FAMILY BUDGET MANAGER SYSTEM

Domingo, Rodney C.
Estorco, Clint Hoven A.
Formilleza, Huone
Panti, Dominic Rodnee A.
Salamanca, Jacob V.

Technological Institute of the Philippines
Quezon City

November 2025

FAMILY BUDGET MANAGER SYSTEM..... 1
Introduction

The Family Budget Manager System is a C++ application and is a console-based application that allows families to keep track of their income and expenses, user profiles (parents and children), and allowance requests. The system archives information in plain text files, and it has role-based menus that allow parents to manage budgets and allowances requested of children. The goals of this project are to learn the software design concepts (data structures, file IO, simple user interface, validation) and create a functional tool that can be used by families to gain a deeper insight and more effectively manage their finances.

SPECIFIC PROBLEM

Many families struggle to keep organized records of incomes, recurring expenses, and allowances.
Common problems:.....4
● No single place to record small daily expenses → leads to inaccurate monthly budgeting.....4
● Children request money without parents having a transparent approval/record system..... 4
● Parents cannot easily see summaries (total income, total expense, balance, largest expenses).....4
● Lack of role separation: children should be able to request money but not edit parent records..... 4
These issues make it hard to plan savings and to teach children financial responsibility

Introduced Solution.....4
We propose a lightweight Family Budget Manager System that:.....4
● Provides two user roles: Parent and Child (children under/over 18 behave slightly differently)..... 4
● Let's parents add, view, edit, delete budget records (income/expense), manage accounts and child profiles, and process allowance requests.....4
● Let children submit requests for allowance; parents can approve or reject requests. Approved requests automatically create expense records..... 4
● Produces summary reports (totals, averages, largest items, pending requests) to help families make informed decisions..... 4
The Project.....4
Objectives..... 5
Flowchart of the System.....5
Pseudocode..... 8
Data Dictionary..... 38
DATA DICTIONARY — Family Budget Manager System..... 39
STRUCT: Budget..... 39
STRUCT: Request..... 39
STRUCT: Account..... 40
STRUCT: Profile..... 40

Global Arrays....41
Counter and ID Variables....41
Helper & Utility Functions....42
File Storage....42
Computation Variables (Used in Reports)....43
Code....48
Results and Discussion....65
Conclusion....71
References....72

Introduction

The Family Budget Manager System is a C++ console-based program that aims to enable families to easily manage and monitor their finances- their income, expenses, user profiles, and allowance requests.

Some common problems with keeping organized financial records among many families include: a lack of a single financial location to capture small miscellaneous day-to-day expenses; missing accurate monthly budgeting; children demanding money without any official system of request and transferring in and out; a lack of easy access to sums of money by the parents; highest expenditure, balances, and total income. Moreover, role separation does not exist; thus, children can access the records of parents unnecessarily and change them, so it would be hard to plan how to save and teach children the importance of becoming financially responsible. In order to overcome these problems, the system offers two separate roles: Parent and Child. The parents are able to add, see, edit, and remove budget records, maintain accounts and child profiles, and handle child allowance requests. Children are able to place requests for allowance, which their parents can either accept or decline; approved requests automatically create expense records. The system generates summary reports, e.g., totals, averages, largest items, and pending requests, which allow families to make wise decisions and help make money management responsible.

As Nakagawa and Shigekawa (2022) state, long-term household financial accounts and structured budgeting techniques are likely to reveal that families are aware of their income and expenditure, prevent deficit, and attain the necessary financial literacy when they keep their financial records well organized.

The Project

Family Budget Manager System helps a household to track income, expenses, and savings towards the realization of financial goals. The major elements are setting financial goals, tracking spending habits,

budget plan, managing credit and debt control, and savings and investment schemes. Compliance with these processes allows families to spend money effectively, deal with debts, save money in the future, and improve their financial situation.

The budgeting system works under a principle whereby the first step is to determine the sources of income and then categorize the expenditures into fixed, variable, and discretionary. This classification can be automated with the administration tools, like spreadsheets or budgeting applications. The most commonly used allocation plan is the 50/30/20, which allocates half of the after-tax income to basic necessities, thirty percent of the income to discretionary wants, and twenty percent to savings and paying back debts.

This section gives the results of the Family Budget Manager System after developing and testing. It talks about the system functionality, usability, and efficiency regarding the simulated situations of the interacting parts of parents and children. The discussion sheds light on each of the features, including budget tracking, allowance requests, and summary reporting, that contributed to the realization of the project objectives. The observations, identified problems, and their solutions are also implemented to assess the effectiveness of the system as a whole. The budget is still in tune with the emerging financial conditions and goals of the household; periodic reviews and changes are necessary (Consumer Financial Protection Bureau, 2023; Lusardi, 2019).

Objectives

The project aims to develop a family budget management system that helps to determine and optimize the family's overall budget by allowing parents and children to monitor, record, and control financial activities in one platform. It ensures proper allocation of resources and promotes responsible spending habits within the family. Specifically, it aims to develop:

- To develop a login interface that determines whether the user is a child or an adult.
- To develop a system that can indicate the amount of the family's expenses for a given period of time.
- To develop a feature that records, edits, deletes, and searches budget data such as income and expenses.
- To develop a system that lets the user (parents) handle the family's finances by monitoring expenses and assigning the children's allowance.
- To develop a request system where children can send allowance or budget requests to parents for approval or rejection.
- To develop a summary and report feature that displays total income, expenses, balance, and pending requests.

Flowchart of the System

The flowchart indicates the processes involved in making the code work in the background. It illustrates the actions needed to operate the system and how the code lets the user interact with it. It showcases the proper sequence of steps and the branching functions for the management and user categories of the users' budgets and changes.

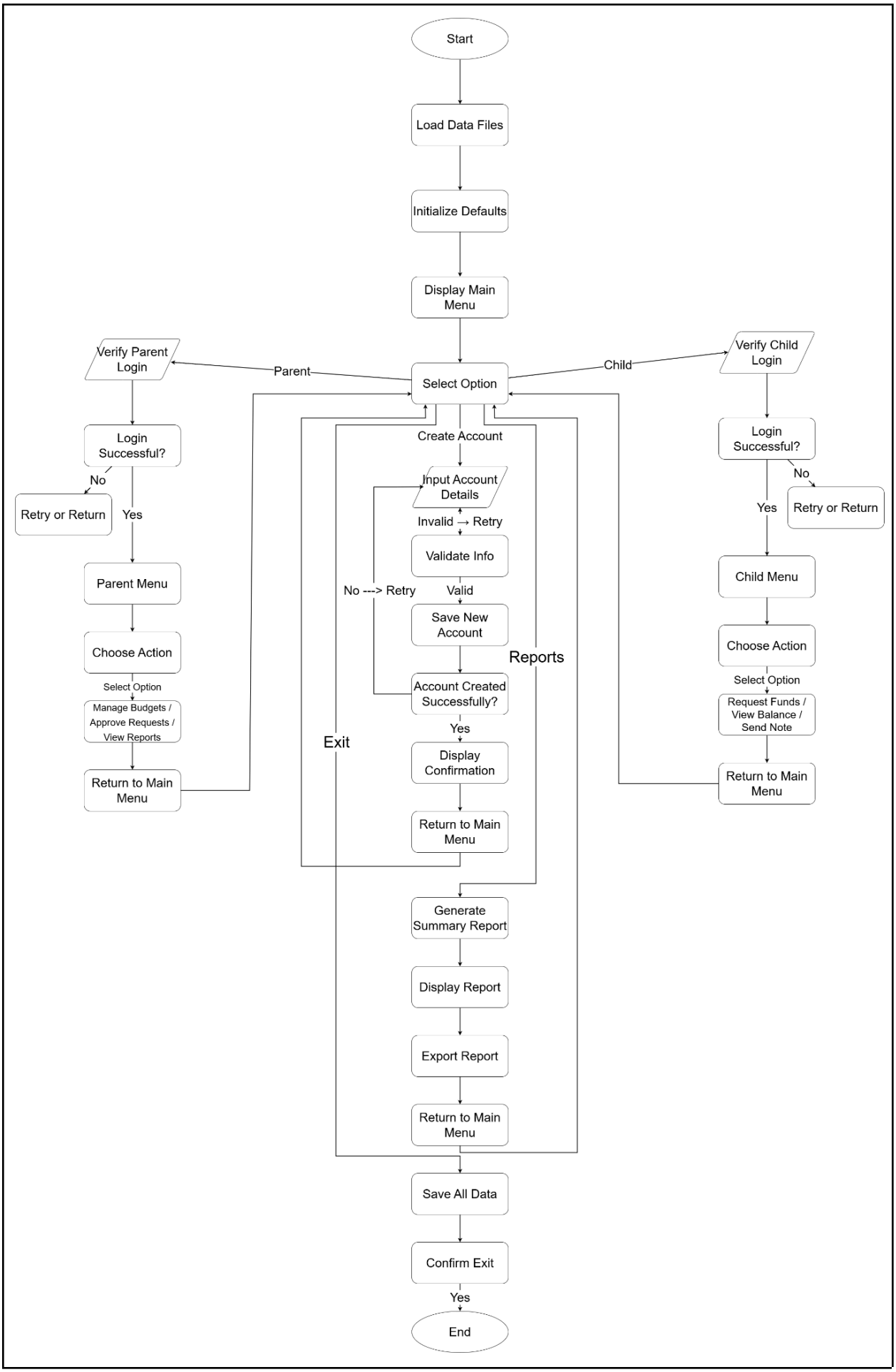
Start
Load Data Files
Initialize Defaults
Display Main Menu
Select Option
Parent
Child
Verify Parent Login
Login Successful?
No
Retry or Return
Yes
Parent Menu
Choose Action
Select Option
Manage Budgets / Approve Requests / View Reports
Return to Main Menu
Verify Child Login
Login Successful?
No
Retry or Return
Yes
Child Menu
Choose Action
Select Option
Request Funds / View Balance / Send Note
Return to Main Menu
Create Account
Input Account Details
Invalid → Retry
Validate Info
No ---> Retry
Valid
Save New Account
Account Created Successfully?
Yes
Display Confirmation
Return to Main Menu
Exit
Reports
Generate Summary Report
Display Report
Export Report
Return to Main Menu
Save All Data
Confirm Exit
Yes
End

Figure 1: FBM Flowchart

The flowchart showcases the main operational processes of a Family Budget Management System, which starts with data initialization and user authentication. The system first differentiates between parent and child account users, along with users setting up an account, and lets them log in to view and use only the tools available in their own menus. The Child branch gets a restricted interface for managing budgets and spending approvals, as well as viewing their budget balance. The Parent section includes the management of all budgets, approval of child requests, and viewing of financial reports. All user paths return to the main navigation menu; after a confirmed exit, the system automatically saves data, and thus, information persists across sessions. This structure provides a secure, role-based framework for collaborative family finance management.

Pseudocode

This pseudocode shows the flow of the Family Budget Management System. It focuses on how budgets, requests, and user accounts are processed and managed. Through this, the system ensures proper tracking of income, expenses, and financial requests between parents and their children.

```
START

CONSTANT MAX_BUDGETS ← 600
CONSTANT MAX_REQUESTS ← 200
CONSTANT MAX_ACCOUNTS ← 200
CONSTANT MAX_PROFILES ← 200


DECLARE STRUCT Budget
     id : number
     name : text
    amount : number
     type : text
    owner : text
```

```
END STRUCT


DECLARE STRUCT Request
     id : number
    childUsername : text
    childName : text
    purpose : text
    amount : number
    status : text
END STRUCT


DECLARE STRUCT Account
    username : text
    password : text
     role : text
     age : number
END STRUCT


DECLARE STRUCT Profile
    username : text
    displayName : text
     age : number
    allowance : number
END STRUCT
```

```
DECLARE budgets[MAX_BUDGETS] AS Budget
DECLARE requests[MAX_REQUESTS] AS Request
DECLARE accounts[MAX_ACCOUNTS] AS Account
DECLARE profiles[MAX_PROFILES] AS Profile

DECLARE budgetCount ← 0
DECLARE nextBudgetId ← 1
DECLARE requestCount ← 0
DECLARE nextRequestId ← 1
DECLARE accountCount ← 0
DECLARE profileCount ← 0



FUNCTION toLowerStr(s : text) RETURNS text
     SET r ← s
     FOR i FROM 0 TO LENGTH(r) - 1 DO
        r[i] ← lowercase(r[i])
     END FOR
    RETURN r
END FUNCTION



FUNCTION isNumberString(s : text) RETURNS logic
```

```
     SET hasDigit ← false
     SET hasDot ← false
     FOR EACH c IN s DO
        IF c IS DIGIT THEN
            hasDigit ← true
        ELSE IF c = '.' AND hasDot = false THEN
            hasDot ← true
        ELSE
            RETURN false
        END IF
     END FOR
    RETURN hasDigit
END FUNCTION



FUNCTION getDoubleInput(prompt : text) RETURNS number
     LOOP FOREVER
        DISPLAY prompt
        INPUT line
        IF line IS EMPTY THEN
            CONTINUE LOOP
        END IF

        IF isNumberString(line) = true THEN
```

```
            TRY
                val ← CONVERT line TO number
                RETURN val
            CATCH ERROR
                DISPLAY "Invalid number format. Try again."
            END TRY
        ELSE
            DISPLAY "Invalid number. Try again."
        END IF
    END LOOP
END FUNCTION



FUNCTION getIntInput(prompt : text) RETURNS number
    LOOP FOREVER
        DISPLAY
        INPUT line
        IF line IS EMPTY THEN no
            CONTINUE LOOP
        END IF

        ok ← true
        i ← 0
        IF line[0] = '-' THEN
```

```
            i ← 1
        END IF

        FOR j FROM i TO LENGTH(line) - 1 DO
            IF line[j] IS NOT DIGIT THEN
                ok ← false
                BREAK
            END IF
        END FOR

        IF ok = true THEN
            TRY
                v ← CONVERT line TO number
                RETURN v
            CATCH ERROR
                DISPLAY "Invalid integer. Try again."
            END TRY
        ELSE
            DISPLAY "Invalid integer. Try again."
        END IF
    END LOOP
END FUNCTION
```

```
FUNCTION getLineInputNonEmpty(prompt : text) RETURNS text
     LOOP FOREVER
        DISPLAY prompt
        INPUT line
        IF line IS NOT EMPTY THEN
            RETURN line
        ELSE
            DISPLAY "Input cannot be empty. Try again."
        END IF
     END LOOP
END FUNCTION



PROCEDURE saveBudgetsToFile()
     OPEN FILE "budgets.txt" FOR WRITING AS f
     IF FILE NOT OPEN THEN
        RETURN
     END IF
     FOR i FROM 0 TO budgetCount - 1 DO
        WRITE budgets[i].id, ",", budgets[i].name, ",",
FORMAT(budgets[i].amount, 2), ",",
               budgets[i].type, ",", budgets[i].owner TO f
     END FOR
    CLOSE FILE f
END PROCEDURE
```

```
END
Start
Procedure loadProfilesFromFile
     Open file "profiles.txt" for reading
     If file cannot be opened then
        Return
    EndIf

     Set profileCount = 0
    While not end of file AND profileCount < MAX_PROFILES do
        Read line
        If line is empty then
            Continue
        EndIf

        Find positions of commas (p1, p2, p3)
        If any comma not found then
            Continue
        EndIf

        Try
            Extract and assign substrings to Profile p:
                p.username = part before p1
```

```
                p.displayName = part between p1 and p2
                p.age = convert part between p2 and p3 to
integer
                p.allowance = convert part after p3 to real
            Add p to profiles array
            Increment profileCount
        Catch any error
            Continue
        EndTry
    EndWhile

    Close file
EndProcedure
End
 Start
Function computeTotalIncome return Real
     Set s = 0
     For i = 0 to budgetCount - 1 do
        If budgets[i].type (converted to lowercase) = "income"
then
            s = s + budgets[i].amount
        EndIf
    EndFor
    Return s
EndFunction
End
```

```
 Start
Function computeTotalExpense return Real
     Set s = 0
     For i = 0 to budgetCount - 1 do
        If budgets[i].type (converted to lowercase) = "expense"
then
            s = s + budgets[i].amount
        EndIf
    EndFor
    Return s
EndFunction
End
 Start
Procedure drawHeaderBox(title)
     Set width = 60
    Print line with "+" and "=" repeated width times
    Compute padding = (width - title length) / 2
    Print title centered within box
    Print bottom line of box
EndProcedure
End
 Start
Procedure showAllBudgetsFancy
     If budgetCount = 0 then
```

```
        Print "No budget records available."
        Return
    EndIf

    Print table header
     For i = 0 to budgetCount - 1 do
        Print each budget's ID, Name, Amount, Type, and Owner
    EndFor
    Print total income, total expense, and balance
EndProcedure
End
 Start
Procedure addBudgetInteractive(owner)
     If budgetCount >= MAX_BUDGETS then
        Print "Budget storage full."
        Return
    EndIf

    Create new Budget b
     Set b.id = nextBudgetId + 1
     Ask user for record name and store in b.name
     Ask user for amount and store in b.amount
     Ask user for type (Income or Expense)
    While type is invalid do
```

```
        Ask again
    EndWhile


     Set b.type = input type
     Set b.owner = owner
     Add b to budgets array
    Increment budgetCount
     Save budgets to file
    Print "Record added."
EndProcedure
End
 Start
Function findBudgetIndexById(id) return Integer
     For i = 0 to budgetCount - 1 do
        If budgets[i].id = id then
            Return i
        EndIf
    EndFor
    Return -1
EndFunction
End
 Start
Procedure editBudgetInteractive(currentUser, isParentOrAdult)
     If budgetCount = 0 then
```

```
        Print "No budgets to edit."
        Return
    EndIf

     Ask user for budget ID to edit
     Set idx = findBudgetIndexById(ID)
     If idx < 0 then
        Print "Budget not found."
        Return
    EndIf

     If user is not parent/adult AND not owner of budget then
        Print "You can only edit your own records."
        Return
    EndIf

    Display current details
     Ask for new name (optional)
     Ask for new amount (optional, must be a number)
     Ask for new type (optional, must be Income or Expense)
    Update values if new input given
     Save budgets to file
    Print "Budget updated."
EndProcedure
```

```
End


Start
Procedure deleteBudgetInteractive(currentUser, isParentOrAdult)
     If budgetCount = 0 then
        Print "No budgets to delete."
        Return
    EndIf

     Ask user for budget ID to delete
     Set idx = findBudgetIndexById(ID)
     If idx < 0 then
        Print "Budget not found."
        Return
    EndIf

     If user is not parent/adult AND not owner of record then
        Print "You can only delete your own records."
        Return
    EndIf

    Shift all elements after idx one position left
    Decrease budgetCount by 1
     Save budgets to file
```

```
    Print "Budget deleted."
EndProcedure
End
 Start
Procedure searchBudgetsMenu
    Print "Search by: 1) Name 2) Type 3) Owner 4) Amount range"
     Get user choice (ch)

     If ch = 1 then
        Ask for name keyword
        For each budget, check if name contains keyword
            If match found, print details
        If no match, print "No matches."

    ElseIf ch = 2 then
        Ask for type (Income/Expense)
        For each budget, check if type matches
            If match found, print details
        If no match, print "No matches."

    ElseIf ch = 3 then
        Ask for owner username
        For each budget, check if owner matches
            If match found, print details
```

```
        If no match, print "No matches."


    ElseIf ch = 4 then
        Ask for low and high amount
        For each budget, check if amount is within range
            If match found, print details
        If no match, print "No matches."
    EndIf
EndProcedure
End
START


PROCEDURE sortBudgetsMenu()
     DISPLAY "Sort by: 1) Amount desc 2) Amount asc 3) Name asc
4) Name desc 5) Type then amount"
     SET ch ← getIntInput("Choice: ")
     IF budgetCount <= 1 THEN
     DISPLAY "Not enough records to sort."
     RETURN
     ENDIF
     IF ch = 1 THEN
     SORT budgets BY amount descending
     ELSE IF ch = 2 THEN
     SORT budgets BY amount ascending
     ELSE IF ch = 3 THEN
```

```
     SORT budgets BY name ascending
     ELSE IF ch = 4 THEN
     SORT budgets BY name descending
     ELSE IF ch = 5 THEN
     SORT budgets BY type ascending THEN amount descending
     ELSE
     DISPLAY "Invalid choice."
     RETURN
     ENDIF
    saveBudgetsToFile()
     DISPLAY "Sorted and saved."
END PROCEDURE



PROCEDURE showSummary()
     SET income ← computeTotalIncome()
     SET expense ← computeTotalExpense()
     SET balance ← income - expense
     SET incCount, expCount ← 0
     SET avgInc, avgExp ← 0
     SET maxInc, maxExp ← -1
     SET maxIncName, maxExpName ← ""

     FOR i FROM 0 TO budgetCount - 1 DO
```

```
    IF toLowerStr(budgets[i].type) = "income" THEN
        incCount ← incCount + 1
        avgInc ← avgInc + budgets[i].amount
        IF budgets[i].amount > maxInc THEN
            maxInc ← budgets[i].amount
             maxIncName ← budgets[i].name
        ENDIF
    ELSE
        expCount ← expCount + 1
        avgExp ← avgExp + budgets[i].amount
        IF budgets[i].amount > maxExp THEN
            maxExp ← budgets[i].amount
             maxExpName ← budgets[i].name
        ENDIF
    ENDIF
    ENDFOR

    IF incCount > 0 THEN avgInc ← avgInc / incCount ENDIF
    IF expCount > 0 THEN avgExp ← avgExp / expCount ENDIF

    SET pending ← 0
    FOR i FROM 0 TO requestCount - 1 DO
    IF toLowerStr(requests[i].status) = "pending" THEN
        pending ← pending + 1
```

```
     ENDIF

     ENDFOR


    drawHeaderBox("SUMMARY REPORT")

     DISPLAY total income, expense, balance, counts, averages,
largest income and expense, and pending requests

END PROCEDURE




PROCEDURE createAccount()

     IF accountCount >= MAX_ACCOUNTS THEN

     DISPLAY "Account storage full."

     RETURN

     ENDIF

     INPUT a.username

     WHILE a.username IS EMPTY DO

     DISPLAY "Username cannot be empty."

     INPUT a.username

     ENDWHILE

     SET exists ← FALSE

     FOR i FROM 0 TO accountCount - 1 DO

     IF toLowerStr(accounts[i].username) =
toLowerStr(a.username) THEN

          exists ← TRUE

     ENDIF
```

```
    ENDFOR
    IF exists THEN
    DISPLAY "Username exists. Aborting."
    RETURN
    ENDIF
    INPUT a.password
    INPUT a.role
    WHILE a.role NOT "parent" AND NOT "child" DO
    DISPLAY "Role must be Parent or Child."
    INPUT a.role
    ENDWHILE
    a.age ← getIntInput("Enter age: ")
    STORE a INTO accounts[accountCount]
    accountCount ← accountCount + 1
   saveAccountsToFile()

    IF toLowerStr(a.role) = "child" THEN
    IF profileCount >= MAX_PROFILES THEN
         DISPLAY "Profile storage full."
         RETURN
    ENDIF
    CREATE new profile p
    p.username ← a.username
    INPUT p.displayName
    IF p.displayName IS EMPTY THEN p.displayName ← a.username
```

```
ENDIF

     p.age ← a.age

     p.allowance ← 0

     STORE p INTO profiles[profileCount]

     profileCount ← profileCount + 1

        saveProfilesToFile()

     ENDIF

     DISPLAY "Account created."

END PROCEDURE



FUNCTION loginAccount() RETURNS Account*

     INPUT username, password

     FOR i FROM 0 TO accountCount - 1 DO

     IF accounts[i].username = username AND
accounts[i].password = password THEN

          RETURN reference to accounts[i]

     ENDIF

     ENDFOR

     DISPLAY "Invalid credentials."

     RETURN null

END FUNCTION



FUNCTION findProfileByUsername(uname) RETURNS Profile*
```

```
    FOR i FROM 0 TO profileCount - 1 DO
    IF toLowerStr(profiles[i].username) = toLowerStr(uname)
THEN
        RETURN reference to profiles[i]
    ENDIF
    ENDFOR
    RETURN null
END FUNCTION



PROCEDURE submitRequest(acc)
    IF acc.role ≠ "child" THEN
    DISPLAY "Only child accounts can submit requests."
    RETURN
    ENDIF
    IF requestCount >= MAX_REQUESTS THEN
    DISPLAY "Request storage full."
    RETURN
    ENDIF
    CREATE new Request r
    r.id ← nextRequestId
    nextRequestId ← nextRequestId + 1
    r.childUsername ← acc.username
    p ← findProfileByUsername(acc.username)
    IF p EXISTS THEN
```

```
     r.childName ← p.displayName

     ELSE

     INPUT r.childName

     IF EMPTY(r.childName) THEN r.childName ← acc.username
ENDIF

     ENDIF

     INPUT r.purpose

     r.amount ← getDoubleInput("Enter requested amount: ")

     r.status ← "Pending"

     STORE r INTO requests[requestCount]

     requestCount ← requestCount + 1

    saveRequestsToFile()

     DISPLAY "Request submitted."

END PROCEDURE



PROCEDURE parentProcessRequests()

     IF requestCount = 0 THEN

     DISPLAY "No requests."

     RETURN

     ENDIF

     DISPLAY table of all requests

     id ← getIntInput("Enter request ID to process (0 to
cancel): ")

     IF id = 0 THEN RETURN ENDIF
```

```
     FIND index of request by ID
     IF NOT FOUND THEN
     DISPLAY "Request not found."
     RETURN
     ENDIF
     IF status ≠ "pending" THEN
     DISPLAY "Already processed."
     RETURN
     ENDIF


     DISPLAY "1) Approve  2) Reject"
     ch ← getIntInput("Choice: ")
     IF ch = 1 THEN
        requests[idx].status ← "Approved"
     IF budgetCount < MAX_BUDGETS THEN
          CREATE new Budget b
          b.id ← nextBudgetId
            nextBudgetId ← nextBudgetId + 1
          b.name ← "Approved Request - " +
requests[idx].childName + " - " + requests[idx].purpose
          b.amount ← requests[idx].amount
          b.type ← "Expense"
          b.owner ← "system"
          STORE b INTO budgets[budgetCount]
```

```
            budgetCount ← budgetCount + 1
            saveBudgetsToFile()
          DISPLAY "Request approved and recorded as expense."
     ELSE
          DISPLAY "Budget storage full."
     ENDIF
     ELSE IF ch = 2 THEN
        requests[idx].status ← "Rejected"
     DISPLAY "Request rejected."
     ELSE
     DISPLAY "Invalid choice."
     ENDIF
    saveRequestsToFile()
END PROCEDURE



PROCEDURE manageProfiles()
     DISPLAY "Profile Manager Menu"
     ch ← getIntInput("Choice: ")
     IF ch = 0 THEN RETURN ENDIF
     IF ch = 1 THEN
     DISPLAY all profiles
     ELSE IF ch = 2 THEN
     uname ← getLineInputNonEmpty("Enter username: ")
```

```
     p ← findProfileByUsername(uname)

     IF p IS NULL THEN DISPLAY "Profile not found." RETURN
ENDIF

     a ← getDoubleInput("Enter new allowance: ")

     p.allowance ← a

        saveProfilesToFile()

     DISPLAY "Allowance updated."

     ELSE IF ch = 3 THEN

     IF profileCount >= MAX_PROFILES THEN DISPLAY "Profile
storage full." RETURN ENDIF

     CREATE new Profile p

     p.username ← getLineInputNonEmpty("Enter username (must
match account): ")

     VERIFY account exists

     INPUT p.displayName, p.age, p.allowance

     STORE p INTO profiles[profileCount]

     profileCount ← profileCount + 1

        saveProfilesToFile()

     DISPLAY "Profile added."

     ELSE IF ch = 4 THEN

     uname ← getLineInputNonEmpty("Enter username to remove: ")

     FIND index by username

     IF NOT FOUND THEN DISPLAY "Profile not found." RETURN
ENDIF

     SHIFT profiles left

     profileCount ← profileCount - 1
```

```
        saveProfilesToFile()
     DISPLAY "Profile removed."
     ELSE
     DISPLAY "Invalid choice."
     ENDIF
END PROCEDURE



PROCEDURE manageAccounts()
     DISPLAY "Account Manager Menu"
     ch ← getIntInput("Choice: ")
     IF ch = 0 THEN RETURN ENDIF
     IF ch = 1 THEN
     DISPLAY all accounts
     ELSE IF ch = 2 THEN
     uname ← getLineInputNonEmpty("Enter username to delete: ")
     FIND index by username
     IF NOT FOUND THEN DISPLAY "Account not found." RETURN
ENDIF
     SHIFT accounts left
     accountCount ← accountCount - 1
        saveAccountsToFile()
     DISPLAY "Account deleted."
     ELSE IF ch = 3 THEN
     uname ← getLineInputNonEmpty("Enter username to reset
```

```
password: ")

     FIND index by username

     IF NOT FOUND THEN DISPLAY "Account not found." RETURN
ENDIF

     INPUT new password

     UPDATE password

        saveAccountsToFile()

     DISPLAY "Password reset."

     ELSE

     DISPLAY "Invalid choice."

     ENDIF

END PROCEDURE



PROCEDURE parentMenu(acc)

     pending ← count of requests with status "pending"

     IF pending > 0 THEN DISPLAY number of pending requests
ENDIF

     REPEAT

     DISPLAY parent menu options

     ch ← getIntInput("Choice: ")

     CASE ch OF

          0: BREAK

          1: addBudgetInteractive(acc.username)

          2: showAllBudgetsFancy()
```

```
            3: editBudgetInteractive(acc.username, TRUE)
            4: deleteBudgetInteractive(acc.username, TRUE)
            5: searchBudgetsMenu()
            6: sortBudgetsMenu()
            7: showSummary()
            8: parentProcessRequests()
            9: manageProfiles()
            10: manageAccounts()
            OTHERWISE DISPLAY "Invalid choice."
      ENDCASE
      UNTIL ch = 0
END PROCEDURE



PROCEDURE childMenu(acc)
      isAdult ← (acc.age >= 18)
      REPEAT
      DISPLAY child menu options
      ch ← getIntInput("Choice: ")
      CASE ch OF
            0: BREAK
            1: showAllBudgetsFancy()
            2: submitRequest(acc)
            3: DISPLAY child's own requests
```

```
          4: IF isAdult THEN addBudgetInteractive(acc.username)

          5: IF isAdult THEN
editBudgetInteractive(acc.username, TRUE)

          6: IF isAdult THEN
deleteBudgetInteractive(acc.username, TRUE)

          7: showSummary()

          OTHERWISE DISPLAY "Invalid choice."

     ENDCASE

     UNTIL ch = 0

END PROCEDURE



PROCEDURE bootstrapDefaultsIfEmpty()

     IF accountCount = 0 THEN

     ADD default parent and child accounts

        saveAccountsToFile()

     ENDIF

     IF profileCount = 0 THEN

     ADD default child profile

        saveProfilesToFile()

     ENDIF

END PROCEDURE



PROCEDURE main()
```

```
loadAccountsFromFile()
loadProfilesFromFile()
loadBudgetsFromFile()
loadRequestsFromFile()
bootstrapDefaultsIfEmpty()
drawHeaderBox("FAMILY BUDGET MANAGER")


 REPEAT
 DISPLAY main menu options
 mainChoice ← getIntInput("Enter choice: ")
 IF mainChoice = 5 THEN
      DISPLAY "Goodbye."
      BREAK
 ELSE IF mainChoice = 1 THEN
      DISPLAY "Parent Login"
      acc ← loginAccount()
      IF acc IS parent THEN parentMenu(acc)
 ELSE IF mainChoice = 2 THEN
      DISPLAY "Child Login"
      acc ← loginAccount()
      IF acc IS child THEN childMenu(acc)
 ELSE IF mainChoice = 3 THEN
        createAccount()
 ELSE IF mainChoice = 4 THEN
```

```
            showSummary()

     ELSE

          DISPLAY "Invalid choice."

     ENDIF

     UNTIL mainChoice = 5

END PROCEDURE

END
```

Figure 2: Pseudo-code of the FBM system

Figure 2 presents the Family Budget Manager System's pseudocode, which describes the program's general logic and workflow. In order to effectively organize and manage financial data, it starts by defining constants and data structures like Budget, Request, Account, and Profile. In order to guarantee accurate record keeping and data persistence, the pseudocode also contains essential functions for data management, input validation, income and expense computation, and file handling. It also describes the different menu flows for kids and parents, with a focus on role-based access where kids submit requests for allowances and parents handle money. All things considered, the pseudocode offers a precise framework that directs the creation of the system's code and guarantees that every operation runs logically and methodically.

Data Dictionary

All of the variables, constants, and data types utilized in the Family Budget Manager System are organized in the Data Dictionary. It acts as a guide for comprehending the organization, processing, and storage of data within the program. The data dictionary makes it easier to debug or modify the code in the future and ensures clarity and consistency by listing the characteristics of each data element, such as its name, size, type, and purpose. Additionally, it makes it easier for programmers and readers to understand how each component of the system works with the others, especially when it comes to user accounts, budgets, and requests.

The Family Budget Manager System's comprehensive data dictionary is shown in the table below. It lists every crucial piece of data that the program uses, such as variables that manage calculations and summaries, structures that arrange user and budget data, and constants that specify storage limits. The data name, storage capacity, type of data (integer, string, or double), and a brief explanation of its role in the system are all specified in each row of the table. The table demonstrates how the program effectively

handles various data types to track income, expenses, user profiles, and allowance requests through this well-organized layout.

Table 1: Data DictionaryTable 1: Data Dictionary

Data Name	Size	Data Type	Description
MAX_BUDGETS	600	const int	Sets the limit for how many budget records can be saved.
MAX_REQUESTS	200	const int	Sets the limit for how many requests can be saved.
MAX_ACCOUNTS	200	const int	Sets the limit for how many accounts can be saved.
MAX_PROFILES	200	const int	Sets the limit for how many profiles can be saved.
Budget.id	4 bytes	int	Unique number given to each budget record.
Budget.name	variable	string	The name of the budget item.

Data Name	Size	Data Type	Description
Budget.amount	8 bytes	double	The total money for the budget item.
Budget.type	variable	string	Shows if it's "Income" or "Expense."
Budget.owner	variable	string	Username of who owns the budget.
Request.id	4 bytes	int	Unique number given to each request.
Request.childUsername	variable	string	Username of the child making the request.
Request.childName	variable	string	Name of the child.
Request.purpose	variable	string	Reason why the request was made.
Request.amount	8 bytes	double	Amount of money being requested.

Data Name	Size	Data Type	Description
Request.status	variable	string	Shows if the request is “Pending,” “Approved,” or “Rejected.”
Account.username	variable	string	Login name used for the account.
Account.password	variable	string	Password for logging in.
Account.role	variable	string	Identifies if the user is a “Parent” or “Child.”
Account.age	4 bytes	int	Age of the account owner.
Profile.username	variable	string	Username connected to the profile.
Profile.displayName	variable	string	The name shown in the system.
Profile.age	4 bytes	int	Age of the child.
Profile.allowance	8 bytes	double	The child’s current allowance or balance.

Data Name	Size	Data Type	Description
budgets[MAX_BUDGETS]	600 records	Budget[]	List that stores all budget records.
requests[MAX_REQUESTS]	200 records	Request[]	List that stores all request records.
accounts[MAX_ACCOUNTS]	200 records	Account[]	List that stores all user accounts.
profiles[MAX_PROFILES]	200 records	Profile[]	List that stores all user profiles.
budgetCount	4 bytes	int	Counts how many budgets are saved.
nextBudgetId	4 bytes	int	Gives the next available budget ID.
requestCount	4 bytes	int	Counts how many requests are saved.
nextRequestId	4 bytes	int	Gives the next available request ID.

Data Name	Size	Data Type	Description
accountCount	4 bytes	int	Counts how many accounts are saved.
profileCount	4 bytes	int	Counts how many profiles are saved.
toLowerStr(const string&)	N/A	string (function)	Converts text to lowercase.
isNumberString(string)	N/A	bool (function)	Checks if the input is a number.
getDoubleInput(prompt)	N/A	double (function)	Gets a number with decimals from the user.
getIntInput(prompt)	N/A	int (function)	Gets a whole number from the user.
getLineInputNonEmpty(prompt)	N/A	string (function)	Gets a line of text that isn't empty.
saveBudgetsToFile / loadBudgetsFromFile	N/A	file I/O functions	Saves or loads budget data from budgets.txt.

Data Name	Size	Data Type	Description
saveRequestsToFile / loadRequestsFromFile	N/A	file I/O functions	Saves or loads requests from requests.txt.
saveAccountsToFile / loadAccountsFromFile	N/A	file I/O functions	Saves or loads accounts from accounts.txt.
saveProfilesToFile / loadProfilesFromFile	N/A	file I/O functions	Saves or loads profiles from profiles.txt.
budgets.txt	varies	text file	Stores budget information.
requests.txt	varies	text file	Stores request information.
accounts.txt	varies	text file	Stores account information.
profiles.txt	varies	text file	Stores profile information.
computeTotalIncome()	N/A	double (function)	Calculates total income.
computeTotalExpense()	N/A	double (function)	Calculates total expenses.

Data Name	Size	Data Type	Description
drawHeaderBox(title)	N/A	void (function)	Displays a title box on the screen.
showAllBudgetsFancy()	N/A	void (function)	Shows all budget records in a table.
addBudgetInteractive(owner)	N/A	void (function)	Lets the user add a new budget.
editBudgetInteractive(currentUser,isParentOrAdult)	N/A	void (function)	Lets the user edit an existing budget.
deleteBudgetInteractive(currentUser,isParentOrAdult)	N/A	void (function)	Lets the user delete a budget record.
searchBudgetsMenu()	N/A	void (function)	Lets the user search budgets.
sortBudgetsMenu()	N/A	void (function)	Lets the user sort budgets.
showSummary()	N/A	void (function)	Shows income, expenses, balance, and totals.

Data Name	Size	Data Type	Description
createAccount()	N/A	void (function)	Creates a new account (and profile if child).
loginAccount()	N/A	Account* (function)	Checks login and returns account details.
findProfileByUsername(uname)	N/A	Profile* (function)	Finds a profile using a username.
submitRequest(acc)	N/A	void (function)	Let a child send a money request.
parentProcessRequests()	N/A	void (function)	Let the parent approve or reject requests.
manageProfiles()	N/A	void (function)	Lets the user view, edit, or delete profiles.
manageAccounts()	N/A	void (function)	Lets the user view, reset, or delete accounts.
parentMenu(acc)	N/A	void (function)	Shows menu options for the parent user.

Data Name	Size	Data Type	Description
childMenu(acc)	N/A	void (function)	Shows menu options for the child user.
bootstrapDefaultsIfEmpty()	N/A	void (function)	Creates default accounts if no data exists.
main()	N/A	int (function)	Starts the whole program.
income	8 bytes	double	Stores total income value.
expense	8 bytes	double	Stores total expenses.
balance	8 bytes	double	Difference between income and expenses.
incCount, expCount	4 bytes each	int	Counts income and expense entries.
avgInc, avgExp	8 bytes each	double	Average income and expenses.
maxInc, maxExp	8 bytes each	double	Highest income and expense recorded.

Data Name	Size	Data Type	Description
maxIncName, maxExpName	variable	string	Names of the highest income and expense.
pending	4 bytes	int	Number of requests still pending.

Code

The primary program code created for the Family Budget Manager System is shown in the section that follows. It describes the fundamental systems and features that allow role-based access control, budget tracking, and user account management.

```
// for Funcs.h
#ifndef FUNCS_H
#define FUNCS_H
#include <string>
#include "Structs.h"
using namespace std;

string toLowerStr(const string &s);
bool isNumberString(const string &s);
double getDoubleInput(const string &prompt);
int getIntInput(const string &prompt);
string getLineInputNonEmpty(const string &prompt);
void saveBudgetsToFile();
void loadBudgetsFromFile();
void saveRequestsToFile();
void loadRequestsFromFile();
void saveAccountsToFile();
void loadAccountsFromFile();
void saveProfilesToFile();
```

```
void loadProfilesFromFile();
double computeTotalIncome();
double computeTotalExpense();
void drawHeaderBox(const string &title);
void showAllBudgetsFancy();
void addBudgetInteractive(const string &owner);
int findBudgetIndexById(int id);
void editBudgetInteractive(const string &currentUser, bool
isParentOrAdult);
void deleteBudgetInteractive(const string &currentUser, bool
isParentOrAdult);
void searchBudgetsMenu();
void sortBudgetsMenu();
void showSummary();
void createAccount();
Account* loginAccount();
Profile* findProfileByUsername(const string &uname);
void submitRequest(const Account &acc);
void parentProcessRequests();
void manageProfiles();
void manageAccounts();
void parentMenu(Account &acc);
void childMenu(Account &acc);
void bootstrapDefaultsIfEmpty();

#endif

// for Structs.h
#ifndef STRUCTS_H
#define STRUCTS_H
#include <string>
#include "Funcs.h"
using namespace std;

const int MAX_BUDGETS = 600;
const int MAX_REQUESTS = 200;
const int MAX_ACCOUNTS = 200;
const int MAX_PROFILES = 200;

struct Budget {
    int id;
    string name;
```

```
    double amount;
    string type;
    string owner;
};

struct Request {
    int id;
    string childUsername;
    string childName;
    string purpose;
    double amount;
    string status;
};

struct Account {
    string username;
    string password;
    string role;
    int age;
};

struct Profile {
    string username;
    string displayName;
    int age;
    double allowance;
};
#endif

// for globals.h
#ifndef GLOBALS_H
#define GLOBALS_H

#include "Structs.h"

extern Budget budgets[600];
extern int budgetCount;
extern int nextBudgetId;

extern Request requests[200];
extern int requestCount;
extern int nextRequestId;
```

```
extern Account accounts[200];
extern int accountCount;

extern Profile profiles[200];
extern int profileCount;

#endif

(view Appendix A for the whole code)
```

Figure 3. Header files of the Family Budget Manager System

Three (3) header files are used by the Family Budget Manager System to facilitate organized and modular program development. The primary data structures, such as Account, Budget, and Request, which hold crucial user data, budget records, and fund requests, are defined in the StructFBT.h file. The function prototypes that manage essential tasks like file handling, data processing, budget management, and user login are contained in the Funcs.h file. The program preserves clarity and minimizes redundancy by keeping these declarations apart from their implementations. To guarantee consistency between various modules, a global header file may also contain shared variables and constants. Using header files enhances readability, maintainability, and scalability overall, encouraging effective and organized coding techniques across the Family Budget Manager System.

Results and Discussion

In this part, the developers will outline how the program works, with what functionality it is a successful system to manage and organize family budgets.

The first function of the program will ask the user if they are to log in as a parent or a child, create accounts if they were to add new users, and view a summary of the budgets, as the screenshot below shows it.

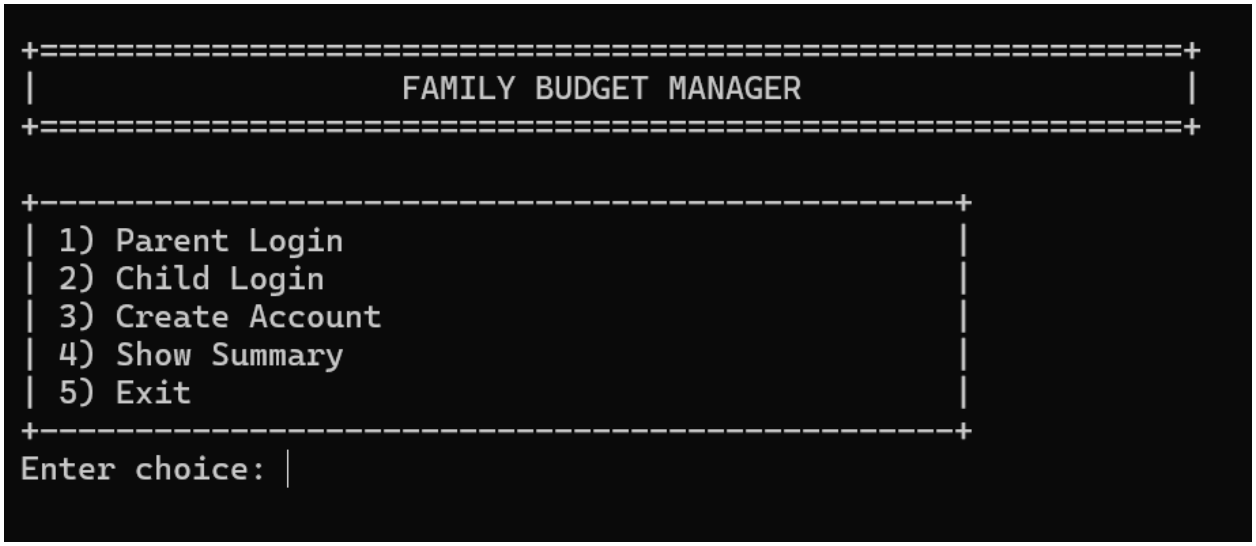

Figure 1: Login Interface of the FBM system

The Family Budget Manager (FBM) System starts with an easy-to-use login interface that is intended to authenticate users before granting them access to the system, as illustrated in Figure 1. This interface asks users to create an account, view a summary of financial data, or log in as a parent or child. By requiring legitimate credentials prior to entry, its simplicity guarantees accessibility even for users who are not familiar with technical applications while upholding security and data integrity.

By giving the user the option to log in, create an account, or view a financial summary, the interface also acts as a gateway to the system's essential features. This design promotes role separation and reinforces accountability within the family structure. The system protects the security and integrity of financial records by requiring login credentials, making it impossible for unauthorized people to access or alter private information.

The login interface's effectiveness demonstrates the developers' focus on data security and usability. Its design maintains distinct functional boundaries between different user types while promoting accessibility. The system's balance between ease of use and security is reflected in the login procedure, which guarantees that each subsequent action is connected to a verified identity.

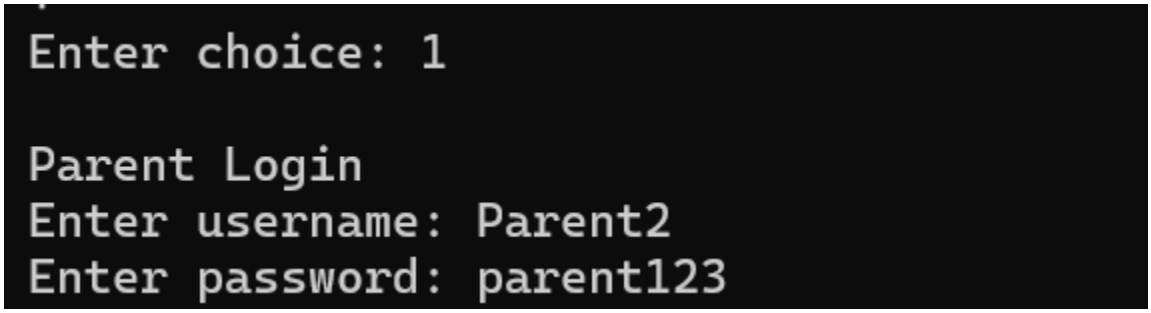

Figure 2: Logging in as a parent

The Family Budget Manager System's role-based access control is demonstrated by the parent login interface, as seen in Figure 2. Before the parent can access the main features of the system, they must enter legitimate credentials. The confidentiality of family records is preserved by this authentication, which guarantees that only authorized users can handle or view financial data.

Additionally, the system's emphasis on responsibility and accuracy is reflected in the login process. The system prevents unauthorized changes and guarantees that each user only interacts with features relevant to their role by separating parent and child logins. The integrity of stored data is strengthened and accountability is supported by this structural design.

All things considered, this feature demonstrates how the system's designers incorporated security and ease of use. In addition to safeguarding private information, the parent login is the initial stage of the process that links each financial procedure to a verified user.

Figure 3: Parent Menu of the FBM

The parent menu is the focal point of the FBM System's financial management, as seen in Figure 3. It gives parents total control over all aspects of household budgeting, including the ability to add, view, edit, and remove budget records. By providing parents with direct access to important functions, this arrangement encourages a methodical approach to financial management.

Even users with little technical expertise can navigate the menu effectively thanks to its design, which makes sure that every function is clearly labeled. The developers' emphasis on developing a user-friendly interface that is consistent with actual financial management procedures is evident in this simplicity.

The system efficiently combines several processes into a single, cohesive dashboard through the parent menu. It strengthens the program's usefulness and usability by increasing workflow efficiency and lowering the possibility of misunderstandings or improper data handling.

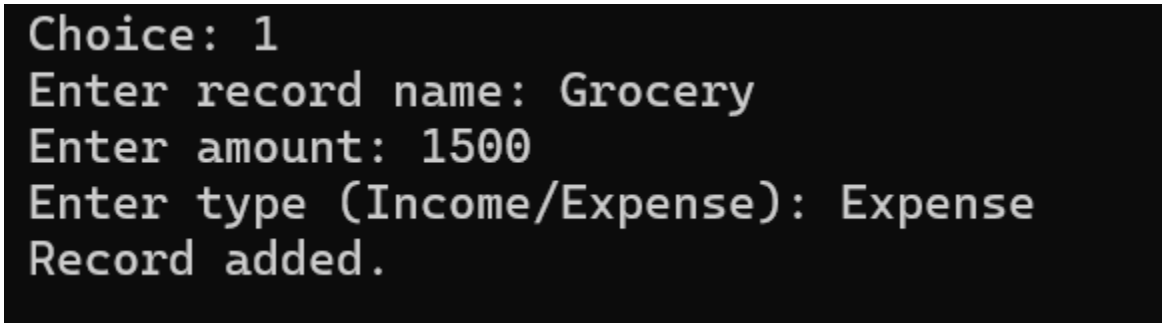

Figure 4: Adding a Budget record as a Parent

The parent can add a budget record by entering the necessary details, such as the record name, amount, and type (income or expense), as illustrated in Figure 4. Before confirming the addition of the record, the system walks the user through each input field to ensure accuracy.

The system's emphasis on accurate data entry is demonstrated by this procedure. The software avoids typical mistakes and guarantees that each financial transaction is accurately classified and documented by asking for particular inputs. Financial tracking on a daily and monthly basis is made easier with this feature.

The addition function demonstrates the developers' focus on data consistency and interactivity. It helps parents keep accurate and current records, which raises household financial awareness.

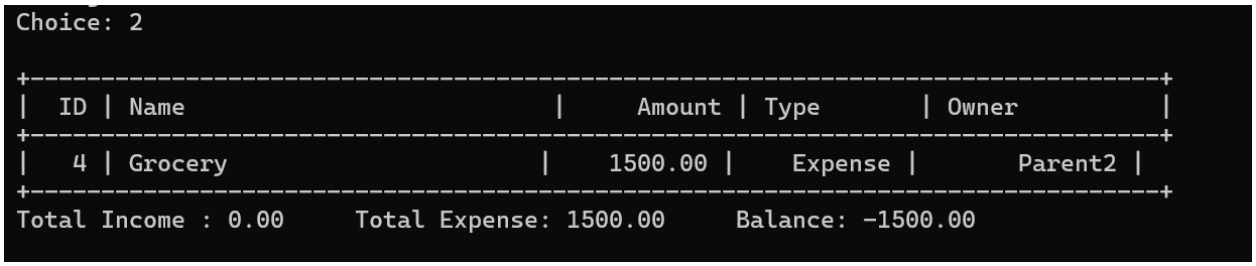

Figure 5: Table that shows the budget that was added

The system arranges all financial entries into a structured table format, as seen in Figure 5. A clear picture of the family's financial activities is provided by each record, which contains information like ID, name, amount, type, and owner.

Budget monitoring is made easier and more readable with this visual presentation. Parents can make well-informed decisions about savings or spending adjustments by simply identifying trends in their income and expenses.

Overall, the developers' intention to combine accessibility and organization is reflected in the table design. It guarantees that data is transparent and traceable and enables users to swiftly assess their financial situation.

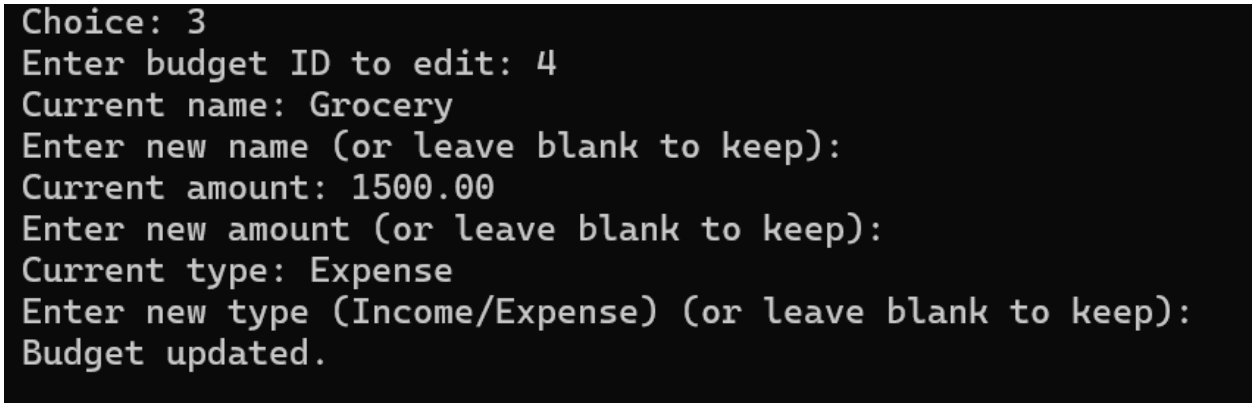

Figure 6: Editing process of the budget based on its ID

As shown in Figure 6, the editing function enables parents to modify existing budget records by entering the corresponding ID. The system offers flexibility for updates by asking for optional modifications to the record name, amount, or type.

This feature demonstrates how the system can adjust to actual financial fluctuations. Users can maintain their financial data up to date without having to erase or recreate records by making simple changes.

The FBM System's goal of upholding precise and dynamic financial management is strengthened by the edit feature. It guarantees that all modifications are instantly reflected in the data that is stored, promoting continuous accuracy.

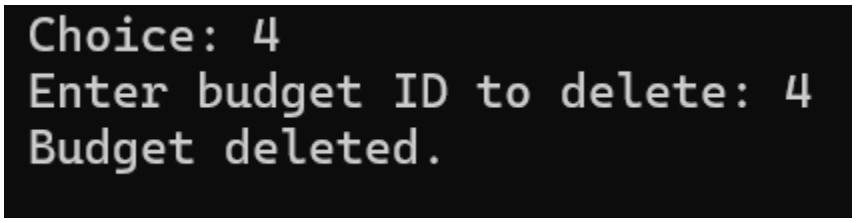

Figure 7: Deletion of the Budget based on its ID

By entering the record's ID, parents can delete unwanted or out-of-date budget records, as illustrated in Figure 7. After that, the entry is permanently removed by the system, leaving only pertinent financial information.

This procedure aids in keeping data storage neat and orderly. The system improves clarity and lessens clutter when examining financial data by eliminating superfluous entries.

Efficiency and control within the application are demonstrated by the delete function. It enables users to responsibly manage records while preserving the financial database's integrity.

Figure 8: Searching Interface

The search function enables users to find particular financial records by name, type, owner, or amount range, as illustrated in Figure 8. This speeds up and improves the efficiency of data retrieval, especially when dealing with a lot of entries.

The developers' attention to detail and convenience is evident in the inclusion of several search parameters. Depending on the user's requirements, it enables both general and targeted searches.

All things considered, this feature improves the FBM System's usability by facilitating rapid information access and increasing the effectiveness of managing and reviewing financial records.

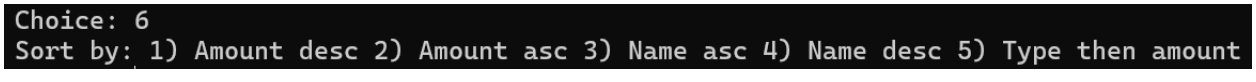

Figure 9: Sorting Interface

The sorting feature allows users to organize financial records according to various criteria like amount, name, or type, as illustrated in Figure 9. Depending on their preferences, users can arrange data in either ascending or descending order.

This function offers a straightforward but efficient way to examine financial trends. It aids parents in determining which household expenses are the highest or which sources of income make up the largest portion of the budget.

The FBM System's analytical capabilities were enhanced by the developers' integration of sorting options. This feature strengthens financial management transparency and improves decision-making.

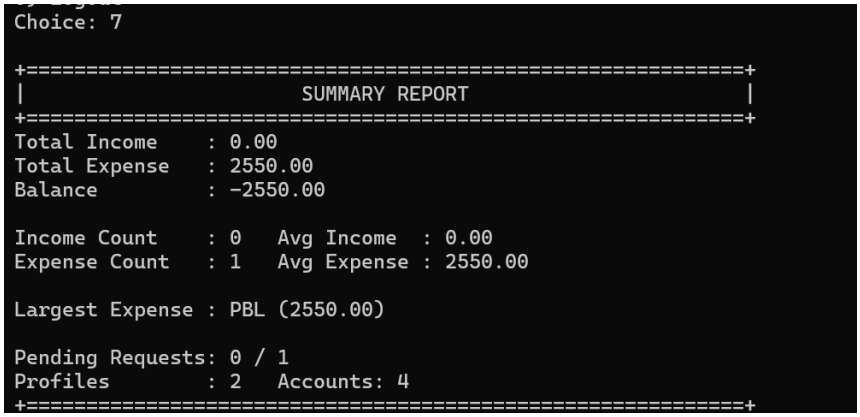

Figure 10: Summary of the budget, requests, profiles, and accounts

The summary feature offers a thorough overview of the family's financial situation, including total income, total expenses, balance, and pending requests, as seen in Figure 10. It compiles financial information into a single, easily readable report.

This feature shows how financial planning and literacy are supported by the system. Users can determine whether their current spending is in line with their financial objectives and make necessary adjustments.

The system's overarching goal of streamlining financial analysis and promoting responsible family budgeting is embodied in the summary report.

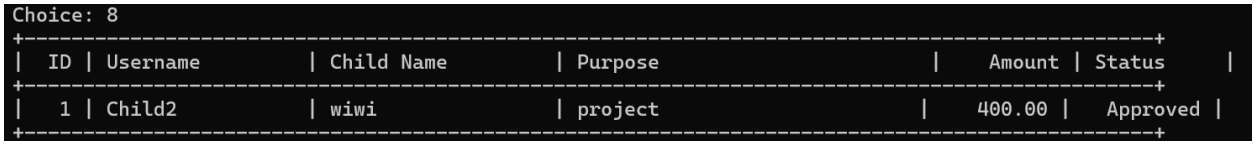

Figure 11: Table of budget requests sent by the child

The allowance request management feature enables parents to view, accept, or reject requests made by their children, as illustrated in Figures 11 and 11a. Requests that are approved are automatically saved in the requests.txt file as expenses.

This feature facilitates open communication between parents and kids. Because each request and approval is recorded for recordkeeping, it also establishes accountability.

The developers' intention to teach financial responsibility is reflected in the request management system. It offers a useful strategy for handling allowances while upholding justice and openness.

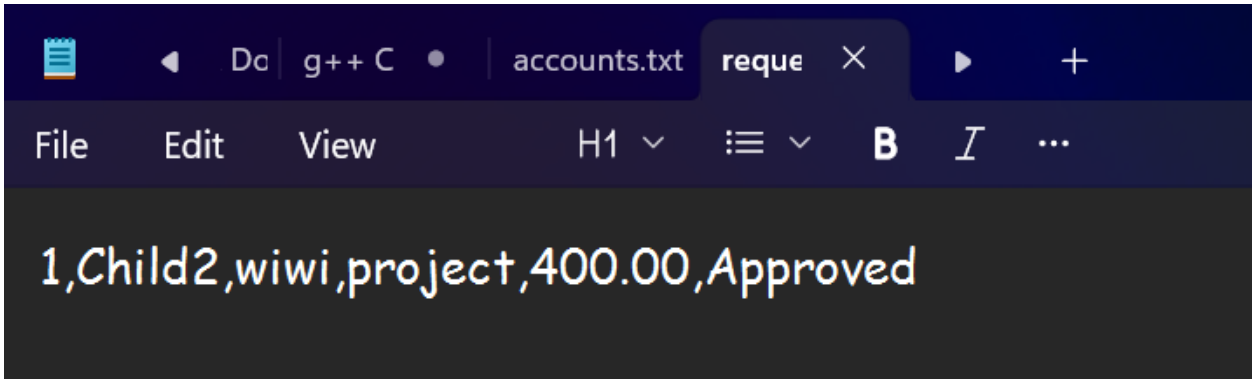

Figure 11a: The said request was uploaded in requests.txt as a file

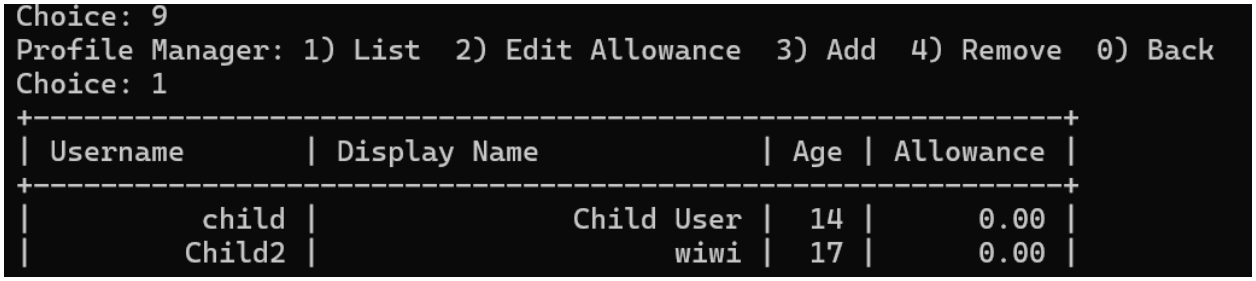

Figure 12: Profile manager Interface and list of profiles of the children

Parents can view, edit, add, or remove child profiles using the profile manager, as illustrated in Figure 12. Additionally, they have the ability to manage other profile details and modify allowances.

This feature promotes individualized and well-organized recordkeeping by guaranteeing that every child's financial activity is monitored under their unique profile.

All things considered, the profile manager aids in effective account management by guaranteeing that every user functions within their proper roles and permissions.
When choosing 10, they can manage, reset a password for a user, and remove an account.

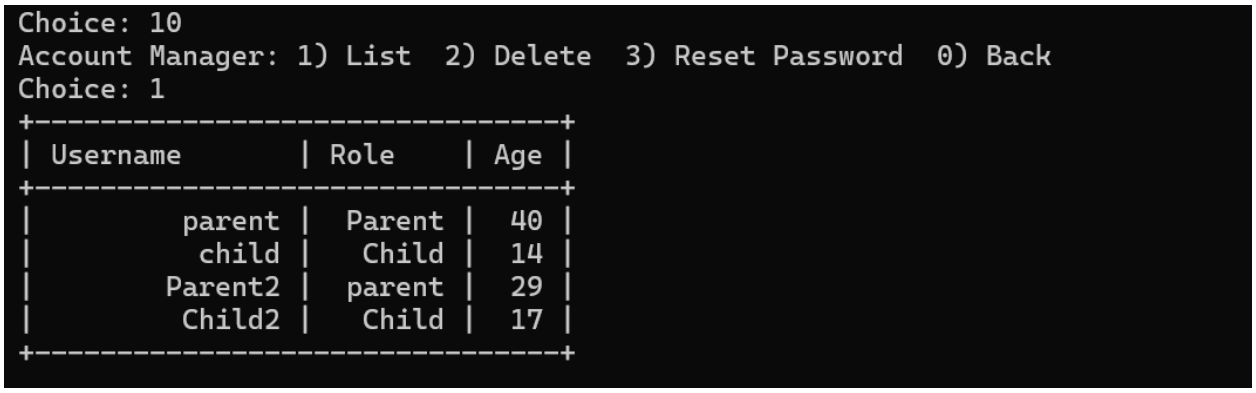

Figure 13: Account manager Interface and list of accounts created

That's all for the parent menu

Now, for the child menu, there are two types: a young adult menu and a child menu

For the young adult menu, here are what they can do as shown below

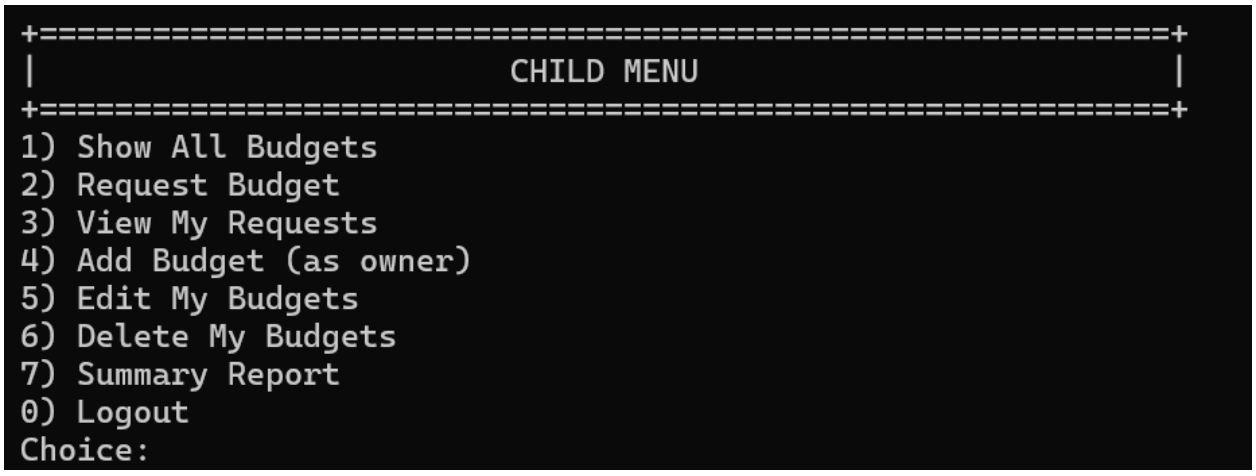

Figure 14: Young adult Menu (Age 18)

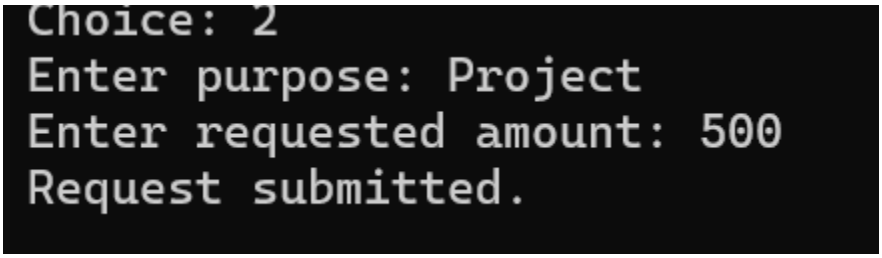

Figure 14a: User requesting a budget

Users who are at least 18 years old have restricted access to the young adult menu, as seen in Figures 14 and 14a. Although they are unable to change system settings or profiles, these users are able to request budgets, monitor financial activity, and view approval results.

This arrangement preserves parental supervision while promoting independence. It helps young adults form sound financial habits by striking a balance between accountability and autonomy.

The system's educational approach, which teaches financial management through controlled participation and real-world simulation, is reflected in the design of this menu.

Figure 14b: Viewing of said Request

Young adults can see whether their requests were accepted or denied, as seen in Figure 14b. Clear communication and trust between parents and children are fostered by this transparency.

By demonstrating how financial decisions are made and assessed, the display of request outcomes further strengthens learning.

This feature demonstrates how the FBM System strengthens its educational goal by incorporating accountability and transparency into each user interaction.

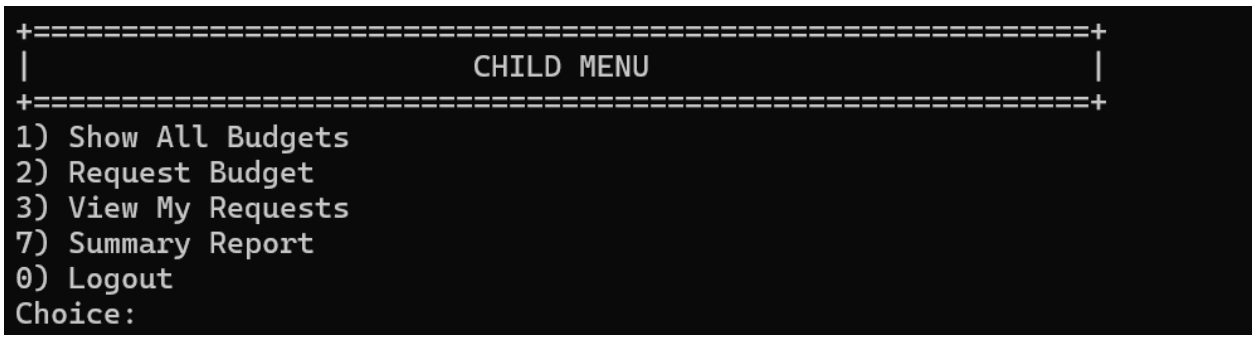

Figure 15: Child Menu

The child menu offers a streamlined interface intended for users under the age of eighteen, as seen in Figure 15. They can check overall summaries, send allowance requests, and view family budgets.

This menu's limitations prevent kids from editing or deleting records, protecting data integrity and enabling active learning through observation.

Overall, this interface supports the system's primary objective of teaching budgeting techniques in a safe setting by encouraging early financial awareness and responsible behavior.

These results demonstrate that the Family Budget Manager System successfully achieves its goal of encouraging openness and financial literacy in the home. The system helps parents keep control over the family's finances while teaching kids responsible money management by clearly defining user roles and automating record management.

Conclusion

In summary, the Family Budget Manager System effectively achieved its goals of assisting families in keeping track of, managing, and recording their financial transactions on a single platform. Through controlled access and request features, the system taught kids responsible budgeting practices while enabling parents to keep organized records, monitor income and expenses, and respond to allowance requests. During testing, all of the main features—such as budget management, summary reports, and login—performed well. It is advised that the system be improved in the future by adding a graphical user interface for better usability, integrating a database for better data security and storage, and adding extra features like expense categories or automated reports. The system would be more effective, user-friendly, and flexible for regular family use with these enhancements.

References

Consumer Financial Protection Bureau. (2023). Managing your finances. https://www.consumer.ftc.gov/

Nakagawa, H., & Shigekawa, J. (2024). *Strategy and belief for successful family budget management: A case study of family account books for over 50 years*. *International Journal of Home Economics.* *https://www.ifhe.org/fileadmin/user_upload/IJHE-Vol-17-Iss-1-A5-Nakagawa.pdf*

Lusardi, A. (2019). Financial literacy and the need for financial education: Evidence and implications. Journal of Financial Literacy and Wellbeing, 1(1), 1-8.

Appendices

Appendix A: FBM raw code

```
#include <iostream>
#include <string>
#include <fstream>
#include <iomanip>
#include <algorithm>
using namespace std;

const int MAX_BUDGETS = 600;
const int MAX_REQUESTS = 200;
const int MAX_ACCOUNTS = 200;
const int MAX_PROFILES = 200;

struct Budget {
    int id;
    string name;
    double amount;
    string type;
    string owner;
};

struct Request {
    int id;
    string childUsername;
    string childName;
    string purpose;
    double amount;
    string status;
};

struct Account {
    string username;
    string password;
    string role;
    int age;
};

struct Profile {
    string username;
    string displayName;
    int age;
```

```
    double allowance;
};

Budget budgets[MAX_BUDGETS];
int budgetCount = 0;
int nextBudgetId = 1;

Request requests[MAX_REQUESTS];
int requestCount = 0;
int nextRequestId = 1;

Account accounts[MAX_ACCOUNTS];
int accountCount = 0;

Profile profiles[MAX_PROFILES];
int profileCount = 0;

string toLowerStr(const string &s) {
    string r = s;
    for (size_t i = 0; i < r.size(); ++i) r[i] =
static_cast<char>(tolower(r[i]));
    return r;
}

bool isNumberString(const string &s) {
    bool hasDigit = false;
    bool hasDot = false;
    for (char c : s) {
        if (isdigit(static_cast<unsigned char>(c))) hasDigit = true;
        else if (c == '.' && !hasDot) hasDot = true;
        else return false;
    }
    return hasDigit;
}

double getDoubleInput(const string &prompt) {
    while (true) {
        cout << prompt;
        string line;
        getline(cin, line);
        if (line.empty()) continue;
        if (isNumberString(line)) {
```

```
            try {
                double val = stod(line);
                return val;
            } catch (...) {
                cout << "Invalid number format. Try again.\n";
            }
        } else {
            cout << "Invalid number. Try again.\n";
        }
    }
}

int getIntInput(const string &prompt) {
    while (true) {
        cout << prompt;
        string line;
        getline(cin, line);
        if (line.empty()) continue;
        bool ok = true;
        size_t i = 0;
        if (line[0] == '-') i = 1;
        for (; i < line.size(); ++i) if (!isdigit(static_cast<unsigned
char>(line[i]))) { ok = false; break; }
        if (ok) {
            try {
                int v = stoi(line);
                return v;
            } catch (...) {
                cout << "Invalid integer. Try again.\n";
            }
        } else {
            cout << "Invalid integer. Try again.\n";
        }
    }
}

string getLineInputNonEmpty(const string &prompt) {
    while (true) {
        cout << prompt;
        string line;
        getline(cin, line);
        if (!line.empty()) return line;
```

```
        cout << "Input cannot be empty. Try again.\n";
    }
}

void saveBudgetsToFile() {
    ofstream f("budgets.txt");
    if (!f) return;
    for (int i = 0; i < budgetCount; ++i) {
        f << budgets[i].id << "," << budgets[i].name << "," << fixed <<
setprecision(2) << budgets[i].amount << "," << budgets[i].type << "," <<
budgets[i].owner << "\n";
    }
    f.close();
}

void loadBudgetsFromFile() {
    ifstream f("budgets.txt");
    if (!f) return;
    budgetCount = 0;
    nextBudgetId = 1;
    string line;
    while (getline(f, line) && budgetCount < MAX_BUDGETS) {
        if (line.empty()) continue;
        size_t p1 = line.find(',');
        if (p1 == string::npos) continue;
        size_t p2 = line.find(',', p1 + 1);
        if (p2 == string::npos) continue;
        size_t p3 = line.find(',', p2 + 1);
        if (p3 == string::npos) continue;
        size_t p4 = line.find(',', p3 + 1);
        if (p4 == string::npos) continue;
        Budget b;
        try {
            b.id = stoi(line.substr(0, p1));
            b.name = line.substr(p1 + 1, p2 - p1 - 1);
            b.amount = stod(line.substr(p2 + 1, p3 - p2 - 1));
            b.type = line.substr(p3 + 1, p4 - p3 - 1);
            b.owner = line.substr(p4 + 1);
            budgets[budgetCount++] = b;
            if (b.id >= nextBudgetId) nextBudgetId = b.id + 1;
        } catch (...) {
            continue;
```

```
        }
    }
    f.close();
}

void saveRequestsToFile() {
    ofstream f("requests.txt");
    if (!f) return;
    for (int i = 0; i < requestCount; ++i) {
        f << requests[i].id << "," << requests[i].childUsername << "," <<
requests[i].childName << "," << requests[i].purpose << "," << fixed <<
setprecision(2) << requests[i].amount << "," << requests[i].status << "\n";
    }
    f.close();
}

void loadRequestsFromFile() {
    ifstream f("requests.txt");
    if (!f) return;
    requestCount = 0;
    nextRequestId = 1;
    string line;
    while (getline(f, line) && requestCount < MAX_REQUESTS) {
        if (line.empty()) continue;
        size_t p1 = line.find(',');
        size_t p2 = (p1==string::npos)?string::npos:line.find(',', p1 + 1);
        size_t p3 = (p2==string::npos)?string::npos:line.find(',', p2 + 1);
        size_t p4 = (p3==string::npos)?string::npos:line.find(',', p3 + 1);
        size_t p5 = (p4==string::npos)?string::npos:line.find(',', p4 + 1);
        if (p1==string::npos || p2==string::npos || p3==string::npos ||
p4==string::npos || p5==string::npos) continue;
        Request r;
        try {
            r.id = stoi(line.substr(0, p1));
            r.childUsername = line.substr(p1 + 1, p2 - p1 - 1);
            r.childName = line.substr(p2 + 1, p3 - p2 - 1);
            r.purpose = line.substr(p3 + 1, p4 - p3 - 1);
            r.amount = stod(line.substr(p4 + 1, p5 - p4 - 1));
            r.status = line.substr(p5 + 1);
            requests[requestCount++] = r;
            if (r.id >= nextRequestId) nextRequestId = r.id + 1;
        } catch (...) {
```

```
            continue;
        }
    }
    f.close();
}

void saveAccountsToFile() {
    ofstream f("accounts.txt");
    if (!f) return;
    for (int i = 0; i < accountCount; ++i) {
        f << accounts[i].username << "," << accounts[i].password << "," <<
accounts[i].role << "," << accounts[i].age << "\n";
    }
    f.close();
}

void loadAccountsFromFile() {
    ifstream f("accounts.txt");
    if (!f) return;
    accountCount = 0;
    string line;
    while (getline(f, line) && accountCount < MAX_ACCOUNTS) {
        if (line.empty()) continue;
        size_t p1 = line.find(',');
        size_t p2 = (p1==string::npos)?string::npos:line.find(',', p1 + 1);
        size_t p3 = (p2==string::npos)?string::npos:line.find(',', p2 + 1);
        if (p1==string::npos || p2==string::npos || p3==string::npos)
continue;
        Account a;
        try {
            a.username = line.substr(0, p1);
            a.password = line.substr(p1 + 1, p2 - p1 - 1);
            a.role = line.substr(p2 + 1, p3 - p2 - 1);
            a.age = stoi(line.substr(p3 + 1));
            accounts[accountCount++] = a;
        } catch (...) {
            continue;
        }
    }
    f.close();
}
```

```
void saveProfilesToFile() {
    ofstream f("profiles.txt");
    if (!f) return;
    for (int i = 0; i < profileCount; ++i) {
        f << profiles[i].username << "," << profiles[i].displayName << ","
<< profiles[i].age << "," << fixed << setprecision(2) <<
profiles[i].allowance << "\n";
    }
    f.close();
}

void loadProfilesFromFile() {
    ifstream f("profiles.txt");
    if (!f) return;
    profileCount = 0;
    string line;
    while (getline(f, line) && profileCount < MAX_PROFILES) {
        if (line.empty()) continue;
        size_t p1 = line.find(',');
        size_t p2 = (p1==string::npos)?string::npos:line.find(',', p1 + 1);
        size_t p3 = (p2==string::npos)?string::npos:line.find(',', p2 + 1);
        if (p1==string::npos || p2==string::npos || p3==string::npos)
continue;
        Profile p;
        try {
            p.username = line.substr(0, p1);
            p.displayName = line.substr(p1 + 1, p2 - p1 - 1);
            p.age = stoi(line.substr(p2 + 1, p3 - p2 - 1));
            p.allowance = stod(line.substr(p3 + 1));
            profiles[profileCount++] = p;
        } catch (...) {
            continue;
        }
    }
    f.close();
}

double computeTotalIncome() {
    double s = 0.0;
    for (int i = 0; i < budgetCount; ++i) if (toLowerStr(budgets[i].type)
== "income") s += budgets[i].amount;
    return s;
```

```
}

double computeTotalExpense() {
    double s = 0.0;
    for (int i = 0; i < budgetCount; ++i) if (toLowerStr(budgets[i].type)
== "expense") s += budgets[i].amount;
    return s;
}

void drawHeaderBox(const string &title) {
    int width = 60;
    cout << "\n+" << string(width, '=') << "+\n";
    int padding = (width - (int)title.size()) / 2;
    cout << "|" << string(padding, ' ') << title << string(width - padding
- (int)title.size(), ' ') << "|\n";
    cout << "+" << string(width, '=') << "+\n";
}

void showAllBudgetsFancy() {
    if (budgetCount == 0) {
        cout << "\nNo budget records available.\n";
        return;
    }
    cout <<
"\n+----------------------------------------------------------------------
---------+\n";
    cout << "|  ID | Name                           |     Amount | Type
| Owner          |\n";
    cout <<
"+----------------------------------------------------------------------
-------+\n";
    for (int i = 0; i < budgetCount; ++i) {
        cout << "| " << setw(3) << budgets[i].id << " | " << left <<
setw(28) << budgets[i].name << right << " | "
             << setw(10) << fixed << setprecision(2) << budgets[i].amount
<< " | " << setw(10) << budgets[i].type << " | " << setw(13) <<
budgets[i].owner << " |\n";
    }
    cout <<
"+----------------------------------------------------------------------
-------+\n";
    cout << "Total Income : " << fixed << setprecision(2) <<
```

```
computeTotalIncome() << "     Total Expense: " << fixed << setprecision(2)
<< computeTotalExpense() << "     Balance: " << fixed << setprecision(2) <<
(computeTotalIncome() - computeTotalExpense()) << "\n";
}

void addBudgetInteractive(const string &owner) {
    if (budgetCount >= MAX_BUDGETS) { cout << "Budget storage full.\n";
return; }
    Budget b;
    b.id = nextBudgetId++;
    cout << "Enter record name: ";
    getline(cin, b.name);
    b.amount = getDoubleInput("Enter amount: ");
    cout << "Enter type (Income/Expense): ";
    string t;
    getline(cin, t);
    while (toLowerStr(t) != "income" && toLowerStr(t) != "expense") {
        cout << "Type must be Income or Expense. Enter again: ";
        getline(cin, t);
    }
    b.type = t;
    b.owner = owner;
    budgets[budgetCount++] = b;
    saveBudgetsToFile();
    cout << "Record added.\n";
}

int findBudgetIndexById(int id) {
    for (int i = 0; i < budgetCount; ++i) if (budgets[i].id == id) return
i;
    return -1;
}

void editBudgetInteractive(const string &currentUser, bool isParentOrAdult)
{
    if (budgetCount == 0) { cout << "No budgets to edit.\n"; return; }
    int id = getIntInput("Enter budget ID to edit: ");
    int idx = findBudgetIndexById(id);
    if (idx < 0) { cout << "Budget not found.\n"; return; }
    if (!isParentOrAdult && toLowerStr(budgets[idx].owner) !=
toLowerStr(currentUser)) { cout << "You can only edit your own records.\n";
return; }
```

```
    cout << "Current name: " << budgets[idx].name << "\n";
    cout << "Enter new name (or leave blank to keep): ";
    string temp;
    getline(cin, temp);
    if (!temp.empty()) budgets[idx].name = temp;
    cout << "Current amount: " << budgets[idx].amount << "\n";
    cout << "Enter new amount (or leave blank to keep): ";
    string amtLine;
    getline(cin, amtLine);
    if (!amtLine.empty()) {
        while (!isNumberString(amtLine)) { cout << "Invalid number. Enter
again: "; getline(cin, amtLine); }
        budgets[idx].amount = stod(amtLine);
    }
    cout << "Current type: " << budgets[idx].type << "\n";
    cout << "Enter new type (Income/Expense) (or leave blank to keep): ";
    string newType;
    getline(cin, newType);
    if (!newType.empty()) {
        while (toLowerStr(newType) != "income" && toLowerStr(newType) !=
"expense") { cout << "Type must be Income or Expense. Enter again: ";
getline(cin, newType); }
        budgets[idx].type = newType;
    }
    saveBudgetsToFile();
    cout << "Budget updated.\n";
}

void deleteBudgetInteractive(const string &currentUser, bool
isParentOrAdult) {
    if (budgetCount == 0) { cout << "No budgets to delete.\n"; return; }
    int id = getIntInput("Enter budget ID to delete: ");
    int idx = findBudgetIndexById(id);
    if (idx < 0) { cout << "Budget not found.\n"; return; }
    if (!isParentOrAdult && toLowerStr(budgets[idx].owner) !=
toLowerStr(currentUser)) { cout << "You can only delete your own
records.\n"; return; }
    for (int i = idx; i < budgetCount - 1; ++i) budgets[i] = budgets[i +
1];
    --budgetCount;
    saveBudgetsToFile();
    cout << "Budget deleted.\n";
```

```
}

void searchBudgetsMenu() {
    cout << "Search by: 1) Name 2) Type 3) Owner 4) Amount range\n";
    int ch = getIntInput("Choice: ");
    if (ch == 1) {
        string key = getLineInputNonEmpty("Enter name keyword: ");
        bool found = false;
        for (int i = 0; i < budgetCount; ++i) if
(toLowerStr(budgets[i].name).find(toLowerStr(key)) != string::npos) {
            if (!found) { cout << setw(5) << "ID" << setw(30) << "Name" <<
setw(12) << "Amount" << setw(12) << "Type" << setw(18) << "Owner" << endl;
cout << string(80, '-') << endl; }
            found = true;
            cout << setw(5) << budgets[i].id << setw(30) << budgets[i].name
<< setw(12) << fixed << setprecision(2) << budgets[i].amount << setw(12) <<
budgets[i].type << setw(18) << budgets[i].owner << endl;
        }
        if (!found) cout << "No matches.\n";
    } else if (ch == 2) {
        string type = getLineInputNonEmpty("Enter type (Income/Expense):
");
        bool found = false;
        for (int i = 0; i < budgetCount; ++i) if
(toLowerStr(budgets[i].type) == toLowerStr(type)) {
            if (!found) { cout << setw(5) << "ID" << setw(30) << "Name" <<
setw(12) << "Amount" << setw(12) << "Type" << setw(18) << "Owner" << endl;
cout << string(80, '-') << endl; }
            found = true;
            cout << setw(5) << budgets[i].id << setw(30) << budgets[i].name
<< setw(12) << fixed << setprecision(2) << budgets[i].amount << setw(12) <<
budgets[i].type << setw(18) << budgets[i].owner << endl;
        }
        if (!found) cout << "No matches.\n";
    } else if (ch == 3) {
        string owner = getLineInputNonEmpty("Enter owner username: ");
        bool found = false;
        for (int i = 0; i < budgetCount; ++i) if
(toLowerStr(budgets[i].owner) == toLowerStr(owner)) {
            if (!found) { cout << setw(5) << "ID" << setw(30) << "Name" <<
setw(12) << "Amount" << setw(12) << "Type" << setw(18) << "Owner" << endl;
cout << string(80, '-') << endl; }
```

```
            found = true;
            cout << setw(5) << budgets[i].id << setw(30) << budgets[i].name
<< setw(12) << fixed << setprecision(2) << budgets[i].amount << setw(12) <<
budgets[i].type << setw(18) << budgets[i].owner << endl;
        }
        if (!found) cout << "No matches.\n";
    } else if (ch == 4) {
        double low = getDoubleInput("Enter low: ");
        double high = getDoubleInput("Enter high: ");
        bool found = false;
        for (int i = 0; i < budgetCount; ++i) if (budgets[i].amount >= low
&& budgets[i].amount <= high) {
            if (!found) { cout << setw(5) << "ID" << setw(30) << "Name" <<
setw(12) << "Amount" << setw(12) << "Type" << setw(18) << "Owner" << endl;
cout << string(80, '-') << endl; }
            found = true;
            cout << setw(5) << budgets[i].id << setw(30) << budgets[i].name
<< setw(12) << fixed << setprecision(2) << budgets[i].amount << setw(12) <<
budgets[i].type << setw(18) << budgets[i].owner << endl;
        }
        if (!found) cout << "No matches.\n";
    } else cout << "Invalid choice.\n";
}

void sortBudgetsMenu() {
    cout << "Sort by: 1) Amount desc 2) Amount asc 3) Name asc 4) Name desc
5) Type then amount\n";
    int ch = getIntInput("Choice: ");
    if (budgetCount <= 1) { cout << "Not enough records to sort.\n";
return; }
    if (ch == 1) {
        sort(budgets, budgets + budgetCount, [](const Budget &a, const
Budget &b){ return a.amount > b.amount; });
    } else if (ch == 2) {
        sort(budgets, budgets + budgetCount, [](const Budget &a, const
Budget &b){ return a.amount < b.amount; });
    } else if (ch == 3) {
        sort(budgets, budgets + budgetCount, [](const Budget &a, const
Budget &b){ return toLowerStr(a.name) < toLowerStr(b.name); });
    } else if (ch == 4) {
        sort(budgets, budgets + budgetCount, [](const Budget &a, const
Budget &b){ return toLowerStr(a.name) > toLowerStr(b.name); });
```

```
    } else if (ch == 5) {
        sort(budgets, budgets + budgetCount, [](const Budget &a, const
Budget &b){
            if (toLowerStr(a.type) != toLowerStr(b.type)) return
toLowerStr(a.type) < toLowerStr(b.type);
            return a.amount > b.amount;
        });
    } else { cout << "Invalid choice.\n"; return; }
    saveBudgetsToFile();
    cout << "Sorted and saved.\n";
}

void showSummary() {
    double income = computeTotalIncome();
    double expense = computeTotalExpense();
    double balance = income - expense;
    int incCount = 0, expCount = 0;
    double avgInc = 0.0, avgExp = 0.0;
    double maxInc = -1.0, maxExp = -1.0;
    string maxIncName, maxExpName;
    for (int i = 0; i < budgetCount; ++i) {
        if (toLowerStr(budgets[i].type) == "income") {
            ++incCount;
            avgInc += budgets[i].amount;
            if (budgets[i].amount > maxInc) { maxInc = budgets[i].amount;
maxIncName = budgets[i].name; }
        } else {
            ++expCount;
            avgExp += budgets[i].amount;
            if (budgets[i].amount > maxExp) { maxExp = budgets[i].amount;
maxExpName = budgets[i].name; }
        }
    }
    if (incCount) avgInc /= incCount;
    if (expCount) avgExp /= expCount;
    int pending = 0;
    for (int i = 0; i < requestCount; ++i) if
(toLowerStr(requests[i].status) == "pending") ++pending;
    drawHeaderBox("SUMMARY REPORT");
    cout << "Total Income    : " << fixed << setprecision(2) << income <<
"\n";
    cout << "Total Expense   : " << fixed << setprecision(2) << expense <<
```

```
"\n";
    cout << "Balance         : " << fixed << setprecision(2) << balance <<
"\n\n";
    cout << "Income Count    : " << incCount << "   Avg Income  : " <<
fixed << setprecision(2) << avgInc << "\n";
    cout << "Expense Count   : " << expCount << "   Avg Expense : " <<
fixed << setprecision(2) << avgExp << "\n\n";
    if (maxInc >= 0) cout << "Largest Income  : " << maxIncName << " (" <<
fixed << setprecision(2) << maxInc << ")\n";
    if (maxExp >= 0) cout << "Largest Expense : " << maxExpName << " (" <<
fixed << setprecision(2) << maxExp << ")\n";
    cout << "\nPending Requests: " << pending << " / " << requestCount <<
"\n";
    cout << "Profiles        : " << profileCount << "   Accounts: " <<
accountCount << "\n";
    cout << "+" << string(60, '=') << "+\n";
}

void createAccount() {
    if (accountCount >= MAX_ACCOUNTS) { cout << "Account storage full.\n";
return; }
    Account a;
    cout << "Enter username: ";
    getline(cin, a.username);
    while (a.username.empty()) { cout << "Username cannot be empty. Try
again: "; getline(cin, a.username); }
    bool exists = false;
    for (int i = 0; i < accountCount; ++i) if
(toLowerStr(accounts[i].username) == toLowerStr(a.username)) { exists =
true; break; }
    if (exists) { cout << "Username exists. Aborting.\n"; return; }
    cout << "Enter password: ";
    getline(cin, a.password);
    cout << "Enter role (Parent/Child): ";
    getline(cin, a.role);
    while (toLowerStr(a.role) != "parent" && toLowerStr(a.role) != "child")
{ cout << "Role must be Parent or Child. Enter role: "; getline(cin,
a.role); }
    a.age = getIntInput("Enter age: ");
    accounts[accountCount++] = a;
    saveAccountsToFile();
    if (toLowerStr(a.role) == "child") {
```

```
        if (profileCount >= MAX_PROFILES) { cout << "Profile storage
full.\n"; return; }
        Profile p;
        p.username = a.username;
        cout << "Enter display name for child: ";
        getline(cin, p.displayName);
        if (p.displayName.empty()) p.displayName = a.username;
        p.age = a.age;
        p.allowance = 0.0;
        profiles[profileCount++] = p;
        saveProfilesToFile();
    }
    cout << "Account created.\n";
}

Account* loginAccount() {
    cout << "Enter username: ";
    string user;
    getline(cin, user);
    cout << "Enter password: ";
    string pass;
    getline(cin, pass);
    for (int i = 0; i < accountCount; ++i) if (accounts[i].username == user
&& accounts[i].password == pass) return &accounts[i];
    cout << "Invalid credentials.\n";
    return nullptr;
}

Profile* findProfileByUsername(const string &uname) {
    for (int i = 0; i < profileCount; ++i) if
(toLowerStr(profiles[i].username) == toLowerStr(uname)) return
&profiles[i];
    return nullptr;
}

void submitRequest(const Account &acc) {
    if (toLowerStr(acc.role) != "child") { cout << "Only child accounts can
submit requests.\n"; return; }
    if (requestCount >= MAX_REQUESTS) { cout << "Request storage full.\n";
return; }
    Request r;
    r.id = nextRequestId++;
```

```
    r.childUsername = acc.username;
    Profile* p = findProfileByUsername(acc.username);
    if (p) r.childName = p->displayName;
    else { cout << "Enter child display name: "; getline(cin, r.childName);
if (r.childName.empty()) r.childName = acc.username; }
    cout << "Enter purpose: ";
    getline(cin, r.purpose);
    r.amount = getDoubleInput("Enter requested amount: ");
    r.status = "Pending";
    requests[requestCount++] = r;
    saveRequestsToFile();
    cout << "Request submitted.\n";
}

void parentProcessRequests() {
    if (requestCount == 0) { cout << "No requests.\n"; return; }
    cout <<
"+-------------------------------------------------------------------------
----------------------+\n";
    cout << "|  ID | Username         | Child Name         | Purpose
|    Amount | Status     |\n";
    cout <<
"+-------------------------------------------------------------------------
----------------------+\n";
    for (int i = 0; i < requestCount; ++i) {
        cout << "| " << setw(3) << requests[i].id << " | " << left <<
setw(16) << requests[i].childUsername << " | " << setw(18) <<
requests[i].childName << " | " << setw(28) << requests[i].purpose << " | "
<< right << setw(9) << fixed << setprecision(2) << requests[i].amount << "
| " << setw(10) << requests[i].status << " |\n";
    }
    cout <<
"+-------------------------------------------------------------------------
----------------------+\n";
    int id = getIntInput("Enter request ID to process (0 to cancel): ");
    if (id == 0) return;
    int idx = -1;
    for (int i = 0; i < requestCount; ++i) if (requests[i].id == id) { idx
= i; break; }
    if (idx == -1) { cout << "Request not found.\n"; return; }
    if (toLowerStr(requests[idx].status) != "pending") { cout << "Already
processed.\n"; return; }
```

```
    cout << "1) Approve  2) Reject\n";
    int ch = getIntInput("Choice: ");
    if (ch == 1) {
        requests[idx].status = "Approved";
        if (budgetCount < MAX_BUDGETS) {
            Budget b;
            b.id = nextBudgetId++;
            b.name = "Approved Request - " + requests[idx].childName + " -
" + requests[idx].purpose;
            b.amount = requests[idx].amount;
            b.type = "Expense";
            b.owner = "system";
            budgets[budgetCount++] = b;
            saveBudgetsToFile();
            cout << "Request approved and recorded as expense.\n";
        } else cout << "Budget storage full; cannot record expense.\n";
    } else if (ch == 2) {
        requests[idx].status = "Rejected";
        cout << "Request rejected.\n";
    } else cout << "Invalid choice.\n";
    saveRequestsToFile();
}

void manageProfiles() {
    cout << "Profile Manager: 1) List  2) Edit Allowance  3) Add  4) Remove
0) Back\n";
    int ch = getIntInput("Choice: ");
    if (ch == 0) return;
    if (ch == 1) {
        if (profileCount == 0) cout << "No profiles.\n";
        else {
            cout <<
"+----------------------------------------------------------------+\n";
            cout << "| Username       | Display Name              | Age |
Allowance |\n";
            cout <<
"+----------------------------------------------------------------+\n";
            for (int i = 0; i < profileCount; ++i) cout << "| " << setw(14)
<< profiles[i].username << " | " << setw(24) << profiles[i].displayName <<
" | " << setw(3) << profiles[i].age << " | " << setw(9) << fixed <<
setprecision(2) << profiles[i].allowance << " |\n";
            cout <<
```

```
"+------------------------------------------------------------+\n";
        }
    } else if (ch == 2) {
        string uname = getLineInputNonEmpty("Enter username: ");
        Profile* p = findProfileByUsername(uname);
        if (!p) { cout << "Profile not found.\n"; return; }
        double a = getDoubleInput("Enter new allowance: ");
        p->allowance = a;
        saveProfilesToFile();
        cout << "Allowance updated.\n";
    } else if (ch == 3) {
        if (profileCount >= MAX_PROFILES) { cout << "Profile storage
full.\n"; return; }
        Profile p;
        p.username = getLineInputNonEmpty("Enter username (must match
account): ");
        bool accExists = false;
        for (int i = 0; i < accountCount; ++i) if
(toLowerStr(accounts[i].username) == toLowerStr(p.username)) { accExists =
true; break; }
        if (!accExists) { cout << "No account with that username; create
account first.\n"; return; }
        p.displayName = getLineInputNonEmpty("Enter display name: ");
        p.age = getIntInput("Enter age: ");
        p.allowance = getDoubleInput("Enter allowance: ");
        profiles[profileCount++] = p;
        saveProfilesToFile();
        cout << "Profile added.\n";
    } else if (ch == 4) {
        string uname = getLineInputNonEmpty("Enter username to remove: ");
        int idx = -1;
        for (int i = 0; i < profileCount; ++i) if
(toLowerStr(profiles[i].username) == toLowerStr(uname)) { idx = i; break; }
        if (idx == -1) { cout << "Profile not found.\n"; return; }
        for (int i = idx; i < profileCount - 1; ++i) profiles[i] =
profiles[i + 1];
        profileCount--;
        saveProfilesToFile();
        cout << "Profile removed.\n";
    } else cout << "Invalid choice.\n";
}
```

```
void manageAccounts() {
    cout << "Account Manager: 1) List  2) Delete  3) Reset Password  0)
Back\n";
    int ch = getIntInput("Choice: ");
    if (ch == 0) return;
    if (ch == 1) {
        if (accountCount == 0) cout << "No accounts.\n";
        else {
            cout << "+-------------------------------+\n";
            cout << "| Username       | Role    | Age |\n";
            cout << "+-------------------------------+\n";
            for (int i = 0; i < accountCount; ++i) cout << "| " << setw(14)
<< accounts[i].username << " | " << setw(7) << accounts[i].role << " | " <<
setw(3) << accounts[i].age << " |\n";
            cout << "+-------------------------------+\n";
        }
    } else if (ch == 2) {
        string uname = getLineInputNonEmpty("Enter username to delete: ");
        int idx = -1;
        for (int i = 0; i < accountCount; ++i) if
(toLowerStr(accounts[i].username) == toLowerStr(uname)) { idx = i; break; }
        if (idx == -1) { cout << "Account not found.\n"; return; }
        for (int i = idx; i < accountCount - 1; ++i) accounts[i] =
accounts[i + 1];
        accountCount--;
        saveAccountsToFile();
        cout << "Account deleted.\n";
    } else if (ch == 3) {
        string uname = getLineInputNonEmpty("Enter username to reset
password: ");
        int idx = -1;
        for (int i = 0; i < accountCount; ++i) if
(toLowerStr(accounts[i].username) == toLowerStr(uname)) { idx = i; break; }
        if (idx == -1) { cout << "Account not found.\n"; return; }
        cout << "Enter new password: ";
        string np; getline(cin, np);
        accounts[idx].password = np;
        saveAccountsToFile();
        cout << "Password reset.\n";
    } else cout << "Invalid choice.\n";
}
```

```
void parentMenu(Account &acc) {
    int ch;
    int pending = 0;
    for (int i = 0; i < requestCount; ++i) if
(toLowerStr(requests[i].status) == "pending") ++pending;
    if (pending > 0) cout << "You have " << pending << " pending
request(s).\n";
    while (true) {
        drawHeaderBox("PARENT MENU");
        cout << "1) Add Budget Record\n2) Show All Budgets\n3) Edit
Budget\n4) Delete Budget\n5) Search / Filter\n6) Sort Budgets\n7) Summary
Report\n8) View & Process Requests\n9) Manage Profiles\n10) Manage
Accounts\n0) Logout\n";
        ch = getIntInput("Choice: ");
        if (ch == 0) break;
        if (ch == 1) addBudgetInteractive(acc.username);
        else if (ch == 2) showAllBudgetsFancy();
        else if (ch == 3) editBudgetInteractive(acc.username, true);
        else if (ch == 4) deleteBudgetInteractive(acc.username, true);
        else if (ch == 5) searchBudgetsMenu();
        else if (ch == 6) sortBudgetsMenu();
        else if (ch == 7) showSummary();
        else if (ch == 8) parentProcessRequests();
        else if (ch == 9) manageProfiles();
        else if (ch == 10) manageAccounts();
        else cout << "Invalid choice.\n";
    }
}

void childMenu(Account &acc) {
    bool isAdult = acc.age >= 18;
    while (true) {
        drawHeaderBox("CHILD MENU");
        cout << "1) Show All Budgets\n2) Request Budget\n3) View My
Requests\n";
        if (isAdult) cout << "4) Add Budget (as owner)\n5) Edit My
Budgets\n6) Delete My Budgets\n";
        cout << "7) Summary Report\n0) Logout\n";
        int ch = getIntInput("Choice: ");
        if (ch == 0) break;
        if (ch == 1) showAllBudgetsFancy();
        else if (ch == 2) submitRequest(acc);
```

```
        else if (ch == 3) {
            bool found = false;
            cout << "+-----------------------------------------+\n";
            cout << "| ID | Purpose                    | Amount |\n";
            cout << "+-----------------------------------------+\n";
            for (int i = 0; i < requestCount; ++i) if
(toLowerStr(requests[i].childUsername) == toLowerStr(acc.username)) {
                found = true;
                cout << "| " << setw(3) << requests[i].id << " | " << left
<< setw(26) << requests[i].purpose << " | " << right << setw(7) << fixed <<
setprecision(2) << requests[i].amount << " |\n";
            }
            cout << "+-----------------------------------------+\n";
            if (!found) cout << "No requests found.\n";
        } else if (isAdult && ch == 4) addBudgetInteractive(acc.username);
        else if (isAdult && ch == 5) editBudgetInteractive(acc.username,
true);
        else if (isAdult && ch == 6) deleteBudgetInteractive(acc.username,
true);
        else if (ch == 7) showSummary();
        else cout << "Invalid choice.\n";
    }
}

void bootstrapDefaultsIfEmpty() {
    if (accountCount == 0) {
        Account p; p.username = "parent"; p.password = "parent123"; p.role
= "Parent"; p.age = 40;
        accounts[accountCount++] = p;
        Account c; c.username = "child"; c.password = "child123"; c.role =
"Child"; c.age = 14;
        accounts[accountCount++] = c;
        saveAccountsToFile();
    }
    if (profileCount == 0) {
        Profile p; p.username = "child"; p.displayName = "Child User";
p.age = 14; p.allowance = 0.0;
        profiles[profileCount++] = p;
        saveProfilesToFile();
    }
}
```

```
int main() {
    loadAccountsFromFile();
    loadProfilesFromFile();
    loadBudgetsFromFile();
    loadRequestsFromFile();
    bootstrapDefaultsIfEmpty();
    drawHeaderBox("FAMILY BUDGET MANAGER");
    while (true) {
        cout << "\n+-------------------------------------------------+\n";
        cout << "| 1) Parent Login                                 |\n";
        cout << "| 2) Child Login                                  |\n";
        cout << "| 3) Create Account                               |\n";
        cout << "| 4) Show Summary                                 |\n";
        cout << "| 5) Exit                                         |\n";
        cout << "+-------------------------------------------------+\n";
        int mainChoice = getIntInput("Enter choice: ");
        if (mainChoice == 5) { cout << "Goodbye.\n"; break; }
        if (mainChoice == 1) {
            cin.ignore(numeric_limits<streamsize>::max(), '\n');
            cout << "Parent Login\n";
            Account *acc = loginAccount();
            if (acc && toLowerStr(acc->role) == "parent") parentMenu(*acc);
            else cout << "Login failed or not a parent account.\n";
        } else if (mainChoice == 2) {
            cin.ignore(numeric_limits<streamsize>::max(), '\n');
            cout << "Child Login\n";
            Account *acc = loginAccount();
            if (acc && toLowerStr(acc->role) == "child") childMenu(*acc);
            else cout << "Login failed or not a child account.\n";
        } else if (mainChoice == 3) {
            cin.ignore(numeric_limits<streamsize>::max(), '\n');
            createAccount();
        } else if (mainChoice == 4) {
            showSummary();
        } else cout << "Invalid choice.\n";
    }
    return 0;
}
```

APPENDIX B: Drive of the Project File

https://drive.google.com/drive/folders/175geeeUYvY46yDRSvtZxB6IgK2DSLHuz?usp=sharing