Langage C avancé

Laurent Bellon

2025-2026

Préparation de l’Environnement de Développement C

Outils essentiels pour coder, compiler et déboguer

L’environnement de développement est un ensemble d’outils indispensables pour écrire, compiler, déboguer et exécuter des programmes en langage C.

Cette présentation explore :

- Le compilateur **GCC**
- L’éditeur de texte **VS Code**
- L’environnement de développement intégré **Code::Blocks**

GCC : Le Compilateur Standard du C

GCC (GNU Compiler Collection) est un compilateur libre, open-source et standard de facto pour le langage C, développé par la Free Software Foundation.

Caractéristiques :

- **Compatibilité étendue :** Linux, Windows, macOS, architectures variées.
- **Performance :** Génère du code machine optimisé.
- **Diagnostics :** Options puissantes pour avertissements et débogage.
- **Intégration :** Facilité d’intégration avec éditeurs et IDE.

Installation rapide :

- **Linux :** `sudo apt install gcc` (souvent préinstallé)
- **Windows :** Inclus dans MinGW ou WSL
- **macOS :** `xcode-select --install` (via Xcode Command Line Tools)

Compilation et exécution de base :

```
# Exemple : un fichier `hello.c`
gcc hello.c -o hello
./hello
```

Le compilateur transforme votre code source en un programme exécutable.

Visual Studio Code : L'Éditeur Léger et Extensible

VS Code est un éditeur de texte léger, gratuit et open-source, très populaire et extensible.

Fonctionnalités clés pour le C :

- **Support multi-langages :** via des extensions, notamment l'extension officielle "C/C++" par Microsoft.
- **Intégration complète :** Débogueur, gestionnaire de tâches (pour la compilation), terminal intégré.
- **Productivité :** Intellisense (autocomplétion et suggestions de code).
- **Interface :** Moderne et hautement personnalisable.

Compilation intégrée avec `tasks.json` : VS Code permet d'automatiser la compilation en un clic, par exemple avec GCC.

```
{
    "version": "2.0.0",
    "tasks": [{
        "type": "shell",
        "label": "compiler C",
        "command": "gcc",
        "args": ["-g", "${file}", "-o", "${fileBasenameNoExtension}"],
        "group": "build"
    }]
}
```

Ce fichier configure VS Code pour appeler GCC automatiquement.

Code::Blocks : L'IDE Complet pour C/C++

Code::Blocks est un Environnement de Développement Intégré (IDE) open-source, conçu spécifiquement pour le développement en C/C++.

Caractéristiques principales :

- **Interface complète :** Regroupe gestionnaire de projets, éditeur de code et compilateur intégré.
- **Débogueur graphique :** Facilite la recherche et la correction d'erreurs.
- **Multi-plateforme :** Disponible sur Windows, Linux et macOS.
- **Configuration simplifiée :** Intégration transparente avec GCC.

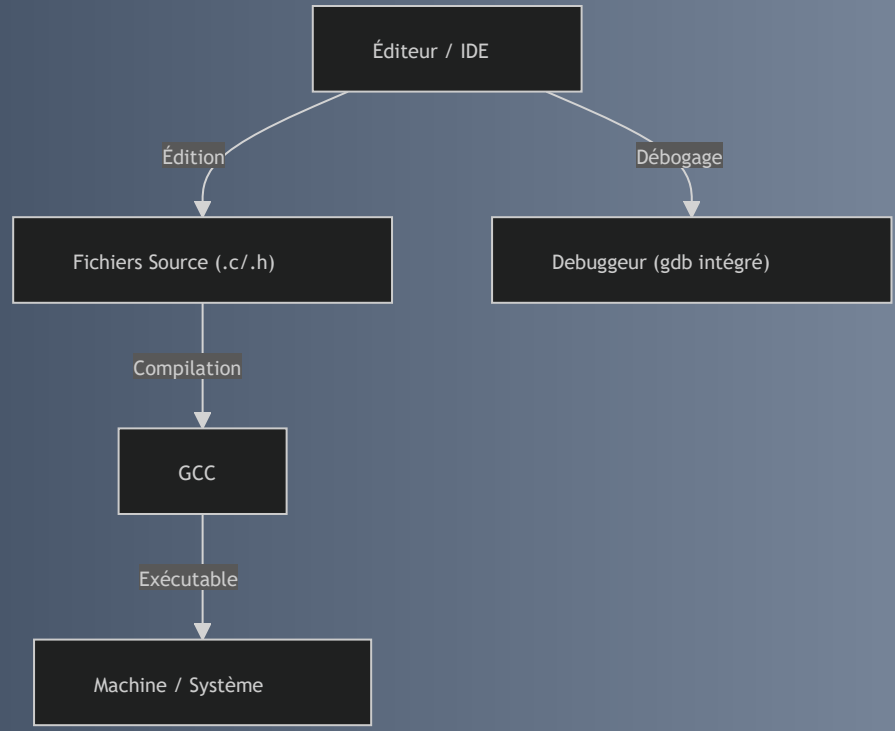
Utilisation :

- Création de projets C/C++ simplifiée.
- Compilation et débogage via des menus graphiques intuitifs.

Idéal pour un développement structuré avec tous les outils à portée de main.

Comprendre l'Écosystème des Outils C

Voici comment les différents outils interagissent au sein de votre environnement de développement :



Résumé des outils :

Outil	Type	Points forts	Usage typique
GCC	Compilateur	Performance, portabilité	Compilation rapide en ligne de commande
VS Code	Éditeur léger	Très personnalisable, extensions	Programmeurs préférant flexibilité
Code::Blocks	IDE complet	Intégration, facilité d'utilisation	Développement structuré projets C

Conclusion & Ressources

Cette présentation fixe les fondations matérielles pour coder efficacement en C à travers des outils modernes polyvalents et accessibles.

Sources utilisées :

- GNU GCC official website - https://gcc.gnu.org/
- Visual Studio Code - https://code.visualstudio.com/docs/languages/cpp
- Code::Blocks - http://www.codeblocks.org/
- TutsMake - How to compile C program using GCC
- Microsoft Docs - VS Code C++ Tutorial

Préparation de l’Environnement de Développement C

Introduction à l’Environnement C

Pourquoi est-ce crucial ?

- L’installation d’un compilateur et d’un IDE est une étape incontournable pour programmer en C.
- Cela permet de transformer votre code source en programme exécutable et de gérer vos projets.

Nous aborderons :

1. L’installation de GCC (le compilateur C incontournable).
2. L’installation de Code::Blocks (un IDE populaire et adapté aux débutants).

Installation de GCC (GNU Compiler Collection)

GCC : Le Compilateur C Incontournable

- Transforme le code source C en exécutable.

1. Sur Windows (via MinGW-w64)

- **Téléchargement :** Site officiel MinGW-w64 (https://www.mingw-w64.org/) ou installateur winlibs.com.
- **Installation :** Choisissez la version (64/32 bits) et l'architecture (x86_64), spécifiez un dossier (ex: `C:\mingw-w64`).
- **PATH :** Ajoutez le chemin du dossier `bin` de MinGW (ex: `C:\mingw-w64\bin`) à la variable d'environnement PATH.
- **Vérification :** Ouvrez `cmd` et tapez `gcc --version`.

2. Sur Linux

- Ouvrez un terminal et lancez :

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

- `build-essential` installe GCC et les outils de compilation.
- **Vérification :** `gcc --version`.

3. Sur macOS

- Installez les outils en ligne de commande Xcode :

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

- Ou via Homebrew :

```
brew install gcc
```

Installation de Code::Blocks (IDE)

Code::Blocks : L'Environnement de Développement Intégré (IDE)

- Open-source, facilite la gestion de projets C/C++.
- Intègre facilement GCC.

1. Sur Windows

- **Téléchargement :** Site officiel Code::Blocks (http://www.codeblocks.org/downloads).
- **Recommandé :** Choisissez le package complet avec MinGW intégré (ex: `codeblocks-20.03mingw-setup.exe`).
- **Installation :** Exécutez le fichier, suivez l'assistant (options par défaut).

2. Sur Linux

- Installer via gestionnaire de paquets :

```
sudo apt install codeblocks
```

- Code::Blocks détectera automatiquement GCC s'il est déjà installé.

3. Sur macOS

- **Téléchargement :** Fichier `.dmg` officiel de Code::Blocks (http://www.codeblocks.org/downloads).
- Suivez les instructions d'installation.

Premier Lancement et Test

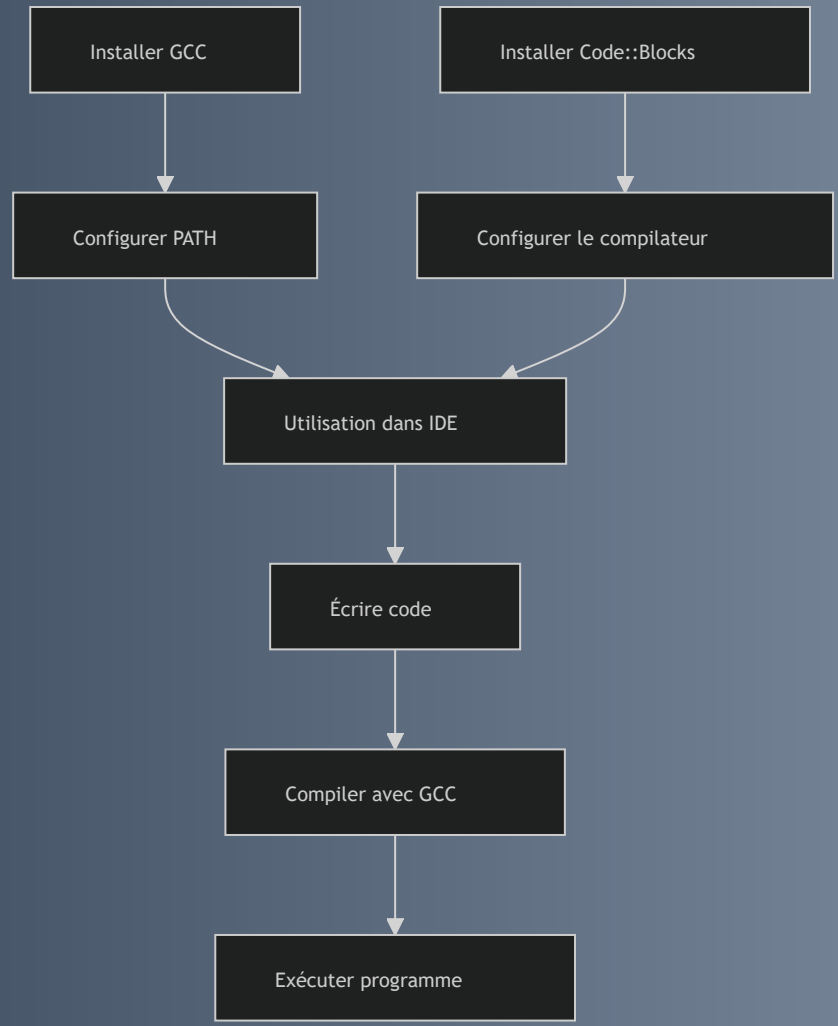
Après l'installation de Code::Blocks (Windows)

1. **Lancement et configuration :**
    1. Lancez Code::Blocks.
    2. Dans `Settings > Compiler`, vérifiez que le compilateur GNU GCC est sélectionné.
2. **Création d'un projet simple :**
    1. `File > New > Project > Console Application > C`.

3. **Compilation et Exécution :**
    1. L'IDE propose deux boutons principaux :
        1. **Build** (ou "Compiler") : transforme le code source en exécutable.
        2. **Run** : exécute le programme généré.

Exemple de code à tester (dans `main.c`) :

```c
#include <stdio.h>

int main() {
    printf("Hello, Code::Blocks and GCC!\n");
    return 0;
}
```


Installer GCC
Installer Code::Blocks
Configurer PATH
Configurer le compilateur
Utilisation dans IDE
Écrire code
Compiler avec GCC
Exécuter programme

Description du flux :

1. **Installation de GCC** : Le compilateur est mis en place.
2. **Configuration du PATH** : Permet au système de trouver GCC.
3. **Installation de Code::Blocks** : L'IDE est installé.
4. **Configuration du compilateur** : L'IDE est connecté à GCC.
5. **Utilisation dans l'IDE** : L'environnement est prêt.
6. **Écriture du code** : Le développeur rédige son programme.
7. **Compilation avec GCC** : L'IDE utilise GCC pour transformer le code en exécutable.
8. **Exécution du programme** : Le programme est lancé.

Conclusion & Ressources

Ce qu'il faut retenir :

- Vous disposez maintenant d'un environnement stable et complet.
- Cet environnement est prêt pour :
  - Écrire du code C.
  - Compiler vos programmes.
  - Exécuter vos applications.
  - Déboguer vos projets.

Sources utilisées :

- GCC official site - https://gcc.gnu.org/install/
- MinGW-w64 - https://www.mingw-w64.org/
- Code::Blocks download - http://www.codeblocks.org/downloads
- Ubuntu Documentation - https://help.ubuntu.com/community/InstallingCompilers
- Homebrew for macOS - https://brew.sh/

Rappels et Approfondissements : Types de données et Opérateurs avancés en C

Le langage C, par sa flexibilité et sa puissance, est fondamental pour le développement de programmes efficaces et performants. Une maîtrise approfondie de ses types de données et opérateurs est la clé pour écrire du code sûr et optimisé.

Les Types de Données : Les Fondations

Comprendre comment C gère les données est crucial.

Types de base :

- `int` : Entier (généralement 4 octets)
- `float` : Nombre à virgule flottante simple précision
- `double` : Nombre à virgule flottante double précision
- `char` : Caractère (1 octet, code ASCII)

Types dérivés et avancés (partie 1) :

- **Tableaux (Arrays)**
  - Séquence contiguë d'éléments de même type.

```
int tab[5] = {1, 2, 3, 4, 5};
```

- **Pointeurs**
  - Variable stockant l'adresse mémoire d'une autre variable. Cœur de la gestion mémoire en C.

```
int a = 10;
int *ptr = &a; // ptr pointe vers a
```

- **Structures (struct)**
  - Regroupement d'éléments hétérogènes sous un même type.

```
struct Point { int x; int y; };
struct Point p1 = {10, 20};
```

Types dérivés et avancés (partie 2) :

- **Unions**
  - Stockent différentes données au même emplacement mémoire (mutuellement exclusives).
- **Enums (énumérations)**
  - Types symboliques représentant des constantes entières nommées.

```
enum Days {LUNDI, MARDI, MERCREDI};
enum Days today = MARDI;
```

```c
#include <stdio.h>

typedef enum {
    DEVICE_OFF,
    DEVICE_ON,
    DEVICE_ERROR
} DeviceStatus;

int main() {
    DeviceStatus status = DEVICE_OFF;

    if (status == DEVICE_OFF) {
        printf("L'appareil est éteint.\n");
    }

    status = DEVICE_ON;

    if (status == DEVICE_ON) {
        printf("L'appareil est allumé.\n");
    }

    return 0;
}
```

```c
#include <stdio.h>

typedef enum {
    TYPE_INT,
    TYPE_FLOAT,
    TYPE_STRING
} ValueType;

typedef union {
    int i;
    float f;
    const char *s;
} Value;

typedef struct {
    ValueType type;
    Value value;
} TypedValue;
```

-->Suite

```c
int main() {
    TypedValue tv;

    tv.type = TYPE_INT;
    tv.value.i = 42;

    if (tv.type == TYPE_INT) {
        printf("Valeur entière : %d\n", tv.value.i);
    }

    tv.type = TYPE_STRING;
    tv.value.s = "Hello world";

    if (tv.type == TYPE_STRING) {
        printf("Valeur chaîne : %s\n", tv.value.s);
    }

    return 0;
}
```

Les Opérateurs Avancés : Manipulation Fine

Opérateurs d'adresse et de pointeur :

- `&` (adresse de) : Récupère l'adresse d'une variable.
- `*` (déréférencement) : Accède à la valeur pointée par un pointeur.

```
int a = 5;
int *p = &a;      // p stocke l'adresse de a
printf("%d\n", *p); // affiche 5
```

Opérateurs bit à bit : Manipulent les bits d'une valeur entière, essentiels en programmation bas niveau.

- `&` (ET bit à bit)
- `|` (OU bit à bit)
- `^` (OU exclusif bit à bit)
- `~` (complément à un bit à bit)
- `<<` (décalage à gauche)
- `>>` (décalage à droite)

```
unsigned char a = 5;       // binaire: 00000101
unsigned char b = a << 1;  // décalage à gauche → 00001010 = 10
```

Les Opérateurs Avancés : Logique et Contrôle

Opérateur ternaire : Forme compacte d'un `if-else` qui retourne une valeur.

```
int y = (x > 0) ? 1 : -1;
```

Opérateurs d'incrémentation et décrémentation (`++` , `--`) : Augmentent ou diminuent la valeur d'une variable de 1. Leur position (préfixe `++i` vs postfixe `i++`) impacte l'ordre d'évaluation.

Opérateurs de conversion de type (cast) : Forcent la conversion d'un type en un autre.

```
float f = 3.14;
int i = (int)f; // i vaut 3
```

Exemple global illustratif :

```c
#include <stdio.h>
struct Coord { int x; int y; };
int main() {
    int a = 10; int *p = &a;
    printf("Adr de a : %p\n", (void*)p); // Pointeur
    printf("Val pointée : %d\n", *p); // Déréférencement

    struct Coord pt = {5, 7}; struct Coord *ppt = &pt;
    printf("Coords : (%d, %d)\n", ppt→x, ppt→y); // Structure & Pointeur

    unsigned char val = 1 << 3; // Décalage bit à bit
    printf("1 << 3 = %d\n", val);

    int b = -10; int signe = (b ⩾ 0) ? 1 : -1; // Ternaire
    printf("Signe de b : %d\n", signe);
    return 0;
}
```

Ce qu’il faut retenir et nos sources

Ce qu’il faut retenir : Ce cours fournit un cadre solide sur les types de données et opérateurs avancés en C. Ces bases sont indispensables pour aborder des sujets plus complexes comme la gestion mémoire, les algorithmes performants, et les structures de données personnalisées, ouvrant la voie à des développements plus sophistiqués et robustes.

Sources utilisées :

- Descriptif Programmation C 2024-2025 - Scribd
- Langage C avancé - ENSTA (PDF)
- Comprendre les types et opérateurs en C - Blog Alphorm
- C documentation et références traditionnelles (ISO C standards)

Rappels et Approfondissements: Portée des variables (Storage Classes) en langage C

En langage C, la portée, la durée de vie (lifetime) et la visibilité des variables sont régies par les classes de stockage (« storage classes »). Ces classes définissent où la variable est accessible, combien de temps elle existe en mémoire, et si elle est visible en dehors d'un fichier source. Comprendre ces concepts permet d'écrire un code clair, efficace et évite des erreurs liées à la gestion des variables.

Définitions clés

- **Portée (Scope)** : Région du programme où une variable est accessible.
- **Durée de vie (Lifetime)** : Temps pendant lequel une variable conserve une valeur en mémoire.
- **Lien (Linkage)** : Visibilité d'un symbole (variable/fonction) entre plusieurs fichiers sources.

L'ensemble de ces caractéristiques sont définies via les classes de stockage suivantes.

`auto`

- Par défaut, toutes les variables locales.
- Durée : vie du bloc dans lequel elles sont définies (ex: une fonction).
- Pas de visibilité hors du bloc.

Exemple `auto` :

```c
void fonction() {
    int val = 5;  // auto implicite
    printf("%d\n", val);
}
```

`register`

- Indique au compilateur de suggérer un stockage dans un registre CPU (accès rapide).
- Ne peut pas avoir l'adresse prise via `&` (généralement).
- Durée de vie similaire à `auto`.

Exemple `register` :

```c
void boucle() {
    register int i;
    for(i = 0; i < 10; i++) {
        printf("%d ", i);
    }
}
```

`static`

- **Variable locale :** Conserve sa valeur entre plusieurs appels à la fonction. Initialisée une seule fois.
- **Variable globale :** Limite sa visibilité au fichier source (linkage interne).
- **Durée de vie :** celle de tout le programme.

Exemple `static` local :

```c
void compteur() {
    static int count = 0;  // initialisée une seule fois
    count++;
    printf("Compteur = %d\n", count);
}
```

Chaque appel à `compteur` affichera une valeur incrémentée.

`extern`

- Déclare une variable ou fonction définie dans un autre fichier source.
- Elle n’est pas allouée dans le fichier où elle est déclarée.
- Le linkage est externe.

Exemple `extern` :

Dans fichier1.c :

```c
int valeur = 42;
```

Dans fichier2.c :

```c
extern int valeur;
void afficher() {
    printf("%d\n", valeur);
}
```

Les Classes de Stockage en C

Classe de stockage	Portée	Durée de vie	Linkage	Description
auto	Locale (bloc)	Durée automatique	Aucun	Variable locale classique, créée à l'entrée du bloc, détruite à la sortie. Par défaut pour les variables locales.
register	Locale (bloc)	Durée automatique	Aucun	Variable locale suggérée pour stockage dans registre CPU, accès rapide. Variable locale avec possibilité d'optimisation.
static	Locale ou globale	Persistante (durée du programme)	Aucun (locale) ou Interne (globale)	Local statique : variable locale qui conserve sa valeur entre les appels. Globale statique : variable ou fonction visible seulement dans le fichier source (linkage interne).
extern	Globale	Persistante (durée du programme)	Externe	Variable/fonction définie dans un autre fichier ou dans le même fichier, déclarée pour usage externe.

```c
#include <stdio.h>

int global_var = 100;       // Durée persistante, linkage externe par défaut
static int static_var = 200; // Durée persistante, linkage interne

void demo() {
    auto int auto_var = 10;     // portée locale, durée automatique
    register int reg_var = 20;  // portée locale, durée automatique

    static int static_local = 0; // durée persistante, portée locale
    static_local++;
    printf("auto_var=%d, reg_var=%d, static_local=%d\n", auto_var, reg_var, static_local);
}

int main() {
    extern int global_var;
    printf("global_var=%d, static_var=%d\n", global_var, static_var);

    demo();
    demo();
    demo();

    return 0;
}
```

Sources utilisées

- Storage Classes in C - GeeksforGeeks
- Module 8 C - Storage Classes, UHCL
- Storage classes in C - Augustine Joseph - Medium
- Scope, Visibility and Lifetime of a Variable in C - Scaler Topics

Ce qu’il faut retenir

Ce cours synthétise les notions fondamentales de portée et durée de vie en C à travers les classes de stockage, afin de clarifier comment les variables sont gérées par le compilateur et exécutées en mémoire.

Qualificatifs `volatile` et `restrict`

- `volatile` et `restrict` : deux qualificatifs C spécifiques.
- `volatile` : Indique au compilateur de ne pas optimiser certaines variables, souvent liées au matériel ou à des contextes multithreading.
- `restrict` : Aide le compilateur à optimiser l'accès mémoire sous certaines conditions d'absence d'aliasing.
- Comprendre ces qualificatifs est essentiel pour écrire un code fiable et performant.

`volatile` : Interdire l'Optimisation du Compilateur

- **Définition :**
  - Prévient le compilateur que la valeur de la variable peut changer à tout moment **en dehors** du flux du programme (matériel, interruption, autre thread).
  - Le compilateur doit toujours relire la valeur depuis la mémoire et ne pas optimiser les accès.
- **Utilisations typiques :**
  - Accès aux registres matériels (ex: ports d'E/S).
  - Variables partagées entre threads (sans protection explicite).
  - Variables modifiées dans une interruption ou un signal.
- **Exemple Clé :**

```
volatile int flag = 0; // modifiée par une interruption
// ...
while(flag == 0) {
    // attente active, lecture forcée à chaque boucle
}
```

Sans `volatile`, le compilateur pourrait optimiser la boucle en supposant que `flag` ne change pas, menant à une boucle infinie.

`restrict` : Permettre l'Optimisation Mémoire

- **Définition (Standard C99) :**
  - Indique qu'un pointeur est le **seul moyen validé** d'accéder à la mémoire pointée durant sa portée.
  - Le compilateur peut alors optimiser en assumant qu'il n'y a pas d'aliasing (pas d'accès à la même mémoire via un autre pointeur).
- **Utilisations typiques :**
  - Fonctions manipulant des pointeurs où on peut garantir un accès non aliasé.
  - Optimisation des accès mémoire pour accélérer les boucles, calculs vectoriels, etc.
- **Exemple Clé :**

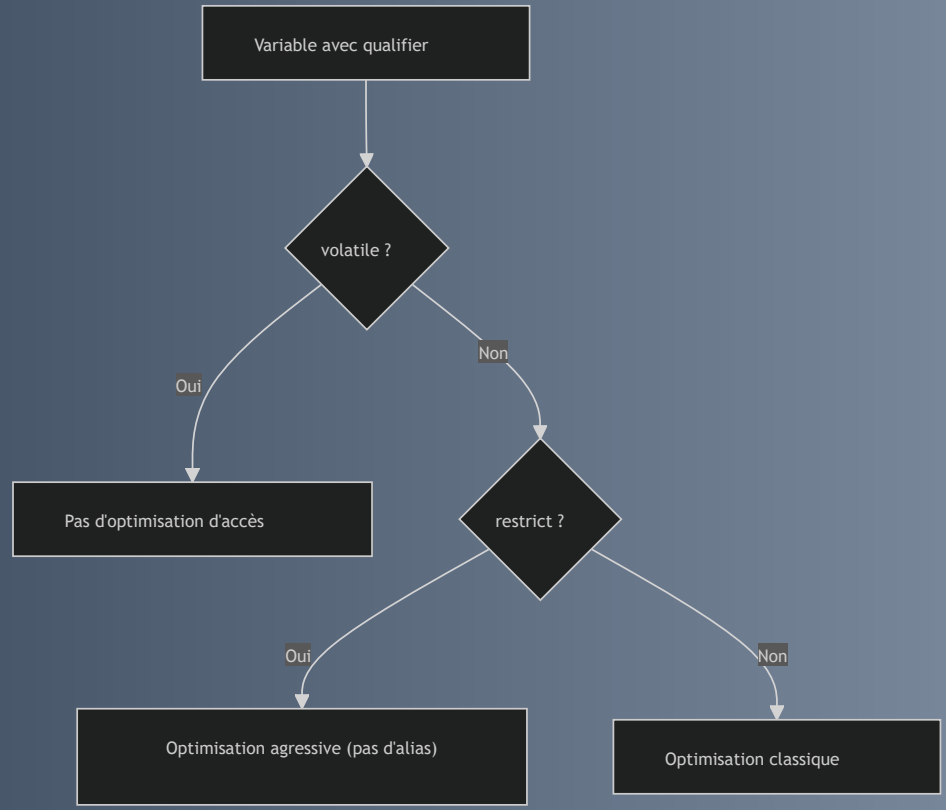
```c
void copy_arrays(int * restrict dest, const int * restrict src, size_t n) {
    for(size_t i = 0; i < n; i++) {
        dest[i] = src[i];
    }
}
```

`restrict` *garantit ici que* `dest` *et* `src` *ne se chevauchent pas, permettant un code plus optimisé.*

`volatile` vs `restrict` : Buts et Impacts

Qualificatif	But principal	Impact sur le compilateur
`volatile`	Interdire l’optimisation des accès mémoire car la variable peut changer de façon externe	Lecture/écriture forcée à chaque accès, minimise optimisation
`restrict`	Permettre au compilateur d’optimiser en précisant absence d’aliasing mémoire	Optimisations agressives possibles, moins de contraintes d’alias

Rôle des Qualificatifs & Notes Essentielles



- **Notes Complémentaires :**
  - `volatile` ne garantit pas la synchronisation entre threads : il faut des primitives spécifiques (mutex, atomic operations).
  - `restrict` doit être utilisé avec précaution : violer son hypothèse (aliasing réel) peut entraîner des comportements indéfinis.
  - Ces keywords sont orthogonaux et peuvent être combinés : `volatile int * restrict p;`

Ce qu’il faut retenir & Ressources

- **Points Clés :**
  - `volatile` et `restrict` sont des outils essentiels pour maîtriser la gestion mémoire.
  - Ils permettent d’optimiser le code et d’interagir avec le matériel ou des systèmes multi-threadés.
- **Sources Consultées :**
  - Qualifiers (const, volatile, restrict) en C - OpenClassrooms
  - Use of volatile and restrict Keywords - GeeksforGeeks
  - Understanding C’s volatile keyword - Embedded.com
  - ISO/IEC 9899:1999 - Programming languages — C

Rappels et Approfondissements : Les Standards C (C11, C17, C23)

Évolution du langage C : principales nouveautés (alignement, `_Generic`, threads C11)

Depuis la publication originelle du standard C (C89/90), le langage C a continué d'évoluer avec des mises à jour majeures : C11, C17, et C23. Ces standards ont introduit des fonctionnalités visant à améliorer la sécurité, la portabilité, la gestion de la concurrence et la flexibilité du langage, tout en conservant sa simplicité et sa performance.

Nous allons détailler ici quelques-unes des nouveautés clés introduites en C11, notamment :

- Les attributs d'alignement (`_Alignas` , `_Alignof`).
- Le mot-clé `_Generic` pour une forme de polymorphisme.
- La bibliothèque standard pour la gestion des threads.

Évolution du langage C : Standards récents

Standard	Année	Description courte
C11	2011	Ajout de la gestion des threads, `_Generic`, alignement mémoire avec `_Alignas` et `_Alignof`.
C17	2017	Correction de bugs, clarifications, pas de nouveautés majeures.
C23	à paraître / adoption progressive	Améliorations syntaxiques, fonctions supplémentaires, meilleur support Unicode, etc.

Alignement mémoire (`_Alignas` et `_Alignof`)

But de l'alignement L'alignement garantit qu'une variable est placée à une adresse mémoire multiple d'un certain nombre. Cela peut améliorer les performances d'accès et éviter des défauts matériels.

Nouveautés C11

- `_Alignas` : Permet de spécifier explicitement l'alignement d'une variable ou d'un type.
- `_Alignof` : Récupère l'alignement requis pour un type donné.

Exemple

```c
#include <stdio.h>
#include <stdalign.h> // depuis C11

struct S {
    char c;
    _Alignas(16) int x;
};

int main() {
    printf("Alignement de int : %zu\n", _Alignof(int));
    printf("Alignement de struct S : %zu\n", _Alignof(struct S));

    struct S s;
    printf("Adresse de s.x : %p\n", (void*)&s.x);
    return 0;
}
```

Le mot-clé `_Generic` : polymorphisme statique (C11)

Introduit en C11, `_Generic` permet d'écrire des macros qui choisissent un comportement ou une fonction selon le type de l'argument. Cela rapproche C d'un polymorphisme statique, similaire à ce qu'on trouve en C++.

Exemple d'utilisation

```c
#include <stdio.h>

#define print_type(x) _Generic((x), \
    int: printf("int: %d\n", x), \
    float: printf("float: %f\n", x), \
    double: printf("double: %f\n", x), \
    default: printf("Type inconnu\n"))

int main() {
    print_type(3);           // Affiche "int: 3"
    print_type(3.14f);       // Affiche "float: 3.140000"
    print_type(2.71828);     // Affiche "double: 2.718280"
    print_type("texte");     // Affiche "Type inconnu"
    return 0;
}
```

Ce code choisit la bonne fonction `printf` en fonction du type de la variable passée à la macro.

Threads en C11 : prise en charge standard

Bibliothèque `<threads.h>` C11 introduit une interface standardisée pour la création et la gestion de threads, de la synchronisation (mutex, variables condition) et d'autres primitives de concurrence.

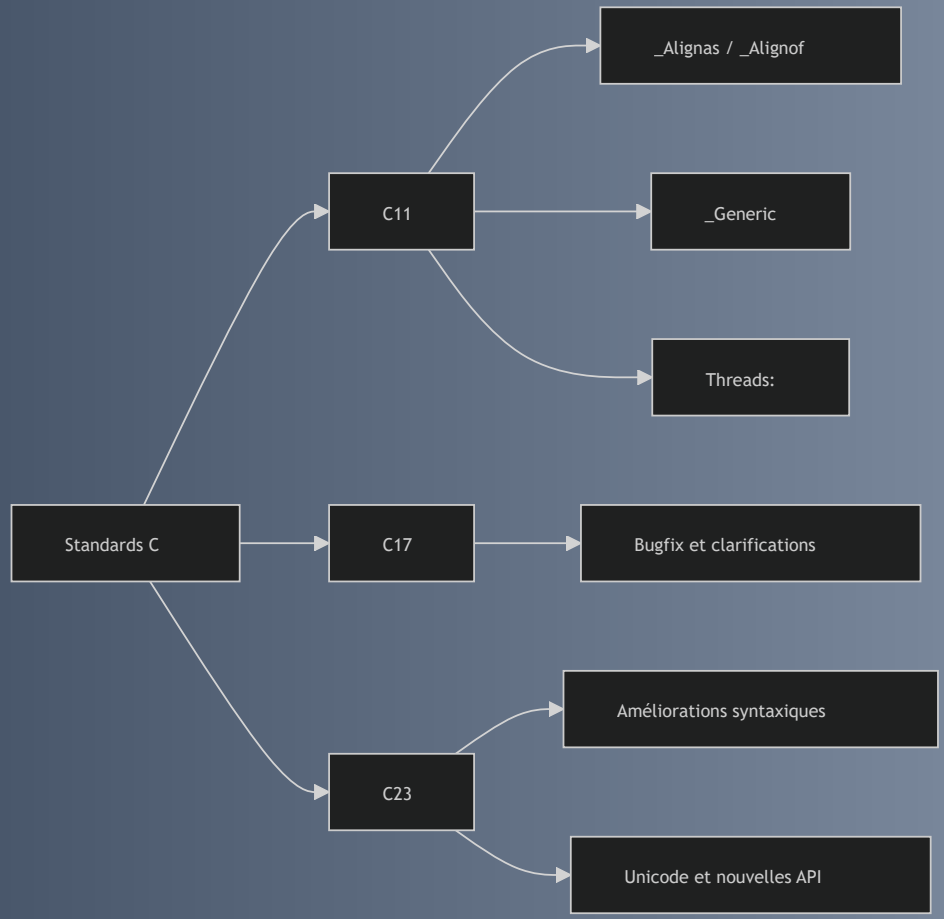
Fonctions principales

- `thrd_create` : crée un nouveau thread.
- `thrd_join` : attend la fin d'un thread.
- `mtx_init`, `mtx_lock`, `mtx_unlock` : mutex.
- `cnd_wait`, `cnd_signal` : variables condition.

Exemple de création simple d'un thread

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

int thread_func(void *arg) {
    int *num = (int*)arg;
    printf("Hello from thread %d\n", *num);
    return 0;
}

int main() {
    thrd_t thread;
    int arg = 42;

    if (thrd_create(&thread, thread_func, &arg) ≠ thrd_success) {
        printf("Erreur création thread\n");
        return 1;
    }

    thrd_join(thread, NULL);
    return 0;
}
```

Points Clés et Ressources



Ce qu’il faut retenir Les évolutions modernes de C offrent aux développeurs des outils pour écrire du code plus robuste, performant et adapté aux environnements concurrents, tout en conservant l’esprit du langage.

Sources utilisées

- Documents officiels ISO C - Standard C11
- C11 Standard Library - cppreference
- Understanding C11 Threads - IBM Developer
- C11 _Generic explained - GeeksforGeeks
- C11 alignment - cppreference
- C23 draft and proposals - WG14

Rappels et Approfondissements : Les Standards C (C11, C17, C23)

Améliorations du mot-clé `const` en C23 et nouvelles fonctions de bibliothèque

Avec la standardisation récente du C23, le langage C a intégré plusieurs améliorations visant à renforcer la sécurité, la lisibilité du code et à enrichir la bibliothèque standard. Parmi celles-ci, des évolutions notables concernent l'utilisation du qualificatif `const` ainsi que l'ajout de nouvelles fonctions pratiques dans la bibliothèque.

`const` en C23 : Contexte et Améliorations

Problème antérieur

- Le mot-clé `const` indique qu'une variable ne doit pas être modifiée après son initialisation.
- Avant C23, son usage dans certains contextes (pointeurs composés, conversions) était limité ou source d'ambiguïtés.

Nouvelles règles de qualification `const`

- Le standard C23 introduit des règles plus souples et précises concernant la propagation de `const` et la conversion entre pointeurs qualifiés.
- **Bénéfices :**
  - Conversions plus sûres entre pointeurs `const` imbriqués (ex: `const int *` vers `const int * const *`).
  - Meilleure prise en charge des expressions constantes dans des contextes plus variés.

`const` en C23 : Exemple illustratif

```
const int a = 10;
const int *p1 = &a;
const int * const *p2 = &p1;  // Conversion désormais mieux supportée en C23
```

Avant C23, certains compilateurs pouvaient rejeter cette conversion stricte à cause d'une qualification `const` "trop profonde". Les nouvelles règles en C23 apportent une flexibilité bienvenue et évitent des erreurs de compilation inutiles.

Nouvelles fonctions de la bibliothèque standard C23

Fonctions d'aide à la manipulation mémoire

- `memset_explicit` : Version de `memset` garantie non optimisable, utile pour effacer des zones sensibles (ex : clés cryptographiques) en évitant que le compilateur supprime ce nettoyage.

```
void *memset_explicit(void *ptr, int value, size_t num);
```

Fonctions temporelles

- Ajout de nouvelles fonctions pour une meilleure gestion du temps, incluant :
  - `timespec_getres()` : Récupérer la résolution d'un objet temporel.
  - Extensions pour faciliter le travail avec `struct timespec`.

Fonctions d'analyse de chaîne et conversion

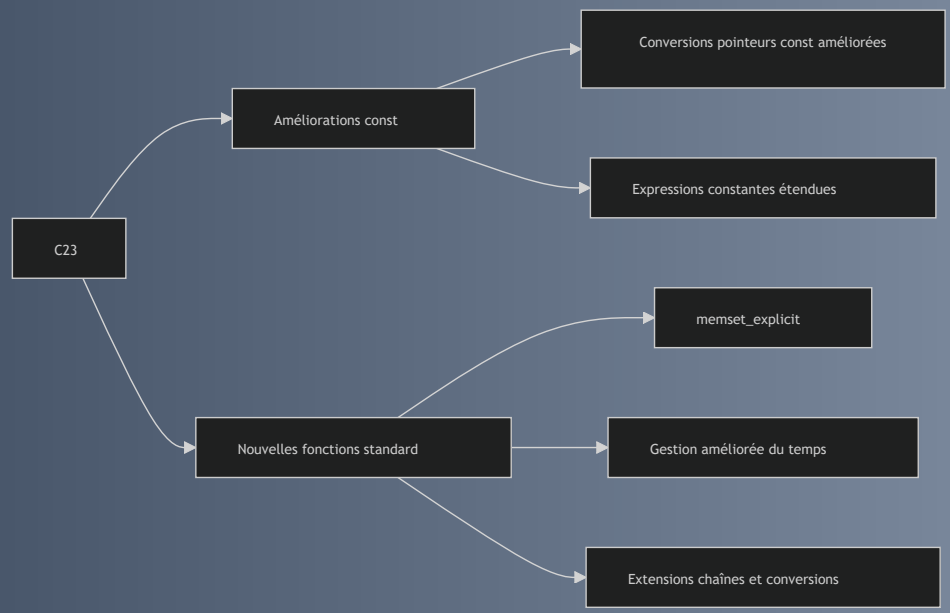
- Quelques ajouts qui simplifient le traitement de texte et conversions dans `<string.h>` et `<stdlib.h>`.

`memset_explicit` : Un exemple concret

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

int main() {
    char password[16] = "secretPass123";
    printf("Mot de passe avant nettoyage : %s\n", password);

    memset_explicit(password, 0, sizeof(password));  // Nettoyage mémoire garanti

    printf("Mot de passe après nettoyage : %s\n", password);  // Affiche chaîne vide
    return 0;
}
```

Cette fonction s'assure que le compilateur ne supprimera pas l'appel à `memset`, contrairement à `memset` classique dont l'appel pourrait être optimisé ou supprimé si la variable n'est plus utilisée ensuite, un aspect crucial pour la sécurité des données sensibles.

Synthèse et Ressources



Cette présentation met en lumière les avancées majeures introduites dans le standard C23 pour la gestion améliorée de `const` — offrant plus de flexibilité et de sécurité dans le typage — ainsi que les extensions fonctionnelles en bibliothèque qui facilitent le développement moderne.

Sources utilisées :

- ISO/IEC 9899:2023 Draft
- C23 new features - GitHub: gcc wiki
- memset_explicit introduction - Microsoft Documentation
- Exploring the new features of C23 - Embedded Artistry
- Stack Overflow - Discussion sur const et C23

Gestion Avancée de la Mémoire : Allocation Dynamique

Rappels et Erreurs Courantes (fuites mémoire, double free)

Introduction

La gestion dynamique de la mémoire en C permet d'allouer et libérer la mémoire au moment de l'exécution (runtime), par opposition à la mémoire statique ou automatique. Elle utilise les fonctions standards `malloc`, `calloc`, `realloc` et `free`.

Une mauvaise utilisation peut entraîner des problèmes classiques :

- **Fuites mémoire** : mémoire non libérée.
- **Double free** : libération multiple d'un même bloc.

Ce cours synthétise les principes essentiels pour éviter ces erreurs.

Principales fonctions d’allocation dynamique

Fonction	Description	Particularités
`malloc`	Alloue un bloc mémoire brut de taille indiquée (non initialisée).	Retourne un pointeur `void*` sur la mémoire allouée, ou `NULL` si échec.
`calloc`	Alloue un bloc pour `n` éléments de taille `size` et initialise à 0.	Plus sûr si besoin d’un buffer initialisé à zéro.
`realloc`	Redimensionne un bloc mémoire précédemment alloué.	Peut déplacer le bloc, retourne un nouveau pointeur, `NULL` si échec.
`free`	Libère un bloc mémoire précédemment alloué.	Ne supprime pas la variable, juste la mémoire.

Utilisation et Exemples : `malloc` et `free`

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

int main() {
    int *tab = (int *)malloc(5 * sizeof(int));
    if (tab == NULL) {
        perror("malloc échoué");
        return 1;
    }
    for (int i = 0; i < 5; i++) {
        tab[i] = i * i;
        printf("%d ", tab[i]);
    }
    printf("\n");

    free(tab); // Libération de la mémoire
    return 0;
}
```

Utilisation et Exemples : `calloc` et `realloc`

`calloc`

```c
int *tab_zero = (int *)calloc(5, sizeof(int));
if (tab_zero == NULL) {
    perror("calloc échoué");
    return 1;
}
// tab_zero est initialisé à zéro
free(tab_zero);
```

`realloc`

```c
int *tab_resize = (int *)malloc(3 * sizeof(int));
// ... initialisation ...
tab_resize = (int *)realloc(tab_resize, 6 * sizeof(int));
if (tab_resize == NULL) {
    perror("realloc échoué");
    // Attention: éviter fuite mémoire si realloc échoue
}
// ... initialiser nouvelle zone ...
free(tab_resize);
```

Erreurs Courantes et Cycle de Vie

1. Fuites mémoire

Oubli de `free` dans un programme de longue durée = consommation excessive mémoire.

```
int *p = malloc(10 * sizeof(int));
// utilisation mais pas de free → FUITE MÉMOIRE
```

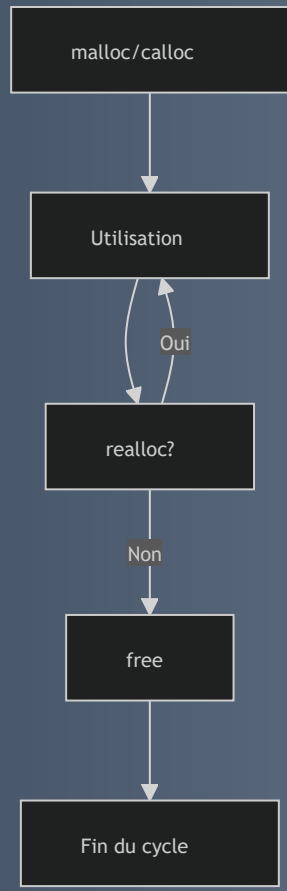
Remarque : un programme court qui termine libère généralement la mémoire, mais ce n'est pas une bonne pratique.

2. Double free

Libération d'un même bloc plusieurs fois = comportement indéfini et souvent crash.

```
int *p = malloc(10 * sizeof(int));
free(p);
// free(p); // ERREUR : double free
```

Diagramme : Cycle de vie d’un bloc mémoire



Bonnes Pratiques & Conclusion

Bonnes pratiques

- Toujours vérifier le résultat de `malloc/calloc/realloc` (!= NULL).
- Initialiser la mémoire après allocation si nécessaire (sauf avec `calloc`).
- Après `free(ptr)`, il est recommandé de faire `ptr = NULL` pour éviter un double free accidentel.
- Éviter de réassigner directement le résultat de `realloc` :

```
int *tmp = realloc(ptr, new_size);
if (tmp ≠ NULL) { ptr = tmp; }
else { /* gestion erreur, ptr reste valide */ }
```

- Utiliser des outils de détection des fuites mémoire (Valgrind, AddressSanitizer).

Sources utilisées

- malloc, calloc, realloc, free - cppreference
- Gestion mémoire en C - OpenClassrooms
- Common Memory Management Errors in C - GeeksforGeeks
- Valgrind Documentation

Ce cours rassemble les mécanismes fondamentaux de la gestion dynamique mémoire en C, ainsi que les erreurs typiques à éviter pour garantir la stabilité et l'efficacité des programmes.

Gestion Avancée de la Mémoire : Allocation Dynamique

Sécuriser vos allocations C avec `malloc`, `calloc`, `realloc`, `free`

L’allocation dynamique de mémoire est essentielle en C pour gérer des données de taille variable. Une gestion efficace repose sur l’utilisation correcte des fonctions d’allocation et l’adoption de stratégies robustes.

Il est crucial de sécuriser le programme par la vérification systématique des retours d’allocation et des techniques adaptées pour limiter la fragmentation et optimiser la mémoire.

Stratégies d’Allocation Mémoire

Lorsqu’une zone mémoire doit évoluer, il est courant d’allouer une capacité initiale, puis de l’agrandir via `realloc` selon les besoins.

Stratégie classique : Doublement progressif On multiplie généralement la capacité par 2 à chaque saturation. Cela limite les appels coûteux à `realloc` (copies de données).

```c
size_t capacity = 4;
size_t size = 0; // éléments utilisés
int *array = malloc(capacity * sizeof(int));
if (array == NULL) { /* gestion d'erreur */ }

for (int i = 0; i < 10; i++) {
    if (size == capacity) {
        size_t new_capacity = capacity * 2;
        int *tmp = realloc(array, new_capacity * sizeof(int));
        if (tmp == NULL) {
            // Erreur : libérer array, exit
            free(array);
            exit(EXIT_FAILURE);
        }
        array = tmp;
        capacity = new_capacity;
    }
    array[size++] = i;
}
```

Allocation par Blocs ou Pools

Pour des allocations répétées d’objets de taille fixe, préallouer une zone (pool) réduit la fragmentation et augmente la rapidité, souvent dans les systèmes embarqués.

Vérification Obligatoire des Retours d'Allocation

Pourquoi vérifier ?

Un échec d'allocation (`malloc`, `calloc`, `realloc`) se traduit par un retour de `NULL`. Utiliser ce pointeur `NULL` sans contrôle entraîne un comportement indéfini, potentiellement un plantage.

Comment vérifier ?

- **Toujours tester** que le pointeur retourné n'est pas `NULL` avant d'utiliser la mémoire.
- En cas d'erreur, **libérer les ressources** et sortir proprement ou prévoir un plan de secours.

Exemple simple

```c
int *ptr = malloc(100 * sizeof(int));
if (ptr == NULL) {
    fprintf(stderr, "Erreur : allocation mémoire échouée\n");
    exit(EXIT_FAILURE); // Quitter le programme proprement
}
// Utilisation de ptr
free(ptr); // Libérer la mémoire après usage
```

Vérification Spécifique pour `realloc`

`realloc` peut retourner `NULL` tout en **conservant le pointeur original valide**.

Erreur à éviter :

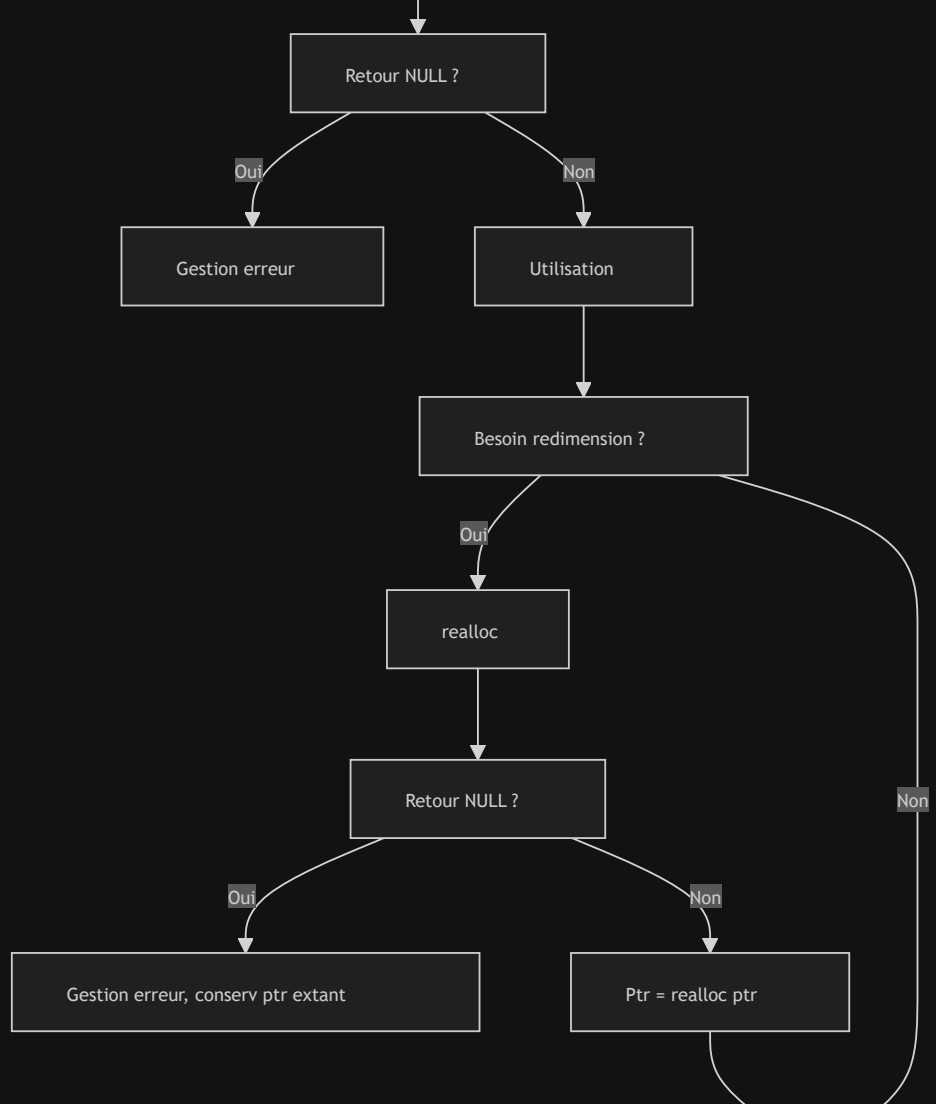
```
ptr = realloc(ptr, new_size); // RISQUE de FUITE si realloc retourne NULL
```

Si `realloc` échoue et retourne `NULL`, la référence à l'ancienne zone mémoire (initialement pointée par `ptr`) est perdue, créant une fuite mémoire car `ptr` est écrasé.

Méthode correcte et sécurisée :

```
int *tmp = realloc(ptr, new_size);
if (tmp == NULL) {
    // Gestion d'erreur
    // ATTENTION : ptr est toujours valide et pointe vers l'ancienne zone !
    // Vous pouvez continuer à utiliser ptr ou le libérer si l'échec est critique.
} else {
    ptr = tmp; // Seulement si realloc a réussi
}
```

Cette approche garantit que la mémoire originale est toujours accessible même si le redimensionnement échoue.


Retour NULL ?
Oui
Non
Gestion erreur
Utilisation
Besoin redimension ?
Oui
Non
realloc
Retour NULL ?
Oui
Non
Gestion erreur, conserv ptr extant
Ptr = realloc ptr

Points Clés & Ressources

Ce qu’il faut retenir

Ce cours a exposé des stratégies d’allocation efficaces et sécurisées, tout en insistant sur la nécessité de vérifier systématiquement les retours d’appel d’allocation mémoire pour garantir la robustesse des programmes en C.

Sources

- Dynamic memory allocation in C - GeeksforGeeks
- C malloc and realloc best practices - Stack Overflow
- Memory Management in C - OpenClassrooms
- Valgrind documentation - Memcheck

Gestion Avancée de la Mémoire :

Pointeurs de Fonctions et `void*`

Introduction

Les pointeurs en C ne servent pas qu'à référencer des données. Ils permettent aussi de :

- Manipuler des fonctions via des **pointeurs de fonctions**.
- Gérer des adresses de mémoire de type inconnu avec des **pointeurs génériques (`void*`)**.

Ces concepts sont essentiels pour la flexibilité et la puissance du langage C. Ce cours vise à clarifier leur syntaxe, leur usage pratique et les précautions nécessaires.

Pointeurs de Fonctions : Définition et Syntaxe

Définition

- Variable qui stocke l'adresse d'une fonction.
- Permet d'appeler la fonction via ce pointeur.
- Mécanisme puissant pour :
  - Callbacks
  - Tables de dispatch dynamiques
  - Polymorphisme limité

Syntaxe de Déclaration

- Pointeur vers une fonction qui prend un `int` et retourne un `int` :

```
int (*ptr_func)(int);
```

  - Les parenthèses autour de `*ptr_func` sont nécéssaires pour indiquer que `ptr_func` est un pointeur.

Pointeurs de Fonctions : Exemple et Usages Typiques

Exemple Simple

```c
#include <stdio.h>

int carre(int x) {
    return x * x;
}

int main() {
    int (*fptr)(int) = carre;  // Affectation de l'adresse
    int val = 5;
    printf("Carré de %d = %d\n", val, fptr(val)); // Appel via pointeur

    return 0;
}
```

Usages Typiques

- **Callbacks** dans des API (ex : fonctions de comparaison pour `qsort`).
- **Tables de fonctions dynamiques** (ex : menu interactif).
- **Implémentations de machines à états**.

Pointeurs Génériques (`void*`) : Définition et Exemple

Définition

- `void*` est un pointeur **sans type**.
- Peut pointer vers n'importe quel type de donnée.
- Utilisé pour écrire des fonctions et structures génériques en C.

Conversion Implicite et Explicite

- Un pointeur typé peut être converti **implicitement** en `void*`.
- Un `void*` doit être **casté explicitement** en pointeur du type attendu avant utilisation.

Exemple

```c
void affiche_int(void *ptr) {
    int *iptr = (int *)ptr; // Cast explicite nécessaire
    printf("%d\n", *iptr);
}

int main() {
    int a = 42;
    affiche_int(&a); // Conversion implicite de &a (int*) en void*
    return 0;
}
```

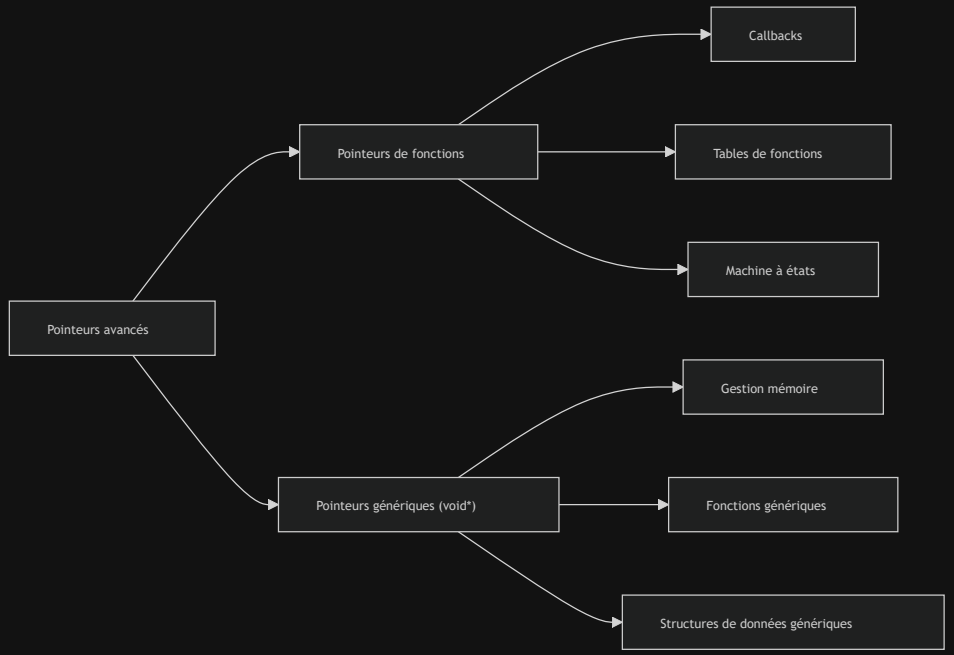
Usages Avancés et Précautions

Utilisations Courantes de `void*`

- Fonctions génériques de gestion mémoire (`malloc` retourne `void*`).
- APIs génériques (`qsort` , `bsearch`).
- Stockage générique dans des structures de données (listes chaînées).

Relation entre les deux types de pointeurs

- Il n’est **pas possible** de convertir directement un pointeur de fonction en `void*` (ou inversement) sans provoquer un comportement indéfini. Ces pointeurs sont distincts.



Précautions et Bonnes Pratiques

- Assurez-vous que le pointeur de fonction pointe vers une fonction compatible avec la signature attendue.
- Ne pas faire de conversions entre pointeurs de fonction et `void*`.
- Avec `void*`, le cast explicite est obligatoire avant l'accès.

Ce qu’il faut retenir et Ressources

En bref : Les pointeurs de fonctions et les pointeurs génériques (`void*`) sont des composants indispensables en C pour une programmation flexible, modulaire et générique. Leur maîtrise permet d’écrire des codes plus adaptables et puissants.

Sources utilisées :

- Pointer to function in C - GeeksforGeeks
- Void pointer in C - Tutorialspoint
- ISO/IEC 9899:2018 - Standard C
- The GNU C Library - Function pointers
- Wikipedia - Void pointer

Gestion Avancée de la Mémoire

Pointeurs Avancés : Tableaux de Pointeurs et Structures

La manipulation de tableaux de pointeurs et de pointeurs pointant sur des structures en C permet une gestion flexible et performante des données complexes. C’est essentiel pour créer des tableaux dynamiques d’objets, gérer des collections hétérogènes, ou construire des structures de données efficaces.

Ce cours explore les concepts, la syntaxe et les bonnes pratiques liées à ces usages majeurs.

1. Les Tableaux de Pointeurs

Concept & Syntaxe Un tableau de pointeurs est un tableau dont les éléments sont des pointeurs.

- **Exemple simple (statique) :**

```
char *jours[] = {"Lundi", "Mardi", "Mercredi", "Jeudi", "Vendredi"};
// jours est un tableau de 5 pointeurs vers des chaînes constantes.
```

Utilisation Dynamique Créer un tableau de pointeurs dont les éléments (les chaînes ici) sont alloués dynamiquement.

- **Méthode :** `malloc` pour le tableau de pointeurs, puis `strdup` pour chaque chaîne.
- **Exemple :**

```
char **noms = malloc(n * sizeof(char*)); // Tableau de pointeurs
noms[0] = strdup("Alice"); // Allocation pour chaque chaîne
// ...
// N'oubliez pas de libérer chaque élément (noms[i]) avant de libérer le tableau (noms).
```

2. Les Pointeurs vers Structures

Déclaration et Accès Permet de manipuler des instances de structures de manière indirecte.

- **Exemple :**

```c
struct Point { int x, y; };
struct Point p = {10, 20};
struct Point *ptr = &p; // ptr pointe sur p

// Accès aux membres via le pointeur
printf("Coordonnées : (%d, %d)\n", ptr->x, ptr->y);
// On utilise l'opérateur "->" pour accéder aux membres via un pointeur.
```

Allocation Dynamique de Structures Créer une structure sur le tas, utile pour des objets de durée de vie variable.

- **Fonction type :**

```c
struct Point *creer_point(int x, int y) {
    struct Point *p = malloc(sizeof(struct Point)); // Allocation
    if (p != NULL) {
        p->x = x; p->y = y;
    }
    return p;
}
// N'oubliez pas de free(p) après utilisation.
```

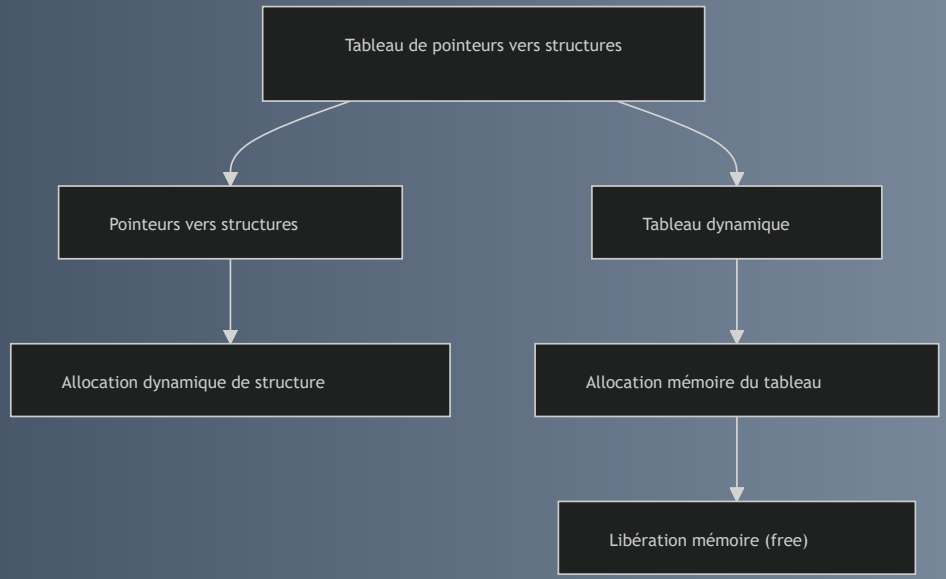
3. Tableaux de Pointeurs vers Structures

Collections Dynamiques d'Objets Combine les deux concepts précédents pour gérer des collections de structures allouées dynamiquement.

- **Exemple :** Un tableau de pointeurs `struct Point *`

```c
int n = 2;
struct Point **tab_points = malloc(n * sizeof(struct Point*));
// tab_points est un tableau de pointeurs vers des structures Point

tab_points[0] = creer_point(1, 2); // Chaque élément pointe vers une structure allouée
tab_points[1] = creer_point(3, 4);

// Accès : tab_points[i]→x

// Libération :
for (int i = 0; i < n; i++) {
    free(tab_points[i]); // Libérer chaque structure
}
free(tab_points); // Libérer le tableau de pointeurs
```

4. Interactions Pointeurs et Structures



Ce diagramme illustre la relation hiérarchique et les dépendances des concepts abordés pour la gestion de structures dynamiques via des pointeurs.

5. Bonnes Pratiques & Pour Aller Plus Loin

Bonnes Pratiques Essentielles

- **Vérifiez toujours :** Le résultat des `malloc` et `strdup` doit être contrôlé (`≠ NULL`).
- **Libération mémoire :** Assurez-vous de libérer *toutes* les zones mémoire allouées (structures, chaînes, tableaux).
- **Opérateurs :** Utilisez `→` pour les membres via un pointeur et `.` pour une structure directe.
- **Modularité :** Pour des tableaux dynamiques complexes, créez des fonctions dédiées à leur création et destruction pour garantir la cohérence mémoire.

Ce qu'il faut retenir Ce cours clarifie l'usage des tableaux de pointeurs et des pointeurs sur structures afin de manipuler efficacement des données complexes, dynamiques et extensibles en C.

Sources Utilisées

- Using pointers to structs - GeeksforGeeks
- Arrays of pointers - C Programming
- Dynamic array of pointers - Stack Overflow
- C Structures and Pointers - Tutorialspoint
- ISO/IEC 9899:2018 Standard C

Listes Chaînées, Piles et Files

Introduction aux Structures Dynamiques

Les listes chaînées, piles et files sont des structures de données fondamentales pour la gestion dynamique des collections d'éléments en mémoire. Elles permettent une flexibilité et une efficacité accrues par rapport aux tableaux statiques.

Cette présentation synthétique explore leurs caractéristiques, fonctionnements et offre des exemples d'implémentation simple en C.

Listes Chaînées : La Base Dynamique

Définition

Une **liste chaînée** est une structure de données composée d'éléments (nœuds). Chaque nœud contient une donnée et un pointeur vers le nœud suivant. Elles permettent des insertions/suppressions efficaces n'importe où dans la liste, contrairement aux tableaux.

Structure Type en C

```c
typedef struct Node {
    int value;
    struct Node *next;
} Node;
```

Clé : Chaque maillon ("Node") se lie au suivant.

Listes Chaînées : Opérations Essentielles

Insertion en Tête

L'insertion d'un nouvel élément en début de liste est une opération rapide.

```c
Node *inserer_en_tete(Node *head, int val) {
    Node *new_node = malloc(sizeof(Node));
    if (new_node == NULL) return head; // Gestion erreur
    new_node->value = val;
    new_node->next = head; // Le nouveau nœud pointe vers l'ancienne tête
    return new_node;       // Le nouveau nœud devient la tête
}
```

Parcours de la Liste

Pour afficher ou traiter chaque élément.

```c
void afficher_liste(Node *head) {
    Node *current = head;
    while (current ≠ NULL) {
        printf("%d → ", current→value);
        current = current→next;
    }
    printf("NULL\n");
}
```

Piles (Stack) : La Logique LIFO

Définition

Une **pile** suit une logique **Last In, First Out (LIFO)** : le dernier élément inséré est le premier à être extrait. Imaginez une pile d'assiettes.

Implémentation Simplifiée (avec liste chaînée)

Les piles sont souvent implémentées en utilisant des listes chaînées, l'insertion et la suppression se faisant à la même extrémité (la "tête" de la liste).

```c
typedef Node Stack; // Une pile est simplement un pointeur vers un nœud

Stack *push(Stack *s, int val) { // Empiler
    return inserer_en_tete(s, val);
}

Stack *pop(Stack *s, int *val) { // Dépiler
    if (s == NULL) return NULL; // Pile vide
    *val = s->value;
    Node *temp = s->next;
    free(s); // Libère la mémoire de l'élément dépilé
    return temp;
}
```

Files (Queue) : La Logique FIFO

Définition

Une **file** respecte la logique **First In, First Out (FIFO)** : le premier élément inséré est le premier à être extrait. Comme une file d'attente réelle.

Implémentation Simplifiée (avec liste chaînée)

Pour une file, les insertions se font à la fin et les extractions au début. Cela nécessite de gérer à la fois la "tête" (`front`) et la "queue" (`rear`).

```c
typedef struct Queue {
    Node *front; // Tête de la file
    Node *rear;  // Fin de la file
} Queue;

void enqueue(Queue *q, int val) { // Ajouter à la fin
    Node *new_node = malloc(sizeof(Node)); //  ... (code d'insertion)
    //  ... Le nouveau nœud est ajouté après 'q->rear'
    //  ... et 'q->rear' est mis à jour.
}

int dequeue(Queue *q, int *val) { // Retirer du début
    if (q->front == NULL) return 0; // File vide
    Node *temp = q->front;
    *val = temp->value;
    q->front = q->front->next; // La nouvelle tête devient l'élément suivant
    if (q->front == NULL) q->rear = NULL; // Si la file est vide après, maj 'rear'
    free(temp); // Libère la mémoire
    return 1;
}
```

Synthèse & Ressources Utiles

Ce qu’il faut retenir

- Les listes chaînées facilitent les insertions/suppressions en temps constant (O(1)) en tête.
- Piles et files sont des abstractions mais souvent implémentées via listes chaînées ou tableaux.
- Gérer correctement la libération de mémoire est crucial dans les structures dynamiques.
- Choisir la structure selon le besoin spécifique d’ordre d’accès (LIFO vs FIFO).

Sources utilisées

- Linked List in C - GeeksforGeeks
- Stack Data Structure in C - Programiz
- Queue Data Structure in C - GeeksforGeeks
- CS50 Lecture - Data Structures
- Tutorialspoint - Linked List

Structures de Données Avancées

Listes, Piles, Files : Implémentations efficaces

Introduction

Les listes doublement chaînées et les buffers circulaires sont des structures de données optimisées pour des opérations fréquentes d'insertion, suppression et gestion mémoire circulaire.

Elles améliorent la flexibilité par rapport aux listes simplement chaînées et aux files classiques en offrant une meilleure gestion des accès et des performances.

1. Listes doublement chaînées (Doubly Linked Lists)

Concept : Séquence de nœuds où chaque nœud possède deux pointeurs : un vers l'élément suivant (`next`) et un vers l'élément précédent (`prev`). Cela permet un parcours bidirectionnel et simplifie la suppression.

Structure de nœud :

```c
typedef struct Node {
    int value;
    struct Node *prev;
    struct Node *next;
} Node;
```

Avantages :

- Parcours en avant et en arrière possible.
- Suppression et insertion en O(1) sans nécessiter de parcourir la liste pour trouver le prédécesseur.
- Utile pour les piles, files, et autres structures complexes (ex: deque).

Listes doublement chaînées : Opérations clés

Exemple d'insertion en début :

```c
Node* insert_head(Node *head, int val) {
    Node *new_node = malloc(sizeof(Node));
    if (!new_node) return head;
    new_node->value = val;
    new_node->prev = NULL;
    new_node->next = head;
    if (head) head->prev = new_node;
    return new_node;
}
```

Exemple de suppression d'un nœud :

```c
Node* delete_node(Node *head, Node *node) {
    if (!head || !node) return head;
    if (node->prev) node->prev->next = node->next;
    else head = node->next;
    if (node->next) node->next->prev = node->prev;
    free(node);
    return head;
}
```

2. Buffers circulaires (Circular Buffers)

Concept : Structure de données en mémoire utilisée comme une file circulaire. Une fois la fin du buffer atteinte, l'écriture ou la lecture reprend au début, évitant un coûteux décalage des valeurs.

Caractéristiques :

- Taille fixe (statique ou dynamique).
- Deux indices : `head` (point d'écriture), `tail` (point de lecture).
- Gestion des cas de buffer plein ou vide par des contrôles spécifiques.

Structure typique :

```
typedef struct {
    int *buffer;
    size_t head;
    size_t tail;
    size_t max;    // capacité du buffer
    int full;      // indicateur buffer plein
} CircularBuffer;
```

Buffers circulaires : Opérations clés

Exemple d'insertion (enqueue) :

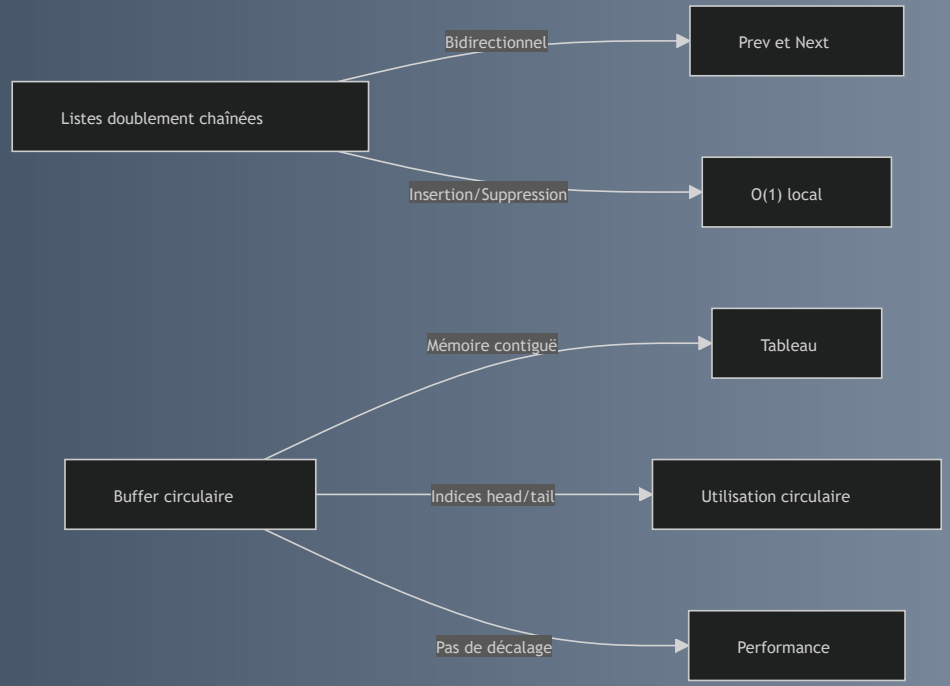
```c
void cb_enqueue(CircularBuffer *cb, int val) {
    if (cb->full) {
        // Optionnel : écraser le plus ancien ou gestion d'erreur
        cb->tail = (cb->tail + 1) % cb->max;
    }
    cb->buffer[cb->head] = val;
    cb->head = (cb->head + 1) % cb->max;
    cb->full = (cb->head == cb->tail);
}
```

Exemple de lecture (dequeue) :

```c
int cb_dequeue(CircularBuffer *cb, int *val) {
    if (cb->head == cb->tail && !cb->full) return 0; // buffer vide
    *val = cb->buffer[cb->tail];
    cb->tail = (cb->tail + 1) % cb->max;
    cb->full = 0;
    return 1;
}
```

Comparaison, usages et ce qu’il faut retenir

Comparaison des structures :



Résumé des usages :

Structure	Avantages	Usages typiques
Listes doublement chaînées	Parcours bidirectionnel, suppression rapide	Implémentation de déques, gestion d'historique
Buffers circulaires	Utilisation fixe de mémoire, performances optimales	Communication en temps réel, buffers audio/vidéo

Ce qu'il faut retenir : Ce cours présente deux structures performantes permettant un contrôle efficace des collections grâce à une gestion mémoire optimisée et adaptée aux besoins d'accès rapide et modifiable.

Sources utilisées :

- Doubly Linked List - GeeksforGeeks
- Circular Buffer - Embedded Artistry
- Circular Queue Implementation - GeeksforGeeks
- [Data Structures and Algorithms in C, Adam Drozdek]
- Coding Labs - Circular Buffer in C

Structures de Données Avancées

Arbres Binaires de Recherche et Arbres Équilibrés

- **Introduction aux ABR**
  - Permettent de stocker, rechercher, insérer et supprimer efficacement des données ordonnées.
  - Visent des performances optimales, idéalement en temps logarithmique (O(log n)).
  - Nécessitent des **arbres équilibrés** pour éviter la dégradation en liste chaînée (O(n)).
- **Définition d'un ABR**
  - Chaque nœud respecte la propriété d'ordre :
    - Éléments du sous-arbre gauche < Valeur du nœud.
    - Éléments du sous-arbre droit > Valeur du nœud.
  - Facilite une recherche rapide par parcours directionnel.

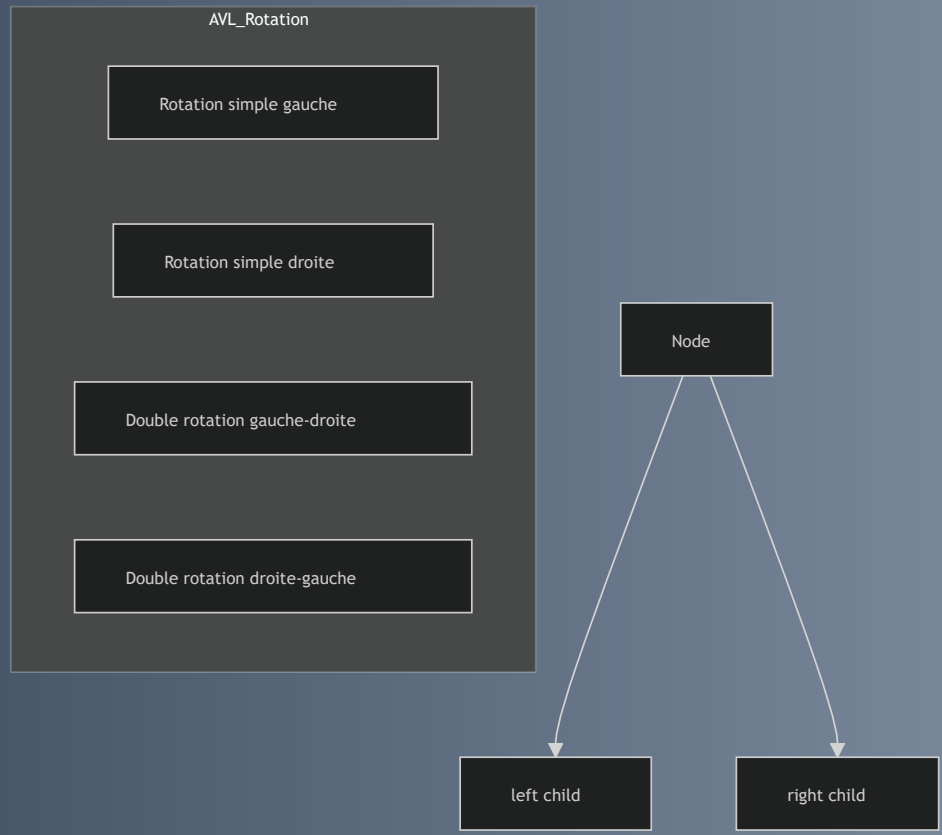
- **Structure d’un nœud (exemple en C)**

```c
typedef struct Node {
    int key;
    struct Node *left;
    struct Node *right;
} Node;
```

- **Opérations clés : Recherche et Insertion**
  - La recherche suit une logique récursive, se dirigeant vers le sous-arbre gauche ou droit selon la valeur recherchée.
  - L’insertion trouve la position appropriée en respectant la propriété d’ordre, puis crée un nouveau nœud.

L’impératif de l’Équilibrage : Maintenir la performance

- **Pourquoi équilibrer ?**
  - Un ABR classique peut devenir déséquilibré, ressemblant alors à une liste chaînée.
  - Dans ce cas, les opérations passent de O(log n) à O(n), perdant leur efficacité.
  - Un arbre équilibré maintient une hauteur minimale pour garantir des performances O(log n).
- **Principaux types d’arbres équilibrés**
  - **Arbre AVL**
    - La différence de hauteur entre le sous-arbre gauche et droit de chaque nœud ne dépasse jamais 1 (facteur d’équilibre $\in \{-1, 0, 1\}$).
    - Des **rotations** (simples ou doubles) sont effectuées automatiquement pour corriger les déséquilibres lors des insertions/suppressions.
  - **Arbre Rouge-Noir**
    - Moins strict que l’AVL, plus couramment utilisé.
    - Les nœuds sont colorés en rouge ou noir selon des règles spécifiques.
    - Ces règles garantissent un équilibre suffisant sans être aussi rigoureuses que l’AVL.



- **Structure d’un ABR :** Chaque nœud pointe vers ses enfants gauche et droit, formant une hiérarchie ordonnée.
- **Rotations AVL :** Ces opérations restructurent l’arbre localement pour rétablir son équilibre après une modification, assurant ainsi la propriété AVL.

Complexité des Opérations : Impact de l’équilibrage

L’équilibrage est crucial pour la performance des structures de données.

Opération	ABR non équilibré	Arbres équilibrés (AVL, Rouge-Noir)
Recherche	O(n)	O(log n)
Insertion	O(n)	O(log n)
Suppression	O(n)	O(log n)

- **ABR non équilibré :** Dans le pire des cas (données déjà triées), l’arbre dégénère en liste chaînée, et les opérations deviennent linéaires.
- **Arbres équilibrés :** Grâce à leurs mécanismes d’auto-équilibrage, ils garantissent des performances logarithmiques même dans le pire des cas, assurant ainsi une efficacité constante.

Ce qu’il faut retenir & Références

- **Synthèse :**
  - Les **arbres binaires de recherche (ABR)** sont fondamentaux pour manipuler des ensembles de données dynamiques et ordonnés.
  - L’intégration de mécanismes d’**équilibrage automatique** (comme dans les arbres AVL ou Rouge-Noir) est essentielle.
  - Ces mécanismes garantissent une performance optimale et stable (O(log n)) même dans les scénarios les plus défavorables, grâce à des règles spécifiques et des opérations de rotation.
- **Sources utilisées :**
  - Binary Search Tree - GeeksforGeeks
  - AVL Tree - Programiz
  - Red Black Tree - GeeksforGeeks
  - [Introduction to Algorithms (Cormen et al.) - Chapitre sur arbres équilibrés]
  - Wikipedia - Binary Search Tree
  - Wikipedia - AVL Tree

Représentations de Graphes : Matrice vs. Listes d’adjacence

Un graphe est une structure fondamentale composée de **sommets** (nœuds) et d’**arêtes** (arcs) reliant certains d’entre eux.

Pour exploiter efficacement un graphe en mémoire, le choix de sa représentation est crucial. Deux approches principales s’offrent à nous :

1. **La Matrice d’adjacence**
2. **Les Listes d’adjacence**

Le choix idéal dépendra de la **densité du graphe** (nombre d’arêtes par rapport au nombre maximal possible) et des **opérations** que l’on souhaite prioritairement réaliser.

Matrice d’adjacence : Accès direct et simplicité

Description : Une matrice carrée (N×N, N = nombre de sommets) où chaque entrée `[i][j]` indique la présence (et parfois le poids) d’une arête du sommet `i` vers `j`.

- **Graphe non orienté :** la matrice est symétrique.
- **Graphe orienté :** les deux sens sont distincts.

Avantages :

- **Accès direct et rapide** à l’existence d’une arête : **O(1)**.
- Représentation simple et intuitive.

Inconvénients :

- **Occupation mémoire importante : O(N²)**, même pour des graphes peu denses.
- Itération sur les voisins d’un sommet coûte **O(N)**.

Exemple :

```
int matrice_adj[4][4] = {
    {0, 1, 0, 0}, // Sommet 0 connecté au 1
    {1, 0, 1, 1}, // Sommet 1 connecté aux 0, 2, 3
    {0, 1, 0, 0}, // Sommet 2 connecté au 1
    {0, 1, 0, 0}  // Sommet 3 connecté au 1
};
```

Listes d’adjacence : Optimisation mémoire pour les graphes clairsemés

Description : Chaque sommet possède une liste (chaînée ou tableau dynamique) énumérant ses voisins directs.

- C’est une représentation plus économe en mémoire, particulièrement adaptée aux **graphes clairsemés** (peu d’arêtes).

Avantages :

- **Utilisation mémoire efficace :** proportionnelle au nombre d’arêtes (souvent bien inférieure à N^2).
- **Itération rapide sur les voisins** d’un sommet : coût proportionnel à son degré.

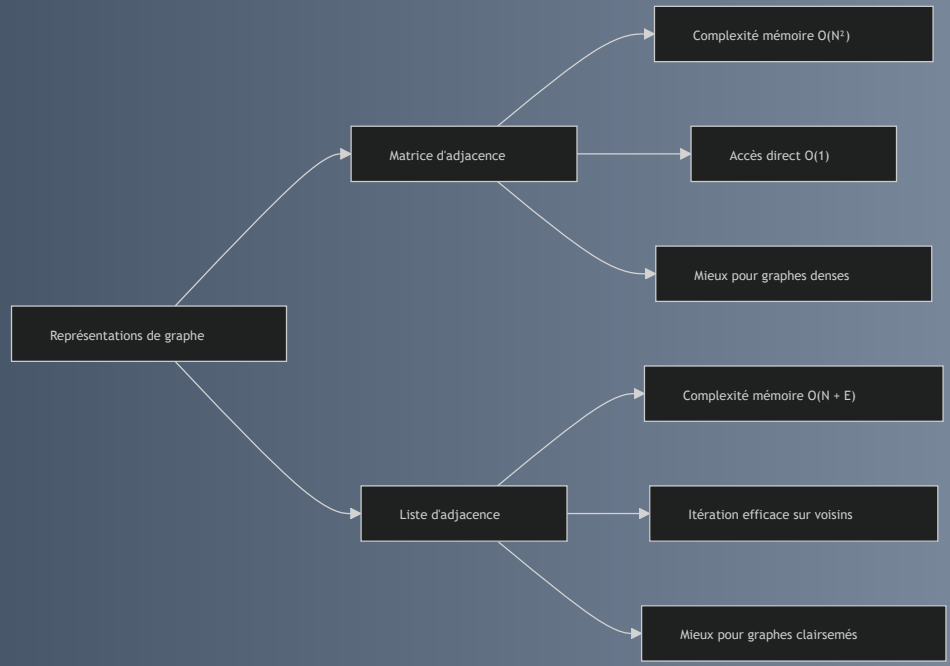
Inconvénients :

- **Accès à l’existence d’une arête :** potentiellement O(k) (k = degré du sommet, nécessite le parcours de la liste).
- Implémentation un peu plus complexe.

Exemple (structures clés) :

```c
typedef struct AdjNode {
    int vertex;
    struct AdjNode* next;
} AdjNode;

typedef struct Graph {
    int numVertices;
    AdjNode** adjLists; // Tableau de pointeurs vers les listes de voisins
} Graph;

void addEdge(Graph* graph, int src, int dest) {
    // Ajoute 'dest' à la liste de 'src'
    // Et 'src' à la liste de 'dest' si graphe non orienté
}
```

Représentations de graphes : Un choix éclairé par la densité



Matrice d’adjacence vs. Listes : Un tableau comparatif

Critère	Matrice d’adjacence	Liste d’adjacence
Espace mémoire	$O(N^2)$ (fixe)	O(N + E) (variable)
Recherche d’arête	O(1)	O(k) (k = degré sommet)
Parcours voisins	O(N)	O(k)
Facilité d’impl.	Simple	Plus élaborée
Adapté pour	Graphes denses	Graphes clairs

(N = nombre de sommets, E = nombre d’arêtes, k = degré du sommet)

Choisir sa représentation : Synthèse et Ressources

Ce qu'il faut retenir :

- Le choix entre matrice et liste d'adjacence dépend fortement de la **densité du graphe** et des **opérations prioritaires** à effectuer.
- La **matrice d'adjacence** permet un **accès très rapide** au statut d'une arête (existe ou non) mais est **gourmande en mémoire** pour les graphes vastes et peu denses.
- Les **listes d'adjacence** offrent une **plus grande économie mémoire** et une **efficacité supérieure** pour la traversal des voisins, ce qui en fait un choix idéal pour la **majorité des graphes réels** qui sont souvent clairsemés.

Pour aller plus loin (Sources utilisées) :

- Graph Representation - GeeksforGeeks
- Graph data structure - Wikipedia
- C implementation of adjacency list - Programiz
- [Introduction to Algorithms (Cormen et al.), chapitre sur graphes]

Manipulation de Fichiers et Entrées/Sorties Avancées

Fichiers Texte et Binaire : Fonctions clés `fopen`, `fread`, `fwrite`, `fseek`, `ftell`

La gestion des fichiers en C repose sur un ensemble de fonctions standards. Celles-ci permettent d'ouvrir, lire, écrire et manipuler des fichiers, qu'ils soient texte ou binaires, avec un contrôle précis. Elles offrent la flexibilité nécessaire pour naviguer dans un fichier, gérer des blocs de données et contrôler la position du curseur.

Ouvrir un Fichier : La Fonction `fopen`

Syntaxe

```
FILE *fopen(const char *filename, const char *mode);
```

- `filename` : Chemin vers le fichier.
- `mode` : Spécifie le type d’accès (lecture, écriture, ajout, binaire, etc.).

Principaux modes d’ouverture

Mode	Description
`"r"`	Lecture seule
`"w"`	Écriture (écrase le fichier ou le crée)
`"a"`	Ajout (écrit à la fin du fichier)
`"rb"`, `"wb"`, `"ab"`	Modes équivalents pour les fichiers binaires
`"r+"`, `"w+"`, `"a+"`	Lecture et écriture simultanées

Exemple et vérification

```
FILE *file = fopen("data.bin", "rb");
if (file == NULL) {
    perror("Erreur ouverture fichier"); // Gérer l'erreur
    // ... sortie du programme ou autre gestion
}
// N'oubliez pas de fermer : fclose(file);
```

Lecture et Écriture Binaire : `fread` et `fwrite`

Ces fonctions sont essentielles pour manipuler des blocs de données en binaire.

Syntaxe

```
size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream);
size_t fwrite(const void *ptr, size_t size, size_t nmemb, FILE *stream);
```

- `ptr` : Pointeur vers la zone mémoire où stocker/lire les données.
- `size` : Taille en octets d'un seul élément à lire/écrire.
- `nmemb` : Nombre d'éléments à lire/écrire.
- `stream` : Pointeur vers le fichier ouvert (`FILE *`).

Elles retournent le nombre d'éléments effectivement lus ou écrits.

Exemple : Écriture et Lecture d'un tableau d'entiers

```c
// Écriture
int tab_write[5] = {10, 20, 30, 40, 50};
FILE *f_w = fopen("entiers.bin", "wb");
if (f_w) {
    fwrite(tab_write, sizeof(int), 5, f_w);
    fclose(f_w);
}

// Lecture
int tab_read[5];
FILE *f_r = fopen("entiers.bin", "rb");
if (f_r) {
    fread(tab_read, sizeof(int), 5, f_r);
    fclose(f_r);
    for (int i=0; i<5; i++) {
        printf("%d ", tab_read[i]); // Affiche : 10 20 30 40 50
    }
}
```

Se Déplacer dans un Fichier : `fseek` et `ftell`

Ces fonctions permettent un contrôle précis de la position de lecture/écriture.

`fseek` : Positionner le curseur

```c
int fseek(FILE *stream, long offset, int whence);
```

- Permet de positionner le curseur de lecture/écriture.
- `offset` : Déplacement en octets.
- `whence` : Point de référence pour l'offset :
  - `SEEK_SET` : Début du fichier.
  - `SEEK_CUR` : Position actuelle.
  - `SEEK_END` : Fin du fichier.

- Retourne `0` si succès, sinon une erreur.

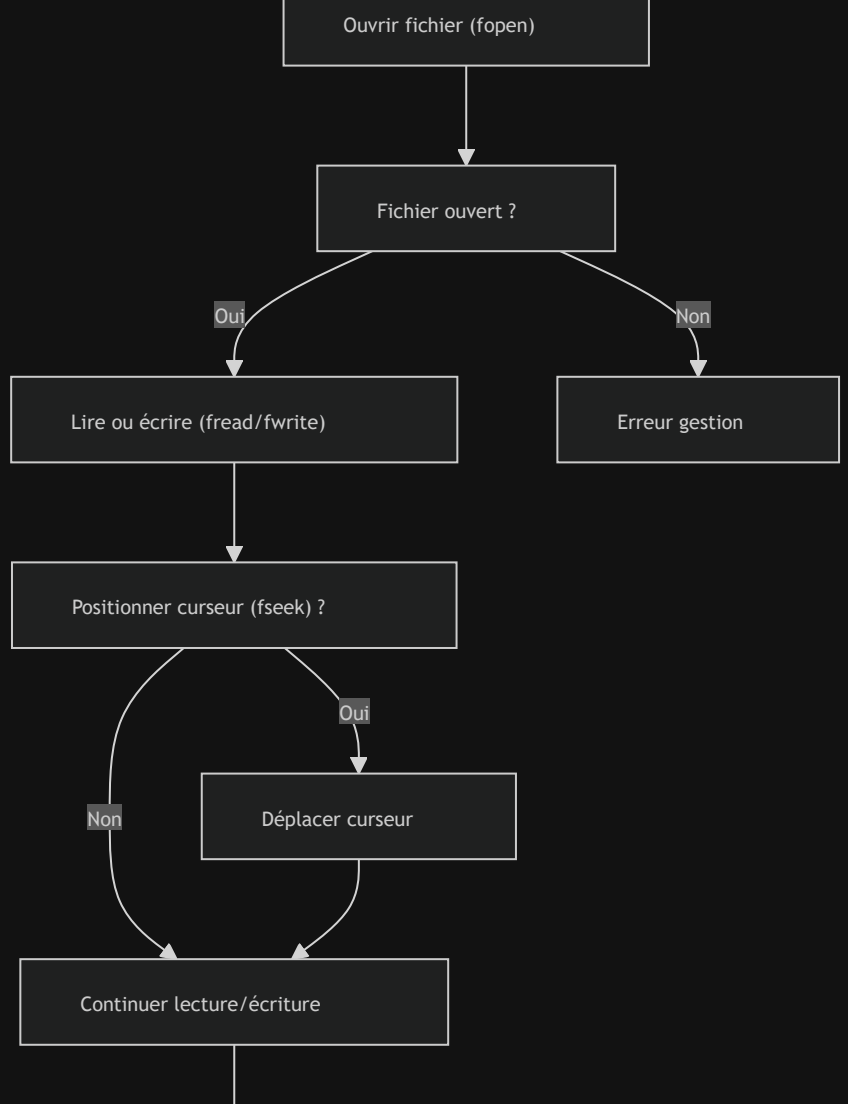
`ftell` : Obtenir la position actuelle

```
long ftell(FILE *stream);
```

- Retourne la position actuelle du curseur dans le fichier (en octets depuis le début).

Exemple : Lire un entier au milieu du fichier

```
FILE *f = fopen("entiers.bin", "rb");
if (f) {
    fseek(f, 2 * sizeof(int), SEEK_SET); // Aller au 3e entier (index 2)
    int val;
    fread(&val, sizeof(int), 1, f);
    printf("3e entier = %d\n", val); // Affiche : 3e entier = 30
    fclose(f);
}
```


Ouvrir fichier (fopen)
Fichier ouvert ?
Oui
Non
Lire ou écrire (fread/fwrite)
Erreur gestion
Positionner curseur (fseek) ?
Oui
Non
Déplacer curseur
Continuer lecture/écriture

Conseils pratiques

- **Vérifier l'ouverture :** Toujours s'assurer que `fopen` n'a pas retourné `NULL`.
- **Gérer les erreurs :** Vérifier les retours de `fread` ou `fwrite` (nombre d'éléments lus/écrits).
- **Fermer les fichiers :** Toujours utiliser `fclose` pour libérer les ressources.
- **Modes appropriés :** Utiliser `"rb"`/`"wb"` pour le binaire, `"r"`/`"w"` pour le texte.
- **Navigation efficace :** `fseek` et `ftell` sont clés pour les fichiers volumineux ou l'accès ciblé.

Ce qu’il faut retenir & Ressources

En résumé

Ce cours a exploré comment ouvrir, lire, écrire et naviguer dans des fichiers en C avec les fonctions standard. Ces outils permettent un contrôle fin de l’accès aux données, qu’elles soient en format texte ou binaire, un aspect fondamental de la programmation système.

Sources utilisées

- C Standard I/O Library - cppreference
- fopen - GNU C Library documentation
- Fread and fwrite in C - GeeksforGeeks
- fseek and ftell - Tutorialspoint
- Wikipedia : stdio.h

Gestion des erreurs lors des opérations sur les fichiers

Fichiers Texte et Binaire en C

Les opérations sur fichiers en C (ouverture, lecture, écriture, fermeture) peuvent échouer. Fichier inexistant, permissions insuffisantes, disque plein, ou erreur d’E/S sont des raisons courantes. Une bonne gestion des erreurs est indispensable pour éviter les comportements imprévisibles et permettre un diagnostic précis.

1. L’ouverture du fichier : Le premier test

La première étape consiste à contrôler que le fichier s’est bien ouvert.

```c
FILE *file = fopen("data.txt", "r");
if (file == NULL) {
    perror("Erreur ouverture fichier");
    return EXIT_FAILURE;
}
```

- `fopen` retourne `NULL` si l’ouverture échoue.
- La fonction `perror` affiche un message d’erreur associé à la variable globale `errno`.

2. Erreurs lors des lectures et écritures

`fread` et `fwrite` retournent le nombre d'éléments effectivement lus ou écrits. Si ce nombre est inférieur à l'attendu, une erreur ou fin de fichier a pu se produire.

Lecture (`fread`) :

```c
size_t n = fread(buffer, sizeof(char), taille, file);
if (n < taille) {
    if (feof(file))
        printf("Fin de fichier atteinte\n");
    else if (ferror(file)) {
        perror("Erreur lors de la lecture");
    }
}
```

* `feof(FILE*)` détecte la fin de fichier. * `ferror(FILE*)` indique une erreur d'E/S.

Écriture (`fwrite`) :

```c
size_t n = fwrite(buffer, sizeof(char), taille, file);
if (n < taille) {
    perror("Erreur lors de l'écriture");
}
```

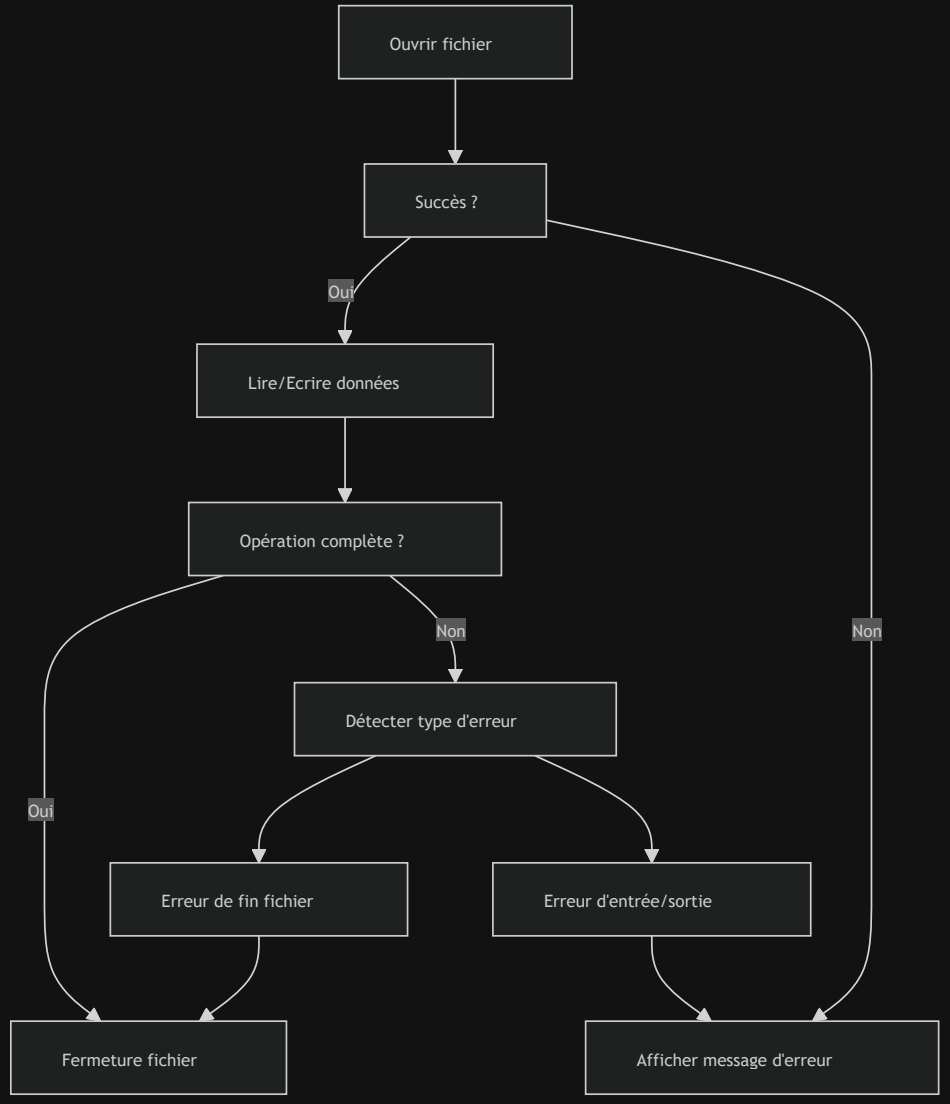
3. Gestion avancée et cas spécifiques

Réinitialiser les indicateurs d'erreur :

```
clearerr(file); // Réinitialise les indicateurs d'erreur et de fin de fichier.
```

Déplacement et consultation de position (`fseek`, `ftell`) :

```
if (fseek(file, 0, SEEK_SET) != 0) {
    perror("Erreur lors du déplacement dans le fichier");
}

long pos = ftell(file);
if (pos == -1L) {
    perror("Erreur lors de la consultation de la position");
}
```


Ouvrir fichier
Succès ?
Oui
Non
Lire/Ecrire données
Opération complète ?
Oui
Non
Détecter type d'erreur
Erreur de fin fichier
Erreur d'entrée/sortie
Fermeture fichier
Afficher message d'erreur

Exemple de flux classique avec gestion d'erreur :

```c
FILE *file = fopen("data.txt", "r");
if (!file) { perror("Ouverture"); return EXIT_FAILURE; }

char buffer[128];
while (fgets(buffer, sizeof(buffer), file) ≠ NULL) {
    printf("%s", buffer);
}

if (ferror(file)) { perror("Erreur de lecture"); }
fclose(file);
```

5. Bonnes Pratiques & Ressources

Ce qu'il faut retenir :

- Toujours tester la valeur de retour des fonctions d'E/S.
- Utiliser `perror` ou `strerror(errno)` pour des messages d'erreur compréhensibles.
- Gérer la fin de fichier distinctement des erreurs d'E/S.
- Fermer le fichier même en cas d'erreur pour libérer les ressources.

Ce contenu clarifie les mécanismes standards pour détecter, signaler et gérer les erreurs lors des opérations sur fichiers, offrant une maîtrise nécessaire pour produire du code robuste en entrée/sortie.

Sources utilisées :

- Error Handling in C File I/O - GeeksforGeeks
- FILE * - GNU C Library
- fread, fwrite, fseek, ftell - cppreference
- C error handling - POSIX standards
- perror function - Linux man pages

Introduction aux Entrées/Sorties Bufférisées en C

Comprendre les Tampons (Buffers)

Les entrées/sorties (I/O) en C peuvent être **bufférisées** ou **non bufférisées**. La plupart des fonctions de la bibliothèque standard utilisent des tampons mémoire (buffers) pour optimiser les performances.

Qu'est-ce qu'un tampon ? C'est une zone mémoire temporaire où les données sont stockées avant d'être lues ou écrites physiquement.

- **E/S bufférisée** : Les données sont d'abord copiées dans/le tampon, puis transférées en un bloc.
- **E/S non bufférisée** : Chaque opération est immédiatement exécutée au niveau matériel.

But principal : Réduire le nombre d'appels systèmes coûteux en regroupant plusieurs petites opérations en une seule grande.

Tampons dans les Flux Fichier C (`stdio.h`)

La bibliothèque standard `stdio.h` utilise un tampon pour les flux `FILE*`.

- **Écriture** : Les données sont stockées dans un tampon en mémoire, puis écrites sur le disque lorsque le tampon est plein, lors d'un `fflush` ou lors de la fermeture du fichier.
- **Lecture** : Un bloc de données est lu dans le tampon, puis fourni au programme petit à petit.

Modes de buffering :

Mode	Description	Exemples typiques
Buffered (plein)	Buffer en mémoire, écrit/lit par blocs.	Fichiers sur disque
Line buffered	Buffer vidé à chaque retour à la ligne (`\n`).	Sortie standard, consoles
Unbuffered	Pas de buffer, chaque opération est immédiate.	Erreur standard, flux critiques

Comprendre l’Impact du Buffering

Le buffering peut retarder l’affichage ou l’écriture réelle des données, menant à des comportements inattendus.

Exemple concret :

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

int main() {
    printf("Début"); // Pas de retour à la ligne
    sleep(5);        // Pause 5 secondes
    printf(" Fin\n");
    return 0;
}
```

- Le texte "Début" n’apparaît pas immédiatement car il est mis en tampon (mode `line buffered` pour `stdout` mais sans `\n`).
- Il est affiché seulement lorsque `printf(" Fin\n")` vide le tampon (grâce au `\n`) ou à la fin du programme.

Contrôler la Vidange du Tampon

Forcer la vidange (`fflush`)

Avec `fflush(FILE *stream)`, on vide explicitement le tampon d'écriture du flux spécifié.

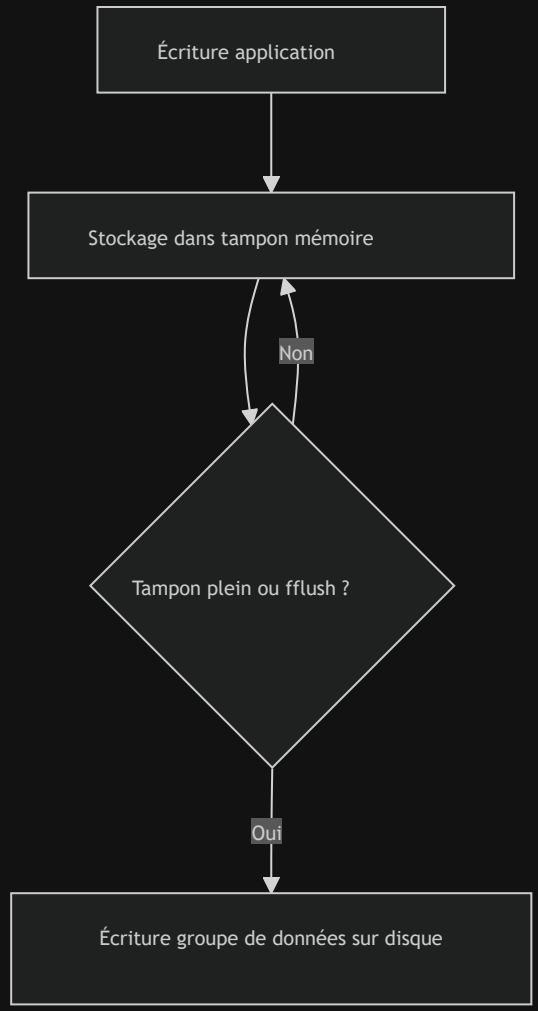
```c
#include <stdio.h>
#include <unistd.h>

int main() {
    printf("Début");
    fflush(stdout); // "Début" est affiché immédiatement
    sleep(5);
    printf(" Fin\n");
    return 0;
}
```

Entrées/Sorties Non Bufférisées

Pour des opérations immédiates sans tampon, on utilise des fonctions de bas niveau (ex. POSIX `write`).

```c
#include <unistd.h> // Pour write et STDOUT_FILENO

int main() {
    write(STDOUT_FILENO, "Début", 5); // Écriture immédiate
    sleep(5);
    write(STDOUT_FILENO, " Fin\n", 5);
    return 0;
}
```

Chaque écriture est directe et immédiate, sans retard lié au tampon.


Écriture application
Stockage dans tampon mémoire
Non
Tampon plein ou fflush ?
Oui
Écriture groupe de données sur disque

Points Clés et Ressources

Ce qu'il faut retenir : Ce module clarifie la nature des tampons dans les entrées/sorties en C, leur influence sur le comportement des programmes, et comment les contrôler pour garantir un bon équilibre entre performance et réactivité.

Sources utilisées :

- Buffering - cppreference
- stdio buffering - GNU C Library
- How I/O buffering works - GNU Libc manual
- fflush - POSIX specification
- Difference between buffered and unbuffered I/O - Stack Overflow

Manipulation de Fichiers et Entrées/Sorties

Contrôle des Buffers : `setbuf`, `fflush` et `fsync`

Introduction

Les opérations d'Entrées/Sorties (E/S) en C utilisent des tampons (buffers) pour optimiser les performances. Ce module explore comment contrôler ces buffers, assurant à la fois l'efficacité et la persistance des données.

- `setbuf` : Configure le comportement du buffer d'un flux.
- `fflush` : Force la vidange immédiate du buffer.
- `fsync` : Synchronise les données avec le support de stockage physique.

1. `setbuf` : Configuration du Buffer d’un Flux

Description

`setbuf` permet d’associer ou de dissocier un tampon mémoire personnalisé pour un flux `FILE*`.

Prototype

```
void setbuf(FILE *stream, char *buf);
```

- `stream` : Le flux (ex. `stdout`, fichier ouvert).
- `buf` :
  - Pointeur vers un buffer utilisateur (taille `BUFSIZ`).
  - `NULL` pour désactiver le buffering (mode non bufférisé).

Effets

- `buf ≠ NULL` : Le flux utilise le tampon fourni.
- `buf == NULL` : Le flux devient non bufferisé (opérations immédiates).

Exemple

```c
char my_buffer[BUFSIZ];
setbuf(stdout, my_buffer); // Utilise un buffer personnalisé

// Pour désactiver le buffering de stdout
setbuf(stdout, NULL);
```

2. `fflush` : Vidange Forcée du Buffer C

Description

`fflush` force l'écriture immédiate des données présentes dans le tampon mémoire C vers le fichier ou le flux associé (vers le buffer du noyau).

Prototype

```c
int fflush(FILE *stream);
```

- Si `stream == NULL`, `fflush` agit sur **tous les flux** ouverts en écriture.

Retour

- `0` en cas de succès.
- `EOF` en cas d'erreur.

Exemple

```c
printf("Message immédiat");
fflush(stdout); // Vide le tampon de stdout pour un affichage instantané
```

3. `fsync` : Synchronisation Disque au Niveau Système

Description

`fsync` est une fonction POSIX qui force la synchronisation des données d'un fichier ouvert depuis le buffer du noyau vers le disque physique. Elle agit **après** la gestion du tampon par la bibliothèque C.

Prototype

```
int fsync(int fd);
```

- `fd` : descripteur de fichier obtenu avec `fileno(FILE*)`.

Utilité

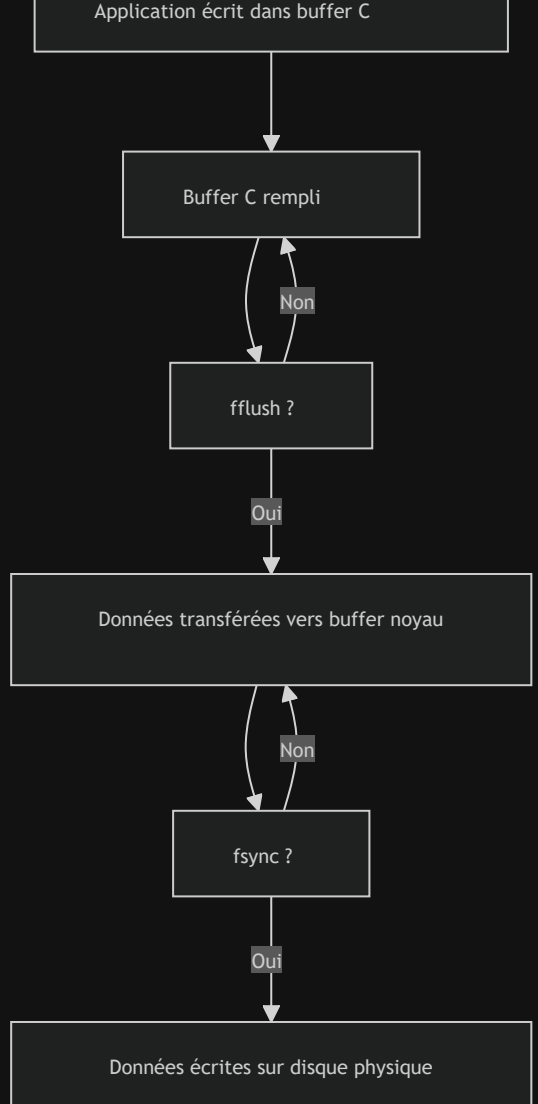
Assurer que les données sont écrites physiquement sur le support de stockage, crucial pour la durabilité et la récupération des données.

Exemple

```
FILE *file = fopen("data.txt", "w");
fprintf(file, "Données critiques\n");
fflush(file);         // Vide le buffer C vers le buffer noyau
fsync(fileno(file));  // Force l'écriture du buffer noyau sur le disque
fclose(file);
```

Différences et Complémentarités

Fonction	Type de Buffering	Objectif	Niveau
`setbuf`	Buffer bibliothèque C	Configurer/désactiver le buffer	Application
`fflush`	Buffer bibliothèque C	Forcer la vidange du buffer	Application
`fsync`	Buffer système d’exploitation	Synchroniser les données sur disque	Système


Application écrit dans buffer C
Buffer C rempli
Non
fflush ?
Oui
Données transférées vers buffer noyau
Non
fsync ?
Oui
Données écrites sur disque physique

Points Clés & Ressources

Ce qu'il faut retenir

- `setbuf` doit être appelé **avant** la première opération I/O sur le flux.
- `fflush` garantit l'écriture dans le buffer du noyau, mais **pas nécessairement sur disque**.
- `fsync` garantit la persistance des données sur le support physique.
- En l'absence de `fflush`, `fsync` ne synchronisera pas toutes les données (le buffer C n'étant pas vidé).
- Ces fonctions sont essentielles pour les contextes où la perte de données est critique (systèmes embarqués, bases de données).

Sources utilisées

- setbuf - C Standard Library - cppreference
- fflush - C Standard Library - cppreference
- fsync - POSIX - man7.org
- Buffering - GNU Libc manual
- Buffering explained - Stack Overflow

Programmation Système et Multitâche : `fork`, `exec`, `wait`

Introduction au Multitâche : Le Cœur d'UNIX/Linux

Dans les systèmes UNIX/Linux, le multitâche repose sur la création et la gestion des processus.

Les appels système fondamentaux pour la gestion des processus sont :

- `fork` : Crée un nouveau processus.
- `exec` : Remplace le programme courant d'un processus.
- `wait` : Synchronise un parent avec la fin d'un processus enfant.

Ces fonctions permettent de construire des applications système complexes et réactives.

Fonction `fork()` : La Création de Processus

1. Description `fork()` crée un processus enfant en dupliquant le processus appelant. Le processus parent et le fils s'exécutent alors simultanément.

2. Prototype

```
pid_t fork(void);
```

3. Retourne :

- Dans le **processus parent** : le PID (identifiant) du fils.
- Dans le **processus fils** : 0.
- En cas d'erreur : -1.

4. Exemple

```c
pid_t pid = fork();
if (pid < 0) {
    perror("fork échoué");
}
else if (pid == 0) {
    // Code exécuté par le fils
    printf("Fils, PID=%d\n", getpid());
}
else {
    // Code exécuté par le parent
    printf("Parent, PID=%d, PID fils=%d\n", getpid(), pid);
}
```

Fonction `exec()` : Le Remplacement du Programme

1. Description Les fonctions `exec` remplacent l'image mémoire du processus courant par un nouveau programme. Le processus conserve son PID, mais la mémoire, le code et la pile sont remplacés.

2. Famille des fonctions

- `execl`, `execv`, `execle`, `execve`, `execlp`, `execvp`, avec variantes selon la manière de passer les arguments et l'environnement.
- Exemples : `execl("/bin/ls", "ls", "-l", NULL);` ou `execvp("ls", args);`

3. Prototype générique

```
int execvp(const char *file, char *const argv[]);
```

- Ne retourne jamais si succès.
- En cas d'erreur, renvoie -1 et définit `errno`.

4. Exemple

```
char *args[] = {"ls", "-l", NULL};
execvp("ls", args);
perror("execvp échoué");  // n'est atteint que si execvp échoue
```

Fonction `wait()` : Attente et Synchronisation

1. Description `wait` suspend le processus parent jusqu'à la terminaison d'un de ses enfants.

2. Prototype

```
pid_t wait(int *status);
```

- Retourne le PID de l'enfant terminé.
- `status` est un entier où est stocké le code de fin.

3. Analyse du code retourné

- Macro `WIFEXITED(status)` : vrai si le fils s'est terminé normalement.
- Macro `WEXITSTATUS(status)` : donne le code de retour du fils.

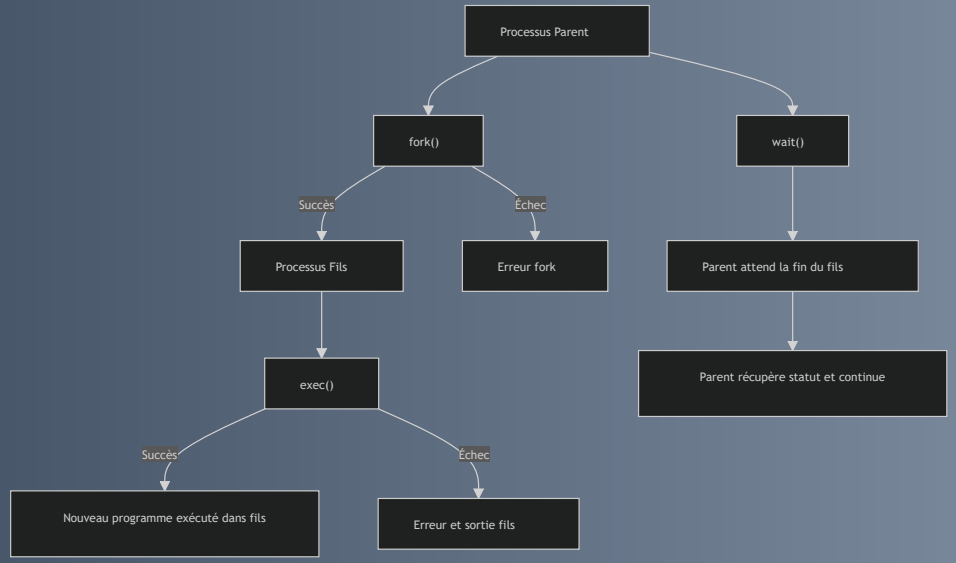
4. Exemple

```c
int status;
pid_t pid = wait(&status);
if (pid > 0) {
    if (WIFEXITED(status)) {
        printf("Enfant %d terminé avec code %d\n", pid, WEXITSTATUS(status));
    }
}
else {
    perror("wait échoué");
}
```

Exemple Complet & Cycle de Vie d’un Processus

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>

int main() {
    pid_t pid = fork();
    if (pid < 0) { perror("fork"); return EXIT_FAILURE; }
    else if (pid == 0) {
        // Processus fils: remplacer par un autre programme
        char *args[] = {"ls", "-l", NULL};
        execvp("ls", args);
        perror("execvp"); // si execvp échoue
        exit(EXIT_FAILURE);
    }
    else {
        // Processus parent : attendre la fin du fils
        int status;
        wait(&status);
        if (WIFEXITED(status)) {
            printf("Fils terminé avec code %d\n", WEXITSTATUS(status));
        }
    }
    return EXIT_SUCCESS;
}
```

Diagramme : Cycle de vie avec `fork` / `exec` / `wait`



Synthèse & Ressources

Ce qu'il faut retenir Ces fonctions constituent la base pour la gestion des processus en multitâche sous UNIX/Linux. `fork` crée un processus enfant, `exec` charge un nouveau programme dans ce processus, et `wait` permet au parent de gérer la fin des processus enfants, assurant ainsi une synchronisation efficace dans les applications système.

Sources utilisées

- fork - Linux man page
- exec - Linux man page
- wait - Linux man page
- [Advanced Programming in the UNIX Environment - Stevens, Rago]
- [The Linux Programming Interface - Kerrisk]

Programmation Système : Gestion des Signaux

Mécanismes d’interruption asynchrone

Les signaux sont un mécanisme d’interruption asynchrone dans les systèmes UNIX/Linux. Ils permettent de notifier un processus d’un événement ou d’une condition particulière (ex: interruption clavier, fin d’un fils, erreur critique).

La gestion des signaux repose sur plusieurs concepts fondamentaux :

- Identification du signal
- Action par défaut
- Gestion personnalisée via des *handlers*
- Masquage temporaire

Qu’est-ce qu’un Signal et Ses Actions

Un **signal** est une notification envoyée à un processus ou un groupe de processus pour interrompre son déroulement normal et déclencher une action particulière.

Signaux standard courants

Signal	Description
`SIGINT`	Interruption clavier (Ctrl+C)
`SIGTERM`	Demande d’arrêt
`SIGKILL`	Arrêt forcé (irrécupérable)
`SIGCHLD`	Fin d’un processus fils
`SIGALRM`	Alarme temporisée

Actions des signaux

Un signal peut entraîner :

- **Action par défaut** (exemple : terminer le processus)
- **Ignoré**
- **Traitement personnalisé** via une fonction appelée *handler* (gestionnaire de signal)

Installer un Gestionnaire de Signal

1. Fonction `signal()`

```
#include <signal.h>

void (*signal(int sig, void (*handler)(int)))(int);
```

- Permet d'assigner un `handler` (fonction) pour le signal `sig`.
- Le `handler` est une fonction prenant un `int` (le signal reçu).

Exemple d'utilisation simple :

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

void handler_SIGINT(int sig) {
    printf("Signal SIGINT reçu (%d)\n", sig);
}

int main() {
    signal(SIGINT, handler_SIGINT);
    while(1); // Boucle infinie pour attendre
    return 0;
}
```

2. Fonction `sigaction()` (recommandée)

Remplace `signal()` pour plus de contrôle (masquage temporaire de signaux, comportements portables).

```c
#include <signal.h>
//  ...
struct sigaction sa;
sa.sa_handler = handler_SIGINT; // Votre handler
sigemptyset(&sa.sa_mask);       // Aucun signal bloqué pendant le handler
sa.sa_flags = 0;                // Options supplémentaires
sigaction(SIGINT, &sa, NULL);   // Enregistrement pour SIGINT
```

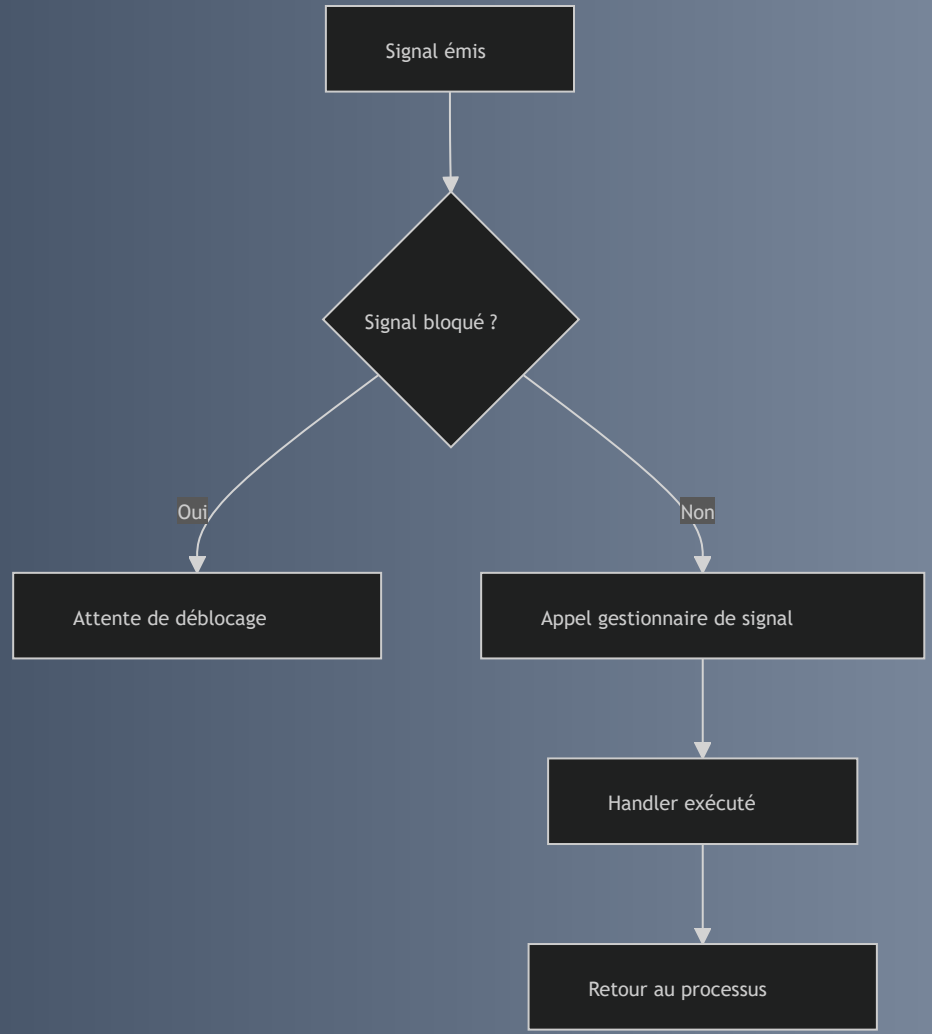
Masquage, Multitâche et Cycle de Traitement

Masquage et blocage des signaux

- `sigprocmask` permet de bloquer certains signaux afin d'éviter leur traitement pendant une section critique.
- Le masque est un ensemble de signaux bloqués temporairement.

Gestion des signaux en multitâche

- Les signaux peuvent réveiller des processus en attente (`pause()`, `sigsuspend()`).
- `SIGCHLD` est souvent utilisé pour notifier la fin d'un processus fils, permettant au parent de faire un `wait()` et d'éviter les processus zombies.


Signal émis
Signal bloqué ?
Oui
Non
Attente de déblocage
Appel gestionnaire de signal
Handler exécuté
Retour au processus

Exemple Pratique : Récupérer un processus fils (`SIGCHLD`)

Cet exemple montre comment un processus parent peut être notifié de la fin d'un de ses fils via `SIGCHLD` et le récupérer correctement pour éviter les processus zombies.

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <signal.h>
#include <sys/wait.h>

void handler_SIGCHLD(int sig) {
    int status;
    pid_t pid = wait(&status); // Récupère le statut du fils terminé
    printf("Fils terminé, PID=%d, status=%d\n", pid, status);
}
```

--> Suite

```c
int main() {
    signal(SIGCHLD, handler_SIGCHLD); // Enregistre le handler pour SIGCHLD

    pid_t pid = fork(); // Création d'un processus fils
    if (pid == 0) {
        // Code du processus fils
        printf("Processus fils lancé\n");
        sleep(2); // Le fils travaille 2 secondes
        exit(42); // Le fils se termine avec un code 42
    } else {
        // Code du processus parent
        printf("Processus parent attend ...\n");
        while(1) pause(); // Le parent se met en pause, attendant un signal
    }
    return 0;
}
```

Points Clés et Ressources

Points importants à retenir

- `SIGKILL` et `SIGSTOP` ne peuvent pas être capturés ni ignorés.
- Les handlers doivent éviter les fonctions non réentrantes (ex: éviter `printf` dans un handler en contexte critique).
- Utiliser `sigaction` plutôt que `signal` pour garantir portabilité et un meilleur contrôle.
- La gestion correcte des signaux évite les processus zombies et améliore la robustesse des applications multitâches.

Sources utilisées

- signal(7) — Linux manual page
- sigaction - Linux man page
- [Advanced Programming in the UNIX Environment - Stevens, Rago]
- [The Linux Programming Interface - Kerrisk]
- POSIX Signal Concepts - The Open Group

Programmation Système et Multitâche

Introduction aux Threads (Pthreads C11)

Les threads permettent d'exécuter plusieurs séquences d'instructions concouramment au sein d'un même processus, partageant la mémoire.

La bibliothèque **POSIX Threads (Pthreads)** est l'interface standard sous Unix/Linux pour la programmation multithreadée.

La fonction `pthread_create` est essentielle : elle lance un nouveau thread d'exécution.

Qu’est-ce qu’un Thread ?

- Un **thread** est une unité d’exécution plus légère qu’un processus.
- Il partage le même espace mémoire que les autres threads du processus.
- Il permet un parallélisme coopératif ou préemptif efficace.

`pthread_create` : Création d'un Thread

Prototype de la fonction

```c
#include <pthread.h>

int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
                   void *(*start_routine)(void *), void *arg);
```

Description des paramètres :

- `thread` : Pointeur vers `pthread_t` où sera stocké l'identifiant du thread créé.
- `attr` : Paramétrage optionnel du thread (peut être `NULL` pour valeurs par défaut).
- `start_routine` : Fonction exécutée par le thread (doit retourner `void *` et prendre un `void *` en paramètre). C'est le point de départ de l'exécution du thread.
- `arg` : Argument passé à la fonction `start_routine`.

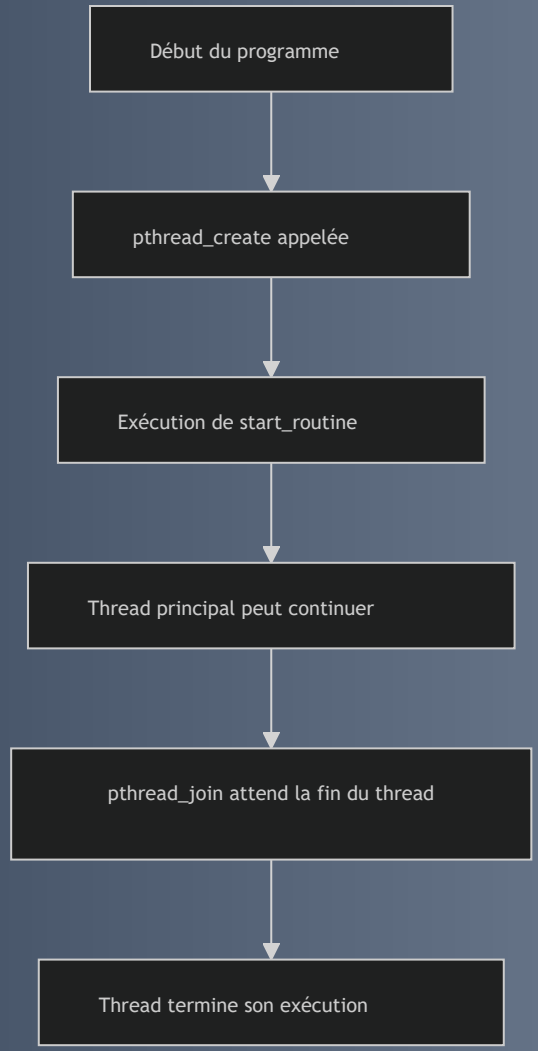
Retour : `0` en cas de succès, code d'erreur sinon.

Exemple Pratique et Synchronisation

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

void *thread_function(void *arg) {
    int id = *((int *)arg);
    printf("Thread %d lancé\n", id);
    return NULL;
}

int main() {
    pthread_t thread;
    int id = 1;

    if(pthread_create(&thread, NULL, thread_function, &id) != 0) {
        perror("Erreur pthread_create");
        return EXIT_FAILURE;
    }

    // Attendre la fin du thread
    pthread_join(thread, NULL);

    printf("Thread terminé\n");

    return EXIT_SUCCESS;
}
```

Explications :

- `start_routine` est l'entrée du thread.
- L'argument `arg` permet de passer des données (un pointeur) au thread.
- `pthread_join` permet de synchroniser : il bloque le thread appelant (ici `main`) jusqu'à la fin du thread cible.
- **Attention** : L'argument passé à `start_routine` (`&id` ici) doit rester valide tant que le thread l'utilise. Dans cet exemple, `id` est sûre car `pthread_join` assure que `main` ne se termine pas avant le thread enfant.


Début du programme
pthread_create appelée
Exécution de start_routine
Thread principal peut continuer
pthread_join attend la fin du thread
Thread termine son exécution

Bonnes pratiques essentielles :

- Toujours vérifier le code retour de `pthread_create`.
- Protéger l'accès aux ressources partagées (avec mutex, sémaphores, etc.).
- Ne pas retourner de pointeur vers des variables locales depuis `start_routine` (elles disparaîtront avec la fin de la fonction).
- Utiliser `pthread_exit` pour terminer proprement un thread si nécessaire.

Ce qu’il faut retenir & Ressources

Ce qu’il faut retenir :

Ce cours aborde la création basique de threads avec `pthread_create`, en explicitant les mécanismes d’appel et de synchronisation, essentiels pour initier la programmation multitâche efficace sous UNIX/Linux.

Sources utilisées :

- pthread_create on man7.org
- POSIX Threads Programming - IEEE Std
- Thread basics - cppreference
- Beej’s Guide to Pthreads

Programmation Système et Multitâche

Synchronisation des Threads : Mutex, Sémaphores, Race Conditions & Deadlocks

Introduction aux Threads : Gérer la Concurrence

- **Contexte :** Les threads partagent les mêmes ressources mémoire, ce qui peut engendrer des problèmes d'accès concurrent.
- **Problèmes majeurs :**
  - **Race Condition (Condition de course) :** Accès et modification simultanés d'une même donnée sans protection, menant à des résultats incohérents ou imprévisibles.
    - *Exemple :* Un compteur partagé incrémenté par plusieurs threads sans synchronisation peut donner une valeur finale erronée.
  - **Deadlock (Interblocage) :** Deux threads ou plus se bloquent mutuellement, attendant indéfiniment la libération de ressources détenues par l'autre.
- **Solution :** Utiliser des mécanismes de synchronisation comme les **mutex** et les **sémaphores** pour garantir un accès sûr et ordonné aux ressources partagées.

Mutex : Le Verrou d'Exclusion Mutuelle

- **Description :** Un **mutex** (pour Mutual Exclusion) est un verrou binaire qui protège une section critique. Il garantit qu'**un seul thread à la fois** peut accéder à cette section.

- **Interface de base (Pthreads C11) :**

```c
#include <pthread.h>

pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER; // Initialisation statique

pthread_mutex_lock(&mutex);   // Verrouillage : attend si déjà pris
// Section critique
pthread_mutex_unlock(&mutex); // Déverrouillage
```

- **Principe d'utilisation :**

  1. Déclarer et initialiser un mutex.
  2. Avant d'accéder à la ressource partagée (section critique), verrouiller le mutex (`lock`).
  3. Après avoir terminé l'accès, déverrouiller le mutex (`unlock`).

- *Objectif :* Empêcher les race conditions en assurant l'accès exclusif à une donnée à un instant T.

Sémaphore : Contrôler l’Accès Concurrentiel

- **Description :** Un **sémaphore** gère un compteur interne. Il permet à un **nombre défini de threads** d’entrer simultanément dans une section critique.
  - Si le compteur est initialisé à 1, il fonctionne comme un mutex.
  - Si le compteur est supérieur à 1, il autorise plusieurs threads en parallèle (jusqu’à la limite du compteur).
- **Interface POSIX :**

```c
#include <semaphore.h>

sem_t sem;
sem_init(&sem, 0, 1); // Initialise le compteur à 1

sem_wait(&sem);  // Opération P : décrémente le compteur, attend s'il est à zéro
// Section critique
sem_post(&sem);  // Opération V : incrémente le compteur, libère un thread en attente
```

- **Avantage :** Plus flexible que le mutex pour les scénarios où un accès limité mais concurrent est acceptable (ex: pool de connexions, buffer limité).

Deadlock : Comprendre et Prévenir l'Interblocage

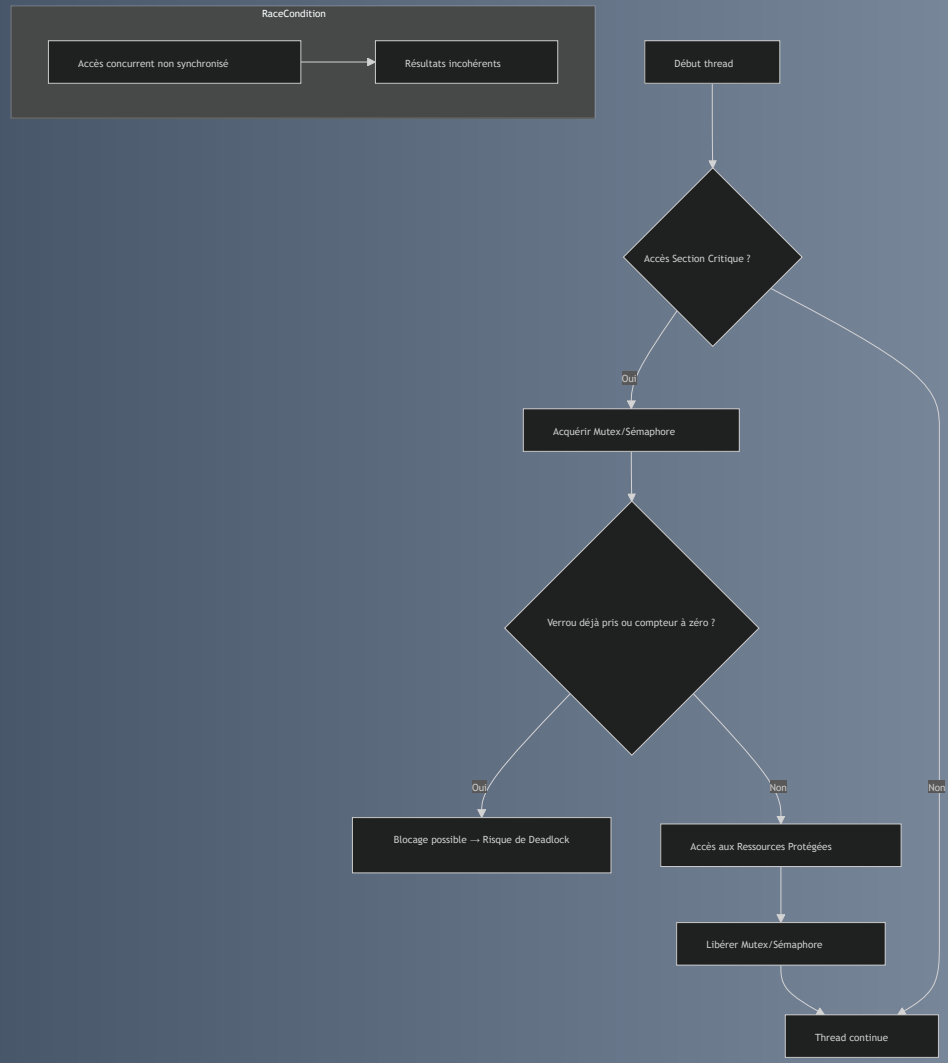
- **Mécanisme :** Se produit souvent lorsque plusieurs threads essaient d'acquérir plusieurs verrous (mutex) dans un **ordre différent**, créant une attente circulaire.

- **Conditions de Coffman (causes classiques de Deadlock) :**

  1. **Exclusion Mutuelle :** Au moins une ressource doit être non-partageable (ex: un mutex).
  2. **Attente Non Préemptive :** Une ressource allouée ne peut être retirée de force à un thread.
  3. **Attente Circulaire :** Un ensemble de threads attend chacun une ressource détenue par le suivant dans le cycle.
  4. **Ressources Détenues :** Un thread détient déjà au moins une ressource et en attend une autre.

- **Prévention Pratique :**

  - Acquérir les mutex dans un **ordre global cohérent** pour tous les threads.
  - Utiliser des fonctions comme `pthread_mutex_trylock` pour éviter le blocage définitif et permettre un retour arrière.
  - Concevoir des flux de ressources simplifiés pour minimiser les dépendances croisées.


RaceCondition
Accès concurrent non synchronisé
Résultats incohérents
Début thread
Accès Section Critique ?
Oui
Non
Acquérir Mutex/Sémaphore
Verrou déjà pris ou compteur à zéro ?
Oui
Non
Blocage possible → Risque de Deadlock
Accès aux Ressources Protégées
Libérer Mutex/Sémaphore
Non
Thread continue

Synthèse & Pour Aller Plus Loin

Points Clés à Retenir :

- **Mutex :** Essentiel pour l'exclusion mutuelle, protège les sections critiques uniques.
- **Sémaphore :** Offre un contrôle plus fin en permettant l'accès à un nombre limité de ressources.
- La gestion des **Deadlocks** requiert un ordre d'acquisition rigoureux et une conception préventive.
- Identifier et résoudre les **Race Conditions** est crucial par des tests et analyses approfondis.

La synchronisation par mutex et sémaphore est un outil incontournable pour éviter les erreurs dues à l'accès concurrent aux ressources partagées. Comprendre et gérer les problèmes classiques comme deadlock et race condition permet d'écrire des programmes multitâches robustes et prévisibles.

Sources Utilisées :

- pthread_mutex_lock - Linux man page
- sem_wait - Linux man page
- Deadlock - Wikipedia
- Race condition - Wikipedia
- Advanced Programming in the UNIX Environment - Stevens, Rago
- The Linux Programming Interface - Kerrisk

Préprocesseur Avancé :

Macros à Arguments Variables (`__VA_ARGS__`)

Introduction aux Macros Variadiques

Le préprocesseur C permet de définir des macros acceptant un nombre variable d'arguments. Cette fonctionnalité est particulièrement utile pour le débogage, le logging ou la création de fonctions wrappers génériques.

1. Principe de base

- En standards C99 et supérieurs, une macro variadique est définie ainsi :

```
#define NomMacro(arg_fixe, ...)
```

- `...` (les points de suspension) représente les arguments variables.
- Ces arguments sont accessibles dans le corps de la macro via le mot-clé spécial `__VA_ARGS__`.

Syntaxe et Premier Exemple

2. Syntaxe de base

- Utilisation courante pour remplacer `printf` :

```
#define LOG(fmt, ...) printf(fmt, __VA_ARGS__)
```

La macro `LOG` prend un format (`fmt`) et un nombre variable d'arguments.

3. Exemple concret : Macro `LOG`

```
#include <stdio.h>

#define LOG(fmt, ...) printf("[LOG] " fmt "\n", __VA_ARGS__)

int main() {
    LOG("Valeur = %d", 42); // OK
    LOG("Test sans argument variable"); // Erreur !
    return 0;
}
```

Attention : La deuxième invocation (`LOG("Test sans argument variable");`) génère une erreur car `__VA_ARGS__` est vide, laissant une virgule de trop avant la parenthèse fermante de `printf`.

Gérer l’absence d’arguments variables

4. Correction des erreurs avec `##__VA_ARGS__` Pour éviter les erreurs lorsque `__VA_ARGS__` est vide, une extension courante (supportée par GCC/Clang) est d’utiliser `##__VA_ARGS__`.

```
#define LOG(fmt, ...) printf("[LOG] " fmt "\n", ##__VA_ARGS__)
```

Le préprocesseur supprime la virgule précédant `##__VA_ARGS__` si `__VA_ARGS__` est vide. Cela permet des appels sans arguments variables sans provoquer d'erreur syntaxique.

Exemple Amélioré et Applications

5. Exemple amélioré avec `##__VA_ARGS__`

```c
#include <stdio.h>

#define LOG(fmt, ...) printf("[LOG] " fmt "\n", ##__VA_ARGS__)

int main() {
    LOG("Valeur = %d", 42);             // Fonctionne
    LOG("Test sans argument variable"); // Fonctionne aussi !
    return 0;
}
```

Avec `##__VA_ARGS__`, la macro fonctionne quelle que soit la présence d'arguments variables.

6. Applications avancées

- Macros de débogage et de trace personnalisées.
- Génération de code répétitif avec paramétrage flexible.
- Création de fonctions wrappers modulaires et adaptables.

Points Importants et Ressources

Ce qu'il faut retenir :

- `__VA_ARGS__` nécessite un compilateur compatible C99 ou supérieur.
- La syntaxe `##__VA_ARGS__` est une extension (GCC/Clang) pour gérer l'absence d'arguments.
- Évitez d'utiliser `__VA_ARGS__` dans des macros complexes sans tests rigoureux.
- Le préprocesseur ne vérifie pas les types : soyez prudent lors du passage d'arguments.

Sources utilisées :

- Variadic Macros - cppreference
- GCC Variadic Macros
- Stack Overflow - How to omit the comma in **VA_ARGS** if empty?
- ISO C99 Standard
- The C Preprocessor in Depth - RW AS

Préprocesseur Avancé : Concaténation (`##`) et Stringification (`#`)

Le préprocesseur C propose des opérateurs spéciaux pour manipuler les identificateurs et les chaînes dans les macros : la **concaténation de tokens** à l'aide de `##` et la **stringification** avec `#`.

Ces opérateurs permettent une métaprogrammation puissante, générant du code flexible et dynamique.

L'Opérateur de Concaténation :

1. Principe L'opérateur `##` fusionne (concatène) deux tokens en un seul lors du remplacement macro. Ce mécanisme est utile pour générer des noms de variables, fonctions, ou autres identificateurs de manière automatique.

2. Syntaxe

```
#define CONCAT(a, b) a ## b
```

3. Exemple simple

```c
#include <stdio.h>
#define CONCAT(a, b) a ## b

int main() {
    int xy = 42;
    printf("%d\n", CONCAT(x, y)); // équivalent à printf("%d\n", xy);
    return 0;
}
```

Ici, `CONCAT(x, y)` devient `xy`, accédant à la variable `xy`.

L’Opérateur de Stringification : `#`

1. Principe L’opérateur `#` transforme un argument macro en une chaîne de caractères (string literal). Cela sert à générer des messages, du code d’impression ou des noms convertis en chaîne.

2. Syntaxe

```
#define STRINGIFY(x) #x
```

3. Exemple simple

```
#include <stdio.h>
#define STRINGIFY(x) #x

int main() {
    printf("Le texte : %s\n", STRINGIFY(Hello World));
    return 0;
}
```

Le texte `Hello World` est converti en chaîne `"Hello World"` dans la macro.

Combinaison et Applications Pratiques

1. Exemple combiné

```c
#include <stdio.h>
#define MAKE_VAR(name, num) name ## num
#define TO_STRING(x) #x

int main() {
    int MAKE_VAR(var, 1) = 10;
    printf("Value of %s = %d\n", TO_STRING(var1), var1);
    return 0;
}
```

Sortie : `Value of var1 = 10`

2. Utilité en programmation avancée

- Générer automatiquement des noms (fonctions, variables).
- Automatiser la création de structures et d'identifiants.
- Produire des messages de débogage intégrant du code source.
- Créer des macros plus polyvalentes.

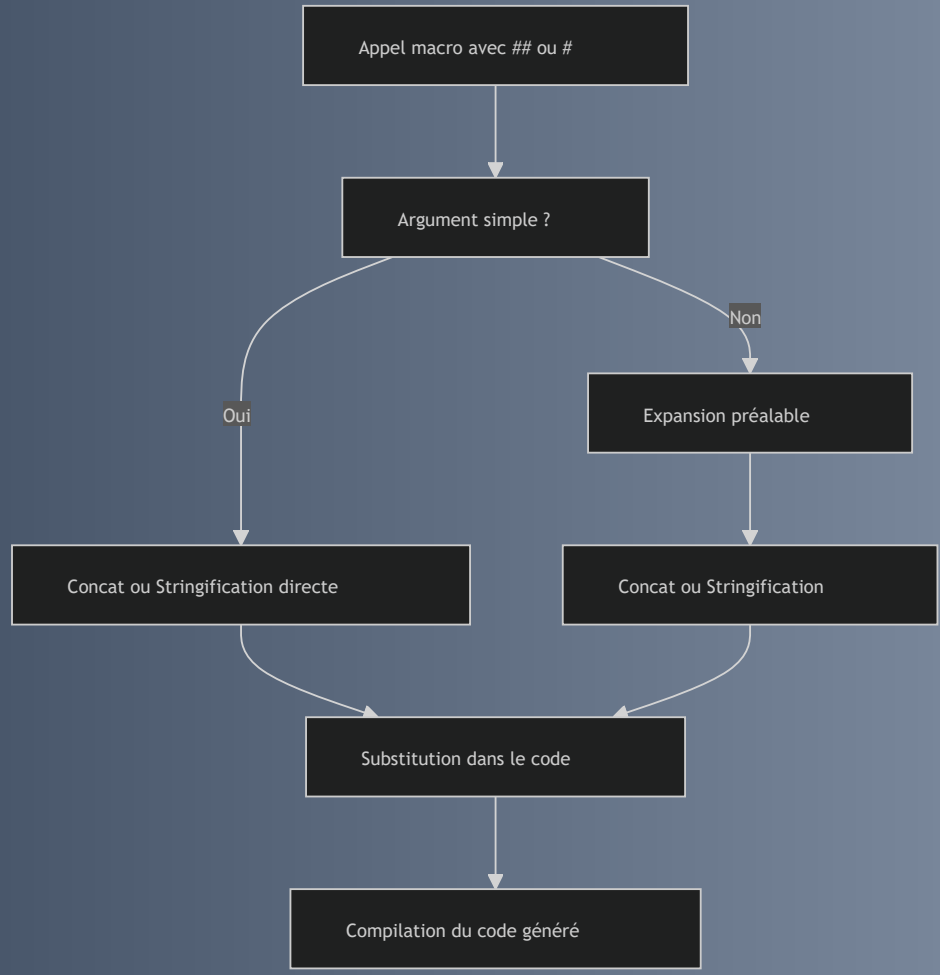
Précautions et Stratégies d’Expansion

1. Attention aux limites

- La concaténation fusionne les tokens littéralement : le résultat doit être un identificateur valide.
- Le stringification ne fait pas d’évaluation : il transforme l’argument brut en chaîne.
- L’expansion correcte peut nécessiter des macros intermédiaires.

2. Exemple avec macro intermédiaire

```c
#define STR_HELPER(x) #x
#define STR(x) STR_HELPER(x) // Pour forcer l'expansion avant stringification

#define VERSION 2
int main() {
    printf("Version corrigée : %s\n", STR(VERSION)); // affiche "2"
    return 0;
}
```


Appel macro avec ## ou #
Argument simple ?
Oui
Non
Concat ou Stringification directe
Expansion préalable
Concat ou Stringification
Substitution dans le code
Compilation du code généré

Bilan et Pour Aller Plus Loin

Ce qu'il faut retenir La concaténation et la stringification dans les macros C fournissent un véritable métalangage pour manipuler le code à la compilation, facilitant l'écriture d'outils génériques, de code auto-généré et d'améliorations robustes dans les programmes complexes.

Sources utilisées

- C Preprocessor - gcc.gnu.org
- Stringification and Token Pasting - cppreference
- The C Preprocessor in Depth - J. Handy
- ISO C Standard n1256 (chapitre 6.10.3)
- Stack Overflow : Differences between # and ##

Préprocesseur Avancé et Macros :

Inclusions Conditionnelles et Attributs (C23)

- **Le Préprocesseur C** : Un outil essentiel pour adapter la compilation, contrôler les inclusions et ajuster le comportement du compilateur.
- **Directives Clés** : `#if`, `#ifdef`, `#ifndef`, `#error`, `#pragma`.
- **Objectif** : Écrire du code portable, sûr et maintenable.

Contrôler l’inclusion de code : `#if`, `#ifdef`, `#ifndef`

- `#if EXPRESSION`
  - Évalue une expression entière à la compilation.
  - Inclut ou exclut du code selon que l’expression est non nulle ou nulle.

```
#define VERSION 2

#if VERSION ⩾ 2
    // Code pour version 2 ou plus
#else
    // Version antérieure
#endif
```

- `#ifdef MACRO`
  - Inclut le code si la macro est définie.
- `#ifndef MACRO`
  - Inclut le code si la macro n’est pas définie.

```
#ifdef DEBUG
    printf("Mode debug activé\n");
#endif

#ifndef RELEASE
    // Code pour non-release
#endif
```

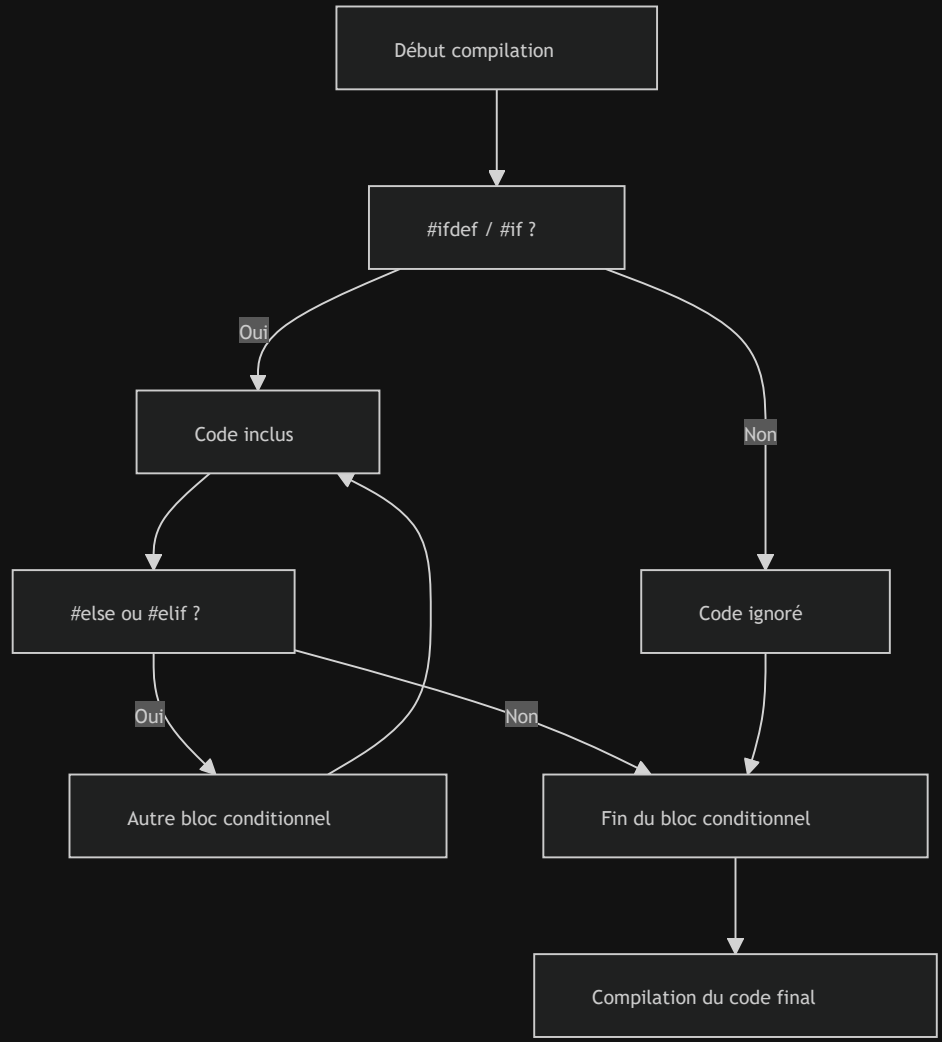
Gérer les erreurs et les comportements spécifiques du compilateur

- `#error`
  - Génère une erreur de compilation avec un message personnalisé.
  - **Utilité** : Imposer des prérequis ou interdire des configurations spécifiques.

```
#ifndef MIN_VERSION
#error "MIN_VERSION doit être défini"
#endif
```

- `#pragma`
  - Directive spécifique au compilateur pour des ordres spéciaux.
  - Varie selon les compilateurs, mais la norme C23 introduit des attributs standards.
  - **Exemples courants :**
    - `#pragma once` (inclusion unique de fichier - GCC/Clang)
    - `#pragma GCC poison strcpy` (interdiction d'utiliser `strcpy` - GCC/Clang)
  - **C23 :** Normalisation de directives `#pragma` pour des attributs comme `assume`, permettant au compilateur d'optimiser.

```c
#include <stdio.h>

#define VERSION 1

#ifndef FEATURE_ENABLED
#error "FEATURE_ENABLED doit être défini" // Vérifie un prérequis
#endif

#if VERSION >= 2 // Code conditionnel selon la version
void new_feature() {
    printf("Nouvelle fonctionnalité activée\n");
}
#else
void old_feature() {
    printf("Ancienne fonctionnalité\n");
}
#endif

int main() {
#ifdef FEATURE_ENABLED // Active/désactive une fonctionnalité
#if VERSION >= 2
    new_feature();
#else
    old_feature();
#endif
#else
    printf("Feature désactivée\n");
#endif
```


Début compilation
#ifdef / #if ?
Oui
Non
Code inclus
Code ignoré
#else ou #elif ?
Oui
Non
Autre bloc conditionnel
Fin du bloc conditionnel
Compilation du code final

Points clés à retenir :

- **Robustesse :** Toujours vérifier les macros critiques avec `#ifndef` + `#error`.
- **Inclusion :** Utiliser `#pragma once` (si supporté) pour éviter les doubles inclusions.
- **Portabilité :** Les directives `#pragma` sont spécifiques ; vérifier la documentation du compilateur. La norme C23 vise à améliorer cette portabilité.

Ressources :

- cppreference - Preprocessor directives
- GCC Preprocessor - Pragmas
- C23 draft standard - WG14
- Microsoft Docs - #pragma (C/C++)
- Stack Overflow - Using #error

Préprocesseur Avancé et Macros

Introduction aux Attributs C23

La norme C23 introduit des **attributs standards** pour améliorer la clarté du code et les messages du compilateur.

Deux attributs clés sont :

- `[[maybe_unused]]` : Pour marquer des variables ou fonctions potentiellement inutilisées.
- `[[fallthrough]]` : Pour documenter le comportement intentionnel des `switch` avec chute de cas (*fallthrough*).

Ces attributs aident à produire un code plus propre et sans avertissements inutiles.

L’Attribut `[[maybe_unused]]`

But : Indique qu’une déclaration (variable, fonction, type) peut ne pas être utilisée sans générer d’avertissement de compilation. Utile pour :

- Un code plus propre, sans warnings superflus.
- Des conditions de compilation où l’utilisation varie selon le contexte.

Syntaxe :

```
[[maybe_unused]] int x;
[[maybe_unused]] static void foo(void);
```

Exemple :

```c
#include <stdio.h>

[[maybe_unused]] static int valeur_non_utilisee;

void fonction() {
    int a = 5;
    // valeur_non_utilisee pas utilisée ici, pas d'avertissement
    printf("a = %d\n", a);
}
```

Sans l'attribut, la plupart des compilateurs signaleraient `valeur_non_utilisee` *comme non utilisée.*

L'Attribut `[[fallthrough]]`

But : Indique explicitement qu'un `case` dans une instruction `switch` doit "tomber" (*fall through*) dans le `case` suivant.

- Prévient les avertissements des compilateurs modernes sur un `fallthrough` souvent accidentel.

Syntaxe :

```
switch(value) {
    case 1:
        // code
        [[fallthrough]]; // Intention de chuter dans le cas 2
    case 2:
        // code lorsque cas 1 tombe sur cas 2
        break;
}
```

Exemple :

```c
#include <stdio.h>

void test(int val) {
    switch(val) {
        case 1:
            printf("Cas 1\n");
            [[fallthrough]];
        case 2:
            printf("Cas 2\n");
            break;
        default:
            printf("Autre cas\n");
    }
}
// test(1) affiche "Cas 1" puis "Cas 2"
// test(2) affiche "Cas 2" uniquement
```

Avantages :

- **Réduction du bruit** dans les messages d’avertissement.
- **Documentation implicite** des intentions du programmeur.
- Amélioration de la **qualité et maintenance** du code.
- Harmonisation avec les pratiques modernes des compilateurs.

Bénéfices et Intégration dans l'Écosystème C/C++

Compatibilité :

- Ces attributs sont repris dans **C++17** et versions ultérieures.
- Support aujourd'hui dans :
  - GCC (depuis 9)
  - Clang (depuis 7)
  - MSVC (depuis Visual Studio 2019)
- Utiliser `__has_cpp_attribute` pour vérifier leur disponibilité.

Ce qu’il faut retenir & Pour aller plus loin

Ce qu’il faut retenir :

- Avec `[[maybe_unused]]` et `[[fallthrough]]`, la norme C23 formalise des pratiques existantes en C++ et comme extensions compilateurs.
- Leur usage explicite permet de produire des programmes plus explicites et robustes.
- Ils évitent les faux positifs dans les outils d’analyse statique et de compilation.

Sources utilisées :

- ISO/IEC JTC1 SC22 WG14, Draft C23 standard
- GCC Attributes
- Clang Attributes Reference
- cppreference - [[maybe_unused]]
- cppreference - [[fallthrough]]

Debugging Avancé avec GDB :

Des Outils pour une Analyse Précise

Introduction : Le débogueur GDB est un outil puissant pour analyser et corriger les programmes en C. Pour gagner en efficacité lors du diagnostic, il est possible d'utiliser des fonctionnalités avancées telles que les **breakpoints conditionnels**, les **watchpoints** et l'intégration de scripts **Python** pour automatiser et étendre le débogage.

Breakpoints Conditionnels :

Quand s’arrêter, et pourquoi

1. Principe : Un breakpoint conditionnel suspend l’exécution uniquement quand une condition spécifiée est vraie, permettant d’ignorer des itérations inutiles ou d’autres situations non pertinentes.

2. Syntaxe :

```
break <fonction|ligne> if <condition>
```

3. Exemple :

```
(gdb) break my_function if x == 42
```

L’exécution s'arrêtera dans `my_function` seulement si la variable `x` vaut 42.

Watchpoints :

Surveiller les Changements de Données

1. Principe : Surveille un emplacement mémoire (variable ou expression) et interrompt l'exécution lorsque cette valeur change (lecture ou écriture selon le type).

2. Commandes principales :

- `watch <expression>` : s'arrête à chaque modification de `<expression>`.
- `rwatch <expression>` : s'arrête quand l'expression est lue.
- `awatch <expression>` : s'arrête quand l'expression est lue ou écrite.

3. Exemple :

```
(gdb) watch compteur
```

L'exécution se bloque dès que `compteur` est modifié.

Scripting Python dans GDB :

Automatisation et Personnalisation

1. Utilités : GDB embarque un interpréteur Python, permettant :

- D'écrire des commandes personnalisées.
- D'automatiser des tâches répétitives.
- D'étendre le débogueur (analyse spécifique, affichage personnalisé).

2. Exemple simple : définir une commande Python

```
(gdb) python
> class Hello(gdb.Command):
>     # ... (implémentation de la commande Hello) ...
> Hello()
> end
(gdb) hello world
Hello from Python in GDB! Arg: world
```

3. Exemple avancé : affichage personnalisé d'une structure

- Création d'un "pretty printer" Python pour afficher des structures complexes de manière lisible (ex: `struct Point`).
- Améliore la compréhension des données spécifiques à votre application.

GDB Avancé :

Synthèse et Cycle d’Analyse

Résumé des commandes utiles :

Commande GDB	Description
`break <loc> if <cond>`	Point d’arrêt conditionnel
`watch <var>`	Surveillance écriture sur variable
`rwatch <var>`	Surveillance lecture sur variable
`awatch <var>`	Surveillance lecture/écriture sur variable
`python ... end`	Définir commandes/logiciel via scripts Python

Maîtriser GDB :

Efficacité accrue du Débogage

Ce qu'il faut retenir : L'exploitation avancée des breakpoints conditionnels, watchpoints et commandes Python dans GDB permet d'affiner le processus de débogage, réduisant le temps d'identification des anomalies et augmentant la précision d'observation des comportements internes du programme.

Sources utilisées :

- GDB Documentation - Breakpoints
- GDB Documentation - Watchpoints
- GDB Python API
- Tutorialspoint - GDB Watchpoints
- Embedded Artistry - Python scripting in GDB

Debugging, Profiling et Optimisation

Introduction à `gprof` et `valgrind`

Introduction

Pour améliorer les performances et la fiabilité des programmes, il est crucial d'analyser leur comportement en temps d'exécution.

Deux outils majeurs pour cette tâche :

- `gprof` : Pour le profilage CPU (analyse des performances).
- `valgrind` : Pour la détection des erreurs et fuites mémoire.

`gprof` : Profilage de Performances

Fonctionnement : `gprof` analyse les performances de code compilé avec l'option `-pg`. Il mesure :

- Le temps d'exécution par fonction.
- Le nombre d'appels à chaque fonction.
- La hiérarchie d'appels (`call graph`).

Utilisation :

1. **Compilation :**
```
gcc -pg -o mon_programme mon_programme.c
```
2. **Exécution :** (Produit `gmon.out`)
```
./mon_programme
```
3. **Analyse :** (Génère `rapport.txt` avec statistiques et `call graph`)
```
gprof mon_programme gmon.out > rapport.txt
```

Exemple : Pour un code avec `fonction_lente()` :

```
gcc -pg -o exemple exemple.c
./exemple
gprof exemple gmon.out > profil.txt
less profil.txt
```

`valgrind` : Détection des Erreurs Mémoire et Fuites

Fonctionnement : Valgrind exécute un programme en instrumentant son accès mémoire pour détecter :

- Fuites mémoire non libérées.
- Utilisations de mémoire non initialisée.
- Accès hors limites.
- Double libération, etc.

Commande de base : L'option `--leak-check=full` affiche les détails des fuites mémoire avec pile d'appel.

```
valgrind --leak-check=full ./mon_programme
```

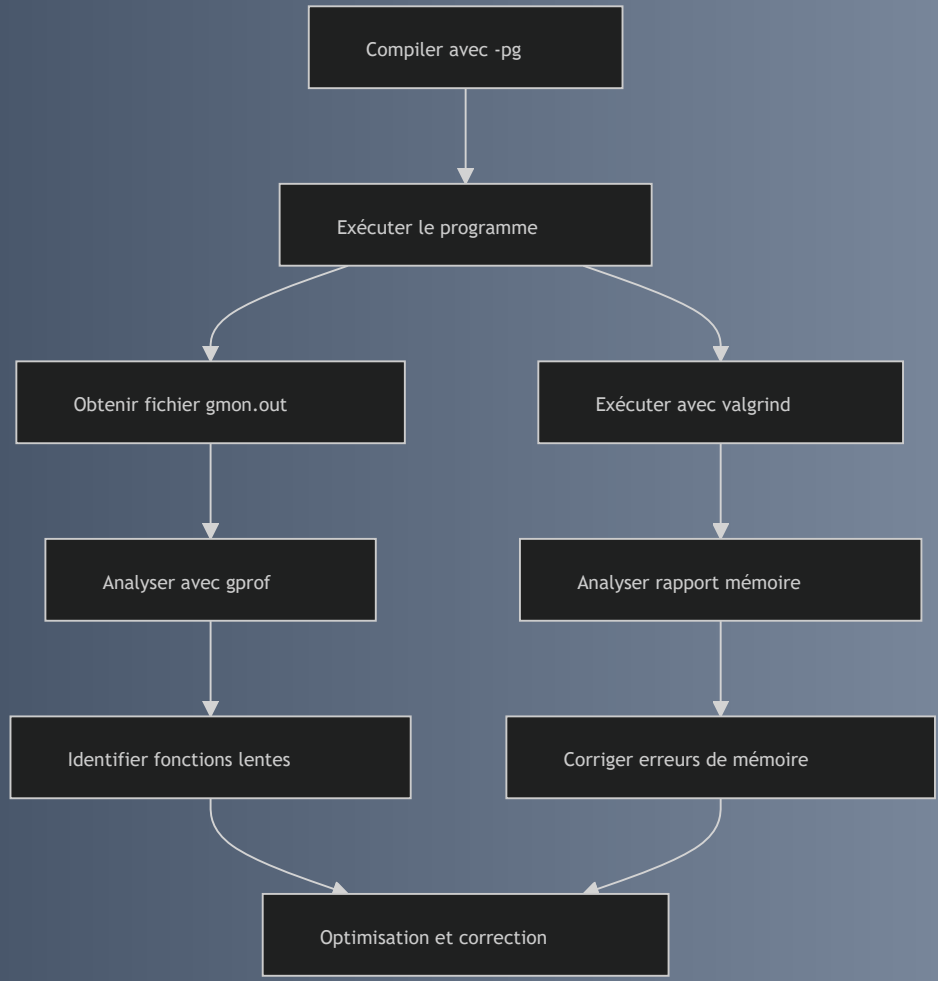
Exemple : Pour un code avec `malloc` mais sans `free` (fuite mémoire) :

```
valgrind --leak-check=full ./mon_programme
```

=> Rapport indiquant la fuite mémoire avec localisation dans le code.

Complémentarité des Outils

Outil	Usage principal	Points forts
`gprof`	Profilage CPU	Visualisation du temps et des appels
`valgrind`	Analyse mémoire (fuites, erreurs)	Détection précise et pile d’appel


Compiler avec -pg
Exécuter le programme
Obtenir fichier gmon.out
Exécuter avec valgrind
Analyser avec gprof
Analyser rapport mémoire
Identifier fonctions lentes
Corriger erreurs de mémoire
Optimisation et correction

Bonnes Pratiques

- `gprof` : Toujours profiler un programme compilé avec `-pg` pour la validité des résultats.
- `valgrind` : Utiliser Valgrind dès que vous soupçonnez des problèmes de mémoire (programmes longs, allocation dynamique intensive).
- **Couplage :** Combiner les résultats des deux outils pour optimiser efficacement (corriger les erreurs mémoire *avant* d'améliorer les performances).
- **Précision :** Garder à l'esprit que la précision dépend du compilateur, de l'architecture et de la charge système.

Ce qu’il faut retenir & Sources

Le tandem de `gprof` et `valgrind` constitue une base complémentaire puissante pour diagnostiquer la performance et la qualité mémoire d’un programme C, facilitant ainsi la correction ciblée et l’optimisation effective du code.

Sources utilisées :

- GNU gprof Manual
- Valgrind Official Documentation
- Tutorialspoint - gprof
- Tutorialspoint - Valgrind
- Stack Overflow - When to use gprof vs valgrind

Profiling & Optimisation en C : Boucles et Fonctions Inline

- L’optimisation du code C améliore significativement les performances d’un programme.
- Cible principale : Les **boucles** (structures de contrôle) et la gestion des **fonctions**.
- **Objectif :** Atteindre une efficacité maximale sans compromettre la clarté et la maintenabilité du code.

Optimisation des Boucles : Réduire l'overhead

- **Pourquoi optimiser les boucles ?**
  - Elles représentent souvent les portions les plus exécutées du programme.
  - Leur optimisation réduit drastiquement le temps d'exécution.
- **Technique clé : Le Déroulement (Loop Unrolling)**
  - **Principe :** Répéter explicitement le corps de la boucle plusieurs fois.
  - **Bénéfice :** Réduit le nombre total d'itérations, diminuant ainsi les coûts liés au test de condition et aux sauts.

Exemple : Déroulement par 4

```
// Avant :
for (int i=0; i<8; i++) {
    array[i] *= 2;
}

// Après :
for (int i=0; i<8; i+=4) { // Moins de vérifications de condition
    array[i] *= 2;
    array[i+1] *= 2;
    array[i+2] *= 2;
    array[i+3] *= 2;
}
```

Optimisation des Boucles : Calculs Invariants & Accès Mémoire

- **Minimisation des calculs internes**

  - **Principe :** Factoriser les calculs qui ne changent pas d'une itération à l'autre (calculs invariants) hors de la boucle.
  - **Bénéfice :** Évite des calculs redondants à chaque itération.

  Exemple :

```
// Moins efficace : `expensive_computation()` répétée N fois
for (int i=0; i<n; i++) {
    int val = expensive_computation();
    process(array[i], val);
}

// Plus efficace : `expensive_computation()` calculée une seule fois
int val = expensive_computation();
for (int i=0; i<n; i++) {
    process(array[i], val);
}
```

- **Accès mémoire optimisé**

  - **Principe :** Structurer les accès aux données pour qu'ils soient séquentiels.
  - **Bénéfice :** Profite de l'architecture du cache processeur, réduisant les latences mémoire.

Fonctions `inline` : Réduire le coût des appels

- **Principe de fonctionnement**
  - Demande au compilateur d'insérer directement le corps de la fonction à l'endroit de son appel.
  - Évite le coût lié à l'appel de fonction (gestion de pile, sauts).
- **Syntaxe simple**

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

- **Avantages clés**
  - Réduction significative du temps d'appel de fonction, surtout pour les fonctions courtes et fréquemment utilisées.
  - Facilite l'application d'optimisations supplémentaires par le compilateur.
- **Limitations et bonnes pratiques**
  - Ne pas abuser avec des fonctions volumineuses : peut augmenter la taille du code (code bloat) et dégrader le cache instruction.
  - `inline` n'est qu'une suggestion au compilateur ; il peut l'ignorer si l'optimisation n'est pas jugée bénéfique.

Synergie : Boucles Déroulées & Fonctions Inline

L'efficacité maximale est souvent atteinte en combinant ces techniques.

Exemple Combiné : Calcul de la somme des carrés

```c
#include <stdio.h>

inline int square(int x) { // Fonction courte et inline
    return x * x;
}

int main() {
    int array[8] = {1,2,3,4,5,6,7,8};
    int sum = 0;

    // Boucle déroulée avec fonction inline
    for (int i=0; i<8; i+=4) { // Réduction des itérations
        sum += square(array[i]);
        sum += square(array[i+1]);
        sum += square(array[i+2]);
        sum += square(array[i+3]);
    }
    printf("Sum of squares = %d\n", sum);
    return 0;
}
```

Maîtrise des Performances : Bilan et Outils

Ce qu'il faut retenir :

- L'optimisation des boucles et l'utilisation judicieuse des fonctions `inline` améliorent sensiblement les performances des programmes C.
- Elles permettent de maintenir un code clair et modulaire.
- Ces techniques sont un levier puissant pour corriger les goulets d'étranglement identifiés lors de la phase de *profiling*.

Ressources complémentaires :

- GCC Inline Functions
- LLVM Loop Optimization Passes
- The Art of Loop Optimization
- ISO C11 Standard — Inline Functions
- Stack Overflow - When to use inline functions?

Bonnes Pratiques C : Nommage & Gestion des Erreurs

La qualité et la maintenabilité d’un programme C dépendent de pratiques rigoureuses.

Objectifs des Conventions de Nommage

- Faciliter la lecture, compréhension et maintenance du code.
- Éviter les collisions entre symboles.
- Donner des indices sur le rôle ou le type d’une entité.

Importance du Préfixe pour les Bibliothèques

- Utiliser un préfixe unique (ex: `mylib_`) pour les symboles d’une bibliothèque afin d’éviter les conflits.

Pratiques courantes

Élément	Convention recommandée	Exemple
Variables	`snake_case` (minuscules, séparées par `_`)	`total_count`, `buffer_size`
Constantes	`SCREAMING_SNAKE_CASE` (majuscules avec `_`)	`MAX_BUFFER_SIZE`, `ERROR_CODE`
Fonctions	`snake_case` (minuscules, peuvent utiliser `_`)	`read_data()`, `compute_sum()`
Types (struct, enum)	Majuscule initiale (`CamelCase`)	`FileContext`, `HttpStatus`
Macros	`SCREAMING_SNAKE_CASE` (majuscules avec `_`)	`ASSERT`, `MIN`

Principe des Codes de Retour

- Une fonction signale son succès ou son échec via une valeur entière.
- `0` indique souvent le succès (convention POSIX).
- Des valeurs négatives ou positives spécifiques indiquent le type d'erreur.

Exemple Classique

```c
int ouvrir_fichier(const char *nom) {
    FILE *f = fopen(nom, "r");
    if (f == NULL) {
        return -1; // Échec
    }
    // ... Traitement ...
    fclose(f);
    return 0; // Succès
}
```

Avantages

- Simple et standardisé.
- Facile à vérifier après chaque appel de fonction.
- Compatible avec les API POSIX et les systèmes Unix.

Gestion des Erreurs : Utilisation de `errno`

Principe de `errno`

- `errno` est une variable globale (définie dans `<errno.h>`), modifiée par les fonctions système en cas d'erreur.
- Elle fournit un code d'erreur précis (ex: `EACCES`, `ENOMEM`).

Exemple d'utilisation

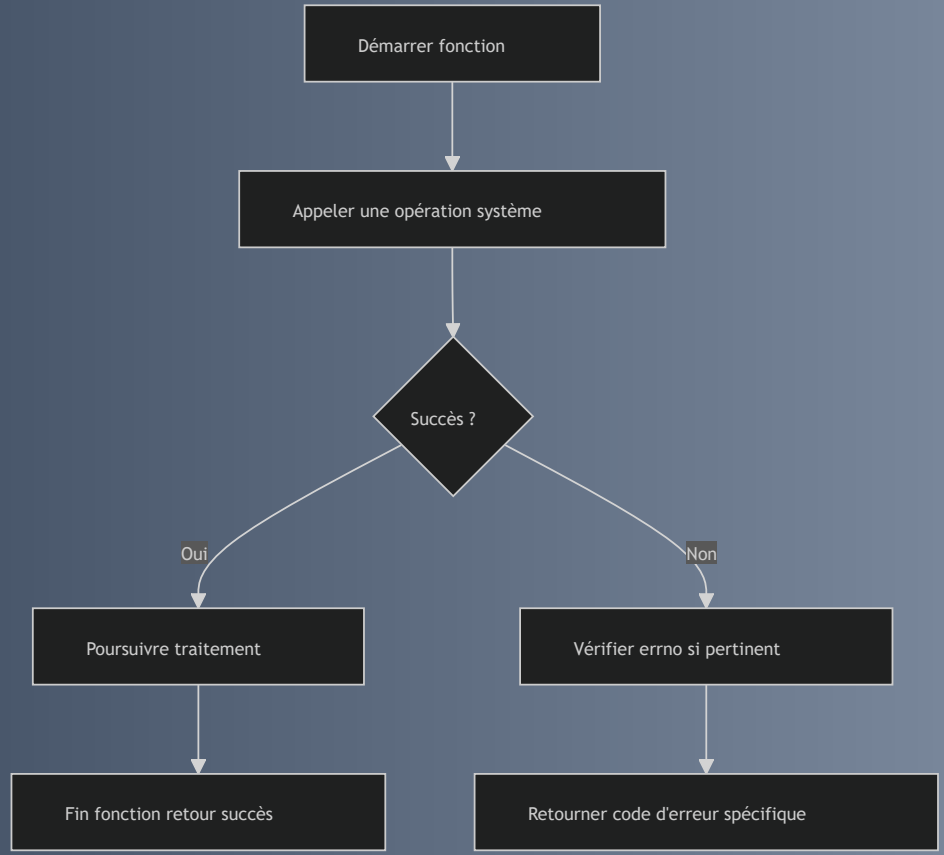
```c
#include <stdio.h>
#include <errno.h>
#include <string.h> // Pour strerror

void ouvrir_fichier_avec_errno(const char *nom) {
    FILE *f = fopen(nom, "r");
    if (!f) {
        printf("Erreur ouverture fichier : %s\n", strerror(errno));
    } else {
        // ...
        fclose(f);
    }
}
```

Manipulation Prudente de `errno`

- Toujours initialiser `errno = 0` avant un appel système sensible.
- Ne tester `errno` que si la fonction appelée indique un échec (par son code de retour).

```c
#include <stdio.h>
#include <errno.h>
#include <string.h>
#define MYLIB_ERROR_OPEN_FILE -1
#define MYLIB_ERROR_MEMORY   -2
#define MYLIB_SUCCESS          0

int mylib_load_data(const char *filename) {
    errno = 0; // Réinitialiser errno
    FILE *file = fopen(filename, "r");
    if (!file) return MYLIB_ERROR_OPEN_FILE;

    void *buffer = malloc(1024);
    if (!buffer) { fclose(file); return MYLIB_ERROR_MEMORY; }

    // Traitement ...
    free(buffer); fclose(file);
    return MYLIB_SUCCESS;
}

int main() {
    int status = mylib_load_data("data.txt");
    if (status ≠ MYLIB_SUCCESS) {
        if (status == MYLIB_ERROR_OPEN_FILE)
            printf("Erreur ouverture fichier : %s\n", strerror(errno));
        else if (status == MYLIB_ERROR_MEMORY)
            printf("Erreur allocation mémoire\n");
    } else { printf("Chargement réussi\n"); }
```

Flux de Gestion d'Erreur



Robustesse et Lisibilité : Ce qu'il faut retenir

En bref La cohérence des noms et la gestion rigoureuse des erreurs à l'aide de codes de retour et de `errno` renforcent la robustesse et la lisibilité des programmes C, facilitant ainsi leur maintenance et leur évolution.

Sources utilisées

- GNU Coding Standards - Naming Conventions
- The C Programming Language, Kernighan & Ritchie
- Linux errno man pages
- POSIX Error Numbers
- Stack Overflow - Naming conventions in C
- Michael Kerrisk, The Linux Programming Interface

Voici une présentation de 6 slides, conçue pour être accessible et percutante, en se basant strictement sur le texte fourni.

Slide 1

Documentation du code : Clé de la compréhension logicielle

La **documentation du code** est essentielle pour garantir la compréhension, la maintenance et la réutilisation du logiciel. Elle explique le *pourquoi* et le *comment* des décisions prises.

Ses rôles fondamentaux :

- **Clarté** : Explique la logique et les objectifs des parties complexes.
- **Maintenance** : Facilite la correction de bugs et l'ajout de fonctionnalités.
- **Collaborativité** : Permet à une équipe de travailler efficacement.
- **Transfert de compétences** : Réduit la dépendance aux individus.
- **Référencement automatique** : Permet la génération de documents techniques via outils.

Slide 2

Formes variées de documentation

La documentation peut prendre diverses formes, chacune adaptée à un besoin spécifique pour améliorer la lisibilité et l'utilisabilité du code.

Type	Description	Exemple
Commentaires inline	Explications courtes dans le code (ligne ou bloc)	`// Calcul de la moyenne`
Documentation des fonctions	Description des paramètres, retours, effets, préconditions	`/** Additionne deux entiers */`
Documentation externe	Guides utilisateurs, diagrammes d'architecture	Wiki, README, Doxygen généré doc
Nom et conventions	Identifier les variables/fonctions de façon parlante	`calculate_sum()` vs `cs()`

Commenter efficacement : Les bonnes pratiques

Pour une documentation utile et durable, adoptez ces principes clés :

- **Commenter ce qui est utile, pas l'évident**

```
int total = a + b; // addition de a et b (inutile à commenter)
```

- **Utiliser un format standard pour documenter les fonctions (exemple Doxygen)**

```
/**
 * @brief Calcule la somme de deux entiers.
 * @param a Premier entier
 * @param b Deuxième entier
 * @return Somme de a et b
 */
int sum(int a, int b) {
    return a + b;
}
```

- **Mettre à jour la documentation** lors des modifications. Ne pas laisser de commentaire obsolète qui induit en erreur.

Automatisation et exemple pratique

Optimisez la documentation grâce aux outils et illustrez son application concrète :

- **Utiliser des outils automatiques**
  - Doxygen, Sphinx pour générer des documents.
  - Intégration dans CI/CD pour vérifier la conformité.
- **Exemple concret : Documentation de fonction C**

```c
/**
 * @brief Ouvre un fichier en mode lecture.
 *
 * Cette fonction tente d'ouvrir le fichier spécifié et retourne
 * un pointeur sur FILE ou NULL si l'ouverture échoue.
 *
 * @param filename Chemin vers le fichier
 * @return FILE* pointeur sur le fichier ouvert, ou NULL en cas d'erreur
 */
FILE* open_file(const char* filename) {
    FILE* f = fopen(filename, "r");
    if (!f) {
        perror("Erreur ouverture fichier");
    }
```

Slide 5

Visualiser le processus de documentation

Un processus structuré garantit une documentation complète et à jour, optimisant le cycle de développement.

```
Error: Parse error on line 6:
...ser format standard (Doxygen, etc.)]
----------------------^
Expecting 'SQE', 'DOUBLECIRCLEEND', 'PE', '-)', 'STADIUMEND', 'SUBROUTINEEND', 'PIPE', 'CYLINDEREND', 'DIAMOND_STOP', 'TAGEND',
'TRAPEND', 'INVTRAPEND', 'UNICODE_TEXT', 'TEXT', 'TAGSTART', got 'PS'
```

Slide 6

Synthèse & Ressources pour aller plus loin

Ce qu'il faut retenir : La documentation bien construite rend le code plus accessible, réduit les erreurs liées aux incompréhensions et accélère les cycles de développement en facilitant la collaboration et la maintenance.

Sources utiles :

- Doxygen Manual
- Google C++ Style Guide - Comments (applicable au C)
- Linux kernel documentation style
- Stack Overflow - Best practices for commenting
- [The Art of Readable Code, Dustin Boswell & Trevor Foucher]

Voici une présentation de 6 slides, conçue pour être pédagogique, accessible et engageante, en s’appuyant uniquement sur le texte fourni.

Design Patterns en C : Singleton & Factory

Structurer et optimiser le code procédural

- Les design patterns, reconnus en programmation orientée objet, s’adaptent au langage C pour structurer et optimiser le code.
- Nous abordons deux patterns fondamentaux : **Singleton** et **Factory**.
- Leur implémentation effective en C permet une meilleure organisation et maintenance.

Le Pattern Singleton : Une Instance Unique

Objectif :

- Garantir qu'une seule instance d'une structure de données (ex: configuration, logger) existe pendant tout le cycle de vie du programme.

Implémentation en C :

- Utilise une variable `static` dans un fichier source (`singleton_logger`).
- Une fonction d'accès globale (`get_logger`) assure l'initialisation unique.

```c
// logger.c - extrait conceptuel
static Logger singleton_logger; // L'instance unique

Logger* get_logger(void) {
    static int initialized = 0;
    if (!initialized) {
        singleton_logger.log = log_impl; // Initialisation
        initialized = 1;
    }
    return &singleton_logger;
}
```

Bénéfice : Accès global et garanti à une ressource unique sans dupliquer.

Le Pattern Factory : Création Abstraite d'Objets

Objectif :

- Dissocier la création d'objets de leur utilisation.
- Permet de gérer différentes variantes d'objets sans exposer leur création complexe.

Implémentation en C :

- Une fonction "factory" retourne des pointeurs sur des structures.
- Ces structures respectent une interface commune (ex: pointeur de fonction `draw`).

```c
// shape.h - Interface commune
typedef struct Shape {
    void (*draw)(struct Shape *); // Fonction de dessin générique
} Shape;

// Fonctions factory pour créer des formes spécifiques
Shape* create_circle(void);
Shape* create_rectangle(void);
```

Bénéfice : Améliore la modularité et l'extensibilité en masquant la logique de création.

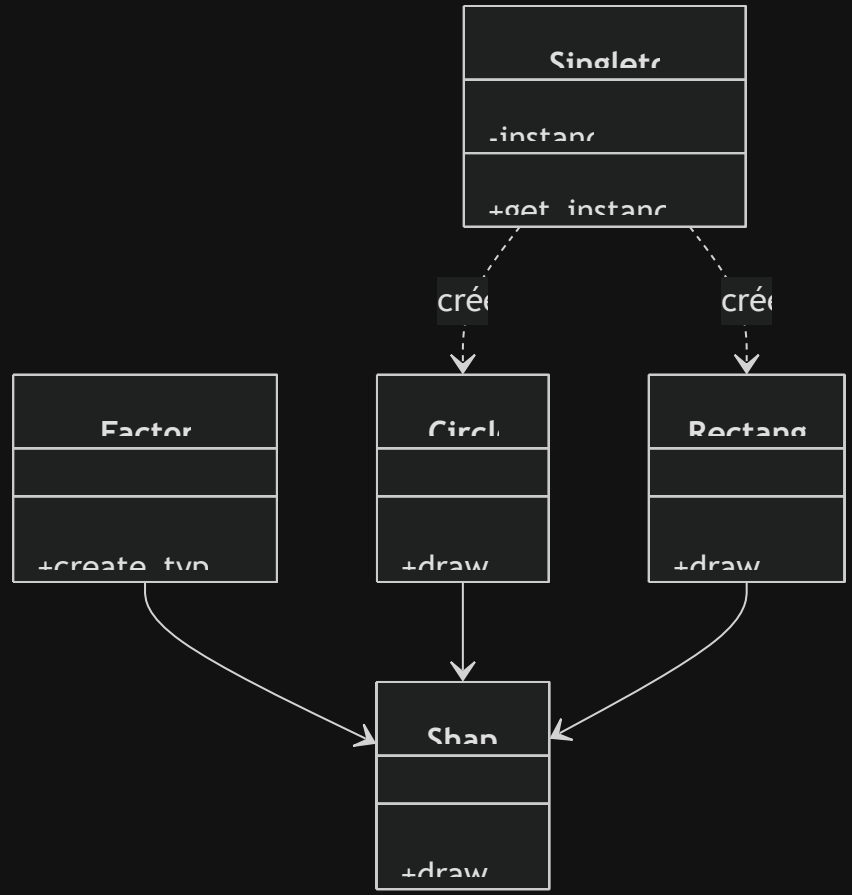
Sécurité et Gestion de la Concurrence

Contexte Multithread :

- **Singleton :** La création doit être protégée contre les appels concurrents (ex: utilisation de `pthread_mutex_lock` lors de l'initialisation).
- **Factory :** Doit garantir qu'aucune ressource partagée critique n'est corrompue lors de la création d'objets.

Gestion de la Mémoire :

- Toujours vérifier les retours d'allocation mémoire (`malloc`) pour éviter des déréférencements nuls et les plantages.



- Ce diagramme illustre les relations entre les patterns :
  - Le **Singleton** gère une instance unique.

L’Essentiel à Retenir & Ressources Utiles

Ce qu’il faut retenir :

- Le **Singleton** assure un accès global et unique à une ressource.
- Le **Factory** masque la complexité des créations d’objets.
- Ces patterns améliorent la modularité et l’extensibilité, même dans un langage procédural comme C.

Sources utilisées :

- Gang of Four Design Patterns (adaptation principes)
- [Modern C Design, Andrei Alexandrescu]
- Embedded.com - Design patterns in C
- GitHub examples Singleton and Factory in C
- Stack Overflow - Implementing Singleton in C

Voici une présentation structurée en 6 slides, conçue pour être pédagogique et accessible, en utilisant uniquement les informations fournies.

Sécurité en C : Buffer et Integer Overflows

Gestion des vulnérabilités critiques

Le langage C, avec son accès bas niveau et sa gestion manuelle de la mémoire, est particulièrement exposé à des vulnérabilités classiques. Les **buffer overflows** et les **integer overflows** peuvent provoquer des comportements erratiques, des corruptions mémoire, et ouvrir la porte à des attaques de sécurité.

Ce que nous allons explorer :

- Comprendre les mécanismes des buffer et integer overflows.
- Identifier leurs conséquences et les risques associés.
- Mettre en œuvre des techniques de prévention et de bonnes pratiques.

Comprendre le Buffer Overflow

Quand l'écriture dépasse les limites

1. Principe : Un buffer overflow survient lorsqu'un programme tente d'écrire au-delà des limites allouées d'une zone mémoire (buffer), altérant des données adjacentes ou le flux d'exécution.

2. Exemple classique :

```
void vulnerable_function(char *input) {
    char buffer[10];
    strcpy(buffer, input);  // Pas de vérification de taille
}
// Avec un input trop long ("AAAAAAAAAAAAAAAAAAAA"), le buffer est débordé.
```

Conséquences : * Corruption mémoire, pouvant entraîner un crash. * Exécution de code arbitraire par un attaquant.

Prévention :

- **Fonctions sécurisées :** `strncpy()`, `snprintf()`.
- **Vérification systématique :** Toujours vérifier la taille avant toute copie.
- **Outils d'analyse :** Utiliser Valgrind, AddressSanitizer (ASan).

L'Integer Overflow et ses Risques

Le débordement silencieux des calculs

1. Principe : Un integer overflow se produit lorsque le résultat d'une opération arithmétique dépasse la capacité maximale du type entier utilisé. Cela cause un débordement silencieux, souvent avec un résultat inattendu.

2. Exemple :

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

int main() {
    unsigned int a = UINT_MAX; // Valeur maximale pour unsigned int
    unsigned int b = a + 1;    // Dépassement, b revient à 0
    printf("Résultat : %u\n", b); // Affiche 0
    return 0;
}
```

Risques : * **Erreurs logiques :** Mauvais calculs impactant le déroulement du programme. * **Vulnérabilités :** Erreur d'allocation mémoire si la taille calculée est incorrecte.

Prévention :

- **Types adaptés :** Utiliser des types entiers plus larges (`uint64_t`) si nécessaire.
- **Contrôles avant calcul :** Vérifier les bornes avant les opérations arithmétiques.
- **Fonctions dédiées :** Utiliser des fonctions comme `__builtin_add_overflow` (GCC) pour détecter les dépassements.

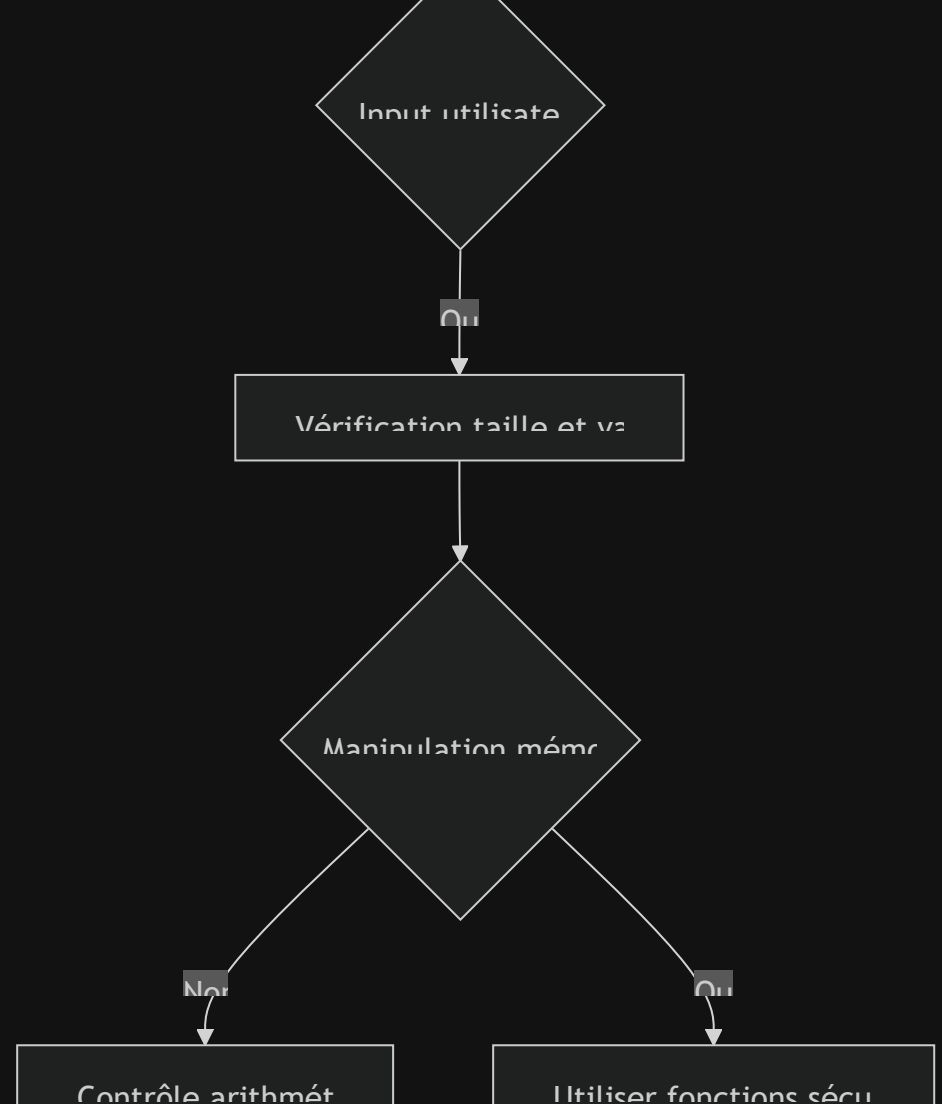
Protéger votre code C efficacement

Bonnes pratiques de sécurité en C :

Vulnérabilité	Techniques de prévention
Buffer Overflow	Limiter la taille des entrées, fonctions sûres (`strlcpy` , `snprintf`), vérifications systématiques
Integer Overflow	Vérifier les bornes avant calcul, types adaptés, fonctions de détection intégrées
Gestion mémoire	Initialisation des buffers, éviter les accès hors limites
Analyse statique/dynamique	Utiliser outils comme Coverity, Valgrind, ASan

Exemple corrigé du Buffer Overflow :

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

void safe_function(char *input) {
    char buffer[10];
    // Copie sécurisée avec limitation de la taille
    strncpy(buffer, input, sizeof(buffer) - 1);
    buffer[sizeof(buffer) - 1] = '\0';  // Toujours terminer par un \0
    printf("Données reçues : %s\n", buffer);
}

int main() {
```


Input utilisate
Ou
Vérification taille et va
Manipulation mémo
Non
Ou
Contrôle arithmét
Utiliser fonctions sécu

En Bref : Sécurité et Références Clés

Synthèse et ressources pour approfondir

En intégrant des contrôles rigoureux et en adoptant les fonctions sécurisées, le code C peut être protégé efficacement contre des vulnérabilités majeures telles que les buffer overflows et integer overflows, évitant ainsi des erreurs critiques et renforçant la sécurité globale des applications.

Sources utilisées :

- OWASP - Buffer Overflow
- CWE-121: Stack-based Buffer Overflow
- Integer Overflow and How to Avoid It in C
- GCC Built-in Overflow Checking
- [Secure Coding in C and C++, Robert C. Seacord]
- Valgrind Official Documentation
- AddressSanitizer (ASan)