Guide de Programmation : De C à Go en 3 Mois

Introduction

Ce guide est conçu pour vous accompagner dans un parcours d'apprentissage intensif de trois mois, vous menant des bases de la programmation système avec C et C++ à la maîtrise de langages modernes et performants comme Rust et Go. En suivant ce programme à raison de deux heures par jour, vous acquerrirez des compétences solides et recherchées dans le développement logiciel.

Chaque semaine est structurée autour de concepts clés, de ressources d'apprentissage sélectionnées pour leur qualité et de projets pratiques pour consolider vos connaissances. Ce guide n'est pas seulement un plan d'étude, c'est une feuille de route pour devenir un développeur polyvalent et compétent.

Semaine 1 : Les bases de C

Concepts à apprendre : * **Variables, types de données, et modificateurs** : Comprendre comment déclarer et utiliser différents types de données (int, float, char, double, etc.) et leurs modificateurs (short, long, signed, unsigned). * **Opérateurs (arithmétiques, relationnels, logiques)** : Maîtriser les opérations de base, les comparaisons et les conditions logiques. * **Structures de contrôle : boucles (for, while, do-while) et conditions (if, else, switch)** : Apprendre à contrôler le flux d'exécution du programme. * **Fonctions : déclaration, définition, et appel** : Organiser le code en blocs réutilisables. * **Pointeurs : déclaration, initialisation, arithmétique des pointeurs** : Comprendre l'accès direct à la mémoire, un concept fondamental en C. * **Tableaux : déclaration, initialisation, et relation avec les pointeurs** : Gérer des collections de données du même type. * **Gestion de la mémoire : la pile (stack) et le tas (heap)** : Distinguer les deux principales zones de mémoire et leur utilisation.

Exemples de code :

```c
#include <stdio.h>

int main() {
    // Variables et types de données
    int age = 30;
    float temperature = 25.5;
    char grade = 'A';
    printf("Age: %d, Temperature: %.1f, Grade: %c\n", age, temperature, grade);

    // Opérateurs arithmétiques
    int a = 10, b = 5;
    printf("Sum: %d, Difference: %d\n", a + b, a - b);

    // Structures de contrôle (boucle for)
    printf("Numbers from 0 to 4:\n");
    for (int i = 0; i < 5; i++) {
        printf("%d ", i);
    }
    printf("\n");

    // Fonctions (exemple simple)
    int sum_result = add(10, 20);
    printf("Sum of 10 and 20: %d\n", sum_result);

    // Pointeurs
    int num = 100;
    int *ptr = &num; // ptr stocke l'adresse de num
    printf("Value of num: %d, Value through pointer: %d\n", num, *ptr);
    printf("Address of num: %p, Value of ptr: %p\n", &num, ptr);

    // Tableaux
    int numbers[5] = {10, 20, 30, 40, 50};
    printf("First element of array: %d\n", numbers[0]);
    printf("Second element using pointer arithmetic: %d\n", *(numbers + 1));

    return 0;
}

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

Ressources : * Tutoriel C de W3Schools * Tutoriel C interactif de learn-c.org * Tutoriel C de GeeksforGeeks

Projet : Calculatrice en ligne de commande (CLI) en C

Créez une calculatrice simple qui prend des arguments en ligne de commande pour effectuer des opérations arithmétiques de base (addition, soustraction, multiplication, division). Voici une base pour commencer :

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

int main(int argc, char *argv[]) {
    if (argc != 4) {
        printf("Usage: %s <nombre1> <opérateur> <nombre2>\n", argv[0]);
        return 1;
    }

    double num1 = atof(argv[1]);
    char operator = argv[2][0];
    double num2 = atof(argv[3]);
    double result;

    switch (operator) {
        case '+':
            result = num1 + num2;
            break;
        case '-':
            result = num1 - num2;
            break;
        case 'x': // Utiliser 'x' pour la multiplication pour éviter les problèmes de shell avec '*'
            result = num1 * num2;
            break;
        case '/':
            if (num2 == 0) {
                printf("Erreur: Division par zéro !\n");
                return 1;
            }
            result = num1 / num2;
            break;
        default:
            printf("Opérateur invalide. Utilisez +, -, x, ou /.\n");
            return 1;
    }

    printf("Résultat: %.2f\n", result);

    return 0;
}
```

Pour compiler et exécuter :

```
gcc calculator.c -o calculator
./calculator 10 + 5
./calculator 20 x 3
```

Semaine 2 : Approfondissement en C

Concepts à apprendre : * **Allocation de mémoire dynamique :** `malloc`, `calloc`, `realloc`, `free` : Gérer la mémoire sur le tas pour des structures de données de taille variable. * **Structures (`struct`) et unions (`union`)** : Définir des types de données

complexes et hétérogènes. * **Entrées/sorties de fichiers : `fopen`, `fclose`, `fread`, `fwrite`, `fprintf`, `fscanf`** : Interagir avec le système de fichiers pour lire et écrire des données.

Exemples de code :

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

// Exemple de structure
typedef struct {
    char name[50];
    int age;
} Person;

int main() {
    // Allocation dynamique de mémoire pour un entier
    int *ptr_int = (int *) malloc(sizeof(int));
    if (ptr_int == NULL) {
        printf("Erreur d'allocation mémoire.\n");
        return 1;
    }
    *ptr_int = 10;
    printf("Valeur allouée dynamiquement: %d\n", *ptr_int);
    free(ptr_int); // Libérer la mémoire

    // Allocation dynamique pour une structure
    Person *p = (Person *) malloc(sizeof(Person));
    if (p == NULL) {
        printf("Erreur d'allocation mémoire pour Person.\n");
        return 1;
    }
    strcpy(p->name, "Alice");
    p->age = 25;
    printf("Personne: %s, Age: %d\n", p->name, p->age);
    free(p);

    // Entrée/sortie de fichier
    FILE *file = fopen("example.txt", "w"); // Ouvrir en mode écriture
    if (file == NULL) {
        printf("Erreur lors de l'ouverture du fichier.\n");
        return 1;
    }
    fprintf(file, "Ceci est un exemple de texte.\n");
    fprintf(file, "Écrit dans un fichier.\n");
    fclose(file);

    file = fopen("example.txt", "r"); // Ouvrir en mode lecture
    if (file == NULL) {
        printf("Erreur lors de l'ouverture du fichier.\n");
        return 1;
    }
    char buffer[100];
    while (fgets(buffer, sizeof(buffer), file) != NULL) {
        printf("Lu du fichier: %s", buffer);
    }
    fclose(file);

    return 0;
}
```

Projet : Compteur de mots (clone de `wc`)

Développez un programme qui compte le nombre de lignes, de mots et de caractères dans un fichier, de manière similaire à la commande `wc` d'Unix. Voici un exemple de base :

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

int main(int argc, char *argv[]) {
    if (argc != 2) {
        fprintf(stderr, "Usage: %s <filename>\n", argv[0]);
        return 1;
    }

    FILE *file = fopen(argv[1], "r");
    if (file == NULL) {
        perror("Erreur lors de l'ouverture du fichier");
        return 1;
    }

    int lines = 0;
    int words = 0;
    int chars = 0;
    int in_word = 0; // Flag pour savoir si on est à l'intérieur d'un mot
    int c;

    while ((c = fgetc(file)) != EOF) {
        chars++;
        if (c == '\n') {
            lines++;
        }
        if (isspace(c)) {
            in_word = 0;
        } else if (in_word == 0) {
            words++;
            in_word = 1;
        }
    }

    // Si le fichier ne se termine pas par un retour à la ligne, la dernière ligne n'est pas comptée
    // On ajoute 1 si le fichier n'est pas vide et ne se termine pas par un retour à la ligne
    if (chars > 0 && c != '\n') {
        lines++;
    }

    printf("\t%d\t%d\t%d %s\n", lines, words, chars, argv[1]);

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

Pour compiler et exécuter :

```
gcc wc_clone.c -o wc_clone
echo "Hello world\nThis is a test." > test.txt
./wc_clone test.txt
```

Bonus : Implémentation d'un vecteur dynamique simple

Pour aller plus loin, essayez d'implémenter une structure de données de type vecteur (tableau dynamique) en C, avec des fonctions pour ajouter des éléments et libérer la mémoire.

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

typedef struct {
    int *data;
    size_t size;     // Nombre d'éléments actuellement stockés
    size_t capacity; // Capacité totale du vecteur
} IntVector;

// Initialise un nouveau vecteur
void init_vector(IntVector *vec, size_t initial_capacity) {
    vec->data = (int *) malloc(initial_capacity * sizeof(int));
    if (vec->data == NULL) {
        perror("Erreur d'allocation mémoire pour le vecteur");
        exit(EXIT_FAILURE);
    }
    vec->size = 0;
    vec->capacity = initial_capacity;
}

// Ajoute un élément à la fin du vecteur
void add_element(IntVector *vec, int element) {
    if (vec->size == vec->capacity) {
        // Doubler la capacité si nécessaire
        vec->capacity *= 2;
        vec->data = (int *) realloc(vec->data, vec->capacity * sizeof(int));
        if (vec->data == NULL) {
            perror("Erreur de réallocation mémoire pour le vecteur");
            exit(EXIT_FAILURE);
        }
    }
    vec->data[vec->size++] = element;
}

// Libère la mémoire allouée par le vecteur
void free_vector(IntVector *vec) {
    free(vec->data);
    vec->data = NULL;
    vec->size = 0;
    vec->capacity = 0;
}

int main() {
    IntVector my_vector;
    init_vector(&my_vector, 2); // Commencer avec une capacité de 2

    add_element(&my_vector, 10);
    add_element(&my_vector, 20);
    add_element(&my_vector, 30); // Cela devrait déclencher une réallocation

    printf("Éléments du vecteur: ");
    for (size_t i = 0; i < my_vector.size; i++) {
        printf("%d ", my_vector.data[i]);
    }
    printf("\n");
    printf("Taille actuelle: %zu, Capacité: %zu\n", my_vector.size,
my_vector.capacity);

    free_vector(&my_vector);
```

```
    return 0;
}
```

Semaine 3 : Les bases de C++

Concepts à apprendre : * **Classes et objets : attributs, méthodes, constructeurs, destructeurs** : La pierre angulaire de la programmation orientée objet en C++. * **Références et pointeurs : différences et cas d'utilisation** : Comprendre quand utiliser l'un ou l'autre pour manipuler la mémoire. * **Conteneurs de la STL (Standard Template Library) : `std::vector`, `std::map`** : Utiliser des structures de données génériques et performantes. * **RAII (Resource Acquisition Is Initialization) : un principe fondamental en C++ pour la gestion des ressources** : Assurer la libération automatique des ressources (mémoire, fichiers, etc.) via la durée de vie des objets.

Exemples de code :

```cpp
#include <iostream>
#include <vector>
#include <map>
#include <string>

// Exemple de classe simple
class Dog {
public:
    std::string name;
    int age;

    // Constructeur
    Dog(std::string name, int age) : name(name), age(age) {
        std::cout << name << " the dog is born!\n";
    }

    // Méthode
    void bark() {
        std::cout << name << " says Woof!\n";
    }

    // Destructeur
    ~Dog() {
        std::cout << name << " the dog passed away.\n";
    }
};

int main() {
    // Création d'objets
    Dog myDog("Buddy", 3);
    myDog.bark();

    // Références vs Pointeurs
    int a = 10;
    int &ref_a = a; // Référence à a
    int *ptr_a = &a; // Pointeur vers a

    std::cout << "a: " << a << ", ref_a: " << ref_a << ", *ptr_a: " << *ptr_a
<< "\n";
    ref_a = 20; // Modifier via référence
    std::cout << "a after ref_a change: " << a << "\n";
    *ptr_a = 30; // Modifier via pointeur
    std::cout << "a after ptr_a change: " << a << "\n";

    // std::vector (conteneur dynamique)
    std::vector<int> numbers = {1, 2, 3, 4, 5};
    numbers.push_back(6);
    std::cout << "Elements in vector: ";
    for (int num : numbers) {
        std::cout << num << " ";
    }
    std::cout << "\n";

    // std::map (tableau associatif)
    std::map<std::string, std::string> capitals;
    capitals["France"] = "Paris";
    capitals["Germany"] = "Berlin";
    std::cout << "Capital of France: " << capitals["France"] << "\n";

    // RAII (std::vector gère automatiquement sa mémoire)
    // L'objet myDog est détruit automatiquement à la fin de main()
```

```
    return 0;
}
```

Ressources : * C++ Crash Course * Tutoriel C++ de W3Schools

Projet : Système de compte bancaire avec des classes

Modélisez un système de compte bancaire simple en utilisant des classes pour représenter les clients et les comptes. Implémentez des fonctionnalités pour déposer, retirer de l'argent et consulter le solde. Voici une structure de base :

```cpp
#include <iostream>
#include <string>
#include <vector>

class BankAccount {
private:
    std::string accountNumber;
    double balance;

public:
    BankAccount(std::string accNum, double initialBalance) :
        accountNumber(accNum), balance(initialBalance) {}

    void deposit(double amount) {
        if (amount > 0) {
            balance += amount;
            std::cout << "Dépot de " << amount << " effectué. Nouveau solde : "
<< balance << "\n";
        } else {
            std::cout << "Le montant du dépot doit être positif.\n";
        }
    }

    void withdraw(double amount) {
        if (amount > 0 && balance >= amount) {
            balance -= amount;
            std::cout << "Retrait de " << amount << " effectué. Nouveau solde :
" << balance << "\n";
        } else if (amount <= 0) {
            std::cout << "Le montant du retrait doit être positif.\n";
        } else {
            std::cout << "Solde insuffisant pour le retrait de " << amount <<
". Solde actuel : " << balance << "\n";
        }
    }

    double getBalance() const {
        return balance;
    }

    std::string getAccountNumber() const {
        return accountNumber;
    }
};

int main() {
    BankAccount myAccount("FR123456789", 1000.0);
    std::cout << "Compte " << myAccount.getAccountNumber() << " créé avec un
solde de " << myAccount.getBalance() << "\n";

    myAccount.deposit(200.0);
    myAccount.withdraw(150.0);
    myAccount.withdraw(1500.0);

    std::cout << "Solde final du compte " << myAccount.getAccountNumber() << "
: " << myAccount.getBalance() << "\n";

    return 0;
}
```

Semaine 4 : Synthèse et Transition

Concepts à apprendre : * **Comparaison des modèles de mémoire de C et C++** : Comprendre les différences fondamentales dans la gestion de la mémoire, notamment l'allocation manuelle en C vs. l'utilisation de RAII et des conteneurs en C++. * **Interoperabilité entre C et C++** : Apprendre comment les fonctions et les structures C peuvent être utilisées dans du code C++ et vice-versa (avec `extern "C"`).

Exemples de code :

```cpp
// C++ code calling a C function
#include <iostream>

// Déclaration d'une fonction C (doit être dans un fichier .c)
extern "C" {
    void greet_from_c();
}

int main() {
    std::cout << "Hello from C++!\n";
    greet_from_c(); // Appel de la fonction C
    return 0;
}
```

```c
// C code (greet.c)
#include <stdio.h>

void greet_from_c() {
    printf("Hello from C!\n");
}
```

Pour compiler et exécuter (sur Linux/macOS) :

```
gcc -c greet.c -o greet.o
g++ main.cpp greet.o -o mixed_lang_app
./mixed_lang_app
```

Projet : Liste de tâches en C et en C++

Implémentez une application de liste de tâches simple en C (en utilisant des tableaux et `realloc`) et en C++ (en utilisant `std::vector`). Comparez les deux implémentations en termes de complexité du code, de sécurité et de facilité de maintenance.

Exemple de liste de tâches en C (avec `realloc`) :

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

#define INITIAL_CAPACITY 2

typedef struct {
    char *description;
    int completed;
} Task;

typedef struct {
    Task *tasks;
    int count;
    int capacity;
} TodoListC;

void init_todo_list_c(TodoListC *list) {
    list->tasks = (Task *) malloc(sizeof(Task) * INITIAL_CAPACITY);
    if (list->tasks == NULL) {
        perror("Failed to allocate memory for tasks");
        exit(EXIT_FAILURE);
    }
    list->count = 0;
    list->capacity = INITIAL_CAPACITY;
}

void add_task_c(TodoListC *list, const char *description) {
    if (list->count == list->capacity) {
        list->capacity *= 2;
        list->tasks = (Task *) realloc(list->tasks, sizeof(Task) * list-
>capacity);
        if (list->tasks == NULL) {
            perror("Failed to reallocate memory for tasks");
            exit(EXIT_FAILURE);
        }
    }
    list->tasks[list->count].description = strdup(description);
    if (list->tasks[list->count].description == NULL) {
        perror("Failed to duplicate task description");
        exit(EXIT_FAILURE);
    }
    list->tasks[list->count].completed = 0;
    list->count++;
    printf("Task added: %s\n", description);
}

void list_tasks_c(const TodoListC *list) {
    printf("\n--- Your Tasks (C) ---\n");
    if (list->count == 0) {
        printf("No tasks yet.\n");
        return;
    }
    for (int i = 0; i < list->count; i++) {
        printf("%d. [%c] %s\n", i + 1, list->tasks[i].completed ? 'x' : ' ',
list->tasks[i].description);
    }
    printf("----------------------\n");
}

void free_todo_list_c(TodoListC *list) {
```

```c
    for (int i = 0; i < list->count; i++) {
        free(list->tasks[i].description);
    }
    free(list->tasks);
    list->tasks = NULL;
    list->count = 0;
    list->capacity = 0;
}

int main_c() {
    TodoListC my_todo_list;
    init_todo_list_c(&my_todo_list);

    add_task_c(&my_todo_list, "Learn C pointers");
    add_task_c(&my_todo_list, "Implement wc clone");
    add_task_c(&my_todo_list, "Understand malloc/free");

    list_tasks_c(&my_todo_list);

    free_todo_list_c(&my_todo_list);
    return 0;
}
```

Exemple de liste de tâches en C++ (avec `std::vector`) :

```cpp
#include <iostream>
#include <vector>
#include <string>

struct TaskCpp {
    std::string description;
    bool completed;

    TaskCpp(const std::string& desc) : description(desc), completed(false) {}
};

class TodoListCpp {
private:
    std::vector<TaskCpp> tasks;

public:
    void addTask(const std::string& description) {
        tasks.emplace_back(description);
        std::cout << "Task added: " << description << "\n";
    }

    void listTasks() const {
        std::cout << "\n--- Your Tasks (C++) ---\n";
        if (tasks.empty()) {
            std::cout << "No tasks yet.\n";
            return;
        }
        for (size_t i = 0; i < tasks.size(); ++i) {
            std::cout << i + 1 << ". [" << (tasks[i].completed ? 'x' : ' ') <<
"] " << tasks[i].description << "\n";
        }
        std::cout << "----------------------\n";
    }

    // std::vector gère automatiquement la mémoire, pas besoin de free
explicite
};

int main_cpp() {
    TodoListCpp my_todo_list;

    my_todo_list.addTask("Learn C++ classes");
    my_todo_list.addTask("Use STL containers");
    my_todo_list.addTask("Understand RAII");

    my_todo_list.listTasks();

    return 0;
}
```

Comparaison :

Caractéristique	Implémentation C (avec `realloc`)	Implémentation C++ (avec `std::vector`)
Gestion mémoire	Manuelle (`malloc`, `realloc`, `free`)	Automatique (RAII via `std::vector`)
Complexité code	Plus élevée, nécessite une gestion explicite des pointeurs et de la mémoire	Plus faible, abstractions de haut niveau simplifient le code
Sécurité	Risque d'erreurs mémoire (fuites, accès invalides) plus élevé	Moins de risques d'erreurs mémoire grâce à RAII et aux conteneurs
Facilité maintenance	Plus difficile à maintenir et à faire évoluer	Plus facile à maintenir et à faire évoluer
Performance	Potentiellement plus fine si optimisée manuellement	Très performante, optimisée par les implémentations de la STL

Cette comparaison met en évidence les avantages de C++ pour la productivité et la sécurité du code, tout en conservant une grande partie de la performance de C.

Mois 2 : Plongée au cœur de Rust

Semaine 5 : Les bases de Rust

Concepts à apprendre : * **Variables, mutabilité et "shadowing"** : Comprendre comment déclarer des variables, les rendre mutables et le concept de "shadowing" en Rust. * **Propriété (ownership), emprunt (borrowing) et durées de vie (lifetimes)** : Les concepts fondamentaux de Rust pour la gestion de la mémoire sans ramasse-miettes ni pointeurs nuls. * **Structures (`struct`) et énumérations (`enum`)** : Définir des types de données personnalisés. * **Les bases de Cargo, le gestionnaire de paquets et de build de Rust** : Créer, compiler et gérer des projets Rust.

Exemples de code :

```rust
fn main() {
    // Variables et mutabilité
    let x = 5; // Immuable par défaut
    println!("The value of x is: {}", x);

    let mut y = 5; // Mutable
    println!("The value of y is: {}", y);
    y = 6;
    println!("The new value of y is: {}", y);

    // Shadowing
    let z = 5;
    let z = z + 1; // z est "shadowed" par une nouvelle variable
    let z = z * 2;
    println!("The value of z is: {}", z);

    // Ownership
    let s1 = String::from("hello");
    let s2 = s1; // s1 est déplacé vers s2, s1 n'est plus valide
    // println!("s1: {}", s1); // Erreur: valeur empruntée après déplacement
    println!("s2: {}", s2);

    // Borrowing (références)
    let s3 = String::from("world");
    let len = calculate_length(&s3); // &s3 est une référence (emprunt)
    println!("The length of '{}' is {}.", s3, len);

    // Structures
    struct User {
        username: String,
        email: String,
        sign_in_count: u64,
        active: bool,
    }

    let user1 = User {
        email: String::from("someone@example.com"),
        username: String::from("someusername123"),
        active: true,
        sign_in_count: 1,
    };
    println!("User email: {}", user1.email);

    // Énumérations
    enum IpAddrKind {
        V4,
        V6,
    }

    let four = IpAddrKind::V4;
    let six = IpAddrKind::V6;

    // Cargo: Créer un nouveau projet avec `cargo new <project_name>`
    // Compiler avec `cargo build`
    // Exécuter avec `cargo run`
}

fn calculate_length(s: &String) -> usize { // s est une référence à un String
    s.len()
}
```

Ressources : * [The Rust Programming Language (The Rust Book), Chapitres 1–6]

Projet : Application de liste de tâches en ligne de commande (CLI) en Rust

Développez une application de liste de tâches en ligne de commande qui permet d'ajouter, de supprimer et de lister des tâches. Utilisez `struct` pour représenter une tâche et `Vec` pour stocker les tâches. Voici une base pour commencer :

Créez un nouveau projet Cargo :

```
 cargo new rust_todo_cli
cd rust_todo_cli
```

Modifiez `src/main.rs` :

```rust
use std::io::{self, Write};

struct Task {
    description: String,
    completed: bool,
}

impl Task {
    fn new(description: String) -> Task {
        Task { description, completed: false }
    }

    fn display(&self, index: usize) {
        let status = if self.completed { "[x]" } else { "[ ]" };
        println!("{}. {} {}", index, status, self.description);
    }
}

fn main() {
    let mut tasks: Vec<Task> = Vec::new();

    loop {
        println!("\n--- Rust Todo CLI ---");
        println!("1. Add task");
        println!("2. List tasks");
        println!("3. Mark task as completed (not implemented yet)");
        println!("4. Remove task (not implemented yet)");
        println!("5. Exit");
        print!("Enter your choice: ");
        io::stdout().flush().unwrap();

        let mut choice = String::new();
        io::stdin().read_line(&mut choice).expect("Failed to read line");
        let choice: u32 = match choice.trim().parse() {
            Ok(num) => num,
            Err(_) => {
                println!("Invalid input. Please enter a number.");
                continue;
            }
        };

        match choice {
            1 => {
                print!("Enter task description: ");
                io::stdout().flush().unwrap();
                let mut description = String::new();
                io::stdin().read_line(&mut description).expect("Failed to read
line");
                tasks.push(Task::new(description.trim().to_string()));
                println!("Task added!");
            }
            2 => {
                if tasks.is_empty() {
                    println!("No tasks to display.");
                } else {
                    println!("\n--- Your Tasks ---");
                    for (i, task) in tasks.iter().enumerate() {
                        task.display(i + 1);
                    }
                    println!("--------------------");
                }
```

```
            }
            5 => {
                println!("Exiting...");
                break;
            }
            _ => println!("Invalid choice. Please try again."),
        }
    }
}
```

Pour exécuter :

```
cargo run
```

Semaine 6 : Gestion des erreurs et Collections

Concepts à apprendre : * **Les types `Result` et `Option` pour la gestion des erreurs** : Apprendre à gérer les erreurs récupérables et l'absence de valeur de manière idiomatique en Rust, sans exceptions. * **Les collections `Vec` (vecteur) et `HashMap` (table de hachage)** : Utiliser les structures de données dynamiques fournies par la bibliothèque standard de Rust. * **Le filtrage par motif (pattern matching) avec `match`** : Une fonctionnalité puissante pour contrôler le flux du programme en fonction de la structure des données.

Exemples de code :

```rust
use std::fs::File;
use std::io::Read;
use std::collections::HashMap;

fn main() {
    // Exemple avec Option (pour les valeurs potentiellement absentes)
    let some_number = Some(5);
    let no_number: Option<i32> = None;

    match some_number {
        Some(n) => println!("There is a number: {}", n),
        None => println!("There is no number."),
    }

    // Exemple avec Result (pour les erreurs récupérables)
    let file_result = File::open("hello.txt");

    let file = match file_result {
        Ok(file) => file,
        Err(error) => panic!("Problem opening the file: {:?}", error),
    };
    println!("File opened successfully!");

    // Vec (vecteur dynamique)
    let mut numbers = vec![1, 2, 3];
    numbers.push(4);
    println!("Vector: {:?}", numbers);

    // HashMap (table de hachage)
    let mut scores = HashMap::new();
    scores.insert(String::from("Blue"), 10);
    scores.insert(String::from("Yellow"), 50);

    let team_name = String::from("Blue");
    let score = scores.get(&team_name).copied().unwrap_or(0);
    println!("Score for {}: {}", team_name, score);

    // Pattern matching avec match
    let config_max = Some(3u8);
    match config_max {
        Some(max) => println!("The maximum is configured to be {}", max),
        _ => (),
    }
}
```

Projet : Gestionnaire de notes

Créez un programme qui permet de sauvegarder et de charger des notes textuelles dans un fichier. Utilisez `Result` pour la gestion des erreurs de fichier et `Vec<String>` pour stocker les notes.

Créez un nouveau projet Cargo :

```
cargo new note_manager
cd note_manager
```

Modifiez `src/main.rs` :

```rust
 use std::io::{self, Write, BufReader, BufRead};
use std::fs::{File, OpenOptions};

const NOTES_FILE: &str = "notes.txt";

fn main() {
    loop {
        println!("\n--- Note Manager ---");
        println!("1. Add a note");
        println!("2. List all notes");
        println!("3. Exit");
        print!("Enter your choice: ");
        io::stdout().flush().unwrap();

        let mut choice = String::new();
        io::stdin().read_line(&mut choice).expect("Failed to read line");
        let choice = choice.trim();

        match choice {
            "1" => add_note().expect("Failed to add note"),
            "2" => list_notes().expect("Failed to list notes"),
            "3" => {
                println!("Exiting...");
                break;
            }
            _ => println!("Invalid choice. Please try again."),
        }
    }
}

fn add_note() -> io::Result<()> {
    print!("Enter your note: ");
    io::stdout().flush().unwrap();
    let mut note = String::new();
    io::stdin().read_line(&mut note).expect("Failed to read note");

    let mut file = OpenOptions::new()
        .create(true)
        .append(true)
        .open(NOTES_FILE)?;

    writeln!(file, "{}", note.trim())?;
    println!("Note added successfully!");
    Ok(())
}

fn list_notes() -> io::Result<()> {
    let file = File::open(NOTES_FILE);
    let file = match file {
        Ok(file) => file,
        Err(ref error) if error.kind() == io::ErrorKind::NotFound => {
            println!("No notes found. Add some first!");
            return Ok(());
        }
        Err(error) => return Err(error),
    };

    println!("\n--- Your Notes ---");
    let reader = BufReader::new(file);
    for (i, line) in reader.lines().enumerate() {
        println!("{}. {}", i + 1, line?);
```

```
    }
    println!("------------------");
    Ok(())
}
```

Pour exécuter :

```
cargo run
```

Semaine 7 : Traits et Génériques

Concepts à apprendre : * Les traits (`Display`, `Debug`, et la création de traits personnalisés) : Définir des comportements partagés entre différents types, similaire aux interfaces en Java ou aux classes abstraites en C++. * **La généricité dans les fonctions et les structures** : Écrire du code flexible qui fonctionne avec plusieurs types sans duplication. * **Modules et crates pour organiser le code** : Structurer des projets Rust de grande taille.

Exemples de code :

```rust
use std::fmt;

// Trait personnalisé
trait Greeter {
    fn greet(&self) -> String;
}

// Implémentation du trait pour un type struct
struct Person {
    name: String,
    age: u8,
}

impl Greeter for Person {
    fn greet(&self) -> String {
        format!("Hello, my name is {}.", self.name)
    }
}

// Implémentation du trait pour un type enum
enum Animal {
    Dog(String),
    Cat(String),
}

impl Greeter for Animal {
    fn greet(&self) -> String {
        match self {
            Animal::Dog(name) => format!("Woof! I am {}.", name),
            Animal::Cat(name) => format!("Meow! I am {}.", name),
        }
    }
}

// Fonction générique qui accepte tout type implémentant le trait Greeter
fn say_hello<T: Greeter>(item: T) {
    println!("{}", item.greet());
}

// Structure générique
struct Point<T> {
    x: T,
    y: T,
}

// Implémentation du trait Display pour Person
impl fmt::Display for Person {
    fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {
        write!(f, "Person: {} ({} years old)", self.name, self.age)
    }
}

// Implémentation du trait Debug pour Person (souvent dérivé automatiquement)
#[derive(Debug)]
struct DebugPerson {
    name: String,
    age: u8,
}

fn main() {
    let person = Person { name: String::from("Alice"), age: 30 };
```

```
    say_hello(person);

    let dog = Animal::Dog(String::from("Buddy"));
    say_hello(dog);

    let cat = Animal::Cat(String::from("Whiskers"));
    say_hello(cat);

    let integer_point = Point { x: 5, y: 10 };
    let float_point = Point { x: 1.0, y: 4.5 };

    println!("Integer Point: ({}, {})", integer_point.x, integer_point.y);
    println!("Float Point: ({}, {})", float_point.x, float_point.y);

    let display_person = Person { name: String::from("Bob"), age: 22 };
    println!("Display Person: {}", display_person);

    let debug_person = DebugPerson { name: String::from("Charlie"), age: 40 };
    println!("Debug Person: {:?}", debug_person);

    // Modules et crates (exemple conceptuel)
    // Pour organiser le code, on utilise des modules (mod.rs, ou dossiers)
    // et des crates (lib.rs, main.rs)
    // Par exemple, dans un fichier `my_module.rs`:
    // pub fn public_function() { /* ... */ }
    // fn private_function() { /* ... */ }
    // Dans main.rs: `mod my_module; use my_module::public_function;`
}
```

Projet : Parseur JSON basique

Implémentez un parseur JSON simple capable de traiter des paires clé:valeur de base. Vous pouvez utiliser des énumérations pour représenter les différents types de valeurs JSON (chaîne, nombre, booléen, null) et des traits pour sérialiser/désérialiser des objets. Pour ce projet, nous allons nous concentrer sur la lecture d'un JSON simple et l'extraction de valeurs.

Créez un nouveau projet Cargo :

```
 cargo new json_parser
cd json_parser
```

Modifiez `src/main.rs` :

```rust
use std::collections::HashMap;

// Représentation simple d'une valeur JSON
enum JsonValue {
    String(String),
    Number(f64),
    Boolean(bool),
    Null,
    Object(HashMap<String, JsonValue>),
    Array(Vec<JsonValue>),
}

// Un trait pour la désérialisation (simplifié pour l'exemple)
trait FromJson {
    fn from_json(value: &JsonValue) -> Option<Self> where Self: Sized;
}

impl FromJson for String {
    fn from_json(value: &JsonValue) -> Option<Self> {
        if let JsonValue::String(s) = value {
            Some(s.clone())
        } else {
            None
        }
    }
}

impl FromJson for f64 {
    fn from_json(value: &JsonValue) -> Option<Self> {
        if let JsonValue::Number(n) = value {
            Some(*n)
        } else {
            None
        }
    }
}

// Fonction très basique pour "parser" une chaîne JSON simple (non robuste)
// Ceci est une simplification extrême et ne gère pas les JSON complexes, les échappements, etc.
// L'objectif est de montrer l'utilisation des concepts de Rust.
fn parse_simple_json(json_str: &str) -> Option<JsonValue> {
    let trimmed = json_str.trim();
    if trimmed.starts_with("{") && trimmed.ends_with("}") {
        let content = &trimmed[1..trimmed.len() - 1];
        let parts: Vec<&str> = content.split(":").collect();
        if parts.len() == 2 {
            let key = parts[0].trim().trim_matches("\"").to_string();
            let value_str = parts[1].trim();

            if value_str.starts_with("\"") && value_str.ends_with("\"") {
                let value = value_str.trim_matches("\"").to_string();
                let mut map = HashMap::new();
                map.insert(key, JsonValue::String(value));
                return Some(JsonValue::Object(map));
            } else if let Ok(num) = value_str.parse::<f64>() {
                let mut map = HashMap::new();
                map.insert(key, JsonValue::Number(num));
                return Some(JsonValue::Object(map));
            } else if value_str == "true" || value_str == "false" {
                let mut map = HashMap::new();
```

```
                map.insert(key, JsonValue::Boolean(value_str == "true"));
                return Some(JsonValue::Object(map));
            } else if value_str == "null" {
                let mut map = HashMap::new();
                map.insert(key, JsonValue::Null);
                return Some(JsonValue::Object(map));
            }
        }
    }
    None
}

fn main() {
    let json_data = r#"{"name": "Rust"}"#;
    let json_data_num = r#"{"version": 1.70}"#;

    if let Some(JsonValue::Object(map)) = parse_simple_json(json_data) {
        if let Some(name_value) = map.get("name") {
            if let Some(name) = String::from_json(name_value) {
                println!("Parsed name: {}", name);
            }
        }
    }

    if let Some(JsonValue::Object(map)) = parse_simple_json(json_data_num) {
        if let Some(version_value) = map.get("version") {
            if let Some(version) = f64::from_json(version_value) {
                println!("Parsed version: {}", version);
            }
        }
    }

    // Pour un parseur JSON robuste, utilisez la crate `serde_json`:
    // Ajoutez `serde = { version = "1.0", features = ["derive"] }` et
`serde_json = "1.0"` à Cargo.toml
    // Exemple avec serde_json:
    // use serde_json::Value;
    // let json_str = r#"{"language": "Rust", "year": 2015}"#;
    // let parsed: Value = serde_json::from_str(json_str).unwrap();
    // println!("Language: {}", parsed["language"]);
}
```

Pour exécuter :

```
cargo run
```

Semaine 8 : Concurrence et Asynchronisme

Concepts à apprendre : * **Les threads en Rust avec `std::thread`** : Apprendre à créer et gérer des threads pour l'exécution parallèle de tâches. * **Les bases de la programmation asynchrone avec `tokio`** : Comprendre les concepts d'async/await et comment utiliser un runtime asynchrone pour des opérations non bloquantes.

Exemples de code :

```rust
use std::thread;
use std::time::Duration;
use tokio::time::sleep;

#[tokio::main]
async fn main() {
    // Exemple avec std::thread (concurrence basée sur les threads OS)
    let handle = thread::spawn(|| {
        for i in 1..10 {
            println!("hi number {} from the spawned thread!", i);
            thread::sleep(Duration::from_millis(1));
        }
    });

    for i in 1..5 {
        println!("hi number {} from the main thread!", i);
        thread::sleep(Duration::from_millis(1));
    }

    handle.join().unwrap(); // Attendre que le thread spawned se termine

    println!("\n--- Async with Tokio ---");

    // Exemple avec Tokio (concurrence asynchrone)
    // Nécessite `tokio = { version = "1", features = ["full"] }` dans
Cargo.toml
    async_main().await;
}

async fn async_main() {
    let task1 = tokio::spawn(async {
        for i in 1..5 {
            println!("async task 1: {}", i);
            sleep(Duration::from_millis(10)).await;
        }
    });

    let task2 = tokio::spawn(async {
        for i in 1..5 {
            println!("async task 2: {}", i);
            sleep(Duration::from_millis(5)).await;
        }
    });

    // Attendre que les tâches asynchrones se terminent
    task1.await.unwrap();
    task2.await.unwrap();

    println!("All async tasks completed.");
}
```

Pour que l'exemple Tokio fonctionne, ajoutez ceci à votre `Cargo.toml` :

```toml
[dependencies]
tokio = { version = "1", features = ["full"] }
```

Ressources : * The Rust Async Book

Projet : Téléchargeur de fichiers concurrent

Écrivez un programme qui télécharge plusieurs fichiers de manière concurrente en utilisant les threads ou `tokio`. Pour cet exemple, nous utiliserons `tokio` et la crate `reqwest` pour les requêtes HTTP.

Créez un nouveau projet Cargo :

```
 cargo new concurrent_downloader
cd concurrent_downloader
```

Modifiez `Cargo.toml` pour ajouter les dépendances :

```
 [dependencies]
tokio = { version = "1", features = ["full"] }
reqwest = { version = "0.11", features = ["json", "stream"] }
futures = "0.3"
```

Modifiez `src/main.rs` :

```rust
use reqwest::Client;
use tokio::fs::File;
use tokio::io::AsyncWriteExt;
use futures::stream::StreamExt;

#[tokio::main]
async fn main() -> Result<(), Box<dyn std::error::Error>> {
    let urls = vec![
        "https://www.rust-lang.org/logos/rust-logo-512x512.png",
        "https://www.rust-lang.org/logos/rust-logo-128x128.png",
        "https://www.rust-lang.org/logos/rust-logo-64x64.png",
    ];

    let client = Client::new();
    let mut handles = vec![];

    for url in urls {
        let client = client.clone();
        let filename =
url.split('/').last().unwrap_or("downloaded_file").to_string();
        let handle = tokio::spawn(async move {
            println!("Starting download of {}", filename);
            match download_file(client, url, &filename).await {
                Ok(_) => println!("Finished downloading {}", filename),
                Err(e) => eprintln!("Error downloading {}: {}", filename, e),
            }
        });
        handles.push(handle);
    }

    // Attendre que toutes les tâches de téléchargement se terminent
    for handle in handles {
        handle.await?;
    }

    println!("All downloads attempted.");
    Ok(())
}

async fn download_file(client: Client, url: &str, filename: &str) -> Result<(),
reqwest::Error> {
    let response = client.get(url).send().await?.error_for_status()?;

    let mut file = File::create(filename).await?;
    let mut stream = response.bytes_stream();

    while let Some(chunk) = stream.next().await {
        file.write_all(&chunk?).await?;
    }

    Ok(())
}
```

Pour exécuter :

```
cargo run
```

Mois 3 : Rust Avancé et Go

Semaine 9 : Rust pour le Web

Concepts à apprendre : * **Les bases d'actix-web ou axum pour le développement web en Rust** : Comprendre les principes de construction d'applications web avec un framework asynchrone en Rust. * **Construction d'une API REST HTTP** : Apprendre à définir des routes, gérer les requêtes et les réponses HTTP, et interagir avec des données.

Exemples de code (avec `actix-web`) :

Pour cet exemple, nous utiliserons `actix-web`, un framework web performant pour Rust. Assurez-vous d'avoir `actix-web = "4"` et `tokio = { version = "1", features = ["full"] }` dans votre `Cargo.toml`.

Créez un nouveau projet Cargo :

```
cargo new rust_web_api
cd rust_web_api
```

Modifiez `Cargo.toml` :

```
[dependencies]
actix-web = "4"
tokio = { version = "1", features = ["full"] }
serde = { version = "1.0", features = ["derive"] } # Pour la sérialisation/désérialisation JSON
serde_json = "1.0"
```

Modifiez `src/main.rs` :

```rust
 use actix_web::{get, post, web, App, HttpResponse, HttpServer, Responder};
use serde::{Deserialize, Serialize};

// Définition d'une structure pour nos données de produit
#[derive(Serialize, Deserialize, Clone)]
struct Product {
    id: u32,
    name: String,
    price: f64,
}

// Simule une base de données en mémoire
// Dans une vraie application, vous utiliseriez une base de données persistante
struct AppState {
    products: std::sync::Mutex<Vec<Product>>,
}

#[get("/products")]
async fn get_products(data: web::Data<AppState>) -> impl Responder {
    let products = data.products.lock().unwrap();
    HttpResponse::Ok().json(&*products)
}

#[post("/products")]
async fn add_product(product: web::Json<Product>, data: web::Data<AppState>) ->
impl Responder {
    let mut products = data.products.lock().unwrap();
    products.push(product.into_inner());
    HttpResponse::Created().body("Product added")
}

#[get("/products/{id}")]
async fn get_product_by_id(path: web::Path<u32>, data: web::Data<AppState>) ->
impl Responder {
    let product_id = path.into_inner();
    let products = data.products.lock().unwrap();
    if let Some(product) = products.iter().find(|p| p.id == product_id) {
        HttpResponse::Ok().json(product)
    } else {
        HttpResponse::NotFound().body("Product not found")
    }
}

#[actix_web::main]
async fn main() -> std::io::Result<()> {
    let app_state = web::Data::new(AppState {
        products: std::sync::Mutex::new(vec![]),
    });

    HttpServer::new(move || {
        App::new()
            .app_data(app_state.clone())
            .service(get_products)
            .service(add_product)
            .service(get_product_by_id)
    })
    .bind(("127.0.0.1", 8080))?
    .run()
    .await
}
```

Pour exécuter :

```
cargo run
```

Testez l'API avec `curl` ou un outil comme Postman/Insomnia :

GET tous les produits :

```
curl http://127.0.0.1:8080/products
```

POST un nouveau produit :

```
curl -X POST -H "Content-Type: application/json" -d '{"id": 1, "name":
"Laptop", "price": 1200.0}' http://127.0.0.1:8080/products
```

GET un produit par ID :

```
curl http://127.0.0.1:8080/products/1
```

Projet : API CRUD pour les produits

L'exemple ci-dessus est une base pour une API CRUD. Pour le projet, étendez cette API pour inclure les opérations `Update` (PUT/PATCH) et `Delete` (DELETE) pour les produits. Vous devrez ajouter de nouvelles fonctions de service et les configurer dans l'application `actix-web`.

Semaine 10 : Rust et WebAssembly (WASM)

Concepts à apprendre : * **Compilation de Rust vers WebAssembly** : Comprendre comment transformer le code Rust en un format exécutable par les navigateurs web. * **Utilisation de Rust+WASM dans un navigateur web** : Apprendre à interagir entre JavaScript et le code Rust compilé en WASM.

Exemples de code :

Pour cet exemple, nous utiliserons `wasm-pack` pour compiler Rust en WASM et `webpack` pour le bundler JavaScript. Assurez-vous d'avoir Node.js et npm/yarn installés.

1. **Installer `wasm-pack`** : `bash curl https://rustwasm.github.io/wasm-pack/installer/init.sh -sSf | sh`
2. **Créer un nouveau projet Rust pour WASM** : `bash cargo new --lib rust_wasm_example cd rust_wasm_example`
3. **Modifiez `Cargo.toml`** : ```toml [lib] crate-type = ["cdylib"]

[dependencies] wasm-bindgen = "0.2" ```

1. **Modifiez `src/lib.rs`** : ```rust use wasm_bindgen::prelude::*;

[wasm_bindgen]

pub fn greet(name: &str) -> String { format!("Hello, {} from Rust!", name) }

[wasm_bindgen]

pub fn add(a: u32, b: u32) -> u32 { a + b } ```

1. **Compiler le projet Rust en WASM** : `bash wasm-pack build --target web` Ceci créera un dossier `pkg` avec les fichiers WASM et JavaScript nécessaires.
2. **Créer un projet JavaScript pour utiliser le WASM** : Dans le dossier parent de `rust_wasm_example` (par exemple, `my_web_app`): `bash mkdir my_web_app cd my_web_app npm init -y npm install webpack webpack-cli html-webpack-plugin`
3. **Créez `webpack.config.js` dans `my_web_app`** : ```javascript const path = require("path"); const HtmlWebpackPlugin = require("html-webpack-plugin");

module.exports = { entry: "./index.js", output: { path: path.resolve(__dirname, "dist"), filename: "index.js", }, plugins: [new HtmlWebpackPlugin({ template: "./index.html", }),], mode: "development", }; ```

1. **Créez `index.html` dans `my_web_app`** : ```html

Rust WASM Integration

```
```

1. **Créez `index.js` dans `my_web_app` :** `javascript import("../rust_wasm_example/pkg").then(wasm => { document.getElementById("greeting").innerText = wasm.greet("World"); document.getElementById("sum").innerText = `Sum of 5 and 10 is: ${wasm.add(5, 10)}`; }).catch(console.error);`

2. **Ajoutez un script de build à `package.json` dans `my_web_app` :** `json "scripts": { "build": "webpack" }`

3. **Construire et servir l'application web :** `bash npm run build` Ouvrez `dist/index.html` dans votre navigateur.

Projet : Filtre d'images dans le navigateur avec Rust+WASM

Créez une application web simple qui applique des filtres à des images en utilisant du code Rust compilé en WebAssembly. Ce projet est plus complexe et nécessiterait l'utilisation de crates Rust comme `image` pour la manipulation d'images et une interface utilisateur plus élaborée en JavaScript/HTML. L'idée est de charger une image dans le navigateur, d'envoyer ses données brutes au module WASM Rust, d'appliquer un filtre (par exemple, niveaux de gris, inversion des couleurs) en Rust, puis de renvoyer les données de l'image modifiée au JavaScript pour l'afficher.

Étapes clés : 1. **Rust (`rust_wasm_image_filter` crate):** * Utiliser la crate `image` pour charger, manipuler et sauvegarder des images. * Exposer des fonctions WASM pour prendre un tableau d'octets (représentant l'image), appliquer un filtre, et retourner un nouveau tableau d'octets. * Gérer les erreurs de manière appropriée. 2. **JavaScript/HTML (`web_app`):** * Un input de type `file` pour charger l'image. * Un élément `canvas` pour afficher l'image originale et l'image filtrée. * Des boutons pour appliquer différents filtres. * Utiliser `FileReader` pour lire l'image en tant que `ArrayBuffer`. * Passer l'ArrayBuffer au module WASM Rust. * Recevoir l'ArrayBuffer de l'image filtrée et l'afficher sur le canvas.

Ce projet est un excellent moyen de voir la puissance de Rust pour des calculs intensifs directement dans le navigateur.

Semaine 11 : Les bases de Go

Concepts à apprendre : * **Variables, fonctions, structs en Go** : Apprendre la syntaxe de base de Go pour la déclaration de variables, la définition de fonctions et la création de types structurés. * **Goroutines et canaux** : Comprendre les primitives de concurrence légères de Go pour l'exécution parallèle et la communication entre goroutines.

Exemples de code :

```go
package main

import (
    "fmt"
    "time"
)

// Définition d'une struct
type Person struct {
    Name string
    Age  int
}

// Fonction simple
func greet(name string) string {
    return fmt.Sprintf("Hello, %s!", name)
}

// Fonction avec plusieurs retours
func swap(x, y string) (string, string) {
    return y, x
}

// Goroutine simple
func worker(id int, done chan bool) {
    fmt.Printf("Worker %d starting\n", id)
    time.Sleep(time.Second)
    fmt.Printf("Worker %d done\n", id)
    done <- true // Envoyer un signal quand le travail est terminé
}

func main() {
    // Variables
    var i int = 10
    j := 20 // Déclaration et initialisation courtes
    fmt.Println("i:", i, "j:", j)

    // Constantes
    const PI = 3.14
    fmt.Println("PI:", PI)

    // Appeler une fonction
    message := greet("Alice")
    fmt.Println(message)

    // Appeler une fonction avec plusieurs retours
    a, b := swap("hello", "world")
    fmt.Println("Swapped:", a, b)

    // Structs
    p := Person{Name: "Bob", Age: 30}
    fmt.Println("Person:", p.Name, p.Age)

    // Goroutines et canaux
    done := make(chan bool, 1)
    go worker(1, done) // Lancer une goroutine
    <-done // Attendre que la goroutine ait terminé

    fmt.Println("Main function finished.")
}
```

Pour exécuter :

```
go run main.go
```

Ressources : * A Tour of Go

Projet : Convertisseur CLI CSV vers JSON

Développez un outil en ligne de commande qui prend un fichier CSV en entrée et le convertit en JSON. Ce projet vous permettra de pratiquer la lecture de fichiers, le parsing de données et la manipulation de structures en Go.

Créez un nouveau module Go :

```
mkdir csv_to_json_converter
cd csv_to_json_converter
go mod init csv_to_json_converter
```

Créez `main.go` :

```go
package main

import (
    "encoding/csv"
    "encoding/json"
    "fmt"
    "io"
    "os"
)

func main() {
    if len(os.Args) < 2 {
        fmt.Println("Usage: go run main.go <input.csv>")
        os.Exit(1)
    }

    filePath := os.Args[1]
    file, err := os.Open(filePath)
    if err != nil {
        fmt.Printf("Error opening file: %v\n", err)
        os.Exit(1)
    }
    defer file.Close()

    reader := csv.NewReader(file)
    reader.FieldsPerRecord = -1 // Permet des enregistrements avec un nombre variable de champs

    headers, err := reader.Read()
    if err != nil {
        fmt.Printf("Error reading CSV headers: %v\n", err)
        os.Exit(1)
    }

    var records []map[string]string
    for {
        record, err := reader.Read()
        if err == io.EOF {
            break
        }
        if err != nil {
            fmt.Printf("Error reading CSV record: %v\n", err)
            os.Exit(1)
        }

        // Créer une map pour chaque enregistrement en utilisant les en-têtes comme clés
        row := make(map[string]string)
        for i, header := range headers {
            if i < len(record) {
                row[header] = record[i]
            }
        }
        records = append(records, row)
    }

    jsonData, err := json.MarshalIndent(records, "", "  ")
    if err != nil {
        fmt.Printf("Error marshaling to JSON: %v\n", err)
        os.Exit(1)
    }
```

```
    fmt.Println(string(jsonData))
}
```

Créez un exemple de fichier CSV, `data.csv` :

```
Name,Age,City
Alice,30,New York
Bob,24,London
Charlie,35,Paris
```

Pour exécuter :

```
go run main.go data.csv > output.json
cat output.json
```

Semaine 12 : Go pour le Backend

Concepts à apprendre : * Construction d'API REST en Go avec `net/http` : Apprendre à créer des serveurs web, gérer les requêtes HTTP, router les URL et envoyer des réponses JSON.

Exemples de code :

Go dispose d'une bibliothèque standard robuste pour le développement web, `net/http`. Nous allons créer une API REST simple pour la gestion des utilisateurs.

Créez un nouveau module Go :

```
mkdir go_user_api
cd go_user_api
go mod init go_user_api
```

Créez `main.go` :

```go
package main

import (
    "encoding/json"
    "fmt"
    "log"
    "net/http"
    "strconv"
    "sync"
)

// User représente un utilisateur dans notre système
type User struct {
    ID   int    `json:"id"`
    Name string `json:"name"`
    Email string `json:"email"`
}

var (
    users  = make(map[int]User)
    nextID = 1
    mu     sync.Mutex // Mutex pour protéger l'accès concurrentiel à la map users
)

// createUserHandler gère la création de nouveaux utilisateurs
func createUserHandler(w http.ResponseWriter, r *http.Request) {
    if r.Method != http.MethodPost {
        http.Error(w, "Method not allowed", http.StatusMethodNotAllowed)
        return
    }

    var newUser User
    decoder := json.NewDecoder(r.Body)
    if err := decoder.Decode(&newUser); err != nil {
        http.Error(w, "Invalid request body", http.StatusBadRequest)
        return
    }

    mu.Lock()
    defer mu.Unlock()

    newUser.ID = nextID
    users[nextID] = newUser
    nextID++

    w.Header().Set("Content-Type", "application/json")
    w.WriteHeader(http.StatusCreated)
    json.NewEncoder(w).Encode(newUser)
}

// getUsersHandler gère la récupération de tous les utilisateurs ou d'un utilisateur spécifique
func getUsersHandler(w http.ResponseWriter, r *http.Request) {
    if r.Method != http.MethodGet {
        http.Error(w, "Method not allowed", http.StatusMethodNotAllowed)
        return
    }

    mu.Lock()
    defer mu.Unlock()
```

```
    idStr := r.URL.Path[len("/users/"):]
    if idStr != "" {
        id, err := strconv.Atoi(idStr)
        if err != nil {
            http.Error(w, "Invalid user ID", http.StatusBadRequest)
            return
        }
        user, ok := users[id]
        if !ok {
            http.Error(w, "User not found", http.StatusNotFound)
            return
        }
        w.Header().Set("Content-Type", "application/json")
        json.NewEncoder(w).Encode(user)
        return
    }

    // Retourner tous les utilisateurs
    allUsers := []User{}
    for _, user := range users {
        allUsers = append(allUsers, user)
    }
    w.Header().Set("Content-Type", "application/json")
    json.NewEncoder(w).Encode(allUsers)
}

// updateUserHandler gère la mise à jour d'un utilisateur existant
func updateUserHandler(w http.ResponseWriter, r *http.Request) {
    if r.Method != http.MethodPut {
        http.Error(w, "Method not allowed", http.StatusMethodNotAllowed)
        return
    }

    idStr := r.URL.Path[len("/users/"):]
    id, err := strconv.Atoi(idStr)
    if err != nil {
        http.Error(w, "Invalid user ID", http.StatusBadRequest)
        return
    }

    var updatedUser User
    decoder := json.NewDecoder(r.Body)
    if err := decoder.Decode(&updatedUser); err != nil {
        http.Error(w, "Invalid request body", http.StatusBadRequest)
        return
    }

    mu.Lock()
    defer mu.Unlock()

    if _, ok := users[id]; !ok {
        http.Error(w, "User not found", http.StatusNotFound)
        return
    }

    updatedUser.ID = id // Assurez-vous que l'ID correspond à l'URL
    users[id] = updatedUser

    w.Header().Set("Content-Type", "application/json")
    json.NewEncoder(w).Encode(updatedUser)
}
```

```go
// deleteUserHandler gère la suppression d'un utilisateur
func deleteUserHandler(w http.ResponseWriter, r *http.Request) {
    if r.Method != http.MethodDelete {
        http.Error(w, "Method not allowed", http.StatusMethodNotAllowed)
        return
    }

    idStr := r.URL.Path[len("/users/"):]
    id, err := strconv.Atoi(idStr)
    if err != nil {
        http.Error(w, "Invalid user ID", http.StatusBadRequest)
        return
    }

    mu.Lock()
    defer mu.Unlock()

    if _, ok := users[id]; !ok {
        http.Error(w, "User not found", http.StatusNotFound)
        return
    }

    delete(users, id)
    w.WriteHeader(http.StatusNoContent)
}

func main() {
    http.HandleFunc("/users", func(w http.ResponseWriter, r *http.Request) {
        switch r.Method {
        case http.MethodPost:
            createUserHandler(w, r)
        case http.MethodGet:
            getUsersHandler(w, r)
        default:
            http.Error(w, "Method not allowed", http.StatusMethodNotAllowed)
        }
    })
    http.HandleFunc("/users/", func(w http.ResponseWriter, r *http.Request) {
        switch r.Method {
        case http.MethodGet:
            getUsersHandler(w, r)
        case http.MethodPut:
            updateUserHandler(w, r)
        case http.MethodDelete:
            deleteUserHandler(w, r)
        default:
            http.Error(w, "Method not allowed", http.StatusMethodNotAllowed)
        }
    })

    fmt.Println("Server starting on :8080...")
    log.Fatal(http.ListenAndServe(":8080", nil))
}
```

Projet : API de gestion des utilisateurs en Go

L'exemple ci-dessus fournit une API CRUD complète pour la gestion des utilisateurs. Pour le projet, vous pouvez : 1. **Ajouter la validation des entrées :** Assurez-vous que

les champs `Name` et `Email` ne sont pas vides lors de la création ou de la mise à jour d'un utilisateur. 2. **Implémenter la persistance des données :** Actuellement, les données sont stockées en mémoire et sont perdues lorsque le serveur s'arrête. Intégrez une base de données simple (par exemple, SQLite avec la bibliothèque `database/sql` et un pilote comme `github.com/mattn/go-sqlite3`) pour stocker les utilisateurs de manière persistante. 3. **Ajouter des tests unitaires :** Écrivez des tests pour les gestionnaires de requêtes HTTP afin de garantir leur bon fonctionnement.

Synthèse : Comparaison Rust vs Go pour les serveurs

Après avoir travaillé avec Rust et Go sur des projets backend, il est utile de comparer leurs forces et faiblesses :

Caractéristique	Rust	Go
Performance	Extrêmement élevée, contrôle de bas niveau, pas de GC	Très élevée, GC performant, goroutines légères
Sécurité mémoire	Garantie par le compilateur (ownership, borrowing, lifetimes)	Gérée par le ramasse-miettes (GC), moins de garanties à la compilation
Concurrence	Modèle basé sur l'ownership (threads, async/await avec `tokio`)	Goroutines et canaux (modèle CSP)
Complexité	Courbe d'apprentissage plus raide (concepts de gestion mémoire)	Plus simple à apprendre et à utiliser, syntaxe concise
Écosystème	En croissance rapide, outils modernes (Cargo, `wasm-pack`)	Mature, bibliothèque standard riche, outils de déploiement simples
Cas d'usage typiques	Systèmes embarqués, jeux, navigateurs, services haute performance	Microservices, outils CLI, applications réseau, DevOps

Conclusion de la comparaison :

- **Choisissez Rust** si la sécurité mémoire sans compromis, la performance maximale et le contrôle de bas niveau sont critiques pour votre application (par exemple, des systèmes où les pannes sont inacceptables ou des services avec des exigences de latence très faibles).

- **Choisissez Go** si la simplicité, la rapidité de développement, la concurrence intégrée et une performance élevée (mais pas nécessairement maximale) sont vos priorités (par exemple, pour des microservices, des API ou des outils de ligne de commande).

Les deux langages sont d'excellents choix pour le développement backend moderne, et le choix dépendra souvent des exigences spécifiques du projet et des préférences de l'équipe.

Conclusion

Félicitations ! En suivant ce programme intensif de trois mois, vous avez acquis une base solide dans quatre langages de programmation essentiels : C, C++, Rust et Go. Vous avez exploré les concepts fondamentaux, approfondi des sujets avancés et mis en pratique vos connaissances à travers divers projets.

Ce guide est une feuille de route, mais l'apprentissage ne s'arrête jamais. Continuez à explorer, à construire et à contribuer à la communauté open source. Le monde du développement logiciel est vaste et en constante évolution, et vos nouvelles compétences vous ouvrent d'innombrables opportunités.

N'oubliez pas que la pratique régulière est la clé de la maîtrise. Même 2 heures par jour, si elles sont dédiées et concentrées, peuvent transformer vos compétences de manière significative. Bonne continuation dans votre parcours de développeur !

Résultat Final en 3 Mois

À la fin de ce programme intensif de 3 mois, vous aurez atteint les objectifs suivants :

- **Compréhension des concepts de bas niveau de C :** Vous maîtriserez les pointeurs, la gestion manuelle de la mémoire et les structures de données fondamentales, essentielles pour comprendre le fonctionnement interne des systèmes.
- **Maîtrise de la POO et du RAII avec C++ :** Vous serez capable de concevoir des applications orientées objet robustes et d'utiliser le principe RAII pour une gestion sécurisée des ressources.

- **Construction d'applications sûres, performantes et multiplateformes en Rust :** Grâce à la compréhension de l'ownership, du borrowing et des lifetimes, vous développerez des applications sans bugs liés à la mémoire, avec des performances exceptionnelles.
- **Développement de backends rapides et d'outils en Go :** Vous exploiterez la concurrence native de Go avec les goroutines et les canaux pour créer des services backend hautement performants et des outils CLI efficaces.

Ce guide vous fournit non seulement les connaissances techniques, mais aussi l'expérience pratique nécessaire pour aborder des projets complexes et vous positionner comme un développeur polyvalent et recherché sur le marché du travail.