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

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:

```c
typedef struct Admin {
    char username[MAX_USERNAME_LENGTH];
    char password[MAX_PASSWORD_LENGTH];
    struct Admin* next;
} Admin;

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:

```c
typedef struct Hospital {
    char name[MAX_NAME_LENGTH];
    char location[MAX_LOCATION_LENGTH];
    char code[MAX_CODE_LENGTH];
    struct Hospital* next;
} Hospital;

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:

```c
typedef struct BloodStock {
    float price;
    uint32_t id;
    uint32_t quantity;
    char bloodGroup[BLOOD_GROUP_NAME_LENGTH];
    struct BloodStock* next;
} BloodStock;

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):

```c
char* addHospital(const char* name, const char*
    location) {
    // ...

    Hospital* newHospital =
        (Hospital*)malloc(sizeof(Hospital));
    if (!newHospital) {
        printf("Error allocating memory for hospital:
            %s\n", strerror(errno));
        return NULL;
    }

    // ...
    newHospital->next = NULL;

    if (hospitalHead == NULL) {
        hospitalHead = newHospital;
    } else {
        Hospital* temp = hospitalHead;
        while (temp->next != NULL) {
            temp = temp->next;
        }
        temp->next = newHospital;
    }

    return newHospital->code;
}
```

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):

```cpp
bool deleteHospital(const char* code) {
    Hospital* current = hospitalHead;
    Hospital* prev = NULL;
    while (current != NULL) {
        if (strcmp(current->code, code) == 0) {
            if (prev == NULL) {
                hospitalHead = current->next;
            } else {
                prev->next = current->next;
            }
            return true;
        }
        prev = current;
        current = current->next;
    }
    return false;
}
```

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):

```cpp
void displayHospitals(void) {
    Hospital* temp = hospitalHead;
    if (temp == NULL) {
        // ...
        return;
    }

```

```
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):

```c
bool changeAdminPassword(const char* username, const
    char* oldPassword, const char* newPassword) {
    // ...
    Admin* temp = adminHead;
    while (temp != NULL) {
        if (strcmp(username, temp->username) == 0 &&
            strcmp(oldPassword, temp->password) == 0) {
            strncpy(temp->password, newPassword,
                sizeof(temp->password) - 1);
            temp->password[sizeof(temp->password) - 1] =
                '\0';
            saveAdminCredentials();
            return true;
        }
        temp = temp->next;
    }
    return false;
}
```

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):

```c
void freeAdmin(void) {
    Admin* current = adminHead;
    while (current != NULL) {
        Admin* temp = current;
        current = current->next;
        free(temp);
    }
    adminHead = NULL;
}
```

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`.

```c
void saveAdminCredentials(void) {
    errno = 0;
    FILE* file = fopen("resources/db/admin_credentials.dat",
        "wb");
    if (!file) {
        if (errno != ENOENT) {
            printf("Error opening admin credentials file:
                %s\n", strerror(errno));
            return;
        }
    }

    Admin* temp = adminHead;
    while (temp != NULL) {
        if (fwrite(temp, sizeof(Admin), 1, file)) {
            temp = temp->next;
        } else {
```

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

```
30              printf("Error allocating memory for admin:
                    ↪ %s\n", strerror(errno));
31              freeAdmin();
32              fclose(file);
33              return;
34          }
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
                  ↪ contain lowercase letters and digits.\n");
9          return false;
10     }
11
12     Admin* temp = adminHead;
13     while (temp != NULL) {
14         if (strcmp(temp->username, username) == 0) {
15             return true;
16         }
17         temp = temp->next;
18     }
19     return false;
20 }
```

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
```

```c
    if (!validateAdmin(currentAdminUsername,
        currentAdminPassword)) {
        printf("Error: Invalid current admin
            credentials.\n");
        return false;
    }

    if (adminExists(username)) {
        printf("Error: Admin already exists.\n");
        return false;
    }

    if (strcmp(username, "") == 0 || strcmp(password, "") ==
        0) {
        printf("Error: Admin credentials cannot be
            empty.\n");
        return false;
    }

    Admin* newAdmin = (Admin*)malloc(sizeof(Admin));
    if (!newAdmin) {
        printf("Error allocating memory for admin: %s\n",
            strerror(errno));
        return false;
    }
    strncpy(newAdmin->username, username,
        sizeof(newAdmin->username) - 1);
    newAdmin->username[sizeof(newAdmin->username) - 1] =
        '\0';
    strncpy(newAdmin->password, password,
        sizeof(newAdmin->password) - 1);
    newAdmin->password[sizeof(newAdmin->password) - 1] =
        '\0';
    newAdmin->next = adminHead;
    adminHead = newAdmin;

    saveAdminCredentials();
    return true;
}
```

Listing 4.5: addAdmin ()

deleteAdmin ()

Deletes an admin from the system.

```c
bool deleteAdmin(const char* username, const char*
    currentAdminUsername, const char*
```

```c
  ↪ currentAdminPassword) {
    if (strcmp(currentAdminUsername, "") == 0 ||
       ↪ strcmp(currentAdminPassword, "") == 0) {
        printf("Error: Current admin credentials cannot be
            ↪ empty.\n");
        return false;
    }

    if (!checkUsername(currentAdminUsername) ||
       ↪ !checkUsername(username)) {
        printf("Error: Invalid username. Username can only
            ↪ contain lowercase letters and digits.\n");
        return false;
    }

    if (!validateAdmin(currentAdminUsername,
       ↪ currentAdminPassword)) {
        printf("Error: Invalid current admin
            ↪ credentials.\n");
        return false;
    }

    if (!adminExists(username)) {
        printf("Error: Admin does not exist.\n");
        return false;
    }

    if (strcmp(username, "") == 0) {
        printf("Error: Admin username cannot be empty.\n");
        return false;
    }

    if (strcmp(username, currentAdminUsername) == 0) {
        printf("Error: Cannot delete current admin.\n");
        return false;
    }

    Admin* temp = adminHead;
    Admin* prev = NULL;

    while (temp != NULL) {
        if (strcmp(temp->username, username) == 0) {
            if (prev == NULL) {
                adminHead = temp->next;
            } else {
                prev->next = temp->next;
            }
            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) /
         ↪ sizeof(availableBloodGroups[0]));
3  }
```

Listing 4.10: isValidBloodGroup ()

`addBloodGroup ()`

Adds a new blood group to the system.

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

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

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",
                 ↪ strerror(errno));
7              return;
8          }
9      }
10
11     BloodStock* temp = bloodHead;
12     while (temp != NULL) {
13         fprintf(file, "%u %s %.2f %u\n", temp->id,
             ↪ temp->bloodGroup, temp->price, temp->quantity);
```

```
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          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",
                  ↪ strerror(errno));
10             freeBloodList ();
11             return;
12         }
13     }
14
15     while (1) {
16         BloodStock* newBlood =
               ↪ (BloodStock*) malloc(sizeof(BloodStock));
17         if (!newBlood) {
18             printf("Error allocating memory for blood group:
                   ↪ %s\n", strerror(errno));
19             freeBloodList ();
20             fclose(file);
21             return;
22         }
23
24         if (fscanf(file, "%u %s %f %u", &newBlood->id,
               ↪ newBlood->bloodGroup, &newBlood->price,
               ↪ &newBlood->quantity) != 4) {
25             free(newBlood);
26             fclose(file);
27             break;
28         }
29
30         newBlood->next = NULL;
31
```

```
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 &&
                       ↪ temp->quantity > 0) {
21                     return true;
22                 }
23             }
24         } else {
25             if (id == NULL) {
26                 if (temp->price > 0) {
27                     return true;
```

```
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) /
           ↪ sizeof(availableBloodGroups[0])); i++) {
3          printf("%u. %s\n", i + 1, availableBloodGroups[i]);
4      }
5  }
```

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",
                   ↪ temp->id, temp->bloodGroup, temp->price,
                   ↪ temp->quantity);
11         } else {
12             printf("%u. %s, Price: N/A, Quantity: N/A\n",
                   ↪ temp->id, temp->bloodGroup);
13         }
```

```
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;
```

```c
    FILE* file = fopen("resources/db/hospitals.txt", "r");
    if (!file) {
        if (errno == ENOENT) {
            return;
        } else {
            printf("Error opening hospitals file: %s\n",
                ↪ strerror(errno));
            freeHospital();
            return;
        }
    }

    while (1) {
        Hospital* newHospital =
            ↪ (Hospital*)malloc(sizeof(Hospital));
        if (!newHospital) {
            printf("Error allocating memory for hospital:
                ↪ %s\n", strerror(errno));
            fclose(file);
            freeHospital();
            return;
        }

        if (fscanf(file, "%[^|]|%[^|]|%[^\n]\n",
            ↪ newHospital->code, newHospital->name,
            ↪ newHospital->location) != 3) {
            free(newHospital);
            fclose(file);
            break;
        }

        newHospital->next = NULL;

        if (hospitalHead == NULL) {
            hospitalHead = newHospital;
        } else {
            Hospital* temp = hospitalHead;
            while (temp->next != NULL) {
                temp = temp->next;
            }
            temp->next = newHospital;
        }
    }

    fclose(file);
}
```

Listing 4.22: loadHospitals ()

saveHospitals ()

Saves hospitals to the file `hospitals.txt`.

```c
void saveHospitals(void) {
    errno = 0;
    FILE* file = fopen("resources/db/hospitals.txt", "w");
    if (!file) {
        printf("Error opening hospitals file: %s\n",
            strerror(errno));
        return;
    }

    Hospital* temp = hospitalHead;
    while (temp != NULL) {
        fprintf(file, "%s|%s|%s\n", temp->code, temp->name,
            temp->location);
        temp = temp->next;
    }
    fclose(file);
}
```

Listing 4.23: saveHospitals ()

addHospital ()

Adds a new hospital to the system.

```c
char* addHospital(const char* name, const char* location) {
    if (strcmp(name, "") == 0 || strcmp(location, "") == 0) {
        printf("Error: Hospital name or location cannot be
            empty.\n");
        return NULL;
    }

    if (containsPipe(name) || containsPipe(location)) {
        printf("Error: Hospital name or location cannot
            contain a pipe character.\n");
        return NULL;
    }

    char code[8];
    char initials[4] = { 0 };
    int initialCount = 0;

    char nameCopy[100];
    strncpy(nameCopy, name, sizeof(nameCopy) - 1);
    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,
             ↪ randomSuffix);
33
34         codeExists = false;
35         Hospital* temp = hospitalHead;
36         while (temp != NULL) {
37             if (strcmp(temp->code, code) == 0) {
38                 codeExists = true;
39                 break;
40             }
41             temp = temp->next;
42         }
43     } while (codeExists);
44
45     Hospital* newHospital =
         ↪ (Hospital*)malloc(sizeof(Hospital));
46     if (!newHospital) {
47         printf("Error allocating memory for hospital: %s\n",
             ↪ strerror(errno));
48         return NULL;
49     }
50
51     strncpy(newHospital->code, code,
         ↪ sizeof(newHospital->code) - 1);
52     newHospital->code[sizeof(newHospital->code) - 1] = '\0';
53     strncpy(newHospital->name, name,
         ↪ sizeof(newHospital->name) - 1);
54     newHospital->name[sizeof(newHospital->name) - 1] = '\0';
55     strncpy(newHospital->location, location,
         ↪ sizeof(newHospital->location) - 1);
56     newHospital->location[sizeof(newHospital->location) - 1]
         ↪ = '\0';
57     newHospital->next = NULL;
58
59     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
               ↪ character.\n");
9          return false;
10     }
11
12     Hospital* temp = hospitalHead;
13     while (temp != NULL) {
14         if (strcmp(temp->code, code) == 0) {
15             return true;
16         }
17         temp = temp->next;
18     }
19     return false;
20 }
```

Listing 4.25: validateHospitalCode ()

deleteHospital ()

Deletes a hospital from the system.

```c
bool deleteHospital(const char* code, const char*
    ↪ adminUsername, const char* adminPassword) {
    if (strcmp(adminUsername, "") == 0 ||
        ↪ strcmp(adminPassword, "") == 0) {
        printf("Error: Admin credentials cannot be
            ↪ empty.\n");
        return false;
    }

    if (!checkUsername(adminUsername)) {
        printf("Error: Invalid admin username. Username can
            ↪ only contain lowercase letters and digits.\n");
        return false;
    }

    if (!validateAdmin(adminUsername, adminPassword)) {
        printf("Error: Invalid admin credentials.\n");
        return false;
    }

    if (strcmp(code, "") == 0) {
        printf("Error: Hospital code cannot be empty.\n");
        return false;
    }

    if (containsPipe(code)) {
        printf("Error: Hospital code cannot contain a pipe
            ↪ character.\n");
        return false;
    }

    if (!validateHospitalCode(code)) {
        printf("Error: Hospital code is invalid.\n");
        return false;
    }

    Hospital* current = hospitalHead;
    Hospital* prev = NULL;
    while (current != NULL) {
        if (strcmp(current->code, code) == 0) {
            if (prev == NULL) {
                hospitalHead = current->next;
            } else {
                prev->next = current->next;
            }
            saveHospitals();
            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
                ↪ character.\n");
9           return NULL;
10      }
11
12      if (!validateHospitalCode(code)) {
13          printf("Error: Hospital code is invalid.\n");
14          return NULL;
15      }
16
17      Hospital* temp = hospitalHead;
18      while (temp != NULL) {
19          if (strcmp(temp->code, code) == 0) {
20              return temp->name;
21          }
22          temp = temp->next;
23      }
24      return NULL;
25  }
```

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,
                  ↪ temp->location);
12          temp = temp->next;
13          if (temp != NULL) {
14              printf("\t--------------------------------\n");
15          }
16      }
17  }
```

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,
      ↪ uint32_t bloodId, uint32_t quantity, const char* date,
      ↪ 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:
                 ↪ %s\n", strerror(errno));
7          }
8          return false;
9      }
10
11     if (containsPipe(name)) {
12         printf("Error: Entity name cannot contain a pipe
               ↪ character.\n");
13         return false;
14     }
15
16     if (type != BUY && type != SELL) {
17         printf("Error: Invalid transaction type.\n");
18         return false;
19     }
20
21     if (!isValidBloodGroup(bloodId)) {
22         printf("Error: Invalid blood group.\n");
23         return false;
24     }
25
26     if (strcmp(name, "") == 0 || quantity <= 0) {
27         printf("Error: Invalid transaction parameters.\n");
28         return false;
29     }
30
31     if (!isValidDate(date)) {
32         printf("Error: Invalid date format.\n");
33         return false;
34     }
35
36     if (token) {
37         fprintf(file, "%s|%s|%u|%u|%s|%s\n", (type == BUY ?
               ↪ "Buy" : "Sell"), name, bloodId, quantity,
               ↪ date, token);
38     } else {
39         fprintf(file, "%s|%s|%u|%u|%s\n", (type == BUY ?
               ↪ "Buy" : "Sell"), name, bloodId, quantity,
               ↪ date);
40     }
41
42     fclose(file);
43     return true;
```

```
44  }
```

Listing 4.30: logTransaction ()

addTransaction ()

Adds a transaction to the system.

```
1  bool addTransaction(TransactionType type, const char* name,
     ↪ uint32_t bloodId, uint32_t quantity) {
2      if (strcmp(name, "") == 0 || quantity <= 0) {
3          printf("Error: Invalid transaction parameters.\n");
4          return false;
5      }
6
7      if (containsPipe(name)) {
8          printf("Error: Entity name cannot contain a pipe
             ↪ character.\n");
9          return false;
10     }
11
12     if (type != BUY && type != SELL) {
13         printf("Error: Invalid transaction type.\n");
14         return false;
15     }
16
17     if (!isBloodAvailable(&bloodId, type)) {
18         printf("No stock available for blood group: %s\n",
             ↪ getBloodGroupById(bloodId));
19         return false;
20     }
21
22     if (type == BUY) {
23         if (!validateHospitalCode(name)) {
24             printf("Error: Invalid hospital code.\n");
25             return false;
26         }
27     }
28
29     char date[MAX_TRANSACTION_DATE_LENGTH];
30     char token[MAX_TRANSACTION_TOKEN_LENGTH] = "";
31
32     if (type == SELL) {
33         printf("Enter the date and time of donation
             ↪ (YYYY-MM-DD): ");
34         fgets(date, sizeof(date), stdin);
35         date[strcspn(date, "\n")] = 0;
36         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
                          ↪ group: %s. Available quantity:
                          ↪ %u\n", getBloodGroupById(bloodId),
                          ↪ stock->quantity);
47                      return false;
48                  }
49
50                  stock->quantity -= quantity;
51                  saveBloodGroups();
52                  break;
53              }
54              stock = stock->next;
55          }
56          time_t now = time(NULL);
57          strftime(date, sizeof(date), "%Y-%m-%d",
              ↪ localtime(&now));
58      }
59
60      if (type == SELL) {
61          srand(time(NULL));
62          sprintf(token, "TOKEN_%d", rand() % 10000);
63          printf("Sell token generated for %s: %s\n", name,
              ↪ token);
64      }
65
66      if (!logTransaction(type, name, bloodId, quantity, date,
          ↪ type == SELL ? token : NULL)) {
67          return false;
68      }
69
70      return true;
71  }
```

Listing 4.31: addTransaction ()

```
displayTransactions ()
```

Displays all transactions.

```c
void displayTransactions(void) {
    errno = 0;
    FILE* file = fopen("resources/db/transactions.log", "r");
    if (!file) {
        if (errno == ENOENT) {
            printf("No registered transactions found.\n");
        } else {
            printf("Error opening transaction log file:
                ↪ %s\n", strerror(errno));
        }
        return;
    }

    char line[256];
    bool hasLogs = false;
    bool firstLog = true;
    char prevLine[256] = { 0 };

    while (fgets(line, sizeof(line), file) != NULL) {
        char type[MAX_TRANSACTION_TOKEN_LENGTH] = "";
        char name[MAX_TRANSACTION_TOKEN_LENGTH] = "";
        uint32_t bloodId = 0;
        uint32_t quantity = 0;
        char date[MAX_TRANSACTION_TOKEN_LENGTH] = "";
        char token[MAX_TRANSACTION_TOKEN_LENGTH] = "";

        if (firstLog) {
            printf("\nRegistered Transactions:\n");
            firstLog = false;
        }

        if (sscanf(line, "%[^|]|%[^|]|%u|%u|%[^|]|%[^|\n]",
            type,
            name,
            &bloodId,
            &quantity,
            date,
            token) >= 5) {

            hasLogs = true;

            if (prevLine[0] != '\0') {
                printf("\t-------------------------------\n");
            }

            printf("\tType: %s\n"
                "\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.

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

Listing 4.34: displayUserMenu ()

displayAdminMenu ()

Displays the admin menu.

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

Listing 4.35: displayAdminMenu ()

clearInputBuffer ()

Clears the input buffer.

```c
void clearInputBuffer(void) {
    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'
             ↪ && *str <= '9')) {
4              return false;
5          }
6          str++;
7      }
8      return true;
9  }
```

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;
```

```c
    while (i < size - 1) {

        ch = getch();

        if (ch == '\r') {

            break;
        } else if (ch == '\b') {

            if (i > 0) {
                i--;
                printf("\b \b");

            }
        } else {
            password[i++] = ch;
            printf("*");

        }
    }
    password[i] = '\0';

    printf("\n");
#else

    struct termios oldt, newt;
    tcgetattr(STDIN_FILENO, &oldt);

    newt = oldt;

    newt.c_lflag &= ~(ECHO);

    tcsetattr(STDIN_FILENO, TCSANOW, &newt);

    fgets(password, size);
    password[strcspn(password, "\n")] = 0;

    tcsetattr(STDIN_FILENO, TCSANOW, &oldt);

    printf("\n");
#endif
}
```

Listing 4.39: getPassword ()

isLeapYear ()

Checks if a year is a leap year.

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

Listing 4.40: isLeapYear ()

isValidDate ()

Checks if a date is valid.

```
bool isValidDate(const char* date) {
    if (strcmp(date, "") == 0) {
        printf("Error: Date cannot be empty.\n");
        return false;
    }

    int year, month, day;

    if (strlen(date) < 8 || strlen(date) > 10) {
        printf("Error: Invalid date format.\n");
        return false;
    }

    if (sscanf(date, "%d-%d-%d", &year, &month, &day) != 3) {
        printf("Error: Invalid date format.\n");
        return false;
    }

    if (month < 1 || month > 12) {
        printf("Error: Invalid month.\n");
        return false;
    }

    int daysInMonth[] = { 31, 28 + (int)isLeapYear(year),
        ↪ 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };

    if (day < 1 || day > daysInMonth[month - 1]) {
        printf("Error: Invalid day.\n");
        return false;
    }

    return true;
}
```

Listing 4.41: isValidDate ()

formatDate ()

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

```c
void formatDate(char* date) {
    if (strcmp(date, "") == 0) {
        printf("Error: Date cannot be empty.\n");
        return;
    }

    if (!isValidDate(date)) {
        printf("Error: Invalid date format.\n");
        return;
    }

    int year, month, day;
    sscanf(date, "%d-%d-%d", &year, &month, &day);
    sprintf(date, "%04d-%02d-%02d", year, month, day);
}
```

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

```c
/*!
 * @file main.c
 *
 * @brief Main source file
 * @details This file contains the implementation of the
   ↪ main function.
 *
 * @author CrimsonCare Team
 * @date 2025-01-18
 *
 * @copyright
 * Copyright (c) 2025 CrimsonCare Team
 *
 * Permission is hereby granted, free of charge, to any
   ↪ person obtaining a copy
 * of this software and associated documentation files (the
   ↪ "Software"), to deal
 * in the Software without restriction, including without
   ↪ limitation the rights
 * to use, copy, modify, merge, publish, distribute,
   ↪ sublicense, and/or sell
 * copies of the Software, and to permit persons to whom the
   ↪ Software is
 * furnished to do so, subject to the following conditions:
 *
 * The above copyright notice and this permission notice
   ↪ shall be included in all
 * 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",
                   ↪ getBloodGroupById(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,
                            ↪ transactionType)) {
106                         printf("No blood available for sale.\n");
107                         break;
108                     }
109                     char donorName[MAX_TRANSACTION_NAME_LENGTH];
110                     uint32_t bloodGroupId;
111                     uint32_t quantity;
112                     printf("Enter Donor Name: ");
113                     fgets(donorName, sizeof(donorName), stdin);
114                     donorName[strcspn(donorName, "\n")] = 0;
115                     if (containsPipe(donorName)) {
116                         printf("Error: Donor name cannot contain
                                ↪ a pipe character.\n");
117                         break;
118                     }
119                     displayBloodStocks();
120                     printf("Enter Blood Group ID: ");
121                     scanf("%u", &bloodGroupId);
122                     clearInputBuffer();
123                     if (!isValidBloodGroup(bloodGroupId)) {
124                         printf("Error: Invalid blood group.\n");
125                         break;
126                     }
127                     if (!isBloodAvailable(&bloodGroupId,
                            ↪ transactionType)) {
128                         printf("Blood %s is not available for
                                ↪ sale.\n",
                                ↪ getBloodGroupById(bloodGroupId));
129                         break;
130                     }
131
132                     printf("Enter Quantity: ");
133                     scanf("%u", &quantity);
134                     clearInputBuffer();
135                     if (addTransaction(transactionType,
                            ↪ donorName, bloodGroupId, quantity)) {
136                         printf("Transaction successful for
                                ↪ %s.\n", donorName);
```

```
137              } else {
138                  printf("Error: Transaction failed.\n");
139              }
140              break;
141          }
142      case 3:
143          displayBloodStocks();
144          break;
145      case 4: {
146              char adminPassword[MAX_PASSWORD_LENGTH];
147              printf("Enter Admin Username: ");
148              fgets(currentAdminUsername,
                  ↪ sizeof(currentAdminUsername), stdin);
149              currentAdminUsername[strcspn(currentAdminUsername,
                  ↪ "\n")] = 0;
150              if (!checkUsername(currentAdminUsername)) {
151                  printf("Error: Invalid username.
                      ↪ Username can only contain
                      ↪ lowercase letters and digits.\n");
152                  break;
153              }
154              printf("Enter Admin Password: ");
155              getPassword(adminPassword,
                  ↪ sizeof(adminPassword));
156
157              if (validateAdmin(currentAdminUsername,
                  ↪ adminPassword)) {
158                isAdmin = true;
159                printf("Admin %s logged in
                      ↪ successfully.\n",
                      ↪ currentAdminUsername);
160
161                  while (isAdmin) {
162                      displayAdminMenu();
163
164                      if (scanf("%u", &choice) != 1) {
165                          printf("Error: Invalid
                              ↪ input.\n");
166                          clearInputBuffer();
167                          continue;
168                      }
169
170                      clearInputBuffer();
171
172                      switch (choice) {
173                      case 1: {
174                              char
                                  ↪ hospitalName[MAX_HOSPITAL_NAME_LEN
```

```
175    char
          ↪ hospitalLocation[MAX_HOSPITAL_LOCA
176    printf("Enter Hospital Name:
          ↪ ");
177    fgets(hospitalName,
          ↪ sizeof(hospitalName),
          ↪ stdin);
178    hospitalName[strcspn(hospitalName,
          ↪ "\n")] = 0;
179    if
          ↪ (containsPipe(hospitalName))
          ↪ {
180      printf("Error: Hospital
            ↪ name cannot
            ↪ contain a pipe
            ↪ character.\n");
181      break;
182    }
183    printf("Enter Hospital
          ↪ Location: ");
184    fgets(hospitalLocation,
          ↪ sizeof(hospitalLocation),
          ↪ stdin);
185    hospitalLocation[strcspn(hospitalLocatio
          ↪ "\n")] = 0;
186    if
          ↪ (containsPipe(hospitalLocation))
          ↪ {
187      printf("Error: Hospital
            ↪ location cannot
            ↪ contain a pipe
            ↪ character.\n");
188      break;
189    }
190    char* hospitalCode =
          ↪ addHospital(hospitalName,
          ↪ hospitalLocation);
191    if (hospitalCode) {
192      printf("Hospital %s
            ↪ added successfully
            ↪ by %s.\n",
            ↪ hospitalName,
            ↪ currentAdminUsername);
193      printf("Hospital Code:
            ↪ %s\n",
            ↪ hospitalCode);
194    } else {
```

```
195                                    printf("Error: Could not
                                          ↪ add hospital.\n");
196                                }
197                                break;
198                            }
199                        case 2: {
200                            displayBloodGroups();
201                            uint32_t bloodGroupId;
202                            uint32_t newQuantity;
203                            printf("Enter Blood Group
                                  ↪ ID: ");
204                            scanf("%u", &bloodGroupId);
205                            clearInputBuffer();
206                            if
                                  ↪ (!isValidBloodGroup(bloodGroupId))
                                  ↪ {
207                                printf("Error: Invalid
                                      ↪ blood group.\n");
208                                break;
209                            }
210                            if
                                  ↪ (!isBloodAvailable(&bloodGroupId,
                                  ↪ SELL)) {
211                                float newPrice;
212                                printf("Price for %s is
                                      ↪ not set.\n",
                                      ↪ getBloodGroupById(bloodGroupId
213                                printf("Enter New Price:
                                      ↪ ");
214                                scanf("%f", &newPrice);
215                                clearInputBuffer();
216                                if
                                      ↪ (updateBloodPrice(bloodGroupId
                                      ↪ newPrice)) {
217                                    printf("Blood price
                                          ↪ for %s updated
                                          ↪ successfully
                                          ↪ by %s.\n",
                                          ↪ getBloodGroupById(bloodGro
                                          ↪ currentAdminUsername);
218                                } else {
219                                    printf("Error: Could
                                          ↪ not update
                                          ↪ blood
                                          ↪ price.\n");
220                                }
221                            }
```

```
222                                         printf("Enter New Quantity:
                                         ↪ ");
223                                         scanf("%u", &newQuantity);
224                                         clearInputBuffer();
225                                         if
                                         ↪ (updateBloodQuantity(bloodGroupId,
                                         ↪ newQuantity)) {
226                                           printf("Blood quantity
                                         ↪ for %s updated
                                         ↪ successfully by
                                         ↪ %s.\n",
                                         ↪ getBloodGroupById(bloodGroupId
                                         ↪ currentAdminUsername);
227                                         } else {
228                                           printf("Error: Could not
                                         ↪ update blood
                                         ↪ quantity.\n");
229                                         }
230                                         break;
231                                     }
232                                 case 3: {
233                                         displayBloodGroups();
234                                         uint32_t bloodGroupId;
235                                         float newPrice;
236                                         printf("Enter Blood Group
                                         ↪ ID: ");
237                                         scanf("%u", &bloodGroupId);
238                                         clearInputBuffer();
239                                         if
                                         ↪ (!isValidBloodGroup(bloodGroupId))
                                         ↪ {
240                                           printf("Error: Invalid
                                         ↪ blood group.\n");
241                                           break;
242                                         }
243                                         printf("Enter New Price: ");
244                                         scanf("%f", &newPrice);
245                                         clearInputBuffer();
246                                         if
                                         ↪ (updateBloodPrice(bloodGroupId,
                                         ↪ newPrice)) {
247                                           printf("Blood price for
                                         ↪ %s updated
                                         ↪ successfully by
                                         ↪ %s.\n",
                                         ↪ getBloodGroupById(bloodGroupId
                                         ↪ currentAdminUsername);
248                                         } else {
```

```
249                                    printf("Error: Could not
                                          ↪ update blood
                                          ↪ price.\n");
250                                }
251                                break;
252                            }
253                        case 4: {
254                            char
                                  ↪ adminOldPassword[MAX_PASSWORD_LENG
255                            char
                                  ↪ adminNewPassword[MAX_PASSWORD_LENG
256                            printf("Enter Old Password:
                                  ↪ ");
257                            getPassword(adminOldPassword,
                                  ↪ sizeof(adminOldPassword));
258                            printf("Enter New Password:
                                  ↪ ");
259                            getPassword(adminNewPassword,
                                  ↪ sizeof(adminNewPassword));
260                            if
                                  ↪ (changeAdminPassword(currentAdminU
                                  ↪ adminOldPassword,
                                  ↪ adminNewPassword)) {
261                              printf("Password for %s
                                    ↪ updated
                                    ↪ successfully.\n",
                                    ↪ currentAdminUsername);
262                            } else {
263                                printf("Error: Could not
                                      ↪ change admin
                                      ↪ password.\n");
264                            }
265                            break;
266                        }
267                        case 5: {
268                            char
                                  ↪ currentAdminPassword[MAX_PASSWORD_
269                            char
                                  ↪ newAdminUsername[MAX_USERNAME_LENG
270                            char
                                  ↪ newAdminPassword[MAX_PASSWORD_LENG
271                            char
                                  ↪ confirmNewAdminPassword[MAX_PASSWO
272                            printf("To continue, please
                                  ↪ enter your (%s)
                                  ↪ current password: ",
                                  ↪ currentAdminUsername);
```

```
273                                    getPassword(currentAdminPassword,
                                         ↪ sizeof(currentAdminPassword));
274                                    if
                                         ↪ (validateAdmin(currentAdminUsernam
                                         ↪ currentAdminPassword))
                                         ↪ {
275                                      printf("Password
                                           ↪ verified
                                           ↪ successfully.\n");
276                                    } else {
277                                      printf("Error: Invalid
                                           ↪ password.\n");
278                                      break;
279                                    }
280                                    printf("Enter New Admin
                                         ↪ Username: ");
281                                    fgets(newAdminUsername,
                                         ↪ sizeof(newAdminUsername),
                                         ↪ stdin);
282                                    newAdminUsername[strcspn(newAdminUsernam
                                         ↪ "\n")] = 0;
283                                    if
                                         ↪ (!checkUsername(newAdminUsername))
                                         ↪ {
284                                      printf("Error: Invalid
                                           ↪ username. Username
                                           ↪ can only contain
                                           ↪ lowercase letters
                                           ↪ and digits.\n");
285                                      break;
286                                    }
287                                    printf("Enter New Admin
                                         ↪ Password: ");
288                                    getPassword(newAdminPassword,
                                         ↪ sizeof(newAdminPassword));
289                                    printf("Confirm New Admin
                                         ↪ Password: ");
290                                    getPassword(confirmNewAdminPassword,
                                         ↪ sizeof(confirmNewAdminPassword));
291                                    if (strcmp(newAdminPassword,
                                         ↪ confirmNewAdminPassword)
                                         ↪ != 0) {
292                                      printf("Error: Passwords
                                           ↪ do not match.\n");
293                                      break;
294                                    }
295                                    if
                                         ↪ (addAdmin(newAdminUsername,
```

```
                                                     ↪ newAdminPassword,
                                                     ↪ currentAdminUsername,
                                                     ↪ currentAdminPassword))
                                                     ↪ {
296                                                    printf("New admin %s
                                                        ↪ added successfully
                                                        ↪ by %s.\n",
                                                        ↪ newAdminUsername,
                                                        ↪ currentAdminUsername);
297                                               } else {
298                                                   printf("Error: Could not
                                                        ↪ add new admin.\n");
299                                               }
300                                               break;
301                                           }
302                                       case 6: {
303                                           char
                                               ↪ delAdminUsername[MAX_USERNAME_LENG
304                                           char
                                               ↪ currentAdminPassword[MAX_PASSWORD_
305                                           printf("To continue, please
                                               ↪ enter your (%s
                                               ↪ current password: ",
                                               ↪ currentAdminUsername);
306                                           getPassword(currentAdminPassword,
                                               ↪ sizeof(currentAdminPassword));
307                                           if
                                               ↪ (validateAdmin(currentAdminUsernam
                                               ↪ currentAdminPassword))
                                               ↪ {
308                                               printf("Password
                                                   ↪ verified
                                                   ↪ successfully.\n");
309                                           } else {
310                                               printf("Error: Invalid
                                                   ↪ password.\n");
311                                               break;
312                                           }
313                                           printf("Enter Admin Username
                                               ↪ to Delete: ");
314                                           fgets(delAdminUsername,
                                               ↪ sizeof(delAdminUsername),
                                               ↪ stdin);
315                                           delAdminUsername[strcspn(delAdminUsernam
                                               ↪ "\n")] = 0;
316                                           if
                                               ↪ (!checkUsername(delAdminUsername))
                                               ↪ {
```

```
317                                     printf("Error: Invalid
                                            ↪ username. Username
                                            ↪ can only contain
                                            ↪ lowercase letters
                                            ↪ and digits.\n");
318                                     break;
319                                 }
320                                 if
                                    ↪ (deleteAdmin(delAdminUsername,
                                    ↪ currentAdminUsername,
                                    ↪ currentAdminPassword))
                                    ↪ {
321                                     printf("Admin %s deleted
                                            ↪ successfully by
                                            ↪ %s.\n",
                                            ↪ delAdminUsername,
                                            ↪ currentAdminUsername);
322                                 } else {
323                                     printf("Error: Could not
                                            ↪ delete admin.\n");
324                                 }
325                                 break;
326                             }
327                         case 7: {
328                             char
                                ↪ delHospitalCode[MAX_HOSPITAL_CODE_
329                             char
                                ↪ currentAdminPassword[MAX_PASSWORD_
330                             printf("To continue, please
                                ↪ enter your (%s)
                                ↪ current password: ",
                                ↪ currentAdminUsername);
331                             getPassword(currentAdminPassword,
                                ↪ sizeof(currentAdminPassword));
332                             if
                                ↪ (validateAdmin(currentAdminUsernam
                                ↪ currentAdminPassword))
                                ↪ {
333                                 printf("Password
                                        ↪ verified
                                        ↪ successfully.\n");
334                             } else {
335                                 printf("Error: Invalid
                                        ↪ password.\n");
336                                 break;
337                             }
338                             printf("Enter Hospital Code
                                ↪ to Delete: ");
```

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

```c
                               case 12:
                                   printf("Exiting admin
                                      ↪ panel...\n");
                                   isAdmin = false;
                                   break;
                               default:
                                   printf("Error: Invalid option.
                                      ↪ Please try again.\n");
                           }
                       }
                   } else {
                       printf("Error: Invalid admin
                          ↪ credentials.\n");
                   }
                   break;
               }
           case 5:
               printf("Exiting...\n");
               break;
           default:
               printf("Error: Invalid option. Please try
                  ↪ again.\n");
           }
           if (choice == 5) {
               break;
           }
       }
       freeAdmin();
       freeHospital();
       freeBloodList();
       return 0;
   }
```

Listing A.1: main.c

A.1.2 admin_manager.c

```c
/*!
 * @file admin_manager.c
 * @brief Admin manager source file
 * @details This file contains the implementation of the
    ↪ admin manager module.
 *
 * @author CrimsonCare Team
 * @date 2025-01-18
 *
 * @copyright
```

```
30  #include "../include/admin_manager.h"
31  #include "../include/misc.h"
32
33   /*!
34   * @brief Admin head pointer
35   * @details This pointer is used to track admin linkedlist
       ↪ on runtime.
36   */
37  Admin* adminHead = NULL;
38
39  /*!
40   * @name saveAdminCredentials
41   * @brief Save admin credentials to file
42   * @details This function saves the linkedlist data
43   * from 'adminHead' to the file
       ↪ 'resources/db/admin_credentials.dat'.
```

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

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

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

```
163      while (temp != NULL) {
164          if (strcmp(temp->username, username) == 0) {
165              return true;
166          }
167          temp = temp->next;
168      }
169      return false;
170  }
171
172  /*!
173   * @name validateAdmin
174   * @brief Validate admin credentials
175   * @details This function traverses the 'adminHead'
         ↪ 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
         ↪ found in the linkedlist,
185   * the function returns true, otherwise false.
186   *
187   * @exception If the @p username or @p password is empty, an
         ↪ error message is displayed.
188   * @exception If the @p username is invalid, an error
         ↪ message is displayed.
189   */
190  bool validateAdmin(const char* username, const char*
       ↪ password) {
191      if (strcmp(username, "") == 0 || strcmp(password, "") ==
           ↪ 0) {
192          printf("Error: Admin credentials cannot be
               ↪ empty.\n");
193          return false;
194      }
195
196      if (!checkUsername(username)) {
197          printf("Error: Invalid username. Username can only
               ↪ contain lowercase letters and digits.\n");
198          return false;
199      }
200
201      Admin* temp = adminHead;
202      while (temp != NULL) {
```

```
203          if (strcmp(username, temp->username) == 0 &&
                ↪ 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
         ↪ 'adminHead' linkedlist
215   * and saves the updated linkedlist to the file
         ↪ 'resources/db/admin_credentials.dat'.
216   *
217   * @param[in] username Admin username
218   * @param[in] password Admin password
219   * @param[in] currentAdminUsername Current admin username
220   * @param[in] currentAdminPassword Current admin password
221   *
222   * @return True if admin is added, False otherwise
223   *
224   * @note The file path 'resources/db' is relative to the
         ↪ project root directory.
225   * Make sure that the folder exists also to run the program
         ↪ from the root directory.
226   *
227   * @pre @p username and @p password are not empty
228   * @pre @p currentAdminUsername and @p currentAdminPassword
         ↪ are not empty
229   * @pre @p username and @p currentAdminUsername are valid
230   * @post If the @p username and @p password are not found in
         ↪ the linkedlist,
231   * the function adds the new admin to the linkedlist and
         ↪ saves the updated linkedlist to the file.
232   *
233   * @exception If the @p username or @p password is empty, an
         ↪ error message is displayed.
234   * @exception If the @p username or @p currentAdminUsername
         ↪ is invalid, an error message is displayed.
235   * @exception If the pair of @p currentAdminUsername and @p
         ↪ currentAdminPassword is invalid,
236   * means that the current admin credentials are not valid,
         ↪ an error message is displayed.
237   * @exception If the @p username already exists, an error
         ↪ message is displayed.
```

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

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

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

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

```c
            printf("Error: Invalid username. Username can only
                ↪ contain lowercase letters and digits.\n");
            return false;
        }

        if (!validateAdmin(username, oldPassword)) {
            printf("Error: Invalid password.\n");
            return false;
        }

        if (strcmp(newPassword, "") == 0) {
            printf("Error: New password cannot be empty.\n");
            return false;
        }

        Admin* temp = adminHead;
        while (temp != NULL) {
            if (strcmp(username, temp->username) == 0 &&
                ↪ strcmp(oldPassword, temp->password) == 0) {
                strncpy(temp->password, newPassword,
                    ↪ sizeof(temp->password) - 1);
                temp->password[sizeof(temp->password) - 1] =
                    ↪ '\0';
                saveAdminCredentials();
                return true;
            }
            temp = temp->next;
        }
        return false;
}

/*!
 * @name displayAdmin
 * @brief Display all admins
 * @details This function displays all admins in the
     ↪ 'adminHead' linkedlist.
 *
 * @post If the 'adminHead' linkedlist is not empty,
 * the function displays all admins in the linkedlist.
 */
void displayAdmin(void) {
    Admin* temp = adminHead;
    printf("\nRegistered Admins:\n");
    while (temp != NULL) {
        printf("\tUsername: %s\n", temp->username);
        temp = temp->next;
        if (temp != NULL) {
            printf("\t--------------------------------\n");
```

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

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

```
 92        }
 93
 94        BloodStock* newGroup =
              ↪ (BloodStock*)malloc(sizeof(BloodStock));
 95        if (!newGroup) {
 96            printf("Error allocating memory for blood group:
                  ↪ %s\n", strerror(errno));
 97            return false;
 98        }
 99        strncpy(newGroup->bloodGroup, bloodGroup,
              ↪ BLOOD_GROUP_NAME_LENGTH - 1);
100        newGroup->bloodGroup[BLOOD_GROUP_NAME_LENGTH - 1] = '\0';
101        newGroup->price = price;
102        newGroup->quantity = quantity;
103        newGroup->id = id;
104        newGroup->next = NULL;
105
106        if (bloodHead == NULL) {
107            bloodHead = newGroup;
108        } else {
109            BloodStock* temp = bloodHead;
110            while (temp->next != NULL) {
111                temp = temp->next;
112            }
113            temp->next = newGroup;
114        }
115        return true;
116    }
117
118    /*!
119     * @name initializeBloodGroups
120     * @brief Initialize blood groups
121     * @details This function helps to initialize the default
             ↪ blood groups
122     * to the 'bloodHead' linkedlist.
123     *
124     * @post The blood groups are added to the 'bloodHead'
             ↪ linkedlist.
125     *
126     * @exception If adding blood group fails, an error message
             ↪ is displayed.
127     */
128    void initializeBloodGroups(void) {
129        for (uint8_t i = 0; i < (sizeof(availableBloodGroups) /
              ↪ sizeof(availableBloodGroups[0])); i++) {
130            if (!addBloodGroup(i + 1, availableBloodGroups[i],
                  ↪ 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
         ↪ updated linkedlist.
180    *
181    * @exception If the @p id is not found in the 'bloodHead'
         ↪ linkedlist, an error message is displayed.
182    */
183   bool updateBloodQuantity(uint32_t id, uint32_t newQuantity) {
184       if (!isValidBloodGroup(id)) {
185           printf("Error: Invalid blood group id.\n");
186           return false;
187       }
188
189       BloodStock* temp = bloodHead;
190       while (temp != NULL) {
191           if (temp->id == id) {
192               temp->quantity = newQuantity;
193               saveBloodGroups();
194               return true;
195           }
196           temp = temp->next;
197       }
198       return false;
199   }
200
201   /*!
202    * @name updateBloodPrice
203    * @brief Update blood price
204    * @details This function updates the blood price of the
         ↪ given blood group id
205    * by traversing the 'bloodHead' linkedlist.
206    *
207    * @param[in] id Blood group id
208    * @param[in] newPrice New price
209    *
210    * @return True if blood price is updated, False otherwise
211    *
212    * @post If the @p id is found in the 'bloodHead' linkedlist,
213    * the function updates the blood price and saves the
         ↪ updated linkedlist.
214    *
```

```
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",
                     ↪ strerror(errno));
258              freeBloodList();
259              return;
260          }
261      }
262
263      while (1) {
264          BloodStock* newBlood =
                 ↪ (BloodStock*)malloc(sizeof(BloodStock));
265          if (!newBlood) {
266              printf("Error allocating memory for blood group:
                     ↪ %s\n", strerror(errno));
267              freeBloodList();
268              fclose(file);
269              return;
270          }
271
272          if (fscanf(file, "%u %s %f %u", &newBlood->id,
                 ↪ newBlood->bloodGroup, &newBlood->price,
                 ↪ &newBlood->quantity) != 4) {
273              free(newBlood);
274              fclose(file);
275              break;
276          }
277
278          newBlood->next = NULL;
279
280          if (bloodHead == NULL) {
281              bloodHead = newBlood;
282          } else {
283              BloodStock* temp = bloodHead;
284              while (temp->next != NULL) {
285                  temp = temp->next;
286              }
287              temp->next = newBlood;
288          }
289      }
290
291      fclose(file);
292  }
293
294  /*!
295   * @name isBloodAvailable
296   * @brief Check if blood is available for a specific
           ↪ 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
               ↪ the 'availableBloodGroups' array.
356   *
357   * @post The available blood groups are displayed.
358   */
359  void displayBloodGroups(void) {
360      for (uint32_t i = 0; i < (sizeof(availableBloodGroups) /
               ↪ sizeof(availableBloodGroups[0])); i++) {
361          printf("%u. %s\n", i + 1, availableBloodGroups[i]);
362      }
363  }
364
365  /*!
366   * @name displayBloodStocks
367   * @brief Display all blood stocks
368   * @details This function displays all the blood stocks in
               ↪ the 'bloodHead' linkedlist.
369   *
370   * @note If Price or Quantity is not available, it is
               ↪ displayed as N/A.
371   *
372   * @post The blood stocks are displayed.
373   */
374  void displayBloodStocks(void) {
375      BloodStock* temp = bloodHead;
376      if (temp == NULL) {
377          printf("No blood available.\n");
378          return;
379      }
380      printf("\nAvailable Blood:\n");
381      while (temp != NULL) {
382          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
        ↪ functions 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   #include "../include/hospital_manager.h"
32   #include "../include/misc.h"
33
34    /*!
35     * @brief Hospital head pointer
36     * @details This pointer is used to track hospital
          ↪ linkedlist on runtime.
37     */
38   Hospital* hospitalHead = NULL;
39
40   /*!
41    * @name loadHospitals
42    * @brief Load hospitals from file
43    * @details This function loads the hospitals from the file
         ↪ 'resources/db/hospitals.txt'
44    * and stores it in the 'hospitalHead' linkedlist.
45    *
46    * @note The file path 'resources/db' is relative to the
         ↪ project root directory.
47    * Make sure that the folder exists also to run the program
         ↪ from the root directory.
48    *
49    * @post If the file is not found, the function does
         ↪ nothing. Otherwise, the hospitals
50    * are loaded from the file and stored in the 'hospitalHead'
         ↪ linkedlist.
51    *
52    * @exception fopen() - If the file cannot be opened, an
         ↪ error message is displayed,
53    * also the function frees the 'hospitalHead' linkedlist.
54    * @exception malloc() - If memory allocation fails, an
         ↪ error message is displayed,
55    * also the function frees the 'hospitalHead' linkedlist.
56    */
57   void loadHospitals(void) {
58       errno = 0;
59       FILE* file = fopen("resources/db/hospitals.txt", "r");
60       if (!file) {
61           if (errno == ENOENT) {
62               return;
63           } else {
```

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

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

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

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

```
227    * @param[in] code Hospital code
228    *
229    * @return True if hospital code is valid, False otherwise
230    *
231    * @pre @p code is not empty and valid
232    * @post If the @p code is found in the 'hospitalHead'
           ↪ linkedlist,
233    * the function returns true. Otherwise, it returns false.
234    *
235    * @exception If the @p code is empty or invalid, an error
           ↪ message is displayed.
236    */
237   bool validateHospitalCode(const char* code) {
238       if (strcmp(code, "") == 0) {
239           printf("Error: Hospital code cannot be empty.\n");
240           return false;
241       }
242
243       if (containsPipe(code)) {
244           printf("Error: Hospital code cannot contain a pipe
                  ↪ character.\n");
245           return false;
246       }
247
248       Hospital* temp = hospitalHead;
249       while (temp != NULL) {
250           if (strcmp(temp->code, code) == 0) {
251               return true;
252           }
253           temp = temp->next;
254       }
255       return false;
256   }
257
258   /*!
259    * @name deleteHospital
260    * @brief Delete hospital
261    * @details This function deletes the hospital with the
           ↪ given code
262    * by traversing the 'hospitalHead' linkedlist.
263    *
264    * @param[in] code Hospital code
265    * @param[in] adminUsername Admin username
266    * @param[in] adminPassword Admin password
267    *
268    * @return True if hospital is deleted, False otherwise
269    *
270    * @pre @p code is not empty and valid
```

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

```c
307              printf("Error: Hospital code is invalid.\n");
308              return false;
309          }
310
311          Hospital* current = hospitalHead;
312          Hospital* prev = NULL;
313          while (current != NULL) {
314              if (strcmp(current->code, code) == 0) {
315                  if (prev == NULL) {
316                      hospitalHead = current->next;
317                  } else {
318                      prev->next = current->next;
319                  }
320                  saveHospitals();
321                  return true;
322              }
323              prev = current;
324              current = current->next;
325          }
326          return false;
327      }
328
329      /*!
330       * @name getHospitalNameByCode
331       * @brief Get hospital name by code
332       * @details This function gets the hospital name by the
333       *     ↪ given code
334       * by traversing the 'hospitalHead' linkedlist.
335       *
336       * @param[in] code Hospital code
337       *
338       * @return Hospital name or NULL if not found
339       *
340       * @pre @p code is not empty and valid
341       * @post If the @p code is found in the 'hospitalHead'
342       *     ↪ linkedlist,
343       * the function returns the hospital name. Otherwise, it
344       *     ↪ returns NULL.
345       *
346       * @exception If the @p code is empty or invalid, an error
347       *     ↪ message is displayed.
348       */
349      char* getHospitalNameByCode(const char* code) {
350          if (strcmp(code, "") == 0) {
351              printf("Error: Hospital code cannot be empty.\n");
352              return NULL;
353          }
354
```

```
351        if (containsPipe(code)) {
352            printf("Error: Hospital code cannot contain a pipe
                  ↪ 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
             ↪ displayed in a formatted manner.
378     */
379    void displayHospitals(void) {
380        Hospital* temp = hospitalHead;
381        if (temp == NULL) {
382            printf("No hospitals registered yet.\n");
383            return;
384        }
385        printf("\nRegistered Hospitals:\n");
386        while (temp != NULL) {
387            printf("\tCode: %s\n"
388                "\tName: %s\n"
389                "\tLocation: %s\n", temp->code, temp->name,
                  ↪ temp->location);
390            temp = temp->next;
391            if (temp != NULL) {
392                printf("\t-------------------------------\n");
393            }
394        }
395    }
```

```
396
397  /*!
398   * @name freeHospital
399   * @brief Free hospital list from memory
400   * @details This function frees the 'hospitalHead' linkedlist
401   * from memory by traversing it.
402   *
403   * @post The 'hospitalHead' linkedlist is freed from memory.
404   */
405  void freeHospital(void) {
406      Hospital* current = hospitalHead;
407      while (current != NULL) {
408          Hospital* temp = current;
409          current = current->next;
410          free(temp);
411      }
412      hospitalHead = NULL;
413  }
```

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
          ↪ person obtaining a copy
15    * of this software and associated documentation files (the
          ↪ "Software"), to deal
16    * in the Software without restriction, including without
          ↪ limitation the rights
17    * to use, copy, modify, merge, publish, distribute,
          ↪ sublicense, and/or sell
18    * copies of the Software, and to permit persons to whom the
          ↪ Software is
19    * 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   #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|%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",
              ↪ getBloodGroupById(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) {
```

```c
            printf("Enter the date and time of donation
                ↪ (YYYY-MM-DD): ");
            fgets(date, sizeof(date), stdin);
            date[strcspn(date, "\n")] = 0;
            if (!isValidDate(date)) {
                printf("Error: Invalid date format.\n");
                return false;
            }
            formatDate(date);
    } else {
            BloodStock* stock = bloodHead;
            while (stock != NULL) {
                if (stock->id == bloodId) {
                    if (stock->quantity < quantity) {
                        printf("Not enough stock for blood
                            ↪ group: %s. Available quantity:
                            ↪ %u\n", getBloodGroupById(bloodId),
                            ↪ stock->quantity);
                        return false;
                    }

                    stock->quantity -= quantity;
                    saveBloodGroups();
                    break;
                }
                stock = stock->next;
            }
            time_t now = time(NULL);
            strftime(date, sizeof(date), "%Y-%m-%d",
                ↪ localtime(&now));
    }

    if (type == SELL) {
        srand(time(NULL));
        sprintf(token, "TOKEN_%d", rand() % 10000);
        printf("Sell token generated for %s: %s\n", name,
            ↪ token);
    }

    if (!logTransaction(type, name, bloodId, quantity, date,
        ↪ type == SELL ? token : NULL)) {
        return false;
    }

    return true;
}

/*!
```

```c
    * @brief Display all transactions
    * @details This function displays all transactions from the
        ↪ file 'resources/db/transactions.log'.
    *
    * @pre The file 'resources/db/transactions.log' exists.
    * @post All transactions are displayed.
    *
    * @exception If the file 'resources/db/transactions.log'
        ↪ cannot be opened, an error message is displayed.
    */
void displayTransactions(void) {
    errno = 0;
    FILE* file = fopen("resources/db/transactions.log", "r");
    if (!file) {
        if (errno == ENOENT) {
            printf("No registered transactions found.\n");
        } else {
            printf("Error opening transaction log file:
                ↪ %s\n", strerror(errno));
        }
        return;
    }

    char line[256];
    bool hasLogs = false;
    bool firstLog = true;
    char prevLine[256] = { 0 };

    while (fgets(line, sizeof(line), file) != NULL) {
        char type[MAX_TRANSACTION_TOKEN_LENGTH] = "";
        char name[MAX_TRANSACTION_TOKEN_LENGTH] = "";
        uint32_t bloodId = 0;
        uint32_t quantity = 0;
        char date[MAX_TRANSACTION_TOKEN_LENGTH] = "";
        char token[MAX_TRANSACTION_TOKEN_LENGTH] = "";

        if (firstLog) {
            printf("\nRegistered Transactions:\n");
            firstLog = false;
        }

        if (sscanf(line, "%[^|]|%[^|]|%u|%u|%[^|]|%[^|\n]",
            type,
            name,
            &bloodId,
            &quantity,
            date,
            token) >= 5) {
```

```
259
260             hasLogs = true;
261
262             if (prevLine[0] != '\0') {
263                 printf("\t----------------------------------\n");
264             }
265
266             printf("\tType: %s\n"
267                 "\tEntity: %s\n"
268                 "\tBlood Group: %s\n"
269                 "\tQuantity: %u\n"
270                 "\tDate: %s",
271                 type,
272                 name,
273                 getBloodGroupById(bloodId),
274                 quantity,
275                 date);
276
277             if (token[0] != '\0') {
278                 printf("\n\tToken: %s\n", token);
279             }
280
281             strncpy(prevLine, line, sizeof(prevLine) - 1);
282             prevLine[sizeof(prevLine) - 1] = '\0';
283         }
284     }
285
286     if (!hasLogs) {
287         printf("No registered transactions found.\n");
288     }
289
290     fclose(file);
291 }
```

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
       ↪ functions for the misc module.
6  *
7  * @author CrimsonCare Team
8  * @date 2025-01-22
```

```c
 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   #include "../include/misc.h"
32
33    /*!
34     * @name displayWelcomeMessage
35     * @brief Display welcome message
36     * @details This function displays the welcome message
37     * by reading from the file 'resources/assets/misc/cc.txt'.
38     *
39     * @note The file 'resources/assets/misc/cc.txt' is a text
         ↪ file
40     * that contains the welcome message.
41     *
42     * @pre The file 'resources/assets/misc/cc.txt' exists.
43     *
```

```c
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");
```

```c
        printf("7. Delete Hospital\n");
        printf("8. Display Admins\n");
        printf("9. Display Hospitals\n");
        printf("10. Display Blood Stocks\n");
        printf("11. Display Transactions\n");
        printf("12. Exit\n");
        printf("Select an option: ");
}

/*!
 * @name clearInputBuffer
 * @brief Clear input buffer
 * @details This function clears the input buffer
 * by reading until a newline character is encountered.
 *
 * @post The input buffer is cleared.
 */
void clearInputBuffer(void) {
    int c;
    while ((c = getchar()) != '\n' && c != EOF);
}

/*!
 * @name checkUsername
 * @brief Check if username is valid
 * @details This function checks if a username is valid.
 *
 * @param[in] str Username to check
 *
 * @return True if username is valid, False otherwise
 *
 * @note Username can only contain lowercase letters and
 *    ↪ digits.
 *
 * @pre @p str is not empty
 * @post If the @p str is valid, the function returns true.
 * @ Otherwise, it returns false.
 */
bool checkUsername(const char* str) {
    while (*str) {
        if (!(*str >= 'a' && *str <= 'z') && !(*str >= '0'
            ↪ && *str <= '9')) {
            return false;
        }
        str++;
    }
    return true;
}
```

```
136
137  /*!
138   * @name containsPipe
139   * @brief Check if string contains pipe
140   * @details This function checks if a string contains a pipe
          ↪ character.
141   *
142   * @param[in] str String to check
143   *
144   * @return True if string contains pipe, False otherwise
145   *
146   * @pre @p str is not empty
147   * @post If the @p str contains a pipe character, the
          ↪ function returns true.
148   * Otherwise, it returns false.
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   *
166   * @param[in,out] password Password
167   * @param[in] size Password size
168   *
169   * @post Updates the @p password with the user's input
170   * through the pointer @p password.
171   */
172  void getPassword(char* password, size_t size) {
173  #ifdef _WIN32
174      size_t i = 0;
175      char ch;
176      while (i < size - 1) {
177
178          ch = getch();
179
180          if (ch == '\r') {
181
```

```c
                    break;
            } else if (ch == '\b') {

                if (i > 0) {
                    i--;
                    printf("\b \b");

                }
            } else {
                password[i++] = ch;
                printf("*");

            }
        }
    password[i] = '\0';

    printf("\n");
#else

    struct termios oldt, newt;
    tcgetattr(STDIN_FILENO, &oldt);

    newt = oldt;

    newt.c_lflag &= ~(ECHO);

    tcsetattr(STDIN_FILENO, TCSANOW, &newt);

    fgets(password, size);
    password[strcspn(password, "\n")] = 0;

    tcsetattr(STDIN_FILENO, TCSANOW, &oldt);

    printf("\n");
#endif
}

/*!
 * @name isLeapYear
 * @brief Check if year is leap year
 * @details This function checks if a year is a leap year.
 *
 * @param[in] year Year to check
 *
 * @return True if year is leap year, False otherwise
 */
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
```

```c
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.
285     *
286     * @param[in,out] date Date to format
287     *
288     * @pre @p date is not empty
289     * @post The @p date is formatted to the format yyyy-mm-dd
290     *     and updates the @p date through the pointer @p date.
290     *
291     * @exception If the @p date is empty, an error message is
292     *     displayed.
292     * @exception If the @p date is invalid, an error message is
293     *     displayed.
293     */
294    void formatDate(char* date) {
295        if (strcmp(date, "") == 0) {
296            printf("Error: Date cannot be empty.\n");
297            return;
298        }
299
300        if (!isValidDate(date)) {
301            printf("Error: Invalid date format.\n");
302            return;
303        }
304
305        int year, month, day;
306        sscanf(date, "%d-%d-%d", &year, &month, &day);
307        sprintf(date, "%04d-%02d-%02d", year, month, day);
308    }
```

Listing A.6: misc.c

A.2   Header Files

A.2.1   admin_manager.h

```c
/*!
 * @file admin_manager.h
 *
 * @brief Admin manager header file
 * @details This file contains the declarations of the
     ↪ functions and structures for the admin manager module.
 *
 * @author CrimsonCare Team
 * @date 2025-01-18
 *
 * @copyright
 * Copyright (c) 2025 CrimsonCare Team
 *
 * Permission is hereby granted, free of charge, to any
     ↪ person obtaining a copy
 * of this software and associated documentation files (the
     ↪ "Software"), to deal
 * in the Software without restriction, including without
     ↪ limitation the rights
 * to use, copy, modify, merge, publish, distribute,
     ↪ sublicense, and/or sell
 * copies of the Software, and to permit persons to whom the
     ↪ Software is
 * furnished to do so, subject to the following conditions:
 *
 * 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
 * LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR
     ↪ OTHERWISE, ARISING FROM,
 * OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR
     ↪ OTHER DEALINGS IN THE
 * SOFTWARE.
 */
```

```
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
             ↪ username.
45    */
46  #define MAX_USERNAME_LENGTH 20
47
48    /*!
49     * @def MAX_PASSWORD_LENGTH
50     * @brief Maximum password length
51     * @details This macro defines the maximum length of the
             ↪ password.
52     */
53  #define MAX_PASSWORD_LENGTH 20
54
55    /*!
56     * @struct Admin
57     * @brief Admin structure
58     * @details This structure represents an admin user in
             ↪ the system.
59     */
60  typedef struct Admin {
61      char username[MAX_USERNAME_LENGTH]; /*!< Admin username
             ↪ */
62      char password[MAX_PASSWORD_LENGTH]; /*!< Admin password
             ↪ */
63      struct Admin* next; /*!< Next admin */
64  } Admin;
65
66  /*!
67   * @name saveAdminCredentials
68   * @brief Save admin credentials to file
69   * @details This function saves the linkedlist data
70   * from 'adminHead' to the file
             ↪ 'resources/db/admin_credentials.dat'.
71   * '.dat' is used to store credentials in binary format for
             ↪ 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
           ↪ is displayed.
118    * @exception If the @p username is invalid, an error
           ↪ message is displayed.
119    */
120   bool adminExists(const char* username);
121
122   /*!
123    * @name validateAdmin
124    * @brief Validate admin credentials
125    * @details This function traverses the `adminHead`
           ↪ linkedlist
126    * and checks if the pair of username and password match.
127    *
128    * @param[in] username Admin username
129    * @param[in] password Admin password
130    *
131    * @return True if credentials are valid, False otherwise
132    *
133    * @pre @p username and @p password are not empty and valid
134    * @post If the pair of @p username and @p password are
           ↪ found in the linkedlist,
135    * the function returns true, otherwise false.
136    *
137    * @exception If the @p username or @p password is empty, an
           ↪ error message is displayed.
138    * @exception If the @p username is invalid, an error
           ↪ message is displayed.
139    */
140   bool validateAdmin(const char* username, const char*
         ↪ password);
141
142   /*!
143    * @name addAdmin
144    * @brief Add admin
145    * @details This function adds a new admin to the
           ↪ `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
           ↪ are not empty
187    * @pre @p username and @p currentAdminUsername are valid
188    * @post If the @p username is found in the linkedlist,
189    * the function deletes the admin from the linkedlist and
           ↪ saves the updated linkedlist to the file.
190    *
191    * @exception If the pair of @p currentAdminUsername and @p
           ↪ currentAdminPassword is invalid,
192    * means that the current admin credentials are not valid,
           ↪ an error message is displayed.
193    * @exception If the @p username does not exist, an error
           ↪ message is displayed.
194    * @exception If the @p username is the same as the current
           ↪ admin username, an error message is displayed.
195    * @exception If the @p username or @p currentAdminUsername
           ↪ is invalid, an error message is displayed.
196    */
197   bool deleteAdmin(const char* username, const char*
         ↪ currentAdminUsername, const char*
         ↪ currentAdminPassword);
198
199   /*!
200    * @name changeAdminPassword
201    * @brief Change admin password
202    * @details This function changes the password of an admin
           ↪ in the 'adminHead' linkedlist
203    * and saves the updated linkedlist.
204    *
205    * @param[in] username Admin username
206    * @param[in] oldPassword Old password
207    * @param[in] newPassword New password
208    *
209    * @return True if password is changed, False otherwise
210    *
211    * @pre @p username and @p oldPassword are not empty
212    * @pre @p username is valid
213    * @pre @p newPassword is not empty
214    * @post If the pair of @p username and @p oldPassword are
           ↪ 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
          ↪ group name.
48     */
49   #define BLOOD_GROUP_NAME_LENGTH 4

50

51    /*!
52     * @struct BloodStock
53     * @brief Blood stock structure
54     * @details This structure represents a blood stock in the
          ↪ system.
55     */
56   typedef struct BloodStock {
57       float price; /*!< Blood price */
58       uint32_t id; /*!< Blood group id */
59       uint32_t quantity; /*!< Blood quantity */
60       char bloodGroup[BLOOD_GROUP_NAME_LENGTH]; /*!< Blood
            ↪ group name */
61       struct BloodStock* next; /*!< Next blood stock */
62   } BloodStock;

63

64   /*!
65    * @brief Globally exposed blood stock head pointer
66    * @details This pointer is used to track blood stock
         ↪ linkedlist on runtime.
67    */
68   extern BloodStock* bloodHead;

69

70   /*!
71    * @name isValidBloodGroup
72    * @brief Check if blood group is valid
73    * @details This function checks if the given blood group id
         ↪ is valid
74    * by checking the size of the `availableBloodGroups` array.
75    *
76    * @param[in] id Blood group id
77    *
78    * @return True if blood group is valid, False otherwise
79    *
80    * @post If the @p id is within the range of the
         ↪ `availableBloodGroups` array,
81    * 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
              ↪ 'bloodHead' linkedlist.
89    *
90    * @param[in] id Blood group id
91    * @param[in] bloodGroup Blood group name
92    * @param[in] price Blood group price
93    * @param[in] quantity Blood group quantity
94    *
95    * @return True if blood group is added, False otherwise
96    *
97    * @pre @p id is valid
98    * @pre @p bloodGroup is not empty
99    * @post Updates the blood stock in the 'bloodHead'
              ↪ linkedlist.
100   *
101   * @exception If the @p bloodGroup is empty, an error
              ↪ message is displayed.
102   * @exception If the @p id is not valid, an error message is
              ↪ displayed.
103   * @exception malloc() - If the memory allocation for the
              ↪ new blood group fails, an error message is displayed.
104   */
105  bool addBloodGroup(uint32_t id, const char* bloodGroup,
          ↪ float price, uint32_t quantity);
106
107  /*!
108   * @name initializeBloodGroups
109   * @brief Initialize blood groups
110   * @details This function helps to initialize the default
              ↪ blood groups
111   * to the 'bloodHead' linkedlist.
112   *
113   * @post The blood groups are added to the 'bloodHead'
              ↪ linkedlist.
114   *
115   * @exception If adding blood group fails, an error message
              ↪ is displayed.
116   */
117  void initializeBloodGroups(void);
118
119  /*!
120   * @name saveBloodGroups
```

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

```
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
        ↪ updated linkedlist.
165   *
166   * @exception If the @p id is not found in the 'bloodHead'
        ↪ linkedlist, an error message is displayed.
167   */
168  bool updateBloodPrice(uint32_t id, float newPrice);
169
170  /*!
171   * @name loadBloodGroups
172   * @brief Load blood groups from file
173   * @details This function loads the blood groups from the
        ↪ file 'resources/db/blood_data.txt'
174   * to the 'bloodHead' linkedlist.
175   *
176   * @note The file path 'resources/db' is relative to the
        ↪ project root directory.
177   * Make sure that the folder exists also to run the program
        ↪ from the root directory.
178   *
179   * @post The blood groups are loaded to the 'bloodHead'
        ↪ linkedlist.
180   *
181   * @exception fopen() - If the file cannot be opened, an
        ↪ error message is displayed.
182   * @exception malloc() - If the memory allocation for the
        ↪ new blood group fails, an error message is displayed.
183   */
184  void loadBloodGroups(void);
185
186  /*!
187   * @name isBloodAvailable
188   * @brief Check if blood is available for a specific
        ↪ transaction type
189   * @details This function checks if blood is available for a
        ↪ specific transaction type
190   * by traversing the 'bloodHead' linkedlist.
191   *
192   * @param[in] id Blood group id, null if to check for any
        ↪ blood
193   * @param[in] type Transaction type
194   *
195   * @return True if blood is available, False otherwise
196   *
197   * @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
          ↪ message is displayed.
203   * @exception If the @p id is not null and is not valid, an
          ↪ error message is displayed.
204   */
205 bool isBloodAvailable(uint32_t* id, TransactionType type);
206
207 /*!
208   * @name displayBloodGroups
209   * @brief Display all blood groups
210   * @details This function displays all the blood groups in
          ↪ the 'availableBloodGroups' array.
211   *
212   * @post The available blood groups are displayed.
213   */
214 void displayBloodGroups(void);
215
216 /*!
217   * @name displayBloodStocks
218   * @brief Display all blood stocks
219   * @details This function displays all the blood stocks in
          ↪ the 'bloodHead' linkedlist.
220   *
221   * @note If Price or Quantity is not available, it is
          ↪ displayed as N/A.
222   *
223   * @post The blood stocks are displayed.
224   */
225 void displayBloodStocks(void);
226
227 /*!
228   * @name getBloodGroupById
229   * @brief Get blood group by id
230   * @details This function returns the blood group name by
          ↪ the given id.
231   *
232   * @param[in] id Blood group id
233   *
234   * @return Blood group name or NULL if not found
235   *
236   * @post If the @p id is valid, the function returns the
          ↪ blood group name.
237   *
```

```
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   *
```

```
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
```

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

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

```
210
211  #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
         ↪ functions and structures for the transaction 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 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
        ↪ 'resources/db/transactions.log'.
77   *
78   * @param[in] type Transaction type
79   * @param[in] name Hospital name
80   * @param[in] bloodId Blood group id
81   * @param[in] quantity Blood quantity
82   * @param[in] date Transaction date
83   * @param[in] token Transaction token
84   *
85   * @return True if transaction is logged, False otherwise
86   *
87   * @note The file path 'resources/db' is relative to the
        ↪ project root directory.
88   * Make sure that the folder exists also to run the program
        ↪ from the root directory.
89   *
90   * @pre @p type is either BUY or SELL
91   * @pre @p name is not empty and valid
92   * @pre @p bloodId is a valid blood group id
93   * @pre @p quantity is greater than 0
94   * @pre @p date is a valid date
95   * @post The transaction is logged to the file
        ↪ 'resources/db/transactions.log'.
96   *
97   * @exception If the file 'resources/db/transactions.log'
        ↪ cannot be opened,
98   * an error message is displayed.
99   * @exception If the @p type is not BUY or SELL, an error
        ↪ message is displayed.
100  * @exception If the @p name is empty or invalid, an error
        ↪ message is displayed.
101  * @exception If the @p bloodId is not a valid blood group
        ↪ id, an error message is displayed.
102  * @exception If the @p quantity is less than or equal to 0,
        ↪ an error message is displayed.
103  * @exception If the @p date is not a valid date, an error
        ↪ message is displayed.
104  */
105 bool logTransaction(TransactionType type, const char* name,
       ↪ uint32_t bloodId, uint32_t quantity, const char* date,
       ↪ const char* token);
106
107 /*!
```

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

```
144   *
145   * @exception If the file 'resources/db/transactions.log'
         ↪ cannot be opened, an error message is displayed.
146   */
147  void displayTransactions(void);
148
149  /*!
150   * @brief Free transaction list from memory
151   */
152  void freeTransaction(void);
153
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
         ↪ functions for the misc module.
6    *
7    * @author CrimsonCare Team
8    * @date 2025-01-22
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   *
```

```c
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
        ↪ file
56    * that contains the welcome message.
57    *
58    * @pre The file 'resources/assets/misc/cc.txt' exists.
59    *
60    * @post The welcome message is displayed.
61    */
62  void displayWelcomeMessage(void);
63
```

```
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
          ↪ digits.
102   *
103   * @pre @p str is not empty
104   * @post If the @p str is valid, the function returns true.
105   * Otherwise, it returns false.
106   */
107  bool checkUsername(const char* str);
108
109
110  /*!
```

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

```
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
           ↪ displayed.
164    * @exception If the @p date is invalid, an error message is
           ↪ displayed.
165    */
166   bool isValidDate(const char* date);
167
168   /*!
169    * @name formatDate
170    * @brief Format date to yyyy-mm-dd
171    * @details This function formats a date string to the
           ↪ format yyyy-mm-dd.
172    *
173    * @param[in,out] date Date to format
174    *
175    * @pre @p date is not empty
176    * @post The @p date is formatted to the format yyyy-mm-dd
           ↪ and updates the @p date through the pointer @p date.
177    *
178    * @exception If the @p date is empty, an error message is
           ↪ displayed.
179    * @exception If the @p date is invalid, an error message is
           ↪ displayed.
180    */
181   void formatDate(char* date);
182
183   #endif
```

Listing A.11: misc.h