

# **AI-based Personal Finance Management System**

Project submitted to the  
SRM University – AP, Andhra Pradesh  
for the partial fulfillment of the requirements to award the degree of

**Bachelor of Technology/Master of Technology**

In

**Computer Science and Engineering  
School of Engineering and Sciences**

Submitted by

**Y.Karthik Krishna (AP23110010530)**

**Shaik Kaisar Parvez (AP23110010522)**

**S.Dinesh (AP23110010507)**

**V.Rama Naidu (AP23110010534)**



Under the Guidance of  
(Supervisor Name)

**SRM University-AP Neerukonda,  
Mangalagiri, Guntur  
Andhra Pradesh - 522 240**



**[Month, Year]**

# Certificate

Date: 12-nov-24

This is to certify that the work present in this Project entitled “AI-based Personal Finance Management System” has been carried out by [Y.Karthik Krishna, Shaik Kaisar Parvez, S.Dinesh V.Rama Naidu] under my/our supervision. The work is genuine, original, and suitable for submission to the SRM University – AP for the award of Bachelor of Technology/Master of Technology in **School of Engineering and Sciences**.

## Supervisor

(Signature)

Prof. / Dr. [Kavitha Rani] Designation,  
Affiliation.

## Co-supervisor

(Signature)

Prof. / Dr. [Name]  
Designation, Affiliation.

## **Acknowledgements**

I would like to thank all those who contributed to the successful completion of this AI-based personal finance management system. I am thankful to Kavitha Rani, for the invaluable guidance and support during the development process. Their expertise and constructive feedback helped shape the project.

I am grateful to my colleagues and peers who cooperated with me and gave me the confidence to proceed with the task. I thank my family for the support and tolerance during all the stages of this project.

I would not have completed this project if it were not for the availability of resources and literature in the field of software development, which inspired me to pursue this project with a good approach

# Table of Contents

Certificate	4
Acknowledgements	6
Table of Contents	7
Abstract	8
List of Tables	9
List of Equations	12
1. 1214	
2. 1618	
3.	184.
	205.
	21References
	25

## Abstract

This program is designed to provide a comprehensive personal finance management tool, enabling users to track income and expenses, set budgets, and receive savings recommendations. Built using C++ and object-oriented principles, the program features classes for managing different types of financial transactions, such as expenses and income, using a base Transaction class with derived classes for specific transaction types. A User class organizes and stores transactions, calculates total expenses, and generates a summary report that includes remaining balance and income breakdowns.

Additionally, the program utilizes a templated Budget class to allow users to define budget categories and amounts, which are stored in a map for quick access and display. The program also includes a Recommendation class that suggests a savings plan based on the user's income and spending patterns, encouraging financial planning through a recommended savings goal.

The program incorporates robust error handling for user input validation, using `shared_ptr` for efficient memory management of transactions and applying exception handling to ensure reliable and user-friendly interaction. With its structured design, this tool serves as a functional prototype for personal finance management, emphasizing modularity, clarity, and ease of use in tracking and planning financial goals.

## CODE:

```
#include <iostream>
#include <vector>
#include <string>
#include <map>
#include <stdexcept>
#include <memory>
#include <iomanip>
#include <limits>

using namespace std;

// Base Transaction Class
class Transaction {
protected:
    double amount;
    string category;
    string date;

public:
    Transaction(double amt, const string& cat, const string& dt)
    {
        amount=amt;
        category=cat;
        date=dt;
    }

    virtual void display() const = 0; // Pure virtual function
    double getAmount() const { return amount; }
    string getCategory() const { return category; }
};

// Derived ExpenseTransaction Class
class ExpenseTransaction : public Transaction {
public:
    ExpenseTransaction(double amt, const string& cat, const string& dt)
        : Transaction(amt, cat, dt) {}

    void display() const override {
        cout << "Expense - Category: " << category << ", Amount: Rs" << fixed << setprecision(2) <<
        amount << ", Date: " << date << endl;
    }
};

// Derived IncomeTransaction Class
class IncomeTransaction : public Transaction {
private:
    string source;

public:
    IncomeTransaction(double amt, const string& src, const string& dt)
        : Transaction(amt, "Income", dt) {
        source=src;
    }

    void display() const override {
        cout << "Income - Source: " << source << ", Amount: Rs" << fixed << setprecision(2) << amount
        << ", Date: " << date << endl;
    }
};
```



```

    }
};

// User Class
class User {
private:
    string name;
    double income;
    vector<shared_ptr<Transaction>> transactions;

public:
    User(const string& userName, double userIncome)
        : name(userName), income(userIncome) {}

    void addTransaction(shared_ptr<Transaction> transaction) {
        if (transaction->getAmount() < 0) {
            throw invalid_argument("Transaction amount cannot be negative.");
        }
        transactions.push_back(transaction);
    }

    void displayTransactions() const {
        cout << name << "'s Transactions:" << endl;
        for (const auto& transaction : transactions) {
            transaction->display();
        }
    }

    double getIncome() const { return income; }

    double calculateTotalExpenses() const {
        double total = 0.0;
        for (const auto& transaction : transactions) {
            if (transaction->getCategory() != "Income") {
                total += transaction->getAmount();
            }
        }
        return total;
    }

    void generateSummaryReport() const {
        cout << "\n--- Summary Report for " << name << " ---" << endl;
        cout << "Total Income: Rs" << fixed << setprecision(2) << income << endl;
        cout << "Total Expenses: Rs" << fixed << setprecision(2) << calculateTotalExpenses() << endl;
        cout << "Remaining Balance: Rs" << fixed << setprecision(2)
            << (income - calculateTotalExpenses()) << endl;
        cout << "-----" << endl;
    }
};

// Budget Template Class
template<typename T>
class Budget {
private:
    map<string, T> categories;

public:
    void setBudget(const string& category, T amount) {
        categories[category] = amount;
    }
}

```

```

void displayBudgets() const {
    cout << "\nBudgets:" << endl;
    for (const auto& pair : categories) {
        cout << pair.first << ": Rs" << fixed << setprecision(2) << pair.second << endl;
    }
}

T getBudgetForCategory(const string& category) const {
    auto it = categories.find(category);
    if (it != categories.end()) {
        return it->second;
    } else {
        throw invalid_argument("Budget category not found.");
    }
}
};

// Recommendation Class
class Recommendation {
public:
    static void suggestSavingsPlan(const User& user) {
        double totalExpenses = user.calculateTotalExpenses();

        // Suggest saving 20% of income
        double savingsGoal = user.getIncome() * 0.20;

        cout << "\n--- Savings Recommendation ---" << endl;
        cout << "Based on your income of Rs"
             << fixed << setprecision(2)
             << user.getIncome()
             << ", we recommend saving at least 20% of your income."
             << endl;

        if (totalExpenses > savingsGoal) {
            cout << "You are currently spending more than your savings goal."
                 << endl;
            cout << "Consider reducing your expenses."
                 << endl;
        } else {
            cout << "You are on track to meet your savings goal!"
                 << endl;
        }

        cout << "-----"
             << endl;
    }
};

// Function to get valid input from the user
double getValidDoubleInput(const string& prompt) {
    double value;

    while (true) {
        cout << prompt;
        cin >> value;

        if (cin.fail()) { // Check for invalid input
            cin.clear(); // Clear the error flag
            cin.ignore(numeric_limits<streamsize>::max(), '\n'); // Discard invalid input

```

```

        cout << "Invalid input. Please enter a numeric value." << endl;
        continue; // Restart the loop
    }

    cin.ignore(numeric_limits<streamsize>::max(), '\n'); // Clear the buffer
    return value; // Return valid input
}

// Function to get a valid string input from the user
string getValidStringInput(const string& prompt) {
    string input;

    while (true) {
        cout << prompt;
        getline(cin, input);

        if (input.empty()) { // Check for empty input
            cout << "Input cannot be empty. Please enter a valid input." << endl;
            continue; // Restart the loop
        }

        return input; // Return valid input
    }
}

int main() {
    try {
        string userName;

        // Get user name and income
        cout << "Enter your name: ";
        getline(cin, userName);

        double userIncome = getValidDoubleInput("Enter your monthly income: ");

        User user(userName, userIncome);

        char addMoreTransactions = 'y';

        // Adding transactions
        while (addMoreTransactions == 'y') {
            char transactionType;

            cout << "\nEnter transaction type (e for expense, i for income): ";
            cin >> transactionType;
            cin.ignore(numeric_limits<streamsize>::max(), '\n'); // Clear the buffer

            if (transactionType == 'e') { // Expense transaction
                double expenseAmount = getValidDoubleInput("Enter expense amount: ");
                string expenseCategory = getValidStringInput("Enter expense category: ");
                string expenseDate = getValidStringInput("Enter expense date (YYYY-MM-DD): ");

                user.addTransaction(make_shared<ExpenseTransaction>(expenseAmount,
                    expenseCategory, expenseDate));

            } else if (transactionType == 'i') { // Income transaction
                double incomeAmount = getValidDoubleInput("Enter income amount: ");
                string incomeSource = getValidStringInput("Enter income source: ");
                string incomeDate = getValidStringInput("Enter income date (YYYY-MM-DD): ");
            }
        }
    }
}

```

```

        user.addTransaction(make_shared<IncomeTransaction>(incomeAmount, incomeSource,
incomeDate));

    } else {
        cout << "Invalid transaction type. Please enter 'e' or 'i'." << endl;
        continue; // Restart loop for valid input
    }

    cout << "Do you want to add another transaction? (y/n): ";
    cin >> addMoreTransactions;
    cin.ignore(numeric_limits<streamsize>::max(), '\n'); // Clear buffer after char input
}

// Display transactions and budget setup
user.displayTransactions();

Budget<double> budget;

char setupBudget = 'y';
while (setupBudget == 'y') {
    string budgetCategory = getValidStringInput("Enter budget category: ");
    double budgetAmount = getValidDoubleInput("Enter budget amount for this category: ");

    budget.setBudget(budgetCategory, budgetAmount);

    cout << "Do you want to add another budget? (y/n): ";
    cin >> setupBudget;
    cin.ignore(numeric_limits<streamsize>::max(), '\n'); // Clear buffer after char input
}

// Display budgets and generate summary report
budget.displayBudgets();
user.generateSummaryReport();

// Provide savings recommendation based on user's data
Recommendation::suggestSavingsPlan(user);


} catch (const exception& e) {
    cerr << "Error: " << e.what() << endl;
}

return 0;
}

```



## List of Tables

Component	Purpose	Input	Output
Transaction Class	Abstract base class to represent a general financial transaction.	<code>amount</code> (double), <code>category</code> (string), <code>date</code> (string)	Abstract display function; provides access to amount and category details.
ExpenseTransaction Class	Represents an expense transaction, derived from <code>Transaction</code> .	Inherits <code>amount</code> , <code>category</code> , <code>date</code> ; <code>display()</code> function.	Displays formatted expense details.
IncomeTransaction Class	Represents an income transaction, derived from <code>Transaction</code> .	Inherits <code>amount</code> , <code>category</code> , <code>date</code> ; <code>source</code> (string); <code>display()</code> function.	Displays formatted income details with source.
User Class	Manages user profile, transactions, and provides summary reports.	<code>name</code> (string), <code>income</code> (double), <code>transactions</code> (vector of <code>shared_ptr&lt;Transaction&gt;</code> )	Adds transactions, displays transactions, generates summary report.
<code>addTransaction()</code>	Adds a transaction to the user's record. Validates non-negative amount. 	<code>transaction</code> ( <code>shared_ptr&lt;Transaction&gt;</code> )	Adds to <code>transactions</code> vector or throws an error if amount is negative.

<code>displayTransactions()</code>	Displays all transactions associated with the user.	None	Prints each transaction by invoking its <code>display()</code> function.
<code>calculateTotalExpenses()</code>	Calculates total expenses for the user.	None	Returns total expenses (excluding income).
<code>generateSummaryReport()</code>	Generates a report with total income, total expenses, and remaining balance.	None	Displays summary of income, expenses, and remaining balance.
Budget Class	Template class that manages budget allocation by category.	<code>categories</code> (map<string, T>)	Allows budget setting and retrieval by category.
<code>setBudget()</code>	Sets a budget amount for a specified category.	<code>category</code> (string), <code>amount</code> (T)	Adds or updates the budget category with amount in <code>categories</code> .
<code>displayBudgets()</code>	Displays all budget categories and amounts.	None	Prints each category and its budgeted amount.

<code>getBudgetForCategory()</code>	Retrieves the budgeted amount for a specified category.	<code>category</code> (string)	Returns the budgeted amount for the specified category, or error if not found.
<b>Recommendation Class</b>	Provides recommendations for savings based on user income and expenses.	<code>user</code> (User)	Suggests savings goal based on 20% of user's income.
<code>suggestSavingsPlan()</code>	Recommends a savings strategy by comparing expenses against a 20% income savings goal.	<code>user</code> (User)	Prints recommendation and advises user on saving or spending adjustments.
<b>Input Validation Functions</b>	Ensures valid input for income, amount, and string entries.	User-provided values (numeric or string)	Returns validated input or prompts user to re-enter valid data.
<code>getValidDoubleInput()</code>	Validates and retrieves a double input from user, handling invalid entries.	<code>prompt</code> (string)	Returns valid double input from user.

<code>getValidStringInput()</code>	Validates and retrieves a string input from user, handling empty entries.	<code>prompt</code> (string)	Returns non-empty string input from user.
<b>Main Function (main)</b>	Orchestrates the program by initializing a user, handling transactions, and setting budgets.	User input via console (name, income, transactions, budgets)	Displays transactions, budgets, summary report, and savings recommendation.



## List of Equations

Here are the main equations represented in the code, each performing essential financial calculations for tracking and planning user expenses, income, and savings goals:

### 1. Total Expenses Calculation:

This equation calculates the total amount spent by the user across all transactions marked as expenses. It is represented by:

$$\text{Total Expenses} = \sum_{i=1}^N \text{expense amount}$$

where  $(N)$  is the number of expense transactions. By iterating over each transaction categorized as an expense, the program sums the amounts, providing insight into the user's spending.

### 2. Remaining Balance Calculation:

The remaining balance is derived by subtracting total expenses from the user's income. This equation helps the user understand how much of their income remains after all expenses:

$$\text{Remaining Balance} = \text{Income} - \text{Total Expenses}$$

### 3. Savings Goal Calculation:

To encourage financial planning, the code suggests saving 20% of the user's income. The savings goal is calculated with:

$$\text{text}\{\text{Savings Goal}\} = \text{text}\{\text{Income}\} \text{ times } 0.20$$

This formula multiplies the total income by 20%, providing a recommended savings target to help the user set aside money regularly.

#### 4. Budget Comparison by Category:

For each budget category, the code calculates the difference between the budgeted amount and actual spending, helping the user determine if they are staying within their budget for each category:

$$\text{text}\{\text{Budget Comparison (Category)}\} = \text{text}\{\text{Budget Amount}\} - \text{text}\{\text{Actual Expense}\}$$

# 1.Introduction

In today's financial landscape, effective budgeting, expense tracking, and savings planning are essential for personal financial well-being. This program offers a structured solution to managing these needs through various classes and modules, providing users with detailed insights into their income, expenses, and budgeting goals. The program is designed using advanced C++ concepts, such as object- oriented programming, polymorphism, smart pointers, and template classes, enabling it to be both modular and extendable. This introduction will cover the purpose, key functionalities, and structure of the code.

## 1. Purpose and Objectives

The primary purpose of this program is to provide users with a robust tool for financial management. Key objectives include:

1. Allowing users to log and categorize financial transactions, both income and expenses.
2. Enabling users to set and review budgets for various categories.
3. Providing an interactive summary report of income, expenses, and remaining balance.
4. Offering tailored savings recommendations based on user income and spending patterns.

This program serves as a comprehensive framework for users who wish to gain better control over their financial activities.

## 2. Structure of the Program

The codebase is organized into a series of classes that handle different aspects of financial management, from transaction tracking to budget planning and savings recommendations. Each component is designed to be both functional and extensible, ensuring that users can easily navigate and manage their financial information.

### 3. Key Components and Functionalities

Each component plays a unique role in achieving the program's overall objectives. Below is an overview of the core components:

#### 3.1. Transaction Management (Transaction, ExpenseTransaction, and IncomeTransaction Classes)

The program begins with a foundational Transaction class, which serves as a base class for handling financial transactions. Each transaction stores details such as the amount, category, and date. This base class is extended by two derived classes:

- ExpenseTransaction – This class represents expense records, capturing outflows of money in specific categories like food, transportation, or utilities.
- IncomeTransaction – This class represents income records, tracking inflows of money from different sources, such as salaries or freelance work.

These classes allow the program to manage and display various types of transactions, facilitating accurate financial tracking and reporting.

#### 3.2. User Profile and Transaction Summary (User Class)

The User class acts as the central hub for each individual's financial information. It contains the following functionalities:

1. Add Transaction– Allows users to add both income and expense transactions, which are stored in a list of `shared\_ptr` objects for memory management.
2. Display Transactions
  - Provides a summary of all transactions, showing category, amount, and date.
3. Calculate Total Expenses – Summarizes expenses by category, excluding income entries.
4. Generate Summary Report – Displays a detailed report of total income, total expenses, and remaining balance for the user.

The User class ensures that all financial data is captured, stored, and processed efficiently, enabling users to maintain a clear overview of their finances.

### 3.3. Budget Planning (Budget Template Class)

The Budget class allows users to set and manage budget limits for specific expense categories. Implemented as a template class, it can support various data types for flexibility:

- Set Budget – Users can assign budget limits for different categories like food, entertainment, and transportation.
- Display Budgets – Lists all budget categories and their respective limits.
- Retrieve Budget – Allows users to check the budget for a specific category.

This class provides a structured way for users to define spending goals, aiding in the effective management of their finances by monitoring their spending against set budgets.

### 3.4. Savings Recommendations (Recommendation Class)

The Recommendation class provides personalized financial guidance based on the user's income and expenses. It suggests a target savings amount—typically 20% of the user's income—and assesses whether current spending habits align with this

goal. If expenses exceed this target, it advises the user to reduce spending; if they are within limits, it offers positive reinforcement. This feature supports users in creating a sustainable financial plan, enhancing both short-term and long-term financial health.

#### 4. User Input and Error Handling

The program uses functions to validate user input for both numeric and string data, ensuring that entries are accurate and meaningful. Any negative values for transaction amounts or invalid budget categories trigger exceptions, guiding users toward correct usage. By integrating error handling and input validation, the program promotes smooth and reliable operation, enhancing user experience.

## 2.Methodology

This financial management tool is designed using object-oriented programming (OOP)\*\* principles to ensure modularity, reusability, and maintainability. It leverages smart pointers for memory management and efficient resource handling. The program is divided into distinct classes, each responsible for specific tasks, such as transaction handling, budgeting, and generating savings recommendations.

### 1. Transaction Management

At the heart of the system is the Transaction class, which stores essential transaction details such as amount, category, and date. This class is extended by two derived classes: ExpenseTransaction for expenses and IncomeTransaction for income. Each derived class includes specific properties and methods for managing their respective transaction types. The use of shared pointers enables dynamic memory management, ensuring that transactions are handled efficiently and without memory leaks.

### 2. User Management

The User class is designed to store and manage individual user data, including their name, monthly income, and a collection of transactions. The class provides functionality to add transactions, calculate total expenses, and generate a detailed summary report of the user's financial activities. This allows users to keep track of their spending and assess the financial health of their accounts in a clear and organized manner.

### 3. Budget Planning

The Budget class is responsible for managing budget limits across different categories, allowing users to set specific amounts they plan to spend in categories such as groceries, entertainment, etc. Using a map, the system stores the category names and their corresponding budget amounts, allowing users to modify and check their budget for each category easily. This feature helps users track their expenses against the predefined budget and make informed financial decisions.

#### 4. Savings Recommendation

The `Recommendation` class provides valuable insights by offering a savings plan based on the user's income. It suggests saving a percentage of the income (e.g., 20%) and evaluates whether the user's spending aligns with their savings goal. If the user is spending more than the suggested savings, the tool advises cutting back on certain expenses, promoting better financial habits and long-term savings goals.

#### 5. Input Validation and Error Handling

The system incorporates input validation to ensure that users enter valid values. It checks for conditions such as negative transaction amounts and empty fields for budget categories. The program also ensures that the user input for both numerical and string values is correct before proceeding. If any invalid data is entered, an error message prompts the user to enter the correct information, thus preventing system crashes or faulty calculations.



## Discussion

This program provides an effective framework for managing personal finances, integrating essential OOP principles like modularity, inheritance, and polymorphism to enable smooth organization and interaction between components. By structuring the code into discrete, well-defined classes, it becomes easier to expand or adjust features without disrupting the whole system. Each component of the code plays a distinct role, which collectively enhances the overall user experience and the program's functionality.

1. **Transaction Management:** The program uses a base `Transaction` class and derived classes `ExpenseTransaction` and `IncomeTransaction` to separate expense and income types. This structure not only organizes transactions by type but also makes it easy to add new types in the future. The `display()` function is a pure virtual function in the base class, making `Transaction` an abstract class and enforcing that each derived class must implement its specific way of displaying transaction information. This approach adds flexibility and scalability to the design.
2. **Smart Pointers for Memory Management:** By utilizing `shared_ptr` for storing transactions, the program ensures efficient memory management. `shared_ptr` handles dynamic memory automatically, freeing it when no longer in use. This reduces the risk of memory leaks and simplifies code maintenance, especially as the transaction list grows.
3. **User Budget and Expense Tracking:** The `User` class enables users to track income, add various transactions, and calculate total expenses. It also allows users to generate summary reports, detailing total income, expenses, and remaining balance. The modular design of this class allows for further enhancements, such as tracking expenses by category over time or generating detailed spending reports.
4. **Budget Setting and Management:** The `Budget` template class introduces flexibility in managing budgets for different categories. This class allows users to set category-specific budgets and retrieve them as needed. It also has potential for expansion: features like category-wise budget alerts, visualizations, or goal-setting could help users adhere to their budgets more effectively.
5. **Savings Recommendation:** The `Recommendation` class incorporates a simple yet practical financial tip: aiming to save 20% of income. This recommendation is based on a commonly suggested rule for healthy savings and is calculated based on the

user's income and expenses. While basic, it gives users a savings goal and suggests adjustments if spending exceeds the savings target. In future versions, this feature could offer more personalized savings advice, factoring in age, income, lifestyle, or financial goals.

6. User-Centered Error Handling: The program includes validation to ensure that users enter correct, non-negative amounts for transactions, categories, and budgets. This reduces potential errors in calculations and helps maintain data integrity, enhancing reliability and user experience.

7. Input Validation: Input validation functions are defined to ensure that users enter valid data, avoiding common input errors. These functions provide prompts and checks for accurate input, such as ensuring that amounts are non-negative and categories are non-empty.

## Concluding Remarks

This C++ program effectively demonstrates a practical application of object-oriented programming for personal finance management. By utilizing classes and inheritance, it achieves modularity and flexibility in handling different types of financial transactions, including expenses and income, while ensuring reusability and scalability of the code.

Through templates, `shared_ptr`, and `map`, the program manages budget categories dynamically and efficiently. It also prioritizes user experience with features like input validation, exception handling, and clear financial summaries. Overall, this program serves as a valuable tool for budgeting and savings planning, providing a solid foundation for further expansion, such as incorporating data persistence, enhanced reporting, and additional financial insights.

## Future Work

This personal finance management program can be further enhanced in several ways to improve functionality, user experience, and scalability. Future work on this project could include the following:

1. **Data Persistence:** Implement file handling or database integration to store and retrieve user data, transactions, and budgets. This would allow users to save and load their financial data, making the tool more practical for long-term use.
2. **Graphical User Interface (GUI):** Adding a GUI would make the program more user-friendly and visually appealing. A GUI could display income, expenses, budgets, and recommendations through charts and graphs, helping users better understand their finances.
3. **Expense Categorization and Analysis:** Enhance the program's categorization features by adding automatic expense categorization and detailed spending analysis. For example, using predefined categories or learning user preferences can make it easier to track spending habits.
4. **Customizable Savings Goals:** Introduce features that allow users to set custom savings goals and track their progress. This would enhance the Recommendation class by offering tailored savings advice based on specific financial targets.
5. **Monthly and Yearly Reports:** Expand the reporting system to generate monthly, quarterly, or yearly summaries, showing trends over time and helping users adjust budgets or savings plans based on patterns.
6. **Currency Conversion:** Adding currency conversion functionality would be beneficial for users managing finances in multiple currencies, especially if the program is used internationally.
7. **Notifications and Alerts:** Integrate notifications for exceeding budget limits, upcoming expenses, or unachieved savings goals to help users stay on top of their financial plans.

8. Security Enhancements: If the program stores sensitive financial data, implementing security measures like encryption and user authentication would be critical to protecting user information.

# References

## 1. C++ Standard Library Documentation

The C++ Standard Library provided essential information on utilizing various libraries, including `<vector>`, `<map>`, `<memory>`, `<string>`, and `<iomanip>`. These libraries were crucial for implementing data structures, memory management, and formatted output in the program.

Reference: "C++ Standard Library Documentation." Available at: [<https://en.cppreference.com/w/>](<https://en.cppreference.com/w/>)

## 2. Object-Oriented Programming in C++

Concepts of object-oriented programming such as inheritance, polymorphism, encapsulation, and templates were foundational to the program's structure. The design of base and derived classes, along with the use of abstract classes and virtual functions, enabled a flexible framework for handling different transaction types.

Reference: Stroustrup, Bjarne. The C++ Programming Language. Addison-Wesley, 2013.

## 3. Smart Pointers and Memory Management in C++

The use of `shared_ptr` from the `<memory>` library was guided by best practices in managing dynamic memory, ensuring efficient memory use and reducing risks of memory leaks.

Reference: Meyers, Scott. Effective Modern C++: 42 Specific Ways to Improve Your Use of C++11 and C++14. O'Reilly Media, 2014.

## 4. Exception Handling and Input Validation

Concepts of exception handling were applied to ensure robust error management, particularly in handling invalid user inputs. This reference provided insight into using try-catch blocks and exception classes to improve user experience.

Reference: "C++ Error Handling (Exception Handling)." Available at: [<https://www.geeksforgeeks.org/c-exceptions/>](<https://www.geeksforgeeks.org/c-exceptions/>)

## 5. Financial Budgeting and Savings Algorithms

For designing the budgeting, expense tracking, and savings recommendation features, general financial budgeting principles were consulted to create a basic yet realistic financial management tool.

Reference: "Financial Literacy and Budgeting Basics." Available at: [<https://www.investopedia.com/>](<https://www.investopedia.com/>)