Rust Project

C3 – Personal Data Management- Manage Income/Expenses

01286120 Elementary Systems Programming

Software Engineering Program

Faculty of Engineering, KMITL

By

66010988 Cusson Laohapatanawong

Introduction

The 'C3 Personal Data Management' project designed to help you manage your income and expenses efficiently. You can record transactions, set budgets, export as HTML table for Big-picture visualization, and track your financial status. After understanding these options, you can use them to manage your personal finances effectively.

How to use

Use >>cargo run<< to initiate the program.

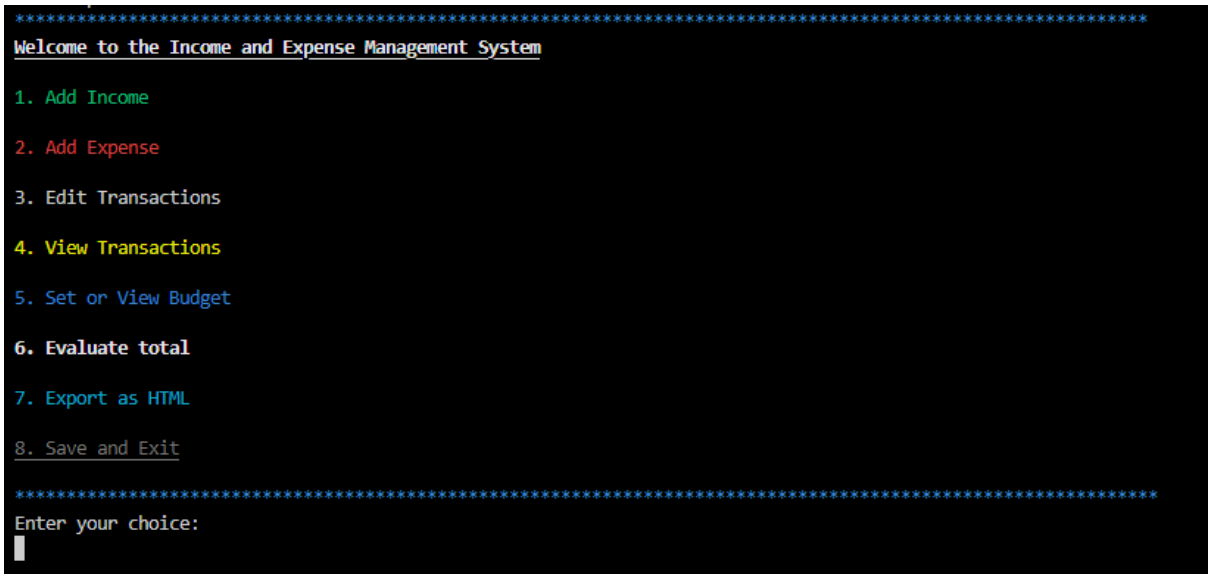


Figure 1 shows interface of the program when initiated

In this document, I will explain the basic usage of this program by going through all the numbers in Figure 1

1. Add Income

User will need to input:

- Transaction name:
- Transaction amount:
- Transaction date:
- Transaction category:

The program then pass all the inputs listed above and stores it in the *Transaction struct.

2. Add Expenses

User will need to input:

- Transaction name:
- Transaction amount:
- Transaction date:
- Transaction category:

The program then pass all the inputs listed above and stores it in the *Transaction struct.

3. Edit Transactions

User will be asked to choose whether to make an edit on a transaction or a budget (t/b). Either way, the selected choice will call the print function on the corresponding struct with index

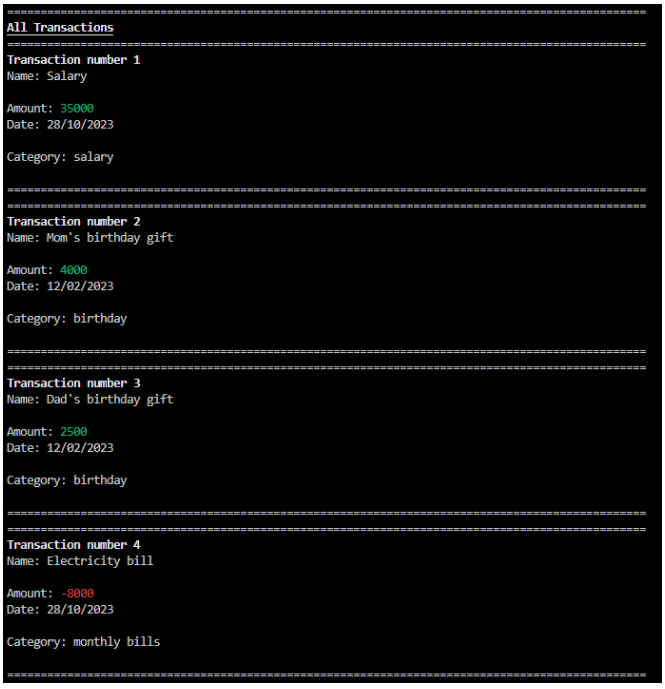


Figure 2.1 shows the output of a print function on a Transaction struct.

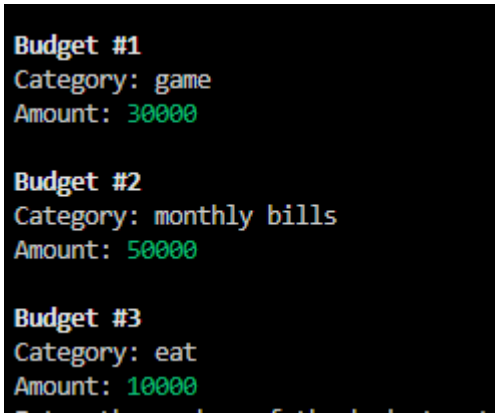


Figure 2.2 shows the output of a print function on a *Budget struct.

Afterward, the user is required to select the index of the transaction (or budget) they wish to modify. The chosen index will then be displayed separately. Subsequently, the user will be prompted to specify which element they want to alter, and the existing value will be substituted with the new one.

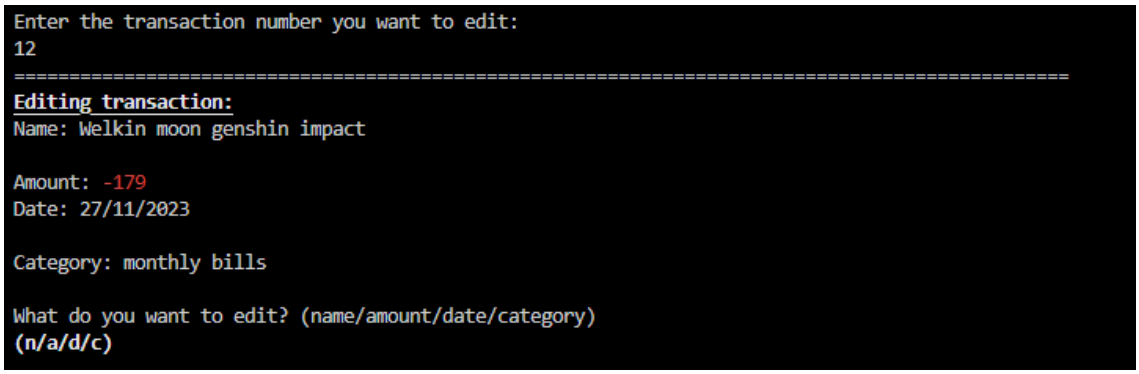


Figure 2.3 shows the chosen transaction index that are being displayed separately.

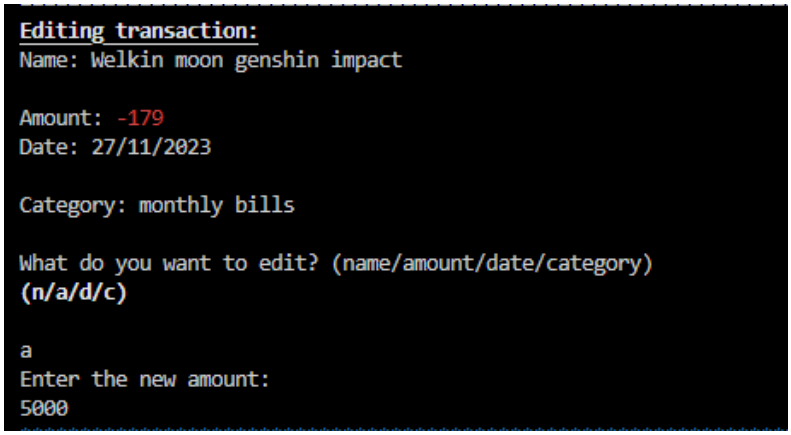


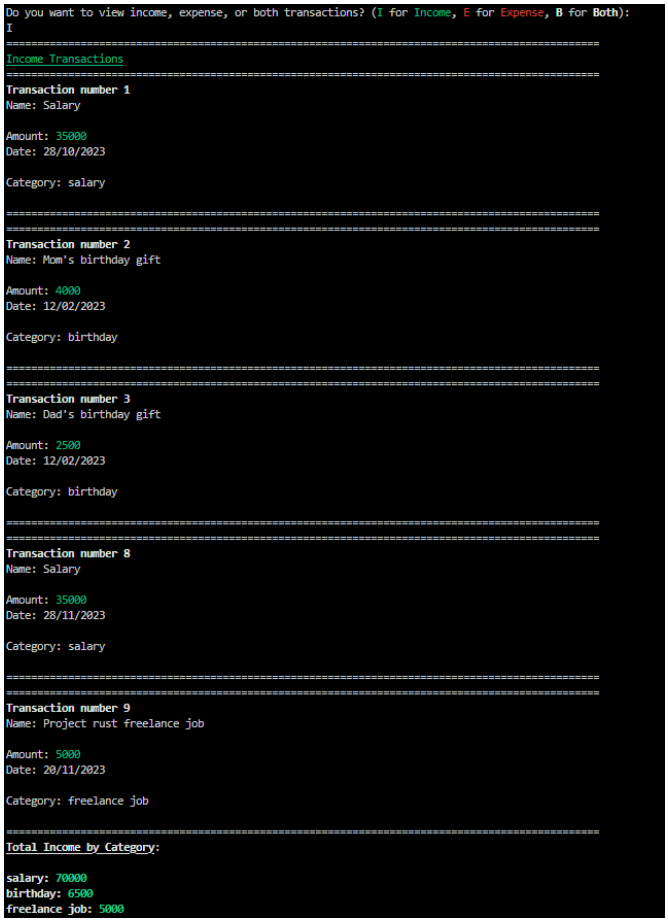
Figure 2.4 shows if the user choose to modify the amount to 5000.

Figure 2.5, the old value is replaced by the new value.

Note that same goes with budget editing as for transaction.

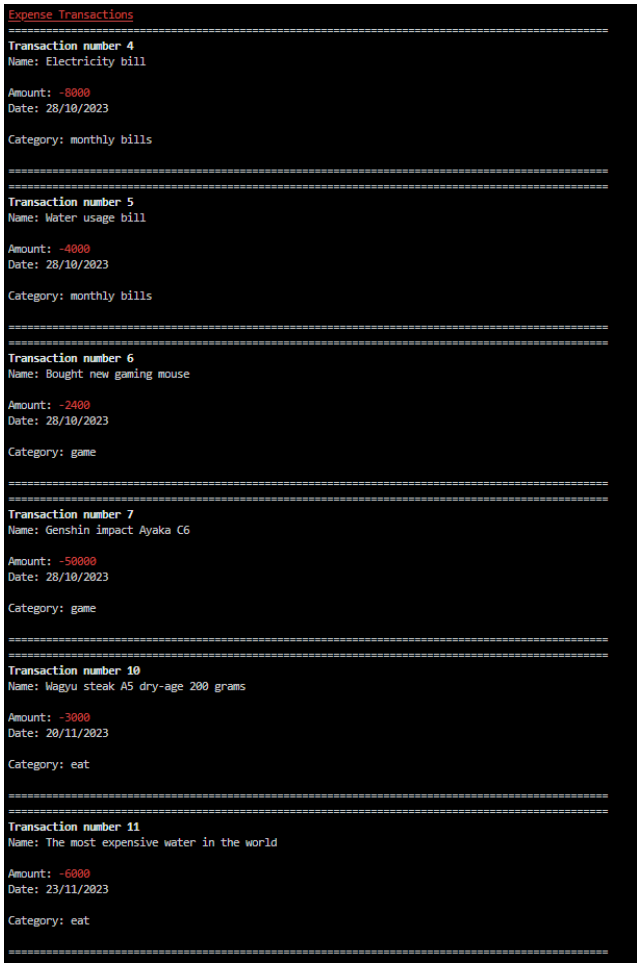
4.View Transactions

You'll be prompted to select whether you want to view income, expense, or both transactions (I for Income, E for Expense, B for Both).



For income, the program will display all income transactions and the sum of all income in each category.

Figure 4.1 shows all income transaction and the sum amount of each category



For expenses, the program will display all expense transactions and the remaining budget of the previously set budgets.

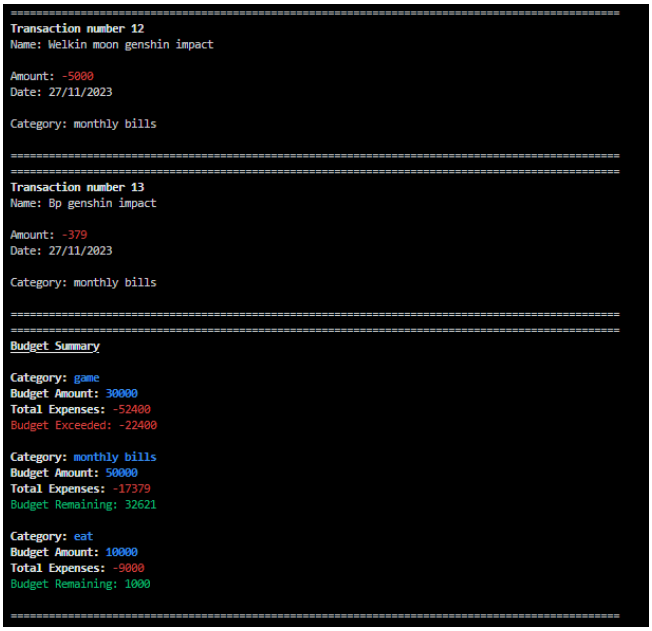
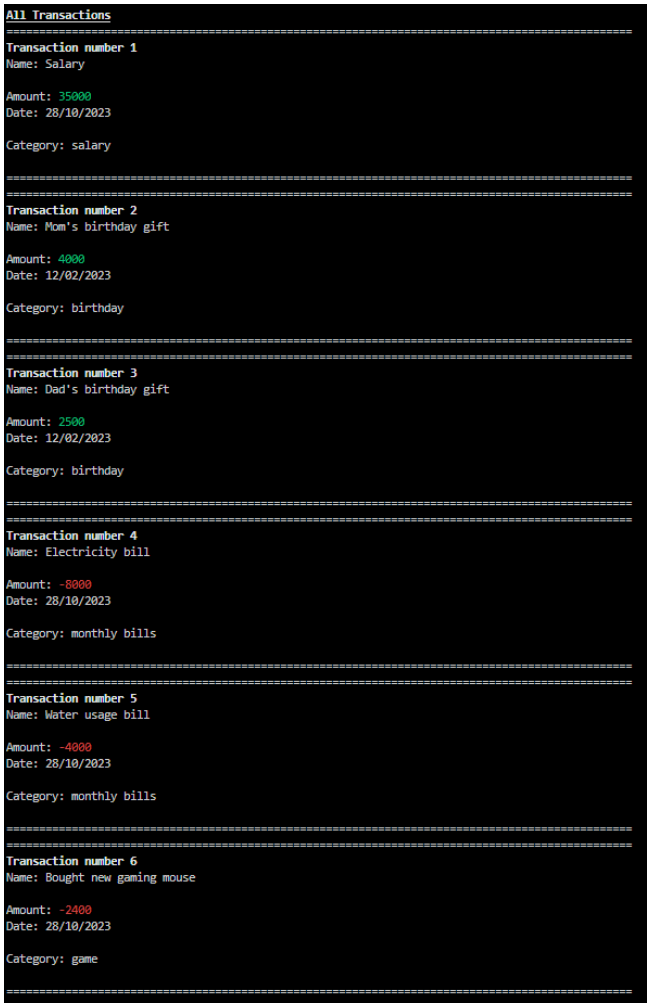


Figure 4.2 , 4.3 shows all expense transactions and remaining amount of the previously set budgets.



For both transactions, it will display both income and expenses, income summary and remaining budgets

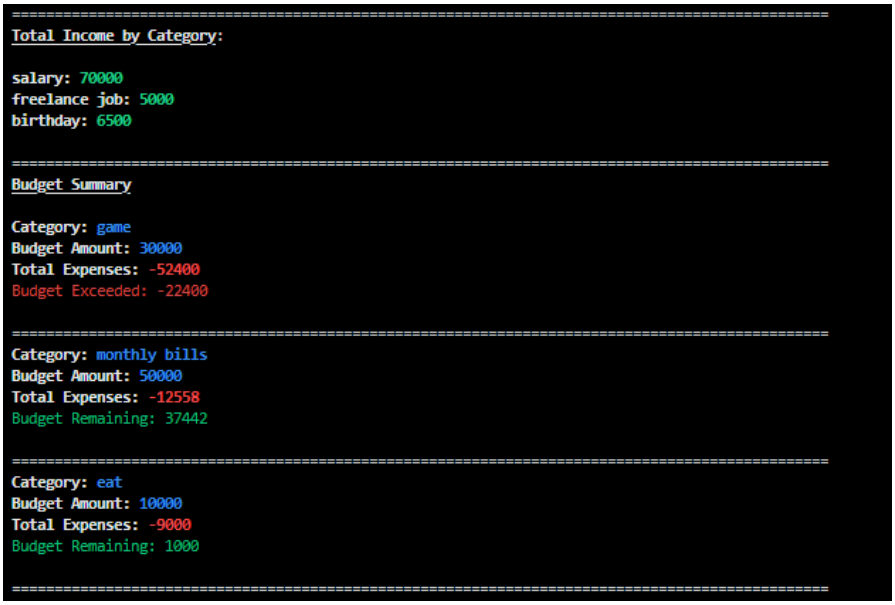


Figure 4.4 shows only parts of the data currently stored in a struct.

Figure 4.5 shows both income summary and remaining budgets.

5. Set or View Budget:

- You can set a budget for specific expense categories or view the existing budgets.
- To set a budget, you'll provide the category and the budget amount.
- The program will store these budgets in the *Budget struct.
- To view budgets, simply run >>5<< again and the program will display a list of categories and their budgeted amounts.

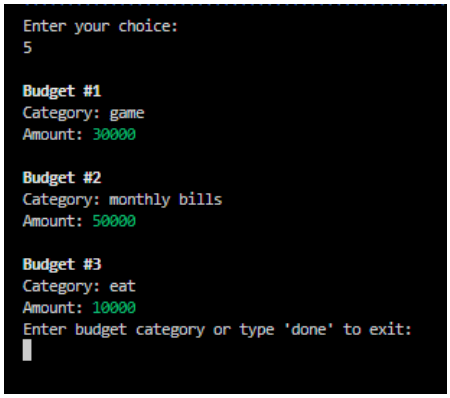


Figure 5.1 shows when user first initiate this feature.

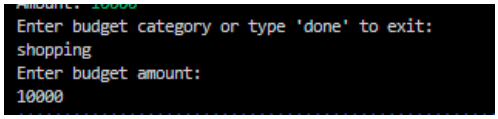


Figure 5.2 shows the example input.

Figure 5.3 shows the current information in the budgets struct.

Note: The following budgets category/amount will be matched with the expense category/amount, and the remaining budgets will be evaluated this way.

6.Evaluate Total:

- This option will provide a summary of all the information in brief.
- It will show the total income by category, followed by the total expense for each budget category.
- If a budget has been exceeded, it will be shown in red.
- The program will then display the total income (in green), total expenses (in red), and the net money.
- If your net money is positive, it will be displayed in green, if it's zero, in black, and if it's negative, in red.

Figure 6.1 shows the summary of all information in brief.

7. Export as HTML

The function exports financial transaction data as an HTML file. The user will be asked whether the files will be exported in "d" (day), "m" (month), or "y" (year).

- If "d" is selected:
  - It retrieves all transaction dates, sorts them, and lets the user choose a date to export.
  - It generates an HTML file containing transaction details for the selected date.

- If "m" is selected:
  - It retrieves unique transaction months, sorts them, and lets the user choose a month to export.
  - It generates an HTML file containing transaction details for the selected month.

- If "y" is selected:
  - It retrieves unique transaction years, sorts them, and lets the user choose a year to export.
  - It generates an HTML file containing transaction details for the selected year.

Common HTML elements like headers and table structure are added to the export data.

The function calculates the net income and expense for the selected time frame.

It appends the transaction data to the export data within the HTML table structure.

Finally, the export data is written to an HTML file named based on the chosen time frame.

(See next page for example execution)

```
Enter your choice:
7
Do you want to export by day, month, or year
(d/m/y) or e to exit

d
Available dates to choose from:
1: 12/02/2023
2: 20/11/2023
3: 23/11/2023
4: 27/11/2023
5: 28/10/2023
6: 28/11/2023
Enter the index of the date you want to export:

4
ayyo = export_27_11_2023.html
Data exported to export_27_11_2023.html
```

Figure 7.1 shows an example execution, suppose the user chooses day “d”, The program extract all the available dates as a choices, suppose that the user chooses 4, 27/11/2023, the file is then created by the name (“export_{}”, chosen_date) as a html table containing the corresponding transaction of the time frame.

Transactions for Date: 27/11/2023

Name	Amount	Date	Category
Welkin moon genshin impact	-179	27/11/2023	monthly bills
Bp genshin impact	-379	27/11/2023	monthly bills
Net amount (Total income - Total expense)	0 - 558 = -558		

Figure 7.2 shows the exported “export_27_11_2023”

For better visualization, I will also provide the transaction for year 2023 as “export_2023”

```
Enter your choice:
7
Do you want to export by day, month, or year
(d/m/y) or e to exit

y
Available years to choose from:
1: 2023
Enter the index of the year you want to export:

1
Data exported to export_2023.html
```

Figure 7.3 shows the input command if the user wants to export by year.

Transactions for Year: 2023

Name	Amount	Date	Category
Salary	35000	28/10/2023	salary
Mom's birthday gift	4000	12/02/2023	birthday
Dad's birthday gift	2500	12/02/2023	birthday
Electricity bill	-8000	28/10/2023	monthly bills
Water usage bill	-4000	28/10/2023	monthly bills
Bought new gaming mouse	-2400	28/10/2023	game
Genshin impact Ayaka C6	-50000	28/10/2023	game
Salary	35000	28/11/2023	salary
Project rust freelance job	5000	20/11/2023	freelance job
Wagyu steak A5 dry-age 200 grams	-3000	20/11/2023	eat
The most expensive water in the world	-6000	23/11/2023	eat
Welkin moon genshin impact	-179	27/11/2023	monthly bills
Bp genshin impact	-379	27/11/2023	monthly bills
Net amount (Total income - Total expense)	81500 - 73958 = 7542		

Figure 7.4 shows the exported “export_2023”.

8.Save and Exit:

This function is used to save your data and exit the program. It consists of saving data to a file and loading previous data.

- **For saving,** The *Appstate struct will be initiated when this function is called. If there is no previous saved data, the *User struct (transactions and budgets) is cloned and push into the *Appstate struct. If there exists the previous save file, It will clear the previous state (to avoid data duplication), update it with the current user data. Either way, the *Appstate struct will be written into a file named “state.json”. (See All * definition (P.14) for more details)
- **For loading,** The program will search for “state.json” file. If this file does not exist, the program will proceed to create a new one to store the data. If the program found the file, the program will read the data from it and load the previously saved transactions and budgets. This allows users to continue working with their existing data. (See All * definition (P.14) for more details)

Figure 8.1 shows the formatted example data within “state.json”. (The actual file won’t be formatted)

All * definition

- *Transaction struct – A struct containing all income and expense transaction define by the Figure *.1.

```
pub struct Transaction {
    pub count: u32,
    pub name: String,
    pub amount: f64,
    pub date: String,
    pub category: String,
    pub is_income: bool,
}
```

Figure *.1 shows all the fields within a Transaction struct.

- ❖ count – a field to indicates the order of the transaction including both income and expenses.
- ❖ name – the name of the transaction that user had inputted.
- ❖ amount – the amount of the transaction that user had inputted.
- ❖ date – the date of the transaction that user had inputted.
- ❖ category – the category of the transaction that user had inputted. The income summary is evaluated based on the matching name of the category.
- ❖ is_income – a bool to indicates whether the transaction are income or expense.

- *Budget struct – A struct containing all budgets define by Figure *.2.

```
pub struct Budget {
    pub count: u32,
    pub category: String,
    pub amount: f64,
}
```

Figure *.2 shows all the field within a Budget struct.

- ❖ count – a field to indicate the index of the budgets inputted by the user.
- ❖ category – a field that will be matched with the expense category to evaluate the remaining budgets.
- ❖ amount – the amount of budget money sets by the user input.

- *User struct – A struct containing Transaction struct and Budget Struct, act as a temporary struct before *Appstate struct.

```
#[derive(Serialize, Deserialize)]
2 implementations
pub struct User {
    pub transactions: Vec<Transaction>,
    pub budgets: Vec<Budget>,
}
```

Figure *.3 shows all the field within the User struct.

- *Appstate struct – A struct that is used to save to “state.json” file which contains a User struct. Appstate struct act as a most recent saves while the User struct takes the data from the Appstate struct for further modification from the user. If the user decided to call the function save_and_exit, The Appstate struct will clear its data (to avoid data duplication) and takes all the recently edited data from the User struct and writes it into “state.json”.

```rust
#[derive(Serialize, Deserialize)]
2 implementations
struct AppState {
    user: lib::User,
}
```

Figure *.4.1 shows the AppState struct.

```rust
let mut app_state: AppState = load_state().unwrap_or_else(op: |_| AppState {
    user: lib::User {
        transactions: vec![],
        budgets: vec![],
    },
});
if !app_state.user.transactions.is_empty() {
    user.transactions = app_state.user.transactions.clone();
}
if !app_state.user.budgets.is_empty() {
    user.budgets = app_state.user.budgets.clone();
}
```

Figure *.4.2 shows the process of loading the information from the “state.json” and the cloning of the User struct.

```rust
"8" => {
    app_state.user.transactions.clear();
    app_state.user.budgets.clear();
    app_state AppState
        .user User
        .transactions Vec<Transaction>
        .extend(iter: user.transactions.iter().cloned());
    app_state.user.budgets.extend(iter: user.budgets.iter().cloned());
    save_state(&app_state);
    println!("Goodbye!");
    break;
}
```

Figure *4.3 shows the method to avoid data duplication when the user called save_and_exit function and writes it into “state.json”.