# Module 9: Interacting with the System (Days 27-29)

## 1. Philosophy Focus: Knowledge Graph & Done-for-you Training Plan

You've built a robust, dynamic, and modular application. The next logical question is, "How do I save my data?" This module directly answers that by introducing file I/O, connecting your knowledge of pointers and structs to the file system. We then introduce command-line arguments, a professional feature that makes your application more flexible and powerful.

### **DAY 27: Reading From and Writing To Files**

**Goal:** Learn the fundamental C functions for file manipulation.

1. The FILE Pointer

In C, you interact with files through a special kind of pointer: FILE \*. Think of it as a handle that keeps track of which file you're working with, where you are in the file, etc. You get this pointer by calling fopen().

**2. The Core Functions: fopen, fprintf, fscanf, fclose**

* fopen("filename.txt", "mode"): Opens a file. The "mode" is a string that specifies what you want to do. Common modes are:
  + "r": Read (file must exist).
  + "w": Write (creates a new file, or **erases** an existing file).
  + "a": Append (adds to the end of an existing file, or creates it if it doesn't exist).
  + It returns a FILE \* on success or NULL on failure. **Always check for NULL!**
* fprintf(file\_ptr, "format string", ...): Just like printf, but it writes to a file instead of the console.
* fscanf(file\_ptr, "format string", ...): Just like scanf, but it reads from a file.
* fclose(file\_ptr): Closes the file, releasing the handle. For every successful fopen, you **must** have a matching fclose to prevent resource leaks.

**Inductive Example:**

#include <stdio.h>  
#include <stdlib.h>  
  
int main(void) {  
 FILE \*outfile = fopen("hello.txt", "w"); // Open for writing  
  
 if (outfile == NULL) {  
 printf("Error: Could not open file for writing.\n");  
 return 1;  
 }  
  
 fprintf(outfile, "Hello, C Files!\n");  
 fprintf(outfile, "This is line %d.\n", 2);  
 fclose(outfile); // Close it  
 printf("Wrote data to hello.txt\n");  
  
 // --- Now, let's read it back ---  
  
 FILE \*infile = fopen("hello.txt", "r"); // Open for reading  
 if (infile == NULL) {  
 printf("Error: Could not open file for reading.\n");  
 return 1;  
 }  
  
 char line1[100];  
 char line2[100];  
 int number;  
  
 // Read the first line  
 fgets(line1, sizeof(line1), infile);   
 // fgets is often safer than fscanf for whole lines  
  
 // Read the formatted second line  
 fscanf(infile, "This is line %d.\n", &number);  
   
 fclose(infile);  
  
 printf("\nRead from file:\n");  
 printf("Line 1: %s", line1); // fgets includes the newline  
 printf("Number from Line 2: %d\n", number);  
  
 return 0;  
}

**3. Day 27 Practice**

1. **Journal Writer:** Write a program that asks the user for a journal entry. Open a file named journal.txt in append mode ("a"). Write the user's entry to the file, followed by a newline. Run the program multiple times to see your journal grow.
2. **Journal Reader:** Write a separate program that opens journal.txt for reading ("r") and prints its entire contents to the console.

### **DAY 28: Making the To-Do List Persistent**

**Goal:** Integrate file I/O into your main project to save and load your task list.

**1. The Strategy**

* **Saving:** When the program is about to exit, we'll open a file (e.g., tasks.csv) in write mode ("w"). We'll loop through our array of tasks and fprintf each one to the file in a structured way. A "comma-separated value" (CSV) format is simple and effective: id,completed,description.
* **Loading:** When the program first starts, we'll check if tasks.csv exists and try to open it in read mode ("r"). We'll read the file line by line, parse the data, and dynamically add each task to our list, realloc-ing as needed.

**2. Day 28 Practice: Project Upgrade**

1. **Create save\_tasks function:**
   * In task.h, add the prototype: void save\_tasks(const char \*filename, struct Task list[], int count);
   * In task.c, implement the function. It should fopen the file in "w" mode. Loop from 0 to count and use fprintf to write each task's id, completed status, and description to the file, separated by commas, with a newline at the end. Don't forget to fclose.
2. **Create load\_tasks function:**
   * This is more challenging. The prototype in task.h will be tricky because it needs to modify your main list variables: int load\_tasks(const char \*filename, struct Task \*\*list\_ptr, int \*count\_ptr, int \*capacity\_ptr);
   * In task.c, implement it. fopen the file in "r" mode. If it fails (e.g., file doesn't exist), that's fine; just return 0.
   * If it opens, read the file line-by-line using fgets. For each line:
     + You need to parse it. sscanf(line, "%d,%d,%[^\n]", &id, &completed, desc\_buffer); is a good way to parse a CSV line. The %[^\n] part means "read every character until you hit a newline".
     + Once parsed, you have the data for a new task. Use your add\_task logic (checking capacity, reallocating, etc.) to add this loaded task to the list.
   * Return the number of tasks loaded.
3. **Integrate into main.c:**
   * At the very beginning of main, call load\_tasks(...).
   * Just before the return 0; at the end of main, call save\_tasks(...).

### **DAY 29: Command-Line Arguments (argc and argv)**

**Goal:** Make your program more professional by allowing the user to specify the data file on the command line.

1. The main function revisited

To accept command-line arguments, you must use a different signature for main:

int main(int argc, char \*argv[])

* int argc: (Argument Count) An integer holding the number of arguments passed to your program. This is always at least 1, because the program's name is the first argument.
* char \*argv[]: (Argument Vector) An array of strings. argv[0] is the program name, argv[1] is the first argument, argv[2] is the second, and so on.

**Example:** If you compile myprogram.c and run ./myprogram hello world, then:

* argc will be 3.
* argv[0] will be ./myprogram.
* argv[1] will be hello.
* argv[2] will be world.

**2. Day 29 Project Upgrade**

1. **Change main's signature:** In main.c, update int main(void) to int main(int argc, char \*argv[]).
2. **Determine the Filename:**
   * Inside main, create a const char \*filename variable.
   * Check the value of argc. If argc == 2, it means the user provided a filename. Set filename = argv[1];.
   * If argc != 2, the user didn't provide a filename. Set filename to a default value, like "tasks.csv".
3. **Use the Filename:** Pass this filename variable to your load\_tasks and save\_tasks calls.
4. **Compile and Test:**
   * gcc main.c task.c -o todo\_app
   * Run ./todo\_app (This will use tasks.csv). Add some tasks.
   * Run ./todo\_app school.txt. Add some school-related tasks.
   * Check your directory. You should now have two separate to-do list files! You've made your app much more versatile.