# Module 8: Building Larger Projects (Days 24-26)

## 1. Philosophy Focus: Knowledge Graph & Interleaved Instruction

This module is the organizational layer on top of everything you've learned. It takes your knowledge of functions, structs, and pointers and teaches you the standard C practice for separating them into logical units. The review project will interleave this new organizational skill with a thorough review of the dynamic to-do list, solidifying your grasp of memory management and data structures.

### **DAY 24: Header and Source Files - The Separation of Concerns**

**Goal:** Understand why and how to split code into interface (.h) and implementation (.c) files.

1. The Problem: The Giant main.c File

Imagine your to-do list program grows to thousands of lines. Finding functions, managing structs, and debugging becomes a nightmare. We need to group related code together. For our to-do list, everything related to the Task struct should be in its own logical unit.

2. The Solution: .h and .c Files

In C, we split a module into two files:

* **The Header File (.h):** This is the **public interface**. It tells *other* parts of the program *what* functions and types are available, but not *how* they work. It contains function declarations (prototypes), struct definitions, and macro definitions.
* **The Source File (.c):** This is the **private implementation**. It contains the actual code (the definitions) for the functions declared in the header file.

**Example:** Let's split our Task logic.

**task.h (The Interface)**

// This file declares WHAT is available for others to use.  
  
// We need these for our struct definition  
#include <stdio.h>  
#include <stdlib.h>  
  
// Struct definition  
struct Task {  
 int id;  
 char \*description; // Changed to a pointer for dynamic allocation  
 int completed;  
};  
  
// Function declarations (prototypes)  
void print\_tasks(struct Task list[], int count);  
  
// Notice no function bodies here!

**task.c (The Implementation)**

// This file defines HOW the functions work.  
// It must include its own header file!  
#include "task.h"   
// Use " " for local headers, < > for system headers.  
  
// Here is the actual function body  
void print\_tasks(struct Task list[], int count) {  
 printf("\n--- TO-DO LIST ---\n");  
 for (int i = 0; i < count; i++) {  
 char status = list[i].completed ? 'X' : ' ';  
 printf("%d. [%c] %s\n", list[i].id, status, list[i].description);  
 }  
 printf("--------------------\n");  
}

**main.c (The Consumer)**

#include <stdio.h>  
#include "task.h" // Include the header to use the Task struct and functions  
  
int main(void) {  
 struct Task my\_tasks[2];  
 my\_tasks[0] = (struct Task){1, "Buy milk", 0};  
 my\_tasks[1] = (struct Task){2, "Learn C", 1};  
   
 // We can call print\_tasks because its prototype is in task.h  
 print\_tasks(my\_tasks, 2);  
  
 return 0;  
}

**3. Day 24 Practice**

1. **Create the Files:** Create the three files above: task.h, task.c, and main.c.
2. **Compile Them:** Compiling multiple files is a new challenge we will address on Day 26. For now, just having the files organized is the goal. If you want to try, the command will look something like this: gcc main.c task.c -o myapp.

### **DAY 25: The Preprocessor and Header Guards**

**Goal:** Learn how to prevent common errors caused by including the same header file multiple times.

1. The Problem: Double Inclusion

What happens if main.c includes A.h and B.h, but A.h also includes B.h? The contents of B.h will be pasted into your code twice. This will cause the compiler to see the same struct definition twice, resulting in a "redeclaration" error.

2. The Solution: Header Guards

We can use special preprocessor directives to ensure the content of a header is only ever included once per compilation unit. The preprocessor runs before the main compiler.

The pattern is a universal convention in C/C++: #ifndef, #define, #endif.

**task.h (Now with a Header Guard)**

#ifndef TASK\_H // If TASK\_H is not yet defined...  
#define TASK\_H // ...then define it,  
  
// --- All of your original header content goes here ---  
#include <stdio.h>  
#include <stdlib.h>  
  
struct Task {  
 int id;  
 char \*description;  
 int completed;  
};  
  
void print\_tasks(struct Task list[], int count);  
// --- End of original header content ---  
  
#endif // And end the 'if' block.

**How it works:** The first time the compiler sees #include "task.h", TASK\_H is not defined, so it processes the whole file and defines TASK\_H. The second time it sees #include "task.h", TASK\_H *is* now defined, so the preprocessor skips everything between #ifndef and #endif.

**Rule of Thumb: Every .h file you ever write must have a header guard.**

**3. Day 25 Practice**

1. **Add Guards:** Add header guards to the task.h file you created yesterday. The name (TASK\_H) should be unique; a common convention is to base it on the filename.

### **DAY 26: Compiling and Linking Multiple Files**

**Goal:** Understand the two-stage process of building an executable from multiple source files.

The gcc command can do everything in one step, but it's useful to understand what's happening behind the scenes.

Stage 1: Compilation

The compiler takes a single .c file and turns it into a machine-code file called an object file (.o). The object file contains the compiled code for that file, but it doesn't know where the functions from other files are yet.

gcc -c main.c -> produces main.o

gcc -c task.c -> produces task.o

(The -c flag means "compile only, do not link.")

Stage 2: Linking

The linker takes all the object files and "links" them together. It resolves the cross-references (e.g., when main.o calls a function that lives in task.o) and combines them into a single, final executable file.

gcc main.o task.o -o myapp -> produces myapp

You can also do it all in one go, which is what we did on Day 24:

gcc main.c task.c -o myapp

**Review Project: Fully Modular To-Do List**

**Goal:** Apply this module's lessons to fully refactor your dynamic to-do list from Module 7.

1. **Create the Files:**
   * task.h: Will contain the Task struct definition and function prototypes for add\_task, print\_tasks, etc. Don't forget the header guard!
   * task.c: Will contain the implementations (function bodies) for all the task-related functions. It must #include "task.h".
   * main.c: Will contain your main program loop (the menu switch statement). It should be much cleaner now, mostly just calling functions defined in task.c. It must also #include "task.h".
2. **Refactor the Code:**
   * Move the Task struct definition into task.h.
   * Move the function definitions (add\_task, print\_tasks) into task.c.
   * Place the function prototypes for those functions into task.h.
   * The add\_task function from Day 23 had a tricky struct Task \*\*list parameter. Make sure you get the prototype in the .h file to match the definition in the .c file exactly.
   * Ensure main.c is now much smaller and focused only on the user interaction loop.
3. Compile and Run: Use the multi-file compilation command to build your project and test it thoroughly.  
   gcc main.c task.c -o todo\_app  
   Then run with ./todo\_app.