# Module 10: Capstone Project - Inventory Management System (Days 30-34)

## 1. Philosophy Focus: Spaced Repetition & Inductive Teaching

This is it. This final project is a comprehensive review of the entire course, designed to make the patterns you've learned second nature. You will be building a complete program from the ground up, applying your knowledge of structs, dynamic arrays, multi-file projects, and file I/O. We will guide the structure, but you will discover the implementation details on your own, solidifying your fluency in C.

### **DAY 30: Project Planning and Design**

**Goal:** Design the data structures and plan the program architecture before writing any code. A good plan is the most important part of a successful project.

1. The Project Brief

We will build a command-line inventory management system for a small store. The program must be able to:

* Add a new product to the inventory (with a unique ID, name, price, and quantity).
* Display a list of all products.
* Update the quantity of an existing product.
* Search for a product by its name.
* Save the inventory to a file.
* Load the inventory from a file when the program starts.

2. Designing the Data Structure (product.h)

What information do we need for each product?

* A unique identifier (an integer is fine).
* A name (a string).
* A price (a floating-point number, like double).
* A quantity in stock (an integer).

Based on this, define the Product struct in a new file, product.h. Don't forget your header guard!

3. Planning the Functions (product.h and product.c)

What actions will we need to perform? Think about the project brief and translate it into function prototypes. Add these to product.h and create empty implementations in a new product.c file.

* void add\_product(...): Will need to take the list, counts, and the new product's info.
* void print\_inventory(...): Will need the list and the count.
* Product\* find\_product\_by\_name(...): A function that searches the list for a product by its name and returns a *pointer* to it if found, or NULL if not found. This is a very common and powerful pattern.
* void save\_inventory(...): Similar to the to-do list.
* void load\_inventory(...): Also similar to the to-do list.

4. Planning the Main Loop (main.c)

In a new main.c file, sketch out the main function's logic.

* It will need variables for the dynamic array (Product \*inventory), count, and capacity.
* It will need to call load\_inventory at the start.
* It will have a while loop with a switch statement for the user menu (Add, List, Update, Search, Exit).
* It will call save\_inventory at the end.

By the end of today, you should have three files (main.c, product.h, product.c) that are mostly empty but contain the complete "skeleton" of your application.

### **DAY 31-32: Core Logic and UI Implementation**

**Goal:** Flesh out the skeleton you designed, building the core functionality of the application.

1. Implement Core Functions

Work through the empty functions in product.c and write the code for them.

* **add\_product**: This will be very similar to your add\_task function. It must check the capacity and realloc the inventory array if it's full before adding the new product.
* **print\_inventory**: This should be straightforward. Loop through the array and print the details of each product in a nicely formatted table.
* **find\_product\_by\_name**: Loop through the inventory. Use strcmp() (from <string.h>) to compare the name of each product with the target name. If you find a match, return the address of that Product struct in the array (&inventory[i]). If the loop finishes without a match, return NULL.

2. Implement the Main Loop

In main.c, write the code for your user menu.

* For "Add", prompt the user for the name, price, and quantity. Then call add\_product.
* For "List", call print\_inventory.
* For "Update", first ask the user for the name of the product to update. Call find\_product\_by\_name.
  + If the function returns NULL, print "Product not found."
  + If it returns a valid pointer, ask the user for the new quantity and update it directly through the pointer (e.g., found\_product->quantity = new\_quantity;).
* Leave "Search" and "Exit" for now.

3. Compile and Test Incrementally

As you complete each function, compile your code (gcc main.c product.c -o inventory) and test it thoroughly. Don't wait until everything is written to compile for the first time. Testing as you go makes debugging much easier.

### **DAY 33: Persistence and Advanced Features**

**Goal:** Make the inventory persistent and implement the search functionality.

1. Implement save\_inventory and load\_inventory

This is a direct application of the skills you learned in Module 9.

* **save\_inventory**: Open the specified file in "w" mode. Loop through the products and use fprintf to save them in a CSV format (e.g., id,price,quantity,name\n). Be careful with the product name, as it's a string.
* **load\_inventory**: Open the file in "r" mode. Read line-by-line with fgets. Parse each line using sscanf (remember the %[^\n] trick for the name). For each parsed product, call your existing add\_product function to add it to the dynamic array.
* Integrate these calls into the start and end of your main function.

2. Implement Search

In your main.c switch statement:

* For the "Search" case, prompt the user for a product name.
* Call find\_product\_by\_name.
* If it returns a pointer, print the details of the found product.
* If it returns NULL, print "Product not found."

3. Final Testing

Your application should now be fully functional. Test all features:

* Add several products. Exit the program.
* Run it again. Does it load the products correctly?
* Update a quantity. Save and reload. Is the new quantity still there?
* Search for a product that exists and one that doesn't.

### **DAY 34: Course Wrap-up and The Path Forward**

**Congratulations!** You have completed this course and built a serious, non-trivial C application. You've journeyed from printf("Hello, World!"); to manual memory management, multi-file projects, and persistent data storage. You should feel proud of what you've accomplished. C is a language that rewards a deep understanding of the machine, and you now have a solid foundation in it.

**Reflecting on the Philosophy:**

* **Spaced Repetition:** You didn't just learn realloc once; you used it in the to-do list and again in the inventory manager. Each core concept was revisited multiple times in new contexts.
* **Knowledge-graph-based learning:** Each module built directly on the last. You couldn't have understood dynamic arrays without pointers, and you couldn't have built a multi-file project without first understanding functions.
* **Inductive teaching:** We provided the goals ("build a persistent to-do list") but you discovered the exact implementation, debugging your own code and learning the patterns far more deeply than if you had just been told them.
* **Interleaved instruction:** You learned a little bit about pointers, then strings, then structs, then came back to use them all together with dynamic memory. This is how deep retention is built.

What's Next?

Your journey as a programmer is just beginning. C is a gateway to understanding almost every other area of computer science. Here are some paths you can explore from here:

1. **Data Structures and Algorithms:** You've built a dynamic array (a vector). Now learn to build Linked Lists, Hash Tables, Trees, and Graphs in C. This is the fundamental basis of all advanced software engineering.
2. **Systems Programming:** Explore how operating systems work. Learn about processes, threads, sockets for networking, and interacting with the OS at a lower level. The book "The C Programming Language" (K&R) is the timeless classic for this.
3. **Embedded Systems:** C is the king of programming for microcontrollers (like Arduinos) and other hardware. If you enjoy making things in the physical world light up and move, this is a fascinating field.
4. **Contribute to Open Source:** Many of the world's most important software projects are written in C (Linux Kernel, Git, SQLite). Find a project you admire, study its code, and try to fix a small bug. It's the fastest way to learn from experts.
5. **Revisit Rust:** Now that you deeply understand manual memory management, go back and look at Rust. You will have a profound new appreciation for the safety guarantees provided by the borrow checker and the concept of ownership. You'll see *why* Rust was designed the way it was.

Thank you for taking this journey. Keep coding, stay curious, and build amazing things.