**problem description:**

The project’s goal is to create a secure **Banking System**/**Bank Management Software**, which is accessibly by the employees of the bank, or those with the authorization to access, and which can be used to handle the basic functionalities that a bank provides.

The basic functionalities implemented so far include the following.

* Creating a New Account.
* Updating the information of an existing account.
* To perform transactions.
* Check the details of an existing account.
* Remove an existing account.
* Display the list of customers.

Further work:  
The primary focus is to implement these basic functionalities first, before moving on to slightly more intricate and detail driven services (such as loans/F.D.’s/Lockers) that a bank is expected to provide.

**requirement specification:**

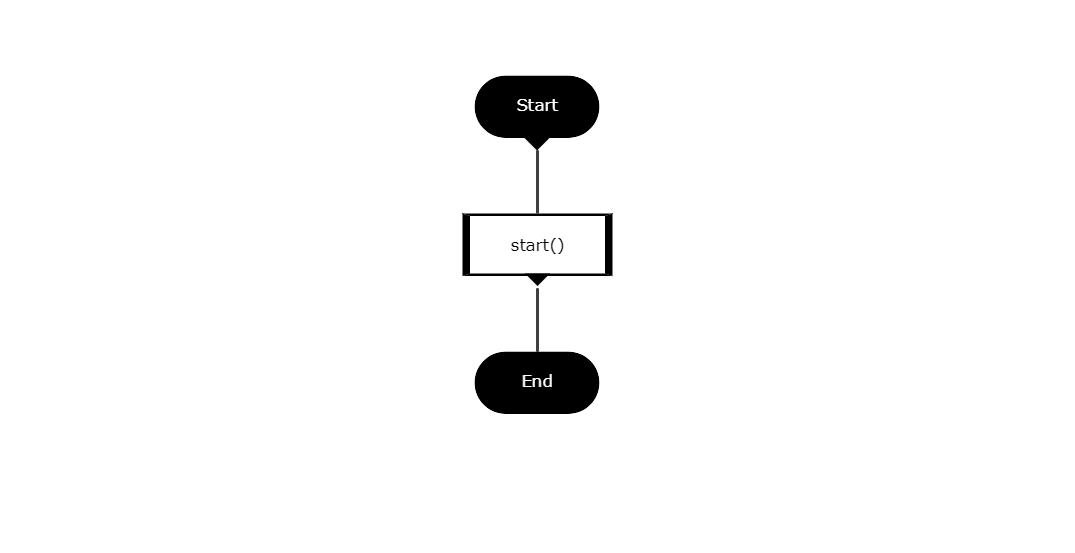
1. Files Required:
   1. Header Files:
      1. Inbuilt Header files:
         1. stdio.h
         2. conio.h
         3. string.h
         4. stdlib.h
         5. math.h
         6. windows.h
      2. User Defined Header files:
         1. data.h
         2. arjun.h
         3. anant.h
   2. Source Code:
      1. anant.c
      2. arjun.c
      3. util.c
      4. bank.c
2. Data Types adopted:
   1. structures
   2. global variables
   3. integers
   4. void
   5. floating point
   6. character
   7. array

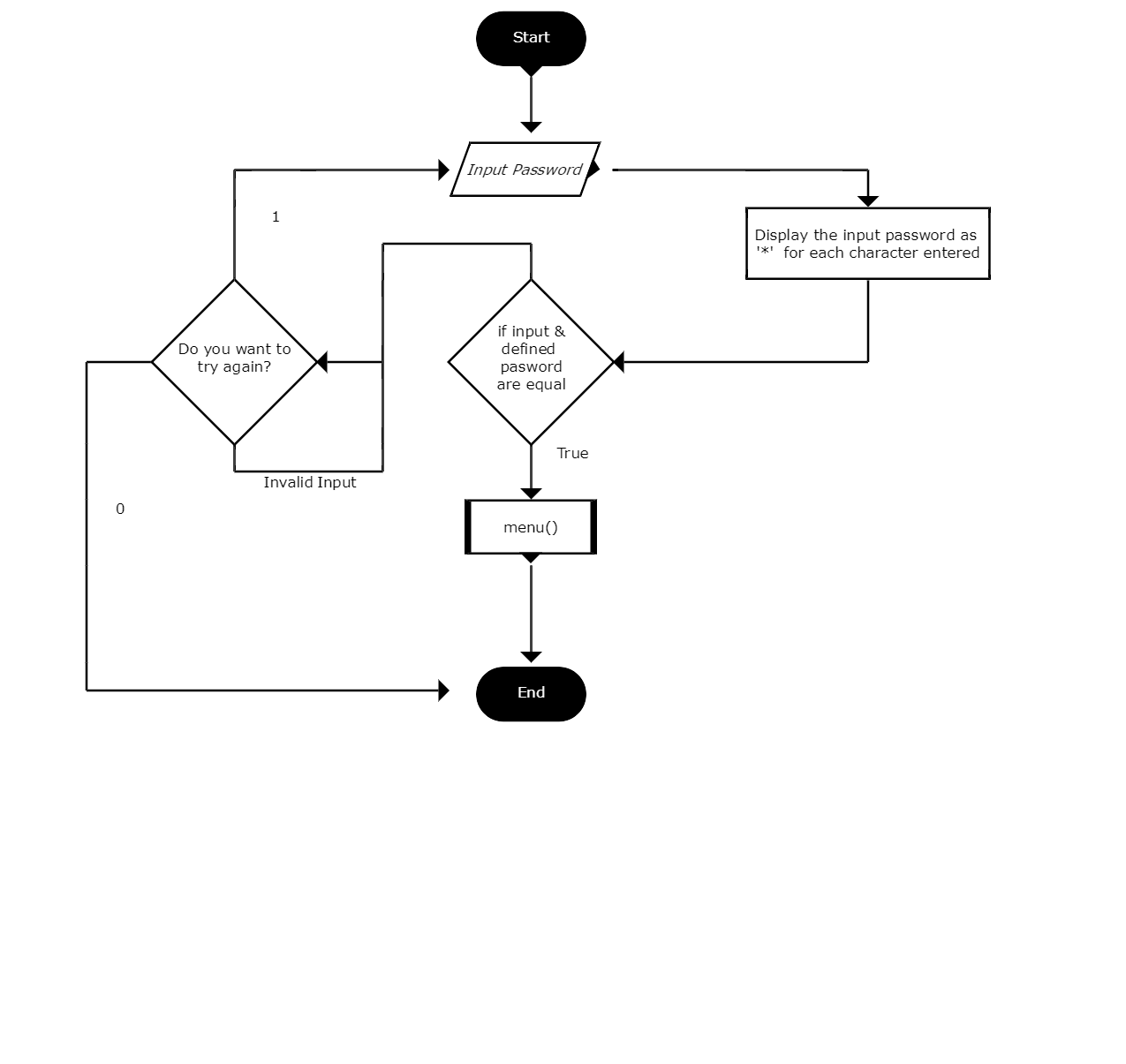
Further work:

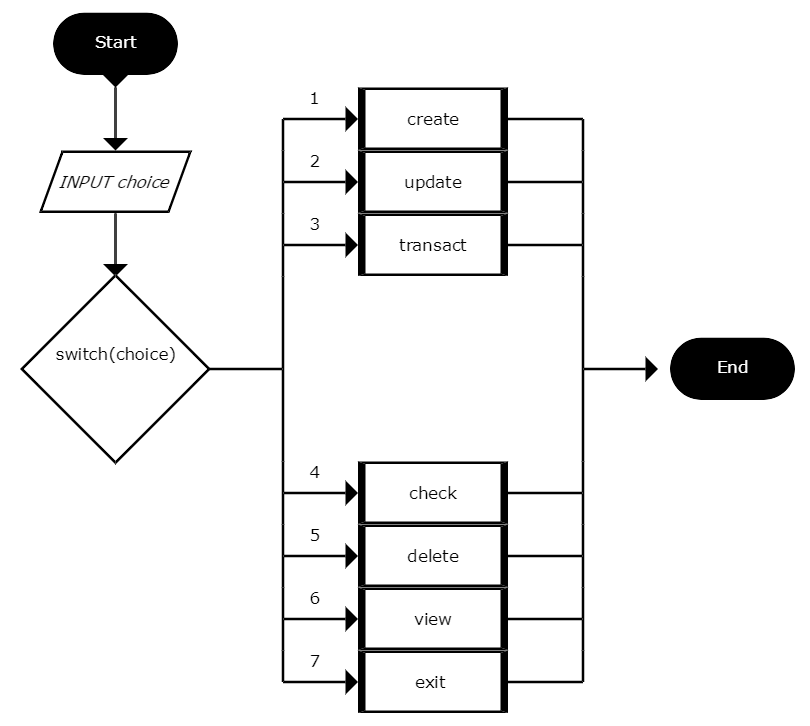
* Currently, the modules divided between the two contributors to the project is split majorly into two files, anant.c, arjun.c; with another file util.c housing the important functions that are required by both anant.c, and arjun.c. This will be changed, and each major option, will be categorized, and re-modularization of the program will be done once more, for better readability of code.
* It must be noted that the program so far, specifically utilizes data types, all of which have explicitly predetermined/predefined sizes. Thus, we will be trying to include provisions for dynamic memory allocation, and be refining the code for optimal memory management. Optimal memory management can be considered by trying to split out a initially defined structure’s variables into variables of a type that is a subset of the initial structure, as per requirement.
* We will also be exploring more avenues and trying to include scenarios in which usage of pointers would be more efficient and flexible.

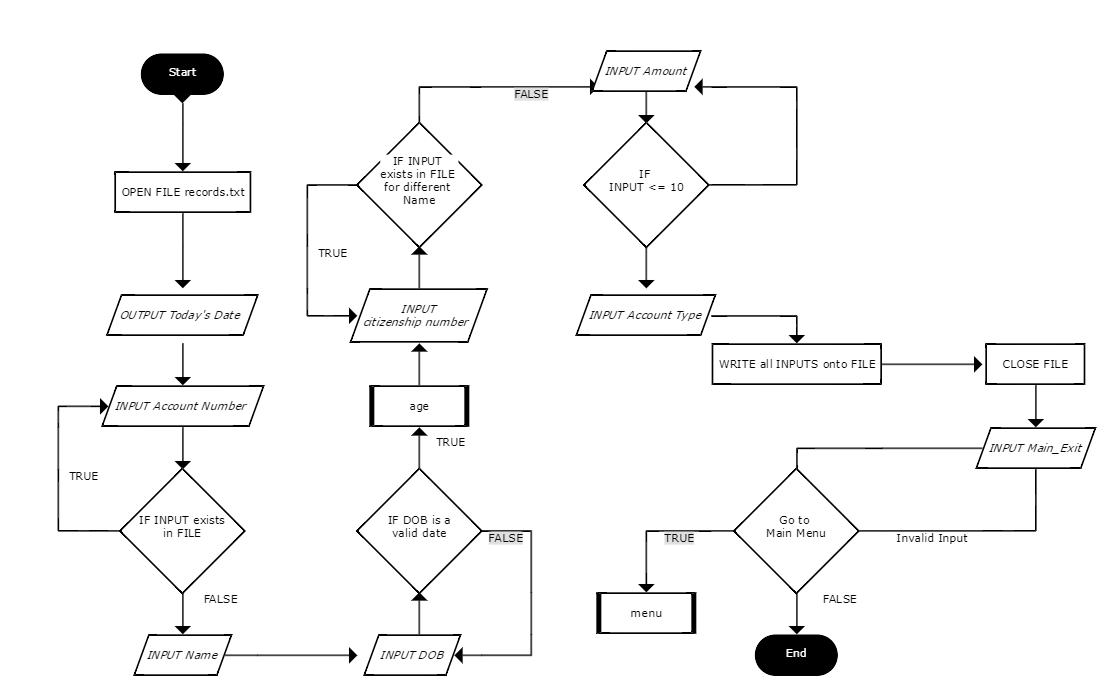
**Flow of the problem:**

**main ():**

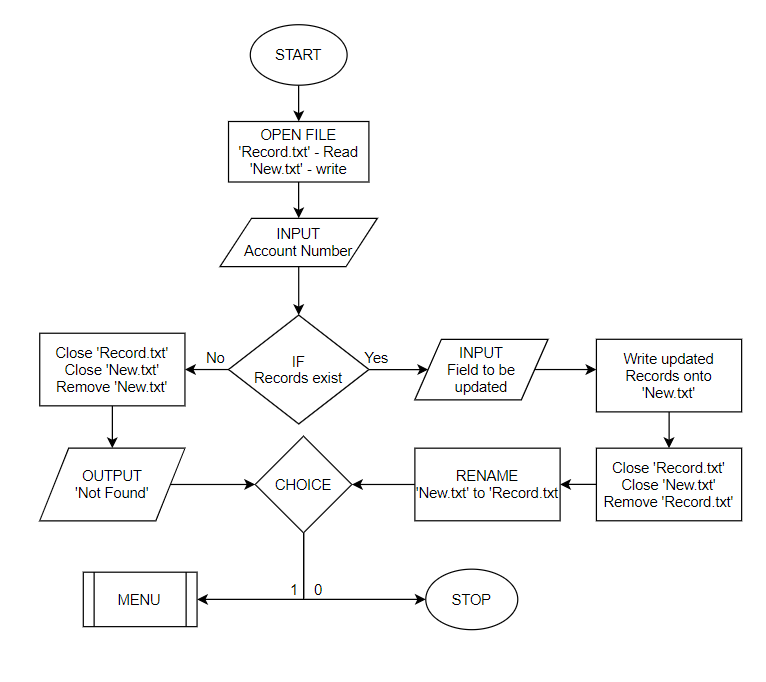


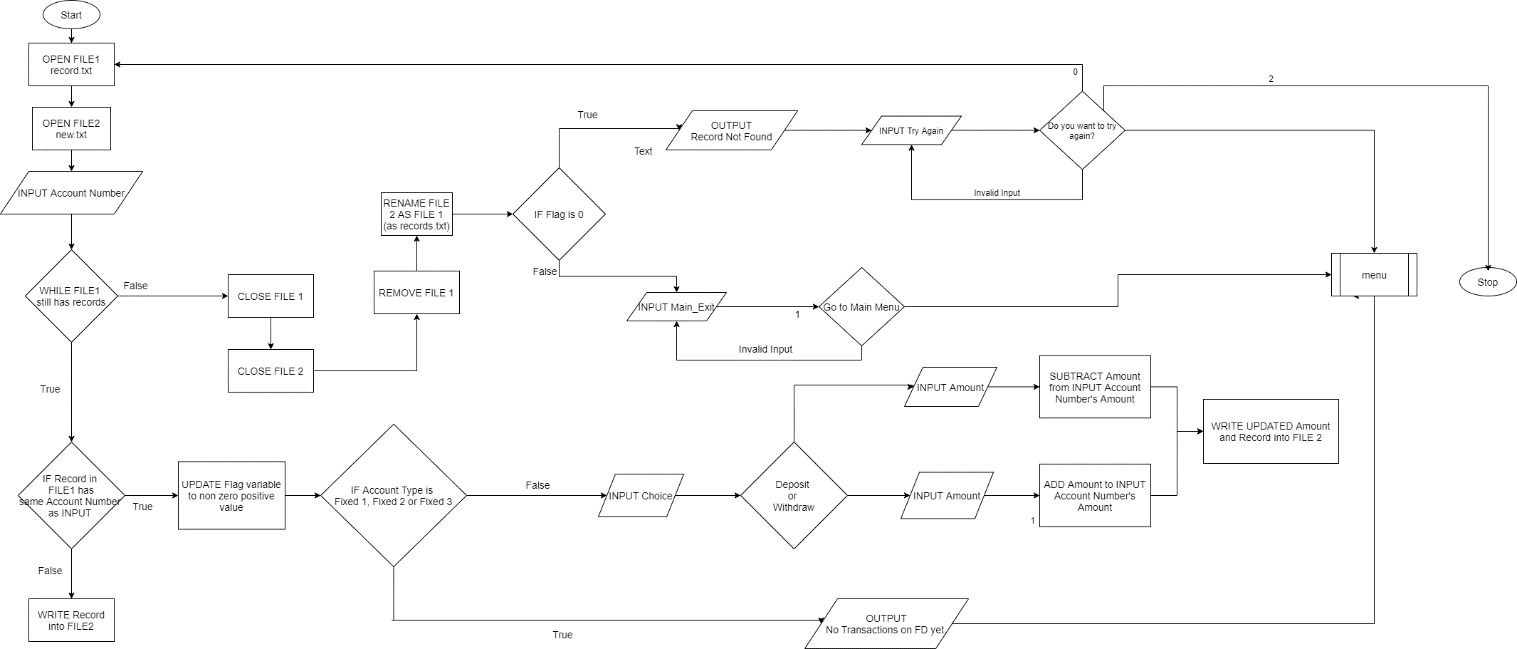
**start ():**

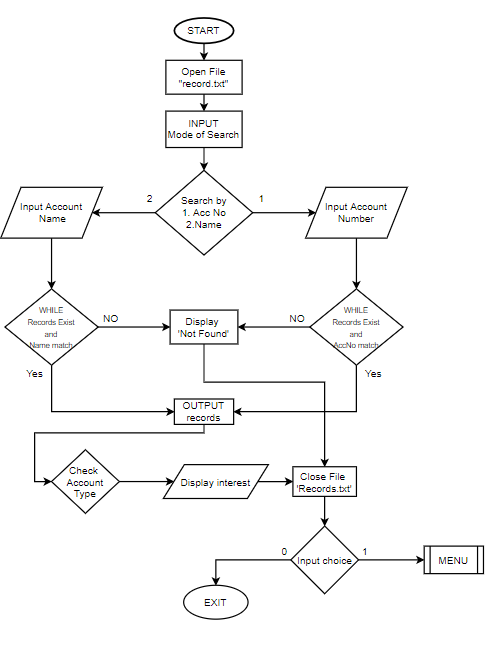
**menu ():  
**

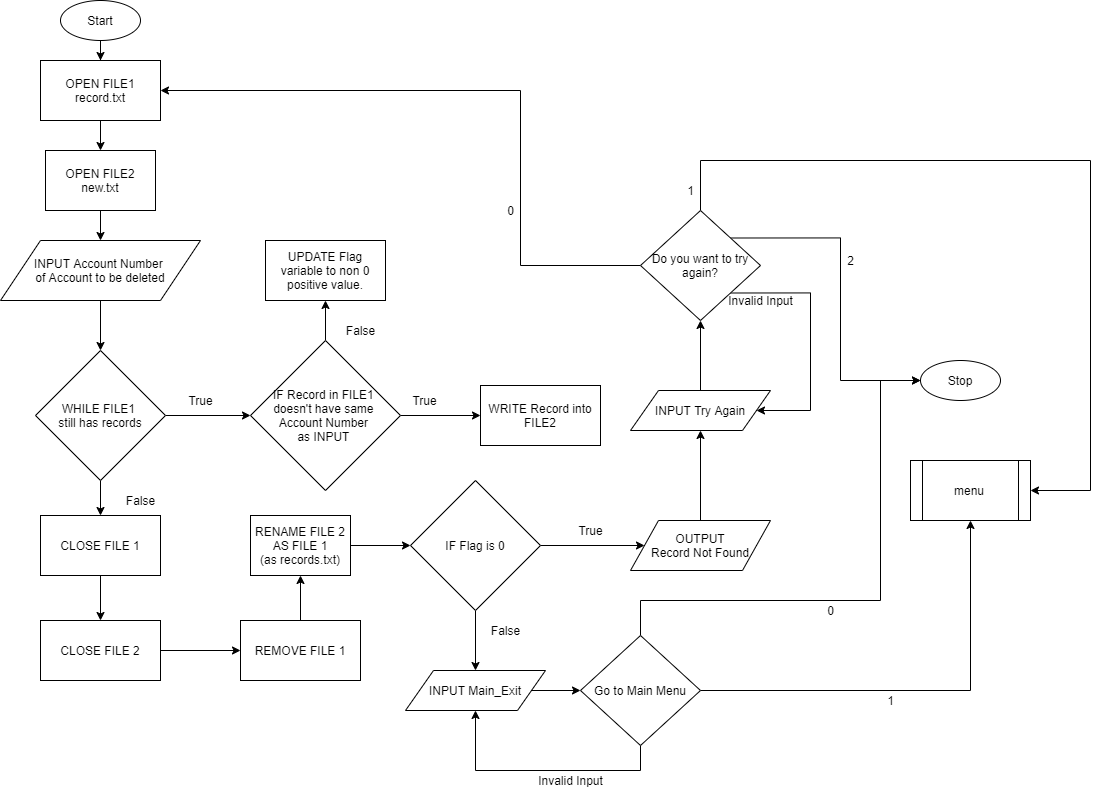
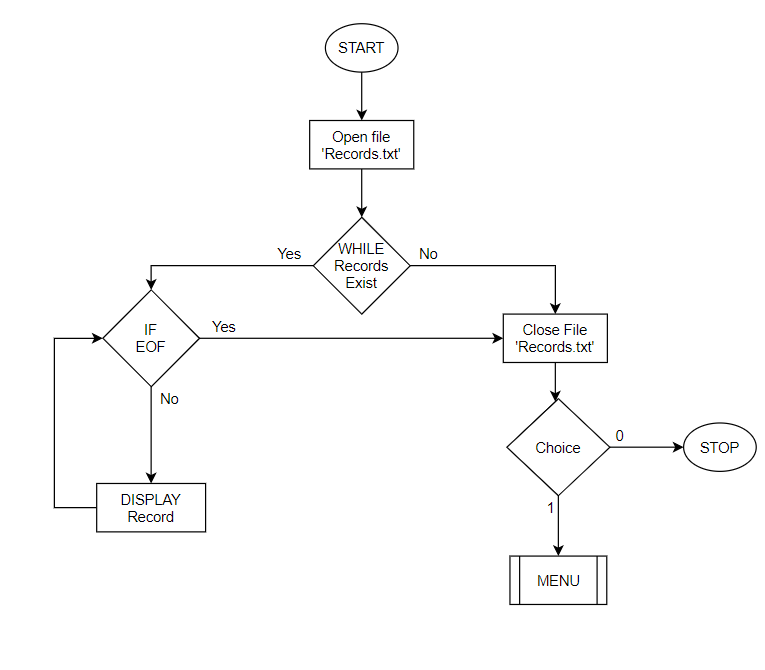
**create ():  
**

**update ():**



**transact():** **check ():**



**delete ():** **view ():** **Rules and recommendations Adopted:**

**Preprocessor (PRE)**

**PRE30-C. Do not create a universal character name through concatenation.**We don’t have the need for special/unique characters that aren’t in the basic character set. Hence, this hasn’t be incorporated.

**PRE31-C. Avoid side effects in arguments to unsafe macros**/\*  
Working on making a variable format macro, for the fscanf and fprintf arguments.  
\*/

**PRE32-C. Do not use preprocessor directives in invocations of function-like macros**/\*  
Working on making a variable format macro, for the fscanf and fprintf arguments.  
\*/  
Conditional macro statements have been evaluated by requiring a separate argument to be passed, as opposed to using #ifdef, or #else.

**PRE00-C. Prefer inline or static functions to function-like macros**inline function defined for calculating interest.

**Declarations and Initialization (DCL)**

**DCL00-A. Declare immutable values using const or enum**

Several immutable values and pointers have been declared using keywords ‘const’ and ‘enum’ in order to improve readability and reliability of the code.

**DCL01-A. Do not reuse variable names in sub-scopes**

All variables declared have carefully thought out names unique to their role and contribution to their individual functions. Hence there is no repetition of variable names within the entirety of the program

**DCL02-A. Use visually distinct identifiers**

Visually distinct identifiers are used with meaningful names to eliminate errors resulting from misreading the spelling of an identifier during the development and review of code. Identifiers are named such that anyone reading the code will be able to understand their use and functionality clearly. Failing to use visually distinct identifiers can result in referencing the wrong object or function, causing unintended program behaviour.

**DCL03-A. Place const as the rightmost declaration specifier**

Immutable objects are const-qualified. Enforcing object immutability using const qualification helps ensure the correctness and security of applications. ‘const’ has been used as the rightmost declaration specifier wherever possible

**DCL04-A. Take care when declaring more than one variable per declaration**

Every declaration has a single variable, on its own line, with an explanatory comment about the role of the variable. Declaring multiple variables in a single declaration can cause confusion regarding the types of the variables and their initial values. If more than one variable is declared in any declaration, care is taken that the type and initialized value of the variable are handled correctly.

**DCL05-A. Use typedefs to improve code readability**

The use of “typedef” is incorporated in the code. Typedefs help improve the general readability and error handling of the code.

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

When used in program logic, literals can reduce the readability of source code. Rather than embed literals in program logic, use appropriately named symbolic constants to clarify the intent of the code. In addition, if a specific value needs to be changed, reassigning a symbolic constant once is more efficient and less error prone than replacing every instance of the value. Constants have replaced literal values such as rates of interest for several types of accounts to enhance code readability

**DCL07-A. Ensure every function has a function prototype**

Function declarators must be declared with the appropriate type information, including a return type and parameter list. If type information is not properly specified in a function declarator, the compiler cannot properly check function type information. Failing to include type information for function declarators can result in [unexpected](https://wiki.sei.cmu.edu/confluence/display/c/BB.+Definitions#BB.Definitions-unexpectedbehavior) or unintended program behaviour.

**DCL09-A. Declare functions that return an errno with a return type of errno\_t**

Failing to test for error conditions can lead to vulnerabilities of varying severity. Functions are programmed to return error codes wherever necessary. They return an error number ‘errno’ with return type of errno\_t. This will in tur reduce the errors caused by programmers misunderstanding the requirement of a return value.

**DCL12-A. Create and use abstract data types**

The use of opaque abstract data types, can significantly reduce the number of defects and vulnerabilities introduced in code, particularly during ongoing maintenance.

**DCL30-C. Declare objects with appropriate storage durations**All objects have appropriate storage durations specified. Even files that are no longer required are closed once required operation have been performed using the files. There are no dynamically allocated variables, since the Bank Management System for effective abstraction, and user friendliness.

**Different Storage Durations:** Objects with different storage durations haven’t been  
made equivalent either at any point of time in the code.

**Output Parameter/Different Return Values:** Functions don’t return addresses/pointers associated with objects with automatic storage duration. This is because, the major functions are implemented in an attempt to make them as self-sufficient as possible, since they do not depend on return values for details such as the details of an account, since it wouldn’t be an effective storage/computing solution.

**DCL31-C. Declare identifiers before using them**

**Implicit int** (variable type specifier): All variables have been type specified and declared before invoking.

**Implicit Function Declaration:** Care has been taken to correctly andexplicitly prototype every function before invoking it.

**Implicit Return Type:** For the functions that have a return value, it has been ensured that the type of the return value is matching with the type specifier declared for the function.

**DCL35-C. Do not convert a function pointer to a function of a different type**

Conversion of a function pointer to a different type of pointer results in undefined behaviour. This means, for example, that a pointer to a function cannot be converted to a pointer to a different type of function. Hence a lot of care has been taken not to let this happen as it would compromise the security of the code.

**DCL36-C. Do not declare an identifier with conflicting linkage classifications**

A thorough attempt has been made to ensure that there is no conflict between the variables’ identifiers and that objects have been declared with appropriate storage durations, in an attempt to try and prevent overlapping of scopes/give way to conflicting linkage classifications.

**DCL37-C. Do not declare or define a reserved identifier**No reserved identifiers have been declared.

Further work:

* We will be trying to include more recommendations, as well rules; whilst prioritizing the inclusion of the rules since they’re more essential to the scope of this project. This will happen simultaneously whilst doing the work required to meet all the other set goals and expectations for the project, as detailed in the “Further work” sections.