COMP 205

System Programming

Team Project

Storage Management System Report

Instructor: Buse Yılmaz

Ertuğ Yağız Özen
042101120

Murathan Çetin
042001039

Berat Kaya
042001046

Abstract

The program is a system that provides basic storage of products. Users can perform various operations such as adding, deleting products, filtering by category or quantity range. This structure stores products using structures and stores data using file input/output. Users can access the system through a menu-based interface and benefit from additional functionality that simplifies inventory management.

1.Introduction

Storage Management System helps to access to user to use bunch of type functionalities for example displaying product information as this project's main idea. Also adding and deleting products, also finding minimum and maximum value of product. With these operations products information can be saved and loaded from a file and with type of products, stocks can be filtering and ranged. With power of including other data's can be called from other files. This attribute of C language data storage and management can be overcome. On this project as in the **Figure 1.0** other files are called.

```
1  #ifndef STORAGE_H
2  #define STORAGE_H
```

```
3  #include <string.h>
4  #include "storage.h"
```

(Figure 1:Definitions)

2.Project Components

2.1 Main Program('main.c')

'main' program is a typical user interface part that we have used. And the main function is just used for calling the '**getInput**' function. But on the same file there are 2 functions are created and initiated '**displayMenu**' and '***getInput***'

```
5  void displayMenu() {
6      printf("\n Lawrence Craft's Storage\n");
7      printf("1. Add Product\n");
8      printf("2. Delete Product\n");
9      printf("3. Show All Products\n");
```

(Figure 2: displayMenu Function)

As can be seen, '**displayMenu**' is just displaying the values stored, deleted etc. and this function will be called.
'**getInput**"s mission is greater. This function takes variables from "**storage.h**" and is used inside of loops and functions. And initiated product's information.
Also Products p variable has been created on that function. Using case add, delete, category, find minimum and maximum values. etc. information.

```
65              case 8:
66                  printf("Category that will be filtered: ");
67                  scanf("%s", category);
68                  filterProductsByCategory(products, productCount, category);
69                  break;
70              case 9:
71                  printf("Min: ");
72                  scanf("%d", &minQuantity);
73                  printf("Max: ");
74                  scanf("%d", &maxQuantity);
75                  filterProductsByQuantity(products, productCount, minQuantity,
                        maxQuantity);
76                  break;
```

(Figure 3: Cases)

2.2 Storage Header ('storage.h')

'**storage.h**' make serves information for the main file. While using typedef for '**Product**' structures encapsulates essential values and information about every product in physical. Also '**storage.h**' file has major task that declares functions for essential operations on product like adding, deleting, saving, loading etc. as can be seen on **Figure 1.2**

- **addProduct**: Used to add a new product.
- **deleteProduct**: Used to delete a specific product.
- **printAllProducts**: Used to display all products.
- **findProductWithLeastStock**: Used to find the product with the least stock.
- **findProductWithMostStock**: Used to find the product with the most stock.
- **saveProductsToFile**: Used to save products to a file.
- **loadProductsFromFile**: Used to load products from a file.
- **filterProductsByCategory**: Used to filter products by category.
- **filterProductsByQuantity**: Used to filter products within a specific quantity range.

```
18 void addProduct(Product *products, int *productCount, const Product newProduct);
19 void deleteProduct(Product *products, int *productCount, int id);
20 void printAllProducts(const Product *products, int productCount);
21 void findProductWithLeastStock(const Product *products, int productCount);
22 void findProductWithMostStock(const Product *products, int productCount);
23 void saveProductsToFile(const Product *products, int productCount);
24 void loadProductsFromFile(Product *products, int *productCount);
25 void filterProductsByCategory(const Product *products, int productCount, const char
       *category);
26 void filterProductsByQuantity(const Product *products, int productCount, int
       minQuantity, int maxQuantity);
```

(Figure 4: Functions in header file)

2.3 Storage Implementation ('storage.c')

'**storage.c**' is the main event stage. Functions that are created on that file are doing every calculation and tasks which are primary works. Like already mentioned save, load, add, delete etc. on (**Figure 4**). This file's module encapsule the core functionality of the project.

```
78
79  void saveProductsToFile(const Product *products, int productCount) {
80      FILE *file = fopen("products.dat", "wb");
81      if (file == NULL) {
82          perror("File Couldn't Open");
83          return;
84      }
85
86      fwrite(products, sizeof(Product), productCount, file);
87      fclose(file);
88  }
```

(Figure 5:)

2.4 Makefile

'**Makefile**' is a file that determines how a program will be compiled, what source files to use, compilation options, and how the program will be built. Makefile is often used in large projects or projects containing multiple source files.

3.Progress of System Flow

3.1 Operation Execution

The selected operation is seamlessly executed by harnessing the functionality encapsulated within the meticulously crafted functions residing in the '**storage.c'** module. To illustrate, the addition of a product unfolds as an orchestrated process involving the meticulous collection of product particulars, orchestrating an orchestrated update to the product array. In a similar vein, the intricacies of filtering are unfurled through a meticulous traversal of the product array, meticulously scrutinizing and sifting through the data based on the finely tuned criteria specified by the discerning user. This intricately woven execution model reflects the sophistication and finesse embedded in the core functionalities of the system.

3.2 User Interaction

Users can interact with the program by menu option. System will respond to the user's inputs and with those inputs the program will be interacted with.

3.3 Data Persistence

Program supports load and save information about product data from a file ('***products.dat***')

4.Key Functionalities of Program

4.1.Data Persistence

‘**storage.c**’ file has operating functions. (‘**saveProductsToFile**’) and (‘**loadProductsFromFile**’) are 2 of them. These functions provide saving and loading stock and identifies and displays minimum and maximum of stock.

4.2.Stock Analysis

Same data persistence functions are called from ‘**storage.c**’ and initiated of. (‘**findProductWithLeastStock**’) and (‘**findProductWithMostStock**’) these are identified stock status.

4.3.Product Management.

(‘addProduct’): Allows the user input for the product, sends the inputs to functions and these data are added to the product storage.

(‘deleteProduct’): Allows the user input for the product, sends the inputs to functions and these data are deleted from the product storage.

(‘printAllProducts’): Display a list for all information available from storage about products.

4.4.Filtering

(‘filterProductsByCategory’): Filters products given information and displayed in specified categories.

(‘filterProductsByQuantity’): Filters products given information and displayed on a specified quantity range.

4.5.User Interface

User interface progress is happening on the '**main.c**' file so it can be primary run.

('displayMenu'): Provide a menu for users as much as understandable for input through available user enters.

('getInput'): Take and capture user input and executing operations.

5.Debugging

Missing Error Checks:

The addProduct function gives a warning when trying to add a product when the warehouse is full, but it does not tell the user which product cannot be added. In this case, more specific information can be given about which product cannot be added. In the deleteProduct function, a warning is given when the product to be deleted cannot be found, but it is not stated which ID could not be found.

File Operations:

File operations have error checks, but filenames are hardwired. This does not provide a more flexible structure and should be available for different filenames. Error checking during file operations can provide more understandable error messages to the user. For example, a more descriptive error message may be given when the file cannot be opened.

User Login Controls:

String reading for user logins is done using %s. This could pose a security risk because no specific size is specified. Secure login operations must be performed by taking into account the size limitations of the scanf function.
When user inputs are incorrect, more descriptive feedback can be given to the user.

General Improvements:

Functions can be made more modular. For example, the functions findProductWithLeastStock and findProductWithMostStock can be converted into a general function to check a specific criterion.

Adding more error handling and checking mechanisms in functions can create a more robust code base.

Performance Improvements:

Some functions can negatively impact performance by using loops over large lists of items. This should be taken into consideration, especially when working on large data sets, and improvements can be made to increase performance.

By focusing on these points in your report, you can make suggestions to increase the reliability and usability of the project. These fixes can make the code more robust and user-friendly.

Challenges We Faced

We encountered some difficulties while developing this project. Checking and securely processing user inputs presented some challenges to avoid memory overflows, especially for string inputs. Enabling the scanf function to retrieve data of certain sizes required some extra steps to ensure safe reading, especially for strings. It was also important to properly handle error situations during file operations and provide understandable feedback to the user. Considering the possibility that performance may be poor when operating loops over the data structure, some improvements were required to ensure that the code was both reliable and performant. To overcome these challenges, we took a more thoughtful and modular approach. We then encountered some difficulties in creating the Makefile. It was important to specify dependencies between source files. For example, the file storage.o depends on the files storage.c and storage.h. Specifying the correct dependencies prevents the compilation process from being repeated unnecessarily when files need to be updated. Although we do this in the vast majority of cases, we still have some gaps.

6.Conclusion

In conclusion, the project encapsulates a multifaceted journey, using together elements of design, functionality, and adaptability. The Storage Management System, at its core, is an intricate innovation and craftsmanship.

6.1.Holistic Project Integration:

The modular architecture, orchestrated in '**storage.h'** and '**storage.c'**, symbolizes the careful integration of various project components

6.2.Project Dynamics in Action:

The core functionalities, meticulously coded in '**storage.c'**, breathe life into the project. The user interface, manifested in '**main.c'**, serves as the gateway for users to interact seamlessly with the underlying complexities. This harmony between design and functionality creates a dynamic environment for project execution.

6.3.User-Centric Design:

The way users see and use the project might look plain, but there's a whole clever setup behind the scenes. The code in **'main.c'** is like a maestro, directing how users can do things smoothly. It's like making sure the project is user-friendly, so anyone can get the hang of it without scratching their heads.

In simple terms, it's the brainy part of the project that makes sure everything clicks for the users. Without it, the project wouldn't be as easy to use, and that's a big deal for making sure it works for everyone. It's like the friendly guide that helps users navigate through the project without getting lost in the techy details

Versatility in Scalability:

Beyond storage management, the project showcases versatility in scalability. The predefined maximum product limit and the ability to read/write product data to a file demonstrate foresight in accommodating varying data volumes. This adaptability positions the project as a versatile solution for diverse data management needs.

REFERENCES

- *Build software better, together*. (2024, January 7). GitHub. https://github.com/topics/storage

- *Build software better, together*. (2023, December 27). GitHub. https://github.com/topics/warehouse-management-system