Expense Tracker:

Expense Tracker Program Documentation

This document provides a comprehensive overview of the C++ Expense Tracker program, detailing its functionality, structure, and usage.

1. Introduction

The C++ Expense Tracker is a console-based application designed to help users manage their personal finances. It allows users to record their expenses and income, categorize them, view a list of all transactions, calculate total spending for a specific month, and persist this data by saving it to and loading it from a file. The program aims to provide a simple and effective way to keep track of financial activities.

2. Program Interface

Compiling and Running the Program:

To use the program, you will need a C++ compiler that supports C++11 or later (due to the use of `stod` and `to_string`).

1. **Save the Code:** Save the C++ code into a file named (for example) `expense_tracker.cpp`.
2. **Compile:** Open a terminal or command prompt and navigate to the directory where you saved the file. Compile the code using a C++ compiler. For example, with g++:
3. `g++ expense_tracker.cpp -o expense_tracker -std=c++11`
4. **Run:** Execute the compiled program: `./expense_tracker`

Program Interaction:

The program interacts with the user through a console-based menu. The user selects options by entering the corresponding number.

Terminating the Program:

To terminate the program, the user selects option '5. Save & Exit' from the main menu. This will save the current expense data to a file before closing the application.

3. Program Execution

Upon starting the program, it will attempt to load existing expense data from a file named `expenses.txt` located in the same directory as the executable. If the file doesn't exist or cannot be read, it will start with a fresh slate.

The user is then presented with a main menu:

`1. Add Expense 2. Add Income 3. View All Records 4. View Total Spent by Month 5. Save & Exit`

1. Add Expense:

- Prompts the user to choose a category for the expense: `Choose a category: 1. Bills 2. Entertainment 3. Food 4. Transportation`
  If an invalid choice is entered, it defaults to "Other".
- Asks the user to enter the date of the expense in YYYY-MM-DD format.
- Asks for the amount of the expense.
- The program validates that the amount is not negative.

2. Add Income:

- Asks the user to enter the date of the income in YYYY-MM-DD format.
- Asks for the income amount.
- The program validates that the amount is not negative.
- Income is stored with the category "Income".

3. View All Records:

- Displays all recorded expenses and income transactions. If no records exist, it will indicate so.
- Each record is displayed in the format: `Date: YYYY-MM-DD | Category: [CategoryName] | Amount: [Amount] Ft`

4. View Total Spent by Month:

- Asks the user to enter the year and month in YYYY-MM format (e.g., 2023-10).
- Calculates and displays the total amount spent (excluding "Income" category) for the specified month.
- The output format is: `Total spent in YYYY-MM: [TotalAmount] Ft`

5. Save & Exit:

- Saves all current expense and income data to the `expenses.txt` file.
- Confirms that data has been saved and then terminates the program.
- Handles potential errors during file saving.

If the user enters an invalid option in the main menu, an "Invalid option." message is displayed, and the menu is shown again.

4. Input and Output

Input:

1. **Menu Choices:** Integer values (1-5 for the main menu, 1-4 for category selection).
2. **Category:** Selected via a numerical choice, which translates to a string (e.g., "Bills", "Food").
3. **Date:** String in YYYY-MM-DD format. The program does not currently validate the correctness of the date format or its components beyond what's required for string operations.
4. **Amount:** Double-precision floating-point number for expenses and income.
5. **Year and Month:** String in YYYY-MM format for viewing monthly totals.
6. **Filename (internal):** The program uses a hardcoded filename `expenses.txt` for saving and loading data.

Output:

1. **Console Messages:** Prompts for user input, feedback messages, and displayed expense/income records.
2. **Expense/Income Display:** `Date: YYYY-MM-DD | Category: [CategoryName] | Amount: [Amount] Ft`
3. **Monthly Total Display:** `Total spent in YYYY-MM: [TotalAmount] Ft`
4. **File Output (`expenses.txt`):** Each line in the file represents one expense or income record in CSV (Comma Separated Values) format: `YYYY-MM-DD,[CategoryName],[Amount]`

1. **Save the Code:** Save the C++ code into a file named (for example) `expense_tracker.cpp`.
2. **Compile:** Open a terminal or command prompt and navigate to the directory where you saved the file. Compile the code using a C++ compiler.
3. **Run:** Execute the compiled program: `./expense_tracker`

5. Program Structure

The program is structured using classes and functions to organize its logic.

Classes:

1. **`Expense` Class:**
   Purpose: Represents a single financial transaction (either an expense or income).
   Private Members:
   - `string category`: Stores the category of the transaction (e.g., "Food", "Bills", "Income").
   - `string date`: Stores the date of the transaction (YYYY-MM-DD).
   - `double amount`: Stores the monetary value of the transaction.

   Public Members:

- `Expense()`: Default constructor, initializes with empty strings and zero amount.
- `Expense(const string& category, const string& date, double amount)`: Parameterized constructor to create an `Expense` object.
- `string getCategory() const`: Returns the category.
- `string getDate() const`: Returns the date.
- `double getAmount() const`: Returns the amount.
- `void display() const`: Prints the expense details to the console.
- `string toFileString() const`: Converts the expense details into a CSV string for file storage.
- `static Expense fromFileString(const string& line)`: A static factory method that parses a CSV string (from the file) and creates an `Expense` object.

2. **`ExpenseTracker` Class:**

- **Purpose:** Manages a collection of `Expense` objects. Handles adding, viewing, saving, and loading expenses.
- **Private Members:**
  - `Expense** expenses`: A dynamic array of pointers to `Expense` objects.

- `int count`: The current number of expenses stored.
- `int capacity`: The current allocated capacity of the `expenses` array.
- `void resize()`: A helper function to double the capacity of the `expenses` array when it's full.

Public Members:

- `ExpenseTracker()`: Constructor, initializes `count` to 0, `capacity` to 10, and allocates the initial `expenses` array.
- `~ExpenseTracker()`: Destructor, deallocates all `Expense` objects pointed to by the `expenses` array and then deallocates the array itself to prevent memory leaks.
- `void addExpense(const string& category, const string& date, double amount)`: Creates a new `Expense` object and adds it to the tracker. Throws an `invalid_argument` if the amount is negative. Calls `resize()` if needed.
- `void viewExpenses() const`: Displays all recorded expenses.
- `void viewTotalSpentByMonth(const string& yearMonth) const`: Calculates and displays the total expenses for a given month (YYYY-MM), excluding "Income" transactions.
- `void saveToFile(const string& filename) const`: Saves all expenses to the specified file. Throws an `ios_base::failure` if the file cannot be opened for writing.
- `void loadFromFile(const string& filename)`: Loads expenses from the specified file. If the file doesn't exist or cannot be opened, it simply returns (allowing the program to start fresh).

Functions:

1. **`string chooseCategory()`:**
    - **Purpose:** Prompts the user to select an expense category from a predefined list.
    - **Returns:** A string representing the chosen category. Defaults to "Other" for invalid input.

2. `Int main()`:

- **Purpose:** The entry point of the program.
- **Logic:**
    - Creates an `ExpenseTracker` object.
    - Attempts to load data from `expenses.txt`. Catches any exceptions during loading and informs the user if data couldn't be loaded.

    - Enters a `do-while` loop to display the main menu and process user choices.
    - Uses a `switch` statement to handle different menu options:
        - Case 1 (Add Expense): Calls `chooseCategory()`, gets date and amount, then calls `tracker.addExpense()`. Includes error handling for `addExpense`.
        - Case 2 (Add Income): Gets date and amount, then calls `tracker.addExpense()` with "Income" as the category. Includes error handling.
        - Case 3 (View All Records): Calls `tracker.viewExpenses()`.
        - Case 4 (View Total Spent by Month): Gets year-month input and calls `tracker.viewTotalSpentByMonth()`.
        - Case 5 (Save & Exit): Calls `tracker.saveToFile()`, prints a message, and breaks the loop. Includes error handling for file saving.
        - Default: Handles invalid menu options.
    - The loop continues until the user chooses option 5.
    - Returns 0 upon successful completion.

Data Structures:

- The primary data structure is a dynamic array of `Expense` pointers (`Expense** expenses`) within the `ExpenseTracker` class. This allows for flexible storage of an unknown number of expense records, with a resizing mechanism to accommodate growth.

File Handling:

- Uses `ofstream` for writing to `expenses.txt`.
- Uses `ifstream` for reading from `expenses.txt`.
- Error handling is implemented for file operations (e.g., inability to open a file for writing).

6. Testing and Verification

To ensure the program functions correctly, the following aspects should be tested:

1. **Adding Expenses/Income:**
    - Add expenses with valid categories, dates, and positive amounts.
    - Add income with valid dates and positive amounts.
    - Attempt to add an expense/income with a negative amount (should display an error and not add the record).
    - Test adding expenses until the initial capacity of the internal array is exceeded to verify the `resize()` functionality.

2. **Viewing Records:**

- View records when the list is empty (should display "No expenses recorded.").
- View records after adding several expenses and income entries. Verify all details (date, category, amount) are displayed correctly.

3. Viewing Total Spent by Month:

- Test with a month that has expenses (excluding income).
- Test with a month that has only income (should result in 0 spent).
- Test with a month that has both expenses and income (should only sum expenses).
- Test with a month that has no transactions.
- Test with various YYYY-MM formats.

4. Saving and Loading Data:

- Add some records, save and exit. Relaunch the program and verify the records are loaded correctly.
- Modify records (e.g., add more), save and exit. Relaunch and verify changes are persistent.
- Test behavior when `expenses.txt` does not exist (should start fresh).
- Manually create an `expenses.txt` with valid and invalid lines to test parsing robustness (though current error handling for `fromFileString` is minimal).
- Test file saving error (e.g., by making the file read-only, though this is harder to simulate programmatically without OS-level changes).

5. Input Validation:

    - Enter invalid choices for menus.
    - While the current code doesn't rigorously validate date formats (e.g., "YYYY-MM-DD"), testing with various string inputs for dates is advisable to understand current behavior.

6. Error Handling:

    - Verify that error messages for negative amounts are displayed.
    - Verify that error messages for file saving issues are displayed.

Verification Process:

- Compare program output with expected results for each test case.

- Inspect the content of `expenses.txt` after saving to ensure data is stored in the correct format.
- Check for memory leaks using appropriate tools if available (though the destructor aims to handle this).

7. Improvements and Extensions

While functional, the program has several areas for potential improvement and extension:

1. **Enhanced Input Validation:**
    - Implement robust date validation (format, valid day/month/year).
    - Validate category choices more strictly if needed, or allow user-defined categories.
    - Handle non-numeric input for amounts and menu choices more gracefully (currently, non-numeric input for `cin >> double` or `cin >> int` can lead to program errors or infinite loops if not handled).
2. **User Interface:**
    - Consider a more interactive or graphical user interface (GUI) instead of a console menu for better usability.
    - Improve console output formatting.
3. **Functionality:**
    - **Edit/Delete Records:** Allow users to modify or remove existing expense/income records.
    - **Reporting:** Generate more detailed reports (e.g., expenses by category over a period, comparison across months, charts).
    - **Filtering/Sorting:** Allow users to filter expenses by date range, category, or amount, and sort them.
    - **User-Defined Categories:** Allow users to add, edit, or delete categories.
    - **Currency Customization:** Allow the user to specify the currency symbol.
    - **Budgeting:** Incorporate features to set budgets for categories and track spending against them.
4. **Error Handling:**
    - More specific error messages.
    - Robust handling of file parsing errors in `loadFromFile` (e.g., if a line in `expenses.txt` is malformed).
5. **Code Structure:**

    - Separate class definitions and implementations into header (.h) and source (.cpp) files for better organization in larger projects.
    - Consider using standard library containers like `std::vector<Expense>` instead of raw dynamic arrays for easier memory management and more features.
6. **Configuration:**
    - Allow the user to specify the data filename instead of hardcoding `expenses.txt`.

8. Conclusion

The C++ Expense Tracker program provides a foundational set of features for personal finance management. It allows users to log expenses and income, view their financial records, and analyze monthly spending. The program demonstrates object-oriented principles through its `Expense` and `ExpenseTracker` classes and handles data persistence through file input/output. While functional, there is significant scope for enhancements in areas such as user interface, input validation, error handling, and advanced financial tracking features.