexibilité de la taille :** Agrandir ou réduire la taille d'un bloc mémoire si les besoins changent (avec `realloc`).
- **Structures de données complexes :** Essentielle pour implémenter des structures dynamiques comme les listes chaînées, les arbres, les graphes, où les éléments (nœuds) sont ajoutés ou supprimés au fur et à mesure.
- **Gestion de la durée de vie :** Créer des données qui doivent survivre à la fonction qui les a créées.

La Pile vs Le Tas (Stack vs Heap)

- **Pile (Stack) :** Gérée automatiquement par le compilateur. Rapide pour allouer/désallouer. Utilisée pour les variables locales automatiques et les informations d'appel de fonction. Taille limitée. Allocation/désallocation LIFO (Last-In, First-Out).
- **Tas (Heap) :** Grande réserve de mémoire gérée manuellement par le programmeur via `malloc`/`calloc`/`realloc`/`free`. Plus lente que la pile. Permet une allocation de taille flexible et une durée de vie contrôlée. Risque de fragmentation et de fuites mémoire si mal gérée.

`malloc()` (Memory Allocation)

`malloc` est la fonction la plus fondamentale pour l'allocation dynamique. Elle permet d'allouer un bloc de mémoire contigu d'une taille spécifiée en octets sur le tas.

Syntaxe : `void* malloc(size_t size);`

- `size`: Le **nombre total d'octets** que vous souhaitez allouer. Il est très courant et **fortement recommandé** d'utiliser l'opérateur `sizeof` pour calculer cette taille de manière portable et correcte. Par exemple, pour allouer de l'espace pour `n` entiers, on utilisera `n * sizeof(int)`.
- **Valeur de retour :**
  - Si l'allocation réussit : `malloc` retourne un pointeur de type **`void*`** pointant vers le début du bloc de mémoire alloué sur le tas. Un `void*` est un pointeur générique qui peut pointer vers n'importe quel type de données, mais il ne peut pas être déréférencé directement. Vous devez le **caster** (convertir explicitement) vers le type de pointeur approprié avant de l'utiliser (par exemple, `(int*)`, `(char*)`, `(struct MaStruct*)`).
  - Si l'allocation échoue (manque de mémoire disponible sur le tas, `size` est 0, etc.) : `malloc` retourne **`NULL`**.
- **Contenu de la mémoire :** La mémoire allouée par `malloc` n'est **PAS initialisée**. Elle contient des données "poubelles" (les bits qui se trouvaient là précédemment). Vous devez initialiser cette mémoire vous-même avant de lire dedans ou de supposer qu'elle contient des zéros.
- **Vérification `NULL` :** Il est **absolument impératif** de toujours vérifier si le pointeur retourné par `malloc` est `NULL` avant d'essayer d'utiliser la mémoire. Tenter de déréférencer un pointeur `NULL` causera un plantage.

```c
#include <stdio.h>
#include <stdlib.h> // Pour malloc, free, NULL, EXIT_FAILURE

int main() {
    int *tableau_entiers = NULL; // Toujours initialiser les pointeurs à
NULL
    int nombre_elements;
    int i;

    printf("Combien d'entiers voulez-vous stocker dynamiquement ? ");
    if (scanf("%d", &nombre_elements) != 1 || nombre_elements <= 0) {
        fprintf(stderr, "Nombre invalide.\n");
        return EXIT_FAILURE; // Ou return 1;
    }
    // Vider le buffer d'entrée
    while(getchar() != '\n');

    // --- Allocation ---
    // Calculer la taille totale en octets : nombre_elements * taille d'un
int
    size_t taille_requise = nombre_elements * sizeof(int);
    printf("Demande d'allocation de %zu octets (%d entiers)...\n",
taille_requise, nombre_elements);

    // Allouer la mémoire et caster le void* retourné en int*
    tableau_entiers = (int*)malloc(taille_requise);

    // --- VÉRIFICATION CRUCIALE ---
    if (tableau_entiers == NULL) {
        fprintf(stderr, "Erreur: Impossible d'allouer la mémoire requise
```

```
    (malloc a retourné NULL).\n");
            perror("Cause possible"); // Affiche une description système de
    l'erreur (si disponible)
            return EXIT_FAILURE; // Quitter si l'allocation échoue
        }

        // --- Utilisation (si allocation réussie) ---
        printf("Mémoire allouée avec succès à l'adresse : %p\n",
    (void*)tableau_entiers);

        // Initialiser la mémoire (car malloc ne le fait pas)
        printf("Initialisation du tableau...\n");
        for (i = 0; i < nombre_elements; i++) {
            tableau_entiers[i] = i * 10; // On peut utiliser la notation
    crochet [] comme avec un tableau normal
        }

        // Afficher le contenu
        printf("Contenu du tableau alloué :\n");
        for (i = 0; i < nombre_elements; i++) {
            printf("  tableau_entiers[%d] = %d\n", i, tableau_entiers[i]);
        }

        // --- Libération (TRÈS IMPORTANT) ---
        printf("Libération de la mémoire allouée...\n");
        free(tableau_entiers); // Retourne la mémoire au système
        tableau_entiers = NULL; // Bonne pratique : mettre le pointeur à NULL
    après free
                                // pour éviter de l'utiliser accidentellement
    (dangling pointer)

        // Tenter d'accéder après free mènerait à un comportement indéfini !
        // if (tableau_entiers != NULL) {
        //    printf("%d\n", tableau_entiers[0]); // Dangereux !
        // }

        return 0; // Succès
    }
```

`calloc()` (Contiguous Allocation)

`calloc` est une alternative à `malloc`, particulièrement utile pour allouer de la mémoire pour des **tableaux**. Elle prend deux arguments : le nombre d'éléments et la taille de chaque élément. Sa caractéristique principale par rapport à `malloc` est qu'elle **initialise la mémoire allouée à zéro**.

Syntaxe : `void* calloc(size_t num_elements, size_t element_size);`

- `num_elements`: Le nombre d'éléments que vous souhaitez allouer (par exemple, le nombre d'entiers dans un tableau).
- `element_size`: La taille en octets de *chaque* élément (typiquement `sizeof(type)`). La taille totale allouée sera `num_elements * element_size`.

- **Valeur de retour :** Identique à `malloc` : un `void*` vers le bloc alloué en cas de succès, ou `NULL` en cas d'échec. Le cast vers le type de pointeur approprié et la vérification de `NULL` sont toujours nécessaires.
- **Contenu de la mémoire :** Tous les octets du bloc alloué sont **garantis d'être initialisés à 0**. Cela inclut la mise à zéro pour les types numériques, le caractère nul `\0` pour les `char`, et le pointeur `NULL` pour les types pointeurs (si la représentation interne de NULL est bien zéro, ce qui est le cas sur quasiment toutes les plateformes modernes).

`calloc` est légèrement plus lente que `malloc` à cause de l'étape d'initialisation à zéro, mais peut être plus pratique et plus sûre si vous avez besoin que la mémoire soit initialisée dès l'allocation. Elle peut aussi offrir une protection supplémentaire contre les dépassements d'entier lors du calcul de la taille totale si `num_elements * element_size` est très grand, bien que ce ne soit pas sa garantie principale.

```c
#include <stdio.h>
#include <stdlib.h> // Pour calloc, free, NULL, EXIT_FAILURE

typedef struct {
    int x;
    double y;
} Point;

int main() {
    Point *points = NULL;
    size_t nombre_points = 3;
    size_t i;

    // --- Allocation et Initialisation avec calloc ---
    printf("Allocation de mémoire pour %zu structures Point avec
calloc...\n", nombre_points);

    // Allouer de la mémoire pour 'nombre_points' structures Point
    // ET initialiser tous les octets à zéro.
    points = (Point*)calloc(nombre_points, sizeof(Point));

    // --- Vérification ---
    if (points == NULL) {
        fprintf(stderr, "Erreur: Impossible d'allouer la mémoire avec
calloc.\n");
        return EXIT_FAILURE;
    }

    printf("Mémoire allouée et initialisée à zéro avec calloc à l'adresse
%p\n", (void*)points);

    // --- Vérifier l'initialisation à zéro ---
    printf("Contenu initial (après calloc) :\n");
    for (i = 0; i < nombre_points; i++) {
        // %g est un format pratique pour les doubles
        printf("  points[%zu] = { x=%d, y=%g }\n", i, points[i].x,
points[i].y); // Doit afficher x=0, y=0
    }
```

```
    // --- Utilisation ---
    points[0].x = 10;
    points[0].y = 1.1;
    points[1].x = -5;
    points[1].y = -2.2;
    // points[2] reste {0, 0.0}

    printf("\nContenu après modification :\n");
     for (i = 0; i < nombre_points; i++) {
        printf("  points[%zu] = { x=%d, y=%g }\n", i, points[i].x,
points[i].y);
    }

    // --- Libération ---
    free(points);
    printf("\nMémoire libérée.\n");
    points = NULL;

    return 0;
}
```

`realloc()` (Re-allocation)

`realloc` est utilisée pour **modifier la taille** d'un bloc de mémoire qui a été précédemment alloué avec `malloc`, `calloc` ou `realloc` lui-même. Cela permet d'**agrandir** ou de **réduire** un bloc de mémoire existant sur le tas.

Syntaxe : `void* realloc(void *ptr, size_t new_size);`

- `ptr`: Le pointeur vers le début du bloc de mémoire **existant** que l'on souhaite redimensionner.
  - Si `ptr` est `NULL`, alors `realloc(NULL, new_size)` se comporte exactement comme `malloc(new_size)`.
  - Si `ptr` n'est *pas* `NULL`, il doit pointer vers un bloc précédemment retourné par `malloc`, `calloc` ou `realloc` et **non encore libéré** par `free`.
- `new_size`: La **nouvelle taille souhaitée** pour le bloc de mémoire, en octets.
  - Si `new_size` est 0 et `ptr` n'est pas `NULL`, alors `realloc(ptr, 0)` se comporte comme `free(ptr)` et retourne `NULL` (comportement standard depuis C99/C11, mais vérifier la documentation de votre compilateur pour les cas limites). Il est généralement plus clair d'appeler `free(ptr)` directement dans ce cas.
- **Valeur de retour :**
  - Si la réallocation réussit : `realloc` retourne un `void*` pointant vers le début du bloc de mémoire redimensionné. **Attention : l'adresse retournée peut être différente de l'adresse originale `ptr` !** Si le bloc ne peut pas être agrandi sur place (pas assez d'espace contigu après), `realloc` peut allouer un *nouveau* bloc plus grand ailleurs en mémoire, *copier* le contenu de l'ancien bloc vers le nouveau, et *libérer* l'ancien bloc automatiquement. Si le bloc est réduit, l'adresse retournée est généralement la même que `ptr`.
  - Si la réallocation échoue (pas assez de mémoire disponible sur le tas pour la `new_size`, par exemple) : `realloc` retourne **`NULL`**. **Important : Dans ce cas d'échec, le bloc de mémoire**

original pointé par `ptr` n'est PAS libéré et reste valide avec sa taille d'origine.

- **Contenu de la mémoire :**
  - Si le bloc est agrandi, le contenu original est préservé. La partie supplémentaire du bloc (au-delà de l'ancienne taille) n'est **pas initialisée** (contient des données poubelles, comme `malloc`).
  - Si le bloc est réduit, le contenu est préservé jusqu'à la `new_size`.

Prudence avec `realloc` : Le Pattern de Sécurité Ne jamais assigner directement le résultat de `realloc` au pointeur original sans passer par une variable temporaire. C'est parce que si `realloc` échoue et retourne `NULL`, vous perdriez la seule référence (`ptr`) vers votre bloc de mémoire original, créant ainsi une **fuite mémoire** (memory leak) car vous ne pourriez plus le libérer avec `free`.

Utilisez toujours ce pattern :

```
// ptr pointe vers un bloc alloué de taille 'ancienne_taille'
size_t nouvelle_taille = /* ... */;
void *temp = realloc(ptr, nouvelle_taille);

if (temp == NULL && nouvelle_taille > 0) { // Vérifier l'échec (et que ce
n'est pas realloc(ptr, 0))
    // Échec de realloc: 'ptr' est toujours valide avec 'ancienne_taille'
    fprintf(stderr, "Erreur: Echec de realloc\n");
    // Gérer l'erreur (ex: continuer avec l'ancien bloc, ou libérer et
quitter)
    // free(ptr); // Si on décide d'abandonner
    // exit(EXIT_FAILURE);
} else {
    // Succès (ou realloc(ptr, 0) qui a libéré ptr)
    // Mettre à jour le pointeur original pour pointer vers le nouveau bloc
    // (qui peut être à la même adresse ou une nouvelle)
    ptr = temp; // temp peut être NULL si nouvelle_taille était 0
    // Maintenant, on peut utiliser 'ptr' avec la 'nouvelle_taille' (si >
0)
}
```

```
#include <stdio.h>
#include <stdlib.h> // Pour malloc, realloc, free, NULL, EXIT_FAILURE
#include <string.h> // Pour memcpy (juste pour l'exemple d'initialisation)

int main() {
    int *nombres = NULL;
    size_t taille_actuelle = 3;
    size_t i;

    // 1. Allocation initiale avec malloc
    nombres = (int*)malloc(taille_actuelle * sizeof(int));
    if (nombres == NULL) {
        perror("malloc initial a échoué");
        return EXIT_FAILURE;
    }
```

```c
    printf("Allocation initiale (%zu entiers) à l'adresse %p\n",
taille_actuelle, (void*)nombres);
    for (i = 0; i < taille_actuelle; i++) {
        nombres[i] = (i + 1) * 10; // 10, 20, 30
    }
    printf("Contenu initial: ");
    for(i=0; i<taille_actuelle; ++i) printf("%d ", nombres[i]);
    printf("\n");

    // 2. Tentative de réallocation pour agrandir (ex: doubler la taille)
    size_t nouvelle_taille = taille_actuelle * 2;
    printf("\nTentative de réallocation pour %zu entiers...\n",
nouvelle_taille);

    // Utilisation du pattern sécurisé avec une variable temporaire
    int *temp = (int*)realloc(nombres, nouvelle_taille * sizeof(int));

    // 3. Vérification du résultat de realloc
    if (temp == NULL) {
        // Échec de realloc: 'nombres' est toujours valide avec
taille_actuelle !
        fprintf(stderr, "realloc a échoué! La mémoire originale est
toujours valide.\n");
        // On pourrait essayer de continuer avec l'ancien tableau,
        // ou ici on décide de libérer et quitter.
        free(nombres); // Libérer l'ancien bloc avant de quitter
        return EXIT_FAILURE;
    }

    // 4. Succès : Mettre à jour le pointeur original et la taille
    nombres = temp; // 'nombres' pointe maintenant vers le bloc
redimensionné (peut-être déplacé)
    taille_actuelle = nouvelle_taille; // Mettre à jour la taille connue
    printf("Réallocation réussie. Nouvelle adresse (peut être différente) :
%p\n", (void*)nombres);

    // 5. Initialiser la nouvelle partie du tableau (indices 3, 4, 5)
    // La partie existante (0, 1, 2) a été préservée par realloc.
    printf("Initialisation de la nouvelle partie...\n");
    for (i = taille_actuelle / 2; i < taille_actuelle; i++) {
        nombres[i] = (i + 1) * 100; // 400, 500, 600
    }

    // 6. Afficher le contenu du tableau redimensionné
    printf("Contenu du tableau après réallocation :\n");
    for (i = 0; i < taille_actuelle; i++) {
        printf("  nombres[%zu] = %d\n", i, nombres[i]);
    }

    // 7. Libérer la mémoire finale redimensionnée
    free(nombres);
    printf("\nMémoire finale libérée.\n");
    nombres = NULL;
```

```
    return 0;
}
```

`free()` : Libérer la mémoire

Toute mémoire allouée dynamiquement avec `malloc`, `calloc` ou `realloc` (et non retournée par `realloc` avec une nouvelle taille de 0) **doit** être explicitement retournée au système lorsque vous n'en avez plus besoin. C'est le rôle **crucial** de la fonction `free()`.

Syntaxe : `void free(void *ptr);`

- `ptr`: Doit être un pointeur qui pointe vers le début d'un bloc de mémoire précédemment retourné par `malloc`, `calloc` ou `realloc` (et qui n'a pas déjà été libéré), ou bien `ptr` peut être `NULL`.
- **Comportement :**
  - Marque le bloc de mémoire pointé par `ptr` comme étant de nouveau disponible pour de futures allocations par `malloc` & co. Le contenu de la mémoire libérée devient indéterminé.
  - Ne modifie **pas** la valeur du pointeur `ptr` lui-même (il contient toujours la même adresse, qui n'est plus valide).
  - Si `ptr` est `NULL`, `free(NULL)` est défini par le standard C comme une opération sûre qui ne fait **rien**. C'est pratique car vous n'avez pas besoin de vérifier `if (ptr != NULL)` *avant* d'appeler `free(ptr)`.

Conséquences de l'oubli de `free()` : Les Fuites de Mémoire (Memory Leaks)

- Si vous perdez le seul pointeur vers un bloc de mémoire alloué dynamiquement (par exemple, en sortant d'une fonction où le pointeur était une variable locale, ou en réassignant le pointeur à une autre adresse) **sans avoir appelé `free()`** sur l'adresse originale, cette mémoire reste marquée comme "utilisée" par votre programme mais devient **inaccessible**. Elle ne peut pas être réutilisée par de futurs `malloc`.
- Si cela se produit répétitivement (par exemple, dans une boucle ou une fonction appelée souvent), votre programme consommera de plus en plus de mémoire système sans jamais la rendre. C'est une **fuite mémoire**. Pour les programmes de longue durée (serveurs, etc.), les fuites mémoire peuvent finir par épuiser toute la mémoire disponible, ralentir le système, voire causer un crash.

Pointeurs Suspendus (Dangling Pointers) et Double Libération (Double Free)

- Après avoir appelé `free(ptr)`, le pointeur `ptr` devient un **pointeur suspendu (dangling pointer)**. Il pointe toujours vers l'ancienne adresse, mais cette mémoire ne vous appartient plus et pourrait être réallouée à tout moment pour autre chose par le système.
- Utiliser un dangling pointer (lire ou écrire via `*ptr` ou `ptr->membre` après `free(ptr)`) conduit à un **comportement indéfini** (crash, corruption de données silencieuse, résultats incorrects...).
- **Bonne pratique :** Immédiatement après avoir appelé `free(ptr)`, mettez le pointeur à `NULL` (`ptr = NULL;`). Cela empêche son utilisation accidentelle ultérieure (car une tentative de déréférencer `NULL` plantera de manière prévisible) et rend un éventuel second appel `free(ptr)` (qui serait maintenant `free(NULL)`) inoffensif.
- Appeler `free()` deux fois sur le même pointeur *non-NULL* (sans réallocation entre les deux) est une erreur appelée **double free**. Cela corrompt généralement les structures internes du gestionnaire de

mémoire du tas et peut mener à des crashs ou des comportements imprévisibles plus tard dans l'exécution.

```c
#include <stdio.h>
#include <stdlib.h> // Pour malloc, free, NULL

int main() {
    int *donnees = NULL;
    size_t taille = 5;
    size_t i;

    printf("Allocation de mémoire pour %zu entiers...\n", taille);
    donnees = (int*)malloc(taille * sizeof(int));

    if (donnees == NULL) {
        perror("Echec de malloc");
        return 1;
    }
    printf("Mémoire allouée à %p.\n", (void*)donnees);

    // Utilisation...
    for (i = 0; i < taille; i++) {
        donnees[i] = i * 100;
    }
    printf("Donnée à l'indice 2 : %d\n", donnees[2]);

    // --- Libération Correcte ---
    printf("\nLibération de la mémoire...\n");
    free(donnees); // La mémoire est retournée au système

    // --- Prévention du Dangling Pointer ---
    donnees = NULL; // Mettre le pointeur à NULL après free
    printf("Pointeur mis à NULL.\n");

    // --- Tentatives d'utilisation après free (évitées si ptr est NULL) --
-
    if (donnees != NULL) {
         // Ce code ne devrait pas s'exécuter car donnees est NULL
        printf("Tentative d'accès après free (ne devrait pas arriver):
%d\n", donnees[0]);
    } else {
        printf("Le pointeur 'donnees' est maintenant NULL, accès impossible
(sécurisé).\n");
    }

    // --- Appeler free sur NULL est sûr ---
    printf("Appel de free(NULL)...\n");
    free(donnees); // Ne fait rien, pas d'erreur.
    printf("free(NULL) terminé sans erreur.\n");

    // --- Erreur courante : Double Free (si on n'avait pas mis donnees à
NULL) ---
```

```
    // int *autre_ptr = (int*)malloc(sizeof(int));
    // if (autre_ptr) {
    //     free(autre_ptr);
    //     // Oubli de mettre à NULL: autre_ptr est dangling
    //     printf("Tentative de double free...\n");
    //     // free(autre_ptr); // COMPORTEMENT INDÉFINI - CRASH PROBABLE !
    // }

    return 0;
}
```

Erreurs courantes avec l'allocation dynamique (Résumé)

La gestion manuelle de la mémoire avec `malloc`, `calloc`, `realloc` et `free` est puissante mais intrinsèquement dangereuse si elle n'est pas faite avec rigueur. Soyez particulièrement vigilant pour éviter :

- **Fuites de mémoire (Memory Leaks) :** Oublier d'appeler `free()` pour chaque bloc alloué qui n'est plus nécessaire, ou perdre le seul pointeur vers un bloc alloué.
- **Pointeurs suspendus (Dangling Pointers) :** Utiliser un pointeur (`*ptr` ou `ptr[...]`) après que la mémoire pointée a été libérée par `free()` (ou déplacée par `realloc`). Mettre le pointeur à `NULL` après `free` aide à prévenir cela.
- **Double Libération (Double Free) :** Appeler `free()` deux fois sur le même pointeur (non-`NULL`) sans réallocation entre les deux. Corrompt le tas. Mettre le pointeur à `NULL` après le premier `free` rend le second `free` inoffensif.
- **Libération Invalide :** Appeler `free()` sur un pointeur qui n'a pas été obtenu via `malloc`/`calloc`/`realloc` (par exemple, l'adresse d'une variable locale `&var_locale`, un pointeur vers une constante littérale, ou un pointeur non initialisé/invalide).
- **Utilisation de mémoire non initialisée :** Lire des données depuis un bloc alloué par `malloc` (ou la partie étendue par `realloc`) avant d'y avoir écrit des valeurs valides. Utiliser `calloc` ou `memset` après `malloc` peut éviter cela.
- **Dépassement de tampon (Buffer Overflow/Underflow) :** Écrire ou lire au-delà des limites (taille demandée lors de l'allocation) du bloc de mémoire alloué (par exemple, accéder à `ptr[taille_allouee]` ou `ptr[-1]`). C'est une cause majeure de bugs et de failles de sécurité.
- **Ignorer la valeur de retour `NULL` :** Ne pas vérifier si `malloc`/`calloc`/`realloc` a réussi (retourné `NULL`), et tenter d'utiliser le pointeur `NULL` comme s'il était valide.
- **Erreur d'un de moins (Off-by-one error) :** Allouer `n` octets mais essayer d'accéder à l'indice `n` (qui est hors limites, les indices valides étant de 0 à `n-1`).

Des outils comme **Valgrind** (sous Linux/macOS) sont extrêmement utiles pour détecter les erreurs de gestion mémoire lors de l'exécution de votre programme. Compiler avec les options d'avertissement maximales (`-Wall -Wextra ...`) peut aussi aider à repérer certaines erreurs potentielles.

---

13. Les Structures

Les tableaux nous permettent de regrouper plusieurs éléments de *même* type. Mais que faire si nous voulons regrouper sous un seul nom des informations de *types différents* qui sont logiquement liées ? Par exemple,

pour représenter une personne, nous pourrions avoir besoin de stocker son nom (une chaîne de caractères), son âge (un entier), et sa taille (un flottant) ensemble. C'est là qu'interviennent les **structures**.

Une **structure** (mot-clé `struct` en C) est un **type de données composite** défini par l'utilisateur. Elle permet de regrouper plusieurs variables (appelées **membres** ou **champs** - *members* ou *fields* en anglais) de types potentiellement différents sous un seul nom. Les structures sont fondamentales en C pour créer des types de données personnalisés et organiser des informations complexes de manière logique. Elles sont similaires aux objets ou classes dans d'autres langages, mais sans les méthodes associées (en C pur).

Définition (`struct Tag { ... };`)

Avant de pouvoir utiliser une structure, vous devez **définir son modèle** ou "patron" (template). Cela se fait avec le mot-clé `struct`, suivi d'un **nom d'étiquette** (tag name) que vous choisissez pour votre type de structure (optionnel mais recommandé), et d'une liste de déclarations de ses membres entre accolades `{}`. La définition se termine **obligatoirement** par un point-virgule `;`.

Syntaxe : `struct NomDeLEtiquette { type1 nom_membre1; type2 nom_membre2; // ... autres membres };` // <-- Point-virgule obligatoire ici !

- `struct NomDeLEtiquette` devient un nouveau nom de type. Pour déclarer une variable de ce type, vous devrez écrire `struct NomDeLEtiquette ma_variable;` (sauf si vous utilisez `typedef`, voir plus loin).
- `nom_membre1`, `nom_membre2`, etc., sont les variables contenues dans la structure. Chaque membre a son propre type et son propre nom, et occupe son propre espace mémoire à l'intérieur de la structure.

```c
#include <stdio.h>
#include <string.h> // Nécessaire plus tard pour copier des chaînes

// Définition de la structure "Etudiant"
// Nom de l'étiquette : Etudiant
struct Etudiant {
    char nom[50];       // Membre : tableau de char pour le nom
    int numero_id;      // Membre : entier pour l'ID
    float moyenne;      // Membre : flottant pour la moyenne
    char section;       // Membre : caractère pour la section (ex: 'A',
'B'...)
}; // Ne pas oublier le point-virgule

// Définition d'une structure "Point"
struct Point {
    double x;
    double y;
};

// Cette partie définit seulement les "plans" des structures.
// Aucune variable de ce type n'est encore créée.
int main() {
    printf("Définitions des structures Etudiant et Point effectuées.\n");
    // L'utilisation viendra dans les exemples suivants.
    return 0;
```

```
}
```

Déclaration et Initialisation (Ordonnée, Désignée C99+)

Une fois la structure définie (comme `struct Etudiant` ci-dessus), vous pouvez déclarer des variables de ce type, comme vous le feriez pour des types de base (`int`, `float`...).

Déclaration : Pour déclarer une variable de type structure, vous devez utiliser le mot-clé `struct` suivi du nom de l'étiquette : `struct NomDeLEtiquette nom_variable1; struct Etudiant etu1, etu2;`

Initialisation : Vous pouvez initialiser une variable de structure lors de sa déclaration. Il y a deux manières principales :

1. **Initialisation Ordonnée (Style C89/C90) :** Fournir une liste de valeurs entre accolades `{}`, dans l'ordre exact où les membres ont été déclarés dans la définition de la structure. `struct Etudiant etu_alice = {"Alice Dupont", 12345, 15.5f, 'B'};`
2. **Initialiseurs Désignés (Designated Initializers - C99+) :** Utiliser la notation `.nom_membre = valeur` à l'intérieur des accolades. Cela permet d'initialiser les membres dans n'importe quel ordre et rend le code plus lisible et moins sujet aux erreurs si l'ordre des membres dans la structure change. Les membres non explicitement initialisés sont mis à zéro (ou `NULL` pour les pointeurs). C'est la méthode **recommandée** si votre compilateur supporte C99 ou plus récent. `struct Point p1 = {.y = 2.5, .x = 1.0}; struct Etudiant etu_bob = {.nom = "Bob Martin", .numero_id = 67890}; // moyenne sera 0.0f, section sera '\0' (0)`

Si une variable structure locale n'est pas initialisée, ses membres contiennent des valeurs indéterminées (comme pour les variables locales de types de base). Les variables structure globales ou statiques sont initialisées à zéro/NULL par défaut.

```c
#include <stdio.h>
#include <string.h> // Pour strcpy

// Reprise des définitions de structures
struct Etudiant {
    char nom[50];
    int numero_id;
    float moyenne;
    char section;
};

struct Point {
    double x;
    double y;
};

int main() {
    // Déclaration de variables de type structure
    struct Etudiant etu1; // Contenu indéterminé (variable locale)
    struct Point centre;
```

```c
    // Initialisation ordonnée
    struct Etudiant etu2 = {"Alice Wonderland", 12345, 16.75f, 'A'};

    // Initialisation désignée (C99+) - Recommandé
    struct Etudiant etu3 = {
        .nom = "Bob Marley",
        .numero_id = 67890,
        .moyenne = 14.2f
        // .section n'est pas spécifiée, sera initialisée à 0 ('\0')
    };
    struct Point origine = {.x = 0.0, .y = 0.0};

    // Assignation aux membres après déclaration (pour etu1)
    // Pour les chaînes, on doit utiliser strcpy (ou snprintf)
    strcpy(etu1.nom, "Charles Dupont");
    etu1.numero_id = 54321;
    etu1.moyenne = 9.5f;
    etu1.section = 'C';

    // Affichage
    printf("Etudiant 1: Nom=\"%s\", ID=%d, Moy=%.2f, Sec=%c\n",
           etu1.nom, etu1.numero_id, etu1.moyenne, etu1.section);
    printf("Etudiant 2: Nom=\"%s\", ID=%d, Moy=%.2f, Sec=%c\n",
           etu2.nom, etu2.numero_id, etu2.moyenne, etu2.section);
    printf("Etudiant 3: Nom=\"%s\", ID=%d, Moy=%.2f, Sec=%c (valeur %d)\n",
           etu3.nom, etu3.numero_id, etu3.moyenne,
           (etu3.section == '\0' ? '?' : etu3.section), etu3.section); //
Affiche '?' si section est \0
    printf("Origine: x=%.1f, y=%.1f\n", origine.x, origine.y);

    return 0;
}
```

Accès aux membres (`.`)

Pour accéder à un membre spécifique d'une variable de structure (que ce soit pour lire sa valeur ou pour la modifier), on utilise l'**opérateur point (`.`)**, aussi appelé opérateur d'accès aux membres (member access operator).

Syntaxe : `nom_variable_structure.nom_membre`

L'opérateur `.` se place entre le nom de la variable de type structure et le nom du membre auquel on veut accéder.

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

struct Produit {
    char nom[100];
```

```c
    double prix_ht;
    int stock;
};

int main() {
    struct Produit produit1;

    // Accès en écriture pour initialiser les membres
    strcpy(produit1.nom, "Stylo Bleu");
    produit1.prix_ht = 1.50;
    produit1.stock = 200;

    // Accès en lecture pour afficher les informations
    printf("--- Détails Produit ---\n");
    printf("Nom         : %s\n", produit1.nom);
    printf("Prix HT     : %.2f Euros\n", produit1.prix_ht);
    printf("Stock       : %d unités\n", produit1.stock);

    // Accès pour modification
    printf("\nRéception de 50 unités supplémentaires...\n");
    produit1.stock = produit1.stock + 50; // ou produit1.stock += 50;
    printf("Nouveau stock : %d unités\n", produit1.stock);

    // Calcul du prix TTC (exemple)
    const float TAUX_TVA = 0.20f;
    double prix_ttc = produit1.prix_ht * (1.0 + TAUX_TVA);
    printf("Prix TTC    : %.2f Euros\n", prix_ttc);

    return 0;
}
```

Pointeurs vers des structures et opérateur flèche (`->`)

Il est très fréquent en C de manipuler non pas les structures directement (qui peuvent être volumineuses à copier), mais des **pointeurs vers des structures**. C'est particulièrement vrai lorsque :

- On alloue des structures dynamiquement sur le tas (avec `malloc`).
- On passe des structures à des fonctions (pour éviter des copies coûteuses ou pour permettre à la fonction de modifier la structure originale via le passage par adresse).

On déclare un pointeur vers une structure comme n'importe quel autre pointeur : `struct NomDeLEtiquette *nom_pointeur;`

Exemple: `struct Etudiant *ptr_etu;`

Pour accéder aux membres de la structure *via le pointeur*, on a deux options syntaxiques :

1. **Déréférencement explicite (`*`) puis accès (`.`):** Il faut d'abord obtenir la structure elle-même en déréférençant le pointeur (`*nom_pointeur`), puis utiliser l'opérateur point (`.`) pour accéder au membre. Les **parenthèses autour du déréférencement sont obligatoires** car l'opérateur `.` a une priorité plus élevée que l'opérateur `*`. Syntaxe : `(*nom_pointeur).nom_membre`

2. **Opérateur flèche (`->`) (Arrow operator) :** C'est un raccourci syntaxique beaucoup plus courant, lisible et spécifiquement conçu pour accéder aux membres d'une structure *via un pointeur*. Syntaxe : `nom_pointeur->nom_membre`

L'expression `nom_pointeur->nom_membre` est **strictement équivalente** à `(*nom_pointeur).nom_membre`.

Règle simple :

- Utilisez l'opérateur point `.` lorsque vous travaillez directement avec une variable de type structure.
- Utilisez l'opérateur flèche `->` lorsque vous travaillez avec un **pointeur** vers une structure.

```c
#include <stdio.h>
#include <stdlib.h> // Pour malloc, free, NULL
#include <string.h>

struct Point {
    int x;
    int y;
};

// Fonction qui prend un POINTEUR vers un Point
void afficher_point(const struct Point *p) { // Utilise const pour indiquer
qu'on ne modifie pas
    if (p == NULL) {
        printf("Erreur: Pointeur de point NULL\n");
        return;
    }
    // Utilisation de l'opérateur flèche -> car p est un pointeur
    printf("Point : (%d, %d)\n", p->x, p->y);

    // Équivalent moins lisible :
    // printf("Point : (%d, %d)\n", (*p).x, (*p).y);
}

// Fonction qui modifie un Point via son pointeur
void deplacer_point(struct Point *p, int dx, int dy) {
     if (p == NULL) return;
     p->x += dx; // Modifie le membre x de la structure pointée
     p->y += dy; // Modifie le membre y de la structure pointée
}

int main() {
    struct Point p1 = {10, 20}; // Variable structure sur la pile
    struct Point *ptr_p1 = &p1; // Pointeur vers p1

    printf("Affichage de p1 via pointeur :\n");
    afficher_point(ptr_p1);

    printf("\nDéplacement de p1 via pointeur...\n");
    deplacer_point(ptr_p1, 5, -3); // Passe l'adresse de p1
```

```
    printf("Nouvelles coordonnées de p1 (via pointeur) :\n");
    afficher_point(ptr_p1);
    printf("Nouvelles coordonnées de p1 (directement) :\n");
    printf("Point : (%d, %d)\n", p1.x, p1.y); // p1 a bien été modifiée

    // --- Avec allocation dynamique ---
    struct Point *ptr_p2 = NULL;
    ptr_p2 = (struct Point*) malloc(sizeof(struct Point)); // Alloue
mémoire pour UN Point

    if (ptr_p2 == NULL) {
        perror("Erreur malloc");
        return 1;
    }
    printf("\nPoint p2 alloué dynamiquement à %p\n", (void*)ptr_p2);

    // Initialisation via le pointeur
    ptr_p2->x = 100;
    ptr_p2->y = 200;
    printf("Affichage de p2 :\n");
    afficher_point(ptr_p2);

    printf("\nDéplacement de p2...\n");
    deplacer_point(ptr_p2, -50, 50);
    printf("Nouvelles coordonnées de p2 :\n");
    afficher_point(ptr_p2);

    // Libération de la mémoire allouée
    free(ptr_p2);
    ptr_p2 = NULL;

    return 0;
}
```

Structures et Allocation Dynamique

Comme vu dans l'exemple précédent, il est très courant d'allouer dynamiquement de la mémoire pour des structures, surtout lorsqu'on ne connaît pas le nombre d'instances nécessaires à l'avance ou pour créer des structures de données liées (listes, arbres...).

Le processus est le même que pour les types de base :

1. Déclarer un **pointeur** vers le type de structure : `struct MonType *ptr;`
2. Utiliser `malloc` ou `calloc` pour allouer la mémoire nécessaire, en utilisant `sizeof(struct MonType)` pour obtenir la bonne taille. `ptr = (struct MonType*) malloc(sizeof(struct MonType));` // Pour une seule structure `ptr = (struct MonType*) calloc(nombre_elements, sizeof(struct MonType));` // Pour un tableau de structures
3. **Vérifier** si le pointeur retourné est `NULL`.
4. Accéder aux membres via l'opérateur flèche `->` : `ptr->membre = valeur;`
5. **Libérer** la mémoire avec `free(ptr)` lorsque vous n'en avez plus besoin.

(L'exemple `code_struct_pointeur_fleche_section13` montre déjà l'allocation dynamique d'une structure `Point`).

`typedef` pour simplifier les noms de types

Écrire `struct NomDeLEtiquette` à chaque fois que vous déclarez une variable ou un paramètre de fonction peut devenir fastidieux. Le mot-clé `typedef` permet de créer un **alias** (un synonyme) pour un type existant, y compris pour les types `struct`.

Syntaxe générale : `typedef type_existant NouvelAlias;`

Utilisation courante avec les structures : On combine souvent la définition de la structure et le `typedef` en une seule déclaration. Il y a deux manières principales :

1. **Avec une structure anonyme (sans étiquette) :**

   ```c
   typedef struct { // Pas d'étiquette ici après 'struct'
       // membres...
       int id;
       char nom[30];
   } Utilisateur_t; // Nom de l'alias (type)
   ```

   Après cela, vous pouvez déclarer des variables directement avec `Utilisateur_t` : `Utilisateur_t user1;`

2. **En donnant un nom d'étiquette ET un alias `typedef` :**

   ```c
   typedef struct _Produit { // Étiquette _Produit (souvent préfixée par _)
       // membres...
       double prix;
       int quantite;
   } Produit_t; // Nom de l'alias (type)
   ```

   Cette forme est utile pour les structures qui se référencent elles-mêmes (comme les nœuds de listes chaînées), car à l'intérieur de la définition, l'alias `Produit_t` n'est pas encore connu, mais l'étiquette `struct _Produit` l'est. Vous pouvez ensuite utiliser soit `struct _Produit`, soit `Produit_t` pour déclarer des variables.

L'utilisation de `typedef` rend le code plus concis et souvent plus facile à lire, en masquant le mot-clé `struct` lors de l'utilisation du type. Une convention fréquente (mais non obligatoire) est de terminer les alias de type créés par `typedef` par `_t` (par exemple, `Personne_t`, `Point_t`).

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

// Méthode 1: Structure anonyme avec typedef
```

```c
typedef struct {
    int jour;
    int mois;
    int annee;
} Date_t; // Date_t est maintenant un alias pour le type structure

// Méthode 2: Étiquette et typedef séparés (ou combinés)
typedef struct _Adresse { // Étiquette _Adresse
    char rue[100];
    char ville[50];
    char code_postal[10];
} Adresse_t; // Adresse_t est l'alias

// Utilisation des types définis par typedef
void afficher_date(Date_t d) {
    // Accès via l'opérateur point '.' car 'd' est une structure, pas un
pointeur
    printf("%02d/%02d/%d", d.jour, d.mois, d.annee);
}

void afficher_adresse(const Adresse_t *a) { // Prend un pointeur constant
vers l'adresse
    if (a == NULL) return;
    // Accès via l'opérateur flèche '->' car 'a' est un pointeur
    printf("  Rue: %s\n", a->rue);
    printf("  Ville: %s\n", a->ville);
    printf("  CP: %s\n", a->code_postal);
}

int main() {
    // Déclaration et initialisation en utilisant les alias typedef
    Date_t aujourdhui = {27, 4, 2025};
    Adresse_t bureau = {
        .rue = "123 Rue de la République",
        .ville = "Lyon",
        .code_postal = "69002"
    };

    printf("Date : ");
    afficher_date(aujourdhui); // Passe la structure par valeur (copie)
    printf("\n");

    printf("Adresse du bureau :\n");
    afficher_adresse(&bureau); // Passe l'adresse de la structure

    return 0;
}
```

Assignation et Copie de structures

Contrairement aux tableaux (dont le nom seul ne peut pas être à gauche d'une assignation), les variables de type `struct` peuvent être **assignées** directement les unes aux autres si elles sont du **même type de structure**.

L'assignation (`=`):

- Effectue une **copie membre à membre**. La valeur de chaque membre de la structure de droite est copiée dans le membre correspondant de la structure de gauche.
- Syntaxe : `structure_destination = structure_source;`

Cela fonctionne aussi lorsque les structures sont passées **par valeur** à une fonction ou retournées **par valeur** par une fonction : une copie membre à membre est effectuée à ce moment-là.

(Voir l'exemple de code `code_struct_copie_section13` pour une illustration pratique).

Attention : Copie Superficielle (Shallow Copy) pour les pointeurs Si une structure contient des **membres qui sont des pointeurs**, l'assignation directe copie uniquement la *valeur du pointeur* (c'est-à-dire l'adresse mémoire qu'il contient), et **non les données *pointées***. Après la copie, les deux structures (l'originale et la copie) auront des membres pointeurs qui pointent vers la **même zone mémoire** sur le tas.

Conséquence : Si vous modifiez les données *via* le pointeur dans l'une des structures, cette modification sera visible depuis l'autre structure, car elles partagent les données pointées. Ce n'est souvent pas le comportement souhaité si on veut des copies indépendantes.

Si vous avez besoin de dupliquer également les données pointées (pour que chaque structure ait sa propre copie indépendante des données), vous devez implémenter une **copie profonde (deep copy)** manuellement. Cela implique généralement :

1. D'allouer de la nouvelle mémoire (avec `malloc`) pour la copie des données pointées.
2. De copier le contenu des données pointées de la source vers la nouvelle mémoire allouée (avec `memcpy` ou d'autres méthodes).
3. D'assigner l'adresse de cette nouvelle mémoire au membre pointeur de la structure de destination.

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

typedef struct {
    char modele[50];
    int annee;
} Voiture;

int main() {
    Voiture voiture1 = {"Renault Clio", 2022};
    Voiture voiture2;
    Voiture voiture3 = {"Peugeot 208", 2023};

    printf("Voiture 1: %s (%d)\n", voiture1.modele, voiture1.annee);

    // Assignation directe (copie membre à membre)
    voiture2 = voiture1;
```

```
    printf("Voiture 2 (après copie de voiture1): %s (%d)\n",
voiture2.modele, voiture2.annee);

    // Modifier voiture2 n'affecte pas voiture1 (car copie)
    voiture2.annee = 2024;
    strcpy(voiture2.modele, "Renault Captur");

    printf("\nAprès modification de voiture2:\n");
    printf("Voiture 1: %s (%d)\n", voiture1.modele, voiture1.annee); //
Reste "Renault Clio", 2022
    printf("Voiture 2: %s (%d)\n", voiture2.modele, voiture2.annee); //
Devient "Renault Captur", 2024

    // On peut aussi initialiser une structure avec une autre
    Voiture voiture4 = voiture3;
     printf("\nVoiture 4 (initialisée depuis voiture3): %s (%d)\n",
voiture4.modele, voiture4.annee);


    return 0;
}
```

Structures imbriquées

Un membre d'une structure peut lui-même être une autre structure. Cela permet de créer des hiérarchies de données logiques et de mieux organiser l'information.

Exemple : Une structure `Etudiant` pourrait contenir une structure `Date` pour la date de naissance et une structure `Adresse`.

Pour accéder aux membres de la structure interne, on **chaîne les opérateurs d'accès** (`.` ou `->`).

- Si `var` est une variable structure : `var.membre_struct_interne.sous_membre`
- Si `ptr` est un pointeur vers une structure : `ptr->membre_struct_interne.sous_membre`
- Si `membre_struct_interne` est lui-même un pointeur : `ptr->ptr_struct_interne->sous_membre`

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

// Structure pour une date
typedef struct {
    int jour;
    int mois;
    int annee;
} Date_t;

// Structure pour une personne, contenant une date de naissance
typedef struct {
    char nom[50];
```

```c
    Date_t date_naissance; // Membre qui est une autre structure
} Personne_t;

// Structure pour un événement, contenant un pointeur vers une date
typedef struct {
    char description[100];
    Date_t *date_evenement; // Pointeur vers une structure Date_t
} Evenement_t;


int main() {
    // Initialisation d'une personne avec une date imbriquée
    Personne_t p1 = {
        .nom = "Alice",
        .date_naissance = { .jour = 15, .mois = 6, .annee = 1998 } //
Initialiseur imbriqué
    };

    // Accès aux membres imbriqués via l'opérateur point '.'
    printf("Personne: %s\n", p1.nom);
    printf("Née le : %02d/%02d/%d\n",
           p1.date_naissance.jour,
           p1.date_naissance.mois,
           p1.date_naissance.annee);

    // Modification d'un membre imbriqué
    p1.date_naissance.annee = 1999;
    printf("Année de naissance corrigée : %d\n", p1.date_naissance.annee);

    // Exemple avec pointeur et structure imbriquée
    Date_t date_reunion = {30, 4, 2025};
    Evenement_t reunion = {"Réunion Projet Alpha", &date_reunion}; //
Stocke l'adresse

    // Accès via pointeur (->) puis via pointeur (->)
    if (reunion.date_evenement != NULL) {
        printf("\nÉvénement: %s\n", reunion.description);
        printf("Date (via pointeur): %02d/%02d/%d\n",
               reunion.date_evenement->jour,  // -> car date_evenement est
un pointeur
               reunion.date_evenement->mois,
               reunion.date_evenement->annee);
    }

    return 0;
}
```

Tableaux de structures

Vous pouvez créer des tableaux dont chaque élément est une structure du même type. C'est extrêmement utile pour gérer des collections d'enregistrements complexes, comme une liste d'étudiants, un inventaire de

produits, une base de données de contacts, etc.

Déclaration : La syntaxe est la même que pour les tableaux de types de base : `TypeStructure nom_tableau[TAILLE];` (Où `TypeStructure` est soit `struct Tag`, soit un alias `typedef`).

Exemple : `struct Etudiant classe[30];` // Un tableau pour 30 étudiants. `Point_t chemin[100];` // Un tableau de 100 points (si Point_t est un typedef).

Initialisation : Comme pour les tableaux multidimensionnels, on utilise des accolades imbriquées. La liste externe délimite le tableau, et chaque ensemble d'accolades internes initialise une structure (un élément du tableau).

(Voir l'exemple de code `code_tableau_struct_section13` pour l'illustration de l'initialisation).

Accès : Pour accéder au membre `m` de la structure située à l'indice `i` du tableau, on combine l'accès à l'élément du tableau avec l'opérateur d'accès au membre : `nom_tableau[i].m`

Si vous avez un pointeur `ptr` qui pointe vers le début du tableau de structures (par exemple, `ptr = bibliotheque;`), vous pouvez accéder au membre `m` de l'élément `i` de plusieurs manières équivalentes :

- Arithmétique des pointeurs explicite : `(ptr + i)->m` (l'arithmétique des pointeurs est mise à l'échelle par `sizeof(TypeStructure)`)
- Notation crochet sur le pointeur : `ptr[i].m` (souvent la plus lisible)

(Voir l'exemple de code `code_tableau_struct_section13` pour une illustration pratique de l'accès).

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

// Définition de la structure Livre
typedef struct {
    char titre[100];
    char auteur[50];
    int annee_pub;
} Livre_t;

#define MAX_LIVRES 5

int main() {
    // Déclaration d'un tableau de structures Livre_t
    Livre_t bibliotheque[MAX_LIVRES];
    int nb_livres = 0; // Pour suivre combien de livres sont réellement
stockés

    // Ajouter des livres au tableau
    if (nb_livres < MAX_LIVRES) {
        strcpy(bibliotheque[nb_livres].titre, "Le Seigneur des Anneaux");
        strcpy(bibliotheque[nb_livres].auteur, "J.R.R. Tolkien");
        bibliotheque[nb_livres].annee_pub = 1954;
        nb_livres++; // Incrémente le compteur de livres
    }
```

```c
    if (nb_livres < MAX_LIVRES) {
        // Utiliser l'indice courant nb_livres pour ajouter le suivant
        strcpy(bibliotheque[nb_livres].titre, "Fondation");
        strcpy(bibliotheque[nb_livres].auteur, "Isaac Asimov");
        bibliotheque[nb_livres].annee_pub = 1951;
        nb_livres++;
    }

     if (nb_livres < MAX_LIVRES) {
        // Initialisation directe d'un élément du tableau
        bibliotheque[nb_livres] = (Livre_t){"Dune", "Frank Herbert", 1965};
        nb_livres++;
    }

    // Afficher le contenu de la bibliothèque
    printf("--- Bibliothèque (%d livres) ---\n", nb_livres);
    for (int i = 0; i < nb_livres; i++) {
        printf("Livre %d:\n", i + 1);
        // Accès aux membres de la structure à l'index i du tableau
        printf("  Titre : %s\n", bibliotheque[i].titre);
        printf("  Auteur: %s\n", bibliotheque[i].auteur);
        printf("  Année : %d\n", bibliotheque[i].annee_pub);
    }

    // Accès via un pointeur vers le tableau
    Livre_t *ptr_biblio = bibliotheque; // Pointeur vers le premier livre
    printf("\nDeuxième livre (via pointeur et ->) :\n");
    if (nb_livres >= 2) {
         // (ptr_biblio + 1) pointe vers le deuxième élément
        printf("  Titre : %s\n", (ptr_biblio + 1)->titre);
        // ptr_biblio[1] est équivalent à *(ptr_biblio + 1)
        printf("  Auteur: %s\n", ptr_biblio[1].auteur);
    }


    return 0;
}
```

Les structures sont un outil essentiel pour organiser les données de manière logique et créer des types personnalisés en C. Combinées avec les pointeurs, l'allocation dynamique, les `typedef`, les tableaux et l'imbrication, elles permettent de construire des structures de données très flexibles et puissantes, adaptées à des problèmes complexes.

---

14. Unions et Énumérations

En plus des structures (`struct`) qui regroupent plusieurs membres ayant chacun leur propre espace mémoire, le C propose deux autres mécanismes pour créer des types de données ou des ensembles de constantes nommées : les **unions** (`union`) et les **énumérations** (`enum`). Elles servent des objectifs différents des structures et entre elles.

- Les **unions** permettent à plusieurs membres (de types potentiellement différents) de **partager la *même* zone mémoire**. Seul un membre peut être "actif" à la fois.
- Les **énumérations** permettent de définir un ensemble de **constantes entières nommées**, améliorant la lisibilité et la maintenabilité du code en remplaçant des valeurs numériques arbitraires par des noms significatifs.

Unions (`union`) : Partager la mémoire

Une **union** ressemble syntaxiquement beaucoup à une structure, mais son fonctionnement interne est radicalement différent. Alors qu'une structure alloue de l'espace mémoire séparé et suffisant pour *tous* ses membres, une **union alloue une seule zone mémoire**, dont la taille est déterminée par la taille de son membre le **plus volumineux**. Tous les membres de l'union (quel que soit leur type) partagent cet unique espace mémoire.

Syntaxe de définition : Similaire à `struct`, mais avec le mot-clé `union`.

`union NomDeLEtiquette { type1 membre1; type2 membre2; // ... autres membres };` // <-- Point-virgule obligatoire

- `union NomDeLEtiquette` définit un nouveau type union.
- Les membres sont déclarés comme dans une structure.

Utilité Principale :

1. **Économie de mémoire :** C'est l'usage le plus courant. Une union est utile lorsque vous avez besoin de stocker différents types de données dans une même variable, mais que vous savez que **seul un de ces types sera valide ou utilisé à un instant T**. Au lieu d'allouer de l'espace pour tous les types possibles (comme le ferait une structure), l'union n'utilise que l'espace nécessaire pour le plus grand type possible parmi ses membres.
2. **Type Punning (Jeu de types) :** Permet d'écrire une donnée en mémoire via un membre de l'union et de la relire (interpréter la même séquence d'octets) via un autre membre de type différent. C'est une technique avancée, souvent utilisée en programmation bas niveau (par exemple, pour examiner la représentation binaire d'un flottant en le lisant comme un entier), mais elle peut dépendre de l'architecture matérielle (endianness, etc.) et doit être utilisée avec une extrême prudence car elle peut violer les règles d'alias strict du C et mener à un comportement indéfini si mal comprise.

Taille d'une union : La taille d'une variable de type union (`sizeof(union Nom...)`) est au moins la taille de son plus grand membre. Le compilateur peut ajouter du "rembourrage" (padding) pour des raisons d'alignement mémoire, donc la taille peut être légèrement supérieure à celle du plus grand membre.

```c
#include <stdio.h>
#include <string.h> // Pour sizeof(char[...])

// Définition d'une union nommée 'Data'
union Data {
    int i;          // Membre entier (souvent 4 octets)
    double d;       // Membre double (souvent 8 octets)
    char str[20];   // Membre chaîne (20 octets)
}; // Ne pas oublier le ;
```

```c
int main() {
    union Data data_union; // Déclaration d'une variable de type union Data

    printf("--- Tailles des membres ---\n");
    printf("Taille de int    : %zu octets\n", sizeof(int));
    printf("Taille de double : %zu octets\n", sizeof(double));
    printf("Taille de char[20]: %zu octets\n", sizeof(char[20]));
    printf("---------------------------\n");

    // La taille de l'union sera au moins celle du plus grand membre (char str[20]).
    // Elle pourrait être légèrement plus grande à cause de l'alignement.
    printf("Taille de union Data : %zu octets\n", sizeof(union Data));
    printf("Taille de la variable data_union : %zu octets\n", sizeof(data_union));

    return 0;
}
```

Accès aux membres et Avertissement Crucial : On accède aux membres d'une union exactement comme pour une structure, en utilisant l'opérateur point (`.`) sur la variable union ou l'opérateur flèche (`->`) sur un pointeur vers l'union.

MAIS ATTENTION : C'est là que réside le danger et la différence fondamentale avec les structures. Étant donné que tous les membres partagent la même mémoire, **seul le dernier membre auquel vous avez *écrit* une valeur est considéré comme valide ou "actif"**. Écrire dans un membre de l'union (par ex., `data.i = 10;`) **écrase** les données qui pouvaient s'y trouver et qui représentaient potentiellement un autre membre (par ex., la valeur de `data.d` ou `data.str` est perdue et remplacée par la représentation binaire de l'entier 10).

Lire un membre différent de celui qui a été écrit en dernier (`printf("%f", data.f);` après avoir fait `data.i = 10;`) conduit à interpréter les octets en mémoire selon le type du membre lu, ce qui donnera généralement une valeur **incohérente ou "corrompue"** (sauf si c'est un cas de *type punning* intentionnel et maîtrisé).

Il est donc de la **responsabilité du programmeur** de savoir (ou de suivre d'une manière ou d'une autre) quel membre de l'union est actuellement "actif" et contient une donnée valide. Une technique très courante consiste à utiliser une **structure** qui encapsule :

1. L'union elle-même.
2. Une variable supplémentaire (souvent une **énumération**) qui indique le type de donnée actuellement stocké dans l'union.

```c
#include <stdio.h>
#include <string.h> // Pour strcpy

// L'union pour stocker différents types
union ValeurPossible {
    int entier;
```

```
    float flottant;
    char caractere;
};

// Énumération pour savoir quel type est stocké
typedef enum { TYPE_AUCUN, TYPE_ENTIER, TYPE_FLOTTANT, TYPE_CARACTERE }
TypeValeur;

// Structure combinant l'union et l'indicateur de type
typedef struct {
    TypeValue type_actuel;      // Indique quel membre de l'union est
valide
    union ValeurPossible valeur; // L'union elle-même
} DonneeVariable;

// Fonction pour afficher la donnée variable en fonction de son type
void afficher_donnee(DonneeVariable d) {
    switch (d.type_actuel) {
        case TYPE_ENTIER:
            printf("Donnée (Entier) : %d\n", d.valeur.entier);
            break;
        case TYPE_FLOTTANT:
            printf("Donnée (Flottant) : %f\n", d.valeur.flottant);
            break;
        case TYPE_CARACTERE:
            printf("Donnée (Caractère) : %c\n", d.valeur.caractere);
            break;
        case TYPE_AUCUN:
            printf("Donnée (Aucune valeur stockée)\n");
            break;
        default:
            printf("Type de donnée inconnu !\n");
    }
}

int main() {
    DonneeVariable ma_donnee;

    // 1. Stocker un entier
    ma_donnee.type_actuel = TYPE_ENTIER;
    ma_donnee.valeur.entier = 42;
    printf("Après stockage entier:\n");
    afficher_donnee(ma_donnee);
    // Lire ma_donnee.valeur.flottant ici donnerait une valeur incohérente

    printf("\n");

    // 2. Stocker un flottant (cela écrase l'entier dans la même zone
mémoire)
    ma_donnee.type_actuel = TYPE_FLOTTANT;
    ma_donnee.valeur.flottant = -9.81f;
    printf("Après stockage flottant:\n");
    afficher_donnee(ma_donnee);
    // Lire ma_donnee.valeur.entier maintenant interpréterait les bits du
```

```
float comme un int

    printf("\n");

     // 3. Stocker un caractère
    ma_donnee.type_actuel = TYPE_CARACTERE;
    ma_donnee.valeur.caractere = '$';
    printf("Après stockage caractère:\n");
    afficher_donnee(ma_donnee);

    return 0;
}
```

Énumérations (`enum`) : Constantes nommées

Une **énumération** (mot-clé `enum`) permet de définir un nouveau **type entier** dont les valeurs possibles sont restreintes à un ensemble de **constantes symboliques nommées** (appelées *énumérateurs*).

L'objectif principal des énumérations est d'améliorer la **lisibilité** et la **maintenabilité** du code. Au lieu d'utiliser des "nombres magiques" (constantes littérales comme 0, 1, 2 dont la signification n'est pas immédiatement évidente dans le code), on utilise des noms descriptifs.

Syntaxe de définition : `enum NomEnumeration { ENUMERATEUR1, ENUMERATEUR2, ENUMERATEUR3 = 10, // On peut assigner une valeur explicite ENUMERATEUR4, // Vaudra 11 (précédent + 1) ENUMERATEUR5 = 10 // Les valeurs ne doivent pas être uniques };` // Point-virgule optionnel ici, mais souvent mis par cohérence

- `enum NomEnumeration` définit un nouveau type.
- `ENUMERATEUR1`, `ENUMERATEUR2`, etc., sont les constantes nommées. Par convention, on les écrit souvent en majuscules.
- **Valeurs par défaut :** Si aucune valeur n'est assignée explicitement, le premier énumérateur vaut **0**, et chaque énumérateur suivant vaut la valeur du précédent **+ 1**.
- **Assignation explicite :** Vous pouvez assigner une valeur entière spécifique à un énumérateur en utilisant `= valeur`. Les énumérateurs suivants sans valeur explicite continueront la séquence à partir de la dernière valeur assignée + 1.
- **Type sous-jacent :** Les énumérateurs sont des constantes de type `int`. Le type `enum NomEnumeration` lui-même est compatible avec `int`, mais utiliser le type `enum` peut améliorer la clarté et permettre certaines vérifications par le compilateur.

```
#include <stdio.h>

// Énumération simple pour les jours de la semaine
// Les valeurs par défaut sont 0, 1, 2, ...
enum JourSemaine {
    LUNDI,    // 0
    MARDI,    // 1
    MERCREDI, // 2
    JEUDI,    // 3
```

```c
    VENDREDI, // 4
    SAMEDI,   // 5
    DIMANCHE  // 6
};

// Énumération pour des codes de statut avec valeurs explicites
enum StatutOperation {
    STATUT_OK = 0,
    STATUT_ERREUR_FICHIER = 10,
    STATUT_ERREUR_MEMOIRE = 20,
    STATUT_ERREUR_RESEAU, // Vaudra 21 (précédent + 1)
    STATUT_INCONNU = -1
};

// Utilisation avec typedef pour un alias plus court
typedef enum {
    ROUGE,  // 0
    VERT,   // 1
    BLEU    // 2
} Couleur_t;

int main() {
    // Les énumérateurs sont des constantes de type int
    printf("Valeurs des jours :\n");
    printf("  LUNDI    = %d\n", LUNDI);
    printf("  MERCREDI = %d\n", MERCREDI);
    printf("  DIMANCHE = %d\n", DIMANCHE);

    printf("\nValeurs des statuts :\n");
    printf("  STATUT_OK            = %d\n", STATUT_OK);
    printf("  STATUT_ERREUR_FICHIER= %d\n", STATUT_ERREUR_FICHIER);
    printf("  STATUT_ERREUR_MEMOIRE= %d\n", STATUT_ERREUR_MEMOIRE);
    printf("  STATUT_ERREUR_RESEAU = %d\n", STATUT_ERREUR_RESEAU); // 21
    printf("  STATUT_INCONNU       = %d\n", STATUT_INCONNU);

    // Utilisation de l'alias typedef
    Couleur_t couleur_fond = BLEU;
    printf("\nCode couleur pour BLEU : %d\n", couleur_fond); // 2

    return 0;
}
```

Utilisation des énumérations :

Une fois une énumération définie (par exemple `enum EtatJeu { MENU, EN_COURS, PAUSE, FIN };`), vous pouvez :

1. **Déclarer des variables** de ce type énuméré : `enum EtatJeu etat_partie;` Ou si vous avez utilisé `typedef` (ex: `typedef enum {...} EtatJeu_t;`) : `EtatJeu_t etat_partie;`
2. **Assigner** une des constantes énumérées à ces variables : `etat_partie = EN_COURS;`

3. **Comparer** ces variables avec les constantes énumérées : `if (etat_partie == PAUSE) { ... }`
4. Les utiliser dans des instructions `switch` pour une meilleure lisibilité :

```
switch (etat_partie) {
    case MENU:
        // Afficher le menu
        break;
    case EN_COURS:
        // Gérer le tour de jeu
        break;
    // ... autres cas ...
    default:
        // Gérer état inconnu
}
```

L'avantage principal est la **clarté du code**. Comparer `etat_partie == EN_COURS` est beaucoup plus explicite que de comparer `etat_partie == 1` (si 1 était le "nombre magique" pour l'état en cours). Si vous devez ajouter ou modifier des états, vous le faites principalement dans la définition de l'`enum`, et le compilateur peut vous aider à repérer les `switch` qui ne gèrent pas tous les cas (avec les bonnes options d'avertissement).

```
#include <stdio.h>

// Énumération pour les directions
typedef enum {
    NORD, // 0
    SUD,  // 1
    EST,  // 2
    OUEST // 3
} Direction_t;

// Fonction qui prend une direction en argument
void deplacer(Direction_t direction) {
    printf("Déplacement vers ");
    switch (direction) {
        case NORD:
            printf("le Nord.\n");
            break;
        case SUD:
            printf("le Sud.\n");
            break;
        case EST:
            printf("l'Est.\n");
            break;
        case OUEST:
            printf("l'Ouest.\n");
            break;
        // Pas de default ici, on suppose que seules les directions valides
```

```
sont passées
    }
}

int main() {
    Direction_t mon_chemin; // Variable de type énuméré

    mon_chemin = EST; // Assignation d'une valeur énumérée
    deplacer(mon_chemin);

    // Comparaison
    if (mon_chemin == EST || mon_chemin == OUEST) {
        printf("C'est un déplacement horizontal.\n");
    }

    mon_chemin = NORD;
    deplacer(mon_chemin);

    return 0;
}
```

En résumé :

- Utilisez les **structures (`struct`)** pour regrouper des données de types différents qui doivent toutes exister simultanément.
- Utilisez les **unions (`union`)** pour économiser de la mémoire lorsque seul un type de donnée parmi plusieurs est valide à un instant T (en gérant explicitement le type actif).
- Utilisez les **énumérations (`enum`)** pour créer des ensembles de constantes entières nommées afin d'améliorer la lisibilité et la maintenabilité de votre code.

---

15. Pointeurs Avancés et Pointeurs de Fonction

Nous avons vu les bases des pointeurs, leur arithmétique et leur relation avec les tableaux et les structures. Cette section explore des concepts un peu plus avancés mais très utiles : les pointeurs vers d'autres pointeurs, les tableaux de pointeurs, et surtout, les pointeurs de fonction, qui permettent une flexibilité et une généricité accrues.

Pointeurs vers pointeurs (`type **ptr`)

Tout comme un pointeur peut stocker l'adresse d'une variable ordinaire (comme un `int` ou un `float`), un pointeur peut aussi stocker l'**adresse d'un autre pointeur**. On parle alors de **pointeur vers pointeur** (ou pointeur double, triple, etc.).

Déclaration : On utilise des astérisques supplémentaires. `type **nom_pointeur_double;` // Pointeur vers un pointeur vers 'type' `type ***nom_pointeur_triple;` // Pointeur vers un pointeur vers un pointeur vers 'type'

Exemple : `int **ptr_vers_ptr_int;` déclare une variable `ptr_vers_ptr_int` qui peut contenir l'adresse d'une variable de type `int *`.

Utilité :

- **Modifier un pointeur passé en argument à une fonction :** Si une fonction doit changer l'adresse contenue dans un pointeur qui lui est passé (par exemple, pour allouer de la mémoire et retourner le pointeur via un paramètre), elle doit recevoir l'adresse *de ce pointeur*, donc un pointeur vers pointeur.
- **Tableaux de chaînes de caractères (style C) :** Comme nous le verrons, `argv` dans `main(int argc, char *argv[])` est un exemple de tableau de pointeurs (`char *[]`), et on peut aussi le manipuler avec un pointeur double (`char **`).
- **Tableaux 2D alloués dynamiquement :** Une technique courante pour créer des tableaux 2D dynamiques consiste à allouer un tableau de pointeurs, puis à allouer chaque ligne (chaque sous-tableau) séparément.

Déréférencement : Pour accéder à la valeur finale, il faut déréférencer autant de fois qu'il y a d'astérisques :

- Si `ptr_vers_ptr` est de type `int **` et pointe vers `ptr` (de type `int *`), qui pointe vers `var` (de type `int`) :
  - `*ptr_vers_ptr` donne la valeur de `ptr` (l'adresse de `var`).
  - `**ptr_vers_ptr` donne la valeur de `var` (l'entier).

```
#include <stdio.h>
#include <stdlib.h> // Pour malloc et free

// Fonction qui alloue de la mémoire pour un entier et retourne son adresse
// via un pointeur vers pointeur.
void allouer_entier(int **ptr_adresse) {
    // ptr_adresse contient l'adresse du pointeur 'mon_ptr' dans main.
    // *ptr_adresse permet donc de modifier 'mon_ptr' lui-même.

    // Alloue de la mémoire pour UN int
    *ptr_adresse = (int*)malloc(sizeof(int));

    if (*ptr_adresse == NULL) {
        perror("Erreur d'allocation mémoire");
        // Dans un vrai programme, gérer l'erreur plus proprement
    } else {
        printf("  (Dans allouer_entier) Mémoire allouée à : %p\n",
(void*)*ptr_adresse);
    }
}

int main() {
    int valeur = 42;
    int *ptr_simple = &valeur; // Pointeur simple vers 'valeur'
    int **ptr_double;          // Pointeur vers un pointeur d'entier

    // Faire pointer ptr_double vers ptr_simple
    ptr_double = &ptr_simple;

    printf("Adresse de 'valeur'     : %p\n", (void*)&valeur);
    printf("Adresse stockée dans ptr_simple : %p\n", (void*)ptr_simple);
    printf("Adresse de 'ptr_simple' : %p\n", (void*)&ptr_simple);
```

```
    printf("Adresse stockée dans ptr_double: %p\n", (void*)ptr_double);

    printf("\nAccès via les pointeurs :\n");
    printf("Valeur via ptr_simple  : %d\n", *ptr_simple);
    printf("Valeur via ptr_double (1 déréf: *ptr_double) : %p (adresse de
valeur)\n", (void*)*ptr_double);
    printf("Valeur via ptr_double (2 déréf: **ptr_double): %d\n",
**ptr_double);

    // Modifier la valeur originale via le pointeur double
    **ptr_double = 50;
    printf("\nAprès **ptr_double = 50 :\n");
    printf("Nouvelle valeur de 'valeur' : %d\n", valeur); // valeur a été
modifiée

    // --- Utilisation avec une fonction modifiant un pointeur ---
    int *mon_ptr = NULL; // Pointeur initialisé à NULL
    printf("\nAvant appel allouer_entier, mon_ptr = %p\n", (void*)mon_ptr);

    // On passe l'adresse de mon_ptr (&mon_ptr) à la fonction
    allouer_entier(&mon_ptr);

    printf("Après appel allouer_entier, mon_ptr = %p\n", (void*)mon_ptr);

    // Vérifier si l'allocation a réussi dans la fonction
    if (mon_ptr != NULL) {
        *mon_ptr = 12345; // Assigner une valeur à la mémoire allouée
        printf("Valeur dans la mémoire allouée : %d\n", *mon_ptr);
        free(mon_ptr); // Libérer la mémoire
        mon_ptr = NULL;
    }

    return 0;
}
```

Tableaux de pointeurs

Tout comme on peut avoir des tableaux d'entiers ou de structures, on peut aussi avoir des **tableaux de pointeurs**. Chaque élément du tableau est un pointeur qui peut stocker l'adresse d'une variable (ou d'un bloc mémoire) du type pointé.

Déclaration : `type *nom_tableau_pointeurs[TAILLE];`

Exemple : `int *tableau_ptr_int[5];` déclare un tableau contenant 5 pointeurs, chacun pouvant pointer vers un `int`. `char *liste_noms[10];` déclare un tableau pouvant contenir 10 pointeurs vers `char`, typiquement utilisé pour stocker les adresses de 10 chaînes de caractères.

Utilisation courante : Tableaux de chaînes C'est l'utilisation la plus fréquente des tableaux de pointeurs. Au lieu de créer un tableau 2D de `char` de taille fixe (ex: `char noms[10][50];`), on peut créer un tableau de `char *` et faire pointer chaque élément vers une chaîne de caractères (soit un littéral de chaîne, soit une chaîne allouée dynamiquement).

Avantages :

- Flexibilité : Chaque chaîne peut avoir une longueur différente.
- Efficacité mémoire (potentielle) : Si les chaînes ont des longueurs très variables, on n'alloue que la mémoire nécessaire pour chaque chaîne + le tableau de pointeurs, au lieu d'un grand bloc rectangulaire potentiellement sous-utilisé.

Inconvénients :

- La mémoire pour les chaînes (si non littérales) et pour le tableau de pointeurs peut être dispersée.
- La gestion de l'allocation/libération (si dynamique) est plus complexe.

```c
#include <stdio.h>

int main() {
    // Tableau de pointeurs vers des littéraux de chaîne
    // Les chaînes "Lundi", "Mardi", etc. sont stockées
    // quelque part en mémoire (souvent en lecture seule).
    // Le tableau 'jours' contient juste les adresses de ces chaînes.
    const char *jours[7] = {
        "Lundi", "Mardi", "Mercredi", "Jeudi",
        "Vendredi", "Samedi", "Dimanche"
    };
    // Note: 'const char *' car on pointe vers des littéraux qui ne doivent
pas être modifiés.

    printf("Les jours de la semaine :\n");
    for (int i = 0; i < 7; i++) {
        // jours[i] est un pointeur char* (l'adresse de la chaîne)
        // %s attend un char*
        printf("  Jour %d : %s (à l'adresse %p)\n", i + 1, jours[i],
(void*)jours[i]);
    }

    // Exemple avec tableau de pointeurs vers int (moins courant, mais
possible)
    int a = 10, b = 20, c = 30;
    int *ptr_vers_int[3]; // Tableau de 3 pointeurs vers int

    ptr_vers_int[0] = &a; // Le premier pointeur pointe vers a
    ptr_vers_int[1] = &b; // Le deuxième pointe vers b
    ptr_vers_int[2] = &c; // Le troisième pointe vers c

    printf("\nValeurs via tableau de pointeurs d'entiers :\n");
    for (int i = 0; i < 3; i++) {
        // ptr_vers_int[i] est un int*
        // *ptr_vers_int[i] déréférence ce pointeur pour obtenir la valeur
int
        printf("  Valeur pointée par ptr_vers_int[%d] : %d\n", i,
*ptr_vers_int[i]);
    }
```

```
    // Modifier une valeur via le tableau de pointeurs
    *ptr_vers_int[1] = 25; // Modifie la valeur de 'b'
    printf("Nouvelle valeur de b : %d\n", b); // Affiche 25

    return 0;
}
```

Assignation : Pour faire pointer un pointeur de fonction vers une fonction spécifique, on lui assigne simplement le **nom de la fonction** (sans les parenthèses `()`). Le nom d'une fonction, comme le nom d'un tableau, se "désintègre" en une adresse (l'adresse du début du code de la fonction) dans ce contexte. On peut aussi utiliser explicitement l'opérateur `&` devant le nom de la fonction, mais c'est moins courant.

Exemple d'assignation (en supposant que `addition` et `soustraction` sont des fonctions compatibles et `operation_ptr` un pointeur de fonction déclaré comme `int (*operation_ptr)(int, int);`):
`operation_ptr = addition;` // operation_ptr pointe maintenant vers la fonction addition
`operation_ptr = &soustraction;` // Fonctionne aussi, pointe vers soustraction

Appel via pointeur : Pour appeler la fonction pointée, on peut utiliser deux syntaxes équivalentes :

1. **Déréférencement explicite :** Déréférencer le pointeur (`*operation_ptr`) pour obtenir la fonction, puis l'appeler avec les arguments entre parenthèses. Les parenthèses autour de `*operation_ptr` sont nécessaires à cause de la priorité des opérateurs. `resultat = (*operation_ptr)(10, 5);`
2. **Syntaxe simplifiée :** Utiliser directement le nom du pointeur comme s'il s'agissait du nom de la fonction. C'est la syntaxe la plus courante et la plus lisible. `resultat = operation_ptr(10, 5);`

Le compilateur comprend que `operation_ptr` est un pointeur de fonction et génère le code pour appeler la fonction dont l'adresse est stockée dedans.

(Voir l'exemple de code `code_pointeur_fonction_base_section15` pour une illustration pratique de la déclaration, l'assignation et l'appel).

```
#include <stdio.h>

// Définition de deux fonctions compatibles
int addition(int x, int y) {
    printf("  (Appel de addition)\n");
    return x + y;
}

int soustraction(int x, int y) {
    printf("  (Appel de soustraction)\n");
    return x - y;
}

// Déclaration d'un type pointeur de fonction pour plus de clarté
(optionnel)
typedef int (*OperationArithmetique)(int, int);
```

```c
int main() {
    // Déclaration d'un pointeur de fonction
    int (*ptr_op)(int, int);
    // Ou en utilisant le typedef :
    // OperationArithmetique ptr_op;

    int a = 100, b = 25;
    int resultat;

    // 1. Assignation du pointeur à la fonction 'addition'
    ptr_op = addition; // ou ptr_op = &addition;
    printf("Appel via ptr_op pointant vers addition :\n");
    // Appel via la syntaxe simplifiée (préférée)
    resultat = ptr_op(a, b);
    printf("Résultat = %d\n", resultat); // 125

    // Appel via la syntaxe de déréférencement explicite
    resultat = (*ptr_op)(a, b);
    printf("Résultat (déréférencement) = %d\n", resultat); // 125

    // 2. Réassignation du pointeur à la fonction 'soustraction'
    ptr_op = soustraction;
    printf("\nAppel via ptr_op pointant vers soustraction :\n");
    resultat = ptr_op(a, b);
    printf("Résultat = %d\n", resultat); // 75

    return 0;
}
```

Passage en argument (Callbacks)

L'une des utilisations les plus puissantes des pointeurs de fonction est de les passer comme **arguments** à d'autres fonctions. La fonction qui reçoit le pointeur peut alors **appeler** la fonction pointée sans connaître son nom exact à l'avance. C'est le mécanisme de **callback** (rappel).

Cela permet de créer des fonctions génériques qui peuvent être personnalisées par l'appelant.

Exemple : Une fonction générique `appliquer_operation` On peut écrire une fonction qui prend deux entiers et un pointeur vers une opération (comme `addition` ou `soustraction`) et qui applique cette opération aux deux entiers.

```c
#include <stdio.h>

// Fonctions d'opération
int addition(int x, int y) { return x + y; }
int soustraction(int x, int y) { return x - y; }
int multiplication(int x, int y) { return x * y; }

// Type pointeur de fonction pour la clarté
typedef int (*FonctionOperation)(int, int);
```

```c
// Fonction générique qui applique une opération passée en argument
void appliquer_et_afficher(int a, int b, FonctionOperation
operation_a_faire) {
    // Vérifier si le pointeur de fonction n'est pas NULL
    if (operation_a_faire != NULL) {
        int resultat = operation_a_faire(a, b); // Appel de la fonction
pointée
        printf("Résultat de l'opération sur %d et %d : %d\n", a, b,
resultat);
    } else {
        printf("Aucune opération fournie.\n");
    }
}

int main() {
    int v1 = 20, v2 = 7;

    printf("Application de l'addition :\n");
    appliquer_et_afficher(v1, v2, addition); // Passe l'adresse de
'addition'

    printf("\nApplication de la soustraction :\n");
    appliquer_et_afficher(v1, v2, soustraction); // Passe l'adresse de
'soustraction'

    printf("\nApplication de la multiplication :\n");
    appliquer_et_afficher(v1, v2, multiplication); // Passe l'adresse de
'multiplication'

    return 0;
}
```

Exemple : `qsort()` de `<stdlib.h>`

Un exemple classique et très utile de callback dans la bibliothèque standard C est la fonction `qsort` (Quick Sort), définie dans `<stdlib.h>`. Elle permet de trier un tableau de n'importe quel type d'éléments, à condition que vous lui fournissiez une **fonction de comparaison** personnalisée.

Syntaxe simplifiée de `qsort` : `void qsort(void *base, size_t nmemb, size_t size, int (*compar)(const void *, const void *));`

- `base`: Pointeur `void*` vers le **premier élément** du tableau à trier.
- `nmemb`: Le **nombre d'éléments** dans le tableau (`size_t`).
- `size`: La **taille en octets** de *chaque élément* du tableau (`size_t`, typiquement `sizeof(element_type)`).
- `compar`: Un **pointeur vers votre fonction de comparaison**. Cette fonction doit :
  - Prendre deux arguments de type `const void *`. Ce sont des pointeurs génériques vers les deux éléments du tableau que `qsort` veut comparer.
  - Retourner un `int` :

- `< 0` si le premier élément est "inférieur" au second.
- `0` si les deux éléments sont considérés comme "égaux".
- `> 0` si le premier élément est "supérieur" au second.

À l'intérieur de votre fonction de comparaison, vous devrez **caster** les pointeurs `void *` vers le type de pointeur correct pour vos éléments (ex: `(const int *)`, `(const struct Personne *)`), puis déréférencer pour accéder aux valeurs et effectuer la comparaison.

```c
#include <stdio.h>
#include <stdlib.h> // Pour qsort
#include <string.h> // Pour strcmp

// --- Comparaison pour des entiers (ordre croissant) ---
int comparer_entiers(const void *pa, const void *pb) {
    // Caster les void* en pointeurs vers le type réel (int*)
    const int *a = (const int *)pa;
    const int *b = (const int *)pb;

    // Déréférencer pour obtenir les valeurs entières
    int val_a = *a;
    int val_b = *b;

    // Comparer et retourner <0, 0, ou >0
    if (val_a < val_b) return -1;
    if (val_a > val_b) return 1;
    return 0;
    // Ou plus concise ment: return (val_a > val_b) - (val_a < val_b);
    // Ou encore : return val_a - val_b; (peut causer un overflow si les
nombres sont très grands/petits)
}

// --- Comparaison pour des chaînes de caractères (ordre alphabétique) ---
// Les éléments du tableau sont des char* (pointeurs vers char)
int comparer_chaines(const void *pa, const void *pb) {
    // Caster les void* en pointeurs vers le type réel (char**)
    // car le tableau contient des char*, pa/pb pointent vers ces char*
    const char **str_a = (const char **)pa;
    const char **str_b = (const char **)pb;

    // Déréférencer une fois pour obtenir les pointeurs char*
    // puis utiliser strcmp pour comparer les chaînes pointées
    return strcmp(*str_a, *str_b);
}

// --- Structure pour l'exemple suivant ---
typedef struct {
    char nom[30];
    int age;
} Personne;

// --- Comparaison pour des structures Personne (par âge croissant) ---
int comparer_personnes_par_age(const void *pa, const void *pb) {
```

```
    // Caster en pointeurs vers le type réel (Personne*)
    const Personne *p1 = (const Personne *)pa;
    const Personne *p2 = (const Personne *)pb;

    // Comparer les membres 'age'
    if (p1->age < p2->age) return -1;
    if (p1->age > p2->age) return 1;
    return 0;
    // Ou : return p1->age - p2->age; (attention à l'overflow potentiel)
}


int main() {
    // 1. Trier un tableau d'entiers
    int nombres[] = {50, 20, 80, 10, 40, 90, 30};
    size_t nb_nombres = sizeof(nombres) / sizeof(nombres[0]);

    printf("Tableau d'entiers avant tri: ");
    for(size_t i=0; i<nb_nombres; ++i) printf("%d ", nombres[i]);
    printf("\n");

    qsort(nombres, nb_nombres, sizeof(int), comparer_entiers);

    printf("Tableau d'entiers après tri: ");
    for(size_t i=0; i<nb_nombres; ++i) printf("%d ", nombres[i]);
    printf("\n\n");

    // 2. Trier un tableau de chaînes (tableau de pointeurs char*)
    const char *mots[] = {"banane", "pomme", "orange", "fraise",
"abricot"};
    size_t nb_mots = sizeof(mots) / sizeof(mots[0]);

    printf("Tableau de chaînes avant tri: ");
    for(size_t i=0; i<nb_mots; ++i) printf("%s ", mots[i]);
    printf("\n");

    qsort(mots, nb_mots, sizeof(char *), comparer_chaines); //
sizeof(char*) !!

    printf("Tableau de chaînes après tri: ");
    for(size_t i=0; i<nb_mots; ++i) printf("%s ", mots[i]);
    printf("\n\n");

    // 3. Trier un tableau de structures
    Personne personnes[] = {{"Alice", 30}, {"Bob", 25}, {"Charlie", 35}};
    size_t nb_personnes = sizeof(personnes) / sizeof(personnes[0]);

     printf("Tableau de personnes avant tri (par âge): \n");
    for(size_t i=0; i<nb_personnes; ++i) printf("  %s (%d ans)\n",
personnes[i].nom, personnes[i].age);

    qsort(personnes, nb_personnes, sizeof(Personne),
comparer_personnes_par_age);
```

```
    printf("Tableau de personnes après tri (par âge): \n");
    for(size_t i=0; i<nb_personnes; ++i) printf("  %s (%d ans)\n",
personnes[i].nom, personnes[i].age);


    return 0;
}
```

Les pointeurs de fonction sont un concept puissant qui permet une grande flexibilité et la création de code plus générique et réutilisable en C. Ils sont à la base de nombreux mécanismes comme les callbacks, les tables de fonctions virtuelles (simulées en C), et les fonctions de bibliothèque configurables comme `qsort`.

---

16. Le Préprocesseur C

Avant que votre code source C ne soit réellement compilé en code machine par le compilateur C, il passe par une étape préliminaire essentielle appelée **préprocessing**. Cette étape est réalisée par un programme distinct appelé le **préprocesseur C** (souvent intégré à l'outil compilateur comme GCC ou Clang, mais conceptuellement séparé).

Le préprocesseur **ne comprend pas la syntaxe C** elle-même. Son rôle est d'effectuer des **manipulations textuelles** sur votre fichier source (`.c`) en fonction de directives spéciales, les **directives de préprocesseur**, qui commencent toutes par le symbole dièse (`#`).

Les tâches principales du préprocesseur sont :

1. **Inclusion de fichiers :** Insérer le contenu d'autres fichiers (généralement les fichiers d'en-tête `.h`) dans le fichier source actuel (directive `#include`).
2. **Substitution de macros :** Remplacer des identifiants symboliques (macros) par le texte qui leur est associé (directive `#define`).
3. **Compilation conditionnelle :** Inclure ou exclure sélectivement des portions de code du fichier source qui sera transmis au compilateur, en fonction de certaines conditions (directives `#if`, `#ifdef`, `#ifndef`, etc.).
4. **Suppression des commentaires :** Les commentaires (`//...` et `/*...*/`) sont retirés du code.

Le résultat de cette étape est un fichier source temporaire, souvent appelé **unité de traduction** (translation unit), qui est ensuite passé au compilateur C proprement dit pour l'analyse syntaxique, sémantique et la génération du code assembleur/machine.

Directive `#include` : Inclusion de fichiers d'en-tête

C'est la directive que vous avez déjà rencontrée le plus souvent. Elle demande au préprocesseur de trouver le fichier spécifié et d'en **insérer le contenu textuel complet** à l'endroit exact où la directive `#include` se trouve dans votre code. Ces fichiers inclus sont presque toujours des **fichiers d'en-tête** (header files), qui portent conventionnellement l'extension `.h`.

Il existe deux formes syntaxiques pour `#include` :

1. `#include <fichier_systeme.h>`

- Utilise des **chevrons** (`< >`).
- Est utilisée pour inclure les fichiers d'en-tête de la **bibliothèque standard C** (comme `stdio.h`, `stdlib.h`, `string.h`, `math.h`, `stdbool.h`, `ctype.h`, etc.) ou d'autres bibliothèques externes installées dans des emplacements système standard connus du compilateur.
- Le préprocesseur cherche ces fichiers dans une liste prédéfinie de répertoires système (souvent spécifiés par les variables d'environnement ou les options du compilateur).

2. `#include "mon_fichier.h"`

- Utilise des **guillemets doubles** (`" "`).
- Est utilisée pour inclure les fichiers d'en-tête que **vous avez créés** pour votre propre projet et qui se trouvent généralement dans le même répertoire que votre fichier source `.c`, ou dans un sous-répertoire relatif.
- Le préprocesseur cherche généralement d'abord ce fichier dans le **répertoire du fichier source actuel**. S'il ne le trouve pas là, il peut ensuite chercher dans les répertoires système (le comportement exact peut varier légèrement selon le compilateur).

Utilité : Les fichiers d'en-tête contiennent typiquement les **déclarations** nécessaires pour utiliser des éléments définis ailleurs :

- Les **prototypes** des fonctions (pour que le compilateur sache comment les appeler correctement).
- Les définitions de **structures**, **unions**, et **énumérations**.
- Les définitions de **macros** et de **constantes symboliques** (`#define`).
- Les alias de type (`typedef`).
- Les déclarations de variables globales `extern` (pour indiquer qu'une variable globale est définie dans un autre fichier `.c`).

Inclure un fichier d'en-tête permet à votre fichier `.c` actuel de connaître et d'utiliser correctement les fonctions, types, et constantes définis ailleurs, assurant ainsi la cohérence et la modularité du projet.

```c
// Inclusion des en-têtes standards nécessaires pour printf et exit
#include <stdio.h>  // Pour printf
#include <stdlib.h> // Pour exit, EXIT_SUCCESS

// Supposons que nous ayons créé nos propres fichiers pour un module
"utils"
// utils.h contiendrait les déclarations (prototypes, etc.)
// utils.c contiendrait les définitions (le code des fonctions)
// #include "utils.h" // On inclurait notre propre header avec des
guillemets

// Prototype qui serait typiquement DANS "utils.h"
// void fonction_utilitaire(int valeur);

int main() {
    printf("Utilisation de printf de <stdio.h>\n");

    // Si "utils.h" était inclus et la fonction définie dans utils.c :
    // fonction_utilitaire(10);
```

```
    printf("Fin normale du programme.\n");
    return EXIT_SUCCESS; // Constante définie dans <stdlib.h> (équivaut
souvent à 0)
}

/*
// Contenu possible de utils.h (exemple)
#ifndef UTILS_H_INCLUDED
#define UTILS_H_INCLUDED

void fonction_utilitaire(int valeur);

#endif // UTILS_H_INCLUDED
*/

/*
// Contenu possible de utils.c (exemple)
#include <stdio.h> // Pour printf dans la fonction
#include "utils.h" // Inclut son propre header

void fonction_utilitaire(int valeur) {
    printf("Fonction utilitaire appelée avec la valeur : %d\n", valeur);
}
*/
```

Directive `#define` : Définir des constantes et des macros

La directive `#define` est l'outil principal du préprocesseur pour effectuer des **substitutions de texte**. Avant la compilation, le préprocesseur remplace **littéralement** toutes les occurrences du nom de la macro (le symbole défini) par le texte de remplacement associé.

Il y a deux formes principales d'utilisation :

1. Constantes Symboliques (Macros Objets) : C'est l'utilisation la plus simple et la plus sûre de `#define`. Elle permet de donner un nom symbolique (par convention, en majuscules) à une valeur constante (nombre, caractère, chaîne littérale, ou même une petite expression).

Syntaxe : `#define NOM_SYMBOLE texte_de_remplacement`

- `NOM_SYMBOLE`: Le nom que vous utiliserez dans votre code.
- `texte_de_remplacement`: Le texte littéral qui substituera *chaque* occurrence de `NOM_SYMBOLE` dans le code source, jusqu'à la fin de la ligne du `#define`.
- **Important :** Il n'y a **pas de point-virgule** à la fin de la ligne `#define`, sauf si vous voulez explicitement que le point-virgule fasse partie du texte de remplacement (ce qui est rare et généralement une erreur pour les constantes).

Avantages :

- **Lisibilité :** Utiliser `TAUX_INTERET` est plus clair que `0.05`.

- **Maintenabilité :** Si la valeur doit changer (par exemple, la taille maximale d'un tableau), vous ne la modifiez qu'à un seul endroit (la ligne `#define`), et le changement est répercuté partout où le symbole est utilisé.

Exemples : `#define PI 3.14159` `#define TAILLE_BUFFER 1024` `#define MESSAGE_ACCUEIL "Bienvenue !"` `#define VRAI 1` `#define FAUX 0` (Bien qu'il soit préférable d'utiliser `<stdbool.h>`)

```c
#include <stdio.h>

// Définition de constantes symboliques
#define MAX_UTILISATEURS 100
#define NOM_FICHIER_LOG "application.log"
#define CODE_ERREUR_OK 0
#define PI 3.14159265

// Notez l'absence de ; à la fin des #define

int main() {
    int utilisateurs_connectes = 57;
    double rayon = 2.5;
    double circonference;

    if (utilisateurs_connectes < MAX_UTILISATEURS) { // MAX_UTILISATEURS
sera remplacé par 100
        printf("Il reste de la place pour de nouveaux utilisateurs.\n");
    }

    circonference = 2 * PI * rayon; // PI sera remplacé par 3.14159265
    printf("La circonférence pour un rayon de %.2f est %.4f\n", rayon,
circonference);

    printf("Les logs sont écrits dans : %s\n", NOM_FICHIER_LOG); //
Remplacement par la chaîne

    return CODE_ERREUR_OK; // Remplacement par 0
}
```

2. Macros de type fonction (Macros avec paramètres) : `#define` peut aussi prendre des arguments entre parenthèses, créant ainsi des macros qui *ressemblent* à des appels de fonction mais qui fonctionnent par **substitution de texte brute**, pas par un véritable appel de fonction.

Syntaxe : `#define NOM_MACRO(param1, param2, ...) (corps_de_remplacement_utilisant_params)`

- Il ne doit **pas y avoir d'espace** entre `NOM_MACRO` et la parenthèse ouvrante `(`.
- `param1`, `param2`, ... : Noms des paramètres utilisés dans le corps de remplacement.
- `corps_de_remplacement...` : Une expression ou un bloc de code (souvent une expression unique) qui utilise les paramètres. Le préprocesseur remplacera littéralement les noms des paramètres par les arguments fournis lors de l'appel de la macro.

Exemple : `#define CARRE(x) ((x) * (x))`

Lorsqu'on écrit `y = CARRE(a + b);` dans le code, le préprocesseur le transforme textuellement en `y = (((a + b)) * ((a + b)));` avant que le compilateur ne voie le code.

Précautions avec les macros de type fonction (TRÈS IMPORTANT) : Les macros de type fonction sont puissantes mais peuvent être source d'erreurs subtiles si elles ne sont pas écrites avec une extrême prudence :

- **Parenthèses OBLIGATOIRES :** Mettez **toujours** des parenthèses autour de **chaque occurrence** de chaque paramètre dans le corps de remplacement, ET mettez des parenthèses autour de **l'expression entière** du corps de remplacement. Cela est crucial pour éviter les problèmes de priorité des opérateurs lorsque les arguments sont des expressions complexes.
  - **Mauvais :** `#define CARRE(x) x * x` -> `CARRE(a + b)` devient `a + b * a + b` (incorrect).
  - **Bon :** `#define CARRE(x) ((x) * (x))` -> `CARRE(a + b)` devient `(((a + b)) * ((a + b)))` (correct).
- **Effets de bord / Évaluations multiples :** Si un argument passé à une macro a un effet de bord (comme `i++`, `getchar()`, ou un appel de fonction qui modifie quelque chose), cet argument peut être **évalué plusieurs fois** si le paramètre correspondant apparaît plusieurs fois dans le corps de la macro. C'est rarement le comportement souhaité.
  - Exemple : `#define MAX(a, b) ((a) > (b) ? (a) : (b))`
  - Si on appelle `m = MAX(x++, y++);`, et si `x` est initialement plus grand que `y`, alors `x++` sera évalué **deux fois** (une fois dans la condition `(a) > (b)` et une fois dans le résultat `(a)`), ce qui incrémentera `x` de 2 au lieu de 1.
- **Pas de vérification de type :** Le préprocesseur effectue une substitution de texte brute. Il ne vérifie pas si les types des arguments passés à la macro sont logiques ou compatibles avec les opérations effectuées dans le corps de la macro.
- **Messages d'erreur :** Les erreurs de compilation résultant d'une macro mal écrite peuvent être difficiles à comprendre, car le compilateur voit le code *après* la substitution par le préprocesseur, pas la macro elle-même.

À cause de ces pièges, pour des opérations simples qui ressemblent à des fonctions et où la performance est critique, on préfère souvent aujourd'hui (depuis C99) les **fonctions `inline`**. Elles offrent les avantages potentiels de performance (en évitant le surcoût d'un appel de fonction) tout en conservant la sémantique et la sécurité d'un véritable appel de fonction (évaluation unique des arguments, vérification de type). Les macros restent cependant utiles pour des tâches très simples, la manipulation de code conditionnelle, ou la génération de code répétitif.

```c
#include <stdio.h>

// Macro pour calculer le carré - AVEC parenthèses (bonne pratique)
#define CARRE(n) ((n) * (n))

// Macro pour trouver le maximum - AVEC parenthèses (bonne pratique)
// ATTENTION aux effets de bord avec cette macro !
#define MAX(a, b) ((a) > (b) ? (a) : (b))
```

```c
// Macro pour afficher un message de débogage avec fichier et ligne
// __FILE__ et __LINE__ sont des macros prédéfinies par le préprocesseur
#define DEBUG_MSG(message) printf("[DEBUG] %s:%d - %s\n", __FILE__,
__LINE__, message)

// Macro multi-lignes (utiliser '\' pour continuer sur la ligne suivante)
// Attention: ceci n'est PAS une fonction, c'est une substitution de bloc.
// Il n'y a pas de portée locale propre à la macro.
#define ECHANGER_INT(a, b) do { \
                             int temp = a; \
                             a = b; \
                             b = temp; \
                           } while (0)
// Le 'do {...} while(0)' est une astuce courante pour créer une macro
// qui se comporte comme une instruction unique (peut être utilisée dans un
if sans accolades).

int main() {
    int v1 = 5;
    int v2 = -3;
    int res;

    res = CARRE(v1);     // Devient res = ((5) * (5));
    printf("CARRE(%d) = %d\n", v1, res); // 25

    res = CARRE(v1 + 1); // Devient res = (((5 + 1)) * ((5 + 1)));
    printf("CARRE(%d + 1) = %d\n", v1, res); // 36

    res = MAX(v1, 10);   // Devient res = ((5) > (10) ? (5) : (10));
    printf("MAX(%d, 10) = %d\n", v1, res); // 10

    // --- Test de l'effet de bord de MAX ---
    int x = 3, y = 5;
    printf("\nAvant MAX: x = %d, y = %d\n", x, y);
    // MAX(x++, y++) devient ((x++) > (y++) ? (x++) : (y++))
    // Comme 3 < 5, la partie fausse est évaluée: y++
    // Mais y++ a déjà été évalué une fois dans la condition !
    res = MAX(x++, y++);
    printf("MAX(x++, y++) = %d\n", res); // Affiche 5 (la valeur de y AVANT
le second incrément)
    printf("Après MAX: x = %d, y = %d\n", x, y); // Affiche x=4, y=7 (x++
une fois, y++ deux fois)
    // Conclusion: NE PAS utiliser d'arguments avec effets de bord dans MAX
!

    // --- Utilisation de DEBUG_MSG ---
    DEBUG_MSG("Début de la section critique.");

    // --- Utilisation de ECHANGER_INT ---
    int n1 = 10, n2 = 20;
    printf("\nAvant échange: n1 = %d, n2 = %d\n", n1, n2);
    ECHANGER_INT(n1, n2); // Le bloc do-while est substitué ici
    printf("Après échange: n1 = %d, n2 = %d\n", n1, n2);
```

```
    return 0;
}
```

Directive `#undef`

La directive `#undef` permet d'"annuler" ou de "supprimer" une définition de macro (objet ou fonction) faite précédemment avec `#define`.

Syntaxe : `#undef NOM_MACRO_OU_CONSTANTE`

Après un `#undef`, le nom spécifié n'est plus reconnu comme une macro par le préprocesseur (jusqu'à ce qu'il soit éventuellement redéfini par un autre `#define`).

Son utilité est relativement limitée mais peut servir dans des cas spécifiques :

- Éviter des conflits de noms si un même nom de macro pourrait être défini différemment dans des en-têtes inclus.
- Changer la définition d'une macro pour une portion limitée du code (souvent en combinaison avec la compilation conditionnelle).
- S'assurer qu'une macro système potentiellement problématique n'est pas définie.

Exemple simple d'utilisation : `#define TAILLE 10 // ... utiliser TAILLE ... #undef TAILLE // TAILLE n'est plus définie à partir d'ici`

`#ifdef TAILLE // Sera faux maintenant // ... code ... #endif`

Macros prédéfinies

Le standard C et les compilateurs définissent un certain nombre de macros utiles qui sont automatiquement disponibles, sans nécessiter de `#define` de votre part. Les plus courantes (standard C) sont :

- `__FILE__` : Se substitue par une chaîne de caractères contenant le nom du fichier source actuel.
- `__LINE__` : Se substitue par un entier représentant le numéro de la ligne actuelle dans le fichier source.
- `__DATE__` : Se substitue par une chaîne contenant la date de compilation (ex: `"Apr 27 2025"`).
- `__TIME__` : Se substitue par une chaîne contenant l'heure de compilation (ex: `"15:30:00"`).
- `__STDC__` : Défini à 1 si le compilateur se conforme strictement au standard C (peut ne pas être défini ou avoir une autre valeur sinon).
- `__STDC_VERSION__` (C99+) : Défini comme une constante entière longue (ex: `201112L` pour C11, `201710L` pour C17/C18) indiquant la version du standard C supportée. Permet la compilation conditionnelle basée sur la version du standard.

Ces macros sont particulièrement utiles pour les messages de débogage, la journalisation (logging), ou la compilation conditionnelle.

(L'exemple de code `code_define_macros_section16` utilisait déjà `__FILE__` et `__LINE__` dans la macro `DEBUG_MSG`).

Compilation Conditionnelle (`#if`, `#ifdef`, `#ifndef`, `#else`, `#elif`, `#endif`)

Ces directives permettent d'**inclure ou d'exclure** des portions de code source du processus de compilation, en fonction de conditions évaluées *par le préprocesseur* (donc avant la compilation C elle-même). C'est un outil extrêmement puissant et fréquemment utilisé pour :

- **Portabilité :** Écrire du code spécifique à un système d'exploitation (ex: Windows vs Linux vs macOS) ou à une architecture matérielle. Des macros prédéfinies par le compilateur (comme `_WIN32`, `__linux__`, `__APPLE__`) sont souvent utilisées ici.
- **Débogage :** Inclure du code de débogage (affichages `printf`, assertions `assert`, etc.) uniquement lors de la compilation en mode "debug" (souvent activé par la définition d'une macro comme `DEBUG` ou `_DEBUG`).
- **Gestion des fonctionnalités :** Activer ou désactiver des fonctionnalités optionnelles du programme via des macros définies à la compilation.
- **Header Guards :** Empêcher les inclusions multiples accidentelles d'un même fichier d'en-tête dans une unité de compilation (le cas d'usage le plus critique et universel).

Les directives principales :

- `#ifdef SYMBOLE` (if defined) : Le bloc de code jusqu'au prochain `#elif`, `#else` ou `#endif` est inclus si `SYMBOLE` a été défini précédemment (soit par `#define SYMBOLE`, soit par une option du compilateur comme `gcc -DSYMBOLE ...`).
- `#ifndef SYMBOLE` (if not defined) : Le bloc de code est inclus si `SYMBOLE` n'a **pas** été défini. Très utilisé pour les *header guards*.
- `#if expression_constante` : Le bloc de code est inclus si l'`expression_constante` est évaluée comme **vraie (non nulle)** par le préprocesseur. L'expression peut utiliser :
  - Des constantes entières (y compris les `enum` et les `#define` numériques).
  - Des opérateurs arithmétiques, relationnels, logiques, bitwise.
  - L'opérateur `defined(SYMBOLE)` ou `defined SYMBOLE` qui vaut 1 si `SYMBOLE` est défini, 0 sinon. C'est souvent plus flexible que `#ifdef`.
- `#elif expression_constante` (else if) : Propose une condition alternative à tester si toutes les conditions précédentes (`#if`, `#ifdef`, `#ifndef`, `#elif`) dans le même bloc étaient fausses.
- `#else` : Le bloc de code suivant est inclus si toutes les conditions précédentes (`#if`, `#ifdef`, `#ifndef`, `#elif`) dans le même bloc étaient fausses.
- `#endif` : Marque **obligatoirement** la fin d'un bloc conditionnel commencé par `#if`, `#ifdef`, ou `#ifndef`. Chaque `#if`/`#ifdef`/`#ifndef` doit avoir un `#endif` correspondant. Ils peuvent être imbriqués.

```c
#include <stdio.h>

// 1. Définir une macro pour activer le mode debug (peut aussi se faire
avec gcc -DDEBUG)
#define DEBUG_MODE 1 // Mettre à 0 pour désactiver

// 2. Définir une macro pour une fonctionnalité optionnelle
// #define USE_FEATURE_X

int main() {

    printf("Début du programme.\n");
```

```c
    // --- Condition basée sur #if et une macro numérique ---
    #if DEBUG_MODE == 1
        printf("[DEBUG] Le mode débogage est activé (via DEBUG_MODE).\n");
        fprintf(stderr, "[DEBUG] Message d'erreur simulé sur stderr.\n");
    #else
        printf("Mode débogage désactivé.\n");
    #endif // DEBUG_MODE == 1

    // --- Condition basée sur la définition d'une macro (#ifdef) ---
    #ifdef USE_FEATURE_X
        printf("La fonctionnalité X est activée.\n");
        // ... code spécifique à la fonctionnalité X ...
    #else
        printf("La fonctionnalité X est désactivée.\n");
    #endif // USE_FEATURE_X

    // --- Condition basée sur l'opérateur defined() (plus flexible) ---
    #if defined(__GNUC__) // __GNUC__ est souvent défini par GCC et Clang
        printf("Compilé avec GCC ou Clang (ou compatible).\n");
    #elif defined(_MSC_VER) // _MSC_VER est défini par le compilateur
Microsoft
        printf("Compilé avec Microsoft Visual C++.\n");
    #else
        printf("Compilateur non identifié (ni GCC/Clang ni MSVC).\n");
    #endif

    // --- Exemple avec #ifndef (typiquement pour valeurs par défaut) ---
    #ifndef BUFFER_SIZE
        #define BUFFER_SIZE 256 // Définit une taille par défaut si non
fournie
        printf("BUFFER_SIZE non défini, utilisation de la valeur par
défaut: %d\n", BUFFER_SIZE);
    #else
        printf("BUFFER_SIZE défini à : %d\n", BUFFER_SIZE);
    #endif // BUFFER_SIZE

    // Pour tester BUFFER_SIZE, compiler avec : gcc -DBUFFER_SIZE=1024 ...

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

Header Guards

L'utilisation la plus importante et la plus systématique de la compilation conditionnelle est la technique des **gardes d'inclusion** (header guards).

Problème : Si un fichier d'en-tête (`fichier.h`) est inclus plusieurs fois dans la même unité de compilation (`.c`), directement ou indirectement (par exemple, `a.c` inclut `b.h` et `c.h`, et `b.h` inclut aussi `c.h`), alors le

contenu de `fichier.h` (définitions de `struct`, `typedef`, prototypes, etc.) serait inséré plusieurs fois. Cela conduirait à des erreurs de compilation pour "redéfinition" de types ou de fonctions (si elles sont `inline`).

Solution : Header Guards On encadre tout le contenu *utile* d'un fichier d'en-tête (`mon_header.h`) par un triplet de directives de préprocesseur :

1. `#ifndef NOM_UNIQUE_POUR_CE_HEADER_H` : Vérifie si une macro spécifique (unique à ce fichier) n'a **pas** encore été définie.
2. `#define NOM_UNIQUE_POUR_CE_HEADER_H` : Si elle n'était pas définie, on la définit **immédiatement**.
3. `// ... tout le contenu du fichier .h ...`
4. `#endif /* NOM_UNIQUE_POUR_CE_HEADER_H */` : Marque la fin du bloc conditionnel.

Fonctionnement :

- La **première fois** que le préprocesseur rencontre `#include "mon_header.h"` dans un fichier `.c`, `NOM_UNIQUE_POUR_CE_HEADER_H` n'est pas défini. Le `#ifndef` est donc vrai. Le préprocesseur entre dans le bloc, définit `NOM_UNIQUE_POUR_CE_HEADER_H`, et traite tout le contenu du header.
- Si, plus tard dans le même fichier `.c`, le préprocesseur rencontre **à nouveau** `#include "mon_header.h"`, il évalue `#ifndef NOM_UNIQUE_POUR_CE_HEADER_H`. Cette fois, la macro *est* définie (lors de la première inclusion). Le `#ifndef` est donc faux. Le préprocesseur **saute** alors tout le contenu jusqu'au `#endif` correspondant, évitant ainsi la ré-inclusion et les erreurs de redéfinition.

Convention de nommage pour la macro : Le nom de la macro doit être **unique** pour chaque fichier d'en-tête. Une convention très répandue est d'utiliser le nom du fichier en majuscules, en remplaçant les points et autres caractères non alphanumériques par des underscores (_), et en ajoutant un suffixe comme `_H` ou `_INCLUDED`. Par exemple, pour `mon_module.h`, on utiliserait `MON_MODULE_H` ou `MON_MODULE_H_INCLUDED`.

```
/* Contenu typique d'un fichier "geometrie.h" */

#ifndef GEOMETRIE_H_INCLUDED // 1. Garde d'inclusion : si pas déjà
défini...
#define GEOMETRIE_H_INCLUDED // 2. ...le définir maintenant.

// --- Début du contenu réel de l'en-tête ---

// Inclure d'autres en-têtes si nécessaire pour les types utilisés ici
#include <math.h> // Pour sqrt dans une éventuelle fonction inline ou macro
#include <stdbool.h> // Si on utilise bool

// Définir des constantes liées à la géométrie
#define PI 3.141592653589793

// Définir des types (struct, typedef)
typedef struct {
    double x;
    double y;
} Point2D_t;
```

```c
typedef struct {
    Point2D_t centre;
    double rayon;
} Cercle_t;

// Déclarer des fonctions (prototypes) définies dans geometrie.c
double distance_points(Point2D_t p1, Point2D_t p2);
double aire_cercle(Cercle_t c);
bool point_dans_cercle(Point2D_t p, Cercle_t c);

// On pourrait aussi définir des fonctions 'static inline' ici
// (leur définition peut être dans le .h car 'static' leur donne un linkage interne)
/*
static inline double carre(double val) {
    return val * val;
}
*/

// --- Fin du contenu réel de l'en-tête ---

#endif /* GEOMETRIE_H_INCLUDED */ // 3. Fin de la garde d'inclusion
```

Alternative moderne : `#pragma once` De nombreux compilateurs modernes supportent également la directive non standard `#pragma once`. Placée au début d'un fichier d'en-tête, elle indique au compilateur de n'inclure ce fichier qu'une seule fois par unité de compilation, même s'il rencontre plusieurs `#include` pour ce fichier.

```c
#pragma once

// ... contenu du fichier .h ...
```

Avantages : Plus court, moins de risque d'erreur de frappe dans le nom de la macro. Inconvénients : Non standard C (bien que très largement supporté), potentiellement moins portable sur des compilateurs très anciens ou exotiques.

En pratique, beaucoup de projets utilisent `#pragma once` pour sa simplicité, mais la méthode `#ifndef`/`#define`/`#endif` reste la solution standard et universellement portable.

Autres Directives

Il existe quelques autres directives de préprocesseur moins fréquemment utilisées :

- **`#error message` :** Arrête immédiatement la compilation et affiche le `message` d'erreur fourni. Utile pour signaler des configurations de compilation incompatibles ou des conditions qui ne devraient jamais se produire au niveau du préprocesseur. `#if VERSION_MAJEURE < 2 #error "Version majeure du module trop ancienne !" #endif`
- **`#warning message` :** Demande au compilateur d'émettre un message d'avertissement (si supporté par le compilateur), mais la compilation continue. Utile pour signaler des configurations obsolètes ou

potentiellement problématiques. `#ifdef FONCTION_OBSOLETE #warning "L'utilisation de FONCTION_OBSOLETE est dépréciée." #endif`

- **`#pragma directive_specifique` :** `#pragma` est un mécanisme standard pour fournir des instructions **spécifiques au compilateur** (non portables). Les directives qui suivent `#pragma` dépendent entièrement du compilateur utilisé (ex: `#pragma omp ...` pour OpenMP, `#pragma pack(...)` pour contrôler l'alignement des structures, `#pragma warning(...)` pour gérer les avertissements spécifiques, etc.). Consultez la documentation de votre compilateur.
- **`#line numero "fichier"` :** Modifie le numéro de ligne (`__LINE__`) et le nom de fichier (`__FILE__`) que le compilateur utilise pour les messages d'erreur suivants. Principalement utilisé par des outils qui génèrent du code C.

Le préprocesseur est une étape puissante mais distincte de la compilation C. Il opère par **manipulation de texte** avant que le compilateur C n'analyse la syntaxe et la sémantique du langage. Comprendre son fonctionnement, en particulier les substitutions de `#define` (et leurs pièges), l'inclusion de fichiers via `#include`, et l'utilisation correcte de la compilation conditionnelle et des header guards, est essentiel pour écrire du code C modulaire, portable et maintenable.

---

17. Organisation du Code en Modules

Lorsque les projets C deviennent plus conséquents qu'un simple fichier `main.c`, il devient crucial de les organiser de manière logique et modulaire. Placer tout le code dans un unique fichier rend la navigation, la compréhension, la maintenance, la réutilisation et la compilation (temps de compilation longs) très difficiles.

La pratique standard et essentielle en C consiste à diviser le projet en plusieurs fichiers, en séparant l'**interface** (ce qu'un module offre) de son **implémentation** (comment il le fait) :

- Des **fichiers source (`.c`)** : Ils contiennent les **implémentations** (le code exécutable) des fonctions et la **définition** (l'allocation mémoire) des variables globales. Chaque fichier `.c` représente souvent un **module** logique du programme (par exemple, un module pour les calculs, un pour l'interface utilisateur, un pour la gestion des données...).
- Des **fichiers d'en-tête (`.h`)** (header files) : Ils contiennent les **déclarations** (les "signatures" ou "contrats") qui décrivent ce que chaque module met à disposition des autres modules. Ils définissent l'**interface publique** du module.

Cette séparation permet :

- **L'encapsulation :** Les détails d'implémentation d'un module sont cachés dans son fichier `.c`. Les autres modules n'ont besoin de connaître que son interface (le `.h`).
- **La compilation séparée :** Chaque fichier `.c` peut être compilé indépendamment en un fichier objet (`.o`). Si vous modifiez un seul fichier `.c`, vous n'avez besoin de recompiler que celui-ci, puis de refaire l'édition de liens, ce qui est beaucoup plus rapide pour les gros projets.
- **La réutilisabilité :** Un module bien défini (avec son `.h` et son `.c`) peut être plus facilement réutilisé dans d'autres projets.
- **La collaboration :** Plusieurs développeurs peuvent travailler sur différents modules (`.c`) simultanément, en se basant sur les interfaces définies dans les `.h`.

Séparer le code en plusieurs fichiers (`.c`)

L'idée est de regrouper les fonctions et les données (idéalement statiques au fichier) qui sont logiquement liées et contribuent à une fonctionnalité spécifique dans un même fichier `.c`.

Par exemple, un projet de calculatrice pourrait avoir :

- `main.c` : Gère la boucle principale, l'interaction utilisateur.
- `operations.c` : Contient les fonctions `addition`, `soustraction`, `multiplication`, etc.
- `affichage.c` : Contient les fonctions pour afficher les résultats ou l'interface.

Chaque fichier `.c` est une **unité de compilation** indépendante.

Utilisation des fichiers d'en-tête (`.h`) pour les déclarations

Comment un fichier (disons `main.c`) peut-il utiliser une fonction (comme `addition`) définie dans un autre fichier (`operations.c`) ? En incluant (`#include`) le **fichier d'en-tête (`.h`)** correspondant au module (`operations.h`).

Le fichier d'en-tête (`module.h`) sert d'**interface publique** pour le module (`module.c`). Il déclare ce que le module offre aux autres parties du programme, sans révéler les détails de son implémentation (qui restent dans le `.c`).

Un fichier d'en-tête typique (`module.h`) contient **exclusivement des déclarations** :

- **Prototypes de fonctions :** Les déclarations (signatures complètes avec types de retour et types des paramètres) des fonctions *publiques* (non `static`) définies dans `module.c`.
- **Définitions de types :** Les définitions complètes de `struct`, `union`, `enum` et les alias `typedef` qui doivent être connus et utilisables par les fichiers qui incluent ce header.
- **Constantes et Macros :** Les définitions `#define` pour les constantes symboliques ou les macros *publiques* que le module expose.
- **Déclarations `extern` de variables globales :** Si un module doit *vraiment* exposer une variable globale (ce qui est **fortement déconseillé** pour une bonne modularité), il la déclare avec le mot-clé `extern` dans le `.h` (par exemple, `extern int compteur_global;`). La **définition** réelle de cette variable (sans `extern`, pour allouer la mémoire) doit se trouver dans **un seul** fichier `.c` du projet.

Règles d'or pour les fichiers `.h` :

1. **JAMAIS de définitions de fonctions (non `inline`) :** N'incluez jamais le *corps* (l'implémentation `{...}`) d'une fonction normale (non `static inline`) dans un fichier `.h`. Si ce `.h` est inclus par plusieurs `.c`, cela mènerait à des erreurs de "définition multiple" lors de l'édition de liens.
2. **JAMAIS de définitions de variables globales :** Ne définissez jamais une variable globale (non `static`) dans un `.h` (ex: `int ma_globale = 10;`). Déclarez-la avec `extern` dans le `.h` et définissez-la (sans `extern`) dans *un seul* fichier `.c`.
3. **TOUJOURS utiliser les "Header Guards" :** Protégez systématiquement le contenu d'un fichier `.h` contre les inclusions multiples en utilisant le préprocesseur (`#ifndef`/`#define`/`#endif`), comme vu dans la section sur le préprocesseur. C'est indispensable.

Conventionnellement, pour un module implémenté dans `foo.c`, on crée un fichier d'interface `foo.h`. Le fichier `foo.c` doit lui-même inclure son propre en-tête (`#include "foo.h"`) en premier. Cela permet au compilateur de vérifier que les définitions dans le `.c` correspondent bien aux déclarations (prototypes) dans le `.h`, assurant la cohérence.

```c
/* Contenu typique d'un fichier "calculs.h" */

#ifndef CALCULS_H_INCLUDED // Garde d'inclusion (Header Guard)
#define CALCULS_H_INCLUDED

// Inclusion d'autres headers si nécessaire pour les types utilisés ici
#include <stdbool.h> // Par exemple, si on utilisait des booléens

// Définition d'un type structure utilisé par le module
typedef struct {
    double x;
    double y;
} Point_t;

// Définition d'une constante symbolique publique
#define VERSION_CALCULS "1.1"

// Prototypes des fonctions publiques définies dans "calculs.c"
// C'est l'interface publique du module.
int ajouter(int a, int b);
int soustraire(int a, int b);
double calculer_distance(Point_t p1, Point_t p2);
bool est_positif(int nombre);

// Déclaration d'une variable globale externe (à éviter si possible !)
// Indique que 'compteur_operations' est défini ailleurs (dans calculs.c)
extern int compteur_operations;

#endif // CALCULS_H_INCLUDED
```

Contenu du Fichier Source (`.c`)

Le fichier `.c` correspondant (par exemple, `calculs.c`) contient l'**implémentation** des fonctions déclarées dans le `.h`, ainsi que la **définition** des variables globales déclarées `extern` dans le `.h`.

Il doit commencer par inclure son propre fichier d'en-tête pour permettre la vérification par le compilateur.

Il peut également contenir :

- Des fonctions **statiques** (`static void fonction_interne(...)`). Ces fonctions ont un **linkage interne** (voir ci-dessous) et ne sont utilisables qu'à l'intérieur de ce fichier `.c`. Elles servent à l'implémentation interne du module mais ne font pas partie de son interface publique.
- Des variables globales **statiques** (`static int compteur_prive = 0;`). Elles ont aussi un linkage interne, existent pendant toute la durée du programme, mais ne sont accessibles que par les fonctions de ce fichier `.c`. C'est la manière de créer des "variables de module" privées.
- D'autres directives `#include` pour les bibliothèques standard ou autres modules dont *cette implémentation* a besoin (par exemple, `#include <math.h>` si on utilise `sqrt`).

```c
/* Contenu typique d'un fichier "calculs.c" */

// 1. Inclure son propre header en premier !
#include "calculs.h"

// 2. Inclure les autres headers nécessaires pour l'implémentation
#include <math.h>   // Pour sqrt, pow
#include <stdio.h>  // Pour d'éventuels printf de débogage internes

// 3. Définition des variables globales déclarées 'extern' dans le .h
//    (Allocation réelle de la mémoire)
int compteur_operations = 0;

// 4. Définition de variables globales ou fonctions statiques (privées au
module)
static const double PRECISION = 0.0001; // Constante interne au module

// Fonction statique (interne au module, non visible de l'extérieur)
// Elle n'a pas besoin d'être déclarée dans calculs.h
static void incrementer_compteur_interne() {
    compteur_operations++;
    // printf("[DEBUG calc] Opération effectuée, compteur = %d\n",
compteur_operations);
}

// 5. Définition (implémentation) des fonctions publiques déclarées dans
calculs.h
int ajouter(int a, int b) {
    incrementer_compteur_interne(); // Appelle une fonction interne
    return a + b;
}

int soustraire(int a, int b) {
    incrementer_compteur_interne();
    return a - b;
}

double calculer_distance(Point_t p1, Point_t p2) {
    incrementer_compteur_interne();
    double dx = p1.x - p2.x;
    double dy = p1.y - p2.y;
    // Utilise les fonctions pow et sqrt de math.h
    return sqrt(pow(dx, 2) + pow(dy, 2));
}

bool est_positif(int nombre) {
     incrementer_compteur_interne();
     return nombre > 0;
}

// On pourrait avoir d'autres fonctions internes ou publiques ici...
```

Utilisation du Module dans un autre fichier (`main.c`)

Un autre fichier source, comme `main.c`, peut maintenant utiliser les fonctionnalités publiques du module `calculs` simplement en **incluant le fichier d'en-tête `calculs.h`**.

L'inclusion du `.h` donne à `main.c` toutes les informations nécessaires (prototypes, définitions de types comme `Point_t`, déclarations `extern`...) pour que le compilateur puisse vérifier la syntaxe et le typage des appels aux fonctions du module `calculs`. `main.c` n'a **pas besoin** de connaître les détails de l'implémentation contenus dans `calculs.c`.

```c
/* Contenu typique d'un fichier "main.c" */

#include <stdio.h>
#include <stdbool.h> // Car on utilise bool

// Inclut l'interface de notre module de calculs
#include "calculs.h"

// main.c n'inclut PAS calculs.c ! Il inclut seulement calculs.h.
// Il n'a pas besoin de connaître l'implémentation des fonctions,
// seulement leurs déclarations (prototypes) fournies par calculs.h.

int main() {
    int res_add, res_sub;
    Point_t ptA = {1.0, 2.0}; // Utilisation du type Point_t défini dans
calculs.h
    Point_t ptB = {4.0, 6.0};
    double dist;
    int nombre_test = -5;

    printf("Utilisation du module de calculs (Version %s)\n",
VERSION_CALCULS); // Utilise la macro de calculs.h

    // Appels aux fonctions publiques déclarées dans calculs.h
    res_add = ajouter(10, 5);
    printf("10 + 5 = %d\n", res_add);

    res_sub = soustraire(10, 5);
    printf("10 - 5 = %d\n", res_sub);

    dist = calculer_distance(ptA, ptB);
    printf("Distance entre (%.1f, %.1f) et (%.1f, %.1f) = %.2f\n",
           ptA.x, ptA.y, ptB.x, ptB.y, dist);

    if (est_positif(nombre_test)) {
         printf("%d est positif.\n", nombre_test);
    } else {
         printf("%d n'est pas positif.\n", nombre_test);
    }
```

```
    // Accès (lecture) à la variable globale (déclarée extern dans
calculs.h)
    printf("Nombre total d'opérations effectuées : %d\n",
compteur_operations);

    // Tentative d'appel à une fonction interne (échouerait à la
compilation ou à l'édition de liens)
    // incrementer_compteur_interne(); // ERREUR: fonction non déclarée ici
et/ou linkage interne

    return 0;
}
```

Notion de linkage (interne/externe)

Le **linkage** (parfois traduit par "édition de liens" en français, bien que le terme anglais soit très courant en contexte C) détermine si un identifiant (nom de fonction ou de variable globale) défini dans un fichier `.c` est **visible et utilisable** par d'autres fichiers `.c` du même projet lors de l'étape finale d'édition de liens.

Il existe principalement deux types de linkage pour les identifiants au niveau fichier (en dehors des fonctions) :

- **Linkage Externe (External Linkage) :**
    - C'est le comportement **par défaut** pour les fonctions et les variables globales (définies en dehors de toute fonction *sans* le mot-clé `static`).
    - Un identifiant avec linkage externe est **visible dans tous les fichiers `.c`** du projet. Le lieur (linker) peut connecter une référence (un appel de fonction, une utilisation de variable globale déclarée `extern`) dans un fichier `.c` à sa définition unique dans un autre fichier `.c`.
    - **Attention aux conflits de noms :** Vous ne pouvez avoir qu'**une seule définition** d'un identifiant à linkage externe dans l'ensemble de votre projet (tous les `.c` et bibliothèques liés ensemble). Avoir deux fonctions `int maFonction()` non-`static` dans deux fichiers `.c` différents provoquera une erreur de "symbole défini plusieurs fois" (multiple definition) lors de l'édition de liens. C'est pourquoi les fonctions "privées" à un module doivent être `static`.
- **Linkage Interne (Internal Linkage) :**
    - On obtient un linkage interne en utilisant le mot-clé **`static`** devant la définition d'une **fonction** ou d'une **variable globale** (définie en dehors de toute fonction).
    - Un identifiant avec linkage interne est **visible uniquement à l'intérieur du fichier `.c` où il est défini**. Il n'est pas "exporté" et ne peut pas être directement référencé depuis d'autres fichiers `.c`, même si son prototype ou sa déclaration `extern` apparaîtrait (à tort) dans un fichier `.h`. Le lieur l'ignore lors de la résolution des symboles entre fichiers.
    - C'est **très utile** pour créer des variables ou des fonctions **"privées"** à un module (`.c`). Elles servent à l'implémentation interne du module mais ne font pas partie de son interface publique et ne peuvent pas entrer en conflit avec des noms utilisés dans d'autres modules. Cela favorise l'**encapsulation** et la **modularité**.

(Note : Le mot-clé `static` a donc deux significations différentes selon le contexte : pour les variables locales, il affecte la durée de vie ; pour les fonctions et variables globales, il affecte le linkage).

```c
/* === Fichier : module_A.h === */
#ifndef MODULE_A_H
#define MODULE_A_H

// Fonction publique (linkage externe par défaut)
void fonction_publique_A();

// Variable globale publique (linkage externe par défaut)
// Déclarée extern ici, définie dans module_A.c
extern int compteur_global_A;

#endif // MODULE_A_H

/* === Fichier : module_A.c === */
#include <stdio.h>
#include "module_A.h" // Inclut son propre header

// Définition de la variable globale publique
int compteur_global_A = 0;

// Variable globale privée (linkage interne)
static int compteur_interne_A = 0;

// Fonction privée (linkage interne)
static void helper_A() {
    printf("  (Dans helper_A - interne)\n");
    compteur_interne_A++;
}

// Définition de la fonction publique
void fonction_publique_A() {
    printf("Appel de fonction_publique_A()\n");
    helper_A(); // Appel d'une fonction interne
    compteur_global_A++;
    printf("  Compteur interne A = %d\n", compteur_interne_A);
    printf("  Compteur global A = %d\n", compteur_global_A);
}


/* === Fichier : main.c === */
#include <stdio.h>
#include "module_A.h" // Inclut l'interface du module A

// extern int compteur_interne_A; // ERREUR: Ne peut pas accéder car static
dans module_A.c
// extern void helper_A();       // ERREUR: Ne peut pas accéder car static
dans module_A.c

int main() {
```

```
    printf("--- Appel 1 ---\n");
    fonction_publique_A(); // OK: Fonction publique

    printf("\n--- Appel 2 ---\n");
    fonction_publique_A(); // OK

    // Accès à la variable globale publique
    compteur_global_A = 100;
    printf("\nCompteur global A modifié depuis main: %d\n",
compteur_global_A);

    printf("\n--- Appel 3 ---\n");
    fonction_publique_A(); // Montre que compteur_global_A a été modifié

    // helper_A(); // ERREUR de compilation ou de linkage
    // compteur_interne_A = 5; // ERREUR de compilation ou de linkage

    return 0;
}

/* Compilation (exemple avec GCC):
   gcc -c module_A.c -o module_A.o
   gcc -c main.c -o main.o
   gcc main.o module_A.o -o programme_final
   ./programme_final
*/
```

Organiser son code en modules distincts avec des fichiers `.c` pour l'implémentation (en utilisant `static` pour cacher les détails internes) et des fichiers `.h` clairs et concis pour les interfaces publiques (avec header guards) est une pratique fondamentale en C. Cela permet de développer des applications de taille conséquente de manière structurée, maintenable, réutilisable et facilite grandement la compilation et la collaboration.

---

18. Compilation et Édition de Liens

Transformer vos fichiers source `.c` et `.h` en un programme exécutable que l'ordinateur peut comprendre implique un processus en plusieurs étapes, généralement orchestré par votre compilateur (comme GCC ou Clang). Comprendre ces étapes est essentiel pour gérer des projets multi-fichiers et utiliser des bibliothèques externes.

Processus : Préprocessing -> Compilation -> Assemblage -> Édition de liens

Même si vous lancez souvent une seule commande comme `gcc main.c utils.c -o mon_prog`, plusieurs étapes distinctes se déroulent en coulisses :

1. **Préprocessing :** Le préprocesseur C (CPP) traite le code source.
    - Il retire tous les commentaires (`//...` et `/*...*/`).
    - Il interprète les directives de préprocesseur (`#...`).

        - `#include` : Insère le contenu des fichiers d'en-tête.
        - `#define` : Effectue les substitutions de macros.
        - `#if`, `#ifdef`, etc. : Exclut ou inclut conditionnellement des portions de code.
    - Le résultat est une **unité de traduction** (translation unit), un fichier source C "pur" et étendu, prêt pour le compilateur. Cette étape est appliquée à *chaque* fichier `.c` individuellement.

2. **Compilation :** Le compilateur C prend l'unité de traduction générée par le préprocesseur.
    - Il analyse le code C pour vérifier la syntaxe et la sémantique (types, etc.).
    - Il traduit le code C en **code assembleur**, un langage de bas niveau spécifique à l'architecture du processeur cible (par exemple, x86-64, ARM).

3. **Assemblage :** L'assembleur (souvent appelé `as`) prend le code assembleur généré.
    - Il traduit les instructions assembleur en **code machine binaire** (les instructions directement compréhensibles par le processeur).
    - Il produit un **fichier objet** (object file), portant généralement l'extension `.o` (sous Linux/macOS) ou `.obj` (sous Windows). Ce fichier contient le code machine pour *un seul* fichier `.c`, ainsi qu'une "table des symboles" listant les fonctions et variables globales définies ou utilisées, et des informations pour l'édition de liens (indiquant où se trouvent les appels à des fonctions externes, par exemple).

4. **Édition de liens (Linking) :** Le **lieur** (linker, souvent appelé `ld`) est l'étape finale.
    - Il prend **tous** les fichiers objets (`.o` ou `.obj`) générés à partir de vos fichiers `.c`.
    - Il prend également les **bibliothèques** (libraries) nécessaires (comme la bibliothèque standard C, ou d'autres bibliothèques externes que vous utilisez, par exemple `math` ou `SDL2`). Les bibliothèques contiennent du code machine précompilé pour des fonctions courantes.
    - Son rôle est de **résoudre les références croisées** entre les fichiers objets et les bibliothèques :
        - Lorsqu'un fichier objet (ex: `main.o`) appelle une fonction définie dans un autre (ex: `utils.o`), le lieur trouve l'adresse de la fonction dans `utils.o` et "relie" l'appel dans `main.o` à cette adresse.
        - Il fait de même pour les fonctions des bibliothèques (comme `printf` ou `sqrt`).
        - Il regroupe tout le code machine nécessaire et les données globales dans un unique **fichier exécutable** final (par exemple, `mon_prog` ou `mon_prog.exe`).

Compilation séparée (`gcc -c`)

Pour les projets multi-fichiers, il est inefficace de recompiler *tous* les fichiers `.c` à chaque petite modification. On utilise la **compilation séparée** : on compile chaque `.c` en `.o` indépendamment, puis on lie tous les `.o` ensemble.

L'option `-c` du compilateur (GCC/Clang) lui demande de s'arrêter après l'étape d'assemblage, produisant uniquement le fichier objet (`.o`) sans tenter l'édition de liens.

```
# Compiler main.c en main.o (sans lier)
# -Wall et -Wextra activent plus d'avertissements (recommandé)
# -std=c11 (ou c17, c18) spécifie la version du standard C
# -g ajoute les informations de débogage
```

```
gcc -Wall -Wextra -std=c11 -g -c main.c -o main.o

# Compiler calculs.c en calculs.o
gcc -Wall -Wextra -std=c11 -g -c calculs.c -o calculs.o

# Compiler affichage.c en affichage.o
gcc -Wall -Wextra -std=c11 -g -c affichage.c -o affichage.o

# À ce stade, nous avons main.o, calculs.o, affichage.o
# mais pas encore de programme exécutable.
```

Édition de liens (`gcc ... *.o -o prog`)

Une fois que tous les fichiers objets nécessaires sont créés (ou mis à jour), l'étape finale est l'**édition de liens**. On fournit au compilateur (qui appellera le lieur en interne) tous les fichiers objets et on lui indique les bibliothèques externes à inclure.

La commande de liaison ressemble à ceci : `gcc fichier1.o fichier2.o ... fichierN.o -o nom_executable [options_de_liaison]`

```
# Lier les fichiers objets créés précédemment pour créer l'exécutable
'mon_programme'
gcc main.o calculs.o affichage.o -o mon_programme

# Si le code utilise des fonctions de la bibliothèque mathématique (ex:
sqrt, pow de math.h)
# il faut explicitement demander au lieur de l'inclure avec l'option -lm
(link math)
# (Sur certains systèmes/compilateurs récents, ce n'est plus toujours
nécessaire, mais c'est une bonne habitude)
gcc main.o calculs.o affichage.o -o mon_programme -lm

# Si le code utilise une bibliothèque externe comme SDL2 (installée sur le
système)
# il faut spécifier les bibliothèques SDL à lier avec -l (ex: -lSDL2 -
lSDL2_image -lSDL2_ttf)
# gcc main.o graphics.o input.o -o mon_jeu_sdl -lSDL2 -lSDL2_image -
lSDL2_ttf
```

Bibliothèques Statiques et Dynamiques (Concept)

Les bibliothèques (libraries) contiennent du code précompilé (fonctions, données) que plusieurs programmes peuvent utiliser. Il existe deux types principaux :

1. **Bibliothèques Statiques (`.a` sous Linux/macOS, `.lib` sous Windows) :**
    - Le code machine des fonctions de la bibliothèque utilisées par votre programme est **copié** directement dans votre fichier exécutable final par le lieur.

- **Avantages :** L'exécutable est autonome (n'a pas besoin de la bibliothèque séparément pour s'exécuter).
- **Inconvénients :** Augmente la taille de chaque exécutable qui l'utilise. Si la bibliothèque est mise à jour, tous les programmes l'utilisant doivent être recompilés et reliés pour en bénéficier.

2. **Bibliothèques Dynamiques (ou Partagées) (`.so` sous Linux, `.dylib` sous macOS, `.dll` sous Windows) :**

   - Le code de la bibliothèque n'est **pas** copié dans votre exécutable. L'exécutable contient seulement des **références** (des "promesses" d'appeler telle fonction de telle bibliothèque).
   - Au moment de l'exécution (ou parfois au chargement), le système d'exploitation charge la bibliothèque dynamique en mémoire (si elle n'y est pas déjà) et résout les références dans votre programme pour qu'il puisse appeler les fonctions de la bibliothèque.
   - **Avantages :** Économise de l'espace disque (la bibliothèque n'existe qu'une fois). Permet des mises à jour de la bibliothèque sans recompiler les programmes qui l'utilisent (tant que l'interface reste compatible). Permet le partage de la mémoire de la bibliothèque entre plusieurs processus l'utilisant.
   - **Inconvénients :** Le programme dépend de la présence de la bonne version de la bibliothèque dynamique sur le système où il s'exécute (risque de "DLL Hell" sous Windows ou problèmes de dépendances similaires). Léger surcoût au chargement/premier appel pour résoudre les liens.

Par défaut, les compilateurs comme GCC essaient généralement de lier les bibliothèques de manière **dynamique** si elles sont disponibles. La bibliothèque standard C est presque toujours liée dynamiquement. Pour les bibliothèques externes comme SDL2, on les installe généralement en tant que bibliothèques dynamiques/partagées.

Utilisation des bibliothèques (`-l`, `-L`, `-I`)

Lorsque vous utilisez des fonctions d'une bibliothèque qui n'est pas la bibliothèque C standard de base (comme `math.h` ou une bibliothèque externe comme SDL), vous devez généralement indiquer au compilateur/lieur comment la trouver :

- **`-l<nom>` (Option du Lieur) :** Demande au lieur de chercher et d'inclure la bibliothèque nommée `lib<nom>.a` (statique) ou `lib<nom>.so`/`.dylib`/`.dll` (dynamique).
  - Ex: `-lm` pour lier la bibliothèque mathématique (`libm.so` ou `libm.a`).
  - Ex: `-lSDL2` pour lier la bibliothèque SDL2 (`libSDL2.so`, etc.).
  - L'ordre des options `-l` peut parfois importer : si la bibliothèque A dépend de la bibliothèque B, mettez `-lA` avant `-lB` sur la ligne de commande.
- **`-L<repertoire>` (Option du Lieur) :** Ajoute `<repertoire>` à la liste des chemins où le lieur doit chercher les fichiers de bibliothèques (`.a`, `.so`, etc.) spécifiés par `-l`. Utile si vos bibliothèques ne sont pas dans les chemins standard.
  - Ex: `gcc main.o -L/chemin/vers/mes/libs -lMaBibliotheque -o prog`
- **`-I<repertoire>` (Option du Compilateur/Préprocesseur) :** Ajoute `<repertoire>` à la liste des chemins où le préprocesseur doit chercher les fichiers d'en-tête (`.h`) inclus avec `#include "..."` ou `#include <...>`. Nécessaire si vos fichiers `.h` ne sont pas dans le répertoire courant ou dans les chemins d'inclusion standard.
  - Ex: `gcc -c main.c -I/chemin/vers/headers/externes -o main.o`

Options courantes du compilateur (GCC/Clang)

Il existe de nombreuses options pour contrôler le compilateur. Voici quelques-unes des plus utiles :

- **`-o <fichier>` :** Spécifie le nom du fichier de sortie (exécutable ou fichier objet si utilisé avec `-c`).
- **`-c` :** Compile et assemble seulement, ne fait pas l'édition de liens. Produit un fichier `.o`.
- **`-g` :** Inclut les informations de débogage dans le fichier objet/exécutable. Essentiel pour utiliser un débogueur comme GDB ou LLDB.
- **`-Wall` :** Active la plupart des avertissements (Warnings ALL). **Fortement recommandé** pour détecter des problèmes potentiels dans votre code.
- **`-Wextra` :** Active des avertissements supplémentaires, également très utiles.
- **`-Werror` :** Traite tous les avertissements comme des erreurs, forçant leur correction. Utile pour maintenir une haute qualité de code.
- **`-std=<standard>` :** Spécifie la version du standard C à utiliser (ex: `-std=c99`, `-std=c11`, `-std=c17`, `-std=c18`, `-std=gnu11` pour activer les extensions GNU en plus de C11).
- **`-O<niveau>` :** Contrôle le niveau d'optimisation du code généré.
  - `-O0` : Aucune optimisation (utile pour le débogage). C'est souvent le défaut sans `-g`.
  - `-O1`, `-O2`, `-O3` : Niveaux d'optimisation croissants (le code devient plus rapide mais peut être plus difficile à déboguer et la compilation est plus longue). `-O2` est un bon compromis courant pour les versions finales.
  - `-Os` : Optimise pour la taille du code.
- **`-D<MACRO>` ou `-D<MACRO>=<valeur>` :** Définit une macro de préprocesseur depuis la ligne de commande, comme si `#define <MACRO>` (ou `#define <MACRO> <valeur>`) était dans le code. Utile pour la compilation conditionnelle (ex: `-DDEBUG`, `-DVERSION=1.2`).
- **`-I<repertoire>` :** Ajoute un répertoire pour la recherche des fichiers d'en-tête (`#include`).
- **`-L<repertoire>` :** Ajoute un répertoire pour la recherche des bibliothèques lors de l'édition de liens.
- **`-l<nom>` :** Demande de lier avec la bibliothèque `lib<nom>`.

Comprendre le processus de compilation et d'édition de liens est fondamental pour travailler sur des projets C non triviaux, pour utiliser des bibliothèques externes, et pour optimiser ou déboguer efficacement votre code. L'utilisation d'outils comme `Make` (voir section suivante) permet d'automatiser et de simplifier la gestion de ces étapes pour des projets complexes.

---

19. Systèmes de Build : Make

Lorsque vous travaillez sur un projet C comportant plusieurs fichiers source, le processus de compilation séparée suivi de l'édition de liens (comme vu dans la section précédente) devient rapidement fastidieux à taper manuellement à chaque modification. De plus, il est inefficace de tout recompiler si un seul fichier a changé.

C'est là qu'interviennent les **systèmes de build** (build systems). Ce sont des outils conçus pour **automatiser** le processus de compilation et d'édition de liens, en ne recompilant que ce qui est strictement nécessaire.

Pourquoi utiliser un système de build ?

- **Automatisation :** Une seule commande (ex: `make`) suffit pour compiler et lier tout le projet.
- **Efficacité :** Ne recompile que les fichiers `.c` qui ont été modifiés (ou dont les dépendances, comme les fichiers `.h`, ont changé) depuis la dernière compilation, ce qui accélère considérablement le cycle

de développement pour les gros projets.
- **Gestion des dépendances :** Permet de définir clairement les relations entre les fichiers (ex: `main.o` dépend de `main.c` et `utils.h`).
- **Portabilité (relative) :** Les Makefiles sont largement utilisés sur les systèmes Unix-like (Linux, macOS) et sont également supportés sur Windows (via MinGW/MSYS2, Cygwin, ou des versions spécifiques comme NMake).
- **Standardisation :** Fournit une manière standard de construire de nombreux projets open-source.

Introduction à `Make`

`make` est l'un des systèmes de build les plus anciens et les plus répandus, en particulier dans l'écosystème C/C++. Il fonctionne en lisant les instructions contenues dans un fichier spécial nommé **`Makefile`** (ou `makefile`) situé à la racine de votre projet.

Un `Makefile` contient essentiellement des **règles** qui décrivent comment créer des fichiers (les **cibles**, *targets*) à partir d'autres fichiers (les **dépendances** ou prérequis, *dependencies*), en utilisant des **commandes** spécifiques (généralement des appels au compilateur/lieur).

Syntaxe de base d'un `Makefile`

Un `Makefile` est composé de règles ayant la forme générale suivante :

```
cible: dépendances commande1 commande2 ...
```

- **`cible` (target) :** Le nom du fichier que la règle vise à créer ou à mettre à jour (par exemple, un fichier objet `.o` ou le fichier exécutable final). Il peut aussi y avoir des cibles "factices" (phony targets) qui ne représentent pas de fichier mais une action (comme `clean` ou `all`).
- **`dépendances` (dependencies/prerequisites) :** Une liste de fichiers (séparés par des espaces) dont la `cible` dépend. Si l'une de ces dépendances est plus récente que la `cible` (ou si la `cible` n'existe pas), `make` exécutera les commandes pour la recréer.
- **`commande` (command/recipe) :** Une ou plusieurs lignes de commandes shell (comme `gcc ...`, `rm ...`) qui sont exécutées pour créer la `cible` à partir des dépendances. **Très important :** Chaque ligne de commande doit **obligatoirement commencer par un caractère de tabulation** (pas des espaces !). C'est une source d'erreur fréquente.

Exemple simple de règle :

```
hello.o: hello.c stdio.h gcc -c hello.c -o hello.o
```

Cette règle signifie :

- Pour créer la cible `hello.o`...
- ...elle dépend de `hello.c` et `stdio.h`.
- Si `hello.o` n'existe pas, ou si `hello.c` ou `stdio.h` a été modifié plus récemment que `hello.o`...
- ...exécute la commande `gcc -c hello.c -o hello.o`.

Fonctionnement de `make` : Lorsque vous tapez `make` dans le terminal (dans le répertoire contenant le `Makefile`), `make` :

1. Lit le `Makefile`.

2. Regarde la **première règle** définie dans le fichier (par défaut) comme étant la cible principale à construire (sauf si vous spécifiez une autre cible : `make ma_cible`).
3. Examine les dépendances de cette cible.
4. Pour chaque dépendance, il vérifie s'il existe une règle pour la créer et si elle doit être mise à jour (en comparant les dates de modification ou si elle n'existe pas). Il applique ce processus récursivement pour toutes les dépendances.
5. Une fois que toutes les dépendances d'une cible sont à jour, `make` vérifie si la cible elle-même doit être reconstruite (si elle n'existe pas ou si une de ses dépendances est plus récente).
6. Si la cible doit être reconstruite, `make` exécute les commandes associées à cette règle.

```
# Makefile simple pour compiler le projet avec main.c et calculs.c
# (Suppose que main.c inclut calculs.h et que calculs.c inclut calculs.h)

# Cible finale : l'exécutable 'mon_programme'
# Dépend des fichiers objets .o
mon_programme: main.o calculs.o
    # Commande de liaison (commence par une TABULATION !)
    # Lie les .o pour créer l'exécutable
    gcc main.o calculs.o -o mon_programme -lm # Ajout de -lm si calculs.c
utilise math.h

# Règle pour créer main.o
# Dépend de main.c et du header calculs.h
main.o: main.c calculs.h
    # Commande de compilation (commence par une TABULATION !)
    # Compile main.c en main.o (option -c)
    gcc -c main.c -o main.o

# Règle pour créer calculs.o
# Dépend de calculs.c et du header calculs.h
calculs.o: calculs.c calculs.h
    # Commande de compilation (commence par une TABULATION !)
    gcc -c calculs.c -o calculs.o

# Cible factice "clean" pour supprimer les fichiers générés
# .PHONY indique que 'clean' n'est pas un fichier réel
.PHONY: clean
clean:
    # Commande de nettoyage (commence par une TABULATION !)
    # rm -f supprime les fichiers sans demander de confirmation (-f)
    rm -f mon_programme main.o calculs.o

# Pour utiliser :
# make           -> Construit mon_programme (la première cible)
# make clean     -> Exécute les commandes de la cible 'clean'
# make mon_programme -> Construit explicitement mon_programme
```

Variables dans Make

Pour éviter la répétition et faciliter la configuration, les `Makefile`s utilisent intensivement des **variables** (parfois appelées macros dans la terminologie `make`).

- **Définition :** On définit une variable avec `=` ou `:=`.
  - `NOM_VARIABLE = valeur` (assignation récursive simple, la valeur est évaluée quand la variable est utilisée)
  - `NOM_VARIABLE := valeur` (assignation simplement étendue, la valeur est évaluée immédiatement à la définition)
  - `NOM_VARIABLE ?= valeur` (assignation conditionnelle, n'assigne que si la variable n'est pas déjà définie)
- **Utilisation :** On référence la valeur d'une variable avec `$(NOM_VARIABLE)` ou `${NOM_VARIABLE}`.

Variables courantes prédéfinies ou conventionnelles :

- `CC` : Le nom du compilateur C (par défaut souvent `cc`, qui est un lien vers `gcc` sur beaucoup de systèmes). Il est bon de le définir explicitement : `CC = gcc`.
- `CFLAGS` : Les options (flags) à passer au compilateur C lors de la compilation des fichiers `.c` en `.o` (ex: `-Wall -Wextra -g -O2 -std=c11`).
- `LDFLAGS` : Les options à passer au lieur (linker) lors de la création de l'exécutable (ex: `-L/chemin/lib`).
- `LDLIBS` ou `LIBS` : La liste des bibliothèques à lier (ex: `-lm -lSDL2`).
- `RM` : La commande pour supprimer des fichiers (par défaut `rm -f`).

Variables automatiques (spéciales, utilisées dans les commandes des règles) :

- `$@` : Le nom de la **cible** de la règle.
- `$<` : Le nom de la **première dépendance**.
- `$^` : La liste de **toutes les dépendances**, séparées par des espaces.
- `*.o` : La liste des fichiers objets.

Utiliser des variables rend le `Makefile` beaucoup plus flexible et facile à maintenir.

```
# Makefile amélioré avec des variables

# --- Variables de Configuration ---
CC = gcc # Compilateur C
# Options de compilation : avertissements, standard C, débogage
CFLAGS = -Wall -Wextra -std=c11 -g
# Options pour l'éditeur de liens (ex: chemin vers bibliothèques)
LDFLAGS =
# Bibliothèques à lier (ex: -lm pour la bibliothèque mathématique)
LIBS = -lm

# Nom de l'exécutable final
TARGET = mon_programme

# Liste des fichiers objets (.o) nécessaires pour l'exécutable
# Make peut déterminer les sources (.c) à partir des objets
OBJS = main.o calculs.o affichage.o
```

```
# --- Règles ---

# Première règle (cible par défaut car c'est la première)
# Dépend de tous les fichiers objets listés dans $(OBJS)
$(TARGET): $(OBJS)
    # Commande de liaison utilisant les variables
    # $@ est la cible (mon_programme)
    # $^ est la liste des dépendances (main.o calculs.o affichage.o)
    $(CC) $(LDFLAGS) $^ -o $@ $(LIBS)

# Règle générique pour créer n'importe quel fichier .o à partir d'un .c
# %.o : Cible correspondant à n'importe quel fichier finissant par .o
# %.c : Dépendance correspondante finissant par .c
# $< est la première dépendance (le fichier .c)
%.o: %.c
    # Commande de compilation utilisant les variables
    $(CC) $(CFLAGS) -c $< -o $@

# Cible factice "all" (souvent utilisée pour être la cible par défaut
explicite)
.PHONY: all
all: $(TARGET)

# Cible factice "clean"
.PHONY: clean
clean:
    # Commande de nettoyage utilisant la variable RM (prédéfinie)
    $(RM) $(TARGET) $(OBJS)

# Pour utiliser :
# make          -> Construit mon_programme (via la cible 'all' ou la
première règle)
# make clean    -> Nettoie les fichiers générés
```

Gestion des dépendances et Recompilation

La grande force de `make` réside dans sa capacité à ne reconstruire que ce qui est nécessaire.

- **Dépendances explicites :** Dans une règle `cible: dependance1 dependance2`, `make` sait que si `dependance1` ou `dependance2` est plus récent que `cible`, il faut exécuter les commandes pour reconstruire `cible`.
- **Dépendances implicites :** `make` a des règles intégrées. Par exemple, il sait qu'un fichier `.o` dépend généralement du fichier `.c` du même nom. La règle générique `%.o: %.c` vue dans l'exemple précédent exploite cela.
- **Dépendances des fichiers d'en-tête :** C'est un point crucial. Si `main.c` et `calculs.c` incluent tous les deux `calculs.h`, et que vous modifiez *uniquement* `calculs.h`, alors `main.o` et `calculs.o` doivent tous les deux être recompilés, car le code qu'ils génèrent pourrait dépendre des changements dans le header. C'est pourquoi les fichiers `.h` doivent apparaître dans les dépendances des fichiers `.o` qui les incluent (comme montré dans les exemples de Makefiles).

- **Comment `make` vérifie :** Il compare les dates de dernière modification des fichiers cibles et de leurs dépendances. Si une dépendance a une date plus récente que la cible, ou si la cible n'existe pas, la reconstruction est déclenchée.

Pour des projets plus complexes, la gestion manuelle des dépendances des headers dans le `Makefile` peut devenir fastidieuse. Des techniques plus avancées existent (souvent en utilisant des fonctionnalités du compilateur comme `gcc -MMD -MP`) pour générer automatiquement ces dépendances, mais cela dépasse le cadre de cette introduction.

Conclusion sur Make

`make` est un outil puissant et standard pour automatiser la compilation de projets C/C++. Bien que sa syntaxe (notamment l'obligation des tabulations) puisse sembler archaïque, sa capacité à gérer les dépendances et à effectuer des recompilations minimales en fait un outil indispensable pour tout projet C de taille non triviale. D'autres systèmes de build plus modernes existent (CMake, Meson, Ninja, etc.), mais `make` reste très répandu et sa compréhension est une compétence précieuse.

---

20. Gestion des Fichiers E S Fichiers

Jusqu'à présent, nos programmes interagissaient principalement avec l'utilisateur via la console (entrée standard `stdin`, sortie standard `stdout`, erreur standard `stderr`). Ces flux sont temporaires ; les données disparaissent une fois le programme terminé.

Pour rendre les données **persistantes** (les conserver entre les exécutions du programme) ou pour traiter de grandes quantités d'informations qui ne tiendraient pas confortablement en mémoire ou ne seraient pas pratiques à saisir à chaque fois, nous devons interagir avec des **fichiers** stockés sur un disque dur, SSD, ou autre support de stockage.

Le C fournit un ensemble de fonctions puissantes, principalement définies dans l'en-tête standard `<stdio.h>`, pour effectuer des opérations d'**Entrée/Sortie (E/S ou I/O - Input/Output)** sur les fichiers. Ces opérations incluent l'ouverture, la fermeture, la lecture et l'écriture de données dans les fichiers.

Concept de Flux (`FILE *`)

Au cœur de la gestion des fichiers en C se trouve le concept de **flux** (stream). Un flux est une abstraction qui représente une séquence de données (généralement des octets) pouvant être lue ou écrite. Les flux peuvent être associés à différentes sources ou destinations : un fichier sur disque, la console, un périphérique réseau, etc. `stdin`, `stdout`, et `stderr` sont des exemples de flux pré-ouverts associés à la console.

Pour interagir avec un fichier sur disque, nous devons l'associer à un flux dans notre programme. En C, cette association est gérée par une structure de données interne appelée `FILE` (définie dans `<stdio.h>`). Cette structure contient toutes les informations nécessaires pour gérer le flux lié au fichier :

- La position actuelle dans le fichier (où se fera la prochaine lecture/écriture).
- Des indicateurs d'état (fin de fichier atteinte, erreur survenue).
- Un tampon (buffer) pour optimiser les lectures/écritures physiques sur le disque.
- D'autres détails de bas niveau.

Important : Nous n'avons **jamais** besoin de connaître ou de manipuler directement les membres de la structure `FILE`. À la place, nous travaillons toujours avec un **pointeur vers cette structure**, c'est-à-dire une variable de type `FILE *`. Ce pointeur agit comme un "descripteur" ou une "poignée" (handle) pour le fichier ouvert au sein de notre programme. Toutes les fonctions d'E/S sur fichier (comme `fopen`, `fclose`, `fread`, `fwrite`, `fprintf`, `fscanf`...) prendront ce `FILE *` comme argument pour savoir sur quel fichier/flux agir.

```
FILE *mon_fichier_ptr; // Déclaration d'un pointeur de fichier
mon_fichier_ptr = NULL; // Bonne pratique de l'initialiser à NULL
```

Ouvrir un fichier : `fopen()`

Avant de pouvoir lire ou écrire dans un fichier sur disque, il faut l'**ouvrir**. La fonction `fopen()` est utilisée pour cela. Elle tente d'établir un lien entre un nom de fichier physique et un flux `FILE *` dans votre programme, en spécifiant le mode d'accès souhaité.

Syntaxe : `FILE* fopen(const char *filename, const char *mode);`

- `filename`: Une chaîne de caractères (`const char *`) contenant le **nom du fichier** (et éventuellement son chemin d'accès).
    - Exemples : `"donnees.txt"`, `"config.ini"`, `"../images/logo.png"`, `"C:\\Users\\Nom\\Documents\\rapport.dat"` (attention aux backslashes sous Windows, il faut les doubler dans une chaîne C ou utiliser des slashes simples `/`), `"/home/user/data/input.bin"`.
- `mode`: Une chaîne de caractères spécifiant le **mode d'ouverture**. Les modes les plus courants sont :
    - **Modes Texte :**
        - `"r"` : **Read (Lecture)**. Ouvre un fichier *existant* en lecture seule. Le curseur (position de lecture) est placé au début du fichier. Si le fichier n'existe pas, `fopen` échoue et retourne `NULL`.
        - `"w"` : **Write (Écriture)**. Ouvre un fichier en écriture. Si le fichier existe, son contenu est **effacé (tronqué à zéro)**. S'il n'existe pas, il est **créé**. Le curseur est placé au début.
        - `"a"` : **Append (Ajout)**. Ouvre un fichier en écriture pour **ajouter** des données à la fin. Si le fichier existe, le curseur est positionné à la fin pour que les écritures suivantes s'ajoutent. S'il n'existe pas, il est créé.
        - `"r+"` : Lecture et Écriture. Le fichier *doit exister*. Curseur au début. Permet de lire et d'écrire dans le fichier.
        - `"w+"` : Lecture et Écriture. Crée le fichier ou **tronque** un fichier existant. Curseur au début.
        - `"a+"` : Lecture et Ajout. Crée le fichier s'il n'existe pas. Le curseur initial pour la lecture est au début, mais les écritures se font **toujours à la fin** du fichier (append).
    - **Modes Binaires :** On obtient les modes binaires en ajoutant la lettre `b` à la fin des modes texte (ex: `"rb"`, `"wb"`, `"ab"`, `"r+b"`, `"w+b"`, `"a+b"`). Le mode binaire est **essentiel** pour lire/écrire des fichiers qui ne sont pas du texte pur (images, sons, exécutables, données structurées sauvegardées directement depuis la mémoire). La principale différence pratique concerne la manière dont les caractères de fin de ligne sont traités (sous Windows, le mode texte peut

convertir `\n` en `\r\n` à l'écriture et inversement à la lecture, ce qui corromprait des données binaires). **Utilisez toujours le mode binaire pour les fichiers non-texte.**

- **Valeur de retour :**
  - Si l'ouverture réussit : `fopen` retourne un pointeur `FILE *` valide, qui représente le flux ouvert associé au fichier.
  - Si une erreur se produit (fichier non trouvé en mode lecture, permissions insuffisantes pour lire/écrire, disque plein, chemin invalide, trop de fichiers ouverts...) : `fopen` retourne `NULL`.

Vérification `NULL` OBLIGATOIRE : Il est **impératif** de toujours vérifier si `fopen` a retourné `NULL` avant de tenter toute autre opération sur le pointeur `FILE *`.

```c
#include <stdio.h>
#include <stdlib.h> // Pour exit() et EXIT_FAILURE

int main() {
    FILE *fichier_lecture = NULL;
    FILE *fichier_ecriture = NULL;
    FILE *fichier_binaire = NULL;

    // --- Tentative d'ouverture en lecture ---
    printf("Tentative d'ouverture de 'mon_fichier.txt' en lecture (mode
'r')...\n");
    fichier_lecture = fopen("mon_fichier.txt", "r");

    // Vérification
    if (fichier_lecture == NULL) {
        // perror affiche un message d'erreur système plus détaillé
        perror("Erreur lors de l'ouverture en lecture");
        // On pourrait continuer, mais ici on signale l'échec potentiel
        printf("  Le fichier n'existe peut-être pas ou problème de
permissions.\n");
    } else {
        printf("  Ouverture en lecture réussie. Pointeur FILE* : %p\n",
(void*)fichier_lecture);
        // !!! Important: Il faudra fermer ce fichier plus tard avec
fclose() !!!
        fclose(fichier_lecture); // On le ferme tout de suite pour
l'exemple
        fichier_lecture = NULL;
    }

    // --- Tentative d'ouverture en écriture ---
    printf("\nTentative d'ouverture de 'sortie.log' en écriture (mode
'w')...\n");
    fichier_ecriture = fopen("sortie.log", "w"); // Crée le fichier ou
l'écrase

    // Vérification
    if (fichier_ecriture == NULL) {
        perror("Erreur lors de l'ouverture en écriture");
```

```
        return EXIT_FAILURE; // Quitter si on ne peut pas écrire
    } else {
        printf("  Ouverture/Création en écriture réussie. Pointeur FILE* :
%p\n", (void*)fichier_ecriture);
        // ... On pourrait écrire ici avec fprintf, fputs ...
        printf("  Fermeture du fichier d'écriture...\n");
        fclose(fichier_ecriture); // Fermer après usage
        fichier_ecriture = NULL;
    }

     // --- Tentative d'ouverture en écriture binaire ---
    printf("\nTentative d'ouverture de 'donnees.bin' en écriture binaire
(mode 'wb')...\n");
    fichier_binaire = fopen("donnees.bin", "wb");

    if (fichier_binaire == NULL) {
        perror("Erreur lors de l'ouverture en écriture binaire");
        return EXIT_FAILURE;
    } else {
        printf("  Ouverture/Création en écriture binaire réussie. Pointeur
FILE* : %p\n", (void*)fichier_binaire);
        // ... On pourrait écrire ici avec fwrite ...
        fclose(fichier_binaire);
        fichier_binaire = NULL;
    }

    return 0;
}
```

Fermer un fichier : `fclose()`

Chaque fichier ouvert avec succès par `fopen` **doit impérativement être fermé** lorsque vous avez fini de l'utiliser. La fonction `fclose()` s'en charge.

Syntaxe : `int fclose(FILE *stream);`

- `stream`: Le pointeur `FILE *` (retourné par `fopen`) du flux que vous souhaitez fermer.
- **Actions effectuées par `fclose` :**
    1. **Vide les tampons (Flushing) :** Si des données ont été écrites dans le flux (par exemple avec `fprintf` ou `fwrite`), elles sont souvent mises en mémoire tampon (buffer) par la bibliothèque C pour des raisons de performance (écrire sur le disque est lent). `fclose` s'assure que toutes les données restantes dans le tampon sont effectivement écrites sur le support physique (disque). **Oublier `fclose` peut donc entraîner une perte de données écrites !**
    2. **Libère les ressources :** Dissocie le pointeur `FILE *` du fichier physique et libère les ressources système (descripteurs de fichiers, tampons mémoire internes, etc.) qui étaient associées à ce flux ouvert. Les systèmes d'exploitation ont une limite (souvent généreuse, mais existante) au nombre de fichiers qu'un programme peut ouvrir simultanément. Ne pas fermer les fichiers peut épuiser ces ressources.
- **Valeur de retour :**

- `0` si la fermeture réussit (y compris le vidage des tampons).
- `EOF` (End Of File, une constante entière négative, souvent -1) en cas d'erreur (par exemple, une erreur lors de l'écriture finale des tampons sur le disque, disque plein, etc.). Vérifier cette valeur est une bonne pratique pour détecter les erreurs d'écriture qui pourraient ne pas avoir été signalées plus tôt, bien que ce soit moins courant que de vérifier le retour de `fopen`.

Règle d'or : Toujours appeler `fclose()` pour chaque `fopen()` réussi, dès que vous n'avez plus besoin du fichier. Une fois `fclose()` appelée, le pointeur `FILE *` devient invalide et ne doit plus être utilisé. Il est conseillé de le mettre à `NULL` après `fclose`.

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

int main() {
    FILE *fichier = NULL;
    const char *nom_fichier = "fichier_temporaire.txt";

    // Ouvrir en mode écriture ('w')
    fichier = fopen(nom_fichier, "w");
    if (fichier == NULL) {
        perror("Impossible d'ouvrir le fichier en écriture");
        return EXIT_FAILURE;
    }
    printf("Fichier '%s' ouvert en mode écriture.\n", nom_fichier);

    // Écrire quelque chose dans le fichier (sera mis en tampon)
    fprintf(fichier, "Première ligne.\n");
    fprintf(fichier, "Deuxième ligne, nombre = %d.\n", 123);
    printf("Données écrites dans le buffer du fichier...\n");

    // Fermer le fichier : c'est ici que les données sont garanties d'être
écrites sur disque
    printf("Tentative de fermeture du fichier (fclose)...\n");
    int resultat_close = fclose(fichier);

    // Vérifier le résultat de fclose
    if (resultat_close == 0) {
        printf("Fichier '%s' fermé avec succès (données écrites).\n",
nom_fichier);
    } else { // EOF a été retourné
        fprintf(stderr, "Erreur lors de la fermeture du fichier '%s'.\n",
nom_fichier);
        perror("Raison possible (fclose)");
        // Même en cas d'erreur, la ressource est généralement libérée,
        // mais les données n'ont peut-être pas été écrites correctement.
    }

    // Le pointeur 'fichier' est maintenant invalide. Mettre à NULL.
    fichier = NULL;

    // Tenter d'écrire à nouveau générerait une erreur ou un crash
    // if (fichier != NULL) { // Cette condition serait fausse
```

```
    //     fprintf(fichier, "Test après fermeture.\n");
    // }

    return 0;
}
```

Lire et Écrire dans un fichier (Mode Texte)

Une fois un fichier ouvert avec le bon mode (`"r"`, `"w"`, `"a"`, `"r+"`, `"w+"`, `"a+"`), vous pouvez y lire ou y écrire des données. Pour les fichiers texte, on utilise souvent des fonctions qui traitent le contenu comme du texte formaté ou des lignes/caractères.

Fonctions formatées (similaires à `printf`/`scanf`) :

- **`int fprintf(FILE *stream, const char *format, ...);`**
  - Fonctionne exactement comme `printf`, mais écrit dans le flux `stream` (le `FILE *` retourné par `fopen`) au lieu de `stdout`.
  - Utilise les mêmes spécificateurs de format (`%d`, `%s`, `%.2f`, etc.).
  - Retourne le nombre de caractères écrits, ou une valeur négative en cas d'erreur.
- **`int fscanf(FILE *stream, const char *format, ...);`**
  - Fonctionne exactement comme `scanf`, mais lit depuis le flux `stream` au lieu de `stdin`.
  - **Nécessite les adresses (`&`)** des variables où stocker les données lues.
  - Retourne le nombre d'éléments lus et assignés avec succès, `0` si l'entrée ne correspondait pas au premier format, ou `EOF` si la fin du fichier est atteinte *avant* toute lecture réussie ou si une erreur de lecture se produit. **Vérifier la valeur de retour est crucial.**
  - **Attention :** `fscanf` partage les mêmes faiblesses que `scanf` (fragilité face aux erreurs de format, danger de `%s` sans largeur, gestion délicate des espaces et `\n`). Lire ligne par ligne avec `fgets` puis analyser avec `sscanf` est souvent plus robuste.

Fonctions par caractère :

- **`int fputc(int c, FILE *stream);`**
  - Écrit le caractère `c` dans le flux `stream`.
  - Retourne le caractère écrit, ou `EOF` en cas d'erreur.
- **`int fgetc(FILE *stream);`**
  - Lit le prochain caractère depuis le flux `stream`.
  - Retourne la valeur `int` du caractère lu, ou `EOF` si la fin du fichier est atteinte ou si une erreur se produit.

Fonctions par chaîne/ligne :

- **`int fputs(const char *s, FILE *stream);`**
  - Écrit la chaîne `s` (jusqu'au `\0` mais sans l'écrire) dans le flux `stream`. **N'ajoute pas** de `\n` automatiquement.
  - Retourne une valeur non négative en cas de succès, ou `EOF` en cas d'erreur.
- **`char *fgets(char *s, int size, FILE *stream);`**

- Lit une ligne (ou jusqu'à `size - 1` caractères) depuis `stream` dans le buffer `s`.
- **Inclut le `\n` final** s'il est lu et s'il y a la place.
- Ajoute **toujours** un `\0` à la fin.
- Retourne `s` en cas de succès, `NULL` en cas d'erreur ou si la fin du fichier est atteinte avant d'avoir lu quoi que ce soit. **C'est la méthode préférée et la plus sûre pour lire du texte ligne par ligne.**

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h> // Pour strlen, strcspn

#define MAX_LIGNE 100

int main() {
    FILE *fichier = NULL;
    const char *nom_fichier = "mon_journal.txt";
    char ligne_lue[MAX_LIGNE];

    // --- Écriture avec fprintf et fputs ---
    fichier = fopen(nom_fichier, "w"); // Ouvre en mode écriture (écrase si
existe)
    if (fichier == NULL) {
        perror("Erreur ouverture écriture");
        return EXIT_FAILURE;
    }
    printf("Fichier '%s' ouvert en écriture.\n", nom_fichier);

    fprintf(fichier, "Journal de bord\n"); // Écrit une ligne formatée
    fprintf(fichier, "Date: %d/%d/%d\n", 27, 4, 2025);
    fputs("Ceci est une entrée simple.\n", fichier); // Écrit une chaîne
(avec \n inclus ici)
    fputs("Une autre entrée.", fichier); // N'ajoute pas de \n
    fputc('\n', fichier); // Ajoute le \n manuellement

    fclose(fichier);
    fichier = NULL;
    printf("Fichier écrit et fermé.\n\n");

    // --- Lecture avec fgets ---
    fichier = fopen(nom_fichier, "r"); // Ouvre en mode lecture
    if (fichier == NULL) {
        perror("Erreur ouverture lecture");
        return EXIT_FAILURE;
    }
    printf("Lecture du fichier '%s' ligne par ligne avec fgets:\n",
nom_fichier);

    int num_ligne = 1;
    // Lit une ligne (max MAX_LIGNE-1 chars) tant que fgets ne retourne pas
NULL
    while (fgets(ligne_lue, sizeof(ligne_lue), fichier) != NULL) {
        // Optionnel: supprimer le \n final lu par fgets
```

```c
        ligne_lue[strcspn(ligne_lue, "\n")] = '\0';

        printf("Ligne %d : %s\n", num_ligne, ligne_lue);
        num_ligne++;
    }

    // Vérifier pourquoi la boucle s'est arrêtée
    if (feof(fichier)) {
        printf("Fin normale du fichier atteinte.\n");
    } else if (ferror(fichier)) {
        perror("Erreur pendant la lecture du fichier");
    }

    fclose(fichier);
    fichier = NULL;
    printf("Fichier lu et fermé.\n");

    // --- Lecture avec fscanf (moins recommandé pour du texte mixte) ---
    /*
    fichier = fopen(nom_fichier, "r");
    if (fichier) {
        char titre[20], type_entree[20];
        int j, m, a, val;
        // Tentatives de lecture formatée (fragile !)
        fscanf(fichier, "%s %s", titre, type_entree); // Lit "Journal" et
"de"
        fscanf(fichier, "%*s %d/%d/%d", &j, &m, &a); // %*s pour ignorer
"Date:", lit la date
        // ... etc ... très complexe et sujet aux erreurs si le format
change.
        fclose(fichier);
    }
    */

    return 0;
}
```

Lire et Écrire dans un fichier (Mode Binaire)

Lorsque vous travaillez avec des fichiers qui ne contiennent pas de texte simple (images, sons, exécutables, archives, sauvegardes de données structurées), ou lorsque vous voulez lire/écrire des blocs de mémoire bruts (comme le contenu d'une structure ou d'un tableau) directement dans un fichier, vous devez utiliser le **mode binaire** (en ajoutant `b` aux modes d'ouverture : `"rb"`, `"wb"`, `"ab"`, `"r+b"`, `"w+b"`, `"a+b"`).

Le mode binaire assure qu'aucune traduction de caractères (comme la conversion des fins de ligne `\n` <-> `\r\n` sous Windows) n'est effectuée. Les octets sont lus et écrits tels quels.

Les fonctions principales pour l'E/S binaire sont `fread` et `fwrite`. Elles sont conçues pour lire et écrire des blocs de mémoire (des "enregistrements" ou "items") d'une taille spécifiée.

- **`size_t fwrite(const void *ptr, size_t size, size_t count, FILE *stream);`**

    - Écrit `count` éléments (items), chacun ayant une taille de `size` octets, depuis le bloc mémoire commençant à l'adresse `ptr`, vers le flux `stream`.
    - `ptr`: Pointeur (`const void *`) vers le début des données en mémoire à écrire.
    - `size`: La taille en octets de *chaque* élément individuel à écrire (souvent obtenu avec `sizeof`).
    - `count`: Le **nombre d'éléments** de taille `size` à écrire. La quantité totale d'octets écrite sera `size * count`.
    - `stream`: Le `FILE *` du fichier ouvert en mode écriture binaire (ex: `"wb"`, `"ab"`, `"w+b"`).
    - **Valeur de retour (`size_t`) :** Le **nombre d'éléments** (`count`) effectivement écrits avec succès. Si cette valeur est inférieure à `count`, une erreur s'est produite (par exemple, disque plein).

- **`size_t fread(void *ptr, size_t size, size_t count, FILE *stream);`**

    - Lit au maximum `count` éléments, chacun ayant une taille de `size` octets, depuis le flux `stream`, et les stocke dans le bloc mémoire commençant à l'adresse `ptr`.
    - `ptr`: Pointeur (`void *`) vers la zone mémoire (buffer) où stocker les données lues. Ce buffer **doit être suffisamment grand** pour contenir `count * size` octets !
    - `size`: La taille en octets de *chaque* élément individuel à lire.
    - `count`: Le nombre maximum d'éléments de taille `size` à essayer de lire.
    - `stream`: Le `FILE *` du fichier ouvert en mode lecture binaire (ex: `"rb"`, `"r+b"`).
    - **Valeur de retour (`size_t`) :** Le **nombre d'éléments** (`count`) effectivement lus avec succès.
        - Si cette valeur est **égale à `count`**, la lecture a réussi complètement.
        - Si cette valeur est **inférieure à `count`** (et potentiellement 0), cela signifie soit que la **fin du fichier** a été atteinte avant d'avoir pu lire tous les éléments demandés, soit qu'une **erreur de lecture** s'est produite. Il faut utiliser `feof()` et `ferror()` (voir section suivante) pour distinguer ces deux cas.

```c
#include <stdio.h>
#include <stdlib.h> // Pour exit, EXIT_FAILURE, NULL
#include <string.h> // Pour strcmp

// Définition d'une structure simple à sauvegarder/charger
typedef struct {
    int id;
    double valeur;
    char nom[30]; // Espace pour le nom + \0
} ElementData;

int main() {
    FILE *fichier_b = NULL;
    const char *nom_fichier = "elements.dat"; // Extension .dat souvent
pour données binaires

    // --- Écriture Binaire d'un tableau de structures ---
    ElementData tableau_a_ecrire[3] = {
        {101, 12.34, "Pomme"},
        {102, 56.78, "Banane"},
        {103, 90.12, "Orange"}
    };
    size_t nb_elements_a_ecrire = 3;
```

```
    fichier_b = fopen(nom_fichier, "wb"); // Ouvre en écriture binaire
(écrase)
    if (fichier_b == NULL) {
        perror("Erreur fopen wb");
        return EXIT_FAILURE;
    }
    printf("Fichier '%s' ouvert en écriture binaire.\n", nom_fichier);

    // Écrire les 3 éléments du tableau en une seule fois
    // ptr: adresse du début du tableau (&tableau_a_ecrire[0] ou juste
tableau_a_ecrire)
    // size: taille d'UNE structure ElementData
    // count: nombre de structures à écrire (3)
    // stream: le pointeur FILE*
    size_t elements_ecrits = fwrite(tableau_a_ecrire, sizeof(ElementData),
nb_elements_a_ecrire, fichier_b);

    if (elements_ecrits != nb_elements_a_ecrire) {
        fprintf(stderr, "Erreur: n'a écrit que %zu éléments sur %zu
demandés.\n",
                elements_ecrits, nb_elements_a_ecrire);
        fclose(fichier_b);
        return EXIT_FAILURE;
    }
    printf("%zu éléments écrits avec succès.\n", elements_ecrits);

    fclose(fichier_b);
    fichier_b = NULL;
    printf("Fichier écrit et fermé.\n\n");

    // --- Lecture Binaire du tableau de structures ---
    ElementData tableau_lu[3]; // Buffer pour recevoir les données lues
    size_t nb_elements_a_lire = 3;

    fichier_b = fopen(nom_fichier, "rb"); // Ouvre en lecture binaire
    if (fichier_b == NULL) {
        perror("Erreur fopen rb");
        return EXIT_FAILURE;
    }
    printf("Fichier '%s' ouvert en lecture binaire.\n", nom_fichier);

    // Lire au maximum 3 éléments de taille sizeof(ElementData) depuis le
fichier
    // et les stocker dans tableau_lu
    size_t elements_lus = fread(tableau_lu, sizeof(ElementData),
nb_elements_a_lire, fichier_b);

    printf("%zu éléments lus depuis le fichier.\n", elements_lus);

    // Vérifier si on a bien lu le nombre attendu
    if (elements_lus < nb_elements_a_lire) {
        if (feof(fichier_b)) {
            fprintf(stderr, "Erreur: Fin de fichier atteinte prématurément
(fichier trop court ?).\n");
```

```
        } else if (ferror(fichier_b)) {
            perror("Erreur de lecture pendant fread");
        } else {
             fprintf(stderr, "Erreur inconnue pendant fread.\n");
        }
        fclose(fichier_b);
        return EXIT_FAILURE;
    }

    // Afficher les données lues
    printf("Contenu lu depuis le fichier :\n");
    for (size_t i = 0; i < elements_lus; i++) {
        printf("  Element %zu: ID=%d, Valeur=%.2f, Nom=\"%s\"\n",
               i, tableau_lu[i].id, tableau_lu[i].valeur,
tableau_lu[i].nom);
    }

    fclose(fichier_b);
    fichier_b = NULL;
    printf("Fichier lu et fermé.\n");

    return 0;
}
```

Important : L'écriture binaire avec `fwrite` d'une structure ou d'un tableau est **directement liée à la représentation en mémoire** de ces données sur la machine où le programme a été compilé. Les fichiers binaires ainsi créés ne sont **pas nécessairement portables** entre :

- Des machines avec des **architectures différentes** (ex: 32 bits vs 64 bits, où la taille des `int` ou des pointeurs peut changer).
- Des machines avec un **boutisme (endianness)** différent (ordre des octets pour les types multi-octets comme `int`, `float`, `double`).
- Des compilateurs différents qui pourraient ajouter un **rembourrage (padding)** différent à l'intérieur des structures pour des raisons d'alignement.

Pour une portabilité maximale des données, il faut utiliser des formats de sérialisation standardisés (comme JSON, XML, Protocol Buffers, ou définir son propre format binaire indépendant de l'architecture). Cependant, pour des sauvegardes rapides sur la même machine ou entre machines compatibles, `fread`/`fwrite` sont très efficaces.

Positionnement dans un fichier (`fseek`, `ftell`, `rewind`)

Parfois, on ne veut pas lire ou écrire un fichier séquentiellement du début à la fin. On peut avoir besoin de se déplacer directement à une position spécifique dans le fichier.

- **`long ftell(FILE *stream);`**
    - Retourne la **position actuelle** du curseur de lecture/écriture dans le flux `stream`, exprimée en nombre d'octets depuis le début du fichier.
    - Retourne `-1L` en cas d'erreur (et positionne `errno`).

- `int fseek(FILE *stream, long offset, int whence);`
    - Déplace le curseur de lecture/écriture du flux `stream`.
    - `offset`: Le nombre d'octets de déplacement (peut être positif ou négatif).
    - `whence`: Le point de référence pour le déplacement :
        - `SEEK_SET` : Déplacement par rapport au **début** du fichier (`offset` doit être >= 0).
        - `SEEK_CUR` : Déplacement par rapport à la **position actuelle** du curseur (`offset` peut être +/-).
        - `SEEK_END` : Déplacement par rapport à la **fin** du fichier (`offset` est souvent <= 0).
    - Retourne `0` en cas de succès, une valeur non nulle en cas d'erreur.
    - **Attention :** L'utilisation de `fseek` avec des valeurs `whence` autres que `SEEK_SET` et un `offset` non nul sur des fichiers ouverts en mode **texte** peut avoir un comportement non portable ou non défini, à cause des conversions potentielles de fin de ligne. Pour un positionnement fiable, utilisez le mode **binaire**.
- `void rewind(FILE *stream);`
    - Replace le curseur de lecture/écriture au **début** du flux `stream`.
    - Équivalent à `fseek(stream, 0L, SEEK_SET)`, mais efface aussi les indicateurs d'erreur et de fin de fichier pour le flux.

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

int main() {
    FILE *fichier = NULL;
    const char *nom_fichier = "positionnement.txt";
    long position;

    // Créer un fichier avec du contenu
    fichier = fopen(nom_fichier, "w");
    if (!fichier) { perror("fopen w"); return 1; }
    fprintf(fichier, "0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ");
    fclose(fichier);
    fichier = NULL;

    // Ouvrir en lecture binaire pour un positionnement fiable
    fichier = fopen(nom_fichier, "rb");
    if (!fichier) { perror("fopen rb"); return 1; }
    printf("Fichier '%s' ouvert en lecture binaire.\n", nom_fichier);

    // 1. Lire la position initiale (devrait être 0)
    position = ftell(fichier);
    printf("Position initiale : %ld\n", position);

    // 2. Lire le 5ème caractère (indice 4)
    // Se déplacer de 4 octets depuis le début
    if (fseek(fichier, 4L, SEEK_SET) != 0) {
        perror("Erreur fseek SEEK_SET");
    } else {
        int c = fgetc(fichier);
```

```
        printf("Après fseek(4, SEEK_SET), caractère lu : %c\n", c); //
Devrait être '4'
        position = ftell(fichier);
        printf("Position actuelle : %ld\n", position); // Devrait être 5
    }

    // 3. Se déplacer de 10 octets depuis la position actuelle
    if (fseek(fichier, 10L, SEEK_CUR) != 0) {
         perror("Erreur fseek SEEK_CUR");
    } else {
        int c = fgetc(fichier);
        printf("Après fseek(10, SEEK_CUR), caractère lu : %c\n", c); //
Devrait être 'E' (indice 14)
        position = ftell(fichier);
        printf("Position actuelle : %ld\n", position); // Devrait être 15
    }

    // 4. Se déplacer à 5 octets avant la fin
    if (fseek(fichier, -5L, SEEK_END) != 0) {
        perror("Erreur fseek SEEK_END");
    } else {
        position = ftell(fichier);
        printf("Après fseek(-5, SEEK_END), position : %ld\n", position); //
36 - 5 = 31
        int c = fgetc(fichier);
        printf("Caractère lu depuis cette position : %c\n", c); // Devrait
être 'V' (indice 31)
    }

    // 5. Revenir au début avec rewind
    rewind(fichier);
    position = ftell(fichier);
    printf("\nAprès rewind(), position : %ld\n", position); // 0
    int c = fgetc(fichier);
    printf("Caractère lu après rewind : %c\n", c); // Devrait être '0'

    fclose(fichier);
    printf("Fichier fermé.\n");

    return 0;
}
```

Gestion des erreurs de fichiers (`feof`, `ferror`, `perror`, `clearerr`)

En plus de vérifier `NULL` après `fopen` et les valeurs de retour des fonctions de lecture/écriture (`fscanf`, `fread`, `fwrite`, `fgetc`, `fgets`...), il existe d'autres fonctions pour diagnostiquer plus précisément les problèmes avec les flux de fichiers :

- **`int feof(FILE *stream);`**

    - Teste l'indicateur de **fin de fichier (End Of File)** pour le flux `stream`. Cet indicateur est positionné lorsqu'une opération de lecture tente de lire *au-delà* de la fin du fichier.
    - Retourne une valeur **non nulle (vrai)** si l'indicateur EOF est positionné, et **zéro (faux)** sinon.
    - **Important :** `feof` ne devient vrai qu'**après** une tentative de lecture qui a échoué *parce que* la fin du fichier a été atteinte. Il ne prédit pas si la *prochaine* lecture atteindra la fin. À utiliser typiquement *après* une fonction comme `fread` ou `fgetc` qui a retourné une valeur indiquant un échec (nombre d'éléments lus < demandé, ou `EOF`), pour déterminer si cet échec est dû à la fin normale du fichier.

- **`int ferror(FILE *stream);`**
    - Teste l'indicateur d'**erreur** pour le flux `stream`. Cet indicateur est positionné si une erreur d'E/S physique (lecture ou écriture) s'est produite sur le flux (ex: problème de disque, flux fermé prématurément...).
    - Retourne une valeur **non nulle (vrai)** si l'indicateur d'erreur est positionné, et **zéro (faux)** sinon.
    - À utiliser *après* une opération d'E/S qui a échoué pour confirmer qu'une erreur système s'est produite (plutôt que juste la fin du fichier pour une lecture).

- **`void perror(const char *s);`**
    - Fonction (définie dans `<stdio.h>` mais utilise la variable globale `errno` de `<errno.h>`) très utile pour afficher un message d'erreur système compréhensible sur `stderr`.
    - Affiche d'abord la chaîne de caractères `s` fournie par l'utilisateur (qui devrait décrire le contexte de l'erreur, ex: "Erreur fopen"), suivie de ": ", puis d'un message textuel décrivant la dernière erreur système enregistrée dans la variable globale `errno`.
    - Typiquement appelée juste après l'échec d'une fonction système (comme `fopen`, `fread`, `fwrite`, `fclose`...) ou après avoir détecté une erreur avec `ferror`.

- **`void clearerr(FILE *stream);`**
    - Réinitialise (remet à zéro) les indicateurs de fin de fichier (`feof`) et d'erreur (`ferror`) pour le flux `stream`. Utile si vous voulez tenter de continuer les opérations sur le flux après avoir détecté et potentiellement géré une condition d'erreur ou de fin de fichier (par exemple, utiliser `fseek` pour revenir en arrière après avoir atteint `EOF`).

```c
#include <stdio.h>
#include <stdlib.h>
#include <errno.h> // Pour errno (utilisé par perror)

int main() {
    FILE *fichier = NULL;
    const char *nom_fichier_inexistant =
"fichier_qui_n_existe_pas_vraiment.xyz";
    const char *nom_fichier_test = "test_lecture_erreur.txt";
    int buffer[10];
    size_t lus;

    // --- Test d'erreur à l'ouverture avec perror ---
    printf("Tentative d'ouverture de '%s' en lecture...\n",
nom_fichier_inexistant);
```

```c
    fichier = fopen(nom_fichier_inexistant, "r");

    if (fichier == NULL) {
        // Afficher notre message + le message d'erreur système détaillé
        fprintf(stderr, "ERREUR: Impossible d'ouvrir le fichier '%s'.\n",
nom_fichier_inexistant);
        perror("Cause système (fopen)"); // Donne la raison (ex: No such
file or directory)
        // errno contient le code d'erreur numérique si besoin
        // printf("(Code d'erreur système errno: %d)\n", errno);
    } else {
        // Ne devrait pas arriver si le fichier n'existe pas
        printf("Fichier ouvert ? Ceci ne devrait pas s'afficher.\n");
        fclose(fichier);
    }

    // --- Test feof et ferror après fread ---
    // Créer un petit fichier pour le test
    fichier = fopen(nom_fichier_test, "wb"); // Écriture binaire
    if (!fichier) { perror("fopen wb test"); return 1; }
    int data[] = {1, 2, 3};
    fwrite(data, sizeof(int), 3, fichier);
    fclose(fichier);

    // Essayer de lire plus que ce qu'il y a dans le fichier
    fichier = fopen(nom_fichier_test, "rb");
    if (!fichier) { perror("fopen rb test"); return 1; }
    printf("\nLecture depuis '%s' (contient 3 entiers)...\n",
nom_fichier_test);

    // Essayer de lire 10 entiers
    lus = fread(buffer, sizeof(int), 10, fichier);
    printf("fread a tenté de lire 10 éléments, et en a lu %zu.\n", lus);

    if (lus < 10) {
        printf("Moins d'éléments lus que demandé. Vérification de la cause
:\n");
        if (feof(fichier)) {
            // C'est le cas attendu ici
            printf("  -> Condition EOF détectée (fin normale du fichier
atteinte).\n");
        }
        if (ferror(fichier)) {
            // Ne devrait pas arriver dans ce cas simple, sauf si problème
disque etc.
            perror("  -> Erreur de lecture détectée par ferror");
            fprintf(stderr, "     (Code d'erreur système errno: %d)\n",
errno);
        }
    } else {
        printf("Tous les 10 éléments ont été lus (ne devrait pas arriver
ici).\n");
    }
```

```
    // On peut effacer les indicateurs si on veut, par exemple, rembobiner
et relire
    clearerr(fichier); // Efface les flags EOF et erreur
    rewind(fichier);   // Remet au début (efface aussi les flags)
    printf("\nAprès clearerr() et rewind(), feof=%d, ferror=%d\n",
feof(fichier), ferror(fichier));

    fclose(fichier);
    fichier = NULL;
    printf("Fichier '%s' fermé.\n", nom_fichier_test);

    return 0;
}
```

La gestion des fichiers est une compétence essentielle en C pour rendre les programmes plus utiles et capables de traiter des données persistantes. Maîtriser `fopen`, `fclose`, les différentes fonctions de lecture/écriture en mode texte (`fprintf`, `fgets`...) et binaire (`fread`, `fwrite`), ainsi que la gestion rigoureuse des erreurs (`NULL`, `feof`, `ferror`, `perror`) et la fermeture systématique des fichiers, est crucial pour écrire des programmes C robustes et fiables qui interagissent avec le système de fichiers.

---

21. Introduction à SDL2

Nous avons couvert les bases et les aspects avancés du langage C lui-même. Cependant, pour créer des applications graphiques interactives comme un jeu d'échecs avec une interface visuelle, le C standard ne suffit pas. Il ne fournit pas de fonctions pour ouvrir des fenêtres, dessiner des images, gérer la souris, jouer du son, etc.

Pour cela, nous devons utiliser des **bibliothèques externes**. Une des bibliothèques multiplateformes les plus populaires et les plus adaptées pour le développement de jeux 2D (et comme base pour la 3D) en C/C++ est **SDL**.

Qu'est-ce que SDL ? (Simple DirectMedia Layer)

SDL (Simple DirectMedia Layer) est une bibliothèque de développement **multiplateforme** (fonctionne sur Windows, macOS, Linux, iOS, Android...) conçue pour fournir un accès de **bas niveau** au matériel audio, au clavier, à la souris, au joystick, et au matériel graphique via OpenGL ou Direct3D. Elle est écrite en C, fonctionne bien avec C++, et possède des bindings pour de nombreux autres langages.

SDL se charge de masquer les différences entre les systèmes d'exploitation, vous permettant d'écrire du code (par exemple, pour ouvrir une fenêtre, afficher une image, détecter un clic de souris) qui fonctionnera de manière similaire sur différentes plateformes.

SDL2 est la version actuelle et majeure de la bibliothèque. Elle est souvent accompagnée de bibliothèques "satellites" pour des fonctionnalités spécifiques :

- **SDL2_image :** Pour charger divers formats d'images (PNG, JPG, BMP, GIF, TIF, etc.).
- **SDL2_ttf :** Pour charger et rendre du texte en utilisant des polices TrueType (`.ttf`).
- **SDL2_mixer :** Pour la lecture de formats audio et la gestion des effets sonores et de la musique.

- **SDL2_net :** Pour la programmation réseau de bas niveau.

Dans le code du jeu d'échecs, nous avons vu l'utilisation de `SDL2`, `SDL2_image` (pour charger les images de l'échiquier et des pièces), et `SDL2_ttf` (pour afficher du texte).

Installation de SDL2, SDL2_image, SDL2_ttf

Avant de pouvoir utiliser SDL dans vos projets C, vous devez installer les bibliothèques de développement sur votre système. La méthode varie selon le système d'exploitation :

- **Linux (Debian/Ubuntu) :** Utilisez le gestionnaire de paquets `apt`. Vous avez besoin des paquets de développement (`-dev`) qui contiennent les fichiers d'en-tête (`.h`) et les fichiers nécessaires à l'édition de liens.

```
sudo apt update
# Bibliothèque principale SDL2
sudo apt install libsdl2-dev
# Bibliothèque pour les images
sudo apt install libsdl2-image-dev
# Bibliothèque pour les polices TTF
sudo apt install libsdl2-ttf-dev
# (Optionnel) Pour le son et le réseau
# sudo apt install libsdl2-mixer-dev libsdl2-net-dev
```

- **Linux (Fedora/CentOS/RHEL) :** Utilisez le gestionnaire de paquets `dnf` (ou `yum` sur les anciennes versions).

```
sudo dnf update
# Bibliothèque principale SDL2
sudo dnf install SDL2-devel
# Bibliothèque pour les images
sudo dnf install SDL2_image-devel
# Bibliothèque pour les polices TTF
sudo dnf install SDL2_ttf-devel
# (Optionnel) Pour le son et le réseau
# sudo dnf install SDL2_mixer-devel SDL2_net-devel
```

- **macOS :** La méthode la plus simple est d'utiliser un gestionnaire de paquets comme **Homebrew** (https://brew.sh/). Si vous n'avez pas Homebrew, installez-le d'abord. Ensuite, dans le Terminal :

```
# Mettre à jour Homebrew
brew update

# Installer SDL2 et les bibliothèques satellites
```

```
brew install sdl2 sdl2_image sdl2_ttf sdl2_mixer sdl2_net
```

- **Windows :** C'est souvent un peu plus complexe. Plusieurs approches :
  - **Télécharger les bibliothèques de développement précompilées :** Allez sur le site officiel de SDL (https://www.libsdl.org/) et des bibliothèques satellites (liens souvent sur le site SDL) et téléchargez les archives "Development Libraries" pour MinGW (si vous utilisez GCC via MinGW/MSYS2) ou pour Visual C++ (si vous utilisez Visual Studio). Vous devrez ensuite décompresser ces archives et configurer manuellement les chemins d'inclusion et de liaison dans votre projet/Makefile/IDE.
  - **Utiliser un gestionnaire de paquets pour Windows :**
    - **MSYS2 :** Si vous utilisez GCC via MSYS2, vous pouvez souvent installer les paquets SDL directement avec `pacman` (les noms peuvent varier, cherchez `mingw-w64-x86_64-SDL2`, `mingw-w64-x86_64-SDL2_image`, etc.).
    - **vcpkg (https://vcpkg.io/) :** Un gestionnaire de paquets C/C++ de Microsoft qui simplifie grandement l'installation de bibliothèques comme SDL2 pour Visual Studio ou MinGW. Après avoir installé vcpkg, vous pouvez faire `vcpkg install sdl2 sdl2-image sdl2-ttf`.

Configuration du projet (Includes et Linker flags)

Une fois les bibliothèques installées, vous devez indiquer à votre compilateur et à votre lieur où les trouver et comment les utiliser.

1. **Includes (`#include`) :** Dans vos fichiers `.c`, vous inclurez les en-têtes SDL nécessaires :

   ```
   #include <SDL2/SDL.h>
   #include <SDL2/SDL_image.h> // Si vous utilisez SDL_image
   #include <SDL2/SDL_ttf.h>   // Si vous utilisez SDL_ttf
   // etc.
   ```

   - **Option `-I` (Compilateur) :** Si les fichiers `.h` de SDL ne sont pas dans un chemin standard connu de votre compilateur (ce qui peut arriver sous Windows si vous avez téléchargé les bibliothèques manuellement), vous devrez ajouter le chemin vers le répertoire `include` de SDL à votre commande de compilation avec l'option `-I`. Par exemple : `gcc -c main.c -o main.o -I/chemin/vers/sdl2/include`

2. **Édition de Liens (`-L`, `-l`) :** Vous devez indiquer au lieur de lier votre programme avec les bibliothèques SDL compilées (`.so`, `.dylib`, `.dll`/`.lib`).

   - **Option `-L` (Lieur) :** Si les fichiers de bibliothèque (`.a`, `.so`, `.lib`, `.dylib`) ne sont pas dans un chemin standard, ajoutez le chemin vers le répertoire `lib` de SDL avec l'option `-L`. Par exemple : `-L/chemin/vers/sdl2/lib`
   - **Option `-l` (Lieur) :** Spécifie les bibliothèques à lier. Le nom suit généralement `-l` sans le préfixe `lib` et sans l'extension.
     - `-lSDL2` : Pour la bibliothèque principale SDL2.
     - `-lSDL2_image` : Pour SDL_image.

        - `-lSDL2_ttf` : Pour SDL_ttf.
        - `-lSDL2_mixer` : Pour SDL_mixer.
        - `-lSDL2_net` : Pour SDL_net.
    - **Cas spécial `main` sous Windows/MinGW :** Souvent, pour SDL2, il faut aussi lier avec `-lmingw32` et parfois `-mwindows` (pour une application graphique sans console).

Exemple de commande de compilation et liaison complète avec GCC (Linux/macOS, en supposant que SDL est dans les chemins standard) :

```
# Compiler les fichiers objets (en ajoutant les flags SDL si nécessaire,
# bien que -I soit souvent géré par pkg-config ou les chemins par défaut)
# gcc -Wall -Wextra -std=c11 -g -c main.c -o main.o $(sdl2-config --cflags)
# gcc -Wall -Wextra -std=c11 -g -c jeu.c -o jeu.o $(sdl2-config --cflags)
# ... (sdl2-config --cflags donne les options -I et -D nécessaires)

# Lier les objets avec les bibliothèques SDL
# gcc main.o jeu.o -o mon_jeu_sdl -lSDL2 -lSDL2_image -lSDL2_ttf
# Ou en utilisant sdl2-config pour obtenir les flags de liaison :
# gcc main.o jeu.o -o mon_jeu_sdl $(sdl2-config --libs) -lSDL2_image -
lSDL2_ttf

# Commande plus complète et portable avec pkg-config (si installé)
# pkg-config --cflags --libs sdl2 SDL2_image SDL2_ttf
# gcc main.c jeu.c -o mon_jeu_sdl $(pkg-config --cflags --libs sdl2
SDL2_image SDL2_ttf) -Wall -Wextra -g
```

Note sur `pkg-config` et `sdl2-config` : Sur les systèmes Unix-like, les outils `pkg-config` ou `sdl2-config` (fourni avec SDL) peuvent être utilisés pour obtenir automatiquement les bonnes options `-I`, `-L`, et `-l` pour votre système, rendant les commandes de compilation plus portables.

Configurer correctement l'environnement de compilation pour utiliser des bibliothèques externes comme SDL est une étape initiale souvent un peu délicate, mais indispensable pour aller au-delà des programmes console basiques en C.

---

22. Initialisation et Fenêtrage

Maintenant que SDL2 est (théoriquement) installé et que vous savez comment configurer votre projet pour l'utiliser, voyons les premières étapes concrètes pour créer une application graphique : initialiser la bibliothèque, créer une fenêtre et préparer le dessin.

Initialisation de SDL (`SDL_Init`)

Avant d'utiliser la plupart des fonctionnalités de SDL, vous devez **initialiser** les sous-systèmes dont vous aurez besoin. La fonction principale pour cela est `SDL_Init()`.

Syntaxe : `int SDL_Init(Uint32 flags);`

- `flags`: Un entier non signé (`Uint32`) composé d'un ou plusieurs **flags** (indicateurs) combinés avec l'opérateur OU bitwise (`|`). Ces flags indiquent quels sous-systèmes initialiser. Les plus courants sont :
  - `SDL_INIT_VIDEO` : Nécessaire pour toute opération vidéo (fenêtres, rendu, événements clavier/souris). C'est presque toujours requis.
  - `SDL_INIT_AUDIO` : Pour la gestion du son.
  - `SDL_INIT_TIMER` : Pour les fonctions de gestion du temps et des timers.
  - `SDL_INIT_EVENTS` : Pour la gestion des événements (généralement inclus par `SDL_INIT_VIDEO`).
  - `SDL_INIT_JOYSTICK` : Pour la gestion des joysticks/manettes.
  - `SDL_INIT_GAMECONTROLLER` : Une interface de plus haut niveau pour les manettes de jeu.
  - `SDL_INIT_EVERYTHING` : Initialise tous les sous-systèmes mentionnés ci-dessus. Pratique pour commencer, mais il est préférable de n'initialiser que ce dont on a besoin.
- **Valeur de retour :** Retourne `0` en cas de succès, ou une valeur négative en cas d'erreur.

Il est crucial de vérifier la valeur de retour de `SDL_Init()` !

Création d'une fenêtre (`SDL_CreateWindow`)

Une fois SDL initialisé (au moins `SDL_INIT_VIDEO`), vous pouvez créer une fenêtre dans laquelle votre application s'affichera.

Syntaxe : `SDL_Window* SDL_CreateWindow(const char *title, int x, int y, int w, int h, Uint32 flags);`

- `title`: La chaîne de caractères (`const char *`) qui apparaîtra dans la barre de titre de la fenêtre.
- `x`, `y`: La position initiale (coordonnées x, y) du coin supérieur gauche de la fenêtre sur l'écran. Vous pouvez utiliser `SDL_WINDOWPOS_UNDEFINED` ou `SDL_WINDOWPOS_CENTERED` pour laisser le système choisir ou centrer la fenêtre.
- `w`, `h`: La largeur (width) et la hauteur (height) de la fenêtre en pixels.
- `flags`: Des indicateurs (`Uint32`) pour spécifier les propriétés de la fenêtre, combinés avec `|`. Quelques flags courants :
  - `SDL_WINDOW_SHOWN`: La fenêtre est visible dès sa création (sinon `SDL_WINDOW_HIDDEN`).
  - `SDL_WINDOW_RESIZABLE`: Permet à l'utilisateur de redimensionner la fenêtre.
  - `SDL_WINDOW_FULLSCREEN`: Tente de créer une fenêtre en plein écran.
  - `SDL_WINDOW_FULLSCREEN_DESKTOP`: Plein écran à la résolution actuelle du bureau (souvent préféré à `SDL_WINDOW_FULLSCREEN`).
  - `SDL_WINDOW_BORDERLESS`: Fenêtre sans bordures ni barre de titre.
  - `SDL_WINDOW_OPENGL`: Indique que vous utiliserez OpenGL pour le rendu dans cette fenêtre (nécessite une configuration supplémentaire).
- **Valeur de retour :** Un pointeur vers la structure `SDL_Window` créée (`SDL_Window*`) en cas de succès, ou `NULL` en cas d'erreur.

Vérifiez toujours si `SDL_CreateWindow` retourne `NULL` ! La variable qui reçoit ce pointeur (ex: `SDL_Window *maFenetre = NULL;`) est votre "poignée" pour interagir avec la fenêtre.

Création d'un Rendu (`SDL_CreateRenderer`)

La fenêtre n'est qu'un conteneur. Pour dessiner efficacement dans cette fenêtre (en utilisant l'accélération matérielle si possible), vous avez besoin d'un **Rendu** (Renderer). Le rendu est une abstraction pour les

opérations de dessin 2D.

Syntaxe : `SDL_Renderer* SDL_CreateRenderer(SDL_Window *window, int index, Uint32 flags);`

- `window`: Le pointeur `SDL_Window*` vers la fenêtre dans laquelle vous voulez dessiner.
- `index`: L'index du "driver" de rendu à utiliser. Mettez `-1` pour laisser SDL choisir le premier qui supporte les `flags` demandés (c'est le choix habituel).
- `flags`: Des indicateurs (`Uint32`) pour spécifier les capacités du rendu, combinés avec `|`. Flags courants :
  - `SDL_RENDERER_ACCELERATED`: Utilise l'accélération matérielle (GPU). C'est ce qu'on veut généralement.
  - `SDL_RENDERER_PRESENTVSYNC`: Active la synchronisation verticale (V-Sync). Tente de synchroniser le rafraîchissement de l'affichage avec celui de l'écran pour éviter le "tearing" (déchirement d'image), mais peut limiter le framerate et introduire un léger délai.
  - `SDL_RENDERER_SOFTWARE`: Force l'utilisation d'un rendu logiciel (CPU), beaucoup plus lent. À n'utiliser qu'en dernier recours ou pour le débogage.
- **Valeur de retour :** Un pointeur vers la structure `SDL_Renderer` créée (`SDL_Renderer*`) en cas de succès, ou `NULL` en cas d'erreur.

Vérifiez toujours si `SDL_CreateRenderer` retourne `NULL` ! La variable qui reçoit ce pointeur (ex: `SDL_Renderer *monRendu = NULL;`) est votre "poignée" pour toutes les opérations de dessin.

Gestion des erreurs SDL (`SDL_GetError`)

Lorsque les fonctions SDL (`SDL_Init`, `SDL_CreateWindow`, `SDL_CreateRenderer`, et beaucoup d'autres) échouent (retournent une valeur négative ou `NULL`), elles enregistrent souvent un message d'erreur plus détaillé. Vous pouvez récupérer ce message avec `SDL_GetError()`.

Syntaxe : `const char* SDL_GetError(void);`

- Retourne une chaîne de caractères (`const char*`) décrivant la dernière erreur SDL qui s'est produite pour le thread courant.
- Il est bon d'afficher ce message (par exemple avec `fprintf(stderr, ...)` ou `SDL_Log`) juste après avoir détecté un échec pour comprendre pourquoi une fonction SDL n'a pas fonctionné.

Fermeture propre (`SDL_Destroy...`, `SDL_Quit`)

Tout comme il faut fermer les fichiers avec `fclose`, il est **essentiel** de libérer les ressources allouées par SDL lorsque votre programme se termine ou n'en a plus besoin. L'ordre est généralement l'inverse de la création :

1. **Détruire le Rendu :** `SDL_DestroyRenderer(SDL_Renderer *renderer);`
2. **Détruire la Fenêtre :** `SDL_DestroyWindow(SDL_Window *window);`
3. **Quitter les sous-systèmes SDL :** `SDL_Quit();` (Cette fonction nettoie tous les sous-systèmes initialisés par `SDL_Init`).

Il faut aussi libérer explicitement les autres ressources SDL que vous auriez créées (Textures, Surfaces, Polices, etc.) avec leurs fonctions `SDL_Destroy...` ou `TTF_CloseFont`, etc., *avant* d'appeler

`SDL_Quit()`. Oublier de libérer ces ressources peut causer des fuites de mémoire ou laisser des ressources système occupées inutilement.

```c
#include <stdio.h>
#include <SDL2/SDL.h> // En-tête principal de SDL

// Constantes pour la fenêtre
const int LARGEUR_FENETRE = 640;
const int HAUTEUR_FENETRE = 480;
const char* TITRE_FENETRE = "Fenêtre SDL Simple";

int main(int argc, char* argv[]) { // Signature complète de main souvent requise par SDL

    SDL_Window* fenetre = NULL;     // Pointeur pour notre fenêtre
    SDL_Renderer* rendu = NULL;   // Pointeur pour notre rendu

    // --- 1. Initialisation de SDL (sous-système vidéo) ---
    printf("Initialisation de SDL...\n");
    // Initialiser uniquement le sous-système vidéo pour cet exemple
    if (SDL_Init(SDL_INIT_VIDEO) < 0) {
        // SDL_Init a retourné une valeur négative : erreur
        fprintf(stderr, "Erreur lors de l'initialisation de SDL : %s\n", SDL_GetError());
        return 1; // Quitter le programme avec un code d'erreur
    }
    printf("SDL initialisé avec succès.\n");

    // --- 2. Création de la fenêtre ---
    printf("Création de la fenêtre...\n");
    fenetre = SDL_CreateWindow(
        TITRE_FENETRE,                  // Titre de la fenêtre (UTF-8)
        SDL_WINDOWPOS_CENTERED,         // Position x initiale (centrée)
        SDL_WINDOWPOS_CENTERED,         // Position y initiale (centrée)
        LARGEUR_FENETRE,                // Largeur de la fenêtre en pixels
        HAUTEUR_FENETRE,                // Hauteur de la fenêtre en pixels
        SDL_WINDOW_SHOWN                // Flags : fenêtre visible dès sa création
    );

    // Vérification de la création de la fenêtre
    if (fenetre == NULL) {
        fprintf(stderr, "Erreur lors de la création de la fenêtre : %s\n", SDL_GetError());
        SDL_Quit(); // Nettoyer SDL avant de quitter
        return 1;
    }
    printf("Fenêtre créée avec succès.\n");

    // --- 3. Création du Rendu ---
    printf("Création du rendu...\n");
    rendu = SDL_CreateRenderer(
```

```
        fenetre,                          // Fenêtre cible
        -1,                               // Index du driver (-1 pour le
premier compatible)
        SDL_RENDERER_ACCELERATED | SDL_RENDERER_PRESENTVSYNC // Flags:
Accélération matérielle + V-Sync
    );

    // Vérification de la création du rendu
    if (rendu == NULL) {
        fprintf(stderr, "Erreur lors de la création du rendu : %s\n",
SDL_GetError());
        SDL_DestroyWindow(fenetre); // Nettoyer la fenêtre avant de quitter
SDL
        SDL_Quit();
        return 1;
    }
    printf("Rendu créé avec succès.\n");

    // --- Le programme principal irait ici ---
    // (Boucle d'événements, dessin, etc.)
    printf("\nFenêtre et rendu prêts. Attente de 3 secondes...\n");
    SDL_SetRenderDrawColor(rendu, 0, 0, 50, 255); // Couleur de fond bleu
foncé
    SDL_RenderClear(rendu); // Efface l'écran avec la couleur choisie
    SDL_RenderPresent(rendu); // Affiche le contenu du rendu à l'écran
    SDL_Delay(3000); // Pause de 3000 millisecondes (3 secondes)

    // --- 4. Nettoyage et Fermeture ---
    printf("\nFermeture de SDL...\n");

    // Détruire le rendu d'abord
    if (rendu) {
        SDL_DestroyRenderer(rendu);
        rendu = NULL; // Bonne pratique
        printf("Rendu détruit.\n");
    }

    // Détruire la fenêtre ensuite
    if (fenetre) {
        SDL_DestroyWindow(fenetre);
        fenetre = NULL; // Bonne pratique
        printf("Fenêtre détruite.\n");
    }

    // Quitter tous les sous-systèmes SDL initialisés
    SDL_Quit();
    printf("SDL quitté.\n");

    return 0; // Succès
}

/* Pour compiler (exemple Linux/macOS avec GCC/Clang):
   gcc nom_fichier.c -o programme -lSDL2
   (ou utiliser pkg-config/sdl2-config comme vu précédemment)
```

```
*/
```

Cette structure (Init -> Create Window -> Create Renderer -> Boucle Principale -> Destroy Renderer -> Destroy Window -> Quit) est la base de la plupart des applications SDL2. La gestion rigoureuse des erreurs (vérification des retours `NULL` ou négatifs et affichage de `SDL_GetError()`) et le nettoyage systématique des ressources sont essentiels pour créer des applications stables.

---

23. Affichage et Images

Maintenant que nous savons initialiser SDL, créer une fenêtre et un rendu (`SDL_Renderer`), nous pouvons commencer à dessiner des choses dans cette fenêtre. Le `SDL_Renderer` est notre principal outil pour toutes les opérations de dessin 2D accélérées matériellement.

Le Rendu (Renderer) : Le Cycle de Dessin

Le dessin avec SDL2 suit généralement un cycle simple à l'intérieur de votre boucle principale (que nous verrons dans la section sur les événements) :

1. **Effacer l'écran :** Préparer le rendu pour le nouveau frame en effaçant le contenu précédent avec une couleur de fond.
2. **Dessiner les éléments :** Utiliser les fonctions du rendu pour dessiner des formes (lignes, rectangles), des images (textures), du texte, etc., dans l'ordre souhaité (ce qui est dessiné en dernier apparaît au-dessus).
3. **Présenter le rendu :** Afficher le résultat final dessiné à l'écran.

Fonctions clés du cycle :

- **`int SDL_SetRenderDrawColor(SDL_Renderer *renderer, Uint8 r, Uint8 g, Uint8 b, Uint8 a);`**
    - Définit la couleur qui sera utilisée pour les opérations de dessin suivantes (comme effacer l'écran ou dessiner des formes géométriques).
    - `renderer`: Le rendu à utiliser.
    - `r`, `g`, `b`: Composantes Rouge, Vert, Bleu de la couleur (valeurs de 0 à 255).
    - `a`: Composante Alpha (opacité) de la couleur (0 = totalement transparent, 255 = totalement opaque).
    - Retourne 0 en cas de succès, une valeur négative en cas d'erreur.
- **`int SDL_RenderClear(SDL_Renderer *renderer);`**
    - Remplit toute la cible de rendu actuelle (généralement la fenêtre entière) avec la couleur définie par le dernier appel à `SDL_SetRenderDrawColor`. C'est la première étape typique de la boucle de dessin pour effacer l'image précédente.
    - Retourne 0 en cas de succès, une valeur négative en cas d'erreur.
- **`void SDL_RenderPresent(SDL_Renderer *renderer);`**

- Met à jour l'écran avec tout ce qui a été dessiné sur le rendu depuis le dernier `SDL_RenderClear`. C'est cette fonction qui rend vos dessins visibles pour l'utilisateur. Si vous utilisez le flag `SDL_RENDERER_PRESENTVSYNC` lors de la création du rendu, cet appel peut bloquer l'exécution jusqu'à la prochaine synchronisation verticale de l'écran pour éviter le déchirement de l'image (tearing).

Couleurs (`SDL_Color`)

SDL représente souvent les couleurs RGBA (Rouge, Vert, Bleu, Alpha) à l'aide de la structure `SDL_Color`, définie dans `SDL_pixels.h` (qui est inclus par `SDL.h`). Sa définition ressemble à ceci (conceptuellement) :

Une structure nommée `SDL_Color` contenant quatre membres de type `Uint8` (entier non signé 8 bits) : `r` pour le rouge, `g` pour le vert, `b` pour le bleu, et `a` pour l'alpha (opacité), chacun allant de 0 à 255.

Vous pouvez créer des variables de ce type pour stocker et réutiliser des couleurs facilement. Par exemple : `SDL_Color couleur_rouge = {255, 0, 0, 255};` // Rouge opaque `SDL_Color couleur_bleu_transparent = {0, 0, 255, 128};` // Bleu semi-transparent

(Voir l'exemple de code `code_sdl_renderer_base_section23` pour une illustration du cycle de base).

```c
#include <stdio.h>
#include <SDL2/SDL.h>

const int LARGEUR_FENETRE = 800;
const int HAUTEUR_FENETRE = 600;

int main(int argc, char* argv[]) {
    SDL_Window* fenetre = NULL;
    SDL_Renderer* rendu = NULL;

    // Initialisation SDL
    if (SDL_Init(SDL_INIT_VIDEO) < 0) {
        fprintf(stderr, "Erreur SDL_Init : %s\n", SDL_GetError());
        return 1;
    }

    // Création Fenêtre
    fenetre = SDL_CreateWindow("Cycle de Rendu SDL",
                               SDL_WINDOWPOS_CENTERED,
SDL_WINDOWPOS_CENTERED,
                               LARGEUR_FENETRE, HAUTEUR_FENETRE,
                               SDL_WINDOW_SHOWN);
    if (fenetre == NULL) {
        fprintf(stderr, "Erreur SDL_CreateWindow : %s\n", SDL_GetError());
        SDL_Quit();
        return 1;
    }

    // Création Rendu
    rendu = SDL_CreateRenderer(fenetre, -1, SDL_RENDERER_ACCELERATED);
    if (rendu == NULL) {
        fprintf(stderr, "Erreur SDL_CreateRenderer : %s\n",
```

```
    SDL_GetError());
        SDL_DestroyWindow(fenetre);
        SDL_Quit();
        return 1;
    }

    // --- Boucle Principale (simplifiée ici pour 1 frame) ---
    printf("Préparation du dessin...\n");

    // 1. Définir la couleur de nettoyage (ex: bleu clair)
    SDL_SetRenderDrawColor(rendu, 135, 206, 250, 255); // R, G, B, A
(opaque)

    // 2. Effacer l'écran avec cette couleur
    SDL_RenderClear(rendu);

    // 3. (Ici on dessinerait d'autres choses...)

    // 4. Présenter le rendu (afficher le fond bleu clair)
    SDL_RenderPresent(rendu);
    printf("Écran effacé en bleu clair. Attente 3 secondes...\n");
    SDL_Delay(3000); // Pause pour voir le résultat

    // --- Nettoyage ---
    SDL_DestroyRenderer(rendu);
    SDL_DestroyWindow(fenetre);
    SDL_Quit();
    printf("SDL fermé.\n");

    return 0;
}

/* Compilation (Linux/macOS):
   gcc main.c -o rendu_simple -lSDL2
*/
```

Dessin de formes simples

Le `SDL_Renderer` fournit des fonctions pour dessiner des formes géométriques de base. La couleur utilisée est celle définie par le dernier appel à `SDL_SetRenderDrawColor()`.

- **`int SDL_RenderDrawPoint(SDL_Renderer *renderer, int x, int y);`**
  - Dessine un pixel unique aux coordonnées `(x, y)`.
- **`int SDL_RenderDrawLine(SDL_Renderer *renderer, int x1, int y1, int x2, int y2);`**
  - Dessine une ligne entre les points `(x1, y1)` et `(x2, y2)`.
- **`int SDL_RenderDrawRect(SDL_Renderer *renderer, const SDL_Rect *rect);`**
  - Dessine le contour d'un rectangle.
  - `rect`: Un pointeur vers une structure `SDL_Rect` définissant la position et la taille du rectangle (`{ int x, y, w, h; }`). Si `rect` est `NULL`, dessine le contour de toute la cible de rendu.

- `int SDL_RenderFillRect(SDL_Renderer *renderer, const SDL_Rect *rect);`
  - Dessine un rectangle **rempli** avec la couleur actuelle.
  - `rect`: Pointeur vers `SDL_Rect`. Si `NULL`, remplit toute la cible.

Il existe aussi des fonctions pour dessiner plusieurs points, lignes ou rectangles en une seule fois (`SDL_RenderDrawPoints`, `SDL_RenderDrawLines`, `SDL_RenderDrawRects`, `SDL_RenderFillRects`) pour de meilleures performances si vous avez beaucoup de formes à dessiner.

```c
#include <stdio.h>
#include <SDL2/SDL.h>

const int LARGEUR_FENETRE = 640;
const int HAUTEUR_FENETRE = 480;

int main(int argc, char* argv[]) {
    SDL_Window* fenetre = NULL;
    SDL_Renderer* rendu = NULL;

    if (SDL_Init(SDL_INIT_VIDEO) < 0) { /* ... gestion erreur ... */ return
1; }

    fenetre = SDL_CreateWindow("Dessin de Formes", SDL_WINDOWPOS_CENTERED,
SDL_WINDOWPOS_CENTERED,
                               LARGEUR_FENETRE, HAUTEUR_FENETRE,
SDL_WINDOW_SHOWN);
    if (!fenetre) { /* ... gestion erreur ... */ SDL_Quit(); return 1; }

    rendu = SDL_CreateRenderer(fenetre, -1, SDL_RENDERER_ACCELERATED);
    if (!rendu) { /* ... gestion erreur ... */ SDL_DestroyWindow(fenetre);
SDL_Quit(); return 1; }

    // --- Boucle de dessin (1 frame) ---

    // 1. Effacer l'écran (fond noir)
    SDL_SetRenderDrawColor(rendu, 0, 0, 0, 255); // Noir opaque
    SDL_RenderClear(rendu);

    // 2. Dessiner des formes

    // Un rectangle rouge rempli
    SDL_Rect rect_rouge = { 50, 50, 200, 100 }; // x, y, largeur, hauteur
    SDL_SetRenderDrawColor(rendu, 255, 0, 0, 255); // Rouge
    SDL_RenderFillRect(rendu, &rect_rouge);

    // Le contour d'un rectangle vert
    SDL_Rect rect_vert_contour = { 300, 100, 150, 200 };
    SDL_SetRenderDrawColor(rendu, 0, 255, 0, 255); // Vert
    SDL_RenderDrawRect(rendu, &rect_vert_contour);

    // Une ligne bleue diagonale
    SDL_SetRenderDrawColor(rendu, 0, 0, 255, 255); // Bleu
    SDL_RenderDrawLine(rendu, 50, 300, 550, 400); // (x1, y1) -> (x2, y2)
```

```
    // Quelques points blancs
    SDL_SetRenderDrawColor(rendu, 255, 255, 255, 255); // Blanc
    SDL_RenderDrawPoint(rendu, 10, 10);
    SDL_RenderDrawPoint(rendu, 11, 10);
    SDL_RenderDrawPoint(rendu, 10, 11);
    SDL_RenderDrawPoint(rendu, 11, 11);

    // 3. Présenter le rendu
    SDL_RenderPresent(rendu);

    printf("Formes dessinées. Attente 5 secondes...\n");
    SDL_Delay(5000);

    // --- Nettoyage ---
    SDL_DestroyRenderer(rendu);
    SDL_DestroyWindow(fenetre);
    SDL_Quit();

    return 0;
}

/* Compilation (Linux/macOS):
   gcc main.c -o formes_simples -lSDL2
*/
```

Chargement d'images avec `SDL_image`

Pour afficher des images (sprites, fonds, etc.), nous utilisons généralement la bibliothèque satellite `SDL_image`. Elle étend les capacités de SDL pour charger différents formats de fichiers image (PNG, JPG, etc.).

Initialisation de `SDL_image` : Avant de charger des images, vous devez initialiser `SDL_image` pour les formats que vous comptez utiliser.

Syntaxe : `int IMG_Init(int flags);`

- `flags`: Indicateurs pour les formats à supporter, combinés avec `|`.
  - `IMG_INIT_PNG`
  - `IMG_INIT_JPG`
  - `IMG_INIT_TIF`
  - `IMG_INIT_WEBP`
- **Valeur de retour :** Un masque de bits des initialiseurs qui ont réussi. Vous devez vérifier si les flags que vous avez demandés sont bien présents dans la valeur retournée.

Chargement d'une image (`IMG_Load`) : Cette fonction charge une image depuis un fichier et la retourne sous forme de `SDL_Surface`.

Syntaxe : `SDL_Surface* IMG_Load(const char *file);`

- `file`: Chemin vers le fichier image.
- **Valeur de retour :** Un pointeur vers une `SDL_Surface` contenant les données de l'image en mémoire CPU, ou `NULL` en cas d'erreur (fichier non trouvé, format non supporté, etc.). Utilisez `IMG_GetError()` pour plus de détails en cas d'échec.

Nettoyage de `SDL_image` : Lorsque vous n'avez plus besoin de charger d'images, appelez `IMG_Quit();` (généralement avant `SDL_Quit()`).

Surfaces (`SDL_Surface`) et Textures (`SDL_Texture`)

SDL2 utilise deux concepts principaux pour manipuler les données graphiques :

- **`SDL_Surface` :** Représente une image en **mémoire CPU (RAM)**. C'est une collection de pixels bruts avec des informations sur le format (couleur, profondeur), la largeur, la hauteur, etc. `IMG_Load` retourne une `SDL_Surface`. Les opérations sur les surfaces (accès aux pixels, modifications) se font via le CPU et peuvent être lentes.
- **`SDL_Texture` :** Représente une image optimisée pour le **rendu par le GPU (mémoire vidéo)**. C'est une version de l'image préparée pour être dessinée très rapidement par le `SDL_Renderer` (qui utilise souvent l'accélération matérielle). On ne peut généralement pas modifier directement les pixels d'une texture de manière efficace.

Le flux de travail typique est :

1. Charger une image depuis un fichier dans une `SDL_Surface` (avec `IMG_Load`).
2. Créer une `SDL_Texture` optimisée pour le rendu à partir de cette `SDL_Surface` (avec `SDL_CreateTextureFromSurface`).
3. Libérer la `SDL_Surface` originale (on n'en a plus besoin, la texture contient les données nécessaires pour le GPU).
4. Utiliser la `SDL_Texture` pour dessiner l'image à l'écran (avec `SDL_RenderCopy`).
5. Libérer la `SDL_Texture` quand on n'en a plus besoin (avec `SDL_DestroyTexture`).

Création d'une texture depuis une surface :

Syntaxe : `SDL_Texture* SDL_CreateTextureFromSurface(SDL_Renderer *renderer, SDL_Surface *surface);`

- `renderer`: Le rendu qui sera utilisé pour afficher cette texture.
- `surface`: La surface (chargée avec `IMG_Load` par exemple) contenant les données de l'image.
- **Valeur de retour :** Un pointeur `SDL_Texture*` vers la texture créée, ou `NULL` en cas d'erreur.

Libération des ressources :

- `void SDL_FreeSurface(SDL_Surface *surface);` : Libère la mémoire utilisée par une surface.
- `void SDL_DestroyTexture(SDL_Texture *texture);` : Libère la mémoire (souvent vidéo) utilisée par une texture.

```
#include <stdio.h>
#include <SDL2/SDL.h>
#include <SDL2/SDL_image.h> // Inclure l'en-tête SDL_image
```

```
const int LARGEUR_FENETRE = 640;
const int HAUTEUR_FENETRE = 480;
const char* CHEMIN_IMAGE = "chemin/vers/votre/image.png"; // REMPLACEZ CECI
!

// Fonction utilitaire pour charger une texture (gère surface et texture)
SDL_Texture* charger_texture(const char* chemin, SDL_Renderer* rendu) {
    SDL_Texture* nouvelle_texture = NULL;
    SDL_Surface* surface_chargee = IMG_Load(chemin); // Charge l'image en
surface

    if (surface_chargee == NULL) {
        fprintf(stderr, "Impossible de charger l'image %s! Erreur
SDL_image: %s\n",
                chemin, IMG_GetError());
    } else {
        // Crée une texture optimisée à partir de la surface
        nouvelle_texture = SDL_CreateTextureFromSurface(rendu,
surface_chargee);
        if (nouvelle_texture == NULL) {
            fprintf(stderr, "Impossible de créer la texture depuis %s!
Erreur SDL: %s\n",
                    chemin, SDL_GetError());
        }
        // Libère l'ancienne surface (on n'en a plus besoin)
        SDL_FreeSurface(surface_chargee);
    }
    return nouvelle_texture;
}


int main(int argc, char* argv[]) {
    SDL_Window* fenetre = NULL;
    SDL_Renderer* rendu = NULL;
    SDL_Texture* texture_image = NULL; // Pointeur pour notre texture

    // Initialisation SDL Vidéo
    if (SDL_Init(SDL_INIT_VIDEO) < 0) { /* ... gestion erreur ... */ return
1; }

    // Initialisation SDL_image (pour PNG ici)
    int imgFlags = IMG_INIT_PNG;
    if (!(IMG_Init(imgFlags) & imgFlags)) {
        fprintf(stderr, "Erreur IMG_Init: %s\n", IMG_GetError());
        SDL_Quit();
        return 1;
    }
    printf("SDL_image initialisé pour PNG.\n");

    // Création Fenêtre
    fenetre = SDL_CreateWindow("Chargement Image", SDL_WINDOWPOS_CENTERED,
SDL_WINDOWPOS_CENTERED,
                               LARGEUR_FENETRE, HAUTEUR_FENETRE,
SDL_WINDOW_SHOWN);
```

```
    if (!fenetre) { /* ... gestion erreur ... */ IMG_Quit(); SDL_Quit();
return 1; }

    // Création Rendu
    rendu = SDL_CreateRenderer(fenetre, -1, SDL_RENDERER_ACCELERATED);
    if (!rendu) { /* ... gestion erreur ... */ SDL_DestroyWindow(fenetre);
IMG_Quit(); SDL_Quit(); return 1; }

    // --- Chargement de l'image en texture ---
    printf("Chargement de l'image '%s'...\n", CHEMIN_IMAGE);
    texture_image = charger_texture(CHEMIN_IMAGE, rendu);

    if (texture_image == NULL) {
        fprintf(stderr, "Échec du chargement de la texture.\n");
        // Nettoyage et sortie en cas d'échec
        SDL_DestroyRenderer(rendu);
        SDL_DestroyWindow(fenetre);
        IMG_Quit();
        SDL_Quit();
        return 1;
    }
    printf("Texture chargée avec succès.\n");

    // --- Ici on utiliserait la texture dans la boucle principale ---
    // (Affichage simple pour l'exemple)
    SDL_SetRenderDrawColor(rendu, 200, 200, 200, 255); // Fond gris clair
    SDL_RenderClear(rendu);
    // Afficher la texture (sera vu dans la partie suivante)
    // SDL_RenderCopy(rendu, texture_image, NULL, NULL); // Dessine sur
tout l'écran
    SDL_RenderPresent(rendu);
    printf("Texture chargée (mais pas encore affichée ici). Attente 3
secondes...\n");
    SDL_Delay(3000);


    // --- Nettoyage ---
    printf("\nFermeture...\n");
    if (texture_image) {
        SDL_DestroyTexture(texture_image); // Libérer la texture !
        texture_image = NULL;
        printf("Texture détruite.\n");
    }
    if (rendu) { SDL_DestroyRenderer(rendu); printf("Rendu détruit.\n"); }
    if (fenetre) { SDL_DestroyWindow(fenetre); printf("Fenêtre
détruite.\n"); }
    IMG_Quit(); // Quitter SDL_image
    printf("SDL_image quitté.\n");
    SDL_Quit(); // Quitter SDL
    printf("SDL quitté.\n");

    return 0;
}
```

```
/* Compilation (Linux/macOS):
   gcc main.c -o charge_image -lSDL2 -lSDL2_image
   (Assurez-vous que le fichier image existe au chemin spécifié)
*/
```

Affichage de textures (`SDL_RenderCopy`, `SDL_Rect`)

Une fois que vous avez une `SDL_Texture` (créée à partir d'une surface ou d'une autre manière), la fonction principale pour la dessiner à l'écran via le rendu est `SDL_RenderCopy()`.

Syntaxe : `int SDL_RenderCopy(SDL_Renderer *renderer, SDL_Texture *texture, const SDL_Rect *srcrect, const SDL_Rect *dstrect);`

- `renderer`: Le rendu sur lequel dessiner.
- `texture`: La texture à dessiner.
- `srcrect` (Source Rectangle) : Un pointeur vers un `SDL_Rect` qui définit quelle **portion** de la texture source utiliser. Si `NULL`, toute la texture est utilisée. Utile pour les feuilles de sprites (sprite sheets) où plusieurs images sont sur la même texture.
- `dstrect` (Destination Rectangle) : Un pointeur vers un `SDL_Rect` qui définit où et à quelle **taille** dessiner la texture (ou la portion source) dans la fenêtre (la cible de rendu). Si `NULL`, la texture (ou la portion source) est étirée pour remplir toute la cible de rendu.
- **Valeur de retour :** 0 en cas de succès, négatif en cas d'erreur.

La structure `SDL_Rect` est simple et définit une zone rectangulaire :

`typedef struct SDL_Rect { int x, y; // Coordonnées du coin supérieur gauche int w, h; // Largeur et hauteur } SDL_Rect;`

Utilisation courante :

- Pour afficher une texture entière à une position `(x, y)` avec sa taille d'origine :
  1. Obtenir la largeur `w` et la hauteur `h` de la texture avec `SDL_QueryTexture(texture, NULL, NULL, &w, &h);`.
  2. Créer un `SDL_Rect` de destination : `SDL_Rect dest = { x, y, w, h };`
  3. Appeler `SDL_RenderCopy(rendu, texture, NULL, &dest);` (`NULL` pour `srcrect` pour utiliser toute la texture).
- Pour afficher une partie d'une texture (sprite) :
  1. Définir un `SDL_Rect` source `src` qui délimite le sprite dans la texture.
  2. Définir un `SDL_Rect` destination `dest` où afficher le sprite à l'écran.
  3. Appeler `SDL_RenderCopy(rendu, texture, &src, &dest);`.

Obtenir les dimensions d'une texture : La fonction `SDL_QueryTexture` permet de récupérer diverses informations sur une texture, notamment ses dimensions.

`int SDL_QueryTexture(SDL_Texture *texture, Uint32 *format, int *access, int *w, int *h);`

- Permet de récupérer le format des pixels (`format`), le mode d'accès (`access`, ex: `SDL_TEXTUREACCESS_STATIC`), la largeur (`w`) et la hauteur (`h`).

- Passez `NULL` pour les informations qui ne vous intéressent pas.
- Pour obtenir juste la largeur et la hauteur : `SDL_QueryTexture(texture, NULL, NULL, &largeur, &hauteur);`

N'oubliez pas d'appeler `SDL_RenderPresent(rendu)` *après* tous vos appels `SDL_RenderCopy` (et autres dessins du frame) pour que le résultat s'affiche réellement à l'écran.

(Voir l'exemple de code `code_sdl_render_texture_section23` pour une illustration pratique).

```
#include <stdio.h>
#include <SDL2/SDL.h>
#include <SDL2/SDL_image.h>

const int LARGEUR_FENETRE = 800;
const int HAUTEUR_FENETRE = 600;
const char* CHEMIN_IMAGE = "assets/blue_pawn.png"; // Assurez-vous que ce
fichier existe

// Fonction utilitaire (identique à l'exemple précédent)
SDL_Texture* charger_texture(const char* chemin, SDL_Renderer* rendu) {
    SDL_Texture* nouvelle_texture = NULL;
    SDL_Surface* surface_chargee = IMG_Load(chemin);
    if (!surface_chargee) {
        fprintf(stderr, "IMG_Load Error: %s\n", IMG_GetError());
        return NULL;
    }
    nouvelle_texture = SDL_CreateTextureFromSurface(rendu,
surface_chargee);
    if (!nouvelle_texture) {
        fprintf(stderr, "SDL_CreateTextureFromSurface Error: %s\n",
SDL_GetError());
    }
    SDL_FreeSurface(surface_chargee); // Libère la surface après création
de texture
    return nouvelle_texture;
}

int main(int argc, char* argv[]) {
    SDL_Window* fenetre = NULL;
    SDL_Renderer* rendu = NULL;
    SDL_Texture* texture_pion = NULL;

    if (SDL_Init(SDL_INIT_VIDEO) < 0) { /* ... gestion erreur ... */ return
1; }
    if (!(IMG_Init(IMG_INIT_PNG) & IMG_INIT_PNG)) { /* ... gestion erreur
... */ SDL_Quit(); return 1; }

    fenetre = SDL_CreateWindow("Affichage Texture", SDL_WINDOWPOS_CENTERED,
SDL_WINDOWPOS_CENTERED,
                               LARGEUR_FENETRE, HAUTEUR_FENETRE,
SDL_WINDOW_SHOWN);
    if (!fenetre) { /* ... gestion erreur ... */ IMG_Quit(); SDL_Quit();
```

```
return 1; }

    rendu = SDL_CreateRenderer(fenetre, -1, SDL_RENDERER_ACCELERATED |
SDL_RENDERER_PRESENTVSYNC);
    if (!rendu) { /* ... gestion erreur ... */ SDL_DestroyWindow(fenetre);
IMG_Quit(); SDL_Quit(); return 1; }

    // Charger la texture du pion
    texture_pion = charger_texture(CHEMIN_IMAGE, rendu);
    if (texture_pion == NULL) {
        // Nettoyage et sortie si le chargement échoue
        SDL_DestroyRenderer(rendu);
        SDL_DestroyWindow(fenetre);
        IMG_Quit();
        SDL_Quit();
        return 1;
    }

    // Obtenir les dimensions de la texture chargée
    int largeur_pion, hauteur_pion;
    if (SDL_QueryTexture(texture_pion, NULL, NULL, &largeur_pion,
&hauteur_pion) != 0) {
        fprintf(stderr, "Erreur SDL_QueryTexture: %s\n", SDL_GetError());
        // Gérer l'erreur... ici on continue avec des tailles par défaut ?
        largeur_pion = 50; hauteur_pion = 50;
    } else {
         printf("Texture chargée : %d x %d pixels\n", largeur_pion,
hauteur_pion);
    }


    // --- Boucle Principale Simplifiée (juste pour l'affichage) ---
    bool continuer = true;
    SDL_Event event;
    int pion_x = 100;
    int pion_y = 100;

    printf("Affichage du pion. Fermez la fenêtre pour quitter.\n");

    while (continuer) {
        // Gestion des événements (juste pour quitter ici)
        while (SDL_PollEvent(&event)) {
            if (event.type == SDL_QUIT) {
                continuer = false;
            }
        }

        // 1. Effacer l'écran (fond blanc)
        SDL_SetRenderDrawColor(rendu, 255, 255, 255, 255);
        SDL_RenderClear(rendu);

        // 2. Préparer le rectangle de destination pour le pion
        SDL_Rect dest_rect = { pion_x, pion_y, largeur_pion, hauteur_pion
};
```

```
        // 3. Copier la texture dans le rendu à la position/taille voulue
        //    Le premier NULL signifie qu'on utilise toute la texture
source.
        SDL_RenderCopy(rendu, texture_pion, NULL, &dest_rect);

        // 4. Présenter le rendu à l'écran
        SDL_RenderPresent(rendu);

        // Petite pause pour ne pas surcharger le CPU
        SDL_Delay(16); // Environ 60 FPS
    }


    // --- Nettoyage ---
    SDL_DestroyTexture(texture_pion);
    SDL_DestroyRenderer(rendu);
    SDL_DestroyWindow(fenetre);
    IMG_Quit();
    SDL_Quit();

    return 0;
}
```

Transparence et Modes de fusion (`SDL_SetTextureBlendMode`, `SDL_SetTextureAlphaMod`)

SDL2 permet de gérer la transparence et de mélanger les couleurs lors du dessin de textures.

- **Transparence des PNG :** Si vous chargez une image PNG avec des zones transparentes ou semi-transparentes (canal alpha), `SDL_image` et `SDL_CreateTextureFromSurface` préservent généralement cette information. Pour que la transparence soit prise en compte lors du dessin avec `SDL_RenderCopy`, vous devez activer le **mode de fusion** (blend mode) pour la texture :
    - `SDL_SetTextureBlendMode(texture, SDL_BLENDMODE_BLEND);`
        - `SDL_BLENDMODE_NONE` (défaut) : La texture écrase ce qui se trouve derrière elle.
        - `SDL_BLENDMODE_BLEND` : Active le mélange alpha standard (la couleur de la texture est mélangée avec la couleur du fond en fonction de l'alpha de la texture).
        - `SDL_BLENDMODE_ADD` : Ajoute les valeurs de couleur (utile pour des effets de lumière).
        - `SDL_BLENDMODE_MOD` : Multiplie les valeurs de couleur (utile pour teinter).
- **Transparence globale (Modulation Alpha) :** Vous pouvez rendre une texture entière plus ou moins transparente, indépendamment de son canal alpha d'origine, en modifiant sa **modulation alpha**.
    - `SDL_SetTextureAlphaMod(texture, Uint8 alpha);`
        - `alpha`: Valeur de 0 (totalement transparent) à 255 (totalement opaque). La valeur alpha de chaque pixel de la texture sera multipliée par `alpha / 255.0` lors du rendu (si le blend mode est activé).
- **Modulation de couleur :** Vous pouvez aussi teinter une texture entière.

    - `SDL_SetTextureColorMod(texture, Uint8 r, Uint8 g, Uint8 b);`
        - Multiplie les composantes R, G, B de chaque pixel par `r/255.0`, `g/255.0`, `b/255.0` respectivement. Mettre 255 pour toutes les composantes ne change pas la couleur. Mettre (255, 0, 0) teintera l'image en rouge.

Pour que la modulation alpha ou couleur fonctionne, le blend mode de la texture doit généralement être activé (`SDL_BLENDMODE_BLEND` ou autre).

```c
#include <stdio.h>
#include <SDL2/SDL.h>
#include <SDL2/SDL_image.h>

// Assurez-vous d'avoir une image PNG avec transparence, par exemple
'fantome.png'
const char* CHEMIN_FANTOME = "assets/blue_pawn.png"; // REMPLACEZ par votre
image PNG transparente
const char* CHEMIN_FOND = "assets/board.png"; // Une image de fond

const int LARGEUR_FENETRE = 800;
const int HAUTEUR_FENETRE = 600;

SDL_Texture* charger_texture(const char* chemin, SDL_Renderer* rendu) {
    SDL_Texture* nouvelle_texture = NULL;
    SDL_Surface* surface_chargee = IMG_Load(chemin);
    if (!surface_chargee) { fprintf(stderr, "IMG_Load Error: %s\n",
IMG_GetError()); return NULL; }
    nouvelle_texture = SDL_CreateTextureFromSurface(rendu,
surface_chargee);
    if (!nouvelle_texture) { fprintf(stderr, "SDL_CreateTextureFromSurface
Error: %s\n", SDL_GetError()); }
    SDL_FreeSurface(surface_chargee);
    return nouvelle_texture;
}

int main(int argc, char* argv[]) {
    SDL_Window* fenetre = NULL;
    SDL_Renderer* rendu = NULL;
    SDL_Texture* tex_fond = NULL;
    SDL_Texture* tex_fantome = NULL;

    if (SDL_Init(SDL_INIT_VIDEO) < 0) { /*...*/ return 1; }
    if (!(IMG_Init(IMG_INIT_PNG) & IMG_INIT_PNG)) { /*...*/ SDL_Quit();
return 1; }

    fenetre = SDL_CreateWindow("Transparence SDL", SDL_WINDOWPOS_CENTERED,
SDL_WINDOWPOS_CENTERED,
                               LARGEUR_FENETRE, HAUTEUR_FENETRE,
SDL_WINDOW_SHOWN);
    if (!fenetre) { /*...*/ IMG_Quit(); SDL_Quit(); return 1; }

    rendu = SDL_CreateRenderer(fenetre, -1, SDL_RENDERER_ACCELERATED |
SDL_RENDERER_PRESENTVSYNC);
```

```
    if (!rendu) { /*...*/ SDL_DestroyWindow(fenetre); IMG_Quit();
SDL_Quit(); return 1; }

    // Charger les textures
    tex_fond = charger_texture(CHEMIN_FOND, rendu);
    tex_fantome = charger_texture(CHEMIN_FANTOME, rendu);

    if (!tex_fond || !tex_fantome) {
        // Nettoyage si une texture n'a pas pu être chargée
        SDL_DestroyTexture(tex_fond);
        SDL_DestroyTexture(tex_fantome);
        SDL_DestroyRenderer(rendu);
        SDL_DestroyWindow(fenetre);
        IMG_Quit();
        SDL_Quit();
        return 1;
    }

    // --- Activer le Blend Mode pour la texture du fantôme ---
    if (SDL_SetTextureBlendMode(tex_fantome, SDL_BLENDMODE_BLEND) != 0) {
         fprintf(stderr, "Erreur SetTextureBlendMode: %s\n",
SDL_GetError());
         // Continuer quand même pour l'exemple
    }

    // Obtenir la taille du fantôme
    int w_fantome, h_fantome;
    SDL_QueryTexture(tex_fantome, NULL, NULL, &w_fantome, &h_fantome);

    // --- Boucle Principale ---
    bool continuer = true;
    SDL_Event event;
    Uint8 alpha_fantome = 255; // Commence opaque
    int direction_alpha = -5; // Diminue l'alpha

    while (continuer) {
        while (SDL_PollEvent(&event)) {
            if (event.type == SDL_QUIT) {
                continuer = false;
            }
        }

        // Modifier l'alpha pour l'effet de fondu
        alpha_fantome += direction_alpha;
        if (alpha_fantome <= 50 || alpha_fantome == 255) {
            direction_alpha *= -1; // Inverser la direction
        }
        // Appliquer la modulation alpha à la texture
        SDL_SetTextureAlphaMod(tex_fantome, alpha_fantome);

        // 1. Effacer (ici, on dessine le fond à la place)
        SDL_RenderCopy(rendu, tex_fond, NULL, NULL); // Dessine le fond sur
tout l'écran
```

```
        // 2. Dessiner le fantôme (semi-)transparent
        SDL_Rect dest_fantome = { 100, 100, w_fantome, h_fantome };
        SDL_RenderCopy(rendu, tex_fantome, NULL, &dest_fantome);

        // Dessiner un autre fantôme teinté en rouge et fixe
        SDL_SetTextureColorMod(tex_fantome, 255, 100, 100); // Teinte rouge
(garde un peu de vert/bleu)
        SDL_SetTextureAlphaMod(tex_fantome, 200); // Légèrement transparent
        dest_fantome.x = 300;
        dest_fantome.y = 200;
        SDL_RenderCopy(rendu, tex_fantome, NULL, &dest_fantome);
        // Remettre les modulations par défaut pour le prochain tour de
boucle
        SDL_SetTextureColorMod(tex_fantome, 255, 255, 255);


        // 3. Présenter
        SDL_RenderPresent(rendu);

        SDL_Delay(30); // Petite pause
    }

    // --- Nettoyage ---
    SDL_DestroyTexture(tex_fantome);
    SDL_DestroyTexture(tex_fond);
    SDL_DestroyRenderer(rendu);
    SDL_DestroyWindow(fenetre);
    IMG_Quit();
    SDL_Quit();

    return 0;
}

/* Compilation (Linux/macOS):
   gcc main.c -o transparence -lSDL2 -lSDL2_image
   (Assurez-vous d'avoir les images aux chemins spécifiés)
*/
```

L'affichage d'images et la gestion de la transparence sont fondamentaux pour créer des interfaces graphiques et des jeux. SDL2, avec l'aide de `SDL_image`, fournit les outils nécessaires pour charger des images, les convertir en textures optimisées pour le GPU, et les afficher à l'écran en utilisant le `SDL_Renderer`, tout en permettant des effets de transparence et de modulation de couleur. N'oubliez jamais de libérer les ressources (`SDL_Surface`, `SDL_Texture`) lorsque vous n'en avez plus besoin.

---

24. Gestion des Événements

Une application graphique ou un jeu doit pouvoir réagir aux actions de l'utilisateur (clavier, souris, fermeture de fenêtre, etc.) et aux événements du système. SDL2 fournit un système de gestion des événements puissant et unifié pour gérer ces interactions.

La Boucle d'Événements (Event Loop)

Le cœur de la plupart des applications SDL2 est la **boucle d'événements** (ou boucle principale). C'est une boucle `while` qui s'exécute continuellement tant que l'utilisateur n'a pas demandé de quitter. À chaque tour de boucle, le programme :

1. **Vérifie** s'il y a des événements en attente (actions utilisateur, événements système).
2. **Traite** chaque événement reçu de manière appropriée.
3. (Généralement) **Met à jour** l'état de l'application ou du jeu en fonction des événements traités.
4. **Dessine** le nouvel état à l'écran (en utilisant le cycle de rendu vu précédemment).

Voici un pseudo-code illustrant la structure typique :

```
bool continuer = true;
while (continuer) {
    // 1. Gérer les événements
    // ...
    // 2. Mettre à jour l'état du jeu/application
    // ...
    // 3. Dessiner la scène
    // ... SDL_RenderClear ...
    // ... SDL_RenderCopy ...
    // ... SDL_RenderPresent ...
}
```

La Structure `SDL_Event`

Tous les événements dans SDL2 sont représentés par une **union** appelée `SDL_Event`. Une union est utilisée car un événement peut être de différents types (clavier, souris, fenêtre, etc.), et chaque type a ses propres informations spécifiques. L'union `SDL_Event` contient des structures pour chaque type d'événement possible.

Le membre le plus important de l'union `SDL_Event` est `type`. C'est un `Uint32` qui indique le **type** de l'événement qui s'est produit. En fonction de cette valeur `type`, vous accéderez au membre approprié de l'union pour obtenir les détails spécifiques de l'événement.

Quelques membres courants de l'union `SDL_Event` (accédés via une variable `e` de type `SDL_Event`) :

- `e.type`: Le type d'événement (ex: `SDL_QUIT`, `SDL_KEYDOWN`, `SDL_MOUSEBUTTONDOWN`).
- `e.window`: Structure contenant les informations pour les événements de fenêtre (`SDL_WindowEvent`).
- `e.key`: Structure contenant les informations pour les événements clavier (`SDL_KeyboardEvent`).
- `e.button`: Structure contenant les informations pour les événements de bouton de souris (`SDL_MouseButtonEvent`).
- `e.motion`: Structure contenant les informations pour les événements de mouvement de souris (`SDL_MouseMotionEvent`).
- `e.wheel`: Structure contenant les informations pour les événements de molette de souris (`SDL_MouseWheelEvent`).

- `e.text`: Structure contenant les informations pour les événements de saisie de texte (`SDL_TextInputEvent`).
- `e.quit`: Structure pour l'événement de fermeture (`SDL_QuitEvent`).

Récupérer les Événements : `SDL_PollEvent`

La fonction principale pour récupérer les événements de la file d'attente de SDL est `SDL_PollEvent()`. SDL maintient une file interne où les événements générés par le système d'exploitation sont stockés jusqu'à ce que votre programme les récupère.

Syntaxe : `int SDL_PollEvent(SDL_Event *event);`

- `event`: Un **pointeur** vers une variable `SDL_Event` que vous avez déclarée dans votre code (par exemple, `SDL_Event mon_evenement;`). Si un événement est en attente dans la file, `SDL_PollEvent` **retire** cet événement de la file et **remplit** la structure pointée par `event` avec les informations de cet événement.
- **Valeur de retour :**
  - `1` (vrai) s'il y avait un événement en attente et que la structure `event` a été remplie.
  - `0` (faux) s'il n'y a **aucun** événement en attente dans la file pour le moment.

Utilisation typique dans la boucle principale : On utilise une boucle `while` avec `SDL_PollEvent` pour s'assurer de traiter *tous* les événements qui pourraient s'être accumulés depuis le dernier tour de la boucle principale. C'est important car plusieurs événements (par exemple, plusieurs pressions de touches ou mouvements de souris) peuvent survenir entre deux affichages.

L'idiome courant est : `SDL_Event e; while (SDL_PollEvent(&e) != 0) { // Traiter l'événement 'e' }`

Cette boucle `while (SDL_PollEvent(...))` s'exécute très rapidement (et la condition est fausse) s'il n'y a pas d'événements, permettant au reste de la boucle principale (mise à jour, dessin) de s'exécuter. S'il y a des événements, elle les traite un par un jusqu'à ce que la file soit vide.

Types d'Événements Courants et Traitement

À l'intérieur de la boucle `while (SDL_PollEvent(&e))`, on utilise généralement une instruction `switch` sur `e.type` pour déterminer quel type d'événement s'est produit et agir en conséquence.

- **`SDL_QUIT` :**
  - Type d'événement : `SDL_QUIT`.
  - Généré lorsque l'utilisateur demande à fermer l'application (en cliquant sur la croix de la fenêtre, en utilisant Alt+F4 sous Windows, Cmd+Q sous macOS, ou via d'autres signaux système).
  - C'est l'événement standard pour indiquer que le programme doit se terminer proprement.
  - Traitement typique : Mettre une variable booléenne (comme `continuer` ou `quitter`) à `false` pour sortir de la boucle principale.
  - Structure associée : `e.quit` (contient `type` et `timestamp`).
- **`SDL_KEYDOWN` et `SDL_KEYUP` :**
  - Types d'événement : `SDL_KEYDOWN` (touche enfoncée), `SDL_KEYUP` (touche relâchée).
  - Générés lorsqu'une touche physique du clavier est respectivement **enfoncée** ou **relâchée**.

    - Les informations détaillées se trouvent dans `e.key` (une structure `SDL_KeyboardEvent`).
    - Membres utiles de `e.key`:
        - `windowID`: L'identifiant de la fenêtre qui avait le focus.
        - `state`: `SDL_PRESSED` ou `SDL_RELEASED`.
        - `repeat`: Vaut 0 pour la première pression (`SDL_KEYDOWN`), non nul si l'événement est dû à la répétition automatique de la touche par le système d'exploitation lorsque la touche est maintenue enfoncée. Utile si vous voulez ignorer les répétitions.
        - `keysym`: Une structure `SDL_Keysym` contenant les informations sur la touche :
            - `keysym.scancode`: Le code physique de la touche (`SDL_Scancode`), indépendant de la disposition du clavier. Utile pour les contrôles de jeu (ZQSD/WASD).
            - `keysym.sym`: Le code symbolique de la touche (`SDL_Keycode`), dépendant de la disposition du clavier (ex: `SDLK_z` sur un clavier AZERTY, `SDLK_w` sur un QWERTY pour la même touche physique). Exemples : `SDLK_UP`, `SDLK_DOWN`, `SDLK_LEFT`, `SDLK_RIGHT`, `SDLK_SPACE`, `SDLK_RETURN` (Touche Entrée), `SDLK_ESCAPE`, `SDLK_a`, `SDLK_b`, `SDLK_0`, `SDLK_KP_ENTER` (Entrée du pavé numérique), etc.
            - `keysym.mod`: Les modificateurs actifs (Shift, Ctrl, Alt, Gui/Cmd) au moment de l'événement. C'est un masque de bits (utiliser `&` pour tester). Exemples : `KMOD_SHIFT`, `KMOD_CTRL`, `KMOD_ALT`, `KMOD_GUI`, `KMOD_CAPS`. Tester si Shift est enfoncé : `if (e.key.keysym.mod & KMOD_SHIFT) { ... }`.
    - Traitement typique : Mettre à jour l'état du jeu (commencer à déplacer un personnage sur `KEYDOWN`, arrêter sur `KEYUP`) ou changer l'état de l'application (ouvrir un menu sur `SDLK_ESCAPE`).

(Voir l'exemple de code `code_sdl_event_loop_section24` pour une illustration pratique du traitement de `SDL_KEYDOWN` et `SDL_KEYUP`).

- **`SDL_MOUSEBUTTONDOWN` et `SDL_MOUSEBUTTONUP` :**

    - Types d'événement : `SDL_MOUSEBUTTONDOWN` (bouton enfoncé), `SDL_MOUSEBUTTONUP` (bouton relâché).
    - Générés lorsqu'un bouton de la souris est respectivement **enfoncé** ou **relâché** pendant que le curseur est sur la fenêtre.
    - Les informations se trouvent dans `e.button` (une structure `SDL_MouseButtonEvent`).
    - Membres utiles de `e.button`:
        - `windowID`: L'identifiant de la fenêtre.
        - `which`: L'identifiant de la souris (utile s'il y en a plusieurs).
        - `button`: Le bouton concerné. Constantes : `SDL_BUTTON_LEFT` (généralement 1), `SDL_BUTTON_MIDDLE` (2), `SDL_BUTTON_RIGHT` (3), `SDL_BUTTON_X1` (4), `SDL_BUTTON_X2` (5) (pour les boutons latéraux "précédent/suivant" courants).
        - `state`: `SDL_PRESSED` ou `SDL_RELEASED`.
        - `clicks`: Nombre de clics détectés rapidement au même endroit (1 pour simple clic, 2 pour double clic, etc.). Utile pour implémenter le double-clic.
        - `x`, `y`: Coordonnées (pixels) de la souris *dans la fenêtre* au moment du clic.
    - Traitement typique : Sélectionner un objet dans le jeu ou l'interface, déclencher une action (tirer, interagir), cliquer sur un bouton d'interface utilisateur.

- **`SDL_MOUSEMOTION` :**
    - Type d'événement : `SDL_MOUSEMOTION`.
    - Généré chaque fois que la souris **se déplace** au-dessus de la fenêtre. Peut être généré très fréquemment !
    - Les informations se trouvent dans `e.motion` (une structure `SDL_MouseMotionEvent`).
    - Membres utiles de `e.motion`:
        - `windowID`: L'identifiant de la fenêtre.
        - `which`: L'identifiant de la souris.
        - `state`: État des boutons de la souris *au moment du mouvement*. C'est un masque de bits. Vous pouvez tester si un bouton est enfoncé pendant le mouvement avec l'opérateur `&` et la macro `SDL_BUTTON()`. Ex: `if (e.motion.state & SDL_BUTTON(SDL_BUTTON_LEFT)) { /* Le bouton gauche est enfoncé pendant le déplacement */ }`.
        - `x`, `y`: Nouvelles coordonnées (pixels) de la souris dans la fenêtre.
        - `xrel`, `yrel`: Déplacement relatif (différence en pixels) depuis le dernier événement de mouvement (`nouvelle_x - ancienne_x`, `nouvelle_y - ancienne_y`). Utile pour les contrôles de type "regarder autour" dans les jeux 3D ou pour le glisser-déposer.
    - Traitement typique : Mettre à jour la position d'un curseur personnalisé, gérer le survol (hover) des éléments d'interface, implémenter le glisser-déposer (drag and drop) en combinaison avec `SDL_MOUSEBUTTONDOWN`/`UP`.
- **`SDL_MOUSEWHEEL` :**
    - Type d'événement : `SDL_MOUSEWHEEL`.
    - Généré lorsque la **molette** de la souris est tournée pendant que le curseur est sur la fenêtre.
    - Les informations se trouvent dans `e.wheel` (une structure `SDL_MouseWheelEvent`).
    - Membres utiles de `e.wheel`:
        - `windowID`: L'identifiant de la fenêtre.
        - `which`: L'identifiant de la souris.
        - `x`, `y`: Déplacement de la molette. `y` est le plus courant (défilement vertical). Une valeur **positive** indique généralement un défilement vers le haut ou loin de l'utilisateur. Une valeur **négative** indique un défilement vers le bas ou vers l'utilisateur. L'amplitude exacte (ex: +1 ou -1, ou plus) peut varier. `x` est pour le défilement horizontal (si supporté).
        - `direction`: Indique si la direction du défilement est "normale" ou "inversée" par rapport aux conventions système (`SDL_MOUSEWHEEL_NORMAL` ou `SDL_MOUSEWHEEL_FLIPPED`). Il est souvent plus simple d'utiliser directement le signe de `e.wheel.y` ou `e.wheel.x`.
    - Traitement typique : Zoomer/dézoomer une vue, faire défiler une liste, un document ou une carte.
- **`SDL_TEXTINPUT` :**
    - Type d'événement : `SDL_TEXTINPUT`.
    - Généré lorsque le système d'exploitation a composé un **caractère textuel** à partir des pressions de touches (prend en compte les majuscules/minuscules via Shift, les caractères accentués, les méthodes d'entrée pour d'autres langues, etc.). C'est l'événement **préférable** si vous voulez récupérer du texte tapé par l'utilisateur pour un champ de saisie, plutôt que d'essayer de le reconstruire à partir des `SDL_KEYDOWN`.

    - Les informations se trouvent dans `e.text` (une structure `SDL_TextInputEvent`).
    - Membre utile de `e.text`:
        - `text`: Une chaîne de caractères (`char[]`, tableau de taille fixe `SDL_TEXTINPUTEVENT_TEXT_SIZE`, généralement 32) contenant le texte UTF-8 entré. Cette chaîne est toujours terminée par `\0`. Elle peut contenir plusieurs octets pour représenter un seul caractère complexe en UTF-8.
    - Traitement typique : Ajouter la chaîne `e.text.text` à un buffer qui stocke le contenu du champ de saisie.
    - **Important :** Pour que les événements `SDL_TEXTINPUT` soient générés, vous devez explicitement **démarrer** la capture de texte avec `SDL_StartTextInput();` lorsque votre champ de saisie a le focus, et l'**arrêter** avec `SDL_StopTextInput();` lorsqu'il perd le focus.

(Voir l'exemple de code `code_sdl_text_input_section24` pour une illustration pratique).

- **`SDL_WINDOWEVENT` :**
    - Type d'événement : `SDL_WINDOWEVENT`.
    - Un type d'événement générique qui signale divers changements liés à la fenêtre elle-même.
    - Les informations se trouvent dans `e.window` (une structure `SDL_WindowEvent`).
    - Le membre `e.window.event` (un `Uint8`) indique le **sous-type** spécifique de l'événement de fenêtre. Quelques exemples importants :
        - `SDL_WINDOWEVENT_SHOWN`: La fenêtre est devenue visible.
        - `SDL_WINDOWEVENT_HIDDEN`: La fenêtre est devenue cachée.
        - `SDL_WINDOWEVENT_EXPOSED`: Une partie de la fenêtre doit être redessinée (par exemple, après avoir été recouverte puis découverte). Utile pour déclencher un redessin complet si nécessaire.
        - `SDL_WINDOWEVENT_MOVED`: La fenêtre a été déplacée. `e.window.data1` (nouvelle position x) et `e.window.data2` (nouvelle position y) contiennent les nouvelles coordonnées.
        - `SDL_WINDOWEVENT_RESIZED`: La taille de la fenêtre a changé (si elle est redimensionnable). `e.window.data1` (nouvelle largeur) et `e.window.data2` (nouvelle hauteur) contiennent la nouvelle taille en pixels. Important pour adapter votre logique de rendu (positions, échelles).
        - `SDL_WINDOWEVENT_SIZE_CHANGED`: La taille a changé (peut être déclenché par `RESIZED` ou d'autres actions système). Souvent traité comme `RESIZED`.
        - `SDL_WINDOWEVENT_MINIMIZED`: Fenêtre réduite dans la barre des tâches.
        - `SDL_WINDOWEVENT_MAXIMIZED`: Fenêtre agrandie pour remplir l'écran.
        - `SDL_WINDOWEVENT_RESTORED`: Fenêtre restaurée à sa taille/position normale après minimisation/maximisation.
        - `SDL_WINDOWEVENT_ENTER`: Le curseur de la souris est entré dans les limites de la fenêtre.
        - `SDL_WINDOWEVENT_LEAVE`: Le curseur de la souris a quitté les limites de la fenêtre.
        - `SDL_WINDOWEVENT_FOCUS_GAINED`: La fenêtre a obtenu le focus clavier (elle est devenue la fenêtre active).
        - `SDL_WINDOWEVENT_FOCUS_LOST`: La fenêtre a perdu le focus clavier. Utile pour mettre le jeu en pause, par exemple.
        - `SDL_WINDOWEVENT_CLOSE`: La fenêtre a été demandée à être fermée par le système (par exemple, via le gestionnaire de fenêtres, différent de `SDL_QUIT` qui vient souvent de

l'intérieur de l'application, mais souvent traité de la même manière : `continuer = false;`).

- Traitement typique : Redessiner après `EXPOSED`, ajuster les dimensions et la disposition après `RESIZED`, mettre le jeu en pause/reprendre sur `FOCUS_LOST`/`GAINED`, etc.

Il existe d'autres types d'événements (joystick, manette de jeu `SDL_CONTROLLER...`, toucher `SDL_FINGER...`, gestion de l'audio, événements utilisateur personnalisés `SDL_USEREVENT`...), mais ceux listés ci-dessus couvrent les interactions les plus courantes pour les applications de bureau et les jeux simples.

```c
#include <stdio.h>
#include <stdbool.h> // Pour bool, true, false
#include <SDL2/SDL.h>

const int LARGEUR_FENETRE = 640;
const int HAUTEUR_FENETRE = 480;

int main(int argc, char* argv[]) {
    SDL_Window* fenetre = NULL;
    SDL_Renderer* rendu = NULL;

    // --- Initialisation ---
    if (SDL_Init(SDL_INIT_VIDEO) < 0) {
        fprintf(stderr, "SDL_Init Error: %s\n", SDL_GetError());
        return 1;
    }
    fenetre = SDL_CreateWindow("Gestion Événements",
SDL_WINDOWPOS_CENTERED, SDL_WINDOWPOS_CENTERED,
                               LARGEUR_FENETRE, HAUTEUR_FENETRE,
SDL_WINDOW_SHOWN);
    if (!fenetre) { fprintf(stderr, "SDL_CreateWindow Error: %s\n",
SDL_GetError()); SDL_Quit(); return 1; }
    rendu = SDL_CreateRenderer(fenetre, -1, SDL_RENDERER_ACCELERATED);
    if (!rendu) { fprintf(stderr, "SDL_CreateRenderer Error: %s\n",
SDL_GetError()); SDL_DestroyWindow(fenetre); SDL_Quit(); return 1; }

    // --- Variables pour la boucle principale ---
    bool continuer = true; // Contrôle la sortie de la boucle
    SDL_Event event;       // Pour stocker l'événement courant
    int mouse_x = 0, mouse_y = 0; // Pour suivre la position de la souris

    // --- Boucle Principale (Event Loop) ---
    printf("Fenêtre ouverte. Interagissez ou fermez la fenêtre.\n");
    while (continuer) {

        // 1. Traitement des événements en attente
        while (SDL_PollEvent(&event)) { // Tant qu'il y a des événements...
            // Traiter l'événement 'event'
            switch (event.type) {
                case SDL_QUIT: // L'utilisateur a demandé à quitter (ex:
croix)
                    printf("Événement SDL_QUIT reçu.\n");
```

```
                    continuer = false; // Met fin à la boucle principale
                    break;

                case SDL_KEYDOWN: // Une touche a été enfoncée
                    // event.key contient les détails
                    printf("Touche enfoncée : %s (Code: %d)\n",
                           SDL_GetKeyName(event.key.keysym.sym), // Nom de
la touche
                           event.key.keysym.sym);               // Code
numérique
                    // Quitter si la touche Échap est pressée
                    if (event.key.keysym.sym == SDLK_ESCAPE) {
                        printf("  Touche Échap pressée, fermeture.\n");
                        continuer = false;
                    }
                    break;

                case SDL_KEYUP: // Une touche a été relâchée
                    printf("Touche relâchée : %s\n",
SDL_GetKeyName(event.key.keysym.sym));
                    break;

                case SDL_MOUSEMOTION: // La souris a bougé
                    // event.motion contient les détails
                    mouse_x = event.motion.x;
                    mouse_y = event.motion.y;
                    // Afficher les coordonnées (peut être verbeux !)
                    // printf("Souris bougée : (%d, %d)\n", mouse_x,
mouse_y);
                    break;

                case SDL_MOUSEBUTTONDOWN: // Un bouton de souris a été
enfoncé
                    // event.button contient les détails
                    printf("Bouton souris enfoncé : %d à (%d, %d), Clics:
%d\n",
                           event.button.button, event.button.x,
event.button.y, event.button.clicks);
                    break;

                case SDL_MOUSEBUTTONUP: // Un bouton de souris a été
relâché
                     printf("Bouton souris relâché : %d à (%d, %d)\n",
                           event.button.button, event.button.x,
event.button.y);
                    break;

                case SDL_MOUSEWHEEL: // La molette a tourné
                    // event.wheel contient les détails
                    printf("Molette souris : y=%d", event.wheel.y);
                    if (event.wheel.direction == SDL_MOUSEWHEEL_FLIPPED) {
                        printf(" (inversé)");
                    }
                    printf("\n");
```

```
                    break;

                // Ajouter d'autres cas ici (SDL_TEXTINPUT,
SDL_WINDOWEVENT, etc.) si nécessaire
            }
        } // Fin de la boucle SDL_PollEvent

        // 2. Mettre à jour l'état (pas grand chose dans cet exemple)
        // ...

        // 3. Dessiner la scène
        // Effacer l'écran (fond gris)
        SDL_SetRenderDrawColor(rendu, 100, 100, 100, 255);
        SDL_RenderClear(rendu);

        // Dessiner un petit carré à la position de la souris
        SDL_Rect curseur_souris = { mouse_x - 2, mouse_y - 2, 5, 5 }; //
Petit carré 5x5 centré
        SDL_SetRenderDrawColor(rendu, 255, 255, 0, 255); // Jaune
        SDL_RenderFillRect(rendu, &curseur_souris);

        // Présenter le rendu
        SDL_RenderPresent(rendu);

        // Petite pause pour ne pas utiliser 100% du CPU
        SDL_Delay(10);

    } // Fin de la boucle principale (while continuer)

    // --- Nettoyage ---
    SDL_DestroyRenderer(rendu);
    SDL_DestroyWindow(fenetre);
    SDL_Quit();
    printf("Application terminée.\n");

    return 0;
}

/* Compilation (Linux/macOS):
   gcc main.c -o events_test -lSDL2
*/
```

---

25. Affichage de Texte avec SDLttf

Afficher du texte statique avec des images pré-rendues est possible, mais pour afficher du texte dynamique (scores, noms, messages, dialogues...), nous avons besoin d'une solution plus flexible. La bibliothèque satellite **SDL2_ttf** permet de charger des fichiers de police TrueType (`.ttf`) et de rendre (dessiner) des chaînes de caractères en utilisant ces polices.

Comme `SDL_image`, `SDL_ttf` doit être initialisée séparément et ses ressources (polices, surfaces/textures de texte) doivent être gérées et libérées correctement.

Initialisation de `SDL_ttf` (`TTF_Init`)

Avant d'utiliser toute fonction de `SDL_ttf`, vous devez initialiser la bibliothèque.

Syntaxe : `int TTF_Init(void);`

- Prend aucun argument.
- **Valeur de retour :** Retourne `0` en cas de succès, ou une valeur négative en cas d'erreur (par exemple, si SDL n'a pas été initialisé correctement avant).

Vérifiez toujours la valeur de retour !

Chargement d'une police (`TTF_OpenFont`)

Pour rendre du texte, vous avez besoin d'un fichier de police (`.ttf`). Vous devez charger ce fichier en mémoire à l'aide de `TTF_OpenFont`.

Syntaxe : `TTF_Font* TTF_OpenFont(const char *file, int ptsize);`

- `file`: Chemin vers le fichier de police (`.ttf`).
- `ptsize`: La taille de la police souhaitée en **points** (point size). C'est une unité typographique standard. La hauteur en pixels résultante dépendra de la police elle-même.
- **Valeur de retour :** Un pointeur vers une structure `TTF_Font` opaque (`TTF_Font*`) représentant la police chargée à la taille spécifiée, ou `NULL` en cas d'erreur (fichier non trouvé, format invalide, mémoire insuffisante...). Utilisez `TTF_GetError()` pour obtenir le message d'erreur.

Vérifiez toujours si le pointeur retourné est `NULL` ! Vous conserverez ce pointeur `TTF_Font*` pour l'utiliser lors du rendu du texte. Une même police peut être ouverte plusieurs fois à différentes tailles.

```
#include <stdio.h>
#include <SDL2/SDL.h>
#include <SDL2/SDL_ttf.h> // Inclure l'en-tête SDL_ttf

const char* CHEMIN_POLICE = "assets/font.ttf"; // REMPLACEZ par le chemin
de votre police .ttf
const int TAILLE_POLICE = 28;

int main(int argc, char* argv[]) {
    // --- Initialisation SDL (Vidéo au minimum) ---
    if (SDL_Init(SDL_INIT_VIDEO) < 0) {
        fprintf(stderr, "SDL_Init Error: %s\n", SDL_GetError());
        return 1;
    }

    // --- Initialisation SDL_ttf ---
    printf("Initialisation de SDL_ttf...\n");
    if (TTF_Init() == -1) { // Vérifie si TTF_Init retourne -1 (erreur)
        fprintf(stderr, "Erreur lors de l'initialisation de SDL_ttf: %s\n",
TTF_GetError());
        SDL_Quit();
        return 1;
    }
```

```
    printf("SDL_ttf initialisé avec succès.\n");

    // --- Chargement de la police ---
    TTF_Font* police = NULL; // Pointeur pour stocker la police chargée
    printf("Chargement de la police '%s' à la taille %d...\n",
CHEMIN_POLICE, TAILLE_POLICE);
    police = TTF_OpenFont(CHEMIN_POLICE, TAILLE_POLICE);

    // Vérification du chargement
    if (police == NULL) {
        fprintf(stderr, "Erreur lors du chargement de la police: %s\n",
TTF_GetError());
        TTF_Quit(); // Nettoyer TTF avant de quitter SDL
        SDL_Quit();
        return 1;
    }
    printf("Police chargée avec succès.\n");

    // --- Ici on utiliserait la police ---
    // ...

    // --- Nettoyage ---
    printf("\nFermeture...\n");
    if (police) {
        TTF_CloseFont(police); // Libérer la police !
        police = NULL;
        printf("Police fermée.\n");
    }
    TTF_Quit(); // Quitter le sous-système TTF
    printf("SDL_ttf quitté.\n");
    SDL_Quit(); // Quitter SDL
    printf("SDL quitté.\n");

    return 0;
}

/* Compilation (Linux/macOS):
   gcc main.c -o ttf_test -lSDL2 -lSDL2_ttf
   (Assurez-vous d'avoir un fichier font.ttf au chemin spécifié)
*/
```

Rendu de texte sur une `SDL_Surface`

Une fois la police chargée, vous pouvez l'utiliser pour "dessiner" une chaîne de caractères donnée sur une `SDL_Surface` en mémoire CPU. `SDL_ttf` propose plusieurs fonctions de rendu, différant par la qualité et la gestion de l'arrière-plan :

- **`SDL_Surface* TTF_RenderText_Solid(TTF_Font *font, const char *text, SDL_Color fg);`**
    - Rendu le plus rapide, qualité la moins bonne (pas d'anti-aliasing).

    - `font`: La police chargée (`TTF_Font*`).
    - `text`: La chaîne de caractères (UTF-8) à rendre.
    - `fg`: La couleur du texte (`SDL_Color`). L'arrière-plan de la surface sera transparent (ou d'une couleur clé si la surface n'a pas de canal alpha).
    - Retourne une nouvelle `SDL_Surface*` contenant le texte rendu, ou `NULL` en cas d'erreur.

- **`SDL_Surface* TTF_RenderText_Shaded(TTF_Font *font, const char *text, SDL_Color fg, SDL_Color bg);`**
    - Rendu avec anti-aliasing (plus lisse).
    - `font`, `text`, `fg`: Comme pour `_Solid`.
    - `bg`: La couleur d'**arrière-plan** (`SDL_Color`) utilisée pour remplir la surface derrière le texte.
    - Retourne une nouvelle `SDL_Surface*` ou `NULL`.

- **`SDL_Surface* TTF_RenderText_Blended(TTF_Font *font, const char *text, SDL_Color fg);`**
    - Rendu avec anti-aliasing (le plus lisse) et un **arrière-plan transparent** (canal alpha). C'est souvent le **meilleur choix** pour intégrer du texte sur d'autres images.
    - `font`, `text`, `fg`: Comme pour `_Solid`.
    - Retourne une nouvelle `SDL_Surface*` (avec un canal alpha) ou `NULL`.

Il existe aussi des variantes (`_UTF8_`, `_UNICODE_`, `_Glyph_`) pour différents encodages ou pour rendre des glyphes spécifiques, mais les fonctions `TTF_RenderText_*` gèrent généralement l'UTF-8 correctement.

Important : Ces fonctions retournent une **nouvelle surface** à chaque appel. Vous êtes responsable de libérer cette surface avec `SDL_FreeSurface()` une fois que vous n'en avez plus besoin (typiquement, après l'avoir convertie en texture).

```c
#include <stdio.h>
#include <SDL2/SDL.h>
#include <SDL2/SDL_ttf.h>

const char* CHEMIN_POLICE = "assets/font.ttf"; // Votre police
const int TAILLE_POLICE = 24;

int main(int argc, char* argv[]) {
    // --- Initialisation SDL et TTF --- (supposée réussie)
    if (SDL_Init(SDL_INIT_VIDEO) < 0) return 1;
    if (TTF_Init() == -1) { SDL_Quit(); return 1; }

    // --- Chargement Police --- (supposé réussi)
    TTF_Font* police = TTF_OpenFont(CHEMIN_POLICE, TAILLE_POLICE);
    if (!police) { TTF_Quit(); SDL_Quit(); return 1; }

    // --- Rendu du texte sur des surfaces ---
    SDL_Color couleurBlanche = {255, 255, 255, 255}; // Blanc opaque
    SDL_Color couleurNoire = {0, 0, 0, 255};       // Noir opaque

    const char* texte = "Bonjour SDL_ttf !";
```

```
    // 1. Rendu "Solid" (rapide, aliasé)
    SDL_Surface* surfaceSolid = TTF_RenderText_Solid(police, texte,
couleurBlanche);
    if (!surfaceSolid) {
        fprintf(stderr, "Erreur TTF_RenderText_Solid: %s\n",
TTF_GetError());
    } else {
        printf("Surface 'Solid' créée : %d x %d pixels\n", surfaceSolid->w,
surfaceSolid->h);
        // ... (on la convertirait en texture pour l'afficher) ...
        SDL_FreeSurface(surfaceSolid); // Libérer la surface après usage
    }

    // 2. Rendu "Shaded" (lisse, fond opaque)
    SDL_Surface* surfaceShaded = TTF_RenderText_Shaded(police, texte,
couleurBlanche, couleurNoire);
    if (!surfaceShaded) {
        fprintf(stderr, "Erreur TTF_RenderText_Shaded: %s\n",
TTF_GetError());
    } else {
        printf("Surface 'Shaded' créée : %d x %d pixels\n", surfaceShaded-
>w, surfaceShaded->h);
        SDL_FreeSurface(surfaceShaded);
    }

    // 3. Rendu "Blended" (lisse, fond transparent) - Généralement le
meilleur choix
    SDL_Surface* surfaceBlended = TTF_RenderText_Blended(police, texte,
couleurBlanche);
    if (!surfaceBlended) {
        fprintf(stderr, "Erreur TTF_RenderText_Blended: %s\n",
TTF_GetError());
    } else {
        printf("Surface 'Blended' créée : %d x %d pixels\n",
surfaceBlended->w, surfaceBlended->h);
        // C'est cette surface qu'on convertirait typiquement en texture
        SDL_FreeSurface(surfaceBlended);
    }


    // --- Nettoyage ---
    TTF_CloseFont(police);
    TTF_Quit();
    SDL_Quit();

    return 0;
}

/* Compilation (Linux/macOS):
   gcc main.c -o ttf_render_test -lSDL2 -lSDL2_ttf
*/
```

Création de Texture depuis la Surface de Texte et Affichage

Comme pour les images chargées avec `SDL_image`, le rendu de texte via `SDL_ttf` produit une `SDL_Surface` en mémoire CPU. Pour afficher ce texte efficacement avec le `SDL_Renderer` (qui utilise le GPU), nous devons convertir cette surface en `SDL_Texture`.

Le processus est identique à celui des images :

1. **Rendre le texte** sur une `SDL_Surface` (en utilisant `TTF_RenderText_Blended` de préférence pour la transparence).
2. **Vérifier** si la surface a été créée (`!= NULL`).
3. **Créer une `SDL_Texture`** à partir de cette surface en utilisant `SDL_CreateTextureFromSurface(rendu, surface_texte)`.
4. **Vérifier** si la texture a été créée (`!= NULL`).
5. **Libérer la `SDL_Surface`** intermédiaire avec `SDL_FreeSurface(surface_texte)`, car elle n'est plus nécessaire (la texture contient les données pour le GPU).
6. (Optionnel) Récupérer les dimensions de la texture créée avec `SDL_QueryTexture` si vous en avez besoin pour le positionnement.
7. Dans la boucle de dessin : **Copier la `SDL_Texture`** sur le rendu à la position souhaitée avec `SDL_RenderCopy(rendu, texture_texte, NULL, &rect_destination)`.
8. Lorsque vous n'avez plus besoin d'afficher ce texte (ou à la fin du programme), **libérer la `SDL_Texture`** avec `SDL_DestroyTexture(texture_texte)`.

Optimisation : Créer une texture à chaque fois que vous voulez afficher du texte peut être inefficace si le texte change souvent mais pas à chaque frame. Une approche courante est de :

- Créer la texture une fois lorsque le texte est généré ou modifié.
- La stocker (par exemple, dans une structure).
- La réutiliser avec `SDL_RenderCopy` à chaque frame tant que le texte n'a pas changé.
- Ne la détruire et la recréer que lorsque le texte lui-même change.

Fermeture de la Police (`TTF_CloseFont`) et Nettoyage (`TTF_Quit`)

Comme pour toutes les ressources allouées, il est essentiel de libérer la mémoire utilisée par les polices chargées lorsque vous n'en avez plus besoin.

- **`void TTF_CloseFont(TTF_Font *font);`**
  - Libère la mémoire associée à une police spécifique qui a été chargée avec `TTF_OpenFont`.
  - Appelez cette fonction pour chaque `TTF_Font*` que vous avez ouvert avec succès.
- **`void TTF_Quit(void);`**
  - Nettoie et ferme le sous-système `SDL_ttf`. Doit être appelée après avoir fermé toutes les polices et avant d'appeler `SDL_Quit()`.

Ne pas fermer les polices ou quitter `SDL_ttf` correctement peut entraîner des fuites de mémoire.

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

```c
#include <SDL2/SDL.h>
#include <SDL2/SDL_ttf.h> // Ne pas oublier cet include

const int LARGEUR_FENETRE = 800;
const int HAUTEUR_FENETRE = 600;
const char* CHEMIN_POLICE = "assets/font.ttf"; // Assurez-vous que ce
fichier existe
const int TAILLE_POLICE_TITRE = 36;
const int TAILLE_POLICE_NORMAL = 24;

// Fonction pour créer une texture de texte (simplifiée, sans gestion
d'erreur robuste ici)
SDL_Texture* creer_texture_texte(SDL_Renderer* rendu, TTF_Font* police,
const char* texte, SDL_Color couleur) {
    SDL_Surface* surface_texte = TTF_RenderText_Blended(police, texte,
couleur);
    if (!surface_texte) {
        fprintf(stderr, "Erreur TTF_RenderText_Blended: %s\n",
TTF_GetError());
        return NULL;
    }
    SDL_Texture* texture_texte = SDL_CreateTextureFromSurface(rendu,
surface_texte);
    if (!texture_texte) {
        fprintf(stderr, "Erreur SDL_CreateTextureFromSurface: %s\n",
SDL_GetError());
    }
    SDL_FreeSurface(surface_texte); // Libère la surface intermédiaire
    return texture_texte;
}

int main(int argc, char* argv[]) {
    SDL_Window* fenetre = NULL;
    SDL_Renderer* rendu = NULL;
    TTF_Font* police_titre = NULL;
    TTF_Font* police_normal = NULL;
    SDL_Texture* tex_titre = NULL;
    SDL_Texture* tex_message = NULL;

    // --- Initialisations ---
    if (SDL_Init(SDL_INIT_VIDEO) < 0) { /*...*/ return 1; }
    if (TTF_Init() == -1) { /*...*/ SDL_Quit(); return 1; }

    fenetre = SDL_CreateWindow("Affichage Texte TTF",
SDL_WINDOWPOS_CENTERED, SDL_WINDOWPOS_CENTERED,
                               LARGEUR_FENETRE, HAUTEUR_FENETRE,
SDL_WINDOW_SHOWN);
    if (!fenetre) { /*...*/ TTF_Quit(); SDL_Quit(); return 1; }

    rendu = SDL_CreateRenderer(fenetre, -1, SDL_RENDERER_ACCELERATED |
SDL_RENDERER_PRESENTVSYNC);
    if (!rendu) { /*...*/ SDL_DestroyWindow(fenetre); TTF_Quit();
SDL_Quit(); return 1; }
```

```
    // --- Chargement des polices ---
    police_titre = TTF_OpenFont(CHEMIN_POLICE, TAILLE_POLICE_TITRE);
    police_normal = TTF_OpenFont(CHEMIN_POLICE, TAILLE_POLICE_NORMAL);
    if (!police_titre || !police_normal) {
        fprintf(stderr, "Erreur TTF_OpenFont: %s\n", TTF_GetError());
        // Nettoyage partiel...
        TTF_CloseFont(police_titre);
        TTF_CloseFont(police_normal);
        SDL_DestroyRenderer(rendu);
        SDL_DestroyWindow(fenetre);
        TTF_Quit(); SDL_Quit(); return 1;
    }

    // --- Création des textures de texte ---
    SDL_Color couleur_blanche = {255, 255, 255, 255};
    SDL_Color couleur_jaune = {255, 255, 0, 255};

    tex_titre = creer_texture_texte(rendu, police_titre, "Titre Principal",
couleur_blanche);
    tex_message = creer_texture_texte(rendu, police_normal, "Ceci est un
message affiché avec SDL_ttf.", couleur_jaune);

    if (!tex_titre || !tex_message) {
        // Nettoyage complet si une texture échoue
        SDL_DestroyTexture(tex_titre); SDL_DestroyTexture(tex_message);
        TTF_CloseFont(police_titre); TTF_CloseFont(police_normal);
        SDL_DestroyRenderer(rendu); SDL_DestroyWindow(fenetre);
        TTF_Quit(); SDL_Quit(); return 1;
    }

    // Obtenir les dimensions des textures pour le positionnement
    int w_titre, h_titre, w_msg, h_msg;
    SDL_QueryTexture(tex_titre, NULL, NULL, &w_titre, &h_titre);
    SDL_QueryTexture(tex_message, NULL, NULL, &w_msg, &h_msg);

    // --- Boucle Principale ---
    bool continuer = true;
    SDL_Event event;

    while (continuer) {
        while (SDL_PollEvent(&event)) {
            if (event.type == SDL_QUIT) {
                continuer = false;
            }
            // Ajouter d'autres traitements d'événements ici si nécessaire
        }

        // --- Dessin ---
        SDL_SetRenderDrawColor(rendu, 30, 30, 70, 255); // Fond bleu nuit
        SDL_RenderClear(rendu);

        // Afficher le titre (centré en haut)
        SDL_Rect rect_titre = { (LARGEUR_FENETRE - w_titre) / 2, 50,
w_titre, h_titre };
```

```
        SDL_RenderCopy(rendu, tex_titre, NULL, &rect_titre);

        // Afficher le message (centré plus bas)
        SDL_Rect rect_msg = { (LARGEUR_FENETRE - w_msg) / 2, 200, w_msg,
h_msg };
        SDL_RenderCopy(rendu, tex_message, NULL, &rect_msg);

        // Présenter
        SDL_RenderPresent(rendu);
        SDL_Delay(10);
    }

    // --- Nettoyage Final ---
    printf("Nettoyage...\n");
    SDL_DestroyTexture(tex_titre);
    SDL_DestroyTexture(tex_message);
    TTF_CloseFont(police_titre);
    TTF_CloseFont(police_normal);
    SDL_DestroyRenderer(rendu);
    SDL_DestroyWindow(fenetre);
    TTF_Quit();
    SDL_Quit();
    printf("Terminé.\n");

    return 0;
}

/* Compilation (Linux/macOS):
   gcc main.c -o affiche_texte -lSDL2 -lSDL2_ttf
   (Assurez-vous d'avoir le fichier font.ttf au chemin spécifié)
*/
```

L'affichage de texte avec `SDL_ttf` suit donc un processus en trois étapes principales : charger la police, rendre le texte souhaité sur une surface temporaire, puis créer une texture à partir de cette surface pour l'affichage rapide par le rendu. N'oubliez pas de gérer les erreurs à chaque étape et de libérer toutes les ressources (polices, surfaces, textures) lorsque vous avez terminé.

---

26. Programmation Parallèle avec OpenMP

Les processeurs modernes contiennent de plus en plus de **cœurs** (cores), capables d'exécuter plusieurs instructions simultanément. Pour exploiter pleinement cette puissance matérielle et accélérer les calculs intensifs, on utilise la **programmation parallèle**. L'idée est de diviser une tâche complexe en sous-tâches plus petites qui peuvent être exécutées en parallèle (en même temps) sur différents cœurs ou processeurs.

Introduction au parallélisme (Threads)

Une des approches courantes pour le parallélisme sur une machine à mémoire partagée (comme un PC standard) est l'utilisation de **threads** (fils d'exécution). Un thread est une séquence d'instructions qui peut

s'exécuter indépendamment au sein d'un même programme (processus). Un programme peut lancer plusieurs threads qui s'exécutent "en parallèle", partageant la même mémoire (variables globales, tas).

Gérer les threads manuellement (avec des bibliothèques comme Pthreads sous Linux/macOS ou l'API Windows) peut être complexe (création, synchronisation, gestion des données partagées).

Concepts d'OpenMP (Approche basée sur les directives)

OpenMP (Open Multi-Processing) est une API (Application Programming Interface) qui simplifie grandement la programmation parallèle à mémoire partagée en C, C++, et Fortran. Au lieu de manipuler directement les threads, OpenMP utilise une approche basée sur des **directives de préprocesseur** (`#pragma omp ...`) et des fonctions de bibliothèque.

Le programmeur insère ces directives spéciales dans le code séquentiel pour indiquer au compilateur quelles parties peuvent être parallélisées. Le compilateur, s'il supporte OpenMP, se charge alors de générer le code nécessaire pour créer et gérer les threads, distribuer le travail, et synchroniser les accès aux données.

Avantages d'OpenMP :

- **Simplicité :** Permet souvent de paralléliser du code séquentiel existant (en particulier les boucles `for`) avec relativement peu de modifications.
- **Portabilité :** Les directives OpenMP sont standardisées et supportées par de nombreux compilateurs majeurs (GCC, Clang, MSVC, Intel...).
- **Performance (potentielle) :** Peut offrir des accélérations significatives sur les machines multi-cœurs pour les tâches parallélisables.
- **Incrémental :** On peut commencer par paralléliser les parties les plus coûteuses d'un programme et ajouter progressivement plus de parallélisme.

OpenMP est particulièrement efficace pour le **parallélisme de données**, où la même opération est appliquée indépendamment à différents éléments d'un grand ensemble de données (par exemple, traiter les pixels d'une image, effectuer des calculs sur les éléments d'un grand tableau).

Configuration du compilateur (`-fopenmp`)

Pour que le compilateur reconnaisse et traite les directives OpenMP (`#pragma omp ...`), vous devez activer le support OpenMP lors de la compilation et de l'édition de liens.

- **Avec GCC ou Clang :** Ajoutez l'option `-fopenmp` aux deux étapes (compilation et liaison).

Exemple de compilation et liaison : `gcc -Wall -Wextra -std=c11 -g -fopenmp mon_fichier.c -o mon_programme -lm` `gcc -Wall -Wextra -std=c11 -g -fopenmp -c fichier1.c -o fichier1.o` `gcc -Wall -Wextra -std=c11 -g -fopenmp -c fichier2.c -o fichier2.o` `gcc -fopenmp fichier1.o fichier2.o -o mon_programme -lm`

- **Avec Visual Studio (MSVC) :** Activez le support OpenMP dans les propriétés du projet :
  - Propriétés du projet -> C/C++ -> Langage -> Prise en charge d'OpenMP : Oui (`/openmp`)

Si vous n'activez pas cette option, le compilateur ignorera simplement les directives `#pragma omp` et le code s'exécutera séquentiellement (sans erreur, mais sans parallélisme).

Directive `#pragma omp parallel for` (Parallélisation de boucles)

La directive la plus couramment utilisée avec OpenMP est `#pragma omp parallel for`. Placée juste avant une boucle `for`, elle indique au compilateur que les itérations de cette boucle peuvent être exécutées en parallèle par une équipe de threads.

Syntaxe de base : `#pragma omp parallel for [clauses ...] for (initialisation; condition; mise_a_jour) { // Corps de la boucle // Chaque itération doit être (majoritairement) indépendante des autres } // Le code après la boucle n'est exécuté qu'après la fin de TOUTES les itérations parallèles.`

Fonctionnement :

1. Lorsque l'exécution atteint le `#pragma omp parallel for`, une équipe de threads est créée (le nombre de threads dépend souvent du système ou peut être contrôlé par des variables d'environnement comme `OMP_NUM_THREADS`).
2. L'ensemble des itérations de la boucle `for` est divisé entre les threads de l'équipe. Chaque thread exécute un sous-ensemble des itérations.
3. Les threads exécutent leur part des itérations en parallèle.
4. Une fois que *tous* les threads ont terminé leur travail sur la boucle, ils se synchronisent à la fin de la région parallèle (implicitement après la boucle `for`), et seul le thread principal continue l'exécution du code qui suit la boucle.

Restrictions importantes pour la boucle `for` : Pour que `#pragma omp parallel for` fonctionne correctement et sans ambiguïté, la boucle `for` doit avoir une forme "canonique" :

- La variable de boucle (ex: `i`) doit avoir un type entier ou pointeur.
- L'initialisation doit être simple (ex: `i = 0`).
- La condition doit être une comparaison simple (ex: `i < N`, `i <= N`, `i > N`, `i >= N`).
- La mise à jour doit être une incrémentation ou décrémentation simple par une constante (ex: `i++`, `++i`, `i--`, `--i`, `i += step`, `i -= step`).
- Les itérations doivent être **largement indépendantes**. Modifier une variable dans une itération qui est lue ou écrite par une autre itération sans protection peut créer des **conditions de concurrence (race conditions)** et des résultats incorrects.

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h> // Inclure l'en-tête OpenMP

#define N 10 // Nombre d'itérations

int main() {
    int i;
    int thread_id;

    printf("Début de la section parallèle.\n");
    printf("Nombre max de threads disponibles : %d\n",
omp_get_max_threads());

    // La directive indique que la boucle for suivante doit être
parallélisée
    #pragma omp parallel for
```

```
    for (i = 0; i < N; i++) {
        // omp_get_thread_num() retourne l'ID du thread courant (0, 1, 2,
...)
        thread_id = omp_get_thread_num();
        // Chaque thread exécute cette ligne pour certaines valeurs de 'i'
        printf("  Thread %d traite l'itération i = %d\n", thread_id, i);

        // Simulation d'un travail
        // Note: sleep n'est pas idéal ici, mais pour l'exemple
        // #include <unistd.h> // Pour sleep (Unix-like)
        // sleep(1);
    } // Barrière implicite ici : attend que tous les threads aient fini la
boucle

    printf("Fin de la section parallèle. Toutes les itérations sont
terminées.\n");

    return 0;
}

/* Compilation (GCC/Clang):
   gcc -Wall -fopenmp exemple_parallel_for.c -o exemple_parallel_for
   Exécution :
   ./exemple_parallel_for
   (La sortie montrera probablement les itérations traitées dans un ordre
    non séquentiel par différents threads)
   Pour contrôler le nombre de threads (ex: 4):
   export OMP_NUM_THREADS=4
   ./exemple_parallel_for
*/
```

Clauses OpenMP (`private`, `shared`, `reduction`, etc.)

Lorsque vous parallélisez une boucle, il est crucial de gérer correctement les variables utilisées à l'intérieur et à l'extérieur de la boucle. OpenMP utilise des **clauses** ajoutées à la directive `#pragma omp ...` pour spécifier le **statut de partage** des variables entre les threads.

- **`shared(liste_variables)` :** (Comportement par défaut pour la plupart des variables définies *avant* la région parallèle). Les variables listées sont **partagées** par tous les threads. Tous les threads lisent et écrivent dans la *même* variable en mémoire. **Attention :** Si plusieurs threads écrivent dans une variable partagée sans synchronisation, cela crée une **condition de concurrence (race condition)** et le résultat est imprévisible.
- **`private(liste_variables)` :** Chaque thread obtient sa **propre copie privée** de chaque variable listée. Les modifications faites par un thread sur sa copie privée ne sont **pas visibles** par les autres threads. La valeur initiale de la copie privée est indéterminée (sauf si combiné avec `firstprivate`). La valeur de la variable originale en dehors de la région parallèle n'est pas affectée par les copies privées. Indispensable pour les variables temporaires utilisées dans la boucle. La variable de boucle (`i` dans `for`) est implicitement `private`.

- **`firstprivate(liste_variables)` :** Comme `private`, mais la copie privée de chaque thread est **initialisée** avec la valeur de la variable originale *avant* d'entrer dans la région parallèle.
- **`lastprivate(liste_variables)` :** Comme `private`, mais la valeur de la copie privée du thread qui a exécuté la **dernière itération** (séquentiellement) de la boucle est recopiée dans la variable originale *après* la région parallèle.
- **`reduction(operateur:liste_variables)` :** Très utile pour combiner les résultats partiels calculés par chaque thread en une seule valeur finale. Chaque thread travaille sur une copie privée de la variable, initialisée de manière appropriée (ex: 0 pour `+`, 1 pour `*`). À la fin de la région parallèle, les valeurs privées de tous les threads sont combinées avec l'opérateur spécifié (`+`, `*`, `-`, `&`, `|`, `^`, `&&`, `||`, `min`, `max`) et le résultat final est stocké dans la variable originale partagée. Essentiel pour calculer des sommes, produits, min/max en parallèle sans condition de concurrence.

D'autres clauses existent (`schedule`, `nowait`, `ordered`, `collapse`...) pour un contrôle plus fin.

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

#define N 1000

int main() {
    int somme_globale = 0; // Variable pour accumuler la somme
    int i;
    int a[N];

    // Initialiser un tableau
    for (i = 0; i < N; i++) {
        a[i] = i + 1; // a = {1, 2, 3, ..., 1000}
    }

    // --- Calcul parallèle de la somme ---
    // 'i' est implicitement private dans la boucle for
    // 'somme_globale' est partagée par défaut, mais on utilise 'reduction'
    // pour éviter les race conditions et combiner les résultats
correctement.
    #pragma omp parallel for reduction(+:somme_globale)
    for (i = 0; i < N; i++) {
        // Chaque thread ajoute sa partie des éléments a[i]
        // à sa propre copie privée de somme_globale (initialisée à 0).
        somme_globale += a[i];

        // Exemple avec variable privée explicite
        int thread_id = omp_get_thread_num();
        int valeur_locale = a[i] * 2; // Calcul intermédiaire
        // Si on voulait juste afficher, pas besoin de clause spéciale pour
valeur_locale
        // printf("Thread %d: i=%d, a[i]=%d, valeur_locale=%d\n",
thread_id, i, a[i], valeur_locale);
    }
    // Fin de la boucle: les sommes privées de chaque thread sont
    // additionnées (+) et le résultat final est mis dans la variable
    // originale 'somme_globale'.
```

```
    // Calcul de la somme attendue (formule de Gauss)
    long long somme_attendue = (long long)N * (N + 1) / 2;

    printf("Somme calculée en parallèle : %d\n", somme_globale);
    printf("Somme attendue (séquentiel): %lld\n", somme_attendue);

    if (somme_globale == somme_attendue) {
        printf("Le résultat parallèle est correct !\n");
    } else {
        printf("ERREUR: Le résultat parallèle est incorrect !\n");
    }

    return 0;
}

/* Compilation (GCC/Clang):
   gcc -Wall -fopenmp exemple_clauses.c -o exemple_clauses
   Exécution :
   ./exemple_clauses
*/
```

Sections critiques (`#pragma omp critical`)

Que faire si plusieurs threads doivent **modifier une même ressource partagée** (une variable globale, une structure de données, un fichier...) et que l'opération ne peut pas être facilement exprimée avec une clause `reduction` ? Par exemple, ajouter des éléments à une liste chaînée partagée ou écrire dans un fichier journal commun.

Si plusieurs threads tentent de modifier la ressource partagée en même temps sans protection, on risque une **condition de concurrence (race condition)**, menant à des résultats incorrects ou à une corruption des données.

La directive `#pragma omp critical [(nom)]` définit une **section critique**. Elle garantit qu'**un seul thread à la fois** peut exécuter le bloc de code qui suit immédiatement la directive. Si un thread arrive à une section critique alors qu'un autre thread s'y trouve déjà, il devra attendre que le premier thread ait terminé le bloc critique.

Syntaxe : `#pragma omp critical [(nom_optionnel)] { // Bloc de code à exécuter en exclusion mutuelle // (Un seul thread à la fois ici) variable_partagee = ...; // ... }`

- `(nom_optionnel)` : Vous pouvez donner un nom (optionnel) à la section critique. Toutes les sections critiques *portant le même nom* dans le programme partagent la même exclusion mutuelle. Les sections critiques *sans nom* partagent toutes une seule et même exclusion globale. Donner des noms permet d'avoir différentes "serrures" pour protéger différentes ressources indépendantes, ce qui peut améliorer la performance en réduisant l'attente inutile.

Inconvénients : Les sections critiques **sérialisent** l'exécution de cette partie du code (un seul thread à la fois), ce qui limite le gain de performance potentiel du parallélisme. Elles doivent être utilisées **uniquement**

lorsque c'est nécessaire et le bloc de code protégé doit être **aussi court que possible**. Si une grande partie de votre boucle parallèle se trouve dans une section critique, le gain de parallélisme sera faible voire nul. D'autres mécanismes de synchronisation plus fins existent (atomiques, locks...) mais sont plus complexes.

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

#define NUM_THREADS 4
#define NUM_ITERATIONS 100000

int main() {
    long long compteur_partage = 0;
    int i;

    // Définir le nombre de threads à utiliser
    omp_set_num_threads(NUM_THREADS);

    printf("Compteur initial : %lld\n", compteur_partage);
    printf("Incrémentation parallèle de %d * %d fois...\n", NUM_THREADS,
NUM_ITERATIONS);

    #pragma omp parallel // Crée une équipe de threads
    {
        int thread_id = omp_get_thread_num();
        long long compteur_local_thread = 0; // Compteur local pour chaque
thread

        // Chaque thread fait un certain nombre d'incrémentations locales
        for (i = 0; i < NUM_ITERATIONS; i++) {
            compteur_local_thread++;
        }

        // --- Section Critique ---
        // Maintenant, chaque thread ajoute sa somme locale au compteur
partagé.
        // Sans protection, plusieurs threads pourraient lire la même
valeur de
        // compteur_partage, l'incrémenter, puis l'écrire, écrasant les
        // mises à jour des autres (race condition).
        #pragma omp critical (mise_a_jour_compteur) // Nom optionnel
        {
            // Un seul thread à la fois peut exécuter ce bloc
            printf("  Thread %d ajoute %lld au compteur partagé (qui vaut
%lld)\n",
                   thread_id, compteur_local_thread, compteur_partage);
            compteur_partage += compteur_local_thread;
        } // Fin de la section critique, le thread suivant peut entrer.

    } // Fin de la région parallèle, tous les threads se rejoignent

    long long resultat_attendu = (long long)NUM_THREADS * NUM_ITERATIONS;
```

```
    printf("\nCompteur final partagé : %lld\n", compteur_partage);
    printf("Résultat attendu       : %lld\n", resultat_attendu);

    if (compteur_partage == resultat_attendu) {
        printf("Le résultat est correct !\n");
    } else {
        printf("ERREUR: Le résultat est incorrect à cause d'une race
condition probable (si #pragma omp critical est commenté) !\n");
    }

    return 0;
}

/* Compilation (GCC/Clang):
   gcc -Wall -fopenmp exemple_critical.c -o exemple_critical
   Exécution :
   ./exemple_critical
   (Essayez de commenter le #pragma omp critical et de relancer pour voir
l'erreur !)
*/
```

(Optionnel) Autres directives et concepts

OpenMP offre bien plus de fonctionnalités pour des scénarios de parallélisme plus complexes :

- `#pragma omp parallel` **:** Crée une équipe de threads qui exécutent *tous* le bloc de code suivant. Utile pour le parallélisme de tâches où différents threads font des choses différentes. Souvent combiné avec :
  - `#pragma omp for` **:** (Sans `parallel`) Distribue les itérations d'une boucle `for` entre les threads de l'équipe *déjà existante*.
  - `#pragma omp sections` **:** Divise un bloc en plusieurs `#pragma omp section`, chaque section étant exécutée par un seul thread de l'équipe.
  - `#pragma omp single` **:** Spécifie qu'un bloc ne doit être exécuté que par *un seul* thread de l'équipe (le premier qui arrive).
  - `#pragma omp master` **:** Similaire à `single`, mais garantit que c'est le thread *maître* (celui qui a lancé la région parallèle) qui exécute le bloc.
  - `#pragma omp task` **:** Pour le parallélisme de tâches plus dynamique et irrégulier (tâches dépendantes, récursives...).
- **Synchronisation :**
  - `#pragma omp barrier` **:** Point de synchronisation où tous les threads doivent arriver avant que quiconque puisse continuer.
  - `#pragma omp atomic` **:** Protège une opération mémoire simple (comme `x++`, `x += val`) pour qu'elle soit atomique (indivisible), souvent plus efficace qu'une section critique pour ces cas simples.
  - `#pragma omp flush` **:** Assure la cohérence de la mémoire entre les threads pour des variables spécifiques.
  - **Locks (Verrous) :** Fonctions (`omp_init_lock`, `omp_set_lock`, `omp_unset_lock`, `omp_destroy_lock`) pour une synchronisation plus fine basée sur des verrous explicites.

OpenMP est un outil puissant pour introduire le parallélisme dans les programmes C/C++. Commencer avec `#pragma omp parallel for` et les clauses `reduction` ou `critical` couvre déjà de nombreux cas d'usage courants, en particulier pour l'accélération de boucles de calcul intensives.

---

27. Techniques de Débogage et Analyse

Écrire du code C sans erreur est un idéal difficile à atteindre, surtout avec la complexité de la gestion manuelle de la mémoire et des pointeurs. Savoir **déboguer** (trouver et corriger les erreurs) et **analyser** (comprendre les performances et les problèmes potentiels) votre code est donc une compétence absolument essentielle pour tout développeur C sérieux.

Nous avons déjà mentionné le débogage simple avec `printf`, mais il existe des outils et techniques beaucoup plus puissants.

Utilisation avancée d'un débogueur (GDB/LLDB)

Les débogueurs comme **GDB** (GNU Debugger, courant sous Linux) et **LLDB** (inclus avec Clang/Xcode sous macOS) sont des outils en ligne de commande (ou intégrés dans des IDE) qui vous permettent de contrôler l'exécution de votre programme étape par étape et d'inspecter son état interne.

Prérequis : Pour utiliser efficacement un débogueur, vous devez compiler votre code avec l'option `-g`, qui demande au compilateur d'inclure les **symboles de débogage** (informations liant le code machine aux lignes de votre code source, noms de variables, etc.) dans l'exécutable.

```
gcc -Wall -Wextra -std=c11 -g mon_fichier.c -o mon_programme
```

Fonctionnalités clés (exemples avec GDB, LLDB a des commandes similaires) :

- **Lancer le débogueur :** `gdb ./mon_programme` (Vous arrivez à l'invite `(gdb)`)
- **Poser des points d'arrêt (Breakpoints) :** Arrêter l'exécution à un endroit précis.
  - `break nom_fonction` : Arrête au début de la fonction `nom_fonction`.
  - `break nom_fichier.c:numero_ligne` : Arrête à la ligne `numero_ligne` du fichier `nom_fichier.c`.
  - `break numero_ligne` : Arrête à la ligne `numero_ligne` du fichier courant.
  - `info break` : Liste les points d'arrêt actifs.
  - `delete numero_breakpoint` : Supprime un point d'arrêt.
- **Exécuter le programme :**
  - `run [args...]` : Lance l'exécution du programme (avec les arguments de ligne de commande optionnels). L'exécution s'arrêtera au premier point d'arrêt rencontré ou à la fin du programme (ou en cas de crash).
- **Contrôler l'exécution (une fois arrêté) :**
  - `continue` (ou `c`) : Reprend l'exécution jusqu'au prochain point d'arrêt ou la fin.
  - `next` (ou `n`) : Exécute la ligne de code suivante. Si la ligne contient un appel de fonction, **n'entre pas** dans la fonction (l'exécute en une seule étape).
  - `step` (ou `s`) : Exécute la ligne de code suivante. Si la ligne contient un appel de fonction, **entre** dans cette fonction et s'arrête à sa première ligne.
  - `finish` : Continue l'exécution jusqu'à la sortie de la fonction courante.

    - `until numero_ligne` : Continue jusqu'à atteindre la ligne spécifiée (dans la fonction courante).
- **Inspecter les données :**
    - `print expression` (ou `p expression`) : Affiche la valeur de l'`expression` (variable, calcul, appel de fonction simple...). Ex: `p ma_variable`, `p *ptr`, `p i*10`, `p mon_tableau[3]`.
    - `info locals` : Affiche les variables locales de la fonction courante.
    - `info args` : Affiche les arguments de la fonction courante.
    - `whatis variable` : Affiche le type de la variable.
    - `x/[format] adresse` : Examine la mémoire à une adresse donnée. Ex: `x/d &ma_variable` (examine comme décimal), `x/s chaine_ptr` (examine comme chaîne), `x/10xb ptr` (examine 10 octets en hexa).
- **Examiner la pile d'appels (Call Stack) :**
    - `backtrace` (ou `bt`) : Affiche la séquence d'appels de fonctions qui a mené au point d'arrêt actuel (très utile pour comprendre comment on est arrivé là, surtout en cas de crash).
    - `frame numero` (ou `f numero`) : Sélectionne une autre "frame" (un autre appel de fonction) dans la pile pour inspecter ses variables locales/arguments.
    - `up`, `down` : Monte ou descend dans la pile d'appels.
- **Points d'arrêt conditionnels :**
    - `break ... if condition` : Arrête au point d'arrêt uniquement si la `condition` (une expression C) est vraie. Ex: `break main.c:42 if i == 100`.
- **Watchpoints :**
    - `watch expression` : Arrête l'exécution chaque fois que la valeur de l'`expression` (souvent une variable) change. Très utile pour traquer les modifications inattendues.
    - `rwatch expression` : Arrête quand l'`expression` est lue.
    - `awatch expression` : Arrête quand l'`expression` est lue ou écrite.

Maîtriser un débogueur demande de la pratique, mais c'est un investissement extrêmement rentable pour gagner du temps et comprendre les erreurs complexes.

Débogage mémoire avec Valgrind

Les erreurs liées à la gestion de la mémoire (pointeurs invalides, fuites, accès hors limites...) sont parmi les plus courantes et les plus difficiles à détecter en C. Un outil comme **Valgrind** (principalement pour Linux, mais des alternatives existent pour d'autres OS) est quasi indispensable pour traquer ces problèmes.

Valgrind exécute votre programme dans un environnement simulé et surveille *toutes* les opérations liées à la mémoire. Son outil principal, **Memcheck**, peut détecter :

- **Utilisation de mémoire non initialisée :** Lecture de variables locales ou de mémoire allouée (`malloc`) avant qu'une valeur ne leur ait été assignée.
- **Accès mémoire invalides :** Lecture ou écriture en dehors des blocs alloués (sur le tas ou la pile), y compris les dépassements de tampon (buffer overflows).
- **Utilisation de mémoire après libération (`free`) :** Accès via un pointeur suspendu (dangling pointer).
- **Double libération (`free`) :** Appeler `free` deux fois sur le même bloc.
- **Libération invalide :** Appeler `free` sur une adresse qui n'a pas été retournée par `malloc`/`calloc`/`realloc`.
- **Fuites de mémoire (Memory Leaks) :** Détecte les blocs de mémoire alloués dynamiquement qui ne sont plus accessibles (aucun pointeur ne pointe vers eux) mais n'ont pas été libérés avec `free` avant

la fin du programme.

Utilisation de base :

1. **Compiler avec `-g` :** Pour que Valgrind puisse indiquer précisément les lignes de code source où les erreurs se produisent. L'optimisation (`-O2`, etc.) peut parfois interférer avec Valgrind, il est souvent préférable de tester avec `-O0` ou `-O1` si vous suspectez un problème lié à l'optimisation. `gcc -Wall -g mon_programme.c -o mon_programme`
2. **Lancer Valgrind :** Exécutez votre programme *via* Valgrind. `valgrind --leak-check=full --show-leak-kinds=all ./mon_programme [args...]`
   - `valgrind` : La commande.
   - `--leak-check=full` : Active la détection détaillée des fuites mémoire.
   - `--show-leak-kinds=all` : Affiche tous les types de fuites (definitely lost, possibly lost, etc.).
   - `./mon_programme [args...]` : Votre programme et ses arguments éventuels.

Valgrind exécutera votre programme (beaucoup plus lentement) et affichera un rapport détaillé sur `stderr` à la fin, listant toutes les erreurs mémoire détectées, avec la pile d'appels (call stack) au moment de l'erreur. Analyser attentivement ce rapport est crucial pour corriger les bugs mémoire.

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

void fonction_avec_fuite() {
    int *bloc = malloc(10 * sizeof(int));
    // Oubli de free(bloc) avant la sortie de la fonction !
    // Le pointeur 'bloc' est perdu, la mémoire fuit.
    if (bloc) { // Vérifie si malloc a réussi
        bloc[0] = 1; // Utilise la mémoire (OK ici)
    }
    printf("  (Sortie de fonction_avec_fuite)\n");
}

int main() {
    int *ptr1 = NULL;
    int *ptr2 = NULL;
    int non_initialisee;
    int tableau[5];

    // 1. Utilisation de mémoire non initialisée
    int somme = non_initialisee + 10; // ERREUR: 'non_initialisee' n'a pas
de valeur définie
    printf("Somme (avec non_initialisee): %d\n", somme); // Comportement
indéfini

    // 2. Accès hors limites (Buffer Overflow)
    tableau[5] = 99; // ERREUR: Indice valide de 0 à 4. Écrit après le
tableau.
    printf("Tableau[5] (hors limites) assigné.\n");

    // 3. Fuite mémoire
    printf("\nAppel de fonction_avec_fuite...\n");
```

```
    fonction_avec_fuite(); // 40 octets (10*sizeof(int)) seront perdus ici

    // 4. Utilisation après free (Dangling Pointer)
    ptr1 = (int*)malloc(sizeof(int));
    if (!ptr1) return 1;
    *ptr1 = 123;
    printf("\nValeur pointée par ptr1 avant free: %d\n", *ptr1);
    free(ptr1); // Libère la mémoire
    printf("Mémoire de ptr1 libérée.\n");
    // *ptr1 = 456; // ERREUR: Utilisation après free !
    // printf("Valeur pointée par ptr1 après free: %d\n", *ptr1); // ERREUR
!

    // 5. Double free
    ptr2 = (int*)malloc(sizeof(int));
    if (!ptr2) return 1;
    printf("\nptr2 alloué.\n");
    free(ptr2);
    printf("ptr2 libéré une fois.\n");
    // free(ptr2); // ERREUR: Double free ! (Mettre ptr2 = NULL après le
premier free éviterait ça)

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

/* Compilation pour Valgrind:
   gcc -Wall -Wextra -g code_erreurs.c -o code_erreurs

   Exécution avec Valgrind:
   valgrind --leak-check=full --show-leak-kinds=all ./code_erreurs

   => Analyser attentivement la sortie de Valgrind pour voir les erreurs
détectées.
*/
```

Analyseurs Statiques

Contrairement aux débogueurs ou à Valgrind qui analysent le programme pendant son *exécution*, les **analyseurs statiques** examinent directement le **code source** sans l'exécuter, à la recherche de motifs de code suspects, de bugs potentiels, de violations de style ou de bonnes pratiques.

Ils peuvent détecter des erreurs que le compilateur (même avec `-Wall -Wextra`) pourrait manquer, ou des problèmes qui ne se manifestent que dans certaines conditions d'exécution difficiles à reproduire.

Exemples d'outils :

- **`cppcheck`** **:** Un outil open-source populaire et facile à utiliser. `cppcheck --enable=all --inconclusive --std=c11 mon_fichier.c` `cppcheck --enable=all --inconclusive --std=c11 mon_projet/` (analyse tous les fichiers C/C++ du répertoire)

- **Analyseur statique de Clang (`clang --analyze`) :** Intégré au compilateur Clang. Très puissant. `clang --analyze mon_fichier.c`
- **SonarQube / SonarLint :** Plateformes d'analyse de code plus complètes, souvent utilisées en intégration continue.
- **Analyseurs intégrés aux IDE :** De nombreux IDE (CLion, Visual Studio...) intègrent des capacités d'analyse statique.

Types d'erreurs détectées (exemples) :

- Utilisation de variables non initialisées.
- Fuites de mémoire potentielles (ex: `malloc` sans `free` correspondant sur certains chemins).
- Déréférencement de pointeurs potentiellement `NULL`.
- Dépassements de tampon possibles.
- Code mort (inatteignable).
- Erreurs logiques courantes.
- Problèmes de style ou non-respect de certaines règles (ex: MISRA C pour l'embarqué).

Utiliser régulièrement un analyseur statique sur votre code est une excellente pratique pour améliorer sa qualité et sa robustesse.

Profiling (Mesurer les performances)

Lorsque votre programme fonctionne correctement mais est trop lent, vous avez besoin de savoir *où* il passe le plus de temps. C'est le rôle du **profiling**. Un **profileur** (profiler) est un outil qui mesure le temps passé dans chaque fonction ou ligne de code pendant l'exécution de votre programme.

Cela vous permet d'identifier les **goulots d'étranglement** (bottlenecks) – les parties du code qui consomment le plus de temps CPU – et de concentrer vos efforts d'optimisation là où ils auront le plus d'impact.

Outils courants :

- **`gprof` (GNU Profiler) :** Un outil classique sous Linux.
    1. **Compiler et lier avec `-pg` :** Ajoute du code d'instrumentation pour le profiling. `gcc -Wall -g -pg mon_programme.c -o mon_programme`
    2. **Exécuter le programme :** Lancez votre programme normalement. Il s'exécutera un peu plus lentement et générera un fichier `gmon.out` à la fin. `./mon_programme [args...]`
    3. **Analyser le résultat :** Utilisez `gprof` pour afficher le rapport. `gprof ./mon_programme gmon.out > rapport_profiling.txt` Le rapport montre le temps passé dans chaque fonction (profil plat) et le graphe d'appels (qui appelle qui et combien de temps y est passé).
- **`perf` (Linux) :** Un outil très puissant intégré au noyau Linux, basé sur l'échantillonnage (sampling). Moins intrusif que `gprof`.
    1. `perf record -g ./mon_programme [args...]` : Enregistre les données de performance.
    2. `perf report` : Analyse et affiche le rapport interactif.
- **Instruments (macOS) :** Inclus avec Xcode, fournit une interface graphique pour divers outils de profiling (Time Profiler, Allocations, etc.).
- **Visual Studio Profiler (Windows) :** Intégré à Visual Studio, offre des capacités de profiling CPU, mémoire, etc., avec une interface graphique.

- **Valgrind (Callgrind/Cachegrind) :** Valgrind inclut aussi des outils comme `callgrind` (pour profiler les appels de fonction et le temps CPU) et `cachegrind` (pour analyser l'utilisation du cache CPU), bien qu'ils ralentissent considérablement l'exécution.

Le profiling est essentiel pour l'optimisation. N'optimisez pas "à l'aveugle" ; mesurez d'abord pour savoir où se situe le problème de performance.

Maîtriser les techniques de débogage avancées avec un débogueur, utiliser des outils d'analyse mémoire comme Valgrind, vérifier votre code avec des analyseurs statiques et mesurer les performances avec des profileurs sont des compétences qui distinguent un programmeur C débutant d'un développeur expérimenté et capable de produire du code robuste et efficace.

---

28. Bonnes Pratiques et Style de Code

Maintenant que vous avez exploré une grande partie du langage C, y compris des concepts avancés comme les pointeurs, l'allocation dynamique, les bibliothèques externes et le parallélisme simple, il est temps de se concentrer sur les aspects qui transforment un code fonctionnel en un code de **haute qualité** : lisible, maintenable, robuste et bien documenté.

Ces bonnes pratiques ne sont pas des règles strictes du langage (le compilateur ne les impose pas toujours), mais elles sont essentielles pour le développement professionnel, le travail en équipe et la pérennité de vos projets.

Style de code (Code Style) : Lisibilité et Cohérence

Un code source est lu beaucoup plus souvent qu'il n'est écrit. Un style de code **cohérent** et **lisible** est donc primordial. Il n'existe pas un unique "bon" style C (différentes équipes ou projets peuvent adopter des conventions légèrement différentes, comme K&R, Allman, GNU), mais l'important est de **choisir une convention et de s'y tenir** tout au long d'un projet.

Points clés pour un bon style :

- **Indentation :** Utilisez systématiquement soit des tabulations, soit un nombre fixe d'espaces (2, 4 ou 8 espaces sont courants) pour indenter les blocs de code `{...}` et les lignes dépendantes (après un `if`, `for`, etc.). La cohérence est essentielle. **Ne mélangez jamais tabulations et espaces pour l'indentation.** La plupart des éditeurs modernes peuvent être configurés pour gérer cela automatiquement.
- **Placement des Accolades (`{}`) :** Choisissez un style et soyez cohérent.
  - *Style K&R (souvent vu en C) :* Accolade ouvrante sur la même ligne que l'instruction (`if`, `while`, `for`, définition de fonction).
  - *Style Allman :* Accolade ouvrante sur la ligne suivante, alignée verticalement avec l'instruction.
  - **Important :** Utilisez **toujours** des accolades pour les blocs `if`, `else`, `for`, `while`, `do...while`, même s'ils ne contiennent qu'une seule instruction. Cela évite les erreurs d'ambiguïté (comme le "dangling else") et facilite grandement l'ajout futur d'instructions.
- **Noms (Variables, Fonctions, Types, Macros) :**
  - Choisissez des noms **significatifs et descriptifs**. Évitez les noms trop courts (`i`, `j`, `k` sont acceptables pour des compteurs de boucle simples, mais pas pour des variables importantes), ambigus ou trompeurs.

    - Adoptez une convention de nommage et respectez-la :
        - `snake_case` : Mots séparés par des underscores (ex: `nombre_elements`, `calculer_total`). Très courant en C.
        - `camelCase` : Premier mot en minuscule, les suivants avec une majuscule initiale (ex: `nombreElements`, `calculerTotal`). Moins traditionnel en C mais parfois utilisé.
    - Pour les constantes `#define` et les énumérateurs `enum`, la convention quasi universelle est d'utiliser des **majuscules** (`TOUT_EN_MAJUSCULES_AVEC_UNDERSCORES`).
    - Pour les types définis avec `typedef` (structs, enums...), une convention courante est d'ajouter un suffixe `_t` (ex: `Point_t`, `EtatJeu_t`).
- **Espacement et Lignes Vides :** Utilisez des espaces autour des opérateurs (`a = b + c;`) et après les virgules pour améliorer la lisibilité. Utilisez des lignes vides pour séparer logiquement des blocs de code (par exemple, entre les définitions de fonctions ou avant une boucle importante).
- **Longueur des Lignes :** Essayez de limiter la longueur des lignes de code (souvent 80 ou 100 caractères maximum) pour éviter les retours à la ligne disgracieux et faciliter la lecture côte à côte.
- **Organisation du fichier :**
    - Placez les `#include` en haut du fichier (d'abord les headers standard `<...>`, puis les headers du projet `"..."`).
    - Définissez les macros et les types globaux après les includes.
    - Placez les prototypes des fonctions `static` (internes au fichier) avant leur première utilisation ou définition.
    - Regroupez les fonctions liées logiquement.
    - La fonction `main` est souvent placée soit au début (si le programme est simple), soit à la fin du fichier `main.c`.

```c
/**
 * @file traitement_donnees.c
 * @brief Exemple illustrant quelques bonnes pratiques de style.
 * (Ceci est un exemple de commentaire de documentation Doxygen)
 */

#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>

// Constante symbolique
#define VALEUR_MAXIMALE 1000

// Type défini avec typedef
typedef struct {
    int id;
    double valeur;
} Element_t;

// Prototype d'une fonction statique (interne)
static bool verifier_element(const Element_t *element);

/**
 * @brief Traite un tableau d'éléments.
 *
```

```c
 * Calcule la somme des valeurs des éléments valides dans le tableau.
 *
 * @param tab Pointeur vers le tableau d'éléments.
 * @param taille Le nombre d'éléments dans le tableau.
 * @return La somme des valeurs des éléments valides, ou -1.0 en cas
d'erreur.
 */
double traiter_tableau(Element_t tab[], size_t taille) {
    // Vérification des arguments (robustesse)
    if (tab == NULL || taille == 0) {
        fprintf(stderr, "Erreur: Tableau invalide ou taille nulle.\n");
        return -1.0; // Valeur d'erreur
    }

    double somme = 0.0;
    printf("Traitement de %zu éléments...\n", taille);

    // Boucle claire avec accolades
    for (size_t i = 0; i < taille; ++i) {
        // Utilisation d'une fonction interne pour la validation
        if (verifier_element(&tab[i])) { // Passe l'adresse de l'élément
            somme += tab[i].valeur;
        } else {
            // Gestion d'un élément invalide (exemple)
            printf("  -> Élément %zu (ID %d) invalide, ignoré.\n", i,
tab[i].id);
        }
    }

    return somme;
}

/**
 * @brief Vérifie si un élément est valide.
 * @param element Pointeur (constant) vers l'élément à vérifier.
 * @return true si l'élément est valide, false sinon.
 */
static bool verifier_element(const Element_t *element) {
    // Vérifie si le pointeur n'est pas NULL (bonne pratique)
    if (element == NULL) {
        return false;
    }
    // Exemple de condition de validité
    if (element->id < 0 || element->valeur > VALEUR_MAXIMALE) {
        return false; // Invalide
    }
    return true; // Valide
}

// Fonction main pour tester
int main() {
    Element_t mes_donnees[4] = {
        {1, 10.5},
        {2, 2000.0}, // Valeur trop grande
```

```
        {-1, 30.0},  // ID invalide
        {4, 40.2}
    };
    size_t nb_elements = sizeof(mes_donnees) / sizeof(mes_donnees[0]);

    double resultat = traiter_tableau(mes_donnees, nb_elements);

    if (resultat >= 0.0) {
        printf("\nSomme des éléments valides : %.2f\n", resultat);
    } else {
        printf("\nLe traitement a échoué.\n");
    }

    return 0;
}
```

Gestion des erreurs robuste

Comme mentionné dans la section sur le débogage, le C n'a pas d'exceptions. La gestion explicite des erreurs est donc **fondamentale** pour la robustesse.

- **Vérifiez TOUTES les valeurs de retour critiques :** Ne supposez *jamais* qu'un appel à une fonction système ou de bibliothèque (surtout celles qui allouent des ressources ou effectuent des E/S) a réussi. Vérifiez systématiquement si `fopen`, `malloc`, `calloc`, `realloc`, `SDL_CreateWindow`, `SDL_CreateRenderer`, etc., retournent `NULL`. Vérifiez si `fread`, `fwrite`, `fscanf` retournent le nombre d'éléments attendu. Vérifiez si les fonctions retournant un code d'erreur (souvent 0 pour succès, non-zéro ou -1/`EOF` pour échec) indiquent un succès.
- **Utilisez `errno`, `perror` et `strerror` :** Pour les fonctions système/bibliothèque standard, lorsque l'échec est indiqué, utilisez `perror("Description contexte")` pour afficher un message d'erreur système significatif sur `stderr`, ou `strerror(errno)` pour obtenir cette chaîne et la journaliser ailleurs. Cela aide énormément au diagnostic.
- **Stratégie de gestion d'erreur :** Décidez comment votre programme doit réagir en cas d'erreur :
  - **Terminer proprement ?** (Souvent nécessaire pour les erreurs critiques comme l'échec d'allocation mémoire initiale). Libérez toutes les ressources déjà allouées avant de quitter (`free`, `fclose`, `SDL_Destroy...`, `SDL_Quit`...). Utilisez `exit(EXIT_FAILURE);` (de `<stdlib.h>`).
  - **Retourner un code d'erreur ?** Vos propres fonctions devraient retourner des valeurs indiquant le succès ou l'échec (par exemple, 0 ou `true` pour succès, non-zéro ou `false` pour échec, ou un pointeur `NULL`). Documentez clairement ces codes de retour.
  - **Tenter de récupérer ?** Dans certains cas (ex: erreur réseau temporaire), vous pourriez vouloir réessayer l'opération.
  - **Informer l'utilisateur :** Affichez des messages clairs (idéalement sur `stderr`) expliquant ce qui n'a pas fonctionné.
- **Validez les entrées externes :** Ne faites jamais confiance aux données venant de l'utilisateur, de fichiers ou du réseau. Vérifiez les plages, les formats, les tailles avant de les utiliser pour éviter les crashs ou les failles de sécurité.

- **Utilisez `assert` pendant le développement :** Utilisez `assert(condition)` (de `<assert.h>`) pour vérifier les préconditions, postconditions et invariants de votre code (des choses qui *doivent* être vraies si la logique est correcte). Si une assertion échoue, le programme s'arrête immédiatement en indiquant l'erreur. C'est très utile pour détecter les bugs logiques tôt. N'oubliez pas de compiler sans `NDEBUG` pour que les assertions soient actives pendant le test/débogage.

```c
#include <stdio.h>
#include <stdlib.h> // Pour malloc, free, exit, EXIT_FAILURE
#include <errno.h>  // Pour errno
#include <string.h> // Pour strerror
#include <assert.h> // Pour assert

// Fonction qui alloue un tableau et le remplit
// Retourne true en cas de succès (et modifie *tab_ptr et *taille_ptr)
// Retourne false en cas d'erreur
int* creer_tableau_rempli(size_t taille, int valeur_init) {
    // Précondition: la taille doit être positive
    assert(taille > 0);

    int *tableau = NULL;

    // 1. Vérifier l'allocation mémoire
    tableau = (int*)malloc(taille * sizeof(int));
    if (tableau == NULL) {
        perror("Erreur d'allocation dans creer_tableau_rempli");
        return NULL; // Retourner NULL pour indiquer l'échec
    }

    // 2. Remplir le tableau (pas d'erreur possible ici a priori)
    for (size_t i = 0; i < taille; ++i) {
        tableau[i] = valeur_init;
    }

    // Postcondition: le pointeur retourné ne doit pas être NULL
    assert(tableau != NULL);
    return tableau; // Retourner le pointeur vers le tableau alloué
}

int main(int argc, char *argv[]) {
    int *mon_tab = NULL;
    size_t taille_demandee = 5;

    // Appel de la fonction et vérification du retour
    mon_tab = creer_tableau_rempli(taille_demandee, 99);

    if (mon_tab == NULL) {
        fprintf(stderr, "Impossible de créer le tableau. Sortie.\n");
        return EXIT_FAILURE; // Quitter proprement en cas d'erreur critique
    }

    // Utilisation du tableau (on sait qu'il est valide ici)
    printf("Tableau créé avec succès :\n");
```

```c
    for (size_t i = 0; i < taille_demandee; ++i) {
        printf("  mon_tab[%zu] = %d\n", i, mon_tab[i]);
    }

    // Ne pas oublier de libérer la mémoire allouée par la fonction !
    free(mon_tab);
    mon_tab = NULL;
    printf("Mémoire du tableau libérée.\n");

    // --- Exemple avec fopen ---
    FILE *f = fopen("fichier_inexistant.txt", "r");
    if (f == NULL) {
        // Utiliser perror est souvent le plus simple ici
        perror("Erreur lors de la tentative d'ouverture du fichier");
        // On pourrait décider de continuer ou de quitter
        // return EXIT_FAILURE;
    } else {
        // ... utiliser le fichier ...
        fclose(f);
    }


    return EXIT_SUCCESS;
}
```

Documentation du code

Un code bien écrit est souvent auto-explicatif dans une certaine mesure, mais une bonne documentation reste essentielle, surtout pour :

- Les interfaces publiques des modules (fichiers `.h`).
- Les fonctions complexes ou non triviales.
- Les structures de données importantes.
- Les algorithmes spécifiques.

Types de documentation :

- **Commentaires dans le code :**
  - **Commentaires d'en-tête de fichier :** Au début de chaque fichier (`.c` et `.h`), inclure un commentaire indiquant le nom du fichier, une brève description de son contenu/rôle, l'auteur, la date, et éventuellement l'historique des modifications ou la licence.
  - **Commentaires d'en-tête de fonction :** Avant chaque définition de fonction (surtout les fonctions publiques), inclure un commentaire décrivant :
    - Ce que fait la fonction (son but).
    - Ses paramètres (nom, type, signification, préconditions éventuelles).
    - Ce qu'elle retourne (signification de la valeur de retour, y compris les cas d'erreur).
    - Les effets de bord éventuels (modification de variables globales, allocation de mémoire à libérer par l'appelant...).

    - **Commentaires internes :** Expliquer les parties complexes, les choix de conception non évidents, ou la logique métier à l'intérieur des fonctions. Éviter de commenter l'évidence.
- **Outils de génération de documentation (ex: Doxygen) :**
    - Des outils comme Doxygen (www.doxygen.org) peuvent analyser des commentaires formatés spécialement dans votre code source (souvent avec des balises comme `@brief`, `@param`, `@return`) pour générer automatiquement une documentation externe (HTML, LaTeX, PDF...).
    - Adopter un format de commentaire compatible Doxygen pour les en-têtes de fichiers et de fonctions est une excellente pratique pour les projets de taille conséquente.

```c
/**
 * @file geometrie.h
 * @brief Déclarations pour des opérations géométriques 2D simples.
 * @author Votre Nom
 * @date 2025-04-27
 * @version 1.0
 */

#ifndef GEOMETRIE_H_INCLUDED
#define GEOMETRIE_H_INCLUDED

#include <stdbool.h> // Pour bool

/**
 * @struct Point2D_t
 * @brief Représente un point dans un espace 2D avec des coordonnées double
précision.
 */
typedef struct {
    double x; /**< Coordonnée sur l'axe des abscisses. */
    double y; /**< Coordonnée sur l'axe des ordonnées. */
} Point2D_t;

/**
 * @brief Calcule la distance euclidienne entre deux points 2D.
 *
 * @param p1 Le premier point.
 * @param p2 Le second point.
 * @return La distance entre p1 et p2 (double). Retourne -1.0 si un
pointeur invalide est passé (bien qu'ici on passe par valeur).
 * @note Utilise la formule sqrt((x2-x1)^2 + (y2-y1)^2).
 */
double calculer_distance(Point2D_t p1, Point2D_t p2);

/**
 * @brief Alloue dynamiquement un nouveau point et l'initialise.
 *
 * @param x_init La coordonnée x initiale.
 * @param y_init La coordonnée y initiale.
 * @return Un pointeur vers le nouveau Point2D_t alloué sur le tas,
 * ou NULL en cas d'échec d'allocation.
 * @warning L'appelant est responsable de libérer la mémoire retournée avec
```

```
free().
 */
Point2D_t* creer_point(double x_init, double y_init);

#endif // GEOMETRIE_H_INCLUDED

/* --- Dans geometrie.c --- */
#include "geometrie.h"
#include <math.h>   // Pour sqrt, pow
#include <stdlib.h> // Pour malloc, free
#include <stdio.h>  // Pour perror

// Implémentation de calculer_distance (pas besoin de commentaire Doxygen
ici, il est dans le .h)
double calculer_distance(Point2D_t p1, Point2D_t p2) {
    double dx = p1.x - p2.x;
    double dy = p1.y - p2.y;
    return sqrt(pow(dx, 2) + pow(dy, 2));
}

// Implémentation de creer_point
Point2D_t* creer_point(double x_init, double y_init) {
    Point2D_t *nouveau_point = (Point2D_t*) malloc(sizeof(Point2D_t));
    if (nouveau_point == NULL) {
        perror("Erreur d'allocation pour creer_point");
        return NULL; // Échec
    }
    // Initialisation des membres via le pointeur
    nouveau_point->x = x_init;
    nouveau_point->y = y_init;
    return nouveau_point; // Succès, retourne le pointeur
}

// --- Dans main.c ---
/*
#include <stdio.h>
#include <stdlib.h> // Pour free
#include "geometrie.h"

int main() {
    Point2D_t a = {1.0, 1.0};
    Point2D_t b = {4.0, 5.0};
    Point2D_t *c = NULL;

    double dist_ab = calculer_distance(a, b);
    printf("Distance AB: %.2f\n", dist_ab);

    c = creer_point(-2.0, -3.0);
    if (c != NULL) {
        printf("Point C créé à (%.1f, %.1f)\n", c->x, c->y);
        double dist_ac = calculer_distance(a, *c); // Déréférence c pour
passer la structure
        printf("Distance AC: %.2f\n", dist_ac);
        free(c); // Ne pas oublier de libérer !
```

```
            c = NULL;
        }

        return 0;
    }
    */
```

Adopter un style de code cohérent, gérer rigoureusement les erreurs, utiliser efficacement les outils de débogage et d'analyse, et documenter correctement votre code sont des pratiques qui demandent de la discipline mais qui sont absolument essentielles pour passer de l'écriture de petits programmes fonctionnels au développement d'applications C robustes, maintenables et professionnelles.

---

29. Conclusion et Prochaines Étapes

Félicitations ! Vous êtes arrivé au terme de ce cours complet sur le langage C, orienté vers la compréhension et le développement de projets comme un jeu d'échecs graphique avec SDL2 et OpenMP.

Récapitulatif des compétences acquises

Au cours de ce parcours, vous avez exploré un large éventail de concepts, depuis les fondations du langage jusqu'à des techniques plus avancées :

- **Fondamentaux du C :** Types de données, opérateurs, structures de contrôle (conditions, boucles), fonctions, portée des variables.
- **Gestion de la Mémoire :** Pointeurs (simples, doubles), arithmétique des pointeurs, relation avec les tableaux, allocation dynamique (`malloc`, `calloc`, `realloc`, `free`) et pièges associés (fuites, dangling pointers).
- **Types Composites :** Tableaux (1D, 2D), chaînes de caractères (convention C, `<string.h>`, `<ctype.h>`), structures (`struct`), unions (`union`), énumérations (`enum`), `typedef`.
- **Modularité et Compilation :** Organisation du code en fichiers `.c` et `.h`, préprocesseur (`#include`, `#define`, compilation conditionnelle, header guards), compilation séparée et édition de liens (`gcc`, `-c`, `-o`, `-l`, `-L`, `-I`).
- **Systèmes de Build :** Introduction à `Make` pour automatiser la compilation.
- **Entrées/Sorties :** Interaction avec la console (`printf`, `scanf`, `getchar`, `putchar`) et gestion des fichiers (`FILE*`, `fopen`, `fclose`, `fprintf`, `fscanf`, `fgets`, `fread`, `fwrite`, `fseek`, `ftell`, gestion des erreurs `feof`/`ferror`/`perror`).
- **Programmation Graphique (SDL2) :** Initialisation (`SDL_Init`, `IMG_Init`, `TTF_Init`), création de fenêtres (`SDL_Window`) et de rendus (`SDL_Renderer`), cycle de dessin (`RenderClear`, `RenderCopy`, `RenderPresent`), gestion des événements (`SDL_Event`, `SDL_PollEvent`), chargement/affichage d'images (`SDL_Surface`, `SDL_Texture`) et de texte (`SDL_ttf`).
- **Parallélisme (OpenMP) :** Introduction au parallélisme de boucles (`#pragma omp parallel for`), gestion des variables partagées/privées et réductions.
- **Débogage et Analyse :** Importance des avertissements (`-Wall`), utilisation d'un débogueur (GDB/LLDB), analyse mémoire (Valgrind), analyse statique, profiling (gprof).
- **Bonnes Pratiques :** Style de code, gestion robuste des erreurs, documentation.

Vous avez également analysé la structure d'un projet C complexe (le jeu d'échecs) pour voir comment ces différents concepts s'articulent dans une application réelle.

Aller plus loin : Prochaines Étapes

Le C est un langage vaste avec un écosystème riche. Votre apprentissage ne fait que commencer ! Voici quelques pistes pour continuer à progresser :

1. **Pratiquer, Pratiquer, Pratiquer :**
    - **Projets Personnels :** Choisissez un projet qui vous motive (un petit jeu, un outil en ligne de commande, un simulateur simple...) et essayez de le réaliser en C. C'est la meilleure façon de consolider vos acquis.
    - **Plateformes d'Exercices :** Des sites comme HackerRank, LeetCode (section C), Codewars proposent des défis de programmation pour aiguiser vos compétences en algorithmique et en C.
    - **Recoder des Outils Simples :** Essayez de réimplémenter des commandes Unix/Linux basiques comme `ls`, `cat`, `grep` (en version simplifiée) pour comprendre leur fonctionnement interne.
2. **Approfondir la Bibliothèque Standard C :** Explorez plus en détail les fonctions offertes par `<string.h>`, `<stdlib.h>`, `<math.h>`, `<time.h>`, `<ctype.h>`, etc. Comprendre ce qui est déjà disponible vous évitera de réinventer la roue.
3. **Structures de Données et Algorithmes en C :** C'est un domaine fondamental. Implémentez vous-même :
    - Listes chaînées (simples, doubles, circulaires).
    - Piles (stacks) et Files (queues).
    - Tables de hachage.
    - Arbres binaires (recherche, équilibrés comme AVL ou Rouge-Noir si vous êtes motivé).
    - Graphes (représentations, parcours BFS/DFS).
    - Algorithmes de tri (bulle, insertion, sélection, fusion, rapide/quicksort) et de recherche (linéaire, binaire).
4. **Programmation Système :** Si l'interaction avec le système d'exploitation vous intéresse :
    - **POSIX (Linux/macOS/Unix) :** Appels système pour les fichiers (au-delà de `stdio.h` : `open`, `read`, `write`, `close`, `stat`), gestion des processus (`fork`, `exec`, `wait`, `pipe`), signaux, threads (Pthreads), mémoire partagée, sockets (programmation réseau).
    - **WinAPI (Windows) :** L'équivalent Windows pour les appels système, la gestion des fenêtres, des processus/threads, etc. (plus complexe).
5. **Optimisation :** Apprenez à utiliser les profileurs pour identifier les goulots d'étranglement et explorez les techniques d'optimisation spécifiques au C (gestion du cache, vectorisation - intrinsics SIMD, réduction des appels système...).
6. **Explorer d'autres Bibliothèques C :**
    - **Graphisme/Jeu :** Raylib (plus simple que SDL pour débuter), Allegro. Pour la 3D : OpenGL (via des bibliothèques comme GLFW ou SDL), Vulkan.
    - **Réseau :** libcurl (HTTP/FTP...), OpenSSL (sécurité), ZeroMQ (messagerie).

    - **GUI (Interfaces Graphiques Utilisateur) :** GTK+, Qt (bien que souvent utilisées avec C++), Nuklear (GUI immédiat).
    - **Bases de Données :** libsqlite3 (pour SQLite), libpq (PostgreSQL), mysqlclient (MySQL/MariaDB).

7. **Systèmes Embarqués :** Si les microcontrôleurs vous attirent, apprenez les spécificités du développement embarqué (contraintes mémoire/temps réel, cross-compilation, interaction matérielle directe - GPIO, I2C, SPI...).

8. **Contribuer à l'Open Source :** Trouvez un projet C qui vous plaît sur des plateformes comme GitHub, commencez par comprendre le code, corriger des bugs simples signalés, puis proposez des améliorations. C'est une excellente façon d'apprendre de code réel et de collaborer.

9. **Lire du Code C de Qualité :** Étudiez le code source de projets C bien établis et réputés (noyau Linux, Git, Redis, SQLite, Nginx, des parties de la libc...). Analysez comment ils sont structurés, comment ils gèrent la mémoire et les erreurs, et les idiomes utilisés.

Conclusion Finale

Le langage C est un outil puissant et fondamental dans le monde de l'informatique. Sa maîtrise demande du temps, de la pratique et de la rigueur, mais elle vous ouvre les portes d'une compréhension profonde du fonctionnement des ordinateurs et vous donne la capacité de créer des logiciels performants et de bas niveau.

Les concepts et techniques que vous avez appris dans ce cours constituent une base solide. Continuez à coder, à expérimenter, à lire du code, à déboguer et à apprendre. Le voyage du développeur est continu !

Bon code et bonne continuation !
