C Programming For Absolute Beginners & Question Set III

Sahas Talasila

Contents

Files Introduction

Open a file: fopen()
Close a file: fclose()
Write to a file: fprintf(), fputs(), fwrite()
Read from a file: fscanf(), fgets(), fread()
Move the file pointer: fseek(), ftell()

Persistent data storage:
Files allow you to store data permanently on disk, even after the program terminates. This is useful for applications that need to retain data between runs, such as databases, configuration files, or log files.

Input/Output operations:
Files provide a way to read and write data to/from external storage devices, such as hard drives, solid-state drives, or flash drives. This enables your program to interact with the outside world, reading input from files and writing output to files.

Modularity and organization:
By breaking down a large program into smaller, independent files, you can:
1. Organize code into logical modules or components.
2. Make it easier to maintain and update individual files without affecting the entire program.
3. Allow multiple developers to work on different files simultaneously.

Reusability:
Files can be designed to be reusable across multiple programs or projects, making it easier to share code and reduce duplication.

Portability:
Files can be used to store platform-independent data, allowing your program to run on different operating systems or architectures without modification.

Error handling and debugging:
Files provide a way to log errors, debug information, or other relevant data, making it easier to diagnose and fix issues in your program.

Configuration and settings:
Files can be used to store configuration settings, user preferences, or other application-specific data, allowing your program to adapt to different environments or user preferences.

```c
// Writing to a File
#include <stdio.h>

int main() {
    FILE *file = fopen("example.txt", "w"); // Open file in write mode
    if (file == NULL) {
        printf("Error opening file.\n");
        return 1;
    }

    fprintf(file, "This is a test file.\n");
    fprintf(file, "Writing to files in C is simple!\n");

    fclose(file); // Close the file
    printf("Data written successfully.\n");

    return 0;
}
```

What if we want to read from a file?

```c
// Reading from a File
```

```c
#include <stdio.h>

int main() {
    FILE *file = fopen("example.txt", "r"); // Open file in read mode
    if (file == NULL) {
        printf("Error opening file.\n");
        return 1;
    }

    char line[100];
    while (fgets(line, sizeof(line), file)) { // Read lines from file
        printf("%s", line);
    }

    fclose(file); // Close the file
    return 0;
}
```

Using `fwrite` and `fread` for Binary Files

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

struct Person {
    char name[50];
    int age;
    float height;
};

int main() {
    struct Person p = {"John Doe", 30, 5.9};
    FILE *file = fopen("person.dat", "wb"); // Open file in binary write mode
    if (file == NULL) {
        printf("Error opening file.\n");
        return 1;
    }

    fwrite(&p, sizeof(struct Person), 1, file); // Write struct to file
    fclose(file);

    // Reading back the binary data
    file = fopen("person.dat", "rb"); // Open file in binary read mode
    if (file == NULL) {
        printf("Error opening file.\n");
        return 1;
    }

    struct Person readPerson;
    fread(&readPerson, sizeof(struct Person), 1, file); // Read struct from file
    fclose(file);

    printf("Name: %s, Age: %d, Height: %.2f\n", readPerson.name, readPerson.age,
   readPerson.height);

    return 0;
}
```

Structs

A struct (short for structure) in C is a user-defined data type that allows you to group variables of different data types under a single name. This grouping makes it easier to organize related data. Unlike arrays (which store multiple elements of the same type), a struct can store a mix of different data types (e.g., `int, float, char[]`, etc.).

Key Points

- Definition: You define a struct with the `struct` keyword, followed by a structure tag (optional) and curly braces containing members of the struct (or fields).
- Members: Each member has its own type and name.
- Instantiation: You can create variables of that struct type once it is defined.
- Accessing Members: Use the dot (.) operator to access members of a struct instance, e.g., `myStructVariabl`
- Pointers: When working with pointers to structs, you use the arrow operator (-¿) to access members, e.g., `myStructPointer->memberName`.

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

// Define a struct to store person's information
struct Person {
    char name[50];
    int age;
};

int main(void) {
    // Create an instance of struct Person
    struct Person john;

    // Assign values to the struct members
    strcpy(john.name, "John Doe");
    john.age = 30;

    // Print out the member values
    printf("Name: %s\n", john.name);
    printf("Age : %d\n", john.age);

    return 0;
}
```

Listing 1: Defining and using a simple struct in C

Explanation:

- `struct Person` is declared with two members: a `char` array `name` (to hold a string) and an `int age`.
- In `main`, we create a variable `john` of type `struct Person`.
- We use `strcpy` to copy the string ``John Doe"` into `john.name`.
- We directly assign an integer to `john.age`.
- We print the values to confirm that the `struct` was populated correctly.

Struct Initialisation and Functions
You can also initialise (creates) structs at the time of declaration or pass them to functions.

```c
#include <stdio.h>

```

```c
// Define a struct to store 2D point coordinates
struct Point {
    float x;
    float y;
};

// Function to print the coordinates of a Point
void printPoint(struct Point p) {
    printf("Point: (%.2f, %.2f)\n", p.x, p.y);
}

int main(void) {
    // Initialize a struct Point instance
    struct Point p1 = {3.5f, 4.8f};

    // Pass the struct to a function
    printPoint(p1);

    return 0;
}
```

Listing 2: Initializing a struct and passing it to a function

Explanation:

- `struct Point` holds two floating-point members, `x` and `y`.
- We define `printPoint` that accepts a `struct Point` argument.
- In `main\, we initialise the \verb`struct p1— with the values `{3.5f, 4.8f}`.
- We pass `p1` to `printPoint`, which prints out its coordinates.

Structs with Nested Structs:
Structs can also contain other structs, which allows complex data structures:

Example 4: Nested Struct Example:

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

struct Address {
    char street[50];
    char city[50];
    char country[50];
};

struct Person {
    char name[50];
    int age;
    struct Address address;  // nested struct
};

int main(void) {
    // Create a Person and fill in details
    struct Person alice;
    strcpy(alice.name, "Alice Wonderland");
    alice.age = 28;

    strcpy(alice.address.street, "123 Imaginary Rd");
    strcpy(alice.address.city, "Fictional City");
    strcpy(alice.address.country, "Neverland");

    // Print out the nested info
    printf("Name: %s\n", alice.name);
```

```
    printf("Age : %d\n", alice.age);
    printf("Address: %s, %s, %s\n",
           alice.address.street,
           alice.address.city,
           alice.address.country);

    return 0;
}
```

Listing 3: A struct that contains another struct

Explanation:

The `Person struct` has its own members (`name, age`) and a nested `struct Address`. We use the dot operator multiple times to access nested fields (e.g. `alice.address.street`).