HW 4: `WALLET` - Developer Documentation

Miriam Gaus
LZ4LOZ

December 1, 2025

This document explains the design and implementation of the Wallet project.

1 Wallet - Overview

The Wallet program is structured into multiple modules: `WalletHandler`, `FileIO`, `InputHandler`, `DateHandler`, and `main`. It manages personal finance data such as income, expenses, categories, and currencies.

2 Modules

- **WalletHandler**: Manages data structures like Wallet, Category, and Entry, and related operations.
- **FileIO**: Handles reading from and writing to wallet database files.
- **InputHandler**: Provides the user interface, input handling, and data presentation.
- **DateHandler**: Supports date normalization and validation functions.
- **main**: Handles the main loop where databases can be loaded and the program terminated.

3 Data Structures

3.1 `typedef struct`

- `Wallet`: Contains collections of categories and entries. Therefore, the entries and categories are stored in 2D arrays.
  - `Entry **entries` Array of pointers to entries
  - `int size` Current number of entries
  - `Category **categories` Array of pointers to categories
  - `int categoriesSize` Current number of categories
- `Category`: Holds category name and total amount.
  - `name` Category name string
  - `total` Total amount in this category in Euro
- `Entry`: Contains date, category ID, amount, currency, and entry type (Income/Expense).
  - `struct tm date` Date of the entry
  - `entry_type type` Type of entry: Income or Expense
  - `char categoryId[50]` Identifier of the category for this entry
  - `double amount` Monetary amount of this entry
  - `char currency[4]` Currency code of the entry amount (e.g., "EUR")

3.2 enum

- `entry_type` Enum representing the type of an entry: Income or Expense.
  - `Income` Entry represents income
  - `Expense` Entry represents expense

4 Algorithms

Memory allocation is managed dynamically to efficiently accommodate the variable number of categories and entries:

- Upon reading the number of categories from the file, the program allocates memory for an array of pointers to `Category` structures using `malloc()`.
- Each individual `Category` struct is then allocated separately to store its name and total.
- Similarly, the number of entries is read, and memory is allocated dynamically for an array of pointers to `Entry` structs.
- Each `Entry` struct stores data including the normalized date, category identifier, monetary amount, currency code, and transaction type.

This dynamic allocation approach allows the program to handle wallets of arbitrary size without reserving excess memory. Memory is carefully managed to avoid leaks by freeing allocated structures when they are no longer needed.

After loading, users can perform various operations such as adding new entries or categories. Therefore, the necessary memory is allocated and then with `realloc` attached to the 2D array of entries or categories.

5 Function List

5.1 WalletHandler

`void printEntry(Entry entry)` Prints the details of a given Entry.

`isCategory(Wallet *wallet, char *category)` Checks if a category with the given name exists in the wallet.
return 1 if the category exists, 0 otherwise.

`isCurrency(char *currency)` Checks if the given currency code is valid/supported.
return 1 if currency is valid, 0 otherwise.

`double transformIntoCurrency(double amount, char *oldCurr, char *newCurr)` Converts the given amount from oldCurr currency to newCurr currency using the exchange rates.
return The converted amount in newCurr currency.

`void addAmountToCategory(double amount, char *categoryId, Wallet *wallet)` Adds the specified amount to the total of the category identified by categoryId in the wallet.

`void addEntry(Wallet *wallet, Entry *entry)` Adds an Entry to the Wallet.

`void addCategory(Wallet *wallet, Category *category)` Adds a Category to the Wallet.

`void freeWallet(Wallet *wallet)` Frees all dynamically allocated memory inside the Wallet structure, including entries and categories arrays.

5.2 FileIO

`int loadWalletFromFile(FILE *file, Wallet *wallet)` Reads wallet data from a file and populates the Wallet struct.
return 1 if successful, 0 on failure.

`void saveWalletToFile(FILE *file, Wallet *wallet)` Writes the Wallet data into a file.

`int openFile(char *path, Wallet *wallet)` Opens a wallet database file, loads its data into Wallet struct.
return 1 if file loaded successfully, 0 otherwise.

`int saveDb(char *path, Wallet *wallet)` Saves the Wallet data to a file at the specified path.
return 1 if successful save, 0 otherwise.

5.3 InputHandler

`void printLine()` Prints a horizontal line of dashes for UI separation.

`void enterEntry(Wallet *wallet)` Prompts user to enter a new entry and adds it to the Wallet.

`void enterCategory(Wallet *wallet)` Prompts user to enter a new category and adds it to the Wallet.

`void evaluateTotalDb(Wallet *wallet)` Evaluates the wallet's totals and highest expense category in a chosen currency.

`void displayTimePeriod(Wallet *wallet)` Displays income, expense, largest expense, and balance for each category during a date range.

`void handleInput(char *path, Wallet *wallet)` Loop to handle user menu input and perform wallet operations.

5.4 DateHandler

`int isDateValid(struct tm date)` Checks if a given date is valid according to the time library. The dates needs to normalized.
return 1 if the date is valid, 0 otherwise.

`int isDateBetween(struct tm date, struct tm start, struct tm end)` Checks if a date lies inclusively between a start and end date. The dates need to be normalized.
1 if date is in the range [start, end], 0 otherwise.

`struct tm normalize(int year, int month, int day)` Normalizes a date given in year, month, day format into a struct tm.
return A struct tm representing the normalized date.

5.5 main

`int main(void)` Program entry point. Allows loading wallet, running commands, and exiting.
return Program exit code.

6 Testing

Testing was performed by:

- Verifying file read/write functionality with valid and invalid files.
- Testing user input flows for adding entries and categories.
- Validating currency conversion and date range summaries.

- Checking error handling for incorrect inputs.

To test the `wallet.txt` file can be used. It is provided in the repository: https://github.com/MiriamGaus/Wallet. There are no automatic tests.