Abstract

The Secure Bank Management System Developed with CERT Secure Coding Principles aims to recreate a real-life banking system scenario along with all its necessary functionalities. The real obstacles this project hopes to overcome is that of numerous potential flaws and security vulnerabilities that may arise, while maintaining a bank management system.

The project hopes to achieve this by mimicking the various tasks and operations that are executed on a day to day basis in a bank while explicitly providing appropriate countermeasures in compliance with CERT specified Rules and Recommendations as seen fit. These tasks include bank account management with administrator rights and privileges, customer-system interactions such as withdrawal/deposit and loan applications. Such operations are prone to several security risks which may lead to data privacy leaks, sensitive information discrepancies and worse. Several obvious vulnerabilities and many application specific vulnerabilities are dealt with varying degree of severity and reasoning.

By the end of this project, we hope to present a system capable and well prepared to handle any sort of security threat or potential vulnerability that may arise, by virtue of adopted mechanism, or design, hence providing any customer with a sense of confidence and satisfaction moving forward.

Table of Contents

1. Introduction

The Secure Bank Management System Software was developed using secured programming techniques, rules and recommendations adhering to the CERT standards using the C programming language.

Financial institutions such as banks play a significant and pivotal role in the balance of economy, and providing a sense of financial security to the general masses. However, since it has been widely accepted that testaments to manual labour in the form of documents alone is antiquated, and tedious, and also difficult for future references. This provides the scope for creating a bank management system, that can perform these tasks, and can help tone down monotonous to very minimalistic standards. The program is designed to illustrate the various daily general functions, and provide the typical services expected from a bank. These include (but are not limited to) services such as customer account management, monetary transaction management among other common duties.

The importance and magnitude of significance that banks have on the everyday functioning of a nation and to the general masses has already been postulated. This consequently means that banking and bank management systems are inherently required to be safe and secure in nature to utilize. Predicting vulnerabilities, and validating data at every step is one of the most demanding and crucial tasks. It is for this purpose, that care has been to enforce numerous standardized CERT Rules and Recommendations in the application, with the twofold goal of minimizing the risk involved in these operations, and providing a safe and secure banking experience for everybody.

It has also been ensured that software was developed in a manner that allows only authorized administrators/staff to navigate and use the system and associated data with ease - with easy to interpret prompts and user-friendly options provided. In order to demonstrate the trade-off between scalability and security that one may sometimes encounter whilst developing such systems, this version of the software has been built to accommodate a localized, particular and decentralized format with the data being stored natively on the custom local storage options available that can be navigated with ease and modified, if necessary.

2. Requirement Analysis

2.1. System Requirements

Processor: Any processor that can seamlessly handle the workload of C programs

Operating System: A flavor of Windows

Compiler: TDM-GCC, with Make tools installed and loaded in PATH

Development Environment: Any that supports development and debugging of C programs

Terminal: Any terminal that can run a C executable created by gcc.

2.2. Software Requirements

2.2.1. User Requirements

The target end-user group is expected to be individuals employed in a bank, with basic knowledge on how to operate computers, and with the required authorization to access to the data corresponding to the customers that the bank caters to.
The consumers for the application however would primarily consist of banks who would want to invest their resources into adopting or testing a secure bank management system.
The suite of features that the consumers can expect, and whose utilization the end users must familiarize themselves with, are akin to the conventional day to day affairs coordinated in a bank. The users can expect a coherent emulation of their daily businesses and responsibilities, through the means of the secure bank management system.

2.2.2. Application Requirements

The primary requirement and ambition that impel this project forward is the end goal of enabling end-users to use the application for carrying out their standard bank duties smoothly, and to provide a feeling of safety and a reality of security to the consumers.

The requirement for a computerized management system for banks is paramount and an inherently expected system in this era of technology and networking, with organizations constantly looking for further growth and expansion of their business ventures, which include banks as well. It helps reduce a lot of manual labor, and increase the overall efficiency of the bank in itself.

However, this brings to light, the first and primary concern one would have to address, while planning to build a bank management application; security and safety.

Banks on the whole, lose billions of dollars every year, because they are targets of tens of thousands of cyber-attacks, and those attempting to commit cyber-fraud. There are extensive reports continuously generated, detailed analysis formulated, and forthright statistics with numbers that don't lie computed that are constantly trying to observe the trends of security standards, and the cyber-attacks ensued upon the software that banks adopt. This explicates the round-the-clock patching and updating of software; all to provide a more airtight working model. Banks consequently have no choice but to invest in the top security measures they can adopt, which expectedly increases their investment margins into security as well. This demands safety, and secure reliability features offered by the application set to manage the entire bank system to be assured uncompromisingly.

The security of a computer system can be breached in a multitude of ways, to quote some, by system failure, theft, inappropriate usage, unauthorized access or computer viruses. Every time you engage in anything that involves a network the security of data security is being put at risk. The effects of a data security breach can be catastrophic. Not just in terms of the interruption to your standard operations but also the potential long-term damage to your reputation.
A few of the biggest issues and problems with that generally banks face are exemplified below.

Non-Compliance
Penalties for non-compliance can be abrupt for banks just not only financially but also in greatly increased oversight. Having compliance standard stimulates banks to focus on cybersecurity. When a bank stays compliant, it ensures that it is meeting consensus security and protecting the customer data.

Loss of data/funds
When a bank gets data breach, consumers lose time and money. A bank may recovery fraudulently spent money fully or partially but it does not work all the time. The action that occurs due to bank data breach is time-consuming, stressful, and full of pressure.

Sensitive Data
It is difficult for consumers to handle data breaches as they know their data and information is in the wrong hands. If a consumer's private data gets stolen, it floats out of control. Cybersecurity is more important to banks and financial institutions as they carry personal and private information of consumers.

These pressing issues that pose a constant threat to the financial institutions that serve as pillars of our society demands the necessity of a safe, and secure banking system.

2.2.3. Module requirements

It is common knowledge that C, for all its glory, does not explicitly support modular programming. However, this does not necessarily mean that developers cannot conceptualize the modules their applications require and associate corresponding source files and header files with said modules.

Such an approach will be adopted, in order to explain the modular breakdown of the secure bank management system, with regards to its requirements.
Although this modularization might come across as an abstract, or vague concept at the moment, as the structure of the system and its design intricacies are elaborated upon, it will be inherently clear which parts implicitly correspond to which module.

Fundamentally, the program can be broken down into modules that serve 3 core purposes. Consequently, these core modules, can be broken down further into sub-modules. We can summarize them as follows.

1. Bank Duties:
   This set of modules correspond to the files which provide the functions which constitute the crux of the secure bank management system. The aforementioned bank duties include (and are currently limited to) performing operations such as,
   - Opening new bank accounts for customers
   - Editing the details of existing customers
   - Viewing the complete bank details of a specific existing customer
   - Viewing the list of all the customers who presently have an account in the bank
   - Managing cash deposits and withdrawal
   - Processing loans
   - Closing customer accounts

2. Input Validation:
   Input validation is critical when it comes to building a secure system, particularly one that is meant to serve for an institution as critically prone to intrusive attacks if not careful enough. The input validation module provides secure input handling techniques and custom methods for accepting input, to the maximum extent of flexibility supplemented by the C programming language. This module contributes significantly to the basic safety requirements of the application.

3. Utility Functions:
   The utility functions module comprises of all the functions that are geared neither towards promoting the critical security concerns of the program, nor the fundamental obligations the bank fulfils. This module comprises of the functions that do not play a critical role in one particular domain, but in affirming that the program in itself works seamlessly. The smaller, and less complicated helper functions also come under the umbrella of the utility functions. Utility functions module helps integrate different aspects, and requirements of the program together, to form a single efficient operational unit, and ergo manifests utilitarianism in this manner.

2.3. Safety Requirements

2.3.1. Programming Language

The C language is statically typed, and weakly enforced.
C provides the advantage of being fast, and light-weight, a much-required feature for an intensive bank management system. However, this speed and agility, come at the cost of many potential vulnerabilities and pitfalls such as overstepping array boundaries, lack of garbage collection facilities, buffer overflows, uninitialized variables, etc., which attract hackers with ease. In an effort to promote security standards of applications, CERT has standardized rules and recommendations. The application will be built following these rules and recommendations, in an attempt to provide the user a safe and secure experience.

2.3.2. Memory

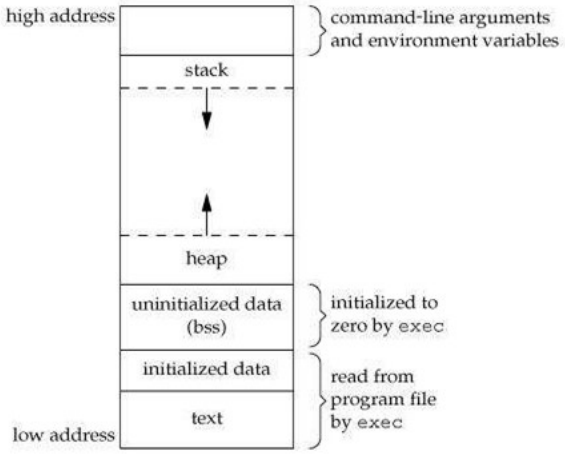


Figure 1. Memory Layout a C program (Tripathi)

When a C program is run on a computer, its executable image is loaded into the RAM of the computer in a very organized manner.
However, the topic of interest to us, must be the potential complications this may give way to.
Since the memory allocated for a program to run is finite, and it is impractical to consider a situation where the memory is indefinitely increased to meet the needs of the program, memory management must be performed acutely. Since the heap grows from a lower address to a higher address, and a stack grows from a higher address to a lower address, a symbiotic equilibrium must always be maintained between them, such that both of them have space to grow.
For the heap, this simply translates to, all memory dynamically allocated must be freed once it has served its purpose. With respect to the stack, this would require the program to be efficiently designed, taking advantage of scope and lifetime of a variable, in a manner that keeps the stack's constitution minimalistic at all times. Another simple design characteristic required would be returning of function calls and effectively handling function calls, so that the stack does not overflow, or the program does not run out of memory to use for execution. It can be veritably assured that the system developed in this project considers these potential issues, and handles them appropriately as well.

2.3.3. Expressions

Since the application's primary goal is to work as a bank management system, it is obvious that there will be parts of the system that handle numerical data and calculations. Operator precedence and associations is something that must always be kept in mind while dealing with calculations and mathematical operations. Thus, in order to prevent any numerical computation and calculation related mishaps, all expressions have been appropriately parenthesized. This not only prevents any unforeseen errors with respect to accounting and calculations, but also provides better readability.

2.3.4. Input / Output

I/O functions play a pivotal role in ensuring the security of an application. If I/O operations are not defined correctly, to the required specifications, then incorrect input can crash the system, which is not an acceptable outcome. Input functions such as *gets* are critically acclaimed to be unreliable. Even reliable functions such as *scanf*, cannot handle errors due to conversion (such as providing a character value when an integer is expected) efficiently. In order to prevent this scenario, custom functions have been defined for the particular data types used, which will perform the tasks of accepting input of the respective data type.

These custom functions implement features that include (but whose scope are not entirely limited to)

- accepting only integer values for numerical types of data
- accepting input only if it has at least a specified number of characters
    - considering only a specified number of characters out of the complete input
- not accepting a null input
- performing input validation as per requirements
- truncating and trimming spaces that may lead to issues while reading the file

Further, after every input is provided, the standard input buffer is cleared. This ensures that the buffer is clear at all times, thus ensuring that at no point, the input data overruns the buffer boundaries.

2.3.5. Security Specifications

We have already discussed the expected services from a secure Bank Management system; to provide users with a comfortable and reliable method of safely accessing their bank accounts and making seamless transactions without ever worrying about information leaks or privacy invasions. To meet the demands of today's world, a simple system would not suffice. This is where our program delivers. The system was built from the ground up, keeping in mind the various possible threats and violations, and discussing the viable rules and recommendations that could be adopted to counter these threats and violations, and to enforce security. The following describes some of the security measures, and rules and recommendations adopted, and their scope.

Fundamental security features

- The system allows only authorised users to access its database using a unique password, which is abstracted. If the password is incorrect due to any reason, it is considered as a security breach. This will immediately terminate the program to avoid any further discrepancies.
- Every successful transaction (with regard to function operations) will lead to calls, either to a predefined system function or other custom functions. This way the control is never lost within the system and hence will not allow unwelcome processes to interfere with the program. During any failed transactions, an error code is immediately returned. This allows the programmer to diagnose the issue appropriately. The error codes and names are designed such that anyone can interpret their significance.
- Many of the functions used take inputs from the user. For this very reason, a set of functions are defined called "input validation functions". Their purpose is to make sure users only provide the program with compatible inputs. Which means that any input that may violate the required data type, contain invalid symbols or signs or exceed the maximum storage space allocated to the variable, are automatically neglected. Failing to satisfy these conditions will lead to a failed transaction.
- The program makes use of files to store information regarding the bank accounts and loan applications. These files are accessed using constant pointers and any function that accesses these files make sure to close them once their needs are met. This way no file is left open beyond its requirement. Also, no file is accessed unnecessarily. This makes sure that the file operations are minimised to improve security and performance.

Rules and Recommendations – Implementations and Necessity

- Preprocessor-based rules and recommendations have been implemented to ensure there are no ambiguities with respect to macro definitions, and preprocessing.
- Declaration and Initialization associated rules and recommendations have been incorporated with careful reference to the lifecycle, and scope of the respective variables, and to promote usage of conventionally interpretable and readable identifiers.

- Rules and recommendations associated with expressions have been implemented in order to ensure integrity of variables, and to avoid any confusion with regards precedence of order of execution of expressions.
- Integer and Floating point based rules and recommendations have been implemented to ensure precision in variable's values, ensuring loop consistency, and correct execution of arithmetic operations, while preserving the integrity and characteristics of the numerical values.
- The rules and recommendations incorporated with respect to Arrays and Characters and Strings require a sound understanding of the working of arrays, and the requirements and issues presented by characters and strings. Enforcing correct memory allocation, bounds for arrays, and operations with valid types, without trying to perform invalid or ill-advised operations is important, and this can be observed, by virtue of instilling the aforementioned rules and recommendations.
- Memory Management based rules and recommendations have been enforced to ensure that effective and proper memory management, in line with realistic memory availability is observed throughout the life of the application, with sensitive information not being available at instances where they are not required.
- Secure Input/Output practices with respect to writing to and reading from files, as well as the buffer itself have been adopted, in line with their corresponding rules and recommendations.
- Error codes have been defined, and features for appropriately handling standard library errors, without relying on indeterminate values. As is convention, sensitive information to a great extent has not been hardcoded, all whilst adhering to constraints. This fulfils the usage of error handling and miscellaneous rules and recommendations as well.

The security features implemented for this project have been considered to be inherently necessary. This means, there aren't distinct rules/requirements that have been implemented that must be considered to have a pivotal significance, since all the rules and recommendations were implemented to have a relatively equal pivotal significance, and were seen as inherent requirements of the system, by the developers. Further analysis and descriptions of the rules and recommendations implemented have been described in sections dedicated for the same. It is hoped that all the numerous rules and recommendations incorporated are sufficient in order to provide a ground-framework for the implementation of a secure bank management system.

3. Design Specification

3.1. Data and Files Hierarchy

3.1.1. Data Types Adopted

1. structures
2. global variables
3. integers
4. void
5. floating point
6. character
7. array

3.1.2. Files Required

Header Files:

1. data.h
2. majorfunction.h
3. utilFunctions.h
4. inputValidation.h

Source Files:

1. bank.c
2. majorfunction.c
3. inputValidation.c
4. util.c
5. loan.c

Data file:

1. record.txt
2. loandetails.txt

Make File:

1. Makefile

Executable:

1. secure_bank_management_system.exe

3.2. Functional Specification

This section will serve as a comprehensive documentation and compendium of the software developed.

3.2.1. Header Files

1. data.h:
   - Structures defined:
     a. struct date:

Member Type	Member Identifier
int	day
int	month
int	year

     b. account:

Member Type	Member Identifier
int	age
long	acc_no
long	phone
float	amount
char array of 10 elements	acc_type
char array of 15 elements	citizenship
char array of 60 elements	address
char array of 60 elements	name
struct date	dob
struct date	deposit
struct date	withdraw
struct date	currloan

   - Macros Defined:

Macro Name	Purpose of Macro
FLUSH	FLUSH is used to flush out the stdin buffer
password	password is used to store the password that is required to log into the system
FORMAT	FORMAT is the format of data that is stored in the records file. This must be the format adhered to

	where reading from/writing into the records file.
PRINTFILE(x)	PRINTFILE(x) is the format that is required for writing into a file, with values stored in an account type structure named "x"
SCANFILE(x)	SCANFILE(x) is the format that is required for reading from a file, values which are then stored into an account type structure named "x"

- Global Variables defined:

Global Variable Type	Global Variable Identifier
int	i
int	j
int	main_exit
account	user

2. majorfunction.h:

- Function prototypes defined:

Function name	Function Return Type	Parameters (parameter_name – parameter_type)
create	void	None
transact	void	None
closeAccount	void	None
transfer	void	None
view_list	void	None
edit	void	None
see	void	None
loan	void	None

3. utilFunctions.h:
   - Function prototypes defined:

Function name	Function Return Type	Parameters (parameter_name – parameter_type)
start	void	None
menu	void	None
fordelay	void	• j – int
interest	float	• amount - float • rate – int • t - float
findAge	int	• current_date - int • current_month- int • current_year - int • birth_date - int • birth_month - int • birth_year- int
passwordAuthentication	int	None
calcEMI	float	• principal- float • ratepercent_per_annum - float • installments - int

4. inputValidation.h:
   - Function prototypes defined:

Function name	Function Return Type	Parameters (parameter_name – parameter_type)
getLong	long	None
getFloat	float	None
getInt	int	None
phoneNumber	long	None
removeSpaces	void	• str – char*
validDate	int	• day - int • month- int • year - int • day1 - int • month1- int • year1 - int

3.2.2. Source Files

In this section, the functionality provided by each function in each of the source files will be elaborated upon.

1. bank.c:

   - main:

   The main function calls the **start** function. Once the **start** function completes execution, the program returns 0 and exits.

2. util.c:

   - start:

   The start function first calls **passwordAuthentication**. If successfully authenticated, then this function calls the function named **menu.** If not successfully authenticated, then the user is given two choices, either to try again, or to quit.

   - menu:

   Menu requires the user to choose what operation they wish to perform. Depending on the operation selected, the respective function is called. After the function is called, every function provides an option to return to menu, or quit the program (or retry with different inputs in some cases). Depending upon the chosen option, the required method is carried out.

   - passwordAuthentication:

   Takes in a user password, abstracting each character with a '*' as it is typed in, and cross references it with the password defined (via macro). If it is a match, then 1 is returned, else, 0 is returned.

   - fordelay:

   Introduces a seamless delay in program execution.

   - findAge:

   Utility function to find age, when current date, and date of birth are provided.

   - interest:

   Utility function to calculate interest, given the time, rate, and principal amount

   - calcEMI:

   Calculates EMI and returns the interest per annum

3. inputValidation.c:

   - getInt:

   getInt helps sanitize input by allowing only integer values to be input. The function has been customized to maximum standards of security, where in the user can only input integer numbers, and the only non-integer keypress that will be registered will be the backspace. The integer value that is entered by the user is then returned.

   - getLong:

   getLong helps sanitize input by ensuring that only values of long type are entered. No other characters except for numbers will be accepted, and the input will be requested again if null or invalid characters are provided as input. The long value that is entered by the user is then returned.

   - getFloat:

   getFloat helps sanitize input by ensuring that only values of float type are entered. No other characters will be accepted except for numbers and a single decimal point, and the input will be requested again if invalid or null characters are provided as input. The float value that is entered by the user is then returned.

   - phoneNumber:

   phoneNumber ensures that only numerical values are entered, and that at least a specified number (in this case, 8) of numerical values must be input. In the event that more numerical values are input, then the first 8 numbers will only be considered. No other characters except for numerical values will be accepted, and the input will be requested again if invalid or null characters are provided input. The 8 digit integer that is entered by the user is then returned.

   - removeSpaces:

   Removes all spaces in a given string that is passes as a parameter.

   - validDate:

   Checks if the date that is passed as a parameter is a valid date or not, with respect to the expected and specified format, and present date. In the event that the date is valid, 1 is returned, else 0 is returned.

4. majorfunction.c:

   - create:

   Creates a new user account. The current date is calculated using the system time. Account Number is a unique value manually entered using **getLong**. If entered account number is in existence, then a new account number is requested. The potential account holder's name and address, desired account type is manually entered, with any spaces present removed using **removeSpaces**. The date of birth is validated using **validate**, and age is calculated using **findAge** function;

passing date of birth and current date as arguments. Citizenship details are also entered and have all spaces removed. In the event the citizenship number is not unique, then the name, and then the date of birth, are cross referenced in the records file. Only if the latter two are matching will the citizenship number be accepted. Else, it will be required to be provided as input again. The phone number is obtained using **phoneNumber.** The initial amount to be deposited is obtained using **getFloat**, and must be greater than $10. All entered details are then written onto a file to maintain records.

- transact:

Assists in performing withdrawals or making deposits. The account number is provided as input. If the account is non-existent, or existent and a Fixed type of account, then the user will not be allowed to make a transaction. They will be prompted with an option to try with a different value, or return to main menu. Else, the user will be allowed to perform transactions.

- view_list:

Displays the list of all customers with a few details such as their account number, postal address and a contact number.

- edit:

Allows editing and updating of the address and/or phone number details of an account associated with a specified account number.

- see:

Displays complete account details of the account associated with the account number

- loan:

Allows users to apply for loans based on their account information. The user must first have an account that qualifies the minimum criteria to apply for a loan. The user can select their preferred loan amount and a duration of monthly instalments based on the options provided. They are then provided with an amount to be paid as instalments where the user gets an option to confirm their loan application. The details of the loan, i.e., amount, period, date and EMI are recorded in a designated file.

- closeAccount:

Function assists in removing/deletion of an existing account. It receives the account number as input. If the account exists, admin rights need to be provided in order to close the account.

3.3. Program Flow

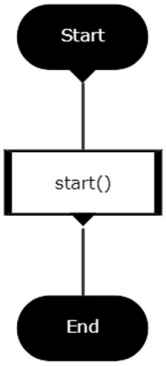
- The program first requires the user to login. This is so that only authorized users are permitted access to the system. It is to be noted that the password that is input will not be visible and will be represented by an asterisk (‘*’) for each character input. On a successful login the program directs the user to the main menu.

- The menu consists a list of the various functions that can be implemented by the program. It receives an integer input which navigates to the desired operation based on the appropriate option. Once the chosen operation is successfully completed, the user can choose to return to the main menu, or exit the program. Within each operation, that the user can choose, care has been taken to ensure that the user will have to input only valid input, at any given point of time.

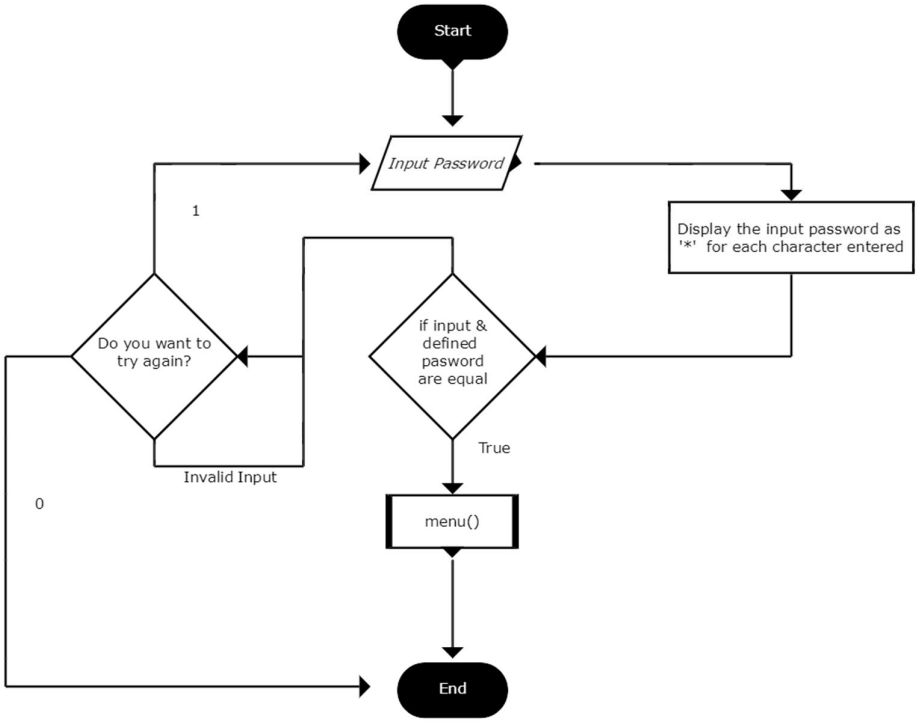
- At every given point, when the user is asked to provide input, a custom function for accepting that input is used for accepting the input. This is because the standard string functions, and standard IO functions offered by C are widely renowned to not have high security standards implemented. Thus, to fine tune the security of the system, depending upon the type of input, the corresponding custom function is invoked. This provides the added advantage of not customizing input constraints to the developer’s/system’s requirements.

3.4. Flowcharts

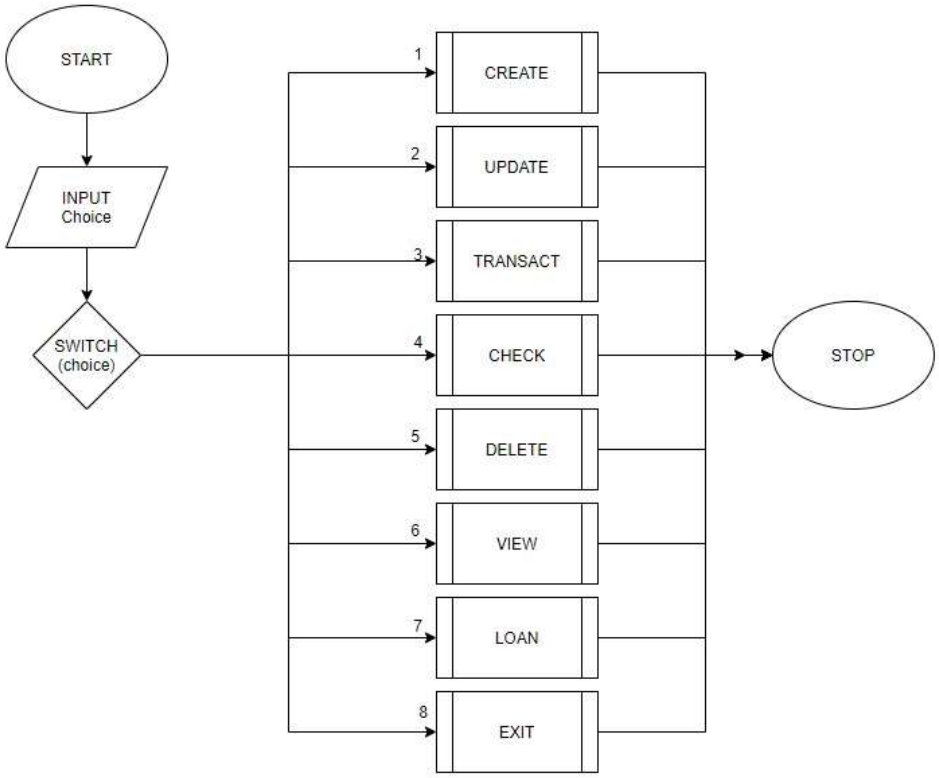
main ():



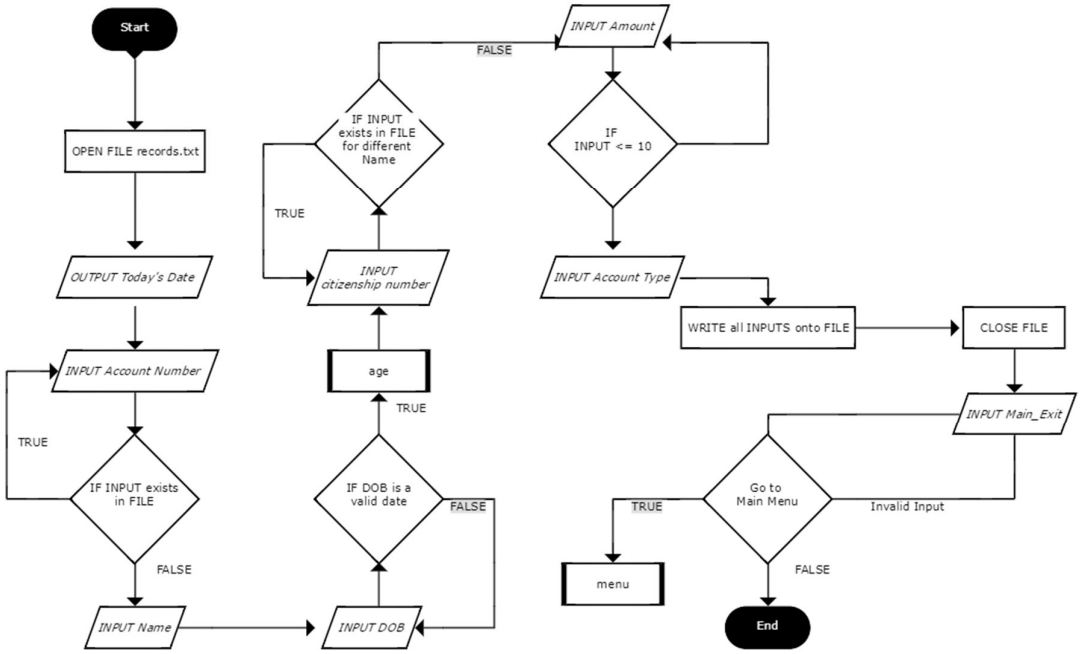
start ():



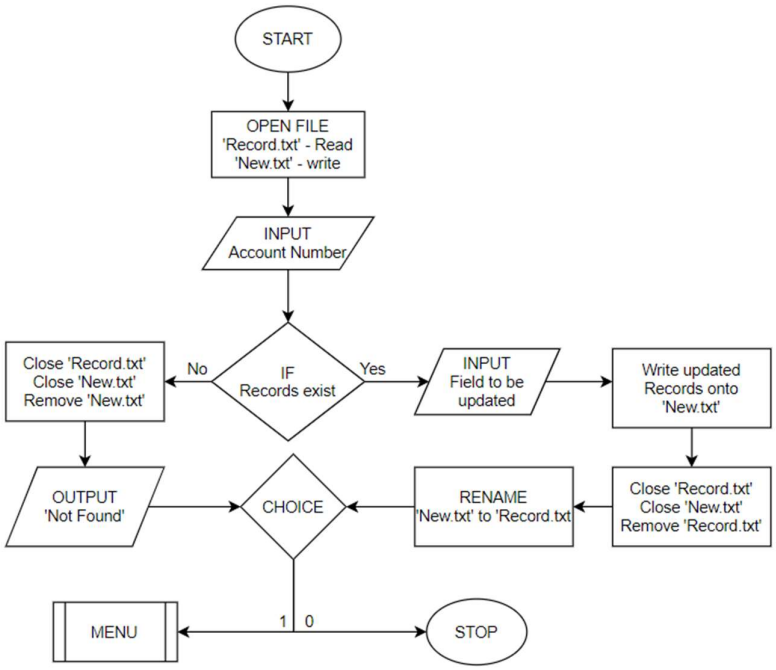
menu ():


START
INPUT
Choice
SWITCH
(choice)
1
CREATE
2
UPDATE
3
TRANSACT
4
CHECK
5
DELETE
6
VIEW
7
LOAN
8
EXIT
STOP

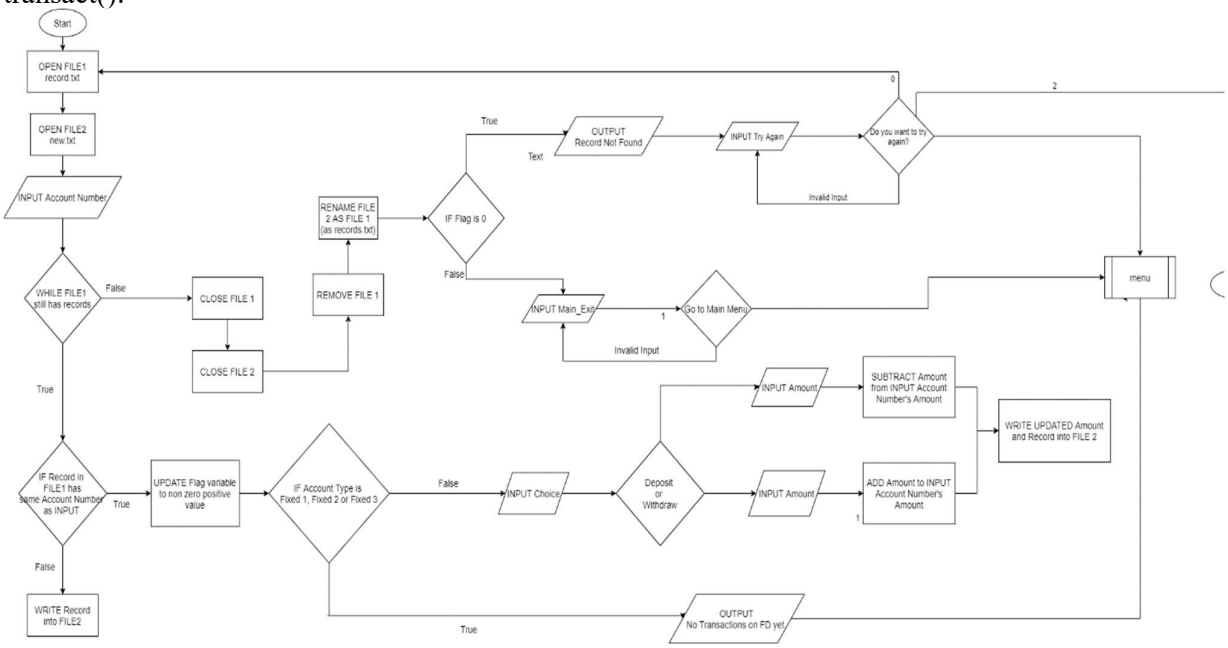
create ():


Start
OPEN FILE records.txt
OUTPUT Today's Date
INPUT Account Number
TRUE
IF INPUT exists in FILE
FALSE
INPUT Name
IF INPUT exists in FILE for different Name
TRUE
INPUT citizenship number
age
TRUE
IF DOB is a valid date
FALSE
INPUT DOB
INPUT Amount
FALSE
IF INPUT <= 10
INPUT Account Type
WRITE all INPUTS onto FILE
CLOSE FILE
INPUT Main_Exit
Go to Main Menu
TRUE
menu
Invalid Input
FALSE
End

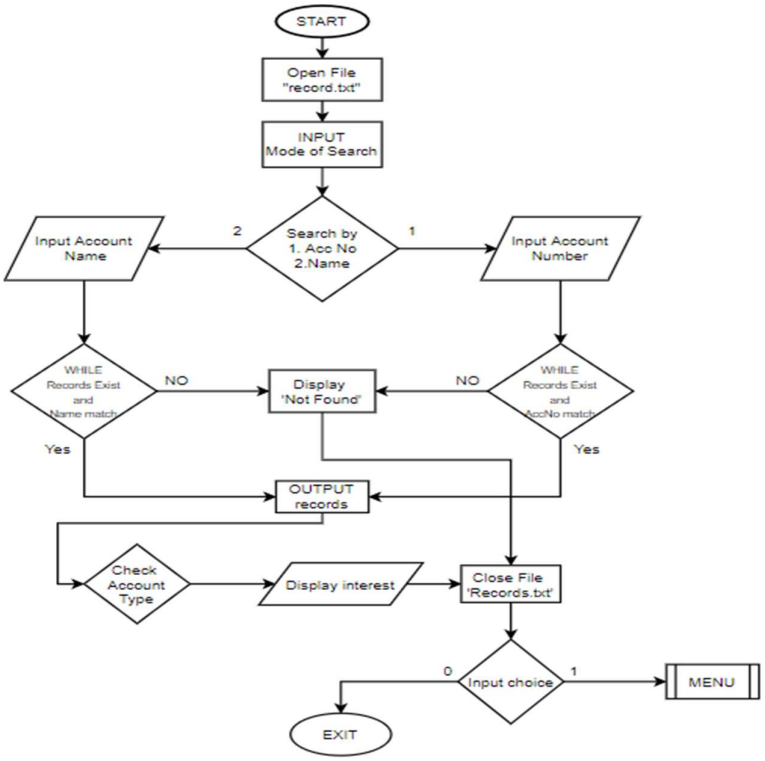
edit ():



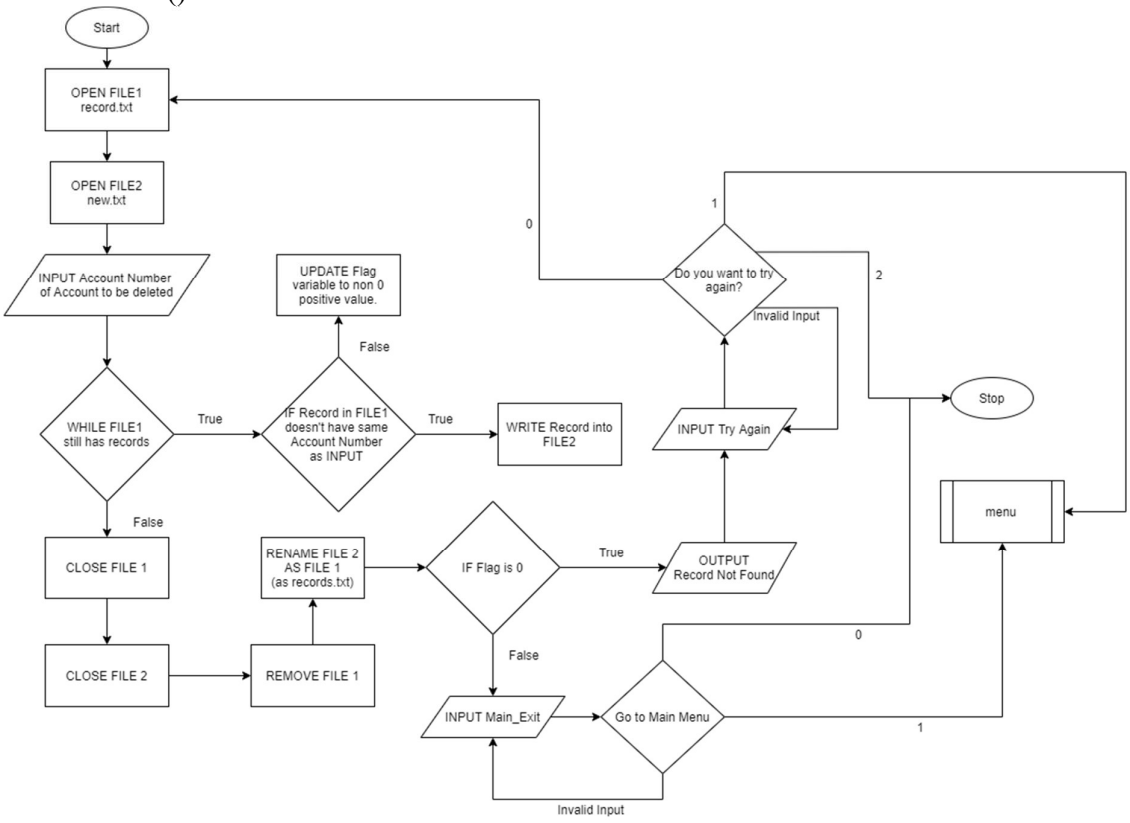
transact():



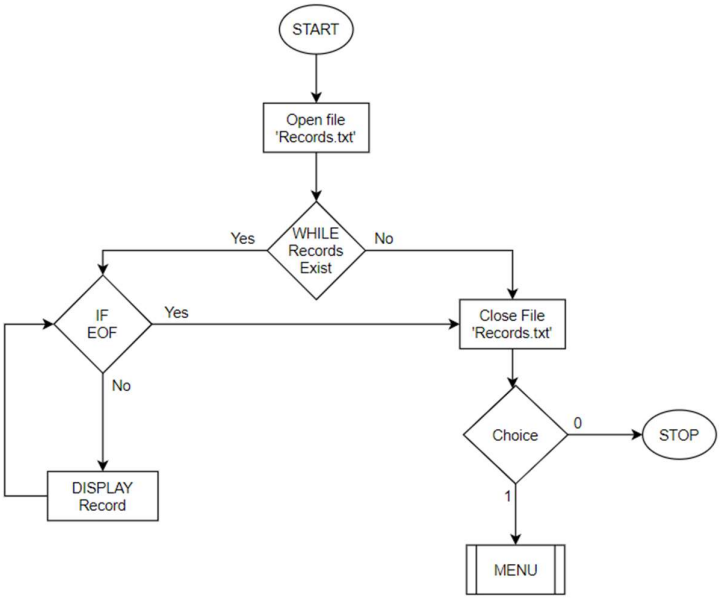
check ():



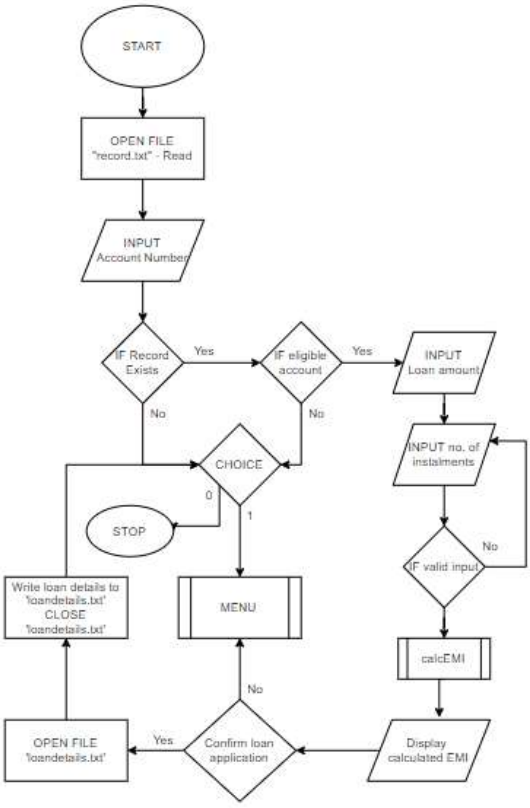
closeAccount():



view():


START
Open file 'Records.txt'
WHILE Records Exist
Yes
No
IF EOF
Yes
No
Close File 'Records.txt'
Choice
0
1
STOP
MENU
DISPLAY Record

loan():


START
OPEN FILE "record.txt" - Read
INPUT Account Number
IF Record Exists
Yes
No
IF eligible account
Yes
No
INPUT Loan amount
INPUT no. of instalments
IF valid input
No
calcEMI
Display calculated EMI
Confirm loan application
Yes
No
OPEN FILE 'loandetails.txt'
Write loan details to 'loandetails.txt' CLOSE 'loandetails.txt'
CHOICE
0
1
STOP
MENU

4. Security Features - Rules and Recommendations Incorporated

4.1. Rules

Preprocessor (PRE)

PRE30-C. Do not create a universal character name through concatenation.

Declarations and Initialization (DCL)

DCL30-C. Declare objects with appropriate storage durations
DCL31-C. Declare identifiers before using them
DCL33-C. Ensure that source and destination pointers in function arguments do not point to overlapping objects if they are restrict qualified
DCL35-C. Do not convert a function pointer to a function of a different type
DCL36-C. Do not declare an identifier with conflicting linkage classifications

Expressions (EXP)

EXP30-C. Do not depend on order of evaluation between sequence points
EXP31-C. Do not modify constant values
EXP33-C. Do not reference uninitialized variables
EXP34-C. Ensure a pointer is valid before dereferencing it
EXP35-C. Do not access or modify the result of a function call after a subsequent sequence point
EXP36-C. Do not cast between pointers to objects or types with differing alignments

Integers (INT)

INT33-C. Ensure that division and remainder operations do not result in divide-by-zero errors
INT33-C. Ensure that division errors
INT30-C. Ensure that unsigned integer operations do not wrap
INT35-C. Use correct integer precisions
INT36-C. Converting a pointer to integer or integer to pointer

Floating Point (FLP)

FLP30-C. Do not use floating-point variables as loop counters
FLP37-C. Do not use object representations to compare floating-point values

Array (ARR)

ARR30-C. Do not form or use out-of-bounds pointers or array subscripts
ARR32-C. Ensure size arguments for variable length arrays are in a valid range
ARR36-C. Do not subtract or compare two pointers that do not refer to the same array
ARR37-C. Do not add or subtract an integer to a pointer to a non-array object
ARR38-C. Guarantee that library functions do not form invalid pointers
ARR39-C. Do not add or subtract a scaled integer to a pointer

Characters and Strings (STR)

STR30-C. Do not attempt to modify string literals
STR31-C. Guarantee that storage for strings has sufficient space for character data and the null terminator

Memory Management (MEM)

MEM30-C. Do not access freed memory
MEM34-C. Only free memory allocated dynamically
MEM35-C. Allocate sufficient memory for an object

Input / Output (FIO)

FIO30-C. Exclude user input from format strings
FIO34-C. Distinguish between characters read from a file and EOF or WEOF
FIO42-C. Close files when they are no longer needed
FIO46-C. Do not access a closed file
FIO47-C. Use valid format strings

Error Handling (ERR)

ERR30-C. Set errno to zero before calling a library function known to set errno, and check errno only after the function returns a value indicating failure
ERR32-C. Do not rely on indeterminate values of errno
ERR33-C. Detect and handle standard library errors

Miscellaneous (MSC)

MSC40-C. Do not violate constraintse:
MSC41-C. Never hard code sensitive information

4.2. Recommendations

Preprocessor (PRE)

PRE31-C. Avoid side effects in arguments to unsafe macros
PRE32-C. Do not use preprocessor directives in invocations of function-like macros

Declarations and Initialization (DCL)

DCL00-A. Declare immutable values using const or enum
DCL01-A. Do not reuse variable names in sub-scopes
DCL02-A. Use visually distinct identifiers
DCL03-A. Place const as the rightmost declaration specifier
DCL04-A. Take care when declaring more than one variable per declaration
DCL05-A. Use typedefs to improve code readability
DCL06-A. Use meaningful symbolic constants to represent literal values in program logic
DCL07-A. Ensure every function has a function prototype
DCL09-A. Declare functions that return an errno with a return type of errno_t
DCL12-A. Create and use abstract data types
DCL37-C. Do not declare or define a reserved identifier

Expressions (EXP)

EXP00-A. Use parentheses for precedence of operation
EXP01-A. Do not take the sizeof a pointer to determine the size of a type
EXP02-A. The second operands of the logical AND and OR operation should not contain any side effects
EXP03-A. Do not assume the size of a structure is the sum of the size of its members
EXP05-A. Do not cast away a const qualification
EXP07-A. use caution with NULL and 0, especially concerning pointers
EXP09-A. Use sizeof to determine the size of a type or variable

Integers (INT)

INT00-C. Understand the data model used by your implementation(s)
INT01-C. Use rsize_t or size_t for all integer values representing the size of an object
INT02-C. Understand integer conversion rules
INT08-C. Verify that all integer values are in range
INT09-C. Ensure enumeration constants map to unique values
INT10-C. Do not assume a positive remainder when using the % operator

Array (ARR)

ARR00-C. Understand how arrays work
ARR01-C. Do not apply the sizeof operator to a pointer when taking the size of an array
ARR02-C. Explicitly specify array bounds, even if implicitly defined by an initializer

Characters and Strings (STR)

STR00-C. Represent characters using an appropriate type
STR01-C. Adopt and implement a consistent plan for managing strings
STR02-C. Sanitize data passed to complex subsystems
STR03-C. Do not inadvertently truncate a string
STR04-C. Use plain char for characters in the basic character set
STR09-C. Don't assume numeric values for expressions with type plain character
STR10-C. Do not concatenate different type of string literals
STR11-C. Do not specify the bound of a character array initialized with a string literal

Memory Management (MEM)

MEM03-C. Clear sensitive information stored in reusable resources
MEM04-C. Beware of zero-length allocations
MEM05-C. Avoid large stack allocations
MEM11-C. Do not assume infinite heap space
MEM12-C. Consider using a goto chain when leaving a function on error when using and releasing resources

Input / Output (FIO)

FIO01-C. Be careful using functions that use file names for identification
FIO03-C. Do not make assumptions about fopen() and file creation
FIO05-C. Identify files using multiple file attributes
FIO06-C. Create files with appropriate access permissions
FIO08-C. Take care when calling remove() on an open file
FIO21-C. Do not create temporary files in shared directories
FIO22-C.Close files before spawning processes
FIO23-C. Do not exit with unflushed data in stdout or stderr
FIO24-C. Do not open a file that is already open

5. Implementation

5.1. Executing The Program

In order to execute the program, all one has to do is run the executable file in any terminal/command prompt.

In case any changes are made to the source code, one can compile and create executables to test by running the command **make** in the terminal.
Note:
However, for Windows systems, or Linux based systems that do not have the make tool pre-installed, this installation must be first done, and the executable of the tool must be added to environment variables, and consequently, the PATH.
One can set the path after installation by simply executing the following command

set PATH=C:\Program Files (x86)\GnuWin32\bin;%PATH%
(default directory)

5.2. Source Code

5.2.1. Header Files

1) data.h

```
/*
    RECOMMENDATIONS/RULES
    PRE31-C. Avoid side effects in arguments to unsafe macros
    PRE32-C. Do not use preprocessor directives in invocations of function-
like macros
    PRE30-C. Do not create a universal character name through
concatenation.
*/

int i,j;
int main_exit;
#define FORMAT "\n%ld %s %d/%d/%d %d %s %s %ld %s %f %d/%d/%d\n"
#define FLUSH fflush(stdin);
#define PRINTFILE(x) (x).acc_no, (x).name, (x).dob.day, (x).dob.month,
(x).dob.year, (x).age, (x).address, (x).citizenship, (x).phone,
(x).acc_type, (x).amt, (x).deposit.day, (x).deposit.month, (x).deposit.year
#define SCANFILE(x)  &(x).acc_no, (x).name, &(x).dob.day, &(x).dob.month,
&(x).dob.year, &(x).age, (x).address, (x).citizenship, &(x).phone,
(x).acc_type, &(x).amt, &((x)).deposit.day, &(x).deposit.month,
&(x).deposit.year
#define password "root"

struct date{
    int month,day,year;
};

typedef struct account{

    char name[60];
    long acc_no;
    int age;
    char address[60];
    char citizenship[15];
    long phone;
    char acc_type[10];
    float amt;
    struct date dob;
    struct date deposit;
    struct date withdraw;
    struct date currloan;


}account;
account user;
```

2) majorfunction.h

```
void create(void);

void transact(void);

void closeAccount(void);



void view_list(void);

void edit(void);

void see(void);

void loan(void);
```

3) utilFunctions.h

```
void start();
void menu();

void fordelay(int j);
int findAge(int current_date, int current_month, int current_year, int
birth_date, int birth_month, int birth_year);
int passwordAuthentication();

float interest(float t, float amount, int rate);
float calcEMI(float principal, float ratepercent_per_annum, int
installments);
```

4) inputValidation.h

```
long getLong();
float getFloat();
int getInt();
long phoneNumber();
void removeSpaces(char *str);
int validDate(int day, int month, int year, int day1, int month1, int
year1);
int Check_Email_Addr(const char *address);
```

5.2.2. Source Files

The source file for each file has been embedded in the respective image.

1. **bank.c**

```c
#include "data.h"
#include "inputValidation.h"
#include "majorfunction.h"
#include "utilFunctions.h"


int main(void)
{
    start();
    return 0;
}
/*
RECOMMENDATIONS FOLLOWED THROUGHOUT.
PREPROCESSOR (PRE):
    CERT C:
        PRE30-C. Do not create a universal character name through
concatenation.
        PRE32-C. Do not use preprocessor directives in invocations of
function-like macros
    Secured Programming with C:
        PRE04-A. Do not reuse a standard header file name
PREPROCESSOR (PRE):
    CERT C:
        DCL30-C. Declare objects with appropriate storage durations
        DCL31-C. Declare identifiers before using them
        DCL37-C. Do not declare or define a reserved identifier
        DCL41-C. Do not declare variables inside a switch statement before
the first case label
    Secured Programming with C:

*/
```

2. majorfunction.c

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <string.h>
#include <conio.h>
#include <windows.h>
#include<ctype.h>
#include "data.h"
#include "inputValidation.h"
#include "majorfunction.h"
#include "utilFunctions.h"

/*
    RECOMMENDATIONS/RULES
    DCL00-A. Declare immutable values using const or enum
    DCL01-A. Do not reuse variable names in sub-scopes
    DCL02-A. Use visually distinct identifiers
    DCL03-A. Place const as the rightmost declaration specifier
    DCL04-A. Take care when declaring more than one variable per declaration
    DCL05-A. Use typedefs to improve code readability
    DCL06-A. Use meaningful symbolic constants to represent literal values in
program logic
    DCL07-A. Ensure every function has a function prototype
    DCL09-A. Declare functions that return an errno with a return type of
errno_t
    DCL12-A. Create and use abstract data types
    DCL37-C. Do not declare or define a reserved identifier

    PRE30-C. Do not create a universal character name through concatenation.
    DCL30-C. Declare objects with appropriate storage durations
    DCL31-C. Declare identifiers before using them
    DCL36-C. Do not declare an identifier with conflicting linkage
classifications
    EXP30-C. Do not depend on order of evaluation between sequence points
    EXP31-C. Do not modify constant values
    EXP33-C. Do not reference uninitialized variables
    EXP34-C. Ensure a pointer is valid before dereferencing it
*/
```

a) create();

```
void create(void){
    account check;
    int choice;

    /*
    RECOMMENDATIONS/RULES
        FIO01-C. Be careful using functions that use file names for identi-
fication
        FIO03-C. Do not make assumptions about fopen() and file creation
        FIO05-C. Identify files using multiple file attributes
        FIO06-C. Create files with appropriate access permissions
        FIO08-C. Take care when calling remove() on an open file
        FIO21-C. Do not create temporary files in shared directories
        FIO22-C. Close files before spawning processes
        FIO23-C. Do not exit with unflushed data in stdout or stderr
        FIO24-C. Do not open a file that is already open

        FIO30-C. Exclude user input from format strings
        FIO34-C. Distinguish between characters read from a file and EOF or
WEOF
        FIO42-C. Close files when they are no longer needed
        FIO46-C. Do not access a closed file
        FIO47-C. Use valid format strings
    */
    FILE *restrict const ptr = fopen("record.txt", "a+");

    FLUSH
    system("cls");
    printf("\t\t\t ADD RECORD  ");
    //Get Time.
    SYSTEMTIME t;
    GetLocalTime(&t);
    printf("\n\n\nToday's date (DD/MM/YYYY): %d/%d/%d", t.wDay, t.wMonth,
t.wYear);
    user.deposit.day = t.wDay;
    user.deposit.month = t.wMonth;
    user.deposit.year = t.wYear;


account_no:
    printf("\nPlease enter the Account Number:");
    FLUSH
    check.acc_no = getLong();
    while (fscanf(ptr, FORMAT, SCANFILE(user)) != EOF)
    {
        printf(FORMAT,PRINTFILE(user));
        if (check.acc_no == user.acc_no)
        {
            printf("Account no. already in use!");
            fseek(ptr, 0, SEEK_SET);
            goto account_no;
            break;
            fordelay(1000000000);
        }
    }
```

```
    user.acc_no = check.acc_no;

    printf("\nEnter the name:");
    scanf("%[^\n]s", user.name);
    removeSpaces(user.name);
    FLUSH

validate_dob:
    printf("\nEnter the date of birth (DD/MM/YYYY):");
    int c = 0;

    while ((scanf("%d/%d/%d",&user.dob.day,&user.dob.month,&user.dob.year) !=
3))
    {
        printf("\nThe above date of birth is invalid.\nEnter a valid date of
birth(mm/dd/yyyy):");
        do { c = getchar(); } while (c != '\n' && c != EOF);  /* flush input
buffer */
    }
    FLUSH

    if(
!(validDate(user.dob.day,user.dob.month,user.dob.year,user.deposit.day,user.d
eposit.month,user.deposit.year))){
        printf("Kindly enter a valid date");
        fordelay(1000000000);
        goto validate_dob;
    }
age:
    user.age = findAge(t.wDay, t.wMonth,
t.wYear,user.dob.day,user.dob.month,user.dob.year);
    FLUSH

address:
    printf("\nEnter address name:");
    FLUSH
    scanf("%[^\n]s", user.address);
    removeSpaces(user.address);


citizenship_validation:
    printf("\nEnter the citizenship number:");
    FLUSH
    scanf("%[^\n]s", user.citizenship);
    removeSpaces(user.citizenship);
    fseek (ptr , 0 , SEEK_SET );
    while (fscanf(ptr, FORMAT, SCANFILE(check))!=EOF)
    {
        printf(FORMAT,PRINTFILE(check));
```

```
            if (!strcmp(check.citizenship,user.citizenship))
            {
                if(strcmp(check.name,user.name) || (check.dob.day!=user.dob.day)
                    || (check.dob.month!=user.dob.month) ||
                    (check.dob.year!=user.dob.year))
                {
                    printf("Uh Oh! Try again with proper credentials.");
                    fordelay(1000000000);
                    fseek (ptr , 0 , SEEK_SET );
                    goto citizenship_validation;
                }

            }
        }
        FLUSH

    phone:
        printf("\nEnter the phone number: ");
        FLUSH
        user.phone = phoneNumber();
        if(user.phone == -1){
            goto phone;
            FLUSH
        }

    amount_to_deposit:

        printf("\nEnter the amount to deposit:$");
        FLUSH
        user.amt = getFloat();
        if(user.amt<10)
        {
            printf("You need to deposit a minimum of $10.");
            goto amount_to_deposit;
        }
        FLUSH

    account_type:
        printf("\nType of account: ");
        printf("\n\t-> Saving\n\t-> Current\n");
        printf("\t-> Fixed1(for 1 year)\n\t-> Fixed2(for 2 years)\n\t->
    Fixed3(for 3 years)");
        printf("\n\n\tEnter your choice:");
        scanf("%s", user.acc_type);
        removeSpaces(user.acc_type);
        FLUSH

        fprintf(ptr, FORMAT, PRINTFILE(user));

        fclose(ptr);
        printf("\nAccount created successfully!");
    add_invalid:
        printf("\n\n\n\t\tEnter 1 to go to the main menu and 0 to exit:");
```

```
        main_exit = getInt();
        if (main_exit == 1)
        {
            system("cls");
            return;
        }
        else if (main_exit == 0){
            system("cls");
            exit(0);

        }

        else
        {
            printf("\nInvalid!\a");
            system("cls");
            goto add_invalid;
        }
    }
```

b) transact();

```
void transact(void){

    int choice, test = 0;
    /*
    RECOMMENDATIONS/RULES
        FIO01-C. Be careful using functions that use file names for identi-
fication
        FIO03-C. Do not make assumptions about fopen() and file creation
        FIO05-C. Identify files using multiple file attributes
        FIO06-C. Create files with appropriate access permissions
        FIO08-C. Take care when calling remove() on an open file
        FIO21-C. Do not create temporary files in shared directories
        FIO22-C. Close files before spawning processes
        FIO23-C. Do not exit with unflushed data in stdout or stderr
        FIO24-C. Do not open a file that is already open

        FIO30-C. Exclude user input from format strings

        FIO34-C. Distinguish between characters read from a file and EOF or
WEOF
        FIO42-C. Close files when they are no longer needed

        FIO46-C. Do not access a closed file

        FIO47-C. Use valid format strings
    */
    FILE *restrict const old = fopen("record.txt", "r") ;
    FILE *restrict const newrec =  fopen("new.txt", "w");

    printf("\nEnter the account no. of the customer:");
    FLUSH
    long transactionAccount;
    transactionAccount = getLong();
    while (fscanf(old, FORMAT, SCANFILE(user)) != EOF)
    {

        if (user.acc_no == transactionAccount)
        {
            test = 1;
            if (strcmpi(user.acc_type, "fixed1") == 0 ||
strcmpi(user.acc_type, "fixed2") == 0 || strcmpi(user.acc_type, "fixed3")
== 0)
            {
                printf("\a\a\a\n\nYou cannot deposit or withdraw cash from
fixed accounts. Kindly wait until more transaction features are made avail-
able.");
                fordelay(1000000000);
                system("cls");
                return;
            }
        transact_account:
            printf("\n\nTransaction:\n\n1: Deposit \n2: Withdrawal \nAny
other number: Exit. \n\nEnter your choice :");
            choice = getInt();
            if (choice == 1)
            {
```

```
                    FLUSH
                    printf("\nEnter the amount you want to deposit: $");
                    float transactionAmount = getFloat();
                    user.amt +=  transactionAmount;
                    fprintf(newrec, FORMAT, PRINTFILE(user));
                    printf("\n\nDeposited successfully!");
                }
                else if(choice == 2)
                {
                    FLUSH
                    printf("\nEnter the amount you want to withdraw: $");
                     float  transactionAmount = getFloat();
                    if(user.amt-10< transactionAmount)
                    {
                        printf("\n\nTransaction declined. \nInsufficient Funds in
account.");
                        goto transact_account;
                    }
                    else
                        user.amt -=  transactionAmount;
                    fprintf(newrec, FORMAT, PRINTFILE(user));
                    printf("\n\nWithdrawn successfully!");
                }
            }
            else
            {
                fprintf(newrec, FORMAT, PRINTFILE(user));
            }
        }
        fclose(old);
        fclose(newrec);
        remove("record.txt");
        rename("new.txt", "record.txt");

        if (test != 1)
        {
            printf("\n\nRecord not found!!");
        transact_invalid:
            printf("\n\n\nEnter 0 to try again,1 to return to main menu and 2 to
exit:");
            main_exit = getInt();
            system("cls");
            if (main_exit == 0)
                transact();
            else if (main_exit == 1)
                return;
            else if (main_exit == 2)
                exit(0);
            else
            {
                printf("\nInvalid  Input.");
                goto transact_invalid;
            }
```

```
        else
        {
            printf("\nEnter 1 to go to the main menu and any other number to
    exit. ");
            main_exit = getInt();
            system("cls");
            if (main_exit == 1)
                return;
            else
                exit(0);
        }
    }
```

c) closeaccount();

```
void closeAccount(void)
{
    /*
    RECOMMENDATIONS/RULES
        FIO30-C. Exclude user input from format strings
        FIO34-C. Distinguish between characters read from a file and EOF or
WEOF
        FIO42-C. Close files when they are no longer needed
        FIO46-C. Do not access a closed file
        FIO47-C. Use valid format strings
    */
    FILE *restrict const old = fopen("record.txt", "r") ;
    FILE *restrict const newrec =  fopen("new.txt", "w");
    int test = 0;
    long rem;

    printf("\nEnter the account no. of the customer you want to delete:");
    FLUSH
    rem = getLong();

    // printf("%ld is going to be deleted. ",rem);
    while (fscanf(old, FORMAT, SCANFILE(user)) != EOF)
    {
        if (user.acc_no != rem){
            // printf(FORMAT, PRINTFILE(add));
            fprintf(newrec, FORMAT, PRINTFILE(user));
        }

        else
        {
            test++;
            printf("\nRecord located.\n");
        }
    }

    fclose(old);
    fclose(newrec);

    if (test == 0)
    {
        printf("\nRecord not found.\a\a\a");

    delete_invalid:
        printf("\nEnter 0 to try deleting another record, 1 to return to
main menu, 2 to exit. ");
        main_exit = getInt();

        if (main_exit == 1)
            return;
        else if (main_exit == 2)
            exit(0);
        else if (main_exit == 0)
            closeAccount();
        else
```

```
            {
                printf("\nInvalid! Kindly input a valid option.\a");
                goto delete_invalid;
            }
        }

        else
        {
            int choice;
            printf("\nYou need to login before you can delete.");

            int loginStatus = passwordAuthentication();

            if(loginStatus){

                remove("record.txt");
                rename("new.txt", "record.txt");
            }
            else{
                remove("new.txt");
                printf("\nSorry, you're not authorized to delete records at this
moment.");
            }

            printf("\nEnter 1 to go to the main menu and any other number to
exit. ");
            main_exit = getInt();
            system("cls");
            if (main_exit == 1)
                return;
            else
                exit(0);
        }
}
```

d) view_list();

```
void view_list()
{
    /*
    RECOMMENDATIONS/RULES
    DCL05-A. Use typedefs to improve code readability
    */
    typedef int flag;
    //Display Records as a List
    /*
    RECOMMENDATION/RULES
    DCL00-A. Declare immutable values using const or enum
    DCL03-A: Place const as the rightmost declaration specifier
    DCL33-C. Ensure that source and destination pointers in function argu-
ments do not point to overlapping objects if they are restrict qualified
    FIO01-C. Be careful using functions that use file names for identifica-
tion
    FIO03-C. Do not make assumptions about fopen() and file creation
    FIO05-C. Identify files using multiple file attributes
    FIO06-C. Create files with appropriate access permissions
    FIO08-C. Take care when calling remove() on an open file
    FIO21-C. Do not create temporary files in shared directories
    FIO22-C. Close files before spawning processes
    FIO23-C. Do not exit with unflushed data in stdout or stderr
    FIO24-C. Do not open a file that is already open
    FIO30-C. Exclude user input from format strings
    FIO34-C. Distinguish between characters read from a file and EOF or
WEOF
    FIO42-C. Close files when they are no longer needed
    FIO46-C. Do not access a closed file
    FIO47-C. Use valid format strings

    */
    FILE *restrict const view = fopen("record.txt","r");
    /*
    RECOMMENDATIONS/RULES
    DCL09-A. Declare functions that return an errno with a return type of
errno_t
    ERR30-C. Set errno to zero before calling a library function known to
set errno, and check errno only after the function returns a value indicat-
ing failure
    ERR32-C. Do not rely on indeterminate values of errno
    ERR33-C. Detect and handle standard library errors
    */
    errno=0;
    int errnum;
    if(view == NULL){
        errnum = errno;
        fprintf(stderr, "ERROR CODE: %d\n", errno);
        perror("Error printed by perror");
        fprintf(stderr, "Error opening file: %s\n", strerror( errnum ));
        exit(0);
    }
```

```
    flag test=0; //Iterate Test variable to keep track of records
    system("cls");

printf("_____________________________________________________________________________
_____________________________________________");
    printf("\nACC. NO.\tNAME\t\t\tADDRESS\t\t\t\tPHONE\n");

printf("_____________________________________________________________________________
_____________________________________________");

    while(fscanf(view,FORMAT, SCANFILE(user))!=EOF)
       {

           printf("\n%ld\t\t
%s\t\t%25.30s\t\t\t%ld",user.acc_no,user.name,user.address,user.phone);
           test++;
       }

    fclose(view);
    //Incase there are no previous records
    if (test==0)
        {   system("cls");
            printf("\nNo Records Found.\n");}

    view_list_invalid:
     printf("\n\nEnter 1 to go to the main menu and 0 to exit. ");
        main_exit = getInt();
        system("cls");
        if (main_exit==1)
            return;
        else if(main_exit==0)
            exit(0);
        else
        {
            printf("\nInvalid!\a");
            goto view_list_invalid;
        }
}
```

e) edit();

```
void edit(void){
    /*
    RECOMMENDATIONS/RULES
    DCL05-A. Use typedefs to improve code readability
    */
    typedef int flag;
    typedef int option;
    //Update Values in pre-existing Records
    option choice;
    flag test=0;

    long acc_no, phone;
    char address[60];

    /*
    RECOMMENDATIONS/RULES
    DCL00-A. Declare immutable values using const or enum
    DCL03-A: Place const as the rightmost declaration specifier
    DCL33-C. Ensure that source and destination pointers in function argu-
ments do not point to overlapping objects if they are restrict qualified
    FIO30-C. Exclude user input from format strings
    FIO34-C. Distinguish between characters read from a file and EOF or
WEOF
    FIO42-C. Close files when they are no longer needed
    FIO46-C. Do not access a closed file
    FIO47-C. Use valid format strings
    */
    FILE *restrict const old=fopen("record.txt","r");
    /*
    RECOMMENDATIONS/RULES
    DCL09-A. Declare functions that return an errno with a return type of
errno_t
    ERR30-C. Set errno to zero before calling a library function known to
set errno, and check errno only after the function returns a value indicat-
ing failure
    ERR32-C. Do not rely on indeterminate values of errno
    ERR33-C. Detect and handle standard library errors
    */
    errno=0;
    int errnum;
    if(old == NULL){
        errnum = errno;
        fprintf(stderr, "ERROR CODE: %d\n", errno);
        perror("Error printed by perror");
        fprintf(stderr, "Error opening file: %s\n", strerror( errnum ));
        exit(0);
    }
    FILE *restrict const newrec=fopen("new.txt","w");

    printf("\nEnter the account no. of the customer whose info you want to
change:");
    FLUSH
```

```
    acc_no = getLong();
    while(fscanf(old,FORMAT, SCANFILE(user))!=EOF)
    {
        if (user.acc_no==acc_no)
        {
            test=1;
            printf("\nWhich information do you want to
change?\n1.Address\n2.Phone\n\nEnter your choice(1 for address and 2 for
phone):");
            FLUSH
            choice = getInt();
            system("cls");
            if(choice==1)
                //Updates Address Value
                {printf("Enter the new address:");
                FLUSH
                scanf("%s",address);
                fprintf(ptr, FORMAT, PRINTFILE(user));
                system("cls");
                printf("Changes saved!");
                }
            else if(choice==2)
                {
                    //Updates Phone No. Value
                    printf("Enter the new phone number:");
                    updatePhone:
                        phone = phoneNumber();
                        if(phone == -1){
                            goto updatePhone;
                            FLUSH
                        }
                fprintf(ptr, FORMAT, PRINTFILE(user));
                system("cls");
                printf("Changes saved!");
                }

        }
        else
             fprintf(ptr, FORMAT, PRINTFILE(user));
    }
    fclose(old);
    fclose(newrec);
    remove("record.txt");
    rename("new.txt","record.txt");
```

```
        if(test!=1)
        {
            //In case of invalid search
            system("cls");
                printf("\nRecord not found!!\a\a\a");
                edit_invalid:
                  printf("\nEnter 0 to try again,1 to return to main menu and 2
    to exit:");
                  scanf("%d",&main_exit);
                  system("cls");
                     if (main_exit==1)
                         //Handles return to main menu
                         return;
                    else if (main_exit==2)
                         //Handles exit operation
                         exit(0);
                    else if(main_exit==0)
                         //Handles retry condition
                         edit();
                    else
                         {printf("\nInvalid!\a");
                         goto edit_invalid;}
            }
        else
            {printf("\n\n\nEnter 1 to go to the main menu and 0 to exit:");
            scanf("%d",&main_exit);
            system("cls");
            if (main_exit==1)
                return;
            else
                exit(0);
            }
    }
```

f) see();

```
void see(void){

    /*
    RECOMMENDATIONS/RULES
    DCL05-A. Use typedefs to improve code readability
    */
    typedef int flag;
    typedef int option;
    //Function to view details of a particular record
    /*
    RECOMMENDATIONS/RULES
    DCL00-A. Declare immutable values using const or enum
    DCL03-A: Place const as the rightmost declaration specifier
    DCL33-C. Ensure that source and destination pointers in function argu-
ments do not point to overlapping objects if they are restrict qualified
    FIO30-C. Exclude user input from format strings
    FIO34-C. Distinguish between characters read from a file and EOF or
WEOF
    FIO42-C. Close files when they are no longer needed
    FIO46-C. Do not access a closed file
    FIO47-C. Use valid format strings
    */
    FILE *restrict const ptr=fopen("record.txt","r");
    account check;
    int errnum;
    /*
    RECOMMENDATIONS/RULES
    DCL09-A. Declare functions that return an errno with a return type of
errno_t
    ERR30-C. Set errno to zero before calling a library function known to
set errno, and check errno only after the function returns a value indicat-
ing failure
    ERR32-C. Do not rely on indeterminate values of errno

    ERR33-C. Detect and handle standard library errors
    */
    errno = 0;
    if(ptr == NULL){
        errnum = errno;
        fprintf(stderr, "ERROR CODE: %d\n", errno);
        perror("Error printed by perror");
        fprintf(stderr, "Error opening file: %s\n", strerror( errnum ));
        exit(0);
    }
    flag test=0;
    int rate;
    option choice;
    float time;
    float intrst;
    /*
    RECOMMENDATIONS/RULES
    DCL06-A. Use meaningful symbolic constants to represent literal values
in program logic
*/
```

```
*/
    enum {RATE_FIXED1=9, RATE_FIXED2=11,RATE_FIXED3=13,RATE_SAVING=8};

    //Searches for record either by account number or name
    FLUSH
    printf("\nEnter the account number:");
    FLUSH
    check.acc_no = getLong();

    while (fscanf(ptr,FORMAT, SCANFILE(user))!=EOF)
    {
        if(user.acc_no==check.acc_no)
        {       system("cls");
                test=1;
                //Each of the following conditions are used to determine
the interest by a certain date based on the type of account
         printf("\nAccount NO.:%ld\nName:%s \nDOB:%d/%d/%d \nAge:%d
\nAddress:%s \nCitizenship No:%s \nPhone number:%ld \nType Of Account:%s
\nAmount deposited: $%.2f \nDate Of Deposit:%d/%d/%d\n\n",PRINTFILE(user));
                //Fixed Accounts
         if(strcmpi(user.acc_type,"fixed1")==0)
                {
                        time=1.0;
                        rate=RATE_FIXED1;
                        intrst=interest(time,user.amt,rate);
                        printf("\n\nYou will get $%.2f as interest on
%d/%d/%d",intrst,user.deposit.month,user.deposit.day,user.deposit.year+1);
                }
          else if(strcmpi(user.acc_type,"fixed2")==0)
                {
                        time=2.0;
                        rate=RATE_FIXED2;
                        intrst=interest(time,user.amt,rate);
                        printf("\n\nYou will get $.%.2f as interest on
%d/%d/%d",intrst,user.deposit.month,user.deposit.day,user.deposit.year+2);

                }
          else if(strcmpi(user.acc_type,"fixed3")==0)
                {
                        time=3.0;
                        rate=RATE_FIXED3;
                        intrst=interest(time,user.amt,rate);
                        printf("\n\nYou will get $.%.2f as interest on
%d/%d/%d",intrst,user.deposit.month,user.deposit.day,user.deposit.year+3);
                }
                //Savings Account
          else if(strcmpi(user.acc_type,"saving")==0)
                {
                        time=(1.0/12.0);
                        rate=RATE_SAVING;
                        intrst=interest(time,user.amt,rate);
```

```
            printf("\n\nYou will get $.%.2f as interest on %d of every
                    month",intrst,user.deposit.day);

                         }
                    //Current Account
                     else if(strcmpi(user.acc_type,"current")==0)
                        {

                            printf("\n\nYou will get no interest\a\a");

                         }

                }
}

    //Incase the record wasnt found
    fclose(ptr);
     if(test!=1)
        {   system("cls");
            printf("\nRecord not found!!\a\a\a");
            see_invalid:
              printf("\nEnter 0 to try again,1 to return to main menu and 2
to exit:");
              scanf("%d",&main_exit);
              system("cls");
                 if (main_exit==1)
                    return;
                else if (main_exit==2)
                    exit(0);
                else if(main_exit==0)
                    see();
                else
                    {
                        system("cls");
                        printf("\nInvalid!\a");
                        goto see_invalid;}
        }
    else
        {printf("\nEnter 1 to go to the main menu and 0 to exit:");
        scanf("%d",&main_exit);}
        if (main_exit==1)
        {
            system("cls");
            return;
        }

        else
           {

             system("cls");
            exit(0);
            }


}
```

3. loan.c

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <string.h>
#include <conio.h>
#include<errno.h>
#include <windows.h>
#include<ctype.h>
#include "data.h"
#include "utilFunctions.h"
#include "inputValidation.h"
void loan(void)
{
    /*
    RECOMMENDATIONS/RULES
    DCL00-A. Declare immutable values using const or enum
    DCL01-A. Do not reuse variable names in sub-scopes
    DCL02-A. Use visually distinct identifiers
    DCL03-A. Place const as the rightmost declaration specifier
    DCL04-A. Take care when declaring more than one variable per declara-
tion
    DCL05-A. Use typedefs to improve code readability
    DCL06-A. Use meaningful symbolic constants to represent literal values
in program logic
    DCL07-A. Ensure every function has a function prototype
    DCL09-A. Declare functions that return an errno with a return type of
errno_t
    DCL12-A. Create and use abstract data types
    DCL37-C. Do not declare or define a reserved identifier

    PRE30-C. Do not create a universal character name through concatena-
tion.
    DCL30-C. Declare objects with appropriate storage durations
    DCL31-C. Declare identifiers before using them
    DCL36-C. Do not declare an identifier with conflicting linkage classi-
fications
    EXP30-C. Do not depend on order of evaluation between sequence points
    EXP31-C. Do not modify constant values
    EXP33-C. Do not reference uninitialized variables
    EXP34-C. Ensure a pointer is valid before dereferencing it
    STR30-C. Do not attempt to modify string literals
    STR31-C. Guarantee that storage for strings has sufficient space for
character data and the null terminator
    MEM30-C. Do not access freed memory
    MEM34-C. Only free memory allocated dynamically
    MEM35-C. Allocate sufficient memory for an object

    */
    typedef int flag;
    typedef int option;
    flag test=0;
    option choice;
    int rate, criteria = 0, no_of_installments = 0;
```

```
float time, EMI, amt, min_amt;
    char ch, violations[100] = "";
    long acc_no;

    system("cls");

    printf("\nWelcome to the loan application page");
    printf("\nEligibility Criteria:\n\nIndividuals who can take a Personal
Loan:\n\n\tSalaried Employees\n\tSalaried doctors\n\tEmployees of public
and private limited companies\n\tGovernment sector employees including
Public Sector Undertakings, Central and Local bodies\n\tMinimum age of 21
years\n\tMaximum age of 60 years at the time of maturity of the Personal
Loan\n\tMinimum net monthly income Rs. 15,000");
    fordelay(1000000000);
    fordelay(1000000000);
    printf("\n\n\n\n\tPlease enter account number: ");
    FLUSH
    acc_no = getLong();

     /*
    RECOMMENDATIONS/RULES
    DCL05-A. Use typedefs to improve code readability
    */



    //Display Records as a List
    /*
    RECOMMENDATION/RULES
    DCL00-A. Declare immutable values using const or enum
    DCL03-A: Place const as the rightmost declaration specifier
    DCL33-C. Ensure that source and destination pointers in function
arguments do not point to overlapping objects if they are restrict
qualified
    */

    FILE *restrict const ptr = fopen("record.txt","r");

    /*
    RECOMMENDATIONS/RULES
    DCL09-A. Declare functions that return an errno with a return type of
errno_t
    ERR30-C. Set errno to zero before calling a library function known to
set errno, and check errno only after the function returns a value
indicating failure
    ERR32-C. Do not rely on indeterminate values of errno

    ERR33-C. Detect and handle standard library errors
    */
    errno = 0;
    int errnum;
    if(ptr == NULL){
        errnum = errno;
        fprintf(stderr, "ERROR CODE: %d\n", errno);
        perror("Error printed by perror");
        fprintf(stderr, "Error opening file: %s\n", strerror( errnum ));
        exit(0);
```

```
    }
    SYSTEMTIME t;
        GetLocalTime(&t);

    /*
    RECOMMENDATIONS/RULES
    DCL06-A. Use meaningful symbolic constants to represent literal values in
program logic
    */

    enum {RATE_FIXED1=9,
RATE_FIXED2=11,RATE_FIXED3=13,RATE_SAVING=8,RATE_CURRENT=8};

    while (fscanf(ptr,FORMAT, SCANFILE(user))!=EOF)
    {
        if(user.acc_no==acc_no)
        {

            system("cls");
            test=1;
            //Each of the following conditions are used to determine the
interest by a certain date based on the type of account
            //Fixed Accounts
            if(strcmpi(user.acc_type,"fixed1")==0)
                {

                rate=RATE_FIXED1;

                min_amt = (float)1000;
                }
            else if(strcmpi(user.acc_type,"fixed2")==0)
                {

                rate=RATE_FIXED2;

                min_amt = (float)1000;

                }
            else if(strcmpi(user.acc_type,"fixed3")==0)
            {

                rate=RATE_FIXED3;

                min_amt = (float)1000;

            }
            //Savings Account
            else if(strcmpi(user.acc_type,"saving")==0)
            {

                rate=RATE_SAVING;

                min_amt = (float)1000;
```

```
            }
            //Current Account
            else if(strcmpi(user.acc_type,"current")==0)
            {
                rate = RATE_CURRENT;

                min_amt = (float)1000;
            }
            strcpy_s(violations,"");
            if (user.age<21 || user.age>60){
                //Failed age criteria
                criteria += 1;
                strncat(violations,"\tFailed Age Criteria",20);
            }


            if (user.amt < min_amt){
                //failed minimum amount criteria
                criteria += 1;
                strncat(violations,"\n\tFailed Min Amount Criteria",50);
            }

            if (criteria != 0){
                printf("\nAccount inelligible for loan application.\nNumber
of violations: %d\n\n",criteria);
                printf("\n%s\n",violations);
                printf("\n\n\n\t\tEnter 1 to go to the main menu and 0 to
exit:");
                scanf("%d", &main_exit);
                system("cls");
                if (main_exit == 1)
                    return;
                else if (main_exit == 0)
                    exit(0);
                else
                {
                    printf("\nInvalid!\a");
                    goto add_invalid;
                }
            }
            else
            {
                user.currloan.day = t.wDay;
                user.currloan.month = t.wMonth;
                user.currloan.year = t.wYear;

                printf("\nPlease Enter Loan Amount: ");
                FLUSH
                amt = getFloat();

                //Entering number of installments
                inputinstallments:
```

```
                printf("\nPlease enter the number of installments (12, 24 or
48 months): ");
                no_of_installments = getInt();

                if (no_of_installments != 12 && no_of_installments!=24 &&
no_of_installments!=48){
                    goto inputinstallments;
                }

                printf("\nType of Account: %s",user.acc_type);
                //Calculating EMI based on given criteria
                EMI = calcEMI(amt, rate, no_of_installments);


                printf("\n\nCalculated EMI: %f",EMI);
                fordelay(1000000000);

                //Confirmation step
                printf("\nApply for loan? (y/n): ");
                scanf("%s",&ch);
                if (ch == 'n'){
                    //Cancelling loan application process
                    printf("\nOperation canceled by user.\nreturning...");
                    fordelay(1000000000);
                    return;
                }
                else if(ch == 'y'){
                    //Processing loan
                    //involves writing loan details onto a file based on the
given criteria
                    printf("\nProcessing...");
                    /*
                    RECOMMENDATIONS/RULES
                    FIO30-C. Exclude user input from format strings
                    FIO34-C. Distinguish between characters read from a file
and EOF or WEOF
                    FIO42-C. Close files when they are no longer needed
                    FIO46-C. Do not access a closed file
                    FIO47-C. Use valid format strings
                    */
                    FILE *loanptr = fopen("loandetails.txt","a");
                    fprintf(loanptr,"%d %s %d/%d/%d %f %f %d\n",user.acc_no,
user.name, user.currloan.day, user.currloan.month, user.currloan.year, amt,
EMI, no_of_installments);
                    fclose(loanptr);
                    fordelay(1000000000);
                    printf("\nLoan application successfull\n");
                    add_invalid:
                    printf("\n\n\n\t\tEnter 1 to go to the main menu and 0 to
exit:");
```

```
                    scanf("%d", &main_exit);
                    system("cls");
                    if (main_exit == 1)
                        return;
                    else if (main_exit == 0)
                        exit(0);
                    else
                    {
                        printf("\nInvalid!\a");
                        goto add_invalid;
                    }
                }

            }


        }
    }
    if (test!=1){
        printf("Account not found");
        fclose(ptr);
        printf("\n\n\n\t\tEnter 1 to go to the main menu and 0 to exit:");
        scanf("%d", &main_exit);
        system("cls");
        if (main_exit == 1)
            return;
        else if (main_exit == 0)
            exit(0);
        else
        {
            printf("\nInvalid!\a");
            goto add_invalid;
        }
    }
}
```

4. util.c

```
#define ENTER 13
#define BKSP 8
#define TAB 9
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <string.h>
#include <conio.h>
#include <windows.h>
#include "data.h"
#include "majorfunction.h"
#include "utilFunctions.h"
#include "inputValidation.h"

/*
    RECOMMENDATIONS/RULES
    DCL00-A. Declare immutable values using const or enum
    DCL01-A. Do not reuse variable names in sub-scopes
    DCL02-A. Use visually distinct identifiers
    DCL03-A. Place const as the rightmost declaration specifier
    DCL04-A. Take care when declaring more than one variable per declaration
    DCL05-A. Use typedefs to improve code readability
    DCL06-A. Use meaningful symbolic constants to represent literal values in
program logic
    DCL07-A. Ensure every function has a function prototype
    DCL09-A. Declare functions that return an errno with a return type of
errno_t
    DCL12-A. Create and use abstract data types
    DCL37-C. Do not declare or define a reserved identifier

    PRE30-C. Do not create a universal character name through concatenation.
    DCL30-C. Declare objects with appropriate storage durations
    DCL31-C. Declare identifiers before using them
    DCL36-C. Do not declare an identifier with conflicting linkage
classifications
    EXP30-C. Do not depend on order of evaluation between sequence points
    EXP31-C. Do not modify constant values
    EXP33-C. Do not reference uninitialized variables
    EXP34-C. Ensure a pointer is valid before dereferencing it
    INT33-C. Ensure that division and remainder operations do not result in
divide-by-zero errors
    INT30-C. Ensure that unsigned integer operations do not wrap
    INT35-C. Use correct integer precisions
    INT36-C. Converting a pointer to integer or integer to pointer
    FLP30-C. Do not use floating-point variables as loop counters
    FLP37-C. Do not use object representations to compare floating-point
values

*/
```

```
    //      PRE31-C. Avoid side effects in arguments to unsafe macros
    // Secured Programming with C:
    //      PRE00-C. Prefer inline or static functions to function-like macros

float interest(float t, float amount, int rate)
{
    return ((rate * t * amount) / 100.0);
}

void fordelay(int j)
{
    int i, k;
    for (i = 0; i < j; i++)
        k = i;
}



int findAge(int current_date, int current_month, int current_year, int
birth_date, int birth_month, int birth_year)
{
    // days of every month
    int month[] = {31, 28, 31, 30, 31, 30, 31,
                   31, 30, 31, 30, 31};

    // if birth date is greater, then current birth
    // month then do not count this month and add 30
    // to the date so as to subtract the date and
    // get the remaining days
    if (birth_date > current_date)
    {
        current_date = current_date + month[birth_month - 1];
        current_month = current_month - 1;
    }

    // if birth month exceeds current month, then do
    // not count this year and add 12 to the month so
    // that we can subtract and find out the difference
    if (birth_month > current_month)
    {
        current_year = current_year - 1;
        current_month = current_month + 12;
    }

    // calculate date, month, year
    int calculated_date = current_date - birth_date;
    int calculated_month = current_month - birth_month;
    int calculated_year = current_year - birth_year;

    // return the present age
    return calculated_year;
}
```

```
void menu(void)
{
    int choice;
    menu:
    system("cls");
    //system("color 99");
    printf("\n\n\t\t\tCUSTOMER ACCOUNT BANKING MANAGEMENT SYSTEM");
    printf("\n\n\n\t\t\t WELCOME TO THE MAIN MENU ");
    printf("\n\n\t\t1.Create new account\n\t\t2.Update information of
existing account\n\t\t3.For transactions\n\t\t4.Check the details of existing
account\n\t\t5.Removing existing account\n\t\t6.View customer's
list\n\t\t7.Loan Application\n\t\t8.Exit\n\n\n\n\n\t\t Enter your choice:");
    scanf("%d", &choice);

    system("cls");
    switch (choice)
    {
    case 1:
        create();
        goto menu;
        break;
    case 2:
        edit();
        goto menu;
        break;
    case 3:
        transact();
        goto menu;
        break;
    case 4:
        see();
        goto menu;
        break;
    case 5:
        closeAccount();
        goto menu;
        break;
    case 6:
        view_list();
        goto menu;
        break;
    case 7:
        loan();
        goto menu;
        break;
    case 8:
        break;
    }
}
```

```
int passwordAuthentication()
{
    /*
RULES/RECOMMENDATIONS
    STR00-C. Represent characters using an appropriate type
    STR01-C. Adopt and implement a consistent plan for managing strings
    STR02-C. Sanitize data passed to complex subsystems
    STR03-C. Do not inadvertently truncate a string
    STR04-C. Use plain char for characters in the basic character set
    STR09-C. Don't assume numeric values for expressions with type plain
character
    STR10-C. Do not concatenate different type of string literals
    STR11-C. Do not specify the bound of a character array initialized with a
string literal
    */
    char pwd[10];int i = 0;
    int p = 0;
    char ch;
    printf("Enter your password. Hit ENTER to confirm.\n");
    printf("Password:");

    while (1)
    {
        ch = getch(); //get key
        if (ch == ENTER || ch == TAB)
        {
            pwd[p] = '\0';
            break;
        }
        else if (ch == BKSP)
        {
            if (p > 0)
            {
                p--;
                printf("\b \b"); //for backspace
            }
        }
        else{
            pwd[p] = ch;
            p += 1;
            printf("* \b"); //to replace password character with *
        }
    }
    if (strcmp(pwd, password) == 0)
    {
        // printf("\n\nPassword Match!\nLOADING");
        return 1;
    }
    else
    {
        printf("\n\nPassword Incorrect.");
        return 0;
    }
```

```
    }
}


// EXP30-C. Do not depend on order of evaluation between sequence points
void start()
{
login_attempt:
    if(passwordAuthentication()){
        printf("\n\nPassword Match!\nLOADING");
        for (i = 0; i <= 6; i++)
        {
            fordelay(100000000);
            printf(".");
        }
        system("cls");
        printf("Login Successful!");
        menu();
        return;
    }

    else
    {
        printf("\n\nWrong password!\a\a\a");
    }

login_try:
        printf("\nEnter 1 to try again and 0 to exit. \n");
        main_exit = getInt();
        if (main_exit == 1)
        {
            system("cls");
            goto login_attempt;
        }

        else if (main_exit == 0)
        {
            system("cls");
            exit(0);
        }
        else
        {
            printf("\nInvalid!");
            fordelay(1000000000);
            system("cls");
            goto login_try;
        }
}
```

```
float calcEMI(float principal, float ratepercent_per_annum, int installments)
{
    /*
    RULES/RECOMMENDATIONS
    INT33-C. Ensure that division and remainder operations do not result in
divide-by-zero errors
    INT30-C. Ensure that unsigned integer operations do not wrap
    INT35-C. Use correct integer precisions
    INT36-C. Converting a pointer to integer or integer to pointer
    FLP30-C. Do not use floating-point variables as loop counters
    FLP37-C. Do not use object representations to compare floating-point
values

    */
    float emi;
    float rate;
    float val;
    printf("Rate of interest per annum for your account:
%f",ratepercent_per_annum);
    rate = ratepercent_per_annum/1200;
    val = pow((1+rate),installments);
    emi = principal*rate*val/(val-1);
    return emi;
}
```

5. inputValidation.c

```
#include "inputValidation.h"
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <string.h>
#include <conio.h>
#include <windows.h>

/*
RULES/RECOMMENDATIONS
PRE30-C. Do not create a universal character name through concatenation.
PRE31-C. Avoid side effects in arguments to unsafe macros
PRE32-C. Do not use preprocessor directives in invocations of function-like
macros
DCL30-C. Declare objects with appropriate storage durations
DCL31-C. Declare identifiers before using them
DCL33-C. Ensure that source and destination pointers in function arguments
do not point to overlapping objects if they are restrict qualified
DCL35-C. Do not convert a function pointer to a function of a different
type
DCL36-C. Do not declare an identifier with conflicting linkage
classifications
INT00-C. Understand the data model used by your implementation(s)
INT01-C. Use rsize_t or size_t for all integer values representing the size
of an object
INT02-C. Understand integer conversion rules
INT08-C. Verify that all integer values are in range
INT09-C. Ensure enumeration constants map to unique values
INT33-C. Ensure that division and remainder operations do not result in
divide-by-zero errors
INT30-C. Ensure that unsigned integer operations do not wrap
INT35-C. Use correct integer precisions
INT36-C. Converting a pointer to integer or integer to pointer
MSC40-C. Do not violate constraintse:
MSC41-C. Never hard code sensitive information

*/


/*
RULES/RECOMMENDATIONS
FLP30-C. Do not use floating-point variables as loop counters
FLP37-C. Do not use object representations to compare floating-point values
STR00-C. Represent characters using an appropriate type
STR01-C. Adopt and implement a consistent plan for managing strings
STR02-C. Sanitize data passed to complex subsystems
STR03-C. Do not inadvertently truncate a string
STR04-C. Use plain char for characters in the basic character set
STR09-C. Don't assume numeric values for expressions with type plain
character
STR10-C. Do not concatenate different type of string literals
STR11-C. Do not specify the bound of a character array initialized with a
string literal
STR30-C. Do not attempt to modify string literals
STR31-C. Guarantee that storage for strings has sufficient space for
character data and the null terminator
*/
```

```c
int getInt(){
    int num = 0, ch;
    do{
        ch = getch();
        if (ch == 127 || ch == 8){
            // printf("\b");
            printf("\b \b");
            num=num/10;
            continue;
        }
        else if(ch>=48&&ch<=57){
            printf("%c",ch);
            num=num*10+(ch-48);
        }
        if(ch == 13){
            break;
        }
    }while(1);
    return num;
}
long getLong(){
    char *p, s[100];
    long n;

    while (fgets(s, sizeof(s), stdin)) {
        n = strtol(s, &p, 10);
        if (p == s || *p != '\n') {
            printf("\nPlease enter a valid number: ");
        }
        else
            break;
    }
    return n;
}

float getFloat(){
    char buffer[100];
    double value;
    char *endptr;

    getFloatValue:
    if (fgets(buffer, sizeof(buffer), stdin) == NULL){
        fflush(stdin);
        goto getFloatValue;
    }

    value = strtod(buffer, &endptr);
    if ((*endptr == '\0') || (isspace(*endptr) != 0)){
        return value;
    }
    else
        goto getFloatValue;
}
```

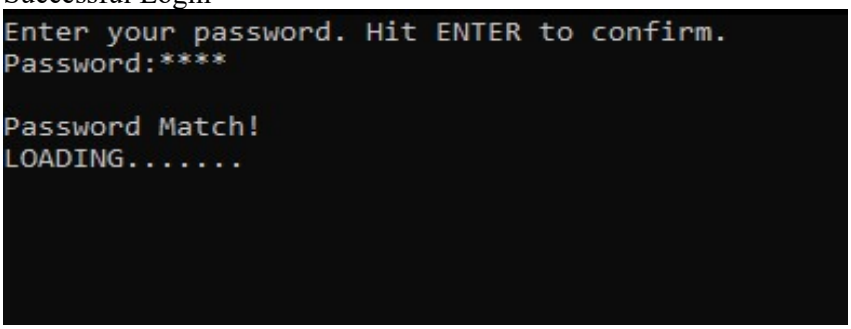
```
/*RULES/RECOMMENDATIONS
FLP30-C. Do not use floating-point variables as loop counters
FLP37-C. Do not use object representations to compare floating-point values
STR00-C. Represent characters using an appropriate type
STR01-C. Adopt and implement a consistent plan for managing strings
STR02-C. Sanitize data passed to complex subsystems
STR03-C. Do not inadvertently truncate a string
STR04-C. Use plain char for characters in the basic character set
STR09-C. Don't assume numeric values for expressions with type plain
character
STR10-C. Do not concatenate different type of string literals
STR11-C. Do not specify the bound of a character array initialized with a
string literal
STR30-C. Do not attempt to modify string literals
STR31-C. Guarantee that storage for strings has sufficient space for
character data and the null terminator


*/
long phoneNumber()
{
        char s[11];
        int cnt=0;
        while(cnt<10) //atmost 8 digits
        {
                s[cnt]=getchar();
                if(!isdigit(s[cnt]))//break if non digit entered
                {
                        s[cnt]='\0';//end string with a null.
                        break;
                }
                cnt++;
        }
    if(cnt!=10){
        printf("Enter a valid phone number. \n");
        return -1;
    }
        s[cnt]='\0';
        return atoi(s);
}

void removeSpaces(char *str)
{
    int count = 0;

    for (int i = 0; str[i]; i++)
        if (str[i] != ' ')
            str[count++] = str[i];
    str[count] = '\0';
}

/*
    EXP00-A. Use parentheses for precedence of operation

    EXP33-C. Do not reference uninitialized variables.
    (All possibilities have been thoroughly considered.)
*/
```

```
int validDate(int dd, int mm, int yy, int day, int month, int year)
{

    if (yy < 1900)
        return 0;
    if ((yy > year) || ((yy == year) && (mm > month)) || ((yy == year) &&
(mm == month) && (dd > day)))
        return 0;

    if (mm > 12)
        return 0;

    if (mm == 2)
    {
        if (((yy % 4 == 0) && (yy % 100 != 0)) || (yy % 400 == 0))
        {
            if (dd > 29)
                return 0;
        }
        else
        {
            if (dd > 28)
                return 0;
        }
    }

    else if ((mm == 1) || (mm == 3) || (mm == 5) || (mm == 7) || (mm == 8)
|| (mm == 10) || (mm == 12))
    {
        if (dd > 31)
            return 0;
    }

    else
    {
        if (dd > 30)
            return 0;
    }
    return 1;
}
/*
RULES/RECOMMENDATIONS
STR00-C. Represent characters using an appropriate type
STR01-C. Adopt and implement a consistent plan for managing strings
STR02-C. Sanitize data passed to complex subsystems
STR03-C. Do not inadvertently truncate a string
STR04-C. Use plain char for characters in the basic character set
STR09-C. Don't assume numeric values for expressions with type plain
character
STR10-C. Do not concatenate different type of string literals
STR11-C. Do not specify the bound of a character array initialized with a
string literal
STR30-C. Do not attempt to modify string literals
STR31-C. Guarantee that storage for strings has sufficient space for
character data and the null terminator
*/
```

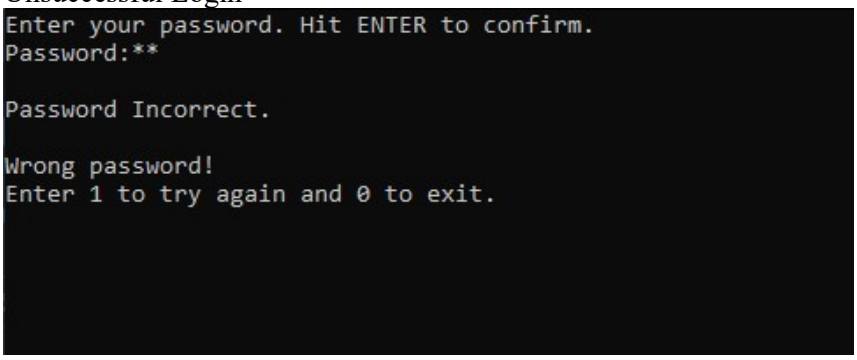
6. Testing and Verification

Logging In:

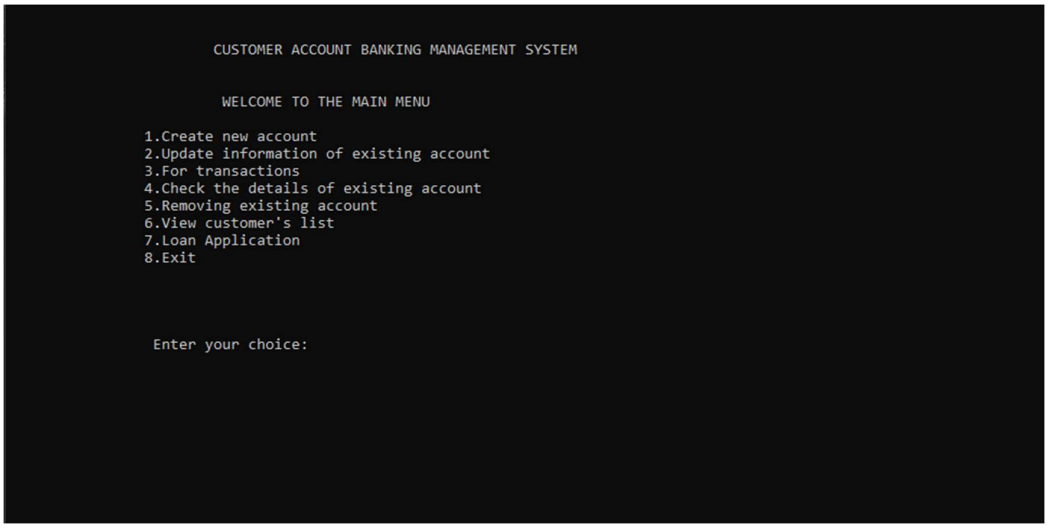
Successful Login



Unsuccessful Login



Main Menu:



Add Account:

Encounters pre-existing account number

```
                        ADD RECORD


Today's date (DD/MM/YYYY): 13/5/2020
Please enter the Account Number:2020107

2020101 Alan 12/2/1985 35 445,MountEdenRoad,MountEden,Auckland 2132143523 1350649340 current 10000.000000 7/5/2020

2020102 Bob 21/6/2010 19 2546,SociosquRd.,Bethlehem,Utah02913 3242342352 1840643873 current 100.000000 7/5/2020

2020103 Carl 12/3/2010 20 6308,LaciniaRoad,SanBernardino,ND09289 2345234242 248576247 current 10000.000000 7/5/2020

2020104 Dave 12/3/2012 8 5543,AliquetSt.,FortDodge,GA20783 214234525235 239576247 current 20.000000 7/5/2020

2020105 Evan 12/3/1984 36 7533,NonRd.,MiamiBeach,NorthDakota58563 234234 1052624954 current 100.000000 7/5/2020

2020106 Frank 12/9/1982 37 9383,SuspendisseAv.,Weirton,IN93479 42342342342 1360468840 saving 100000.000000 7/5/2020

2020107 George 21/7/1990 29 5982,SitAve,LiberalVermont51324 1238493433 1369115954 saving 1000000.000000 7/5/2020
Account no. already in use!
Please enter the Account Number:
```

Encounters invalid date of birth

```
Enter the name:Henry

Enter the date of birth (DD/MM/YYYY):54/23/2000
Kindly enter a valid date
Enter the date of birth (DD/MM/YYYY):
Enter the date of birth (DD/MM/YYYY):29/02/2001
Kindly enter a valid date
Enter the date of birth (DD/MM/YYYY):31/04/2000
Kindly enter a valid date
Enter the date of birth (DD/MM/YYYY):29/02/2000
```

Encounters pre-existing citizenship number

```
Enter the citizenship number:2132143523

2020101 Alan 12/2/1985 35 445,MountEdenRoad,MountEden,Auckland 2132143523 1350649340 current 10000.000000 7/5/2020
Uh Oh! Try again with proper credentials.
Enter the citizenship number:
```

Encounters invalid phone number

```
Enter the phone number: retret
Enter a valid phone number.

Enter the phone number: 12
Enter a valid phone number.

Enter the phone number:
```

Close Account:

```
Enter the account no. of the customer you want to delete:2020107

Record located.

You need to login before you can delete.Enter your password. Hit ENTER to confirm.
Password:****
```

Update Details:

Successfuly update details of an existing account

```
Enter the account no. of the customer whose info you want to change:2020107

Which information do you want to change?
1.Address
2.Phone

Enter your choice(1 for address and 2 for phone):2
```

Confirmation message indicates that the changes have been saved to the file.

```
Changes saved!


Enter 1 to go to the main menu and 0 to exit:
```

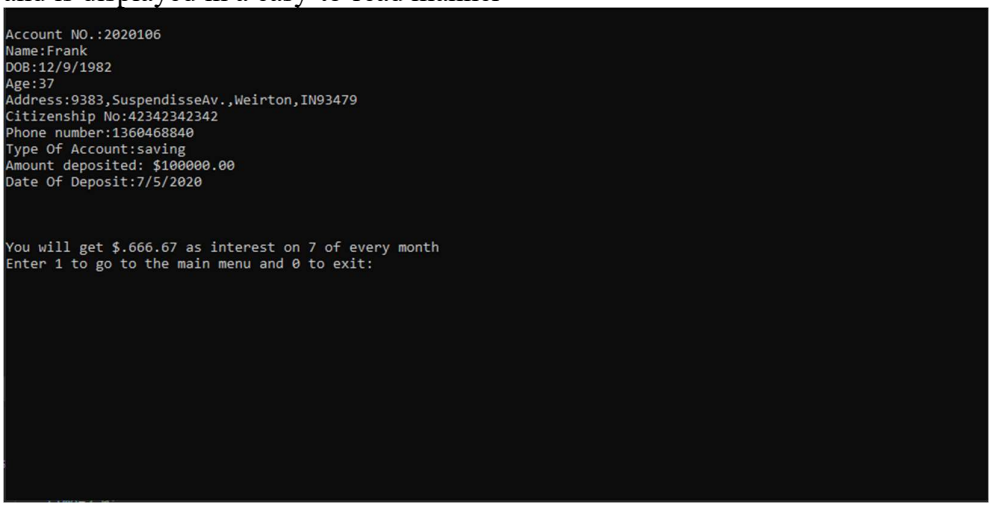
View List:

Shows a list of all existing accounts in an orderly manner prioritising account number and name. No sensitive inormation is displayed on this message although only administrators have access to it.

```
ACC. NO.        NAME            ADDRESS                         PHONE
2020101         Alan            445,MountEdenRoad,MountEden,Au  1350649340
2020102         Bob             2546,SociosquRd.,Bethlehem,Uta  -1840643873
2020103         Carl            6308,LaciniaRoad,SanBernardino  248576247
2020104         Dave            5543,AliquetSt.,FortDodge,GA20  239576247
2020105         Evan            7533,NonRd.,MiamiBeach,NorthDa  -1052624954
2020106         Frank           9383,SuspendisseAv.,Weirton,IN  1360468840

Enter 1 to go to the main menu and 0 to exit.
```

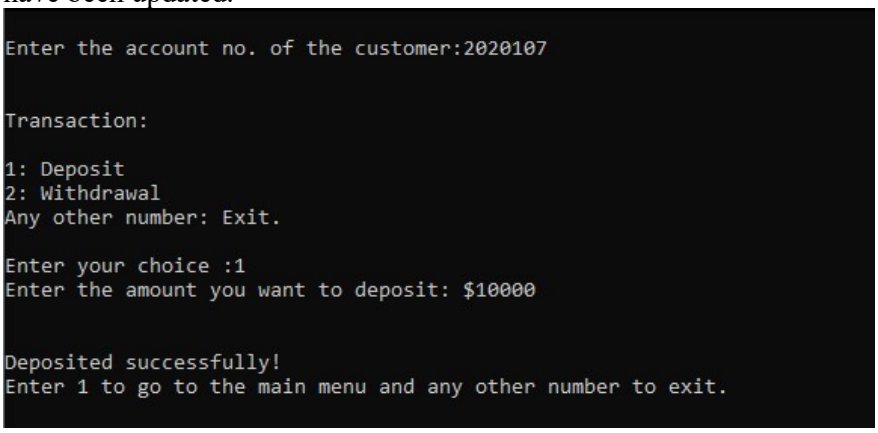
View Details:

Shows details of individual accounts. Each detail type is mentioned in separate lines and is displayed in a easy-to-read manner

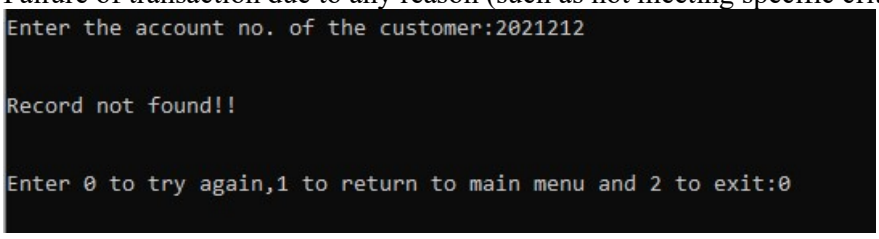


Transactions:

Success
Successful transaction indicates that the transaction has been processed and the records have been updated.



Failure
Failure of transaction due to any reason (such as not meeting specific criteria)



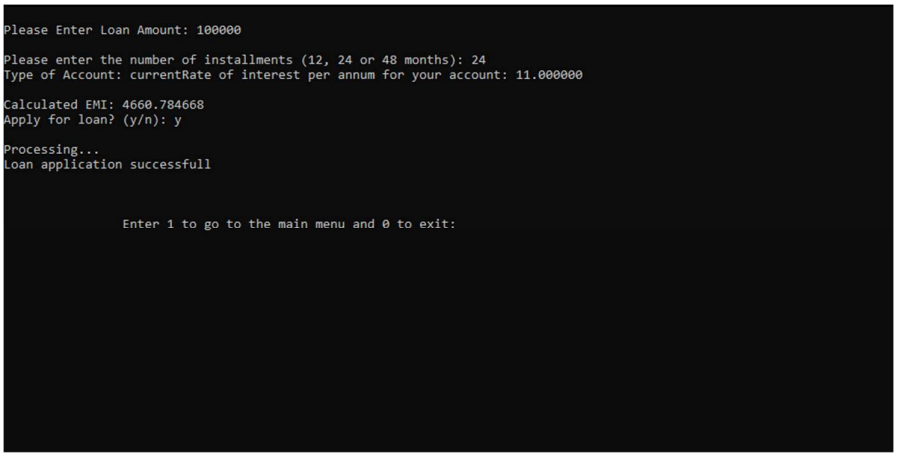
```
Enter the account no. of the customer:2020105


Transaction:

1: Deposit
2: Withdrawal
Any other number: Exit.

Enter your choice :2
Enter the amount you want to withdraw: $10000000000


Transaction declined.
Insufficient Funds in account.

Transaction:

1: Deposit
2: Withdrawal
Any other number: Exit.

Enter your choice :
```

Loan Application:

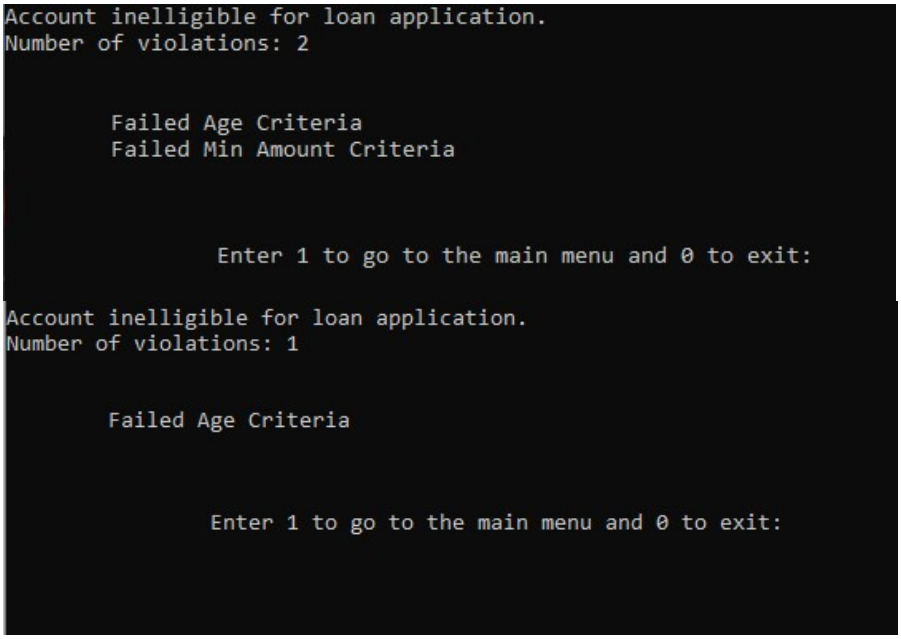
The below image displays the Opening menu of loan application.
It displays the Minimum criteria

```
Welcome to the loan application page
Eligibility Criteria:

Individuals who can take a Personal Loan:

        Salaried Employees
        Salaried doctors
        Employees of public and private limited companies
        Government sector employees including Public Sector Undertakings, Central and Local bodies
        Minimum age of 21 years
        Maximum age of 60 years at the time of maturity of the Personal Loan
        Minimum net monthly income Rs. 15,000



        Please enter account number:
```

Success



Failure



7. Conclusion

Banking systems, being one of the most fundamental systems in today's world – are inherently required to be stable, reliable, and most of all secure; uncompromisingly. Banks are frequent targets for multiple hack attempts and violations by those with malicious intent. Thus given their monumental importance, it is understandable why their security is utmost importance, even at the most elementary level.

It is with that motivation, that the task of creating this project; with a vision of creating a fundamental and secure banking system was undertaken.

The Secure Bank Management System has been built from the fundamentals of secure programming techniques, rules, and recommendations as standardized and documented by the governing body, CERT. These implementations help safeguard the data, and security of their financial footprint as well, while trying to maintain a fulfilling user interaction.

The rudimentary goal of implementation of these rules and recommendations is to develop safe, reliable, and secure systems, which is exactly what the primary basis of this project and it's requirements were, down to the last, smallest detail.

Conformance to the coding rules defined in the CERT standard have proven to be necessary (but not sufficient) to ensure the safety, reliability, and security of software systems developed in the C programming language. This secure system also serves as a testament to the fact that it isn't particularly tedious to adhere to these secure standards, and it doesn't necessarily exhibit a tradeoff with providing a user-friendly interface in the software being developed.
An opportunity to experience these virtues was made available to the developers, throughout the development process of this system.

8. Future Scope and Limitations

Although the application has been built from its grassroots revolving around the fundamentals of secure programming techniques, there is no denying that it could always have future work and updates to go along with the current functionality. Introduction of new features and functionalities would only help increase the scope and extensibility of the application. Further, as secure as the application may be, there still are ultimately little hitches that are bound to show up, even if they're not directly related to the execution of the program. That being said, the following are some of the few limitations, that could be overcome, and functionalities and features that we propose would be appropriate and a good fit for the application that may be undertaken at some point of time in the future.

- Corruption of the records file
  If the records file were to get corrupted by virtue of the user's negligence or any other factor, then all records would be inevitably corrupted as well
- Access to the records file
  If the records file isn't stored in a secure directory, it might give rise to scenarios wherein unauthorized personnel with malicious intents may have access to the data.
- "windows.h"
  The windows.h header has been included in the source code, and is utilized multiple times, particularly for clearing the screen. However, since the windows.h file isn't available for Linux based OSes or other OSes by default, it might give rise to issues, especially during development. This however may be tackled by using a library that provides a similar functionality or by using custom functions that serve the same purpose.
- Introducing database server support.
  Instead of a file stored on the local system/shared network, in one place, it would be a lot more efficient if the data was stored in a database hosted on a server. Further, this would also allow particular schemas to be tailored to suit one's needs, hence providing better meaningful relationships between the data, and making way for possibility of expanding the details that are tracked. In addition, the expected storage size of variables (such as name) also be extended by a far greater deal. This would also resolve any potential conflicts when two users might try to access and manipulate data at the same time. Security issues can also be addressed to a great deal, by following this approach.

9. Bibliography

- CERT. (2016). SEI CERT C Coding Standard: Rules for Developing Safe, Reliable, and Secure Systems. Pennsylvania, Pittsburgh, United States of America.

- John Viega, M. M. (2009). *Secure Programming Cookbook for C and C++ Recipes for Cryptography, Authentication, Input Validation & More.* O'Reilly Media.

- Seacord, R. C. (2013). *Secure Coding in C and C++, 2nd Edition.* Addison-Wesley.

- Tripathi, R. (n.d.). *Memory Layout of C Program.* Retrieved from HackerEarth: https://www.hackerearth.com/de/practice/notes/memory-layout-of-c-program/