



East West University

Faculty of Science and Engineering
Department of Computer Science and Engineering
Course: CSE207 - Data Structures
Academic Session: Fall 2024

Project Report

CrimsonCare Blood Management System

Submitted to

Dr. Hasan Mahmood Aminul Islam, Ph.D.

B.Sc. (CSE, BUET), M.Sc. (Networking & Services, Helsinki), Ph.D. (Aalto)

Assistant Professor

Department of Computer Science and Engineering
East West University

Submitted by

Maysha Taskin Iqra (2023-1-60-152)

Sabiha Akter Chaity (2023-2-60-057)

Sumyya Tabassum (2023-3-60-351)

Arnab Saha (2021-3-60-201)

Md Shahoriyer Nadim (2023-3-60-189)

Md Asaduzzaman Atik (2023-1-60-130)

January, 2025

Abstract

CrimsonCare is a computer program made to manage blood donation and transfusion. It was created as a final project for the CSE207 course at East West University. The goal of the project is to show how well we can use data structures and C programming.

The system helps manage hospitals, blood stock, transactions, and other tasks. Admins can add, delete, and update information about hospitals, blood stocks, and admins itself. It checks all inputs carefully to keep the data correct and the system working well.

We used linked lists to store and manage data. Before the program closes, it frees all memory to stop memory leaks. The system works on Windows, Linux, and macOS, and it supports both Debug and Release modes.

In the future, we plan to add a database SQLite or at least JSON to save data for a longer time and improve security with password hashing. This project shows how we can use data structures and C programming to solve real-life problems.

Contents

1	Introduction	4
1.1	Background	4
1.2	Objectives	4
1.3	Scope	5
1.3.1	Functional Scope	5
1.3.2	Technical Scope	6
1.3.3	Validation and Error Handling	6
1.3.4	Future Enhancements	6
1.3.5	Limitations	7
1.4	Methodology	7
1.4.1	Requirement Analysis	7
1.4.2	System Design	7
1.4.3	Implementation	8
1.4.4	Input Validation and Error Handling	8
1.4.5	Documentation	8
1.4.6	Future Enhancements Planning	8
2	Project Description	9
2.1	Problem Statement	9
2.1.1	Main Problems in Blood Management:	9
2.1.2	The Need for a Solution	10
2.2	Proposed Solution	10
2.2.1	Hospital Management	10
2.2.2	Blood Stock Management	10
2.2.3	Transaction Logging	11
2.2.4	Administrative Functions	11
2.3	Project Structure	12
3	System Design	14
3.1	Architecture	14
3.1.1	High-Level Architecture	14
3.1.2	How the Parts Work Together	15
3.2	Data Saving and Loading	16
3.3	Why This Design is Good	16
3.4	Data Structures	17
3.4.1	Linked List	17
3.4.2	Operation of Linked Lists	19
3.4.3	Memory Management	22
3.5	Modules	23

3.5.1	Main Module	23
3.5.2	Miscellaneous Functions Module	23
3.5.3	Blood Manager Module	24
3.5.4	Hospital Manager Module	24
3.5.5	Admin Manager Module	25
3.5.6	Transaction Manager Module	25
4	Implementation	26
4.1	Development Environment	26
4.2	Function Implementations	27
4.2.1	Admin Manager Module	27
4.2.2	Blood Manager Module	35
4.2.3	Hospital Manager Module	41
4.2.4	Transaction Manager Module	48
4.2.5	Miscellaneous Functions Module	53
4.3	Input Validation	58
4.4	Error Handling	59
5	Usage	60
5.1	Installation	60
5.1.1	Prerequisites	60
5.1.2	Clone the Repository	61
5.1.3	Build for Code::Blocks IDE	61
5.1.4	Build for Command Line (Using Make)	61
5.1.5	Build for Command Line (Without Make)	62
5.2	Running the Application	63
5.2.1	Running the Application on Linux/Mac	63
5.2.2	Running the Application on Windows	63
5.2.3	Application Usage	64
6	Future Work	65
6.1	Enhancements	65
6.2	Database Integration	65
6.3	Security Improvements	65
7	Conclusion	66
7.1	Summary of Achievements	66
7.2	Final Thoughts	66
8	References	67
A	Appendix A: Source Code	68
A.1	Source Code	68
A.1.1	main.c	68
A.1.2	admin_manager.c	81
A.1.3	blood_manager.c	93
A.1.4	hospital_manager.c	104
A.1.5	transaction_manager.c	114
A.1.6	misc.c	121

A.2	Header Files	129
A.2.1	admin_manager.h	129
A.2.2	blood_manager.h	135
A.2.3	hospital_manager.h	142
A.2.4	transaction_manager.h	148
A.2.5	misc.h	152

Chapter 1

Introduction

1.1 Background

Blood is very important in healthcare. It helps save lives in emergencies. To help, we need a system that can manage blood donations, storage, and use.

CrimsonCare is a project for the CSE207 course at East West University. The program helps manage hospitals, track blood stock, and record transactions. It is not meant to replace real-world systems. It is made to show how well we can use data structures and C programming.

The system uses linked lists to store data. Before the program closes, it frees all memory to avoid memory leaks. The program works on many systems like Windows, Linux, and macOS. In the future, we want to make it better by adding a database to save data and making it safer with passwords.

1.2 Objectives

The CrimsonCare Blood Management System has these main goals:

- **Show Skills in Data Structures and C Programming**
 - Use data structures like linked lists to manage data easily.
 - Build a strong console program in C for managing blood records.
- **Create a Full Blood Management System**
 - Make a system that can manage hospitals, blood stock, and transactions.
 - Allow admins to add, remove, or update records for hospitals, blood stocks, and admins itself.
- **Keep Data Safe and Correct**
 - Check all inputs to make sure the data is correct.

- Add error handling to manage errors and keep the system reliable.
- **Support Cross-Platform**
 - Make the program work on Windows, Linux, and macOS.
 - Support both Debug and Release modes.
- **Future Improvements**
 - Add a database or at least more logical file types like JSON.
 - Enhance security with password hashing.
- **Learn More About Data Structures and C Programming**
 - Learn more about data structures and C programming.
 - Improve our skills in data structures and C programming.
- **Provide Clear Documentation**
 - Provide clear documentation for the project.
 - Provide a Doxygen documentation for the project.

1.3 Scope

The scope of the CrimsonCare Blood Management System includes the features and limits listed below:

1.3.1 Functional Scope

- **Hospital Management**
 - Admins can add, delete, and update hospital records.
 - Each record includes the hospital name, location, and a unique code.
- **Blood Stock Management**
 - Admins can add, delete, and update blood stock records.
 - Each record includes the blood type, price, and quantity.
- **Transaction Logging**
 - The system keeps records of blood donations/SELL and blood requests/BUY.
 - Each record includes the type of transaction, blood group, quantity, date, and a unique token (if type is donation/SELL).
- **Administrative Functions**
 - Admins can add, delete, or update records.
 - Admin access requires a password for security.
 - All sensitive actions require admin confirmation.

1.3.2 Technical Scope

- **Programming Language**
 - The program is written in C programming language.
- **Platform Compatibility**
 - The program works on Windows, Linux, and macOS.
- **Build Configurations**
 - The program supports both Debug and Release modes.
- **Data Structures**
 - The program uses linked lists to manage data.
 - The program uses dynamic memory allocation to store data.
 - All memory is cleared before the program closes to prevent memory leaks.

1.3.3 Validation and Error Handling

- **Input Validation**
 - The program checks all inputs to make sure the data is correct and safe.
- **Error Handling**
 - The program adds error handling to manage errors and keep the system reliable.

1.3.4 Future Enhancements

- **More Logical Storage**
 - We plan to add a database or at least more logical file types like JSON to save data for a long time.
- **Enhanced Security**
 - We plan to add password hashing to enhance security.
 - We plan to add encryption to the data to keep it safe.

1.3.5 Limitations

- **Console Application**

- The program is a console application.
- The program does not support graphical user interfaces (GUIs).

- **Limited Features**

- This program uses file based storage.
- The program does not support cloud storage or remote access.
- The program does not support real-time data synchronization.
- The program is not thread-safe.

1.4 Methodology

The CrimsonCare Blood Management System was built step by step to meet the project goals. The steps we followed are explained below:

1.4.1 Requirement Analysis

- **Goal:** To understand what the system should do.
- **What We Did:**
 - We talked as a team to decide on important features like hospital management, blood stock, and transactions.
 - We wrote down the plan and what the system needs to do.

1.4.2 System Design

- **Goal:** To plan how the system will work.
- **What We Did:**
 - Made a diagram to show the system parts and how they connect.
 - Designed linked lists to handle data easily.
 - Created design documents to explain each part of the system.

1.4.3 Implementation

- **Goal:** To build the system based on the plan.
- **What We Did:**
 - Used C programming to make the system.
 - Built parts for hospital management, blood stock, transactions, and admin functions.
 - Made sure the program works on Windows, Linux, and macOS.

1.4.4 Input Validation and Error Handling

- **Goal:** To make sure the data is correct and safe.
- **What We Did:**
 - Checked inputs to prevent invalid data from being added to the system.
 - Added error handling to manage errors and keep the system reliable.
 - Tested the system to find and fix any bugs.

1.4.5 Documentation

- **Goal:** To explain how the system works.
- **What We Did:**
 - Created a Doxygen documentation for the project.

1.4.6 Future Enhancements Planning

- **Goal:** To think about how to make the system better.
- **What We Did:**
 - We planned to add a database or at least more logical file types like JSON to save data for a long time.
 - Planned to add password security and data encryption to make the system safer.

Chapter 2

Project Description

2.1 Problem Statement

Blood management is very important in healthcare. Hospitals need to store, track, and give blood to save lives, especially in emergencies. But many hospitals have problems because they do not have good systems to manage blood.

2.1.1 Main Problems in Blood Management:

- **Tracking Blood Donations and Requests**
 - Hospitals must keep records of blood donations and how blood is used.
 - They need to know the type, amount, and date of each donation.
- **Blood Stock Management**
 - Hospitals must keep track of blood stock levels.
 - They need a system to monitor blood levels and restock when needed.
- **Keeping Data Safe and Correct**
 - Hospitals must ensure that data is safe and correct.
 - They need to check all inputs to prevent invalid data from being added.
- **Administrative Tasks**
 - Adding, deleting, or updating hospital and blood records must be secure.
 - Only authorized people should make changes to the system.
- **Working on Different Systems**
 - Hospitals use Windows, Linux, or macOS.
 - The system must work on all these platforms.

2.1.2 The Need for a Solution

To solve these problems, we need a strong and simple system. The CrimsonCare Blood Management System is designed to:

- Help manage hospitals, blood stocks, and transactions.
- Keep data safe by checking inputs and fixing errors.
- Work on Windows, Linux, and macOS.

By solving these problems, CrimsonCare shows the team's skills in data structures and C programming while giving hospitals a helpful tool for managing blood.

2.2 Proposed Solution

The CrimsonCare Blood Management System gives a simple and strong solution to the problems in blood management. This solution uses data structures and C programming to build an efficient console application. The main parts of the solution are:

2.2.1 Hospital Management

- **What It Does:**
 - Add, delete hospital records.
 - Each record has the hospital name, location, and a unique code.
 - Deleting hospital requires admin confirmation.
- **How It Works:**
 - Uses linked lists to handle a growing list of hospitals.
 - Linked lists allow adding, or finding records easily.
- **Validation:**
 - Checks all inputs to make sure the data is correct.
 - Checks for duplicate hospital codes.

2.2.2 Blood Stock Management

- **What It Does:**
 - Add, delete, and update blood stock records.
 - Each record includes the blood type, price, and quantity.

- **How It Works:**

- Uses linked lists to handle a growing list of blood stocks.
- This makes it easy to update or search blood stock.

- **Validation:**

- Checks all inputs to make sure the data is correct.
- Checks for duplicate blood group codes.

2.2.3 Transaction Logging

- **What It Does:**

- Keep records of blood donations and requests.
- Each record includes the type of transaction, blood group, quantity, date, and a unique token (if type is donation).

- **How It Works:**

- It uses real-time data fetching directly from files.

- **Validation:**

- Checks all inputs to make sure transaction records are correct before saving them.

2.2.4 Administrative Functions

- **What It Does:**

- Add, delete, or update records.
- Admin access requires a password for security.
- All sensitive actions require admin confirmation.
- It stores admin data in a dat file for persistence with a surface-level encryption.
- It prevents deleting self-admin.

- **How It Works:**

- Uses linked lists to manage admin records.
- This allows easy updates and secure operations.

- **Validation:**






















- Checks all inputs to make sure admin data is correct.
- Checks for duplicate admin usernames.











By using these features, the CrimsonCare Blood Management System solves many problems in blood management. This project shows how the team used data structures and C programming to create a helpful and reliable system.

2.3 Project Structure

The CrimsonCare Blood Management System project is organized into different folders and files. This makes the project easy to work on, test, and understand. Below is a list of the folders and files, along with what they do:

Main Files and Folders

-  `.editorconfig`: Keeps coding styles the same in all editors.
-  `.gitignore`: Tells Git to skip certain files and folders during version control.
-  `CrimsonCare.cbp`: Project file for the Code::Blocks IDE.
-  `CrimsonCare.layout` and `CrimsonCare.workspace`: Files to manage how the project looks in Code::Blocks.
-  `Doxyfile`: A settings file used to create automatic documentation using Doxygen.
-  `LICENSE.md`: A file with the rules (license) for using this project.
-  `main.c`: The main file where the program starts running.
-  `Makefile`: A file to build the project using commands in the terminal.
-  `README.md`: A file with an overview of the project, how to install it, and how to use it.
-  `include/`: This folder has header files (`.h`) for different parts of the project:
 -  `admin_manager.h`: Handles admin features.
 -  `blood_manager.h`: Manages blood stocks.
 -  `hospital_manager.h`: Manages hospitals.
 -  `misc.h`: Handles extra functions.
 -  `transaction_manager.h`: Logs transactions.
-  `src/`: This folder has source code files (`.c`) that define how the system works:
 -  `admin_manager.c`: Code for admin functions.
 -  `blood_manager.c`: Code for blood stock functions.
 -  `hospital_manager.c`: Code for hospital functions.
 -  `misc.c`: Code for extra functions.
 -  `transaction_manager.c`: Code for transaction logging.

-  docs/: Contains files made by Doxygen for automatic documentation.
-  report/: Holds the project report:
 -  crimson-care-project-report.tex: A LaTeX file for writing the report.
 -  docs/latex/: Other LaTeX-related files.
-  resources/: Holds extra resources:
 -  db/: Files for database storage.
 -  assets/:
 - *  images/: Images used in the project.
 - *  misc/: Miscellaneous resource files.
 -  cc.txt: Miscellaneous resource file (CrimsonCare ASCII art).

How This Helps

This structure organizes all files and folders for easy use. Every part has its own place, so:

- Developers can quickly find and update code.
- Documentation is clear and accessible.
- Testing and future changes are easier to manage.

This setup ensures the CrimsonCare system stays easy to work with as it grows.

Chapter 3





System Design

3.1 Architecture


The CrimsonCare Blood Management System is designed to be easy to build, change, and fix. It is divided into small parts called modules, where each part has its own job. These parts work together to create the full system. By dividing the system this way, it is simple to test or update one part without breaking the others.

3.1.1 High-Level Architecture


The system is made up of six main parts:

- **Main Module:**
 - **Purpose:** Starts the program, loads data, and shows the main menu for the user.
 -  `main.c`
- **Admin Manager Module:**
 - **Purpose:** Manages admin accounts (add, delete, and update). Protects important functions with admin login.
 -  `admin_manager.c`, `admin_manager.h`
- **Hospital Manager Module:**
 - **Purpose:** Handles hospital information (add, and delete hospital records).
 -  `hospital_manager.c`, `hospital_manager.h`
- **Blood Manager Module:**
 - **Purpose:** Manages blood stock (add, and update blood records).
 -  `blood_manager.c`, `blood_manager.h`

- **Transaction Manager Module:**

- **Purpose:** Logs transactions (add transaction records).
-  `transaction_manager.c, transaction_manager.h`

- **Miscellaneous Functions Module:**

- **Purpose:** Handles small, extra tasks like showing menus, providing secure password input, validating and formatting dates, or clearing the screen.
-  `misc.c, misc.h`

3.1.2 How the Parts Work Together

- **Main Module:**

- Loads data from files.
- Displays the main menu.
- Calls other modules to handle user input.

- **User Menu Features:**

- **Buy Blood:**

- * Checks the hospital code (using Hospital Manager) and blood group/stock (using Blood Manager).
- * Saves the transaction to a file with the current date (using Transaction Manager).

- **Sell Blood:**

- * Checks the blood group.
- * Asks for the donation date and generates a unique token.
- * Saves the transaction to a file with the current date (using Transaction Manager).

- **Display Blood Stocks:**

- * Displays the current blood stock (using Blood Manager).

- **Admin Menu Features:**

- **Add Hospital:**

- * Takes the hospital's name and location.
- * Generates a unique code for the hospital.
- * Saves the hospital record to a `hospitals.txt` file (using Hospital Manager).

- **Update Blood Price or Quantity:**

- * Takes the blood group
- * Validates the blood group.

- * Takes the new price or quantity (while updating the quantity, it checks if the price is 0 or not, if 0 then it first asks for the price).
- * Updates the blood record in the `blood_stock.txt` file (using Blood Manager).
- **Change Admin Password:**
 - * Verifies the current admin.
 - * Takes the new password.
 - * Updates the admin password in the `admin_credentials.dat` file (using Admin Manager).
- **Add/Delete Admin:**
 - * Verifies the current admin.
 - * If Add, Takes the admin's username and password.
 - * If Delete, Takes the admin's username, if the admin is self-admin, it aborts the operation.
 - * Adds or deletes the admin record in the `admin_credentials.dat` file (using Admin Manager).
- **Delete Hospital:**
 - * Verifies the current admin.
 - * Takes the hospital code.
 - * Deletes the hospital record from the `hospitals.txt` file (using Hospital Manager).
- **Show Records:**
 - * Displays the records of hospitals, blood stocks, and transactions (using Hospital Manager, Blood Manager, and Transaction Manager).

3.2 Data Saving and Loading

Each module is responsible for saving its own data to files. For example:

- **Admin Manager:** Saves admin records to `admin_credentials.dat`.
- **Hospital Manager:** Saves hospital records to `hospitals.txt`.
- **Blood Manager:** Saves blood stock records to `blood_stock.txt`.
- **Transaction Manager:** Saves transaction records to `transactions.txt`.

3.3 Why This Design is Good

- **Easy to Change:** You can fix or add features to one part without breaking the others.
- **Reliable:** Each module checks input to prevent errors.

- **Expandable:** New features (like databases) can be added without changing the whole system.

This design makes the CrimsonCare Blood Management System simple, strong, and ready for future updates.

3.4 Data Structures

The CrimsonCare Blood Management System uses **data structures**, to store and manage information. The primary data structure used in this system is the **linked list**. A linked list is a way to store data in a chain-like format, where each piece of data is connected to the next one. This makes it easy to add or remove data as needed.

3.4.1 Linked List

The system uses **linked lists** in many parts to handle data that can grow or shrink. Each module has its own linked list for managing data. Below are the main linked lists used in the system:

Admin Linked List

- **What It Does:** Keeps a list of all admins who can log in to the system.
- **How It Works:**
 - Each piece of the list (called a node) stores the username, password, and a link to the next admin.
 - All nodes are connected in a chain.

Example Code:

```
1     typedef struct Admin {
2         char username[MAX_USERNAME_LENGTH];
3         char password[MAX_PASSWORD_LENGTH];
4         struct Admin* next;
5     } Admin;
6
7     Admin* adminHead = NULL; // The start of the admin list.
```

Listing 3.1: Admin Linked List

Hospital Linked List

- **What It Does:** Keeps a list of all hospitals that use the system.
- **How It Works:**
 - Each node stores the hospital name, location, and a unique code.
 - All nodes are connected in a chain.

Example Code:

```
1  typedef struct Hospital {
2      char name[MAX_NAME_LENGTH];
3      char location[MAX_LOCATION_LENGTH];
4      char code[MAX_CODE_LENGTH];
5      struct Hospital* next;
6  } Hospital;
7
8  Hospital* hospitalHead = NULL; // The start of the
   ↪ hospital list.
```

Listing 3.2: Hospital Linked List

Blood Stock Linked List

- **What It Does:** Keeps a list of all blood stocks.
- **How It Works:**
 - Each node stores the blood group, price, and quantity.
 - All nodes are connected in a chain.

Example Code:

```
1  typedef struct BloodStock {
2      float price;
3      uint32_t id;
4      uint32_t quantity;
5      char bloodGroup[BLOOD_GROUP_NAME_LENGTH];
6      struct BloodStock* next;
7  } BloodStock;
8
9  BloodStock* bloodHead = NULL; // The start of the blood
   ↪ list.
```

Listing 3.3: Blood Stock Linked List

3.4.2 Operation of Linked Lists

Each linked list supports various operations to manage the data efficiently. The following are common operations performed on the linked lists:

Add Data (Insertion)

- **Purpose:** Adds a new node to the linked list.
- **How It Works:**
 - Creates a new node with the desired data.
 - Inserts the new node at the beginning or end of the list.

Example Code (Add Data):

```
1      char* addHospital(const char* name, const char*
2          ↪ location) {
3          // ...
4          Hospital* newHospital =
5              ↪ (Hospital*)malloc(sizeof(Hospital));
6          if (!newHospital) {
7              printf("Error allocating memory for hospital:
8                  ↪ %s\n", strerror(errno));
9              return NULL;
10         }
11         // ...
12         newHospital->next = NULL;
13
14         if (hospitalHead == NULL) {
15             hospitalHead = newHospital;
16         } else {
17             Hospital* temp = hospitalHead;
18             while (temp->next != NULL) {
19                 temp = temp->next;
20             }
21             temp->next = newHospital;
22         }
23         return newHospital->code;
24     }
```

Listing 3.4: Add Hospital

Delete Data (Deletion)

- **Purpose:** Removes a node from the linked list.
- **How It Works:**
 - Searches for the node to delete.
 - Removes the node from the list.

Example Code (Delete Data):

```
1  bool deleteHospital(const char* code) {
2      Hospital* current = hospitalHead;
3      Hospital* prev = NULL;
4      while (current != NULL) {
5          if (strcmp(current->code, code) == 0) {
6              if (prev == NULL) {
7                  hospitalHead = current->next;
8              } else {
9                  prev->next = current->next;
10             }
11             return true;
12         }
13         prev = current;
14         current = current->next;
15     }
16     return false;
17 }
```

Listing 3.5: Delete Hospital

Show Data (Traversal)

- **Purpose:** Displays all data in the linked list.
- **How It Works:**
 - Traverses the list and prints each node's data.

Example Code (Show Data):

```
1  void displayHospitals(void) {
2      Hospital* temp = hospitalHead;
3      if (temp == NULL) {
4          // ...
5          return;
6      }
7  }
```

```
8      // ...
9
10     while (temp != NULL) {
11         // ...
12         temp = temp->next;
13     }
14 }
```

Listing 3.6: Show Hospitals

Find Data (Search)

- **Purpose:** Searches for a node in the linked list.
- **How It Works:**
 - Searches for the node with the given data.

Example Code (Find Data):

```
1     char* getHospitalNameByCode(const char* code) {
2         Hospital* temp = hospitalHead;
3         while (temp != NULL) {
4             if (strcmp(temp->code, code) == 0) {
5                 return temp->name;
6             }
7             temp = temp->next;
8         }
9         return NULL;
10    }
```

Listing 3.7: Find Hospital

Update Data (Modification)

- **Purpose:** Updates a node in the linked list.
- **How It Works:**
 - Searches for the node to update.
 - Updates the node's data.

Example Code (Update Data):

```
1      bool changeAdminPassword(const char* username, const
2      ↪ char* oldPassword, const char* newPassword) {
3          // ...
4          Admin* temp = adminHead;
5          while (temp != NULL) {
6              if (strcmp(username, temp->username) == 0 &&
7              ↪ strcmp(oldPassword, temp->password) == 0) {
8                  strncpy(temp->password, newPassword,
9                  ↪ sizeof(temp->password) - 1);
10                 temp->password[sizeof(temp->password) - 1] =
11                 ↪ '\0';
12                 saveAdminCredentials();
13                 return true;
14             }
15             temp = temp->next;
16         }
17         return false;
18     }
```

Listing 3.8: Update Admin Password

3.4.3 Memory Management

Memory is used when creating new nodes. To prevent problems (like memory leaks), the system **frees memory** before exiting. The following is a common example of how memory is freed:

Example Code (Free Memory):

```
1      void freeAdmin(void) {
2          Admin* current = adminHead;
3          while (current != NULL) {
4              Admin* temp = current;
5              current = current->next;
6              free(temp);
7          }
8          adminHead = NULL;
9      }
```

Listing 3.9: Freeing the Admin List

This ensures that memory is used properly and nothing is wasted.

Why Linked Lists Are Used

- **Flexible:** Data can grow or shrink as needed.


- **Efficient:** Adding or removing data is fast.
- **Simple:** Easy to use and understand.

By using linked lists and good memory management, the CrimsonCare Blood Management System handles its data smoothly and safely.


3.5 Modules

The CrimsonCare Blood Management System is divided into smaller parts, called modules. Each module does specific tasks to make the system work smoothly. Below is an explanation of each module and what it does:

3.5.1 Main Module


- **Purpose:** This module starts the program, loads all the data, and shows the main menu to the user.
-  `main.c`
- **Key Functions:**
 - `main ()`
 - * Starts the program.
 - * Loads blood group data, hospital data, and admin accounts.
 - * Shows the welcome message and user menu.

3.5.2 Miscellaneous Functions Module


- **Purpose:** This module handles small tasks that help the program run better, like showing menus, checking dates, and getting secure input.
-  `misc.c, misc.h`
- **Key Functions:**
 - `displayWelcomeMessage ()`: Shows the welcome message (from the file `cc.txt`).
 - `displayUserMenu ()`: Shows the menu for users.
 - `displayAdminMenu ()`: Shows the menu for admins.
 - `clearInputBuffer ()`: Clears the input buffer to avoid errors.
 - `checkUsername ()`: Checks if a username is valid.
 - `containsPipe ()`: Checks if a string contains a pipe character.
 - `getPassword ()`: Gets a password from the user securely.

- `isLeapYear ()`: Checks if a year is a leap year.
- `isValidDate ()`: Checks if a date is valid.
- `formatDate ()`: Formats a date into the form `yyyy-mm-dd`.

3.5.3 Blood Manager Module


- **Purpose:** This module manages blood stock, like adding, updating, and showing blood records.
-  `blood_manager.c`, `blood_manager.h`
- **Key Functions:**
 - `isValidBloodGroup ()`: Checks if a blood group ID is valid.
 - `addBloodGroup ()`: Adds a new blood group to the system.
 - `initializeBloodGroups ()`: Loads the default blood groups when the system starts.
 - `saveBloodGroups ()`: Saves blood data to the file `blood_data.txt`.
 - `updateBloodQuantity ()`: Changes the amount of blood available for a blood group.
 - `updateBloodPrice ()`: Changes the price of a blood group.
 - `loadBloodGroups ()`: Reads blood data from the file `blood_data.txt`.
 - `isBloodAvailable ()`: Checks if enough blood is available for a request.
 - `displayBloodGroups ()`: Shows all the available blood groups.
 - `displayBloodStocks ()`: Shows the blood stocks in the system.
 - `getBloodGroupById ()`: Finds a blood group's name using its ID.
 - `freeBloodList ()`: Clears all blood data from memory.

3.5.4 Hospital Manager Module


- **Purpose:** This module manages hospital information, like adding, deleting, and showing hospitals.
-  `hospital_manager.c`, `hospital_manager.h`
- **Key Functions:**
 - `loadHospitals ()`: Reads hospital data from the file `hospitals.txt`.
 - `saveHospitals ()`: Saves hospital data to the file `hospitals.txt`.
 - `addHospital ()`: Adds a new hospital to the system.
 - `validateHospitalCode ()`: Checks if a hospital code is valid.
 - `deleteHospital ()`: Removes a hospital from the system using its code.

- `getHospitalNameByCode ()`: Finds a hospital's name using its code.
- `displayHospitals ()`: Shows all hospitals in the system.
- `freeHospital ()`: Clears all hospital data from memory.

3.5.5 Admin Manager Module

- **Purpose:** This module manages admin accounts, like adding, deleting, and updating admin details.
-  `admin_manager.c, admin_manager.h`
- **Key Functions:**
 - `saveAdminCredentials ()`: Saves admin data to the file `admin_credentials.dat`.
 - `loadAdminCredentials ()`: Reads admin data from the file `admin_credentials.dat`.
 - `adminExists ()`: Checks if an admin username already exists.
 - `validateAdmin ()`: Checks if the admin username and password are correct.
 - `addAdmin ()`: Adds a new admin to the system.
 - `deleteAdmin ()`: Removes an admin from the system.
 - `changeAdminPassword ()`: Updates the password for an admin.
 - `displayAdmin ()`: Shows all admins in the system.
 - `freeAdmin ()`: Clears all admin data from memory.

3.5.6 Transaction Manager Module

- **Purpose:** This module keeps track of transactions like blood donations and sales.
-  `transaction_manager.c, transaction_manager.h`
- **Key Functions:**
 - `logTransaction ()`: Saves a transaction to the file `transactions.log`.
 - `addTransaction ()`: Adds a new transaction to the system.
 - `displayTransactions ()`: Shows all transactions from the file `transactions.log`.
 - `freeTransaction ()`: Clears all transaction data from memory.

Chapter 4

Implementation

4.1 Development Environment

The CrimsonCare Blood Management System uses various tools and technologies to ensure its efficient development and operation. These tools help streamline the development process, improve performance, and maintain the system's reliability. Here's an overview of the tools and technologies used:

Integrated Development Environment (IDE)

- **Code::Blocks:** An open-source Integrated Development Environment (IDE) for C/C++ programming. It was used for writing, editing, and debugging the code.

Compiler

- **GCC (GNU Compiler Collection):** The standard compiler for C and C++ used to compile the source code. The project supports both Debug and Release builds.

Build System

- **Make:** A build automation tool used to compile and link the project.

Version Control

- **Git:** A version control system used to manage the source code. The repository is hosted on GitHub.

Documentation

- **Doxygen:** A documentation generator used to create the project documentation.

Text Editor Configuration

- **EditorConfig:** A file format and collection of text editor plugins for maintaining consistent coding styles across different editors and IDEs.

Other Tools

- **LaTeX (MikTeX):** A typesetting system used to generate the project report.

4.2 Function Implementations

The CrimsonCare Blood Management System is divided into smaller parts, called modules. Each module performs specific tasks to help the system function smoothly. Below is a description of the functions implemented in the system.

4.2.1 Admin Manager Module

`saveAdminCredentials ()`

Saves admin data to the file `admin_credentials.dat`.

```
1 void saveAdminCredentials(void) {
2     errno = 0;
3     FILE* file = fopen("resources/db/admin_credentials.dat",
4         ↪ "wb");
5     if (!file) {
6         if (errno != ENOENT) {
7             printf("Error opening admin credentials file:
8                 ↪ %s\n", strerror(errno));
9             return;
10        }
11    }
12
13    Admin* temp = adminHead;
14    while (temp != NULL) {
15        if (fwrite(temp, sizeof(Admin), 1, file)) {
16            temp = temp->next;
17        } else {
```

```
16         printf("Error writing admin credentials: %s\n",
17                ↪ strerror(errno));
18         freeAdmin();
19         fclose(file);
20         return;
21     }
22     fclose(file);
23 }
```

Listing 4.1: saveAdminCredentials ()

loadAdminCredentials ()

Loads admin data from the file admin_credentials.dat.

```
1 void loadAdminCredentials(void) {
2     errno = 0;
3     FILE* file = fopen("resources/db/admin_credentials.dat",
4                        ↪ "rb");
5     if (!file) {
6         if (errno == ENOENT) {
7             Admin* newAdmin = (Admin*)malloc(sizeof(Admin));
8             if (newAdmin) {
9                 strcpy(newAdmin->username, "admin");
10                strcpy(newAdmin->password, "1234");
11                newAdmin->next = NULL;
12                adminHead = newAdmin;
13                saveAdminCredentials();
14            } else {
15                printf("Error allocating memory for admin:
16                       ↪ %s\n", strerror(errno));
17            }
18        } else {
19            printf("Error opening admin credentials file:
20                   ↪ %s\n", strerror(errno));
21        }
22        return;
23    }
24
25    Admin tempAdmin;
26    while (fread(&tempAdmin, sizeof(Admin), 1, file)) {
27        Admin* newAdmin = (Admin*)malloc(sizeof(Admin));
28        if (newAdmin) {
29            *newAdmin = tempAdmin;
30            newAdmin->next = adminHead;
31            adminHead = newAdmin;
32        } else {
```

```
30         printf("Error allocating memory for admin:
31             ↪ %s\n", strerror(errno));
32         freeAdmin();
33         fclose(file);
34         return;
35     }
36     fclose(file);
37 }
```

Listing 4.2: loadAdminCredentials ()

adminExists ()

Checks if an admin username already exists.

```
1 bool adminExists(const char* username) {
2     if (strcmp(username, "") == 0) {
3         printf("Error: Admin username cannot be empty.\n");
4         return false;
5     }
6
7     if (!checkUsername(username)) {
8         printf("Error: Invalid username. Username can only
9             ↪ contain lowercase letters and digits.\n");
10        return false;
11    }
12
13    Admin* temp = adminHead;
14    while (temp != NULL) {
15        if (strcmp(temp->username, username) == 0) {
16            return true;
17        }
18        temp = temp->next;
19    }
20    return false;
21 }
```

Listing 4.3: adminExists ()

validateAdmin ()

Validates admin credentials.

```
1 bool validateAdmin(const char* username, const char*
    ↪ password) {
```

```
2   if (strcmp(username, "") == 0 || strcmp(password, "") ==  
    ↪ 0) {  
3       printf("Error: Admin credentials cannot be  
    ↪ empty.\n");  
4       return false;  
5   }  
6  
7   if (!checkUsername(username)) {  
8       printf("Error: Invalid username. Username can only  
    ↪ contain lowercase letters and digits.\n");  
9       return false;  
10  }  
11  
12  Admin* temp = adminHead;  
13  while (temp != NULL) {  
14      if (strcmp(username, temp->username) == 0 &&  
    ↪ strcmp(password, temp->password) == 0) {  
15          return true;  
16      }  
17      temp = temp->next;  
18  }  
19  return false;  
20 }
```

Listing 4.4: validateAdmin ()

addAdmin ()

Adds a new admin to the system.

```
1  bool addAdmin(const char* username, const char* password,  
    ↪ const char* currentAdminUsername, const char*  
    ↪ currentAdminPassword) {  
2      if (strcmp(currentAdminUsername, "") == 0 ||  
    ↪ strcmp(currentAdminPassword, "") == 0) {  
3          printf("Error: Current admin credentials cannot be  
    ↪ empty.\n");  
4          return false;  
5      }  
6  
7      if (!checkUsername(currentAdminUsername) ||  
    ↪ !checkUsername(username)) {  
8          printf("Error: Invalid username. Username can only  
    ↪ contain lowercase letters and digits.\n");  
9          return false;  
10     }  
11 }
```



```

12     if (!validateAdmin(currentAdminUsername ,
13         ↪ currentAdminPassword)) {
14         printf("Error: Invalid current admin
15             ↪ credentials.\n");
16         return false;
17     }
18
19     if (adminExists(username)) {
20         printf("Error: Admin already exists.\n");
21         return false;
22     }
23
24     if (strcmp(username, "") == 0 || strcmp(password, "") ==
25         ↪ 0) {
26         printf("Error: Admin credentials cannot be
27             ↪ empty.\n");
28         return false;
29     }
30
31     Admin* newAdmin = (Admin*)malloc(sizeof(Admin));
32     if (!newAdmin) {
33         printf("Error allocating memory for admin: %s\n",
34             ↪ strerror(errno));
35         return false;
36     }
37     strncpy(newAdmin->username, username,
38         ↪ sizeof(newAdmin->username) - 1);
39     newAdmin->username[sizeof(newAdmin->username) - 1] =
40         ↪ '\0';
41     strncpy(newAdmin->password, password,
42         ↪ sizeof(newAdmin->password) - 1);
43     newAdmin->password[sizeof(newAdmin->password) - 1] =
44         ↪ '\0';
45     newAdmin->next = adminHead;
46     adminHead = newAdmin;
47
48     saveAdminCredentials();
49     return true;
50 }

```

Listing 4.5: addAdmin ()

deleteAdmin ()

Deletes an admin from the system.

```

1 bool deleteAdmin(const char* username, const char*
    ↪ currentAdminUsername, const char*

```

```
↪ currentAdminPassword) {
2   if (strcmp(currentAdminUsername, "") == 0 ||
    ↪ strcmp(currentAdminPassword, "") == 0) {
3       printf("Error: Current admin credentials cannot be
        ↪ empty.\n");
4       return false;
5   }
6
7   if (!checkUsername(currentAdminUsername) ||
    ↪ !checkUsername(username)) {
8       printf("Error: Invalid username. Username can only
        ↪ contain lowercase letters and digits.\n");
9       return false;
10  }
11
12  if (!validateAdmin(currentAdminUsername,
    ↪ currentAdminPassword)) {
13      printf("Error: Invalid current admin
        ↪ credentials.\n");
14      return false;
15  }
16
17  if (!adminExists(username)) {
18      printf("Error: Admin does not exist.\n");
19      return false;
20  }
21
22  if (strcmp(username, "") == 0) {
23      printf("Error: Admin username cannot be empty.\n");
24      return false;
25  }
26
27  if (strcmp(username, currentAdminUsername) == 0) {
28      printf("Error: Cannot delete current admin.\n");
29      return false;
30  }
31
32  Admin* temp = adminHead;
33  Admin* prev = NULL;
34
35  while (temp != NULL) {
36      if (strcmp(temp->username, username) == 0) {
37          if (prev == NULL) {
38              adminHead = temp->next;
39          } else {
40              prev->next = temp->next;
41          }
42          free(temp);
```

```

43         saveAdminCredentials();
44         return true;
45     }
46     prev = temp;
47     temp = temp->next;
48 }
49 return false;
50 }

```

Listing 4.6: deleteAdmin ()

changeAdminPassword ()

Changes an admin's password.

```

1  bool changeAdminPassword(const char* username, const char*
    ↪ oldPassword, const char* newPassword) {
2      if (strcmp(username, "") == 0 || strcmp(oldPassword, "")
    ↪ == 0) {
3          printf("Error: Username or old password cannot be
    ↪ empty.\n");
4          return false;
5      }
6
7      if (!checkUsername(username)) {
8          printf("Error: Invalid username. Username can only
    ↪ contain lowercase letters and digits.\n");
9          return false;
10     }
11
12     if (!validateAdmin(username, oldPassword)) {
13         printf("Error: Invalid password.\n");
14         return false;
15     }
16
17     if (strcmp(newPassword, "") == 0) {
18         printf("Error: New password cannot be empty.\n");
19         return false;
20     }
21
22     Admin* temp = adminHead;
23     while (temp != NULL) {
24         if (strcmp(username, temp->username) == 0 &&
    ↪ strcmp(oldPassword, temp->password) == 0) {
25             strncpy(temp->password, newPassword,
    ↪ sizeof(temp->password) - 1);
26             temp->password[sizeof(temp->password) - 1] =
    ↪ '\0';

```

```
27         saveAdminCredentials();
28         return true;
29     }
30     temp = temp->next;
31 }
32 return false;
33 }
```

Listing 4.7: changeAdminPassword ()

displayAdmin ()

Displays all admins.

```
1 void displayAdmin(void) {
2     Admin* temp = adminHead;
3     printf("\nRegistered Admins:\n");
4     while (temp != NULL) {
5         printf("\tUsername: %s\n", temp->username);
6         temp = temp->next;
7         if (temp != NULL) {
8             printf("\t-----\n");
9         }
10    }
11 }
```

Listing 4.8: displayAdmin ()

freeAdmin ()

Frees all admin data from memory.

```
1 void freeAdmin(void) {
2     Admin* current = adminHead;
3     while (current != NULL) {
4         Admin* temp = current;
5         current = current->next;
6         free(temp);
7     }
8     adminHead = NULL;
9 }
```

Listing 4.9: freeAdmin ()

4.2.2 Blood Manager Module

isValidBloodGroup ()

Checks if blood group is valid.

```
1 bool isValidBloodGroup(uint32_t id) {
2     return id <= (sizeof(availableBloodGroups) /
3         ↪ sizeof(availableBloodGroups[0]));
4 }
```

Listing 4.10: isValidBloodGroup ()

addBloodGroup ()

Adds a new blood group to the system.

```
1 bool addBloodGroup(uint32_t id, const char* bloodGroup,
2     ↪ float price, uint32_t quantity) {
3     if (strcmp(bloodGroup, "") == 0) {
4         printf("Error: Invalid blood group data.\n");
5         return false;
6     }
7     if (!isValidBloodGroup(id)) {
8         printf("Error: Invalid blood group id.\n");
9         return false;
10    }
11
12    BloodStock* newGroup =
13        ↪ (BloodStock*)malloc(sizeof(BloodStock));
14    if (!newGroup) {
15        printf("Error allocating memory for blood group:
16            ↪ %s\n", strerror(errno));
17        return false;
18    }
19    strncpy(newGroup->bloodGroup, bloodGroup,
20        ↪ BLOOD_GROUP_NAME_LENGTH - 1);
21    newGroup->bloodGroup[BLOOD_GROUP_NAME_LENGTH - 1] = '\0';
22    newGroup->price = price;
23    newGroup->quantity = quantity;
24    newGroup->id = id;
25    newGroup->next = NULL;
26
27    if (bloodHead == NULL) {
28        bloodHead = newGroup;
29    } else {
30        newGroup->next = bloodHead;
31        bloodHead = newGroup;
32    }
33    return true;
34 }
```

```
27     BloodStock* temp = bloodHead;
28     while (temp->next != NULL) {
29         temp = temp->next;
30     }
31     temp->next = newGroup;
32 }
33 return true;
34 }
```

Listing 4.11: addBloodGroup ()

initializeBloodGroups ()

Initializes blood groups available.

```
1 void initializeBloodGroups(void) {
2     for (uint8_t i = 0; i < (sizeof(availableBloodGroups) /
3         ↪ sizeof(availableBloodGroups[0])); i++) {
4         if (!addBloodGroup(i + 1, availableBloodGroups[i],
5             ↪ 0.0, 0)) {
6             printf("Error: Failed to initialize blood group
7             ↪ %s.\n", availableBloodGroups[i]);
8         }
9     }
10 }
```

Listing 4.12: initializeBloodGroups ()

saveBloodGroups ()

Saves blood groups to the file blood_data.txt.

```
1 void saveBloodGroups(void) {
2     errno = 0;
3     FILE* file = fopen("resources/db/blood_data.txt", "w");
4     if (!file) {
5         if (errno != ENOENT) {
6             printf("Error opening blood data file: %s\n",
7                 ↪ strerror(errno));
8             return;
9         }
10    }
11
12    BloodStock* temp = bloodHead;
13    while (temp != NULL) {
14        fprintf(file, "%u %s %.2f %u\n", temp->id,
15            ↪ temp->bloodGroup, temp->price, temp->quantity);
16    }
17 }
```

```
14     temp = temp->next;
15 }
16 fclose(file);
17 }
```

Listing 4.13: saveBloodGroups ()

updateBloodQuantity ()

Updates the blood quantity of a blood group.

```
1 bool updateBloodQuantity(uint32_t id, uint32_t newQuantity) {
2     if (!isValidBloodGroup(id)) {
3         printf("Error: Invalid blood group id.\n");
4         return false;
5     }
6
7     BloodStock* temp = bloodHead;
8     while (temp != NULL) {
9         if (temp->id == id) {
10             temp->quantity = newQuantity;
11             saveBloodGroups();
12             return true;
13         }
14         temp = temp->next;
15     }
16     return false;
17 }
```

Listing 4.14: updateBloodQuantity ()

updateBloodPrice ()

Updates the blood price of a blood group.

```
1 bool updateBloodPrice(uint32_t id, float newPrice) {
2     if (!isValidBloodGroup(id)) {
3         printf("Error: Invalid blood group id.\n");
4         return false;
5     }
6
7     BloodStock* temp = bloodHead;
8     while (temp != NULL) {
9         if (temp->id == id) {
10             temp->price = newPrice;
11             saveBloodGroups();
12             return true;
13         }
14     }
15 }
```

```
13     }
14     temp = temp->next;
15 }
16 return false;
17 }
```

Listing 4.15: updateBloodPrice ()

loadBloodGroups ()

Loads blood groups from the file blood_data.txt.

```
1 void loadBloodGroups(void) {
2     errno = 0;
3     FILE* file = fopen("resources/db/blood_data.txt", "r");
4     if (!file) {
5         if (errno == ENOENT) {
6             initializeBloodGroups();
7             return;
8         } else {
9             printf("Error opening blood data file: %s\n",
10                  ↪ strerror(errno));
11             freeBloodList();
12             return;
13         }
14     }
15
16     while (1) {
17         BloodStock* newBlood =
18             ↪ (BloodStock*)malloc(sizeof(BloodStock));
19         if (!newBlood) {
20             printf("Error allocating memory for blood group:
21                  ↪ %s\n", strerror(errno));
22             freeBloodList();
23             fclose(file);
24             return;
25         }
26
27         if (fscanf(file, "%u %s %f %u", &newBlood->id,
28             ↪ newBlood->bloodGroup, &newBlood->price,
29             ↪ &newBlood->quantity) != 4) {
30             free(newBlood);
31             fclose(file);
32             break;
33         }
34
35         newBlood->next = NULL;
```



```

32     if (bloodHead == NULL) {
33         bloodHead = newBlood;
34     } else {
35         BloodStock* temp = bloodHead;
36         while (temp->next != NULL) {
37             temp = temp->next;
38         }
39         temp->next = newBlood;
40     }
41 }
42
43 fclose(file);
44 }

```

Listing 4.16: loadBloodGroups ()

isBloodAvailable ()

Checks if blood is available.

```

1  bool isBloodAvailable(uint32_t* id, TransactionType type) {
2      if (type != BUY && type != SELL) {
3          printf("Error: Invalid transaction type.\n");
4          return false;
5      }
6
7      if (id != NULL && !isValidBloodGroup(*id)) {
8          printf("Error: Invalid blood group id.\n");
9          return false;
10     }
11
12     BloodStock* temp = bloodHead;
13     while (temp != NULL) {
14         if (type == BUY) {
15             if (id == NULL) {
16                 if (temp->price > 0 && temp->quantity > 0) {
17                     return true;
18                 }
19             } else {
20                 if (temp->id == *id && temp->price > 0 &&
21                     ↪ temp->quantity > 0) {
22                     return true;
23                 }
24             }
25         } else {
26             if (id == NULL) {
27                 if (temp->price > 0) {

```

```
28         }
29     } else {
30         if (temp->id == *id && temp->price > 0) {
31             return true;
32         }
33     }
34 }
35 temp = temp->next;
36 }
37 return false;
38 }
```

Listing 4.17: isBloodAvailable ()

displayBloodGroups ()

Displays all blood groups.

```
1 void displayBloodGroups(void) {
2     for (uint32_t i = 0; i < (sizeof(availableBloodGroups) /
3         ↪ sizeof(availableBloodGroups[0])); i++) {
4         printf("%u. %s\n", i + 1, availableBloodGroups[i]);
5     }
6 }
```

Listing 4.18: displayBloodGroups ()

displayBloodStocks ()

Displays all blood stocks.

```
1 void displayBloodStocks(void) {
2     BloodStock* temp = bloodHead;
3     if (temp == NULL) {
4         printf("No blood available.\n");
5         return;
6     }
7     printf("\nAvailable Blood:\n");
8     while (temp != NULL) {
9         if (temp->price > 0.0) {
10             printf("%u. %s, Price: %.2f, Quantity: %u\n",
11                 ↪ temp->id, temp->bloodGroup, temp->price,
12                 ↪ temp->quantity);
13         } else {
14             printf("%u. %s, Price: N/A, Quantity: N/A\n",
15                 ↪ temp->id, temp->bloodGroup);
16         }
17     }
18 }
```

```
14         temp = temp->next;
15     }
16 }
```

Listing 4.19: displayBloodStocks ()

getBloodGroupById ()

Gets blood group by id.

```
1 char* getBloodGroupById(uint32_t id) {
2     if (!isValidBloodGroup(id)) {
3         printf("Error: Invalid blood group id.\n");
4         return NULL;
5     }
6
7     return availableBloodGroups[id - 1];
8 }
```

Listing 4.20: getBloodGroupById ()

freeBloodList ()

Frees the blood list.

```
1 void freeBloodList(void) {
2     BloodStock* current = bloodHead;
3     while (current != NULL) {
4         BloodStock* temp = current;
5         current = current->next;
6         free(temp);
7     }
8     bloodHead = NULL;
9 }
```

Listing 4.21: freeBloodList ()

4.2.3 Hospital Manager Module

loadHospitals ()

Loads hospitals from the file hospitals.txt.

```
1 void loadHospitals(void) {
2     errno = 0;
```

```
3 FILE* file = fopen("resources/db/hospitals.txt", "r");
4 if (!file) {
5     if (errno == ENOENT) {
6         return;
7     } else {
8         printf("Error opening hospitals file: %s\n",
9             ↪ strerror(errno));
10        freeHospital();
11        return;
12    }
13
14 while (1) {
15     Hospital* newHospital =
16         ↪ (Hospital*)malloc(sizeof(Hospital));
17     if (!newHospital) {
18         printf("Error allocating memory for hospital:
19             ↪ %s\n", strerror(errno));
20         fclose(file);
21         freeHospital();
22         return;
23     }
24
25     if (fscanf(file, "[%^|]|[%^|]|[%^\n]\n",
26         ↪ newHospital->code, newHospital->name,
27         ↪ newHospital->location) != 3) {
28         free(newHospital);
29         fclose(file);
30         break;
31     }
32
33     newHospital->next = NULL;
34
35     if (hospitalHead == NULL) {
36         hospitalHead = newHospital;
37     } else {
38         Hospital* temp = hospitalHead;
39         while (temp->next != NULL) {
40             temp = temp->next;
41         }
42         temp->next = newHospital;
43     }
44
45     fclose(file);
46 }
```

Listing 4.22: loadHospitals ()

saveHospitals ()

Saves hospitals to the file hospitals.txt.

```
1 void saveHospitals(void) {
2     errno = 0;
3     FILE* file = fopen("resources/db/hospitals.txt", "w");
4     if (!file) {
5         printf("Error opening hospitals file: %s\n",
6             ↪ strerror(errno));
7         return;
8     }
9     Hospital* temp = hospitalHead;
10    while (temp != NULL) {
11        fprintf(file, "%s|%s|%s\n", temp->code, temp->name,
12            ↪ temp->location);
13        temp = temp->next;
14    }
15    fclose(file);
16 }
```

Listing 4.23: saveHospitals ()

addHospital ()

Adds a new hospital to the system.

```
1 char* addHospital(const char* name, const char* location) {
2     if (strcmp(name, "") == 0 || strcmp(location, "") == 0) {
3         printf("Error: Hospital name or location cannot be
4             ↪ empty.\n");
5         return NULL;
6     }
7     if (containsPipe(name) || containsPipe(location)) {
8         printf("Error: Hospital name or location cannot
9             ↪ contain a pipe character.\n");
10        return NULL;
11    }
12    char code[8];
13    char initials[4] = { 0 };
14    int initialCount = 0;
15
16    char nameCopy[100];
17    strncpy(nameCopy, name, sizeof(nameCopy) - 1);
18    nameCopy[sizeof(nameCopy) - 1] = '\0';
```

```
19
20     char* token = strtok(nameCopy, " ");
21     while (token != NULL && initialCount < 3) {
22         initials[initialCount++] = token[0];
23         token = strtok(NULL, " ");
24     }
25     initials[initialCount] = '\0';
26
27     bool codeExists;
28     int randomSuffix;
29     do {
30         srand(time(NULL));
31         randomSuffix = rand() % 10000;
32         snprintf(code, sizeof(code), "%s%04d", initials,
33             ↪ randomSuffix);
34
35         codeExists = false;
36         Hospital* temp = hospitalHead;
37         while (temp != NULL) {
38             if (strcmp(temp->code, code) == 0) {
39                 codeExists = true;
40                 break;
41             }
42             temp = temp->next;
43         }
44     } while (codeExists);
45
46     Hospital* newHospital =
47         ↪ (Hospital*)malloc(sizeof(Hospital));
48     if (!newHospital) {
49         printf("Error allocating memory for hospital: %s\n",
50             ↪ strerror(errno));
51         return NULL;
52     }
53
54     strncpy(newHospital->code, code,
55         ↪ sizeof(newHospital->code) - 1);
56     newHospital->code[sizeof(newHospital->code) - 1] = '\0';
57     strncpy(newHospital->name, name,
58         ↪ sizeof(newHospital->name) - 1);
59     newHospital->name[sizeof(newHospital->name) - 1] = '\0';
60     strncpy(newHospital->location, location,
61         ↪ sizeof(newHospital->location) - 1);
62     newHospital->location[sizeof(newHospital->location) - 1]
63         ↪ = '\0';
64     newHospital->next = NULL;
65
66     if (hospitalHead == NULL) {
```

```
60     hospitalHead = newHospital;
61 } else {
62     Hospital* temp = hospitalHead;
63     while (temp->next != NULL) {
64         temp = temp->next;
65     }
66     temp->next = newHospital;
67 }
68
69 saveHospitals();
70 return newHospital->code;
71 }
```

Listing 4.24: addHospital ()

validateHospitalCode ()

Validates hospital code.

```
1 bool validateHospitalCode(const char* code) {
2     if (strcmp(code, "") == 0) {
3         printf("Error: Hospital code cannot be empty.\n");
4         return false;
5     }
6
7     if (containsPipe(code)) {
8         printf("Error: Hospital code cannot contain a pipe
9             ↪ character.\n");
10        return false;
11    }
12
13    Hospital* temp = hospitalHead;
14    while (temp != NULL) {
15        if (strcmp(temp->code, code) == 0) {
16            return true;
17        }
18        temp = temp->next;
19    }
20    return false;
21 }
```

Listing 4.25: validateHospitalCode ()

deleteHospital ()

Deletes a hospital from the system.

```
1 bool deleteHospital(const char* code, const char*
  ↪ adminUsername, const char* adminPassword) {
2     if (strcmp(adminUsername, "") == 0 ||
  ↪ strcmp(adminPassword, "") == 0) {
3         printf("Error: Admin credentials cannot be
  ↪ empty.\n");
4         return false;
5     }
6
7     if (!checkUsername(adminUsername)) {
8         printf("Error: Invalid admin username. Username can
  ↪ only contain lowercase letters and digits.\n");
9         return false;
10    }
11
12    if (!validateAdmin(adminUsername, adminPassword)) {
13        printf("Error: Invalid admin credentials.\n");
14        return false;
15    }
16
17    if (strcmp(code, "") == 0) {
18        printf("Error: Hospital code cannot be empty.\n");
19        return false;
20    }
21
22    if (containsPipe(code)) {
23        printf("Error: Hospital code cannot contain a pipe
  ↪ character.\n");
24        return false;
25    }
26
27    if (!validateHospitalCode(code)) {
28        printf("Error: Hospital code is invalid.\n");
29        return false;
30    }
31
32    Hospital* current = hospitalHead;
33    Hospital* prev = NULL;
34    while (current != NULL) {
35        if (strcmp(current->code, code) == 0) {
36            if (prev == NULL) {
37                hospitalHead = current->next;
38            } else {
39                prev->next = current->next;
40            }
41            saveHospitals();
42            return true;
```



```
43     }
44     prev = current;
45     current = current->next;
46 }
47 return false;
48 }
```

Listing 4.26: deleteHospital ()

getHospitalNameByCode ()

Gets hospital name by code.

```
1 char* getHospitalNameByCode(const char* code) {
2     if (strcmp(code, "") == 0) {
3         printf("Error: Hospital code cannot be empty.\n");
4         return NULL;
5     }
6
7     if (containsPipe(code)) {
8         printf("Error: Hospital code cannot contain a pipe
9             ↪ character.\n");
10        return NULL;
11    }
12
13    if (!validateHospitalCode(code)) {
14        printf("Error: Hospital code is invalid.\n");
15        return NULL;
16    }
17
18    Hospital* temp = hospitalHead;
19    while (temp != NULL) {
20        if (strcmp(temp->code, code) == 0) {
21            return temp->name;
22        }
23        temp = temp->next;
24    }
25    return NULL;
26 }
```

Listing 4.27: getHospitalNameByCode ()

displayHospitals ()

Displays all hospitals.

```

1 void displayHospitals(void) {
2     Hospital* temp = hospitalHead;
3     if (temp == NULL) {
4         printf("No hospitals registered yet.\n");
5         return;
6     }
7     printf("\nRegistered Hospitals:\n");
8     while (temp != NULL) {
9         printf("\tCode: %s\n"
10              "\tName: %s\n"
11              "\tLocation: %s\n", temp->code, temp->name,
12              ↪ temp->location);
13         temp = temp->next;
14         if (temp != NULL) {
15             printf("\t-----\n");
16         }
17     }
18 }

```

Listing 4.28: displayHospitals ()

freeHospital ()

Frees the hospital list.

```

1 void freeHospital(void) {
2     Hospital* current = hospitalHead;
3     while (current != NULL) {
4         Hospital* temp = current;
5         current = current->next;
6         free(temp);
7     }
8     hospitalHead = NULL;
9 }

```

Listing 4.29: freeHospital ()

4.2.4 Transaction Manager Module

logTransaction ()

Logs a transaction.

```

1 bool logTransaction(TransactionType type, const char* name,
2     ↪ uint32_t bloodId, uint32_t quantity, const char* date,
3     ↪ const char* token) {

```

```
2     errno = 0;
3     FILE* file = fopen("resources/db/transactions.log", "a");
4     if (!file) {
5         if (errno != ENOENT) {
6             printf("Error opening transaction log file:
7                 ↪ %s\n", strerror(errno));
8         }
9         return false;
10    }
11
12    if (containsPipe(name)) {
13        printf("Error: Entity name cannot contain a pipe
14            ↪ character.\n");
15        return false;
16    }
17
18    if (type != BUY && type != SELL) {
19        printf("Error: Invalid transaction type.\n");
20        return false;
21    }
22
23    if (!isValidBloodGroup(bloodId)) {
24        printf("Error: Invalid blood group.\n");
25        return false;
26    }
27
28    if (strcmp(name, "") == 0 || quantity <= 0) {
29        printf("Error: Invalid transaction parameters.\n");
30        return false;
31    }
32
33    if (!isValidDate(date)) {
34        printf("Error: Invalid date format.\n");
35        return false;
36    }
37
38    if (token) {
39        fprintf(file, "%s|%s|%u|%u|%s|%s\n", (type == BUY ?
40            ↪ "Buy" : "Sell"), name, bloodId, quantity,
41            ↪ date, token);
42    } else {
43        fprintf(file, "%s|%s|%u|%u|%s\n", (type == BUY ?
44            ↪ "Buy" : "Sell"), name, bloodId, quantity,
45            ↪ date);
46    }
47
48    fclose(file);
49    return true;
```

44 }

Listing 4.30: logTransaction ()

addTransaction ()

Adds a transaction to the system.

```
1 bool addTransaction(TransactionType type, const char* name,
2   ↪ uint32_t bloodId, uint32_t quantity) {
3     if (strcmp(name, "") == 0 || quantity <= 0) {
4         printf("Error: Invalid transaction parameters.\n");
5         return false;
6     }
7
8     if (containsPipe(name)) {
9         printf("Error: Entity name cannot contain a pipe
10        ↪ character.\n");
11        return false;
12    }
13
14    if (type != BUY && type != SELL) {
15        printf("Error: Invalid transaction type.\n");
16        return false;
17    }
18
19    if (!isBloodAvailable(&bloodId, type)) {
20        printf("No stock available for blood group: %s\n",
21        ↪ getBloodGroupId(bloodId));
22        return false;
23    }
24
25    if (type == BUY) {
26        if (!validateHospitalCode(name)) {
27            printf("Error: Invalid hospital code.\n");
28            return false;
29        }
30    }
31
32    char date[MAX_TRANSACTION_DATE_LENGTH];
33    char token[MAX_TRANSACTION_TOKEN_LENGTH] = "";
34
35    if (type == SELL) {
36        printf("Enter the date and time of donation
37        ↪ (YYYY-MM-DD): ");
38        fgets(date, sizeof(date), stdin);
39        date[strcspn(date, "\n")] = 0;
40        if (!isValidDate(date)) {
```

```
37         printf("Error: Invalid date format.\n");
38         return false;
39     }
40     formatDate(date);
41 } else {
42     BloodStock* stock = bloodHead;
43     while (stock != NULL) {
44         if (stock->id == bloodId) {
45             if (stock->quantity < quantity) {
46                 printf("Not enough stock for blood
47                     ↪ group: %s. Available quantity:
48                     ↪ %u\n", getBloodGroupById(bloodId),
49                     ↪ stock->quantity);
50                 return false;
51             }
52             stock->quantity -= quantity;
53             saveBloodGroups();
54             break;
55         }
56         stock = stock->next;
57     }
58     time_t now = time(NULL);
59     strftime(date, sizeof(date), "%Y-%m-%d",
60         ↪ localtime(&now));
61 }
62
63 if (type == SELL) {
64     srand(time(NULL));
65     sprintf(token, "TOKEN_%d", rand() % 10000);
66     printf("Sell token generated for %s: %s\n", name,
67         ↪ token);
68 }
69
70 if (!logTransaction(type, name, bloodId, quantity, date,
71     ↪ type == SELL ? token : NULL)) {
72     return false;
73 }
74
75 return true;
76 }
```

Listing 4.31: addTransaction ()

displayTransactions ()

Displays all transactions.

```
1 void displayTransactions(void) {
2     errno = 0;
3     FILE* file = fopen("resources/db/transactions.log", "r");
4     if (!file) {
5         if (errno == ENOENT) {
6             printf("No registered transactions found.\n");
7         } else {
8             printf("Error opening transaction log file:
9                 ↪ %s\n", strerror(errno));
10        }
11        return;
12    }
13
14    char line[256];
15    bool hasLogs = false;
16    bool firstLog = true;
17    char prevLine[256] = { 0 };
18
19    while (fgets(line, sizeof(line), file) != NULL) {
20        char type[MAX_TRANSACTION_TOKEN_LENGTH] = "";
21        char name[MAX_TRANSACTION_TOKEN_LENGTH] = "";
22        uint32_t bloodId = 0;
23        uint32_t quantity = 0;
24        char date[MAX_TRANSACTION_TOKEN_LENGTH] = "";
25        char token[MAX_TRANSACTION_TOKEN_LENGTH] = "";
26
27        if (firstLog) {
28            printf("\nRegistered Transactions:\n");
29            firstLog = false;
30        }
31
32        if (sscanf(line, "%[^|]|%[^|]|%u|%u|%[^|]|%[^|\n]",
33            type,
34            name,
35            &bloodId,
36            &quantity,
37            date,
38            token) >= 5) {
39
40            hasLogs = true;
41
42            if (prevLine[0] != '\0') {
43                printf("\t-----\n");
44            }
45
46            printf("\tType: %s\n"
47                "\tEntity: %s\n"
```

```
47         "\tBlood Group: %s\n"
48         "\tQuantity: %u\n"
49         "\tDate: %s",
50         type,
51         name,
52         getBloodGroupById(bloodId),
53         quantity,
54         date);
55
56         if (token[0] != '\\0') {
57             printf("\n\tToken: %s\n", token);
58         }
59
60         strncpy(prevLine, line, sizeof(prevLine) - 1);
61         prevLine[sizeof(prevLine) - 1] = '\\0';
62     }
63 }
64
65 if (!hasLogs) {
66     printf("No registered transactions found.\n");
67 }
68
69 fclose(file);
70 }
```

Listing 4.32: displayTransactions ()

4.2.5 Miscellaneous Functions Module

displayWelcomeMessage ()

Displays the welcome message.

```
1 void displayWelcomeMessage(void) {
2     FILE* file = fopen("resources/assets/misc/cc.txt", "r");
3     if (!file) {
4         return;
5     }
6     char buffer[1024];
7     while (fgets(buffer, sizeof(buffer), file) != NULL) {
8         printf("%s", buffer);
9     }
10    fclose(file);
11 }
```

Listing 4.33: displayWelcomeMessage ()

displayUserMenu ()

Displays the user menu.

```
1 void displayUserMenu(void) {
2     printf("\n--- CrimsonCare Blood Bank Management System
      ↪ (User) ---\n");
3     printf("1. Buy Blood\n");
4     printf("2. Sell Blood\n");
5     printf("3. Display Blood Stocks\n");
6     printf("4. Admin Panel\n");
7     printf("5. Exit\n");
8     printf("Select an option: ");
9 }
```

Listing 4.34: displayUserMenu ()

displayAdminMenu ()

Displays the admin menu.

```
1 void displayAdminMenu(void) {
2     printf("\n--- CrimsonCare Blood Bank Management System
      ↪ (Admin) ---\n");
3     printf("1. Add Hospital\n");
4     printf("2. Update Blood Quantity\n");
5     printf("3. Update Blood Price\n");
6     printf("4. Change Admin Password\n");
7     printf("5. Add Admin\n");
8     printf("6. Delete Admin\n");
9     printf("7. Delete Hospital\n");
10    printf("8. Display Admins\n");
11    printf("9. Display Hospitals\n");
12    printf("10. Display Blood Stocks\n");
13    printf("11. Display Transactions\n");
14    printf("12. Exit\n");
15    printf("Select an option: ");
16 }
```

Listing 4.35: displayAdminMenu ()

clearInputBuffer ()

Clears the input buffer.

```
1 void clearInputBuffer(void) {
2     int c;
```



```
3     while ((c = getchar()) != '\n' && c != EOF);  
4 }
```

Listing 4.36: clearInputBuffer ()

checkUsername ()

Checks if a username is valid.

```
1 bool checkUsername(const char* str) {  
2     while (*str) {  
3         if (!(*str >= 'a' && *str <= 'z') && !(*str >= '0'  
4             ↪ && *str <= '9')) {  
5             return false;  
6         }  
7         str++;  
8     }  
9     return true;  
}
```

Listing 4.37: checkUsername ()

containsPipe ()

Checks if a string contains a pipe character.

```
1 bool containsPipe(const char* str) {  
2     while (*str) {  
3         if (*str == '|') {  
4             return true;  
5         }  
6         str++;  
7     }  
8     return false;  
9 }
```

Listing 4.38: containsPipe ()

getPassword ()

Gets the password from the user.

```
1 void getPassword(char* password, size_t size) {  
2     #ifdef _WIN32  
3         size_t i = 0;  
4         char ch;
```

```
5  while (i < size - 1) {
6
7      ch = getch();
8
9      if (ch == '\r') {
10
11          break;
12      } else if (ch == '\b') {
13
14          if (i > 0) {
15              i--;
16              printf("\b \b");
17
18          }
19      } else {
20          password[i++] = ch;
21          printf("*");
22
23      }
24  }
25  password[i] = '\0';
26
27  printf("\n");
28  #else
29
30  struct termios oldt, newt;
31  tcgetattr(STDIN_FILENO, &oldt);
32
33  newt = oldt;
34
35  newt.c_lflag &= ~(ECHO);
36
37  tcsetattr(STDIN_FILENO, TCSANOW, &newt);
38
39  fgets(password, size);
40  password[strcspn(password, "\n")] = 0;
41
42  tcsetattr(STDIN_FILENO, TCSANOW, &oldt);
43
44  printf("\n");
45  #endif
46  }
```

Listing 4.39: getPassword ()

isLeapYear ()

Checks if a year is a leap year.

```
1 bool isLeapYear(int year) {
2     return (year % 4 == 0 && year % 100 != 0) || (year % 400
3         ↪ == 0);
4 }
```

Listing 4.40: isLeapYear ()

isValidDate ()

Checks if a date is valid.

```
1 bool isValidDate(const char* date) {
2     if (strcmp(date, "") == 0) {
3         printf("Error: Date cannot be empty.\n");
4         return false;
5     }
6
7     int year, month, day;
8
9     if (strlen(date) < 8 || strlen(date) > 10) {
10        printf("Error: Invalid date format.\n");
11        return false;
12    }
13
14    if (sscanf(date, "%d-%d-%d", &year, &month, &day) != 3) {
15        printf("Error: Invalid date format.\n");
16        return false;
17    }
18
19    if (month < 1 || month > 12) {
20        printf("Error: Invalid month.\n");
21        return false;
22    }
23
24    int daysInMonth[] = { 31, 28 + (int)isLeapYear(year),
25        ↪ 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };
26
27    if (day < 1 || day > daysInMonth[month - 1]) {
28        printf("Error: Invalid day.\n");
29        return false;
30    }
31
32    return true;
33 }
```

Listing 4.41: isValidDate ()

formatDate ()

Formats a date to the format yyyy-mm-dd.

```
1 void formatDate(char* date) {
2     if (strcmp(date, "") == 0) {
3         printf("Error: Date cannot be empty.\n");
4         return;
5     }
6
7     if (!isValidDate(date)) {
8         printf("Error: Invalid date format.\n");
9         return;
10    }
11
12    int year, month, day;
13    sscanf(date, "%d-%d-%d", &year, &month, &day);
14    sprintf(date, "%04d-%02d-%02d", year, month, day);
15 }
```

Listing 4.42: formatDate ()

4.3 Input Validation

Input validation is a crucial part of the CrimsonCare Blood Management System to ensure that all data entered into the system is correct and safe. The following mechanisms are in place to validate inputs:

- **Username Validation:** The function `checkUsername ()` ensures that usernames only contain lowercase letters and digits. If an invalid username is detected, an error message is displayed.
- **Blood Group Validation:** The function `isValidBloodGroup ()` checks if the provided blood group ID is valid by comparing it against the available blood groups.
- **Hospital Code Validation:** The function `validateHospitalCode ()` ensures that hospital codes are valid by traversing the linked list of hospitals.
- **Admin Credentials Validation:** The function `validateAdmin ()` checks if the provided admin username and password match any existing admin credentials in the system.
- **Date Validation:** The function `isValidDate ()` checks if a provided date is valid.

4.4 Error Handling

Error handling is implemented throughout the CrimsonCare Blood Management System to ensure that any issues are properly managed and communicated to the user. The following mechanisms are in place for error handling:

- **File Operations:** When opening files, the system checks if the file operation was successful. If not, an error message is displayed using `strerror (errno)`. For example, in the function `loadAdminCredentials ()`, an error message is displayed if the admin credentials file cannot be opened.
- **Memory Allocation:** When allocating memory, the system checks if the allocation was successful. If not, an error message is displayed. For example, in the function `addAdmin ()`, an error message is displayed if memory allocation for a new admin fails.
- **Input Errors:** If any input validation fails, an appropriate error message is displayed to the user. For example, if an invalid blood group ID is provided, the function `isValidBloodGroup ()` displays an error message.
- **Transaction Errors:** When adding transactions, the system checks for various errors such as invalid blood group IDs, invalid hospital codes, and invalid dates. If any errors are detected, appropriate error messages are displayed.
- **Admin Operations:** When performing admin operations such as adding or deleting admins, the system checks for errors such as empty credentials, invalid usernames, and duplicate admins. If any errors are detected, appropriate error messages are displayed.

Chapter 5

Usage

5.1 Installation

This section provides the steps to install and set up the CrimsonCare Blood Management System.

5.1.1 Prerequisites

Before you begin, ensure you have the following tools installed on your system:

- **GCC Compiler:** The GNU Compiler Collection (GCC) is a standard compiler for C and C++.

- **Windows:**

1. Download the MinGW installer from the MinGW-w64 project.
2. Choose the appropriate version for your system (32-bit or 64-bit).
3. Run the installer.
4. Once installed, add the MinGW bin directory to your system PATH.
5. Verify the installation by opening Command Prompt and running:

```
gcc --version
```

- **Linux:**

- * **Ubuntu/Debian:**

```
sudo apt update
sudo apt install build-essential
```

- * **Fedora:**

```
sudo dnf groupinstall "Development Tools"
```

- **macOS:**

```
xcode-select --install
```

- **Git:** A version control system to manage source code.
 - Download and install Git from the official Git website. Follow the installation instructions for your operating system.
- **Code::Blocks IDE (Optional):** An open-source Integrated Development Environment (IDE) for C/C++ programming.
 - Download and install Code::Blocks from the official website. Choose the version that includes the MinGW compiler (typically labeled as codeblocks-XX.XXmingw-setup.exe).

5.1.2 Clone the Repository

Clone the repository from GitHub to your local machine:

```
git clone https://github.com/mrasadatik/crimson-care.git
cd crimson-care-main
```

5.1.3 Build for Code::Blocks IDE

1. Open the project in Code::Blocks:
 - Open Code::Blocks IDE.
 - Go to **File** → **Open...** and select `CrimsonCare.cbp`.
2. Build the project:
 - Select the desired build target (Debug or Release).
 - Click on the **Build** button or press **F9**.

5.1.4 Build for Command Line (Using Make)

On Linux/Mac

- **Default Build:**

```
make
```

- **Debug Build:**

```
make debug
```

- **Release Build:**

```
make release
```

On Windows

- **Default Build:**

```
mingw32-make
```

- **Debug Build:**

```
mingw32-make debug
```

- **Release Build:**

```
mingw32-make release
```

5.1.5 Build for Command Line (Without Make)

On Linux/Mac

- **Debug Build:**

```
mkdir -p bin/Debug && gcc -Wall -Wextra -g3 -Iinclude main.c src/*.c -o bin/D
```

- **Release Build:**

```
mkdir -p bin/Release && gcc -Wall -Wextra -O3 -Iinclude main.c src/*.c -o bin/
```

On Windows

- **Debug Build:**

```
mkdir -p bin/Debug && gcc -Wall -Wextra -g3 -mconsole -Iinclude main.c src/*.
```

- **Release Build:**

```
mkdir -p bin/Release && gcc -Wall -Wextra -O3 -mconsole -Iinclude main.c src/
```


5.2 Running the Application

This section provides instructions on how to run the CrimsonCare Blood Management System application after it has been installed and built.

5.2.1 Running the Application on Linux/Mac

After building the project, you can run the application from the command line.

Debug Build

To run the application in Debug mode, use the following command:

```
./bin/Debug/CrimsonCare
```

Release Build

To run the application in Release mode, use the following command:

```
./bin/Release/CrimsonCare
```

5.2.2 Running the Application on Windows

After building the project, you can run the application from the command line.

Debug Build

To run the application in Debug mode, use the following command:

```
bin\Debug\CrimsonCare.exe
```

Release Build

To run the application in Release mode, use the following command:

```
bin\Release\CrimsonCare.exe
```

5.2.3 Application Usage

Once the application is running, you will be presented with the main menu. The main menu provides options for both users and administrators.

User Menu

The default menu is the user menu, which includes the following options:

- **Buy Blood:** Purchase blood from the blood bank.
- **Sell Blood:** Donate blood to the blood bank.
- **Display Blood Stock:** View the current blood stock.
- **Admin Login:** Access the admin panel (requires admin credentials).
- **Exit:** Exit the application.

Admin Menu

After logging in as an admin, you will have access to the admin menu, which includes the following options:

- **Add Hospital:** Add a new hospital to the system.
- **Update Blood Price and Quantity:** Update the price and quantity of blood stocks.
- **Change Admin Password:** Change the password for an admin account.
- **Add Admin:** Add a new admin account.
- **Delete Admin:** Delete an existing admin account.
- **Delete Hospital:** Delete an existing hospital from the system.
- **Display Admin:** View the list of admin accounts.
- **Display Hospital:** View the list of hospitals.
- **Display Blood Stock:** View the current blood stock.
- **Display Transaction:** View the transaction logs.
- **Exit Admin Panel:** Exit the admin panel and return to the user menu.

Chapter 6

Future Work

6.1 Enhancements

Potential enhancements and future improvements include:

- Implementing real-time data synchronization for better data consistency.
- Adding support for cloud storage and remote access.
- Enhancing the system to be thread-safe for better performance.

6.2 Database Integration

Plans for integrating a database include:

- Using SQLite or at least JSON to store data in more structured and stable way.

6.3 Security Improvements

Plans for improving security include:

- Implementing password hashing to enhance security for admin credentials.
- Adding encryption for sensitive data stored in the database.
- Implementing secure communication protocols for data transmission.

Chapter 7

Conclusion

The CrimsonCare Blood Management System is a complete solution that helps hospitals manage blood donations, stock, and transactions. It uses data structures and C programming to create a strong and efficient console application.

7.1 Summary of Achievements

Throughout the development of CrimsonCare, several key objectives were achieved:

- **Data Structures and C Programming:** The project showcased the use of linked lists and dynamic memory allocation to manage data efficiently.
- **Blood Management System:** A full-fledged system was developed to manage hospitals, blood stock, and transactions, allowing admins to add, remove, and update records.
- **Data Integrity and Error Handling:** Input validation and error handling mechanisms were implemented to ensure data correctness and system reliability.
- **Cross-Platform Compatibility:** The system was designed to work on Windows, Linux, and macOS, supporting both Debug and Release builds.

7.2 Final Thoughts

The CrimsonCare Blood Management System shows how data structures and C programming can solve real-world problems. By focusing on the important needs of blood management in hospitals, this project demonstrates technical skill while helping improve healthcare systems.

Chapter 8

References

The following references were used in the development of the CrimsonCare Blood Management System:

- **Code::Blocks:** The IDE used for this project. Available at: [here](#)
- **Doxygen:** The documentation generator used for this project. Available at: [here](#)
- **Git:** The version control system used for this project. Available at: [here](#)
- **GitHub:** The platform used to host the repository. Available at: [here](#)
- **Conventional Commits:** The specification used for commit messages. Available at: [here](#)
- **LaTeX (MikTeX):** The LaTeX distribution used to generate the report. Available at: [here](#)
- **Stack Overflow Question:** How to show enter password in the form of Asterisks (*) on terminal. Available at: [here](#)
- **Stack Overflow Question:** How to display asterisk for input password in C++ using CLion. Available at: [here](#)
- **Dev.to Post:** How to take hidden password from terminal in C/C++. Available at: [here](#)
- **Report Writing Inspiration:**
 - HeadBall Report. Available at: [here](#)
 - Software Engineering Final Year Project Report. Available at: [here](#)
 - rvce-latex/Project-Report-Template. Available at: [here](#)

Appendix A

Appendix A: Source Code

A.1 Source Code

A.1.1 main.c

```
1  /*!
2  * @file main.c
3  *
4  * @brief Main source file
5  * @details This file contains the implementation of the
6      ↪ main function.
7  *
8  * @author CrimsonCare Team
9  * @date 2025-01-18
10 *
11 * @copyright
12 * Copyright (c) 2025 CrimsonCare Team
13 *
14 * Permission is hereby granted, free of charge, to any
15     ↪ person obtaining a copy
16 * of this software and associated documentation files (the
17     ↪ "Software"), to deal
18 * in the Software without restriction, including without
19     ↪ limitation the rights
20 * to use, copy, modify, merge, publish, distribute,
21     ↪ sublicense, and/or sell
22 * copies of the Software, and to permit persons to whom the
23     ↪ Software is
24 * furnished to do so, subject to the following conditions:
25 *
26 * The above copyright notice and this permission notice
27     ↪ shall be included in all
28 * copies or substantial portions of the Software.
```

```
22  *
23  * THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY
    ↳ KIND, EXPRESS OR
24  * IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
    ↳ MERCHANTABILITY,
25  * FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN
    ↳ NO EVENT SHALL THE
26  * AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM,
    ↳ DAMAGES OR OTHER
27  * LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR
    ↳ OTHERWISE, ARISING FROM,
28  * OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR
    ↳ OTHER DEALINGS IN THE
29  * SOFTWARE.
30  */
31 #include "include/admin_manager.h"
32 #include "include/hospital_manager.h"
33 #include "include/blood_manager.h"
34 #include "include/transaction_manager.h"
35 #include "include/misc.h"
36
37 int main(void) {
38     loadBloodGroups();
39     loadHospitals();
40     loadAdminCredentials();
41
42     displayWelcomeMessage();
43
44     uint32_t choice;
45     bool isAdmin = false;
46     char currentAdminUsername[MAX_USERNAME_LENGTH];
47
48     while (1) {
49         displayUserMenu();
50
51         if (scanf("%u", &choice) != 1) {
52             printf("Error: Invalid input.\n");
53             clearInputBuffer();
54             continue;
55         }
56
57         clearInputBuffer();
58
59         switch (choice) {
60             case 1: {
61                 TransactionType transactionType = BUY;
62                 if (!isBloodAvailable(NULL,
    ↳ transactionType)) {
```

```
63         printf("No blood available for
           ↳ purchase.\n");
64         break;
65     }
66     char
           ↳ hospitalCode[MAX_TRANSACTION_NAME_LENGTH];
67     uint32_t bloodGroupId;
68     uint32_t quantity;
69     printf("Enter Entity Name (Hospital Code):
           ↳ ");
70     fgets(hospitalCode, sizeof(hospitalCode),
           ↳ stdin);
71     hospitalCode[strcspn(hospitalCode, "\n")] =
           ↳ 0;
72     if (containsPipe(hospitalCode)) {
73         printf("Error: Hospital code cannot
           ↳ contain a pipe character.\n");
74         break;
75     }
76     if (!validateHospitalCode(hospitalCode)) {
77         printf("Error: Invalid hospital
           ↳ code.\n");
78         break;
79     }
80
81     displayBloodStocks();
82     printf("Enter Blood Group ID: ");
83     scanf("%u", &bloodGroupId);
84     clearInputBuffer();
85     if (!isValidBloodGroup(bloodGroupId)) {
86         printf("Error: Invalid blood group.\n");
87         break;
88     }
89     if (!isBloodAvailable(&bloodGroupId,
           ↳ transactionType)) {
90         printf("Blood %s is not available for
           ↳ purchase.\n",
           ↳ getBloodGroupId(bloodGroupId));
91         break;
92     }
93     printf("Enter Quantity: ");
94     scanf("%u", &quantity);
95     clearInputBuffer();
96     if (addTransaction(transactionType,
           ↳ hospitalCode, bloodGroupId, quantity))
           ↳ {
97         printf("Transaction successful for %s
           ↳ (%s).\n",
```



```

    ↪ getHospitalNameByCode(hospitalCode),
    ↪ hospitalCode);
98     } else {
99         printf("Error: Transaction failed.\n");
100     }
101     break;
102 }
103 case 2: {
104     TransactionType transactionType = SELL;
105     if (!isBloodAvailable(NULL,
106         ↪ transactionType)) {
107         printf("No blood available for sale.\n");
108         break;
109     }
110     char donorName[MAX_TRANSACTION_NAME_LENGTH];
111     uint32_t bloodGroupId;
112     uint32_t quantity;
113     printf("Enter Donor Name: ");
114     fgets(donorName, sizeof(donorName), stdin);
115     donorName[strcspn(donorName, "\n")] = 0;
116     if (containsPipe(donorName)) {
117         printf("Error: Donor name cannot contain
118             ↪ a pipe character.\n");
119         break;
120     }
121     displayBloodStocks();
122     printf("Enter Blood Group ID: ");
123     scanf("%u", &bloodGroupId);
124     clearInputBuffer();
125     if (!isValidBloodGroup(bloodGroupId)) {
126         printf("Error: Invalid blood group.\n");
127         break;
128     }
129     if (!isBloodAvailable(&bloodGroupId,
130         ↪ transactionType)) {
131         printf("Blood %s is not available for
132             ↪ sale.\n",
133             ↪ getBloodGroupId(bloodGroupId));
134         break;
135     }
136     printf("Enter Quantity: ");
137     scanf("%u", &quantity);
138     clearInputBuffer();
139     if (addTransaction(transactionType,
140         ↪ donorName, bloodGroupId, quantity)) {
141         printf("Transaction successful for
142             ↪ %s.\n", donorName);

```

[illegible]

175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194

```
char
    ↪ hospitalLocation[MAX_HOSPITAL_LOCA
printf("Enter Hospital Name:
    ↪ ");
fgets(hospitalName,
    ↪ sizeof(hospitalName),
    ↪ stdin);
hospitalName[strcspn(hospitalName,
    ↪ "\n")] = 0;
if
    ↪ (containsPipe(hospitalName))
    ↪ {
    printf("Error: Hospital
        ↪ name cannot
        ↪ contain a pipe
        ↪ character.\n");
    break;
}
printf("Enter Hospital
    ↪ Location: ");
fgets(hospitalLocation,
    ↪ sizeof(hospitalLocation),
    ↪ stdin);
hospitalLocation[strcspn(hospitalLocati
    ↪ "\n")] = 0;
if
    ↪ (containsPipe(hospitalLocation))
    ↪ {
    printf("Error: Hospital
        ↪ location cannot
        ↪ contain a pipe
        ↪ character.\n");
    break;
}
char* hospitalCode =
    ↪ addHospital(hospitalName,
    ↪ hospitalLocation);
if (hospitalCode) {
    printf("Hospital %s
        ↪ added successfully
        ↪ by %s.\n",
        ↪ hospitalName,
        ↪ currentAdminUsername);
    printf("Hospital Code:
        ↪ %s\n",
        ↪ hospitalCode);
} else {
```

```
195         printf("Error: Could not
196             ↪ add hospital.\n");
197     }
198     break;
199 }
200 case 2: {
201     displayBloodGroups();
202     uint32_t bloodGroupId;
203     uint32_t newQuantity;
204     printf("Enter Blood Group
205         ↪ ID: ");
206     scanf("%u", &bloodGroupId);
207     clearInputBuffer();
208     if
209         ↪ (!isValidBloodGroup(bloodGroupId))
210         ↪ {
211         printf("Error: Invalid
212             ↪ blood group.\n");
213         break;
214     }
215     if
216         ↪ (!isBloodAvailable(&bloodGroupId,
217         ↪ SELL)) {
218         float newPrice;
219         printf("Price for %s is
220             ↪ not set.\n",
221             ↪ getBloodGroupId(bloodGroupId));
222         printf("Enter New Price:
223             ↪ ");
224         scanf("%f", &newPrice);
225         clearInputBuffer();
226         if
227             ↪ (updateBloodPrice(bloodGroupId,
228             ↪ newPrice)) {
229             printf("Blood price
230                 ↪ for %s updated
231                 ↪ successfully
232                 ↪ by %s.\n",
233                 ↪ getBloodGroupId(bloodGroupId),
234                 ↪ currentAdminUsername);
235         } else {
236             printf("Error: Could
237                 ↪ not update
238                 ↪ blood
239                 ↪ price.\n");
240         }
241     }
242 }
```

222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248

```
printf("Enter New Quantity:
↪ ");
scanf("%u", &newQuantity);
clearInputBuffer();
if
↪ (updateBloodQuantity(bloodGroupId,
↪ newQuantity)) {
    printf("Blood quantity
↪ for %s updated
↪ successfully by
↪ %s.\n",
↪ getBloodGroupId(bloodGroupId,
↪ currentAdminUsername));
} else {
    printf("Error: Could not
↪ update blood
↪ quantity.\n");
}
break;
}
case 3: {
    displayBloodGroups();
    uint32_t bloodGroupId;
    float newPrice;
    printf("Enter Blood Group
↪ ID: ");
    scanf("%u", &bloodGroupId);
    clearInputBuffer();
    if
↪ (!isValidBloodGroup(bloodGroupId))
↪ {
        printf("Error: Invalid
↪ blood group.\n");
        break;
    }
    printf("Enter New Price: ");
    scanf("%f", &newPrice);
    clearInputBuffer();
    if
↪ (updateBloodPrice(bloodGroupId,
↪ newPrice)) {
        printf("Blood price for
↪ %s updated
↪ successfully by
↪ %s.\n",
↪ getBloodGroupId(bloodGroupId,
↪ currentAdminUsername));
    } else {
```

249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272

```
        printf("Error: Could not
               ↳ update blood
               ↳ price.\n");
    }
    break;
}
case 4: {
    char
        ↳ adminOldPassword[MAX_PASSWORD_LENGTH];
    char
        ↳ adminNewPassword[MAX_PASSWORD_LENGTH];
    printf("Enter Old Password:
           ↳ ");
    getPassword(adminOldPassword,
               ↳ sizeof(adminOldPassword));
    printf("Enter New Password:
           ↳ ");
    getPassword(adminNewPassword,
               ↳ sizeof(adminNewPassword));
    if
        ↳ (changeAdminPassword(currentAdminUsername,
        ↳ adminOldPassword,
        ↳ adminNewPassword)) {
        printf("Password for %s
               ↳ updated
               ↳ successfully.\n",
               ↳ currentAdminUsername);
    } else {
        printf("Error: Could not
               ↳ change admin
               ↳ password.\n");
    }
    break;
}
case 5: {
    char
        ↳ currentAdminPassword[MAX_PASSWORD_LENGTH];
    char
        ↳ newAdminUsername[MAX_USERNAME_LENGTH];
    char
        ↳ newAdminPassword[MAX_PASSWORD_LENGTH];
    char
        ↳ confirmNewAdminPassword[MAX_PASSWORD_LENGTH];
    printf("To continue, please
           ↳ enter your (%s)
           ↳ current password: ",
           ↳ currentAdminUsername);
```

273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295

```
getPassword(currentAdminPassword ,
    ↪ sizeof(currentAdminPassword));
if
    ↪ (validateAdmin(currentAdminUsername,
    ↪ currentAdminPassword))
    ↪ {
    printf("Password
        ↪ verified
        ↪ successfully.\n");
} else {
    printf("Error: Invalid
        ↪ password.\n");
    break;
}
printf("Enter New Admin
    ↪ Username: ");
fgets(newAdminUsername ,
    ↪ sizeof(newAdminUsername),
    ↪ stdin);
newAdminUsername[strcspn(newAdminUsername,
    ↪ "\n")] = 0;
if
    ↪ (!checkUsername(newAdminUsername))
    ↪ {
    printf("Error: Invalid
        ↪ username. Username
        ↪ can only contain
        ↪ lowercase letters
        ↪ and digits.\n");
    break;
}
printf("Enter New Admin
    ↪ Password: ");
getPassword(newAdminPassword ,
    ↪ sizeof(newAdminPassword));
printf("Confirm New Admin
    ↪ Password: ");
getPassword(confirmNewAdminPassword ,
    ↪ sizeof(confirmNewAdminPassword));
if (strcmp(newAdminPassword ,
    ↪ confirmNewAdminPassword)
    ↪ != 0) {
    printf("Error: Passwords
        ↪ do not match.\n");
    break;
}
if
    ↪ (addAdmin(newAdminUsername ,
```

296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316

```
        ↪ newAdminPassword ,
        ↪ currentAdminUsername ,
        ↪ currentAdminPassword))
        ↪ {
        ↪     printf("New admin %s
        ↪         ↪ added successfully
        ↪         ↪ by %s.\n",
        ↪         ↪ newAdminUsername ,
        ↪         ↪ currentAdminUsername);
    } else {
        ↪     printf("Error: Could not
        ↪         ↪ add new admin.\n");
    }
    ↪ break;
}
case 6: {
    ↪ char
    ↪     delAdminUsername [MAX_USERNAME_LENGTH];
    ↪ char
    ↪     currentAdminPassword [MAX_PASSWORD_LENGTH];
    ↪     printf("To continue, please
    ↪         ↪ enter your (%s)
    ↪         ↪ current password: ",
    ↪         ↪ currentAdminUsername);
    ↪     getPassword(currentAdminPassword ,
    ↪         ↪ sizeof(currentAdminPassword));
    ↪     if
    ↪         ↪ (validateAdmin(currentAdminUsername,
    ↪         ↪ currentAdminPassword))
    ↪         ↪ {
    ↪         ↪     printf("Password
    ↪         ↪         ↪ verified
    ↪         ↪         ↪ successfully.\n");
    ↪     } else {
    ↪         ↪     printf("Error: Invalid
    ↪         ↪         ↪ password.\n");
    ↪         ↪     break;
    ↪     }
    ↪     printf("Enter Admin Username
    ↪         ↪ to Delete: ");
    ↪     fgets(delAdminUsername ,
    ↪         ↪ sizeof(delAdminUsername),
    ↪         ↪ stdin);
    ↪     delAdminUsername[strcspn(delAdminUsername,
    ↪         ↪ "\n")] = 0;
    ↪     if
    ↪         ↪ (!checkUsername(delAdminUsername))
    ↪         ↪ {
```


317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338

```
        printf("Error: Invalid
        ↪ username. Username
        ↪ can only contain
        ↪ lowercase letters
        ↪ and digits.\n");
        break;
    }
    if
        ↪ (deleteAdmin(delAdminUsername,
        ↪ currentAdminUsername,
        ↪ currentAdminPassword))
        ↪ {
        printf("Admin %s deleted
        ↪ successfully by
        ↪ %s.\n",
        ↪ delAdminUsername,
        ↪ currentAdminUsername);
    } else {
        printf("Error: Could not
        ↪ delete admin.\n");
    }
    break;
}
case 7: {
    char
        ↪ delHospitalCode[MAX_HOSPITAL_CODE_
    char
        ↪ currentAdminPassword[MAX_PASSWORD_
    printf("To continue, please
    ↪ enter your (%s)
    ↪ current password: ",
    ↪ currentAdminUsername);
    getPassword(currentAdminPassword,
        ↪ sizeof(currentAdminPassword));
    if
        ↪ (validateAdmin(currentAdminUsername,
        ↪ currentAdminPassword))
        ↪ {
        printf("Password
        ↪ verified
        ↪ successfully.\n");
    } else {
        printf("Error: Invalid
        ↪ password.\n");
        break;
    }
    printf("Enter Hospital Code
    ↪ to Delete: ");
```

```
339         fgets(delHospitalCode ,
340               ↪ sizeof(delHospitalCode),
341               ↪ stdin);
342         delHospitalCode[strcspn(delHospitalCode
343               ↪ "\n")] = 0;
344         if
345             ↪ (containsPipe(delHospitalCode))
346             ↪ {
347             printf("Error: Hospital
348                   ↪ code cannot
349                   ↪ contain a pipe
350                   ↪ character.\n");
351             break;
352         }
353         if
354             ↪ (deleteHospital(delHospitalCode ,
355             ↪ currentAdminUsername ,
356             ↪ currentAdminPassword))
357             ↪ {
358             printf("Hospital %s
359                   ↪ deleted
360                   ↪ successfully by
361                   ↪ %s.\n",
362                   ↪ delHospitalCode ,
363                   ↪ currentAdminUsername);
364         } else {
365             printf("Error: Could not
366                   ↪ delete
367                   ↪ hospital.\n");
368         }
369         break;
370     }
371     case 8: {
372         displayAdmin();
373         break;
374     }
375     case 9: {
376         displayHospitals();
377         break;
378     }
379     case 10: {
380         displayBloodStocks();
381         break;
382     }
383     case 11: {
384         displayTransactions();
385         break;
386     }
387 }
```

```
368         case 12:
369             printf("Exiting admin
370                 ↪ panel...\n");
371             isAdmin = false;
372             break;
373         default:
374             printf("Error: Invalid option.
375                 ↪ Please try again.\n");
376     }
377 } else {
378     printf("Error: Invalid admin
379         ↪ credentials.\n");
380 }
381 break;
382 }
383 case 5:
384     printf("Exiting...\n");
385     break;
386 default:
387     printf("Error: Invalid option. Please try
388         ↪ again.\n");
389 }
390 if (choice == 5) {
391     break;
392 }
393 }
394 freeAdmin();
395 freeHospital();
396 freeBloodList();
397 return 0;
398 }
```

Listing A.1: main.c

A.1.2 admin_manager.c

```
1  /*!
2  * @file admin_manager.c
3  * @brief Admin manager source file
4  * @details This file contains the implementation of the
5      ↪ admin manager module.
6  *
7  * @author CrimsonCare Team
8  * @date 2025-01-18
9  *
10 * @copyright
```

```

10  * Copyright (c) 2025 CrimsonCare Team
11  *
12  * Permission is hereby granted, free of charge, to any
13  *   ↳ person obtaining a copy
14  * of this software and associated documentation files (the
15  *   ↳ "Software"), to deal
16  * in the Software without restriction, including without
17  *   ↳ limitation the rights
18  * to use, copy, modify, merge, publish, distribute,
19  *   ↳ sublicense, and/or sell
20  * copies of the Software, and to permit persons to whom the
21  *   ↳ Software is
22  * furnished to do so, subject to the following conditions:
23  *
24  * The above copyright notice and this permission notice
25  *   ↳ shall be included in all
26  * copies or substantial portions of the Software.
27  *
28  * THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY
29  *   ↳ KIND, EXPRESS OR
30  * IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
31  *   ↳ MERCHANTABILITY,
32  * FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN
33  *   ↳ NO EVENT SHALL THE
34  * AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM,
35  *   ↳ DAMAGES OR OTHER
36  * LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR
37  *   ↳ OTHERWISE, ARISING FROM,
38  * OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR
39  *   ↳ OTHER DEALINGS IN THE
40  * SOFTWARE.
41  */
42 #include "../include/admin_manager.h"
43
44 /*!
45  * @brief Admin head pointer
46  * @details This pointer is used to track admin linkedlist
47  *   ↳ on runtime.
48  */
49 Admin* adminHead = NULL;
50
51 /*!
52  * @name saveAdminCredentials
53  * @brief Save admin credentials to file
54  * @details This function saves the linkedlist data
55  * from 'adminHead' to the file
56  *   ↳ 'resources/db/admin_credentials.dat'.

```

```
43  * '.dat' is used to store credentials in binary format for
    ↳ surface level security.
44  *
45  * @note The file path 'resources/db' is relative to the
    ↳ project root directory.
46  * Make sure that the folder exists also to run the program
    ↳ from the root directory.
47  *
48  * @post The linkedlist data is saved to the file.
49  *
50  * @exception fopen() - If the file cannot be opened, an
    ↳ error message is displayed.
51  * @exception fwrite() - If the file cannot be written, an
    ↳ error message is displayed.
52  */
53 void saveAdminCredentials(void) {
54     errno = 0;
55     FILE* file = fopen("resources/db/admin_credentials.dat",
    ↳ "wb");
56     if (!file) {
57         if (errno != ENOENT) {
58             printf("Error opening admin credentials file:
    ↳ %s\n", strerror(errno));
59             return;
60         }
61     }
62
63     Admin* temp = adminHead;
64     while (temp != NULL) {
65         if (fwrite(temp, sizeof(Admin), 1, file)) {
66             temp = temp->next;
67         } else {
68             printf("Error writing admin credentials: %s\n",
    ↳ strerror(errno));
69             freeAdmin();
70             fclose(file);
71             return;
72         }
73     }
74     fclose(file);
75 }
76
77 /*!
78  * @name loadAdminCredentials
79  * @brief Load admin credentials from file
80  * @details This function loads the admin credentials
81  * from the file 'resources/db/admin_credentials.dat' and
```

```
82  * stores it in the 'adminHead' linkedlist. If file is not
    ↪ found,
83  * it creates a new admin with default credentials and
    ↪ stores it in the file.
84  *
85  * @note The file path 'resources/db' is relative to the
    ↪ project root directory.
86  * Make sure that the folder exists also to run the program
    ↪ from the root directory.
87  *
88  * @post If the file is not found, a new admin is created
    ↪ with default credentials
89  * and stored in the file. If the file is found, the admin
    ↪ credentials are loaded
90  * from the file and stored in the 'adminHead' linkedlist.
91  *
92  * @exception fopen() - If the file cannot be opened, an
    ↪ error message is displayed.
93  * @exception malloc() - If memory allocation fails, an
    ↪ error message is displayed.
94  */
95 void loadAdminCredentials(void) {
96     errno = 0;
97     FILE* file = fopen("resources/db/admin_credentials.dat",
    ↪ "rb");
98     if (!file) {
99         if (errno == ENOENT) {
100             Admin* newAdmin = (Admin*)malloc(sizeof(Admin));
101             if (newAdmin) {
102                 strcpy(newAdmin->username, "admin");
103                 strcpy(newAdmin->password, "1234");
104                 newAdmin->next = NULL;
105                 adminHead = newAdmin;
106                 saveAdminCredentials();
107             } else {
108                 printf("Error allocating memory for admin:
    ↪ %s\n", strerror(errno));
109             }
110         } else {
111             printf("Error opening admin credentials file:
    ↪ %s\n", strerror(errno));
112         }
113         return;
114     }
115
116     Admin tempAdmin;
117     while (fread(&tempAdmin, sizeof(Admin), 1, file)) {
118         Admin* newAdmin = (Admin*)malloc(sizeof(Admin));
```

```
119     if (newAdmin) {
120         *newAdmin = tempAdmin;
121         newAdmin->next = adminHead;
122         adminHead = newAdmin;
123     } else {
124         printf("Error allocating memory for admin:
125             ↪ %s\n", strerror(errno));
126         freeAdmin();
127         fclose(file);
128         return;
129     }
130     fclose(file);
131 }
132
133 /*!
134 * @name adminExists
135 * @brief Check if admin exists
136 * @details This function traverses the 'adminHead'
137     ↪ linkedlist
138 * and checks if the username exists in the list.
139 * @param[in] username Admin username
140 * @return True if admin exists, False otherwise
141 * @pre @p username is not empty and valid
142 * @post If the @p username is found in the linkedlist,
143 * the function returns true, otherwise false.
144 * @exception If the @p username is empty, an error message
145     ↪ is displayed.
146 * @exception If the @p username is invalid, an error
147     ↪ message is displayed.
148 */
149
150 bool adminExists(const char* username) {
151     if (strcmp(username, "") == 0) {
152         printf("Error: Admin username cannot be empty.\n");
153         return false;
154     }
155
156     if (!checkUsername(username)) {
157         printf("Error: Invalid username. Username can only
158             ↪ contain lowercase letters and digits.\n");
159         return false;
160     }
161
162     Admin* temp = adminHead;
```

```

162     while (temp != NULL) {
163         if (strcmp(temp->username, username) == 0) {
164             return true;
165         }
166         temp = temp->next;
167     }
168     return false;
169 }
170
171 /*!
172  * @name validateAdmin
173  * @brief Validate admin credentials
174  * @details This function traverses the 'adminHead'
175  *   ↳ linkedlist
176  * and checks if the pair of username and password match.
177  *
178  * @param[in] username Admin username
179  * @param[in] password Admin password
180  *
181  * @return True if credentials are valid, False otherwise
182  *
183  * @pre @p username and @p password are not empty and valid
184  * @post If the pair of @p username and @p password are
185  *   ↳ found in the linkedlist,
186  * the function returns true, otherwise false.
187  *
188  * @exception If the @p username or @p password is empty, an
189  *   ↳ error message is displayed.
190  * @exception If the @p username is invalid, an error
191  *   ↳ message is displayed.
192  */
193
194 bool validateAdmin(const char* username, const char*
195   ↳ password) {
196     if (strcmp(username, "") == 0 || strcmp(password, "") ==
197   ↳ 0) {
198         printf("Error: Admin credentials cannot be
199   ↳ empty.\n");
200         return false;
201     }
202
203     if (!checkUsername(username)) {
204         printf("Error: Invalid username. Username can only
205   ↳ contain lowercase letters and digits.\n");
206         return false;
207     }
208
209     Admin* temp = adminHead;
210     while (temp != NULL) {

```



```

202         if (strcmp(username, temp->username) == 0 &&
203             ↳ strcmp(password, temp->password) == 0) {
204             return true;
205         }
206         temp = temp->next;
207     }
208     return false;
209 }
210
211 /*!
212 * @name addAdmin
213 * @brief Add admin
214 * @details This function adds a new admin to the
215 ↳ 'adminHead' linkedlist
216 * and saves the updated linkedlist to the file
217 ↳ 'resources/db/admin_credentials.dat'.
218 *
219 * @param[in] username Admin username
220 * @param[in] password Admin password
221 * @param[in] currentAdminUsername Current admin username
222 * @param[in] currentAdminPassword Current admin password
223 *
224 * @return True if admin is added, False otherwise
225 *
226 * @note The file path 'resources/db' is relative to the
227 ↳ project root directory.
228 * Make sure that the folder exists also to run the program
229 ↳ from the root directory.
230 *
231 * @pre @p username and @p password are not empty
232 * @pre @p currentAdminUsername and @p currentAdminPassword
233 ↳ are not empty
234 * @pre @p username and @p currentAdminUsername are valid
235 * @post If the @p username and @p password are not found in
236 ↳ the linkedlist,
237 * the function adds the new admin to the linkedlist and
238 ↳ saves the updated linkedlist to the file.
239 *
240 * @exception If the @p username or @p password is empty, an
241 ↳ error message is displayed.
242 * @exception If the @p username or @p currentAdminUsername
243 ↳ is invalid, an error message is displayed.
244 * @exception If the pair of @p currentAdminUsername and @p
245 ↳ currentAdminPassword is invalid,
246 * means that the current admin credentials are not valid,
247 ↳ an error message is displayed.
248 * @exception If the @p username already exists, an error
249 ↳ message is displayed.

```

```
237  * @exception malloc() - If memory allocation fails, an
    ↪ error message is displayed.
238  */
239  bool addAdmin(const char* username, const char* password,
    ↪ const char* currentAdminUsername, const char*
    ↪ currentAdminPassword) {
240      if (strcmp(currentAdminUsername, "") == 0 ||
    ↪ strcmp(currentAdminPassword, "") == 0) {
241          printf("Error: Current admin credentials cannot be
    ↪ empty.\n");
242          return false;
243      }
244
245      if (!checkUsername(currentAdminUsername) ||
    ↪ !checkUsername(username)) {
246          printf("Error: Invalid username. Username can only
    ↪ contain lowercase letters and digits.\n");
247          return false;
248      }
249
250      if (!validateAdmin(currentAdminUsername,
    ↪ currentAdminPassword)) {
251          printf("Error: Invalid current admin
    ↪ credentials.\n");
252          return false;
253      }
254
255      if (adminExists(username)) {
256          printf("Error: Admin already exists.\n");
257          return false;
258      }
259
260      if (strcmp(username, "") == 0 || strcmp(password, "") ==
    ↪ 0) {
261          printf("Error: Admin credentials cannot be
    ↪ empty.\n");
262          return false;
263      }
264
265      Admin* newAdmin = (Admin*)malloc(sizeof(Admin));
266      if (!newAdmin) {
267          printf("Error allocating memory for admin: %s\n",
    ↪ strerror(errno));
268          return false;
269      }
270      strncpy(newAdmin->username, username,
    ↪ sizeof(newAdmin->username) - 1);
```

```

271     newAdmin->username[sizeof(newAdmin->username) - 1] =
        ↳ '\0';
272     strncpy(newAdmin->password, password,
        ↳ sizeof(newAdmin->password) - 1);
273     newAdmin->password[sizeof(newAdmin->password) - 1] =
        ↳ '\0';
274     newAdmin->next = adminHead;
275     adminHead = newAdmin;
276
277     saveAdminCredentials();
278     return true;
279 }
280
281 /*!
282 * @name deleteAdmin
283 * @brief Delete admin
284 * @details This function deletes an admin from the
        ↳ 'adminHead' linkedlist
285 * and saves the updated linkedlist.
286 *
287 * @param[in] username Admin username
288 * @param[in] currentAdminUsername Current admin username
289 * @param[in] currentAdminPassword Current admin password
290 *
291 * @return True if admin is deleted, False otherwise
292 *
293 * @pre @p username is not empty
294 * @pre @p currentAdminUsername and @p currentAdminPassword
        ↳ are not empty
295 * @pre @p username and @p currentAdminUsername are valid
296 * @post If the @p username is found in the linkedlist,
297 * the function deletes the admin from the linkedlist and
        ↳ saves the updated linkedlist to the file.
298 *
299 * @exception If the pair of @p currentAdminUsername and @p
        ↳ currentAdminPassword is invalid,
300 * means that the current admin credentials are not valid,
        ↳ an error message is displayed.
301 * @exception If the @p username does not exist, an error
        ↳ message is displayed.
302 * @exception If the @p username is the same as the current
        ↳ admin username, an error message is displayed.
303 * @exception If the @p username or @p currentAdminUsername
        ↳ is invalid, an error message is displayed.
304 */
305 bool deleteAdmin(const char* username, const char*
        ↳ currentAdminUsername, const char*
        ↳ currentAdminPassword) {

```

```
306     if (strcmp(currentAdminUsername, "") == 0 ||  
307         ↳ strcmp(currentAdminPassword, "") == 0) {  
308         printf("Error: Current admin credentials cannot be  
309             ↳ empty.\n");  
310         return false;  
311     }  
312  
313     if (!checkUsername(currentAdminUsername) ||  
314         ↳ !checkUsername(username)) {  
315         printf("Error: Invalid username. Username can only  
316             ↳ contain lowercase letters and digits.\n");  
317         return false;  
318     }  
319  
320     if (!validateAdmin(currentAdminUsername,  
321         ↳ currentAdminPassword)) {  
322         printf("Error: Invalid current admin  
323             ↳ credentials.\n");  
324         return false;  
325     }  
326  
327     if (!adminExists(username)) {  
328         printf("Error: Admin does not exist.\n");  
329         return false;  
330     }  
331  
332     if (strcmp(username, "") == 0) {  
333         printf("Error: Admin username cannot be empty.\n");  
334         return false;  
335     }  
336  
337     if (strcmp(username, currentAdminUsername) == 0) {  
338         printf("Error: Cannot delete current admin.\n");  
339         return false;  
340     }  
341  
342     Admin* temp = adminHead;  
343     Admin* prev = NULL;  
344  
345     while (temp != NULL) {  
346         if (strcmp(temp->username, username) == 0) {  
347             if (prev == NULL) {  
348                 adminHead = temp->next;  
349             } else {  
350                 prev->next = temp->next;  
351             }  
352             free(temp);  
353             saveAdminCredentials();  
354         }
```

```
348         return true;
349     }
350     prev = temp;
351     temp = temp->next;
352 }
353 return false;
354 }
355
356 /*!
357  * @name changeAdminPassword
358  * @brief Change admin password
359  * @details This function changes the password of an admin
360  *   ↳ in the 'adminHead' linkedlist
361  * and saves the updated linkedlist.
362  *
363  * @param[in] username Admin username
364  * @param[in] oldPassword Old password
365  * @param[in] newPassword New password
366  *
367  * @return True if password is changed, False otherwise
368  *
369  * @pre @p username and @p oldPassword are not empty
370  * @pre @p username is valid
371  * @pre @p newPassword is not empty
372  * @post If the pair of @p username and @p oldPassword are
373  *   ↳ found in the linkedlist,
374  * the function changes the password of the admin and saves
375  *   ↳ the updated linkedlist.
376  *
377  * @exception If the @p username or @p oldPassword is empty,
378  *   ↳ an error message is displayed.
379  * @exception If the pair of @p username and @p oldPassword
380  *   ↳ is not found in the linkedlist,
381  * an error message is displayed.
382  * @exception If the @p username is invalid, an error
383  *   ↳ message is displayed.
384  */
385 bool changeAdminPassword(const char* username, const char*
386   ↳ oldPassword, const char* newPassword) {
387     if (strcmp(username, "") == 0 || strcmp(oldPassword, "")
388   ↳ == 0) {
389         printf("Error: Username or old password cannot be
390   ↳ empty.\n");
391         return false;
392     }
393
394     if (!checkUsername(username)) {
```

```

386     printf("Error: Invalid username. Username can only
        ↳ contain lowercase letters and digits.\n");
387     return false;
388 }
389
390 if (!validateAdmin(username, oldPassword)) {
391     printf("Error: Invalid password.\n");
392     return false;
393 }
394
395 if (strcmp(newPassword, "") == 0) {
396     printf("Error: New password cannot be empty.\n");
397     return false;
398 }
399
400 Admin* temp = adminHead;
401 while (temp != NULL) {
402     if (strcmp(username, temp->username) == 0 &&
        ↳ strcmp(oldPassword, temp->password) == 0) {
403         strncpy(temp->password, newPassword,
            ↳ sizeof(temp->password) - 1);
404         temp->password[sizeof(temp->password) - 1] =
            ↳ '\0';
405         saveAdminCredentials();
406         return true;
407     }
408     temp = temp->next;
409 }
410 return false;
411 }
412
413 /*!
414  * @name displayAdmin
415  * @brief Display all admins
416  * @details This function displays all admins in the
        ↳ 'adminHead' linkedlist.
417  *
418  * @post If the 'adminHead' linkedlist is not empty,
419  * the function displays all admins in the linkedlist.
420  */
421 void displayAdmin(void) {
422     Admin* temp = adminHead;
423     printf("\nRegistered Admins:\n");
424     while (temp != NULL) {
425         printf("\tUsername: %s\n", temp->username);
426         temp = temp->next;
427         if (temp != NULL) {
428             printf("\t-----\n");

```

```
429     }
430 }
431 }
432
433 /*!
434 * @name freeAdmin
435 * @brief Free admin list
436 * @details This function frees the memory allocated for the
437     ↪ 'adminHead' linkedlist.
438 *
439 * @post The memory allocated for the 'adminHead' linkedlist
440     ↪ is freed.
441 */
442 void freeAdmin(void) {
443     Admin* current = adminHead;
444     while (current != NULL) {
445         Admin* temp = current;
446         current = current->next;
447         free(temp);
448     }
449     adminHead = NULL;
450 }
```

Listing A.2: admin_manager.c

A.1.3 blood_manager.c

```
1 /*!
2 * @file blood_manager.c
3 *
4 * @brief Blood manager source file
5 * @details This file contains the implementation of the
6     ↪ functions for the blood manager module.
7 *
8 * @author CrimsonCare Team
9 * @date 2025-01-18
10 *
11 * @copyright
12 * Copyright (c) 2025 CrimsonCare Team
13 *
14 * Permission is hereby granted, free of charge, to any
15     ↪ person obtaining a copy
16 * of this software and associated documentation files (the
17     ↪ "Software"), to deal
18 * in the Software without restriction, including without
19     ↪ limitation the rights
```

```

16  * to use, copy, modify, merge, publish, distribute,
    ↪ sublicense, and/or sell
17  * copies of the Software, and to permit persons to whom the
    ↪ Software is
18  * furnished to do so, subject to the following conditions:
19  *
20  * The above copyright notice and this permission notice
    ↪ shall be included in all
21  * copies or substantial portions of the Software.
22  *
23  * THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY
    ↪ KIND, EXPRESS OR
24  * IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
    ↪ MERCHANTABILITY,
25  * FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN
    ↪ NO EVENT SHALL THE
26  * AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM,
    ↪ DAMAGES OR OTHER
27  * LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR
    ↪ OTHERWISE, ARISING FROM,
28  * OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR
    ↪ OTHER DEALINGS IN THE
29  * SOFTWARE.
30  */
31 #include "../include/blood_manager.h"
32 #include "../include/transaction_manager.h"
33
34 /*!
35  * @brief Blood stock head pointer
36  * @details This pointer is used to track blood stock
    ↪ linkedlist on runtime.
37  */
38 BloodStock* bloodHead = NULL;
39
40 /*!
41  * @brief Blood groups
42  * @details This array contains the available blood groups.
43  */
44 char* availableBloodGroups[8] = { "A+", "A-", "B+", "B-",
    ↪ "O+", "O-", "AB+", "AB-" };
45
46 /*!
47  * @name isValidBloodGroup
48  * @brief Check if blood group is valid
49  * @details This function checks if the given blood group id
    ↪ is valid
50  * by checking the size of the 'availableBloodGroups' array.
51  *

```



```

52  * @param[in] id Blood group id
53  *
54  * @return True if blood group is valid, False otherwise
55  *
56  * @post If the @p id is within the range of the
57  *       ↪ 'availableBloodGroups' array,
58  * the function returns true. Otherwise, it returns false.
59  */
60 bool isValidBloodGroup(uint32_t id) {
61     return id <= (sizeof(availableBloodGroups) /
62         ↪ sizeof(availableBloodGroups[0]));
63 }
64
65 /*!
66  * @name addBloodGroup
67  * @brief Add blood group
68  * @details This function adds a new blood group to the
69  *       ↪ 'bloodHead' linkedlist.
70  *
71  * @param[in] id Blood group id
72  * @param[in] bloodGroup Blood group name
73  * @param[in] price Blood group price
74  * @param[in] quantity Blood group quantity
75  *
76  * @return True if blood group is added, False otherwise
77  *
78  * @pre @p id is valid
79  * @pre @p bloodGroup is not empty
80  * @post Updates the blood stock in the 'bloodHead'
81  *       ↪ linkedlist.
82  *
83  * @exception If the @p bloodGroup is empty, an error
84  *       ↪ message is displayed.
85  * @exception If the @p id is not valid, an error message is
86  *       ↪ displayed.
87  * @exception malloc() - If the memory allocation for the
88  *       ↪ new blood group fails, an error message is displayed.
89  */
90 bool addBloodGroup(uint32_t id, const char* bloodGroup,
91     ↪ float price, uint32_t quantity) {
92     if (strcmp(bloodGroup, "") == 0) {
93         printf("Error: Invalid blood group data.\n");
94         return false;
95     }
96
97     if (!isValidBloodGroup(id)) {
98         printf("Error: Invalid blood group id.\n");
99         return false;
100     }

```

```

92     }
93
94     BloodStock* newGroup =
95         ↪ (BloodStock*)malloc(sizeof(BloodStock));
96     if (!newGroup) {
97         printf("Error allocating memory for blood group:
98             ↪ %s\n", strerror(errno));
99         return false;
100     }
101     strncpy(newGroup->bloodGroup, bloodGroup,
102         ↪ BLOOD_GROUP_NAME_LENGTH - 1);
103     newGroup->bloodGroup[BLOOD_GROUP_NAME_LENGTH - 1] = '\0';
104     newGroup->price = price;
105     newGroup->quantity = quantity;
106     newGroup->id = id;
107     newGroup->next = NULL;
108
109     if (bloodHead == NULL) {
110         bloodHead = newGroup;
111     } else {
112         BloodStock* temp = bloodHead;
113         while (temp->next != NULL) {
114             temp = temp->next;
115         }
116         temp->next = newGroup;
117     }
118     return true;
119 }
120
121 /*!
122  * @name initializeBloodGroups
123  * @brief Initialize blood groups
124  * @details This function helps to initialize the default
125     ↪ blood groups
126  * to the 'bloodHead' linkedlist.
127  *
128  * @post The blood groups are added to the 'bloodHead'
129     ↪ linkedlist.
130  *
131  * @exception If adding blood group fails, an error message
132     ↪ is displayed.
133  */
134 void initializeBloodGroups(void) {
135     for (uint8_t i = 0; i < (sizeof(availableBloodGroups) /
136         ↪ sizeof(availableBloodGroups[0])); i++) {
137         if (!addBloodGroup(i + 1, availableBloodGroups[i],
138             ↪ 0.0, 0)) {

```

```
131         printf("Error: Failed to initialize blood group
           ↳ %s.\n", availableBloodGroups[i]);
132     }
133 }
134 }
135
136 /*!
137  * @name saveBloodGroups
138  * @brief Save blood groups to file
139  * @details This function saves the linkedlist data
140  * from 'bloodHead' to the file
           ↳ 'resources/db/blood_data.txt'.
141  *
142  * @note The file path 'resources/db' is relative to the
           ↳ project root directory.
143  * Make sure that the folder exists also to run the program
           ↳ from the root directory.
144  *
145  * @post The linkedlist data is saved to the file.
146  *
147  * @exception fopen() - If the file cannot be opened, an
           ↳ error message is displayed.
148  */
149 void saveBloodGroups(void) {
150     errno = 0;
151     FILE* file = fopen("resources/db/blood_data.txt", "w");
152     if (!file) {
153         if (errno != ENOENT) {
154             printf("Error opening blood data file: %s\n",
                   ↳ strerror(errno));
155             return;
156         }
157     }
158
159     BloodStock* temp = bloodHead;
160     while (temp != NULL) {
161         fprintf(file, "%u %s %.2f %u\n", temp->id,
           ↳ temp->bloodGroup, temp->price, temp->quantity);
162         temp = temp->next;
163     }
164     fclose(file);
165 }
166
167 /*!
168  * @name updateBloodQuantity
169  * @brief Update blood quantity
170  * @details This function updates the blood quantity of the
           ↳ given blood group id
```

```

171  * by traversing the 'bloodHead' linkedlist.
172  *
173  * @param[in] id Blood group id
174  * @param[in] newQuantity New quantity
175  *
176  * @return True if blood quantity is updated, False otherwise
177  *
178  * @post If the @p id is found in the 'bloodHead' linkedlist,
179  * the function updates the blood quantity and saves the
180  *     ↪ updated linkedlist.
181  *
182  * @exception If the @p id is not found in the 'bloodHead'
183  *     ↪ linkedlist, an error message is displayed.
184  */
185 bool updateBloodQuantity(uint32_t id, uint32_t newQuantity) {
186     if (!isValidBloodGroup(id)) {
187         printf("Error: Invalid blood group id.\n");
188         return false;
189     }
190
191     BloodStock* temp = bloodHead;
192     while (temp != NULL) {
193         if (temp->id == id) {
194             temp->quantity = newQuantity;
195             saveBloodGroups();
196             return true;
197         }
198         temp = temp->next;
199     }
200     return false;
201 }
202
203 /*!
204  * @name updateBloodPrice
205  * @brief Update blood price
206  * @details This function updates the blood price of the
207  *     ↪ given blood group id
208  * by traversing the 'bloodHead' linkedlist.
209  *
210  * @param[in] id Blood group id
211  * @param[in] newPrice New price
212  *
213  * @return True if blood price is updated, False otherwise
214  *
215  * @post If the @p id is found in the 'bloodHead' linkedlist,
216  * the function updates the blood price and saves the
217  *     ↪ updated linkedlist.
218  *

```

```
215  * @exception If the @p id is not found in the 'bloodHead'
      ↳ linkedlist, an error message is displayed.
216  */
217  bool updateBloodPrice(uint32_t id, float newPrice) {
218      if (!isValidBloodGroup(id)) {
219          printf("Error: Invalid blood group id.\n");
220          return false;
221      }
222
223      BloodStock* temp = bloodHead;
224      while (temp != NULL) {
225          if (temp->id == id) {
226              temp->price = newPrice;
227              saveBloodGroups();
228              return true;
229          }
230          temp = temp->next;
231      }
232      return false;
233  }
234
235  /*!
236  * @name loadBloodGroups
237  * @brief Load blood groups from file
238  * @details This function loads the blood groups from the
      ↳ file 'resources/db/blood_data.txt'
239  * to the 'bloodHead' linkedlist.
240  *
241  * @note The file path 'resources/db' is relative to the
      ↳ project root directory.
242  * Make sure that the folder exists also to run the program
      ↳ from the root directory.
243  *
244  * @post The blood groups are loaded to the 'bloodHead'
      ↳ linkedlist.
245  *
246  * @exception fopen() - If the file cannot be opened, an
      ↳ error message is displayed.
247  * @exception malloc() - If the memory allocation for the
      ↳ new blood group fails, an error message is displayed.
248  */
249  void loadBloodGroups(void) {
250      errno = 0;
251      FILE* file = fopen("resources/db/blood_data.txt", "r");
252      if (!file) {
253          if (errno == ENOENT) {
254              initializeBloodGroups();
255              return;
```

```
256     } else {
257         printf("Error opening blood data file: %s\n",
258             ↪ strerror(errno));
259         freeBloodList();
260         return;
261     }
262 }
263
264 while (1) {
265     BloodStock* newBlood =
266         ↪ (BloodStock*)malloc(sizeof(BloodStock));
267     if (!newBlood) {
268         printf("Error allocating memory for blood group:
269             ↪ %s\n", strerror(errno));
270         freeBloodList();
271         fclose(file);
272         return;
273     }
274
275     if (fscanf(file, "%u %s %f %u", &newBlood->id,
276         ↪ newBlood->bloodGroup, &newBlood->price,
277         ↪ &newBlood->quantity) != 4) {
278         free(newBlood);
279         fclose(file);
280         break;
281     }
282
283     newBlood->next = NULL;
284
285     if (bloodHead == NULL) {
286         bloodHead = newBlood;
287     } else {
288         BloodStock* temp = bloodHead;
289         while (temp->next != NULL) {
290             temp = temp->next;
291         }
292         temp->next = newBlood;
293     }
294 }
295
296 fclose(file);
297 }
298
299 /*!
300 * @name isBloodAvailable
301 * @brief Check if blood is available for a specific
302     ↪ transaction type
```

```

297  * @details This function checks if blood is available for a
      ↪ specific transaction type
298  * by traversing the 'bloodHead' linkedlist.
299  *
300  * @param[in] id Blood group id, null if to check for any
      ↪ blood
301  * @param[in] type Transaction type
302  *
303  * @return True if blood is available, False otherwise
304  *
305  * @pre @p id is null or valid
306  * @pre @p type is BUY or SELL
307  * @post If the @p id is found in the 'bloodHead' linkedlist,
308  * the function returns true. Otherwise, it returns false.
309  *
310  * @exception If the @p type is not BUY or SELL, an error
      ↪ message is displayed.
311  * @exception If the @p id is not null and is not valid, an
      ↪ error message is displayed.
312  */
313  bool isBloodAvailable(uint32_t* id, TransactionType type) {
314      if (type != BUY && type != SELL) {
315          printf("Error: Invalid transaction type.\n");
316          return false;
317      }
318
319      if (id != NULL && !isValidBloodGroup(*id)) {
320          printf("Error: Invalid blood group id.\n");
321          return false;
322      }
323
324      BloodStock* temp = bloodHead;
325      while (temp != NULL) {
326          if (type == BUY) {
327              if (id == NULL) {
328                  if (temp->price > 0 && temp->quantity > 0) {
329                      return true;
330                  }
331              } else {
332                  if (temp->id == *id && temp->price > 0 &&
                      ↪ temp->quantity > 0) {
333                      return true;
334                  }
335              }
336          } else {
337              if (id == NULL) {
338                  if (temp->price > 0) {
339                      return true;

```

```
340     }
341     } else {
342         if (temp->id == *id && temp->price > 0) {
343             return true;
344         }
345     }
346 }
347 temp = temp->next;
348 }
349 return false;
350 }
351
352 /*!
353 * @name displayBloodGroups
354 * @brief Display all blood groups
355 * @details This function displays all the blood groups in
356     ↪ the 'availableBloodGroups' array.
357 * @post The available blood groups are displayed.
358 */
359 void displayBloodGroups(void) {
360     for (uint32_t i = 0; i < (sizeof(availableBloodGroups) /
361         ↪ sizeof(availableBloodGroups[0])); i++) {
362         printf("%u. %s\n", i + 1, availableBloodGroups[i]);
363     }
364 }
365
366 /*!
367 * @name displayBloodStocks
368 * @brief Display all blood stocks
369 * @details This function displays all the blood stocks in
370     ↪ the 'bloodHead' linkedlist.
371 * @note If Price or Quantity is not available, it is
372     ↪ displayed as N/A.
373 * @post The blood stocks are displayed.
374 */
375 void displayBloodStocks(void) {
376     BloodStock* temp = bloodHead;
377     if (temp == NULL) {
378         printf("No blood available.\n");
379         return;
380     }
381     printf("\nAvailable Blood:\n");
382     while (temp != NULL) {
383         if (temp->price > 0.0) {
```



```

383         printf("%u. %s, Price: %.2f, Quantity: %u\n",
           ↪ temp->id, temp->bloodGroup, temp->price,
           ↪ temp->quantity);
384     } else {
385         printf("%u. %s, Price: N/A, Quantity: N/A\n",
           ↪ temp->id, temp->bloodGroup);
386     }
387     temp = temp->next;
388 }
389 }
390
391 /*!
392 * @name getBloodGroupById
393 * @brief Get blood group by id
394 * @details This function returns the blood group name by
           ↪ the given id.
395 *
396 * @param[in] id Blood group id
397 *
398 * @return Blood group name or NULL if not found
399 *
400 * @post If the @p id is valid, the function returns the
           ↪ blood group name.
401 *
402 * @exception If the @p id is not valid, an error message is
           ↪ displayed.
403 */
404 char* getBloodGroupById(uint32_t id) {
405     if (!isValidBloodGroup(id)) {
406         printf("Error: Invalid blood group id.\n");
407         return NULL;
408     }
409
410     return availableBloodGroups[id - 1];
411 }
412
413 /*!
414 * @name freeBloodList
415 * @brief Free blood list
416 * @details This function frees the 'bloodHead' linkedlist.
417 *
418 * @post The 'bloodHead' linkedlist is freed.
419 */
420 void freeBloodList(void) {
421     BloodStock* current = bloodHead;
422     while (current != NULL) {
423         BloodStock* temp = current;
424         current = current->next;

```

```
425     free(temp);  
426 }  
427 bloodHead = NULL;  
428 }
```

Listing A.3: blood_manager.c

A.1.4 hospital_manager.c

```
1  /*!  
2   * @file hospital_manager.c  
3   *  
4   * @brief Hospital manager source file  
5   * @details This file contains the implementation of the  
6   *   ↪ functions for the hospital manager module.  
7   *  
8   * @author CrimsonCare Team  
9   * @date 2025-01-18  
10  *  
11  * @copyright  
12  * Copyright (c) 2025 CrimsonCare Team  
13  *  
14  * Permission is hereby granted, free of charge, to any  
15  *   ↪ person obtaining a copy  
16  * of this software and associated documentation files (the  
17  *   ↪ "Software"), to deal  
18  * in the Software without restriction, including without  
19  *   ↪ limitation the rights  
20  * to use, copy, modify, merge, publish, distribute,  
21  *   ↪ sublicense, and/or sell  
22  * copies of the Software, and to permit persons to whom the  
23  *   ↪ Software is  
24  * furnished to do so, subject to the following conditions:  
25  *  
26  * The above copyright notice and this permission notice  
   ↪ shall be included in all  
   ↪ copies or substantial portions of the Software.  
   *  
   * THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY  
   ↪ KIND, EXPRESS OR  
   * IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF  
   ↪ MERCHANTABILITY,  
   * FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN  
   ↪ NO EVENT SHALL THE  
   * AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM,  
   ↪ DAMAGES OR OTHER
```

```
27  * LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR
    ↳ OTHERWISE, ARISING FROM,
28  * OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR
    ↳ OTHER DEALINGS IN THE
29  * SOFTWARE.
30  */
31  #include "../include/hospital_manager.h"
32
33  /*!
34   * @brief Hospital head pointer
35   * @details This pointer is used to track hospital
    ↳ linkedlist on runtime.
36   */
37  Hospital* hospitalHead = NULL;
38
39  /*!
40   * @name loadHospitals
41   * @brief Load hospitals from file
42   * @details This function loads the hospitals from the file
    ↳ 'resources/db/hospitals.txt'
43   * and stores it in the 'hospitalHead' linkedlist.
44   *
45   * @note The file path 'resources/db' is relative to the
    ↳ project root directory.
46   * Make sure that the folder exists also to run the program
    ↳ from the root directory.
47   *
48   * @post If the file is not found, the function does
    ↳ nothing. Otherwise, the hospitals
49   * are loaded from the file and stored in the 'hospitalHead'
    ↳ linkedlist.
50   *
51   * @exception fopen() - If the file cannot be opened, an
    ↳ error message is displayed,
52   * also the function frees the 'hospitalHead' linkedlist.
53   * @exception malloc() - If memory allocation fails, an
    ↳ error message is displayed,
54   * also the function frees the 'hospitalHead' linkedlist.
55   */
56  void loadHospitals(void) {
57      errno = 0;
58      FILE* file = fopen("resources/db/hospitals.txt", "r");
59      if (!file) {
60          if (errno == ENOENT) {
61              return;
62          } else {
63              printf("Error opening hospitals file: %s\n",
    ↳ strerror(errno));
```

```
64         freeHospital();
65         return;
66     }
67 }
68
69 while (1) {
70     Hospital* newHospital =
71         ↪ (Hospital*)malloc(sizeof(Hospital));
72     if (!newHospital) {
73         printf("Error allocating memory for hospital:
74             ↪ %s\n", strerror(errno));
75         fclose(file);
76         freeHospital();
77         return;
78     }
79
80     if (fscanf(file, "%[^|]|%[^|]|%[^\n]\n",
81         ↪ newHospital->code, newHospital->name,
82         ↪ newHospital->location) != 3) {
83         free(newHospital);
84         fclose(file);
85         break;
86     }
87
88     newHospital->next = NULL;
89
90     if (hospitalHead == NULL) {
91         hospitalHead = newHospital;
92     } else {
93         Hospital* temp = hospitalHead;
94         while (temp->next != NULL) {
95             temp = temp->next;
96         }
97         temp->next = newHospital;
98     }
99 }
100
101 fclose(file);
102 }
103
104 /*!
105  * @name saveHospitals
106  * @brief Save hospitals to file
107  * @details This function saves the hospitals to the file
108     ↪ 'resources/db/hospitals.txt'.
109  *
110  * @note The file path 'resources/db' is relative to the
111     ↪ project root directory.
```

```
106  * Make sure that the folder exists also to run the program
    ↪ from the root directory.
107  *
108  * @post The hospitals from the 'hospitalHead' linkedlist
    ↪ are saved to the file.
109  *
110  * @exception fopen() - If the file cannot be opened, an
    ↪ error message is displayed.
111  */
112 void saveHospitals(void) {
113     errno = 0;
114     FILE* file = fopen("resources/db/hospitals.txt", "w");
115     if (!file) {
116         printf("Error opening hospitals file: %s\n",
            ↪ strerror(errno));
117         return;
118     }
119
120     Hospital* temp = hospitalHead;
121     while (temp != NULL) {
122         fprintf(file, "%s|%s|%s\n", temp->code, temp->name,
            ↪ temp->location);
123         temp = temp->next;
124     }
125     fclose(file);
126 }
127
128 /*!
129  * @name addHospital
130  * @brief Add hospital
131  * @details This function adds a new hospital to the
    ↪ 'hospitalHead' linkedlist.
132  *
133  * @param[in] name Hospital name
134  * @param[in] location Hospital location
135  *
136  * @return Hospital code or NULL if hospital is not added
137  *
138  * @note The hospital code is generated by taking the
    ↪ maximum of the first three
139  * characters of the hospital name and appending a random
    ↪ number between 0000 and 9999.
140  *
141  * @pre @p name is not empty and valid
142  * @pre @p location is not empty and valid
143  * @post The hospital is added to the 'hospitalHead'
    ↪ linkedlist.
144  *
```

```
145  * @exception If the @p name or @p location is empty or
146  ↪ invalid, an error message is displayed.
147  * @exception malloc() - If the memory allocation for the
148  ↪ new hospital fails, an error message is displayed.
149  */
150 char* addHospital(const char* name, const char* location) {
151     if (strcmp(name, "") == 0 || strcmp(location, "") == 0) {
152         printf("Error: Hospital name or location cannot be
153             ↪ empty.\n");
154         return NULL;
155     }
156
157     if (containsPipe(name) || containsPipe(location)) {
158         printf("Error: Hospital name or location cannot
159             ↪ contain a pipe character.\n");
160         return NULL;
161     }
162
163     char code[8];
164     char initials[4] = { 0 };
165     int initialCount = 0;
166
167     char nameCopy[100];
168     strncpy(nameCopy, name, sizeof(nameCopy) - 1);
169     nameCopy[sizeof(nameCopy) - 1] = '\0';
170
171     char* token = strtok(nameCopy, " ");
172     while (token != NULL && initialCount < 3) {
173         initials[initialCount++] = token[0];
174         token = strtok(NULL, " ");
175     }
176     initials[initialCount] = '\0';
177
178     bool codeExists;
179     int randomSuffix;
180     do {
181         srand(time(NULL));
182         randomSuffix = rand() % 10000;
183         snprintf(code, sizeof(code), "%s%04d", initials,
184             ↪ randomSuffix);
185
186         codeExists = false;
187         Hospital* temp = hospitalHead;
188         while (temp != NULL) {
189             if (strcmp(temp->code, code) == 0) {
190                 codeExists = true;
191                 break;
192             }
193         }
194     } while (codeExists);
```

```

188         temp = temp->next;
189     }
190 } while (codeExists);
191
192 Hospital* newHospital =
193     ↪ (Hospital*)malloc(sizeof(Hospital));
194 if (!newHospital) {
195     printf("Error allocating memory for hospital: %s\n",
196         ↪ strerror(errno));
197     return NULL;
198 }
199
200 strncpy(newHospital->code, code,
201     ↪ sizeof(newHospital->code) - 1);
202 newHospital->code[sizeof(newHospital->code) - 1] = '\0';
203 strncpy(newHospital->name, name,
204     ↪ sizeof(newHospital->name) - 1);
205 newHospital->name[sizeof(newHospital->name) - 1] = '\0';
206 strncpy(newHospital->location, location,
207     ↪ sizeof(newHospital->location) - 1);
208 newHospital->location[sizeof(newHospital->location) - 1]
209     ↪ = '\0';
210 newHospital->next = NULL;
211
212 if (hospitalHead == NULL) {
213     hospitalHead = newHospital;
214 } else {
215     Hospital* temp = hospitalHead;
216     while (temp->next != NULL) {
217         temp = temp->next;
218     }
219     temp->next = newHospital;
220 }
221
222 saveHospitals();
223 return newHospital->code;
224 }
225
226 /*!
227  * @name validateHospitalCode
228  * @brief Validate hospital code
229  * @details This function validates the given hospital code
230     ↪ by
231  * traversing the 'hospitalHead' linkedlist.
232  *
233  * @param[in] code Hospital code
234  *
235  * @return True if hospital code is valid, False otherwise

```

```

229  *
230  * @pre @p code is not empty and valid
231  * @post If the @p code is found in the 'hospitalHead'
232  *       ↪ linkedlist,
233  *       the function returns true. Otherwise, it returns false.
234  *
235  * @exception If the @p code is empty or invalid, an error
236  *       ↪ message is displayed.
237  */
238 bool validateHospitalCode(const char* code) {
239     if (strcmp(code, "") == 0) {
240         printf("Error: Hospital code cannot be empty.\n");
241         return false;
242     }
243
244     if (containsPipe(code)) {
245         printf("Error: Hospital code cannot contain a pipe
246             ↪ character.\n");
247         return false;
248     }
249
250     Hospital* temp = hospitalHead;
251     while (temp != NULL) {
252         if (strcmp(temp->code, code) == 0) {
253             return true;
254         }
255         temp = temp->next;
256     }
257     return false;
258 }
259
260 /*!
261  * @name deleteHospital
262  * @brief Delete hospital
263  * @details This function deletes the hospital with the
264  *       ↪ given code
265  *       by traversing the 'hospitalHead' linkedlist.
266  *
267  * @param[in] code Hospital code
268  * @param[in] adminUsername Admin username
269  * @param[in] adminPassword Admin password
270  *
271  * @return True if hospital is deleted, False otherwise
272  *
273  * @pre @p code is not empty and valid
274  * @pre @p adminUsername is not empty and valid
275  * @pre @p adminPassword is not empty

```



```
272  * @post The hospital with the given code is deleted from
      ↳ the 'hospitalHead' linkedlist.
273  *
274  * @exception If the @p code is empty or invalid, an error
      ↳ message is displayed.
275  * @exception If the @p adminUsername is empty or invalid or
      ↳ @p adminPassword is empty, an error message is
      ↳ displayed.
276  * @exception If the pair of @p adminUsername and @p
      ↳ adminPassword is invalid, an error message is
      ↳ displayed.
277  * @exception If the hospital with the given code is not
      ↳ found, an error message is displayed.
278  */
279  bool deleteHospital(const char* code, const char*
      ↳ adminUsername, const char* adminPassword) {
280      if (strcmp(adminUsername, "") == 0 ||
          ↳ strcmp(adminPassword, "") == 0) {
281          printf("Error: Admin credentials cannot be
              ↳ empty.\n");
282          return false;
283      }
284
285      if (!checkUsername(adminUsername)) {
286          printf("Error: Invalid admin username. Username can
              ↳ only contain lowercase letters and digits.\n");
287          return false;
288      }
289
290      if (!validateAdmin(adminUsername, adminPassword)) {
291          printf("Error: Invalid admin credentials.\n");
292          return false;
293      }
294
295      if (strcmp(code, "") == 0) {
296          printf("Error: Hospital code cannot be empty.\n");
297          return false;
298      }
299
300      if (containsPipe(code)) {
301          printf("Error: Hospital code cannot contain a pipe
              ↳ character.\n");
302          return false;
303      }
304
305      if (!validateHospitalCode(code)) {
306          printf("Error: Hospital code is invalid.\n");
307          return false;
```

```
308     }
309
310     Hospital* current = hospitalHead;
311     Hospital* prev = NULL;
312     while (current != NULL) {
313         if (strcmp(current->code, code) == 0) {
314             if (prev == NULL) {
315                 hospitalHead = current->next;
316             } else {
317                 prev->next = current->next;
318             }
319             saveHospitals();
320             return true;
321         }
322         prev = current;
323         current = current->next;
324     }
325     return false;
326 }
327
328 /*!
329 * @name getHospitalNameByCode
330 * @brief Get hospital name by code
331 * @details This function gets the hospital name by the
332     ↪ given code
333 * by traversing the 'hospitalHead' linkedlist.
334 * @param[in] code Hospital code
335 * @return Hospital name or NULL if not found
336 * @pre @p code is not empty and valid
337 * @post If the @p code is found in the 'hospitalHead'
338     ↪ linkedlist,
339 * the function returns the hospital name. Otherwise, it
340     ↪ returns NULL.
341 * @exception If the @p code is empty or invalid, an error
342     ↪ message is displayed.
343 */
344 char* getHospitalNameByCode(const char* code) {
345     if (strcmp(code, "") == 0) {
346         printf("Error: Hospital code cannot be empty.\n");
347         return NULL;
348     }
349
350     if (containsPipe(code)) {
```

```
351     printf("Error: Hospital code cannot contain a pipe
352           ↪ character.\n");
353     return NULL;
354 }
355
356 if (!validateHospitalCode(code)) {
357     printf("Error: Hospital code is invalid.\n");
358     return NULL;
359 }
360
361 Hospital* temp = hospitalHead;
362 while (temp != NULL) {
363     if (strcmp(temp->code, code) == 0) {
364         return temp->name;
365     }
366     temp = temp->next;
367 }
368 return NULL;
369 }
370
371 /*!
372  * @name displayHospitals
373  * @brief Display all hospitals
374  * @details This function displays all the hospitals
375  * in the 'hospitalHead' linkedlist by traversing it.
376  *
377  * @post The hospitals in the 'hospitalHead' linkedlist are
378  * ↪ displayed in a formatted manner.
379  */
380 void displayHospitals(void) {
381     Hospital* temp = hospitalHead;
382     if (temp == NULL) {
383         printf("No hospitals registered yet.\n");
384         return;
385     }
386     printf("\nRegistered Hospitals:\n");
387     while (temp != NULL) {
388         printf("\tCode: %s\n"
389               "\tName: %s\n"
390               "\tLocation: %s\n", temp->code, temp->name,
391               ↪ temp->location);
392         temp = temp->next;
393         if (temp != NULL) {
394             printf("\t-----\n");
395         }
396     }
397 }
```

```
396 /*!
397 * @name freeHospital
398 * @brief Free hospital list from memory
399 * @details This function frees the 'hospitalHead' linkedlist
400 * from memory by traversing it.
401 *
402 * @post The 'hospitalHead' linkedlist is freed from memory.
403 */
404 void freeHospital(void) {
405     Hospital* current = hospitalHead;
406     while (current != NULL) {
407         Hospital* temp = current;
408         current = current->next;
409         free(temp);
410     }
411     hospitalHead = NULL;
412 }
```

Listing A.4: hospital_manager.c

A.1.5 transaction_manager.c

```
1 /*!
2 * @file transaction_manager.c
3 *
4 * @brief Transaction manager source file
5 * @details This file contains the implementation of the
6     ↪ functions for the transaction manager module.
7 *
8 * @author CrimsonCare Team
9 * @date 2025-01-18
10 *
11 * @copyright
12 * Copyright (c) 2025 CrimsonCare Team
13 *
14 * Permission is hereby granted, free of charge, to any
15     ↪ person obtaining a copy
16 * of this software and associated documentation files (the
17     ↪ "Software"), to deal
18 * in the Software without restriction, including without
19     ↪ limitation the rights
20 * to use, copy, modify, merge, publish, distribute,
21     ↪ sublicense, and/or sell
22 * copies of the Software, and to permit persons to whom the
23     ↪ Software is
24 * furnished to do so, subject to the following conditions:
25 *
```

```

20  * The above copyright notice and this permission notice
    ↪ shall be included in all
21  * copies or substantial portions of the Software.
22  *
23  * THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY
    ↪ KIND, EXPRESS OR
24  * IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
    ↪ MERCHANTABILITY,
25  * FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN
    ↪ NO EVENT SHALL THE
26  * AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM,
    ↪ DAMAGES OR OTHER
27  * LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR
    ↪ OTHERWISE, ARISING FROM,
28  * OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR
    ↪ OTHER DEALINGS IN THE
29  * SOFTWARE.
30  */
31  #include "../include/blood_manager.h"
32  #include "../include/transaction_manager.h"
33  #include "../include/misc.h"
34  #include "../include/hospital_manager.h"
35
36  /*!
37   * @name logTransaction
38   * @brief Log transaction to file
39   * @details This function logs a transaction to the file
    ↪ 'resources/db/transactions.log'.
40   *
41   * @param[in] type Transaction type
42   * @param[in] name Hospital name
43   * @param[in] bloodId Blood group id
44   * @param[in] quantity Blood quantity
45   * @param[in] date Transaction date
46   * @param[in] token Transaction token
47   *
48   * @return True if transaction is logged, False otherwise
49   *
50   * @note The file path 'resources/db' is relative to the
    ↪ project root directory.
51   * Make sure that the folder exists also to run the program
    ↪ from the root directory.
52   *
53   * @pre @p type is either BUY or SELL
54   * @pre @p name is not empty and valid
55   * @pre @p bloodId is a valid blood group id
56   * @pre @p quantity is greater than 0
57   * @pre @p date is a valid date

```

```

58  * @post The transaction is logged to the file
    ↪ 'resources/db/transactions.log'.
59  *
60  * @exception If the file 'resources/db/transactions.log'
    ↪ cannot be opened,
61  * an error message is displayed.
62  * @exception If the @p type is not BUY or SELL, an error
    ↪ message is displayed.
63  * @exception If the @p name is empty or invalid, an error
    ↪ message is displayed.
64  * @exception If the @p bloodId is not a valid blood group
    ↪ id, an error message is displayed.
65  * @exception If the @p quantity is less than or equal to
    ↪ 0, an error message is displayed.
66  * @exception If the @p date is not a valid date, an error
    ↪ message is displayed.
67  */
68  bool logTransaction(TransactionType type, const char* name,
    ↪ uint32_t bloodId, uint32_t quantity, const char* date,
    ↪ const char* token) {
69      errno = 0;
70      FILE* file = fopen("resources/db/transactions.log", "a");
71      if (!file) {
72          if (errno != ENOENT) {
73              printf("Error opening transaction log file:
    ↪ %s\n", strerror(errno));
74          }
75          return false;
76      }
77
78      if (containsPipe(name)) {
79          printf("Error: Entity name cannot contain a pipe
    ↪ character.\n");
80          return false;
81      }
82
83      if (type != BUY && type != SELL) {
84          printf("Error: Invalid transaction type.\n");
85          return false;
86      }
87
88      if (!isValidBloodGroup(bloodId)) {
89          printf("Error: Invalid blood group.\n");
90          return false;
91      }
92
93      if (strcmp(name, "") == 0 || quantity <= 0) {
94          printf("Error: Invalid transaction parameters.\n");

```

```

95         return false;
96     }
97
98     if (!isValidDate(date)) {
99         printf("Error: Invalid date format.\n");
100         return false;
101     }
102
103     if (token) {
104         fprintf(file, "%s|%s|%u|%u|%s|\n", (type == BUY ?
            ↪ "Buy" : "Sell"), name, bloodId, quantity,
            ↪ date, token);
105     } else {
106         fprintf(file, "%s|%s|%u|%u|%s|\n", (type == BUY ?
            ↪ "Buy" : "Sell"), name, bloodId, quantity,
            ↪ date);
107     }
108
109     fclose(file);
110     return true;
111 }
112
113 /*!
114  * @name addTransaction
115  * @brief Add transaction
116  * @details This function adds a transaction to the
            ↪ 'transactionHead' linkedlist.
117  *
118  * @param[in] type Transaction type
119  * @param[in] name Hospital name
120  * @param[in] bloodId Blood group id
121  * @param[in] quantity Blood quantity
122  *
123  * @return True if transaction is added, False otherwise
124  *
125  * @note For SELL transaction, the user is asked to enter
            ↪ the date of donation,
126  * and a token is generated for the transaction.
127  *
128  * @pre @p name is not empty and valid
129  * @pre @p type is either BUY or SELL
130  * @pre @p quantity is greater than 0
131  * @pre @p bloodId is a valid blood group id
132  * @post The transaction is logged to the file
            ↪ 'resources/db/transactions.log' through
            ↪ 'logTransaction' function.
133  */

```

```
134  * @exception If the @p name is empty or invalid, an error
      ↳ message is displayed.
135  * @exception If the @p type is not BUY or SELL, an error
      ↳ message is displayed.
136  * @exception If the @p quantity is less than or equal to 0,
      ↳ an error message is displayed.
137  * @exception If the @p bloodId is not a valid blood group
      ↳ id, an error message is displayed.
138  * @exception For BUY transaction, if the @p name is not a
      ↳ valid hospital code, an error message is displayed.
139  * @exception For SELL transaction, if the input date is not
      ↳ a valid date, an error message is displayed.
140  */
141  bool addTransaction(TransactionType type, const char* name,
      ↳ uint32_t bloodId, uint32_t quantity) {
142      if (strcmp(name, "") == 0 || quantity <= 0) {
143          printf("Error: Invalid transaction parameters.\n");
144          return false;
145      }
146
147      if (containsPipe(name)) {
148          printf("Error: Entity name cannot contain a pipe
              ↳ character.\n");
149          return false;
150      }
151
152      if (type != BUY && type != SELL) {
153          printf("Error: Invalid transaction type.\n");
154          return false;
155      }
156
157      if (!isBloodAvailable(&bloodId, type)) {
158          printf("No stock available for blood group: %s\n",
              ↳ getBloodGroupId(bloodId));
159          return false;
160      }
161
162      if (type == BUY) {
163          if (!validateHospitalCode(name)) {
164              printf("Error: Invalid hospital code.\n");
165              return false;
166          }
167      }
168
169      char date[MAX_TRANSACTION_DATE_LENGTH];
170      char token[MAX_TRANSACTION_TOKEN_LENGTH] = "";
171
172      if (type == SELL) {
```



```
173     printf("Enter the date and time of donation
        ↳ (YYYY-MM-DD): ");
174     fgets(date, sizeof(date), stdin);
175     date[strcspn(date, "\n")] = 0;
176     if (!isValidDate(date)) {
177         printf("Error: Invalid date format.\n");
178         return false;
179     }
180     formatDate(date);
181 } else {
182     BloodStock* stock = bloodHead;
183     while (stock != NULL) {
184         if (stock->id == bloodId) {
185             if (stock->quantity < quantity) {
186                 printf("Not enough stock for blood
                    ↳ group: %s. Available quantity:
                    ↳ %u\n", getBloodGroupById(bloodId),
                    ↳ stock->quantity);
187                 return false;
188             }
189
190             stock->quantity -= quantity;
191             saveBloodGroups();
192             break;
193         }
194         stock = stock->next;
195     }
196     time_t now = time(NULL);
197     strftime(date, sizeof(date), "%Y-%m-%d",
        ↳ localtime(&now));
198 }
199
200 if (type == SELL) {
201     srand(time(NULL));
202     sprintf(token, "TOKEN_%d", rand() % 10000);
203     printf("Sell token generated for %s: %s\n", name,
        ↳ token);
204 }
205
206 if (!logTransaction(type, name, bloodId, quantity, date,
    ↳ type == SELL ? token : NULL)) {
207     return false;
208 }
209
210 return true;
211 }
212
213 /*!
```

```
214  * @brief Display all transactions
215  * @details This function displays all transactions from the
216  *           ↪ file 'resources/db/transactions.log'.
217  *
218  * @pre The file 'resources/db/transactions.log' exists.
219  * @post All transactions are displayed.
220  *
221  * @exception If the file 'resources/db/transactions.log'
222  *           ↪ cannot be opened, an error message is displayed.
223  */
224 void displayTransactions(void) {
225     errno = 0;
226     FILE* file = fopen("resources/db/transactions.log", "r");
227     if (!file) {
228         if (errno == ENOENT) {
229             printf("No registered transactions found.\n");
230         } else {
231             printf("Error opening transaction log file:
232                 ↪ %s\n", strerror(errno));
233         }
234     }
235     return;
236 }
237
238 char line[256];
239 bool hasLogs = false;
240 bool firstLog = true;
241 char prevLine[256] = { 0 };
242
243 while (fgets(line, sizeof(line), file) != NULL) {
244     char type[MAX_TRANSACTION_TOKEN_LENGTH] = "";
245     char name[MAX_TRANSACTION_TOKEN_LENGTH] = "";
246     uint32_t bloodId = 0;
247     uint32_t quantity = 0;
248     char date[MAX_TRANSACTION_TOKEN_LENGTH] = "";
249     char token[MAX_TRANSACTION_TOKEN_LENGTH] = "";
250
251     if (firstLog) {
252         printf("\nRegistered Transactions:\n");
253         firstLog = false;
254     }
255
256     if (sscanf(line, "[%^|]|[%^|]|%u|%u|[%^|]|[%^|\n]",
257         type,
258         name,
259         &bloodId,
260         &quantity,
261         date,
262         token) >= 5) {
```

```

259         hasLogs = true;
260
261         if (prevLine[0] != '\0') {
262             printf("\t-----\n");
263         }
264
265         printf("\tType: %s\n"
266             "\tEntity: %s\n"
267             "\tBlood Group: %s\n"
268             "\tQuantity: %u\n"
269             "\tDate: %s",
270             type,
271             name,
272             getBloodGroupById(bloodId),
273             quantity,
274             date);
275
276         if (token[0] != '\0') {
277             printf("\n\tToken: %s\n", token);
278         }
279
280         strncpy(prevLine, line, sizeof(prevLine) - 1);
281         prevLine[sizeof(prevLine) - 1] = '\0';
282     }
283 }
284
285 if (!hasLogs) {
286     printf("No registered transactions found.\n");
287 }
288
289 fclose(file);
290 }

```

Listing A.5: transaction_manager.c

A.1.6 misc.c

```

1  /*!
2   * @file misc.c
3   *
4   * @brief Misc source file
5   * @details This file contains the implementation of the
6   *           ↪ functions for the misc module.
7   *
8   * @author CrimsonCare Team
9   * @date 2025-01-22

```

```
9  *
10 * @copyright
11 * Copyright (c) 2025 CrimsonCare Team
12 *
13 * Permission is hereby granted, free of charge, to any
14   ↳ person obtaining a copy
15 * of this software and associated documentation files (the
16   ↳ "Software"), to deal
17 * in the Software without restriction, including without
18   ↳ limitation the rights
19 * to use, copy, modify, merge, publish, distribute,
20   ↳ sublicense, and/or sell
21 * copies of the Software, and to permit persons to whom the
22   ↳ Software is
23 * furnished to do so, subject to the following conditions:
24 *
25 * The above copyright notice and this permission notice
26   ↳ shall be included in all
27 * copies or substantial portions of the Software.
28 *
29 * THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY
30   ↳ KIND, EXPRESS OR
31 * IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
32   ↳ MERCHANTABILITY,
33 * FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN
34   ↳ NO EVENT SHALL THE
35 * AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM,
36   ↳ DAMAGES OR OTHER
37 * LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR
38   ↳ OTHERWISE, ARISING FROM,
39 * OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR
40   ↳ OTHER DEALINGS IN THE
41 * SOFTWARE.
42 */
43 #include "../include/misc.h"
44
45 /*!
46 * @name displayWelcomeMessage
47 * @brief Display welcome message
48 * @details This function displays the welcome message
49 * by reading from the file 'resources/assets/misc/cc.txt'.
50 *
51 * @note The file 'resources/assets/misc/cc.txt' is a text
52   ↳ file
53 * that contains the welcome message.
54 *
55 * @pre The file 'resources/assets/misc/cc.txt' exists.
56 *
```

```
44  * @post The welcome message is displayed.
45  */
46  void displayWelcomeMessage(void) {
47      FILE* file = fopen("resources/assets/misc/cc.txt", "r");
48      if (!file) {
49          return;
50      }
51      char buffer[1024];
52      while (fgets(buffer, sizeof(buffer), file) != NULL) {
53          printf("%s", buffer);
54      }
55      fclose(file);
56  }
57
58  /*!
59   * @name displayUserMenu
60   * @brief Display user menu
61   * @details This function displays the user menu.
62   *
63   * @post The user menu is displayed.
64   */
65  void displayUserMenu(void) {
66      printf("\n--- CrimsonCare Blood Bank Management System
        ↪ (User) ---\n");
67      printf("1. Buy Blood\n");
68      printf("2. Sell Blood\n");
69      printf("3. Display Blood Stocks\n");
70      printf("4. Admin Panel\n");
71      printf("5. Exit\n");
72      printf("Select an option: ");
73  }
74
75  /*!
76   * @name displayAdminMenu
77   * @brief Display admin menu
78   * @details This function displays the admin menu.
79   *
80   * @post The admin menu is displayed.
81   */
82  void displayAdminMenu(void) {
83      printf("\n--- CrimsonCare Blood Bank Management System
        ↪ (Admin) ---\n");
84      printf("1. Add Hospital\n");
85      printf("2. Update Blood Quantity\n");
86      printf("3. Update Blood Price\n");
87      printf("4. Change Admin Password\n");
88      printf("5. Add Admin\n");
89      printf("6. Delete Admin\n");
```

```
90     printf("7. Delete Hospital\n");
91     printf("8. Display Admins\n");
92     printf("9. Display Hospitals\n");
93     printf("10. Display Blood Stocks\n");
94     printf("11. Display Transactions\n");
95     printf("12. Exit\n");
96     printf("Select an option: ");
97 }
98
99 /*!
100 * @name clearInputBuffer
101 * @brief Clear input buffer
102 * @details This function clears the input buffer
103 * by reading until a newline character is encountered.
104 *
105 * @post The input buffer is cleared.
106 */
107 void clearInputBuffer(void) {
108     int c;
109     while ((c = getchar()) != '\n' && c != EOF);
110 }
111
112 /*!
113 * @name checkUsername
114 * @brief Check if username is valid
115 * @details This function checks if a username is valid.
116 *
117 * @param[in] str Username to check
118 *
119 * @return True if username is valid, False otherwise
120 *
121 * @note Username can only contain lowercase letters and
122 ↪ digits.
123 *
124 * @pre @p str is not empty
125 * @post If the @p str is valid, the function returns true.
126 * Otherwise, it returns false.
127 */
128 bool checkUsername(const char* str) {
129     while (*str) {
130         if ((*str >= 'a' && *str <= 'z') && !(*str >= '0'
131             ↪ && *str <= '9')) {
132             return false;
133         }
134         str++;
135     }
136     return true;
137 }
```

```
136
137 /*!
138 * @name containsPipe
139 * @brief Check if string contains pipe
140 * @details This function checks if a string contains a pipe
141     ↪ character.
142 * @param[in] str String to check
143 * @return True if string contains pipe, False otherwise
144 * @pre @p str is not empty
145 * @post If the @p str contains a pipe character, the
146     ↪ function returns true.
147 * Otherwise, it returns false.
148 */
149
150 bool containsPipe(const char* str) {
151     while (*str) {
152         if (*str == '|') {
153             return true;
154         }
155         str++;
156     }
157     return false;
158 }
159
160 /*!
161 * @name getPassword
162 * @brief Get password
163 * @details This function gets the password from the user
164 * by reading from the standard input.
165 * @param[in,out] password Password
166 * @param[in] size Password size
167 * @post Updates the @p password with the user's input
168 * through the pointer @p password.
169 */
170
171 void getPassword(char* password, size_t size) {
172 #ifdef _WIN32
173     size_t i = 0;
174     char ch;
175     while (i < size - 1) {
176
177         ch = getch();
178
179         if (ch == '\r') {
180
181
```

```
182         break;
183     } else if (ch == '\b') {
184
185         if (i > 0) {
186             i--;
187             printf("\b \b");
188
189         }
190     } else {
191         password[i++] = ch;
192         printf("*");
193
194     }
195 }
196 password[i] = '\0';
197
198 printf("\n");
199 #else
200
201 struct termios oldt, newt;
202 tcgetattr(STDIN_FILENO, &oldt);
203
204 newt = oldt;
205
206 newt.c_lflag &= ~(ECHO);
207
208 tcsetattr(STDIN_FILENO, TCSANOW, &newt);
209
210 fgets(password, size);
211 password[strcspn(password, "\n")] = 0;
212
213 tcsetattr(STDIN_FILENO, TCSANOW, &oldt);
214
215 printf("\n");
216 #endif
217 }
218
219 /*!
220  * @name isLeapYear
221  * @brief Check if year is leap year
222  * @details This function checks if a year is a leap year.
223  *
224  * @param[in] year Year to check
225  *
226  * @return True if year is leap year, False otherwise
227  */
228 bool isLeapYear(int year) {
```



```
229     return (year % 4 == 0 && year % 100 != 0) || (year % 400
        ↪ == 0);
230 }
231
232 /*!
233  * @name isValidDate
234  * @brief Check if date is valid
235  * @details This function checks if a date is valid.
236  *
237  * @param[in] date Date to check
238  *
239  * @return True if date is valid, False otherwise
240  *
241  * @pre @p date is not empty
242  * @post If the @p date is valid, the function returns true.
243  * Otherwise, it returns false.
244  *
245  * @exception If the @p date is empty, an error message is
        ↪ displayed.
246  * @exception If the @p date is invalid, an error message is
        ↪ displayed.
247  */
248 bool isValidDate(const char* date) {
249     if (strcmp(date, "") == 0) {
250         printf("Error: Date cannot be empty.\n");
251         return false;
252     }
253
254     int year, month, day;
255
256     if (strlen(date) < 8 || strlen(date) > 10) {
257         printf("Error: Invalid date format.\n");
258         return false;
259     }
260
261     if (sscanf(date, "%d-%d-%d", &year, &month, &day) != 3) {
262         printf("Error: Invalid date format.\n");
263         return false;
264     }
265
266     if (month < 1 || month > 12) {
267         printf("Error: Invalid month.\n");
268         return false;
269     }
270
271     int daysInMonth[] = { 31, 28 + (int)isLeapYear(year),
        ↪ 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };
272
```

```
273     if (day < 1 || day > daysInMonth[month - 1]) {
274         printf("Error: Invalid day.\n");
275         return false;
276     }
277
278     return true;
279 }
280
281 /*!
282 * @name formatDate
283 * @brief Format date to yyyy-mm-dd
284 * @details This function formats a date string to the
285     ↪ format yyyy-mm-dd.
286 * @param[in,out] date Date to format
287 * @pre @p date is not empty
288 * @post The @p date is formatted to the format yyyy-mm-dd
289     ↪ and updates the @p date through the pointer @p date.
290 * @exception If the @p date is empty, an error message is
291     ↪ displayed.
292 * @exception If the @p date is invalid, an error message is
293     ↪ displayed.
294 */
295 void formatDate(char* date) {
296     if (strcmp(date, "") == 0) {
297         printf("Error: Date cannot be empty.\n");
298         return;
299     }
300
301     if (!isValidDate(date)) {
302         printf("Error: Invalid date format.\n");
303         return;
304     }
305
306     int year, month, day;
307     sscanf(date, "%d-%d-%d", &year, &month, &day);
308     sprintf(date, "%04d-%02d-%02d", year, month, day);
309 }
```

Listing A.6: misc.c

A.2 Header Files

A.2.1 admin_manager.h

```
1  /*!
2  * @file admin_manager.h
3  *
4  * @brief Admin manager header file
5  * @details This file contains the declarations of the
6      ↪ functions and structures for the admin manager module.
7  *
8  * @author CrimsonCare Team
9  * @date 2025-01-18
10 *
11 * @copyright
12 * Copyright (c) 2025 CrimsonCare Team
13 *
14 * Permission is hereby granted, free of charge, to any
15     ↪ person obtaining a copy
16 * of this software and associated documentation files (the
17     ↪ "Software"), to deal
18 * in the Software without restriction, including without
19     ↪ limitation the rights
20 * to use, copy, modify, merge, publish, distribute,
21     ↪ sublicense, and/or sell
22 * copies of the Software, and to permit persons to whom the
23     ↪ Software is
24 * furnished to do so, subject to the following conditions:
25 *
26 * The above copyright notice and this permission notice
27     ↪ shall be included in all
28 * copies or substantial portions of the Software.
29 *
30 * THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY
31     ↪ KIND, EXPRESS OR
32 * IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
33     ↪ MERCHANTABILITY,
34 * FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN
35     ↪ NO EVENT SHALL THE
36 * AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM,
37     ↪ DAMAGES OR OTHER
38 * LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR
39     ↪ OTHERWISE, ARISING FROM,
40 * OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR
41     ↪ OTHER DEALINGS IN THE
42 * SOFTWARE.
43 */
```

```

31 #ifndef ADMIN_MANAGER_H
32 #define ADMIN_MANAGER_H
33
34 #include <stdbool.h>
35 #include <stdint.h>
36 #include <stdio.h>
37 #include <stdlib.h>
38 #include <string.h>
39 #include <errno.h>
40
41 /*!
42 * @def MAX_USERNAME_LENGTH
43 * @brief Maximum username length
44 * @details This macro defines the maximum length of the
45     ↪ username.
46 */
47 #define MAX_USERNAME_LENGTH 20
48
49 /*!
50 * @def MAX_PASSWORD_LENGTH
51 * @brief Maximum password length
52 * @details This macro defines the maximum length of the
53     ↪ password.
54 */
55 #define MAX_PASSWORD_LENGTH 20
56
57 /*!
58 * @struct Admin
59 * @brief Admin structure
60 * @details This structure represents an admin user in
61     ↪ the system.
62 */
63 typedef struct Admin {
64     char username[MAX_USERNAME_LENGTH]; /*!< Admin username
65         ↪ */
66     char password[MAX_PASSWORD_LENGTH]; /*!< Admin password
67         ↪ */
68     struct Admin* next; /*!< Next admin */
69 } Admin;
70
71 /*!
72 * @name saveAdminCredentials
73 * @brief Save admin credentials to file
74 * @details This function saves the linkedlist data
75 * from 'adminHead' to the file
76     ↪ 'resources/db/admin_credentials.dat'.
77 * '.dat' is used to store credentials in binary format for
78     ↪ surface level security.

```

```
72  *
73  * @note The file path 'resources/db' is relative to the
    ↪ project root directory.
74  * Make sure that the folder exists also to run the program
    ↪ from the root directory.
75  *
76  * @post The linkedlist data is saved to the file.
77  *
78  * @exception fopen() - If the file cannot be opened, an
    ↪ error message is displayed.
79  * @exception fwrite() - If the file cannot be written, an
    ↪ error message is displayed.
80  */
81 void saveAdminCredentials(void);
82
83 /*!
84  * @name loadAdminCredentials
85  * @brief Load admin credentials from file
86  * @details This function loads the admin credentials
87  * from the file 'resources/db/admin_credentials.dat' and
88  * stores it in the 'adminHead' linkedlist. If file is not
    ↪ found,
89  * it creates a new admin with default credentials and
    ↪ stores it in the file.
90  *
91  * @note The file path 'resources/db' is relative to the
    ↪ project root directory.
92  * Make sure that the folder exists also to run the program
    ↪ from the root directory.
93  *
94  * @post If the file is not found, a new admin is created
    ↪ with default credentials
95  * and stored in the file. If the file is found, the admin
    ↪ credentials are loaded
96  * from the file and stored in the 'adminHead' linkedlist.
97  *
98  * @exception fopen() - If the file cannot be opened, an
    ↪ error message is displayed.
99  * @exception malloc() - If memory allocation fails, an
    ↪ error message is displayed.
100 */
101 void loadAdminCredentials(void);
102
103 /*!
104  * @name adminExists
105  * @brief Check if admin exists
106  * @details This function traverses the 'adminHead'
    ↪ linkedlist
```

```

107  * and checks if the username exists in the list.
108  *
109  * @param[in] username Admin username
110  *
111  * @return True if admin exists, False otherwise
112  *
113  * @pre @p username is not empty and valid
114  * @post If the @p username is found in the linkedlist,
115  * the function returns true, otherwise false.
116  *
117  * @exception If the @p username is empty, an error message
118  *    → is displayed.
119  * @exception If the @p username is invalid, an error
120  *    → message is displayed.
121  */
122  bool adminExists(const char* username);
123
124  /*!
125  * @name validateAdmin
126  * @brief Validate admin credentials
127  * @details This function traverses the 'adminHead'
128  *    → linkedlist
129  * and checks if the pair of username and password match.
130  *
131  * @param[in] username Admin username
132  * @param[in] password Admin password
133  *
134  * @return True if credentials are valid, False otherwise
135  *
136  * @pre @p username and @p password are not empty and valid
137  * @post If the pair of @p username and @p password are
138  *    → found in the linkedlist,
139  * the function returns true, otherwise false.
140  *
141  * @exception If the @p username or @p password is empty, an
142  *    → error message is displayed.
143  * @exception If the @p username is invalid, an error
144  *    → message is displayed.
145  */
146  bool validateAdmin(const char* username, const char*
147  *    → password);
148
149  /*!
150  * @name addAdmin
151  * @brief Add admin
152  * @details This function adds a new admin to the
153  *    → 'adminHead' linkedlist

```

```

146  * and saves the updated linkedlist to the file
    ↪ 'resources/db/admin_credentials.dat'.
147  *
148  * @param[in] username Admin username
149  * @param[in] password Admin password
150  * @param[in] currentAdminUsername Current admin username
151  * @param[in] currentAdminPassword Current admin password
152  *
153  * @return True if admin is added, False otherwise
154  *
155  * @note The file path 'resources/db' is relative to the
    ↪ project root directory.
156  * Make sure that the folder exists also to run the program
    ↪ from the root directory.
157  *
158  * @pre @p username and @p password are not empty
159  * @pre @p currentAdminUsername and @p currentAdminPassword
    ↪ are not empty
160  * @pre @p username and @p currentAdminUsername are valid
161  * @post If the @p username and @p password are not found in
    ↪ the linkedlist,
162  * the function adds the new admin to the linkedlist and
    ↪ saves the updated linkedlist to the file.
163  *
164  * @exception If the @p username or @p password is empty, an
    ↪ error message is displayed.
165  * @exception If the @p username or @p currentAdminUsername
    ↪ is invalid, an error message is displayed.
166  * @exception If the pair of @p currentAdminUsername and @p
    ↪ currentAdminPassword is invalid,
167  * means that the current admin credentials are not valid,
    ↪ an error message is displayed.
168  * @exception If the @p username already exists, an error
    ↪ message is displayed.
169  * @exception malloc() - If memory allocation fails, an
    ↪ error message is displayed.
170  */
171  bool addAdmin(const char* username, const char* password,
    ↪ const char* currentAdminUsername, const char*
    ↪ currentAdminPassword);
172
173  /*!
174  * @name deleteAdmin
175  * @brief Delete admin
176  * @details This function deletes an admin from the
    ↪ 'adminHead' linkedlist
177  * and saves the updated linkedlist.
178  *

```

```

179  * @param[in] username Admin username
180  * @param[in] currentAdminUsername Current admin username
181  * @param[in] currentAdminPassword Current admin password
182  *
183  * @return True if admin is deleted, False otherwise
184  *
185  * @pre @p username is not empty
186  * @pre @p currentAdminUsername and @p currentAdminPassword
187  *       ↪ are not empty
188  * @pre @p username and @p currentAdminUsername are valid
189  * @post If the @p username is found in the linkedlist,
190  * the function deletes the admin from the linkedlist and
191  *       ↪ saves the updated linkedlist to the file.
192  *
193  * @exception If the pair of @p currentAdminUsername and @p
194  *       ↪ currentAdminPassword is invalid,
195  * means that the current admin credentials are not valid,
196  *       ↪ an error message is displayed.
197  * @exception If the @p username does not exist, an error
198  *       ↪ message is displayed.
199  * @exception If the @p username is the same as the current
200  *       ↪ admin username, an error message is displayed.
201  * @exception If the @p username or @p currentAdminUsername
202  *       ↪ is invalid, an error message is displayed.
203  */
204  bool deleteAdmin(const char* username, const char*
205  *       ↪ currentAdminUsername, const char*
206  *       ↪ currentAdminPassword);
207
208  /*!
209  * @name changeAdminPassword
210  * @brief Change admin password
211  * @details This function changes the password of an admin
212  *       ↪ in the 'adminHead' linkedlist
213  * and saves the updated linkedlist.
214  *
215  * @param[in] username Admin username
216  * @param[in] oldPassword Old password
217  * @param[in] newPassword New password
218  *
219  * @return True if password is changed, False otherwise
220  *
221  * @pre @p username and @p oldPassword are not empty
222  * @pre @p username is valid
223  * @pre @p newPassword is not empty
224  * @post If the pair of @p username and @p oldPassword are
225  *       ↪ found in the linkedlist,

```



```

215  * the function changes the password of the admin and saves
      ↳ the updated linkedlist.
216  *
217  * @exception If the @p username or @p oldPassword is empty,
      ↳ an error message is displayed.
218  * @exception If the pair of @p username and @p oldPassword
      ↳ is not found in the linkedlist,
219  * an error message is displayed.
220  * @exception If the @p username is invalid, an error
      ↳ message is displayed.
221  */
222  bool changeAdminPassword(const char* username, const char*
      ↳ oldPassword, const char* newPassword);
223
224  /*!
225  * @name displayAdmin
226  * @brief Display all admins
227  * @details This function displays all admins in the
      ↳ 'adminHead' linkedlist.
228  *
229  * @post If the 'adminHead' linkedlist is not empty,
230  * the function displays all admins in the linkedlist.
231  */
232  void displayAdmin(void);
233
234  /*!
235  * @name freeAdmin
236  * @brief Free admin list
237  * @details This function frees the memory allocated for the
      ↳ 'adminHead' linkedlist.
238  *
239  * @post The memory allocated for the 'adminHead' linkedlist
      ↳ is freed.
240  */
241  void freeAdmin(void);
242
243  #endif

```

Listing A.7: admin_manager.h

A.2.2 blood_manager.h

```

1  /*!
2  * @file blood_manager.h
3  *
4  * @brief Blood manager header file

```

```
5  * @details This file contains the declarations of the
   * ↪ functions and structures for the blood manager module.
6  *
7  * @author CrimsonCare Team
8  * @date 2025-01-18
9  *
10 * @copyright
11 * Copyright (c) 2025 CrimsonCare Team
12 *
13 * Permission is hereby granted, free of charge, to any
   * ↪ person obtaining a copy
14 * of this software and associated documentation files (the
   * ↪ "Software"), to deal
15 * in the Software without restriction, including without
   * ↪ limitation the rights
16 * to use, copy, modify, merge, publish, distribute,
   * ↪ sublicense, and/or sell
17 * copies of the Software, and to permit persons to whom the
   * ↪ Software is
18 * furnished to do so, subject to the following conditions:
19 *
20 * The above copyright notice and this permission notice
   * ↪ shall be included in all
21 * copies or substantial portions of the Software.
22 *
23 * THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY
   * ↪ KIND, EXPRESS OR
24 * IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
   * ↪ MERCHANTABILITY,
25 * FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN
   * ↪ NO EVENT SHALL THE
26 * AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM,
   * ↪ DAMAGES OR OTHER
27 * LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR
   * ↪ OTHERWISE, ARISING FROM,
28 * OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR
   * ↪ OTHER DEALINGS IN THE
29 * SOFTWARE.
30 */
31 #ifndef BLOOD_MANAGER_H
32 #define BLOOD_MANAGER_H
33
34 #include <stdbool.h>
35 #include <stdint.h>
36 #include <stdio.h>
37 #include <stdlib.h>
38 #include <string.h>
39 #include <errno.h>
```

```

40 #include "misc.h"
41
42 #include "transaction_manager.h"
43
44 /*!
45 * @def BLOOD_GROUP_NAME_LENGTH
46 * @brief Blood group name length
47 * @details This macro defines the length of the blood
48     ↪ group name.
49 */
50 #define BLOOD_GROUP_NAME_LENGTH 4
51
52 /*!
53 * @struct BloodStock
54 * @brief Blood stock structure
55 * @details This structure represents a blood stock in the
56     ↪ system.
57 */
58 typedef struct BloodStock {
59     float price; /*!< Blood price */
60     uint32_t id; /*!< Blood group id */
61     uint32_t quantity; /*!< Blood quantity */
62     char bloodGroup[BLOOD_GROUP_NAME_LENGTH]; /*!< Blood
63             ↪ group name */
64     struct BloodStock* next; /*!< Next blood stock */
65 } BloodStock;
66
67 /*!
68 * @brief Globally exposed blood stock head pointer
69 * @details This pointer is used to track blood stock
70     ↪ linkedlist on runtime.
71 */
72 extern BloodStock* bloodHead;
73
74 /*!
75 * @name isValidBloodGroup
76 * @brief Check if blood group is valid
77 * @details This function checks if the given blood group id
78     ↪ is valid
79 * by checking the size of the 'availableBloodGroups' array.
80 *
81 * @param[in] id Blood group id
82 *
83 * @return True if blood group is valid, False otherwise
84 *
85 * @post If the @p id is within the range of the
86     ↪ 'availableBloodGroups' array,
87 * the function returns true. Otherwise, it returns false.

```

```

82  */
83  bool isValidBloodGroup(uint32_t id);
84
85  /*!
86   * @name addBloodGroup
87   * @brief Add blood group
88   * @details This function adds a new blood group to the
89   *           ↪ 'bloodHead' linkedlist.
90   *
91   * @param[in] id Blood group id
92   * @param[in] bloodGroup Blood group name
93   * @param[in] price Blood group price
94   * @param[in] quantity Blood group quantity
95   *
96   * @return True if blood group is added, False otherwise
97   *
98   * @pre @p id is valid
99   * @pre @p bloodGroup is not empty
100  * @post Updates the blood stock in the 'bloodHead'
101  *        ↪ linkedlist.
102  *
103  * @exception If the @p bloodGroup is empty, an error
104  *            ↪ message is displayed.
105  * @exception If the @p id is not valid, an error message is
106  *            ↪ displayed.
107  * @exception malloc() - If the memory allocation for the
108  *            ↪ new blood group fails, an error message is displayed.
109  */
110  bool addBloodGroup(uint32_t id, const char* bloodGroup,
111  ↪ float price, uint32_t quantity);
112
113  /*!
114   * @name initializeBloodGroups
115   * @brief Initialize blood groups
116   * @details This function helps to initialize the default
117   *           ↪ blood groups
118   * to the 'bloodHead' linkedlist.
119   *
120   * @post The blood groups are added to the 'bloodHead'
121   *       ↪ linkedlist.
122   *
123   * @exception If adding blood group fails, an error message
124   *           ↪ is displayed.
125   */
126  void initializeBloodGroups(void);
127
128  /*!
129   * @name saveBloodGroups

```

```
121  * @brief Save blood groups to file
122  * @details This function saves the linkedlist data
123  * from 'bloodHead' to the file
124  *   ↳ 'resources/db/blood_data.txt'.
125  *
126  * @note The file path 'resources/db' is relative to the
127  *   ↳ project root directory.
128  * Make sure that the folder exists also to run the program
129  *   ↳ from the root directory.
130  *
131  * @post The linkedlist data is saved to the file.
132  *
133  * @exception fopen() - If the file cannot be opened, an
134  *   ↳ error message is displayed.
135  */
136 void saveBloodGroups(void);
137
138 /*!
139  * @name updateBloodQuantity
140  * @brief Update blood quantity
141  * @details This function updates the blood quantity of the
142  *   ↳ given blood group id
143  * by traversing the 'bloodHead' linkedlist.
144  *
145  * @param[in] id Blood group id
146  * @param[in] newQuantity New quantity
147  *
148  * @return True if blood quantity is updated, False otherwise
149  *
150  * @post If the @p id is found in the 'bloodHead' linkedlist,
151  * the function updates the blood quantity and saves the
152  *   ↳ updated linkedlist.
153  *
154  * @exception If the @p id is not found in the 'bloodHead'
155  *   ↳ linkedlist, an error message is displayed.
156  */
157 bool updateBloodQuantity(uint32_t id, uint32_t newQuantity);
158
159 /*!
160  * @name updateBloodPrice
161  * @brief Update blood price
162  * @details This function updates the blood price of the
163  *   ↳ given blood group id
164  * by traversing the 'bloodHead' linkedlist.
165  *
166  * @param[in] id Blood group id
167  * @param[in] newPrice New price
168  *
```

```

161  * @return True if blood price is updated, False otherwise
162  *
163  * @post If the @p id is found in the 'bloodHead' linkedlist,
164  * the function updates the blood price and saves the
165  *   ↪ updated linkedlist.
166  *
167  * @exception If the @p id is not found in the 'bloodHead'
168  *   ↪ linkedlist, an error message is displayed.
169  */
170 bool updateBloodPrice(uint32_t id, float newPrice);
171
172 /*!
173  * @name loadBloodGroups
174  * @brief Load blood groups from file
175  * @details This function loads the blood groups from the
176  *   ↪ file 'resources/db/blood_data.txt'
177  * to the 'bloodHead' linkedlist.
178  *
179  * @note The file path 'resources/db' is relative to the
180  *   ↪ project root directory.
181  * Make sure that the folder exists also to run the program
182  *   ↪ from the root directory.
183  *
184  * @post The blood groups are loaded to the 'bloodHead'
185  *   ↪ linkedlist.
186  *
187  * @exception fopen() - If the file cannot be opened, an
188  *   ↪ error message is displayed.
189  * @exception malloc() - If the memory allocation for the
190  *   ↪ new blood group fails, an error message is displayed.
191  */
192 void loadBloodGroups(void);
193
194 /*!
195  * @name isBloodAvailable
196  * @brief Check if blood is available for a specific
197  *   ↪ transaction type
198  * @details This function checks if blood is available for a
199  *   ↪ specific transaction type
200  * by traversing the 'bloodHead' linkedlist.
201  *
202  * @param[in] id Blood group id, null if to check for any
203  *   ↪ blood
204  * @param[in] type Transaction type
205  *
206  * @return True if blood is available, False otherwise
207  *
208  * @pre @p id is null or valid

```

```
198  * @pre @p type is BUY or SELL
199  * @post If the @p id is found in the 'bloodHead' linkedlist,
200  * the function returns true. Otherwise, it returns false.
201  *
202  * @exception If the @p type is not BUY or SELL, an error
203  *   ↳ message is displayed.
204  * @exception If the @p id is not null and is not valid, an
205  *   ↳ error message is displayed.
206  */
207 bool isBloodAvailable(uint32_t* id, TransactionType type);
208
209 /*!
210  * @name displayBloodGroups
211  * @brief Display all blood groups
212  * @details This function displays all the blood groups in
213  *   ↳ the 'availableBloodGroups' array.
214  *
215  * @post The available blood groups are displayed.
216  */
217 void displayBloodGroups(void);
218
219 /*!
220  * @name displayBloodStocks
221  * @brief Display all blood stocks
222  * @details This function displays all the blood stocks in
223  *   ↳ the 'bloodHead' linkedlist.
224  *
225  * @note If Price or Quantity is not available, it is
226  *   ↳ displayed as N/A.
227  *
228  * @post The blood stocks are displayed.
229  */
230 void displayBloodStocks(void);
231
232 /*!
233  * @name getBloodGroupById
234  * @brief Get blood group by id
235  * @details This function returns the blood group name by
236  *   ↳ the given id.
237  *
238  * @param[in] id Blood group id
239  *
240  * @return Blood group name or NULL if not found
241  *
242  * @post If the @p id is valid, the function returns the
243  *   ↳ blood group name.
244  *
```

```
238  * @exception If the @p id is not valid, an error message is
      ↪ displayed.
239  */
240  char* getBloodGroupById(uint32_t id);
241
242  /*!
243   * @name freeBloodList
244   * @brief Free blood list
245   * @details This function frees the 'bloodHead' linkedlist.
246   *
247   * @post The 'bloodHead' linkedlist is freed.
248   */
249  void freeBloodList(void);
250
251  #endif
```

Listing A.8: blood_manager.h

A.2.3 hospital_manager.h

```
1  /*!
2   * @file hospital_manager.h
3   *
4   * @brief Hospital manager header file
5   * @details This file contains the declarations of the
      ↪ functions and structures for the hospital manager
      ↪ module.
6   *
7   * @author CrimsonCare Team
8   * @date 2025-01-18
9   *
10  * @copyright
11  * Copyright (c) 2025 CrimsonCare Team
12  *
13  * Permission is hereby granted, free of charge, to any
      ↪ person obtaining a copy
14  * of this software and associated documentation files (the
      ↪ "Software"), to deal
15  * in the Software without restriction, including without
      ↪ limitation the rights
16  * to use, copy, modify, merge, publish, distribute,
      ↪ sublicense, and/or sell
17  * copies of the Software, and to permit persons to whom the
      ↪ Software is
18  * furnished to do so, subject to the following conditions:
19  *
```



```
20  * The above copyright notice and this permission notice
    ↪ shall be included in all
21  * copies or substantial portions of the Software.
22  *
23  * THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY
    ↪ KIND, EXPRESS OR
24  * IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
    ↪ MERCHANTABILITY,
25  * FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN
    ↪ NO EVENT SHALL THE
26  * AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM,
    ↪ DAMAGES OR OTHER
27  * LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR
    ↪ OTHERWISE, ARISING FROM,
28  * OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR
    ↪ OTHER DEALINGS IN THE
29  * SOFTWARE.
30  */
31  #ifndef HOSPITAL_MANAGER_H
32  #define HOSPITAL_MANAGER_H
33
34  #include <stdio.h>
35  #include <stdlib.h>
36  #include <string.h>
37  #include <time.h>
38  #include <stdbool.h>
39  #include <errno.h>
40
41  #include "admin_manager.h"
42
43  /*!
44   * @def MAX_HOSPITAL_NAME_LENGTH
45   * @brief Maximum hospital name length
46   * @details This macro defines the maximum length of the
    ↪ hospital name.
47   */
48  #define MAX_HOSPITAL_NAME_LENGTH 100
49
50  /*!
51   * @def MAX_HOSPITAL_LOCATION_LENGTH
52   * @brief Maximum hospital location length
53   * @details This macro defines the maximum length of the
    ↪ hospital location.
54   */
55  #define MAX_HOSPITAL_LOCATION_LENGTH 100
56
57  /*!
58   * @def MAX_HOSPITAL_CODE_LENGTH
```

```
59  * @brief Maximum hospital code length
60  * @details This macro defines the maximum length of the
    ↪ hospital code.
61  */
62  #define MAX_HOSPITAL_CODE_LENGTH 8
63
64  /*!
65   * @struct Hospital
66   * @brief Hospital structure
67   * @details This structure represents a hospital in the
    ↪ system.
68   */
69  typedef struct Hospital {
70      char name[MAX_HOSPITAL_NAME_LENGTH]; /*!< Hospital name
    ↪ */
71      char location[MAX_HOSPITAL_LOCATION_LENGTH]; /*!<
    ↪ Hospital location */
72      char code[MAX_HOSPITAL_CODE_LENGTH]; /*!< Hospital code
    ↪ */
73      struct Hospital* next; /*!< Next hospital */
74  } Hospital;
75
76  /*!
77   * @name loadHospitals
78   * @brief Load hospitals from file
79   * @details This function loads the hospitals from the file
    ↪ 'resources/db/hospitals.txt'
80   * and stores it in the 'hospitalHead' linkedlist.
81   *
82   * @note The file path 'resources/db' is relative to the
    ↪ project root directory.
83   * Make sure that the folder exists also to run the program
    ↪ from the root directory.
84   *
85   * @post If the file is not found, the function does
    ↪ nothing. Otherwise, the hospitals
86   * are loaded from the file and stored in the 'hospitalHead'
    ↪ linkedlist.
87   *
88   * @exception fopen() - If the file cannot be opened, an
    ↪ error message is displayed,
89   * also the function frees the 'hospitalHead' linkedlist.
90   * @exception malloc() - If memory allocation fails, an
    ↪ error message is displayed,
91   * also the function frees the 'hospitalHead' linkedlist.
92   */
93  void loadHospitals(void);
94
```

```
95  /*!
96  * @name saveHospitals
97  * @brief Save hospitals to file
98  * @details This function saves the hospitals to the file
99      ↪ 'resources/db/hospitals.txt'.
100 * 
101 * @note The file path 'resources/db' is relative to the
102     ↪ project root directory.
103 * Make sure that the folder exists also to run the program
104     ↪ from the root directory.
105 * 
106 * @post The hospitals from the 'hospitalHead' linkedlist
107     ↪ are saved to the file.
108 * 
109 * @exception fopen() - If the file cannot be opened, an
110     ↪ error message is displayed.
111 */
112 void saveHospitals(void);
113
114 /*!
115 * @name addHospital
116 * @brief Add hospital
117 * @details This function adds a new hospital to the
118     ↪ 'hospitalHead' linkedlist.
119 * 
120 * @param[in] name Hospital name
121 * @param[in] location Hospital location
122 * 
123 * @return Hospital code or NULL if hospital is not added
124 * 
125 * @note The hospital code is generated by taking the
126     ↪ maximum of the first three
127 * characters of the hospital name and appending a random
128     ↪ number between 0000 and 9999.
129 * 
130 * @pre @p name is not empty and valid
131 * @pre @p location is not empty and valid
132 * @post The hospital is added to the 'hospitalHead'
133     ↪ linkedlist.
134 * 
135 * @exception If the @p name or @p location is empty or
136     ↪ invalid, an error message is displayed.
137 * @exception malloc() - If the memory allocation for the
138     ↪ new hospital fails, an error message is displayed.
139 */
140 char* addHospital(const char* name, const char* location);
141
142 /*!
```

```
132  * @name validateHospitalCode
133  * @brief Validate hospital code
134  * @details This function validates the given hospital code
135  *   ↪ by
136  *   traversing the 'hospitalHead' linkedlist.
137  *
138  * @param[in] code Hospital code
139  *
140  * @return True if hospital code is valid, False otherwise
141  *
142  * @pre @p code is not empty and valid
143  * @post If the @p code is found in the 'hospitalHead'
144  *   ↪ linkedlist,
145  *   the function returns true. Otherwise, it returns false.
146  *
147  * @exception If the @p code is empty or invalid, an error
148  *   ↪ message is displayed.
149  */
150 bool validateHospitalCode(const char* code);
151
152 /*!
153  * @name deleteHospital
154  * @brief Delete hospital
155  * @details This function deletes the hospital with the
156  *   ↪ given code
157  *   by traversing the 'hospitalHead' linkedlist.
158  *
159  * @param[in] code Hospital code
160  * @param[in] adminUsername Admin username
161  * @param[in] adminPassword Admin password
162  *
163  * @return True if hospital is deleted, False otherwise
164  *
165  * @pre @p code is not empty and valid
166  * @pre @p adminUsername is not empty and valid
167  * @pre @p adminPassword is not empty
168  * @post The hospital with the given code is deleted from
169  *   ↪ the 'hospitalHead' linkedlist.
170  *
171  * @exception If the @p code is empty or invalid, an error
172  *   ↪ message is displayed.
173  * @exception If the @p adminUsername is empty or invalid or
174  *   ↪ @p adminPassword is empty, an error message is
175  *   ↪ displayed.
176  * @exception If the pair of @p adminUsername and @p
177  *   ↪ adminPassword is invalid, an error message is
178  *   ↪ displayed.
```

```
169  * @exception If the hospital with the given code is not
    ↪ found, an error message is displayed.
170  */
171  bool deleteHospital(const char* code, const char*
    ↪ adminUsername, const char* adminPassword);
172
173  /*!
174  * @name getHospitalNameByCode
175  * @brief Get hospital name by code
176  * @details This function gets the hospital name by the
    ↪ given code
177  * by traversing the 'hospitalHead' linkedlist.
178  *
179  * @param[in] code Hospital code
180  *
181  * @return Hospital name or NULL if not found
182  *
183  * @pre @p code is not empty and valid
184  * @post If the @p code is found in the 'hospitalHead'
    ↪ linkedlist,
185  * the function returns the hospital name. Otherwise, it
    ↪ returns NULL.
186  *
187  * @exception If the @p code is empty or invalid, an error
    ↪ message is displayed.
188  */
189  char* getHospitalNameByCode(const char* code);
190
191  /*!
192  * @name displayHospitals
193  * @brief Display all hospitals
194  * @details This function displays all the hospitals
195  * in the 'hospitalHead' linkedlist by traversing it.
196  *
197  * @post The hospitals in the 'hospitalHead' linkedlist are
    ↪ displayed in a formatted manner.
198  */
199  void displayHospitals(void);
200
201  /*!
202  * @name freeHospital
203  * @brief Free hospital list from memory
204  * @details This function frees the 'hospitalHead' linkedlist
205  * from memory by traversing it.
206  *
207  * @post The 'hospitalHead' linkedlist is freed from memory.
208  */
209  void freeHospital(void);
```

```
#endif
```

Listing A.9: hospital_manager.h

A.2.4 transaction_manager.h

```
1  /*!  
2   * @file transaction_manager.h  
3   *  
4   * @brief Transaction manager header file  
5   * @details This file contains the declarations of the  
6   *           ↪ functions and structures for the transaction manager  
7   *           ↪ module.  
8   *  
9   * @author CrimsonCare Team  
10  * @date 2025-01-18  
11  *  
12  * @copyright  
13  * Copyright (c) 2025 CrimsonCare Team  
14  *  
15  * Permission is hereby granted, free of charge, to any  
16  *           ↪ person obtaining a copy  
17  * of this software and associated documentation files (the  
18  *           ↪ "Software"), to deal  
19  * in the Software without restriction, including without  
20  *           ↪ limitation the rights  
21  * to use, copy, modify, merge, publish, distribute,  
22  *           ↪ sublicense, and/or sell  
23  * copies of the Software, and to permit persons to whom the  
24  *           ↪ Software is  
25  * furnished to do so, subject to the following conditions:  
26  *  
27  * The above copyright notice and this permission notice  
28  *           ↪ shall be included in all  
29  * copies or substantial portions of the Software.  
30  *  
31  * THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY  
32  *           ↪ KIND, EXPRESS OR  
33  * IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF  
34  *           ↪ MERCHANTABILITY,  
35  * FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN  
36  *           ↪ NO EVENT SHALL THE  
37  * AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM,  
38  *           ↪ DAMAGES OR OTHER  
39  * LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR  
40  *           ↪ OTHERWISE, ARISING FROM,
```

```

28  * OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR
    ↳ OTHER DEALINGS IN THE
29  * SOFTWARE.
30  */
31  #ifndef TRANSACTION_MANAGER_H
32  #define TRANSACTION_MANAGER_H
33
34  #include <stdbool.h>
35  #include <stdint.h>
36  #include <stdio.h>
37  #include <stdlib.h>
38  #include <string.h>
39  #include <time.h>
40  #include <errno.h>
41
42  /*!
43  * @def MAX_TRANSACTION_NAME_LENGTH
44  * @brief Maximum transaction name length
45  * @details This macro defines the maximum length of the
    ↳ transaction name.
46  */
47  #define MAX_TRANSACTION_NAME_LENGTH 50
48
49  /*!
50  * @def MAX_TRANSACTION_DATE_LENGTH
51  * @brief Maximum transaction date length
52  * @details This macro defines the maximum length of the
    ↳ transaction date.
53  */
54  #define MAX_TRANSACTION_DATE_LENGTH 11
55
56  /*!
57  * @def MAX_TRANSACTION_TOKEN_LENGTH
58  * @brief Maximum transaction token length
59  * @details This macro defines the maximum length of the
    ↳ transaction token.
60  */
61  #define MAX_TRANSACTION_TOKEN_LENGTH 12
62
63  /*!
64  * @enum TransactionType
65  * @brief Transaction type enum
66  * @details This enum defines the transaction type.
67  */
68  typedef enum TransactionType {
69      SELL, /*!< Sell transaction type */
70      BUY /*!< Buy transaction type */
71  } TransactionType;

```

```

72
73 /*!
74 * @name logTransaction
75 * @brief Log transaction to file
76 * @details This function logs a transaction to the file
77     ↪ 'resources/db/transactions.log'.
78 * 
79 * @param[in] type Transaction type
80 * @param[in] name Hospital name
81 * @param[in] bloodId Blood group id
82 * @param[in] quantity Blood quantity
83 * @param[in] date Transaction date
84 * @param[in] token Transaction token
85 * 
86 * @return True if transaction is logged, False otherwise
87 * 
88 * @note The file path 'resources/db' is relative to the
89     ↪ project root directory.
90 * Make sure that the folder exists also to run the program
91     ↪ from the root directory.
92 * 
93 * @pre @p type is either BUY or SELL
94 * @pre @p name is not empty and valid
95 * @pre @p bloodId is a valid blood group id
96 * @pre @p quantity is greater than 0
97 * @pre @p date is a valid date
98 * @post The transaction is logged to the file
99     ↪ 'resources/db/transactions.log'.
100 * 
101 * @exception If the file 'resources/db/transactions.log'
102     ↪ cannot be opened,
103 * an error message is displayed.
104 * @exception If the @p type is not BUY or SELL, an error
105     ↪ message is displayed.
106 * @exception If the @p name is empty or invalid, an error
107     ↪ message is displayed.
108 * @exception If the @p bloodId is not a valid blood group
109     ↪ id, an error message is displayed.
110 * @exception If the @p quantity is less than or equal to 0,
111     ↪ an error message is displayed.
112 * @exception If the @p date is not a valid date, an error
113     ↪ message is displayed.
114 */
115 bool logTransaction(TransactionType type, const char* name,
116     ↪ uint32_t bloodId, uint32_t quantity, const char* date,
117     ↪ const char* token);
118
119 /*!

```



```

108  * @name addTransaction
109  * @brief Add transaction
110  * @details This function adds a transaction to the
111  *         ↪ 'transactionHead' linkedlist.
112  *
113  * @param[in] type Transaction type
114  * @param[in] name Hospital name
115  * @param[in] bloodId Blood group id
116  * @param[in] quantity Blood quantity
117  *
118  * @return True if transaction is added, False otherwise
119  *
120  * @note For SELL transaction, the user is asked to enter
121  *         ↪ the date of donation,
122  * and a token is generated for the transaction.
123  *
124  * @pre @p name is not empty and valid
125  * @pre @p type is either BUY or SELL
126  * @pre @p quantity is greater than 0
127  * @pre @p bloodId is a valid blood group id
128  * @post The transaction is logged to the file
129  *         ↪ 'resources/db/transactions.log' through
130  *         ↪ 'logTransaction' function.
131  *
132  * @exception If the @p name is empty or invalid, an error
133  *         ↪ message is displayed.
134  * @exception If the @p type is not BUY or SELL, an error
135  *         ↪ message is displayed.
136  * @exception If the @p quantity is less than or equal to 0,
137  *         ↪ an error message is displayed.
138  * @exception If the @p bloodId is not a valid blood group
139  *         ↪ id, an error message is displayed.
140  * @exception For BUY transaction, if the @p name is not a
141  *         ↪ valid hospital code, an error message is displayed.
142  * @exception For SELL transaction, if the input date is not
143  *         ↪ a valid date, an error message is displayed.
144  */
145  bool addTransaction(TransactionType type, const char* name,
146  *         ↪ uint32_t bloodId, uint32_t quantity);
147
148  /*!
149  * @brief Display all transactions
150  * @details This function displays all transactions from the
151  *         ↪ file 'resources/db/transactions.log'.
152  *
153  * @pre The file 'resources/db/transactions.log' exists.
154  * @post All transactions are displayed.

```

```
144  *
145  * @exception If the file 'resources/db/transactions.log'
146  *   ↪ cannot be opened, an error message is displayed.
147  */
148  void displayTransactions(void);
149
150  /*!
151  * @brief Free transaction list from memory
152  */
153  void freeTransaction(void);
154  #endif
```

Listing A.10: transaction_manager.h

A.2.5 misc.h

```
1  /*!
2  * @file misc.h
3  *
4  * @brief Misc header file
5  * @details This file contains the declarations of the
6  *   ↪ functions for the misc module.
7  *
8  * @author CrimsonCare Team
9  * @date 2025-01-22
10  *
11  * @copyright
12  * Copyright (c) 2025 CrimsonCare Team
13  *
14  * Permission is hereby granted, free of charge, to any
15  *   ↪ person obtaining a copy
16  * of this software and associated documentation files (the
17  *   ↪ "Software"), to deal
18  * in the Software without restriction, including without
19  *   ↪ limitation the rights
20  * to use, copy, modify, merge, publish, distribute,
21  *   ↪ sublicense, and/or sell
22  * copies of the Software, and to permit persons to whom the
23  *   ↪ Software is
24  * furnished to do so, subject to the following conditions:
25  *
26  * The above copyright notice and this permission notice
27  *   ↪ shall be included in all
28  * copies or substantial portions of the Software.
29  *
```

```
23  * THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY
    ↳ KIND, EXPRESS OR
24  * IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
    ↳ MERCHANTABILITY,
25  * FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN
    ↳ NO EVENT SHALL THE
26  * AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM,
    ↳ DAMAGES OR OTHER
27  * LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR
    ↳ OTHERWISE, ARISING FROM,
28  * OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR
    ↳ OTHER DEALINGS IN THE
29  * SOFTWARE.
30  */
31  #ifndef MISC_H
32  #define MISC_H
33
34  #include <stdio.h>
35  #include <stdint.h>
36  #include <string.h>
37  #include <stdlib.h>
38  #include <ctype.h>
39  #include <stdbool.h>
40  #include <errno.h>
41
42  #ifdef _WIN32
43  #include <conio.h>
44  #else
45  #include <termios.h>
46  #include <unistd.h>
47  #endif
48
49  /*!
50   * @name displayWelcomeMessage
51   * @brief Display welcome message
52   * @details This function displays the welcome message
53   * by reading from the file 'resources/assets/misc/cc.txt'.
54   *
55   * @note The file 'resources/assets/misc/cc.txt' is a text
56   ↳ file
57   * that contains the welcome message.
58   *
59   * @pre The file 'resources/assets/misc/cc.txt' exists.
60   *
61   * @post The welcome message is displayed.
62   */
63  void displayWelcomeMessage(void);
```

```
64  /*!
65   * @name displayUserMenu
66   * @brief Display user menu
67   * @details This function displays the user menu.
68   *
69   * @post The user menu is displayed.
70   */
71  void displayUserMenu(void);
72
73  /*!
74   * @name displayAdminMenu
75   * @brief Display admin menu
76   * @details This function displays the admin menu.
77   *
78   * @post The admin menu is displayed.
79   */
80  void displayAdminMenu(void);
81
82  /*!
83   * @name clearInputBuffer
84   * @brief Clear input buffer
85   * @details This function clears the input buffer
86   * by reading until a newline character is encountered.
87   *
88   * @post The input buffer is cleared.
89   */
90  void clearInputBuffer(void);
91
92  /*!
93   * @name checkUsername
94   * @brief Check if username is valid
95   * @details This function checks if a username is valid.
96   *
97   * @param[in] str Username to check
98   *
99   * @return True if username is valid, False otherwise
100  *
101  * @note Username can only contain lowercase letters and
102  *       ↪ digits.
103  *
104  * @pre @p str is not empty
105  * @post If the @p str is valid, the function returns true.
106  * Otherwise, it returns false.
107  */
108 bool checkUsername(const char* str);
109
110 /*!
```

```
111  * @name containsPipe
112  * @brief Check if string contains pipe
113  * @details This function checks if a string contains a pipe
114  *         → character.
115  *
116  * @param[in] str String to check
117  *
118  * @return True if string contains pipe, False otherwise
119  *
120  * @pre @p str is not empty
121  * @post If the @p str contains a pipe character, the
122  *         → function returns true.
123  * Otherwise, it returns false.
124  */
125 bool containsPipe(const char* str);
126
127 /*!
128  * @name getPassword
129  * @brief Get password
130  * @details This function gets the password from the user
131  * by reading from the standard input.
132  *
133  * @param[in,out] password Password
134  * @param[in] size Password size
135  *
136  * @post Updates the @p password with the user's input
137  * through the pointer @p password.
138  */
139 void getPassword(char* password, size_t size);
140
141 /*!
142  * @name isLeapYear
143  * @brief Check if year is leap year
144  * @details This function checks if a year is a leap year.
145  *
146  * @param[in] year Year to check
147  *
148  * @return True if year is leap year, False otherwise
149  */
150 bool isLeapYear(int year);
151
152 /*!
153  * @name isValidDate
154  * @brief Check if date is valid
155  * @details This function checks if a date is valid.
156  *
157  * @param[in] date Date to check
158  *
```

```
157  * @return True if date is valid, False otherwise
158  *
159  * @pre @p date is not empty
160  * @post If the @p date is valid, the function returns true.
161  * Otherwise, it returns false.
162  *
163  * @exception If the @p date is empty, an error message is
164  *   ↪ displayed.
165  * @exception If the @p date is invalid, an error message is
166  *   ↪ displayed.
167  */
168  bool isValidDate(const char* date);
169
170  /*!
171  * @name formatDate
172  * @brief Format date to yyyy-mm-dd
173  * @details This function formats a date string to the
174  *   ↪ format yyyy-mm-dd.
175  *
176  * @param[in,out] date Date to format
177  *
178  * @pre @p date is not empty
179  * @post The @p date is formatted to the format yyyy-mm-dd
180  *   ↪ and updates the @p date through the pointer @p date.
181  *
182  * @exception If the @p date is empty, an error message is
183  *   ↪ displayed.
184  * @exception If the @p date is invalid, an error message is
185  *   ↪ displayed.
186  */
187  void formatDate(char* date);
188
189 #endif
```

Listing A.11: misc.h