



ASSIGNMENT 1

Assignment Due: May 31, before 11.59pm

"Education is what remains after one has forgotten what one has learned in school."

— Albert Einstein

Description:

In this assignment, you will gain hands-on experience with the C programming language. While you are not required to be a C expert to complete the work, you will certainly explore most of the things that we have discussed in class (and a few other things as well).

NOTE: While the assignment document is a bit long, describing the assignment itself is relatively straightforward. Most of the content comprises of helpful hints and advice intended to simplify the process of writing your first C program.

In this assignment, you'll develop a budget tracker system using C programming language. Budget tracking applications are tools designed to help individuals or households monitor their income, expenses, and the overall financial health. By logging transactions and categorizing spending, these applications provide insights into where money is going, support better financial planning, and help users stay within their budgets. Throughout this assignment, you will explore the fundamentals of C programming while applying your skills to design a budget tracking system. This application, of course, will be MUCH simpler.

Your application must display the welcome message, and a main menu as depicted in Fig.1 below. The application will have seven options, and you may assume a perfect user who will always enter an appropriate type of input, for instance an integer for number input.

```
Budget Tracking System
=====
1. Display all entries
2. Expense Distribution
3. Sort Entries
4. Add Income/Expense Entry
5. Modify Entry
6. Filter by Month
7. Exit
Choice: |
```

Figure 1. Main menu

COMP348: PRINCIPLES OF PROGRAMMING LANGUAGES

Implementation Specifications:

Detailed behavior of each menu option is mentioned below with screenshots.

Selecting Option 7 will, of course, exit the program and display a message to say, “Goodbye and thanks for using our budget tracker app”. If Options 1 through 6 are selected, on the other hand, various actions will be performed.

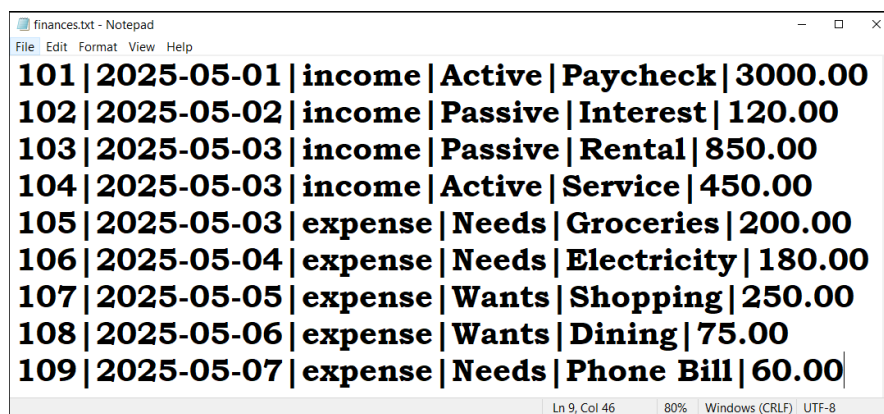
Before we go any further, note that the screen should always be cleared whenever the menu is (re)displayed. In other words, the text printed by your program should not simply keep scrolling down the screen. Doing this is quite easy – we simply use the system function found in `stdlib.h`. Specifically, `system(“clear”)` will clear the screen on a Linux system (e.g., the docker-based installation that the graders will be using.) On Windows, it’s `system(“cls”)` but this form will fail on docker/Linux.

In terms of the menu options, all will be operating on data read from an input file (*i.e.*, when the app begins). The input file is passed to your application via command line argument.

Suppose your program is compiled into a binary executable namely “budgetTracker” the following command is used to pass “finances.txt” as the input database file.

```
./budgetTracker finances.txt
```

As the name implies, it is a simple text file and contains a series of rows, each defining several fields. A sample of such a text file is listed below:



101	2025-05-01	income	Active	Paycheck	3000.00
102	2025-05-02	income	Passive	Interest	120.00
103	2025-05-03	income	Passive	Rental	850.00
104	2025-05-03	income	Active	Service	450.00
105	2025-05-03	expense	Needs	Groceries	200.00
106	2025-05-04	expense	Needs	Electricity	180.00
107	2025-05-05	expense	Wants	Shopping	250.00
108	2025-05-06	expense	Wants	Dining	75.00
109	2025-05-07	expense	Needs	Phone Bill	60.00

Figure 2. A sample “finances.txt” input file

Each line in the file represents one entry. The fields in each row are separated by pipe symbol (‘|’) and can be interpreted as follows:

<entry ID, date, entry type, entry subtype, entry description, amount>

Important: The input data is assumed to have NO errors. Each row is separated by a single newline character and has exactly six fields. The maximum possible lengths of all fields are known. All strings fields are no longer than 100 characters. In case the input file does not comply with the above specification, your program may simply display the following error message and exits abnormally (use standard error console, `stderr`).

COMP348: PRINCIPLES OF PROGRAMMING LANGUAGES

When the app begins, it will read this data before displaying the menu. You are free to store the data in memory any way that you like (i.e., data structures of your choice). Just to be clear, you will not be able to store it as a giant text string, as this will be useless when trying to perform the required operations.

1. Display all entries

Upon selecting the first option, a complete list of entries from the finances.txt will be printed as shown in figure 3.

```
Finances Summary
=====
```

ID	Date	Type	Category	Description	Amount
101	2025-05-01	income	Active	Paycheck	\$3000.00
102	2025-05-02	income	Passive	Interest	\$120.00
103	2025-05-03	income	Passive	Rental	\$850.00
104	2025-05-03	income	Active	Service	\$450.00
105	2025-05-03	expense	Needs	Groceries	\$200.00
106	2025-05-04	expense	Needs	Electricity	\$180.00
107	2025-05-05	expense	Wants	Shopping	\$250.00
108	2025-05-06	expense	Wants	Dining	\$75.00
109	2025-05-07	expense	Needs	Phone Bill	\$60.00

Figure 3. Display all entries

There are a couple of things to emphasize here:

1. We obviously have a neatly formatted column-oriented display. While the exact number of spaces between columns is not important, you can see that everything is lined up properly, making the data very easy to read. You must do something similar.
2. All columns have column headers (e.g., “ID”, “Date”, “Type”, etc.).

2. Expense Distribution

Upon selecting this option, a summary report for the given entries must be displayed (refer to figure 4). The program must print total income, total expenses, expenses separated into needs and wants, and their share in expenses as well as total income, and the net balance.

```
===== Expense Distribution Report =====
Total Income: $4420.00
Total Expenses: $765.00
Needs: $440.00 (57.52% of expenses, 9.95% of income)
Wants: $325.00 (42.48% of expenses, 7.35% of income)
Net Balance: $3655.00
=====
```

Figure 4. Expense distribution

3. Sort Entries

The default sorting order of the table is ascending order by entry ID. But that isn't always very useful. Therefore, we want to be able to modify the listing to reflect a few other useful sorting orders. As such, when we select Option 3, and provide our input, we will see the options shown in Figure 5.

```
Sort Menu
1. Sort by ID
2. Sort by Date
3. Sort by Amount
4. Sort by Description
Choice: 3
```

Figure 5. Sorting menu

If, for instance, we select 3 *i.e.* Amount as the new sort column, all the entries will be sorted by amount in ascending order (refer to Figure 6).

Finances Summary					
=====					
ID	Date	Type	Category	Description	Amount

109	2025-05-07	expense	Needs	Phone Bill	\$60.00
108	2025-05-06	expense	Wants	Dining	\$75.00
102	2025-05-02	income	Passive	Interest	\$120.00
106	2025-05-04	expense	Needs	Electricity	\$180.00
105	2025-05-03	expense	Needs	Groceries	\$200.00
107	2025-05-05	expense	Wants	Shopping	\$250.00
104	2025-05-03	income	Active	Service	\$450.00
103	2025-05-03	income	Passive	Rental	\$850.00
101	2025-05-01	income	Active	Paycheck	\$3000.00

Figure 6. Entries sorted by amount in ascending order

4. Add Income/Expense Entry

This option will allow a user to add an entry to the finances. Upon selecting this option, user will be asked to enter all the necessary values and the entry will be added with an incremented ID number. For instance, a sample add entry operation will look like this (refer to Figure 7).

```
Use today's date? (y/n): n
Enter date (YYYY-MM-DD): 2025-04-23
Type (income/expense): expense
Category: Wants
Description: Movie
Amount: $23.00
Entry added successfully with ID: 110
```

Figure 7. Add a new entry

You should provide an option to add today's date. You can use `localtime()` function from `time.h`. You may assume a perfect user and hence input validation is not required.

5. Modify Entry

Sometimes we must make updates. In this case, we want to update the amount for an entry, the prompt would look like Figure 8. Note that before asking the user for the new values, we first display the current contents. If we don't do this, it will be almost impossible to remember the entry IDs displayed on previous screens.

```

Finances Summary
=====
ID      Date      Type      Category  Description  Amount
-----
109    2025-05-07    expense   Needs     Phone Bill   $60.00
108    2025-05-06    expense   Wants     Dining       $75.00
102    2025-05-02    income    Passive   Interest     $120.00
106    2025-05-04    expense   Needs     Electricity   $180.00
105    2025-05-03    expense   Needs     Groceries    $200.00
107    2025-05-05    expense   Wants     Shopping     $250.00
104    2025-05-03    income    Active    Service      $450.00
103    2025-05-03    income    Passive   Rental       $850.00
101    2025-05-01    income    Active    Paycheck     $3000.00
110    2025-04-23    expense   Wants     Movie        $23.00

Enter ID of entry to modify: 108

Current Details:
ID: 108
Date: 2025-05-06
Type: expense
Category: Wants
Description: Dining
Amount: $75.00

What would you like to modify?
1. Date
2. Amount
Choice: 2
Enter new amount: $77.00
Entry updated successfully.
    
```

Figure 8. Modify an entry

In this case, we have selected ID 108 and have changed the amount from \$75.00 to \$77.00. When we go back to the menu and display all entries again, we should see the change.

6. Filter by Month

Filtering the transactions by month can help us monitor monthly transactions. Upon selecting this option, the user will be asked year and month and only the records matching this criterion will be displayed (refer to Figure 9).

```

Enter year (YYYY): 2025
Enter month (1-12): 4

Entries for 2025-04:
ID      Date      Type      Category  Description  Amount
-----
110    2025-04-23    expense   Wants     Movie        $23.00
    
```

Figure 9. Filter by month

COMP348: PRINCIPLES OF PROGRAMMING LANGUAGES

Extra Credit (10% Points)

A more elaborate application with options like

1. Visual Expense Breakdown: Print a basic text-based bar chart (e.g., using * characters) showing how income is distributed across different expense subtypes.
2. Transaction Search Feature: Allow the user to search for transactions by amount range.
3. Undo Last Action: A basic undo feature to reverse the most recent addition or update.

Evaluation Procedure

Please note that the markers will be using a standard Linux system, likely the same docker installation that most of you should be using. Your code **MUST** compile and run from the Linux command line. If you are using an IDE (e.g., Eclipse or VS Code), rather than a simple editor, you will want to test execution from the command line, since IDEs often create their own paths, folders, environment variables that might prevent the code from running on the command line.

To evaluate the submissions, the marker will simply create a test folder and add a sample input data file plus your source files. Your code will be compiled and run and each of the menu options will be tested. Every student will be graded the same way. Note that you are expected to deallocate any memory that you have allocated during the running of the application. Point value will be deducted if you have memory leaks. The markers will be given a simple spreadsheet that lists the various criteria described above.

There are no mystery requirements. Please note that it is better to have a working version of a slightly restricted program than a non-working version of something that tries to do everything (e.g., you might be able to display the entries, but you can't sort it, or you can display the existing entries, but you can't add/modify any). Hence, make sure that your code compiles and runs. It is virtually impossible to grade an assignment that does not run at all.

DELIVERABLES: Your submission should have multiple source files. You would have the following source files: `main.c` (the main function and the basic GUI), `ordering.c` (any code related to sorting the entries), `data.c` (any code used to read the input file and extract the fields from each row), and `budget.c` (the code used to carry out the logic of the menu options – this will definitely be the biggest file). Each of these may have an associated header file of the same name (with a `.h` extension). The header files must list one or more publicly accessible functions. These `.c` and `.h` files represent the only code that you will submit.

IMPORTANT 1: Basic error checking is required. For example, if you are updating an entry and you use an entry ID that doesn't exist, the program should not crash. As a second example, you should not be able to update an amount with a negative value (e.g. `-23.00`). This makes no sense.

IMPORTANT 2: All files (`.c` and `.h`) must be prepared/saved in the same folder so that they can be compiled from the command line using the following simple gcc command

```
gcc -o budgetTracker main.c budget.c data.c ordering.c
```

You can add the `-Wall -g -gdwarf-4` options during development so that you can test and profile your code. Note that you may include a `README` file if some of the application's functionality is not complete. This will allow the grader to give you value for the parts that are working, instead of assuming that nothing is working properly.

COMP348: PRINCIPLES OF PROGRAMMING LANGUAGES

The grades will use the following command to run your application:

```
./budgetTracker <<.../path-to-the-input-database-file>>
```

Note that the input file will be passed to your program as command line argument. Do not make other assumptions.

Once you are ready to submit, compress all .c/.h/README files (and ONLY these files) into a zip file. The name of the zip file will consist of "a1" + last name + first name + student ID + ".zip", using the underscore character "_" as the separator. For example, if your name is John Smith and your ID is "123456", then your zip file would be combined into a file called a1_Smith_John_123456.zip". The final zip file will be submitted through the course web site on Moodle. You simply upload the file using the link on the assignment web page. Please note that it is your responsibility to check that the proper files have been uploaded. No additional or updated files will be accepted after the deadlines. You cannot say that you accidentally submitted an "early version" to Moodle.

THIS ASSIGNMENT MUST BE SUBMITTED INDIVIDUALLY.

HINTS:

1. Use a struct to represent entry for better organization.
2. Implement separate functions for each functionality.
3. Properly separate declarations and definitions.
4. Include error-handling for file operations (e.g., opening, reading, and writing files).
5. Encourage modular programming and code reusability.
6. Use of string arrays inside structure is allowed.
7. While the row size is limited, there is no limit on the number of rows in the database file. Use dynamic memory allocation.
8. Make sure you properly free up any dynamic memory that you allocate.

PITFALLS:

1. Be mindful of file handling errors; check for file existence before reading.
2. Avoid hardcoded values where possible; use constants for better maintainability.
3. Encourage modular programming and code reusability.

Considerations for the Assignment

When tackling this assignment, it's crucial to recognize that if you were implementing this application in a language like Python, the task would be relatively straightforward. Python offers convenient functions simplifying the development process.

However, in C, complexity increases. I want you to focus on understanding the structure of C and its distinctive features, without getting entangled in obscure issues or common pitfalls. Here are some guidelines to navigate through the intricacies:

- **Start Small:** Begin with manageable tasks. For instance, you might start by displaying a simple menu, then progress to reading input files, then saving them, etc. This can be done on Day 1.
- **Frequent Compilation:** Compile your code often to identify what's working and address current issues promptly. Delaying a resolution is not a good idea.
- **Library Function Reference:** Refer to cplusplus.com for information on standard C library functions, including clear examples of their usage.

COMP348: PRINCIPLES OF PROGRAMMING LANGUAGES

- **Segfaults and Pointers:** When dealing with segmentation faults due to pointer issues, avoid relying solely on print statements. Instead, leverage valgrind included with the Docker image for effective debugging.
- **Simplicity in GUI:** The GUI has been kept simple. Utilize the scanf function for reading keyboard input, and be mindful of newline characters when capturing user input.
- **#define for Constants:** Use #define for fixed values or constants used throughout the program.
- **Sorting:** In terms of sorting, you will use the qsort library function included in the standard libraries. The syntax can be confusing for those who are not used to C. Basic idea is that qsort is a “generic” library function that sorts arrays of generic elements, based upon a comparison function provided by the programmer. As parameters, qsort takes (1) the array to be sorted, (2) the numbers of elements in the array, (3) the size (in bytes) of each element, and (4) the name of the comparison function. In turn, the comparison function takes two parameters: void pointers to any two array elements that qsort will compare during the sort. qsort uses “void” pointers since it doesn’t know *a priori* what you want to sort. So, typically, the programmer casts the pointer to the right type (e.g., void* becomes int*) and then dereferences the pointer to get the element itself. The two elements can then be directly compared for equality. A quick look at a trivial qsort example will show these mechanisms in practice.
- **Reading and Writing Files:** Reading input files involves using fopen in read mode ("r") and fgets/fscanf to read lines as strings. When writing output files, use fopen in write text ("wt") mode and fputs/fprintf to write data to the text files.
- **Pointer Usage:** Expect to use pointers. Be very careful with pointer operations, like declarations, dereferencing, *etc.* Note that a variable holding a pointer is typically declared as int *foo. Here, foo is a pointer to an int (or possibly an array of ints). So, foo[2] could be used to access third int in such an array. It is also possible to de-reference a pointer, which means that we are getting the data that the pointer references. So, syntax like int x = *foo implies that the int pointed to by foo will now be assigned to x.
- **Global Variables:** While global variables are generally avoided, they may be convenient in this application. Declare shared global variables appropriately across multiple source files. When multiple source files are being used, the compiler must be told that a shared global variable is defined somewhere else. For example, if file1.c contains a global variable defined as int foo = 4; then file2.c would include a declaration like extern int foo. This tells the compiler that foo variable declared in file2.c is not a new variable with the same name (which would produce a compiler error) but is just a reference to the foo variable already defined and initialized in file1.cbs
- **Header Files:** When multiple source files are used, header files should be employed to provide prototype information for the compiler. Never include variables or function implementations in a header file. NEVER! They are primarily used for function prototypes, constants, and things like struct definitions.
- **Dynamic Memory Allocation:** Use malloc for dynamically creating data, and free for deallocating memory. Ensure proper matching of malloc and free. When de-allocating memory, you will freely function. To be clear, only use free with a matching malloc call. So, a char array declared as char *buffer = (char*) malloc (10) could later be deallocated with free(buffer). NEVER try to deallocate a char buffer (or any array) defined like this: char buffer [10]. malloc was not used to create this and its memory is fully managed by the language environment.

COMP348: PRINCIPLES OF PROGRAMMING LANGUAGES

- **Structs and Pointers:** Understand the distinction when working with structs and pointers. Be cautious when dealing with dynamically allocated memory for structs. If a struct is defined as `struct foo {int x; int y;};` then we can declare a variable of this type as `struct foo bar`. We can now use `bar.x` or `bar.y` to access the internal values. But if we use `malloc` to dynamically create space for a new struct foo value, like `struct foo *bar = (struct foo *)malloc(sizeof(struct foo))`, then we must use the syntax `bar->x` or `bar->y` to access the value since we are now using a pointer to the foo struct.
- **Data Parsing:** To extract the fields from each input row, you will use the `strtok` function in the standard string library. One simply uses a trivial while loop to separate the string on the `|` delimiter. There are lots of simple examples of its use.
- **Pointers to Pointers:** Recognize the use of pointers to pointers, especially when creating arrays of strings dynamically. Using pointers is crucial in C but it can be confusing at first. A couple of things to keep in mind: If we use `malloc` to create an array (e.g., of ints or chars), we can just use the standard array syntax (e.g., `foo[4]`) to access the elements of the array. The C compiler already knows that the array variable is a pointer. It is also possible to have pointers to pointers. So, if we use `malloc` to create an array of strings called `foo`, this would consist of a pointer to the main array, which would then contain pointers for the strings. So, `foo` would be defined as `char **foo` (or `char* *foo`). In other words, `foo` is pointer to an array of string pointers. This can be confusing at first and can lead to lots of unexpected segmentation faults if you are referencing the wrong pointer. Moreover, if you are deallocating this memory, you must be REALLY careful. Specifically, the strings must be deallocated first, and then the main array. And, just to be clear, freeing the main array does not automatically free the memory used by the strings.

This comprehensive guide aims to help you navigate the challenges of implementing the movie recommendation system in C. Follow these principles to build a robust and well-structured solution.

Use of POD

Attend POD sessions. There are 6 POD sessions that are available every day. The PODs can help with any programming questions. The summer session is very short, and the deadline approaches fast. Plan accordingly.

IMPORTANT!!!

Markers will receive a straightforward spreadsheet containing the specified criteria outlined earlier. There are no undisclosed or hidden requirements. It is emphasized that having a functional version of a somewhat limited program is preferable to having a non-functional version attempting to cover every aspect. Therefore, prioritize ensuring that your code can be successfully compiled and executed. Evaluating an assignment that doesn't run at all is virtually impossible.

COMP348: PRINCIPLES OF PROGRAMMING LANGUAGES

Sample Run

Budget Tracking System

=====

1. Display all entries
 2. Expense Distribution
 3. Sort Entries
 4. Add Income/Expense Entry
 5. Modify Entry
 6. Filter by Month
 7. Exit
- Choice: 1

Finances Summary

=====

ID	Date	Type	Category	Description	Amount

101	2025-05-01	income	Active	Paycheck	\$3000.00
102	2025-05-02	income	Passive	Interest	\$120.00
103	2025-05-03	income	Passive	Rental	\$850.00
104	2025-05-03	income	Active	Service	\$450.00
105	2025-05-03	expense	Needs	Groceries	\$200.00
106	2025-05-04	expense	Needs	Electricity	\$180.00
107	2025-05-05	expense	Wants	Shopping	\$250.00
108	2025-05-06	expense	Wants	Dining	\$75.00
109	2025-05-07	expense	Needs	Phone Bill	\$60.00

Budget Tracking System

=====

1. Display all entries
 2. Expense Distribution
 3. Sort Entries
 4. Add Income/Expense Entry
 5. Modify Entry
 6. Filter by Month
 7. Exit
- Choice: 2

===== Expense Distribution Report =====

Total Income: \$4420.00

Total Expenses: \$765.00

Needs: \$440.00 (57.52% of expenses, 9.95% of income)

Wants: \$325.00 (42.48% of expenses, 7.35% of income)

Net Balance: \$3655.00

=====

Budget Tracking System

=====

1. Display all entries
 2. Expense Distribution
 3. Sort Entries
 4. Add Income/Expense Entry
 5. Modify Entry
 6. Filter by Month
 7. Exit
- Choice: 3

COMP348: PRINCIPLES OF PROGRAMMING LANGUAGES

Sort Menu

1. Sort by ID
2. Sort by Date
3. Sort by Amount
4. Sort by Description

Choice: 3

Entries sorted by amount.

Finances Summary

=====

ID	Date	Type	Category	Description	Amount

109	2025-05-07	expense	Needs	Phone Bill	\$60.00
108	2025-05-06	expense	Wants	Dining	\$75.00
102	2025-05-02	income	Passive	Interest	\$120.00
106	2025-05-04	expense	Needs	Electricity	\$180.00
105	2025-05-03	expense	Needs	Groceries	\$200.00
107	2025-05-05	expense	Wants	Shopping	\$250.00
104	2025-05-03	income	Active	Service	\$450.00
103	2025-05-03	income	Passive	Rental	\$850.00
101	2025-05-01	income	Active	Paycheck	\$3000.00

Budget Tracking System

=====

1. Display all entries
2. Expense Distribution
3. Sort Entries
4. Add Income/Expense Entry
5. Modify Entry
6. Filter by Month
7. Exit

Choice: 4

Use today's date? (y/n): n

Enter date (YYYY-MM-DD): 2025-04-23

Type (income/expense): expense

Category: Wants

Description: Movie

Amount: \$23.00

Entry added successfully with ID: 110

Budget Tracking System

=====

1. Display all entries
2. Expense Distribution
3. Sort Entries
4. Add Income/Expense Entry
5. Modify Entry
6. Filter by Month
7. Exit

Choice: 5

COMP348: PRINCIPLES OF PROGRAMMING LANGUAGES

Finances Summary

=====

ID	Date	Type	Category	Description	Amount

109	2025-05-07	expense	Needs	Phone Bill	\$60.00
108	2025-05-06	expense	Wants	Dining	\$75.00
102	2025-05-02	income	Passive	Interest	\$120.00
106	2025-05-04	expense	Needs	Electricity	\$180.00
105	2025-05-03	expense	Needs	Groceries	\$200.00
107	2025-05-05	expense	Wants	Shopping	\$250.00
104	2025-05-03	income	Active	Service	\$450.00
103	2025-05-03	income	Passive	Rental	\$850.00
101	2025-05-01	income	Active	Paycheck	\$3000.00
110	2025-04-23	expense	Wants	Movie	\$23.00

Enter ID of entry to modify: 108

Current Details:

ID: 108

Date: 2025-05-06

Type: expense

Category: Wants

Description: Dining

Amount: \$75.00

What would you like to modify?

1. Date

2. Amount

Choice: 2

Enter new amount: \$77.00

Entry updated successfully.

Budget Tracking System

=====

1. Display all entries

2. Expense Distribution

3. Sort Entries

4. Add Income/Expense Entry

5. Modify Entry

6. Filter by Month

7. Exit

Choice: 1

Finances Summary

=====

ID	Date	Type	Category	Description	Amount

109	2025-05-07	expense	Needs	Phone Bill	\$60.00
108	2025-05-06	expense	Wants	Dining	\$77.00
102	2025-05-02	income	Passive	Interest	\$120.00
106	2025-05-04	expense	Needs	Electricity	\$180.00
105	2025-05-03	expense	Needs	Groceries	\$200.00
107	2025-05-05	expense	Wants	Shopping	\$250.00
104	2025-05-03	income	Active	Service	\$450.00
103	2025-05-03	income	Passive	Rental	\$850.00
101	2025-05-01	income	Active	Paycheck	\$3000.00
110	2025-04-23	expense	Wants	Movie	\$23.00

COMP348: PRINCIPLES OF PROGRAMMING LANGUAGES

Budget Tracking System

=====

1. Display all entries
2. Expense Distribution
3. Sort Entries
4. Add Income/Expense Entry
5. Modify Entry
6. Filter by Month
7. Exit

Choice: 6

Enter year (YYYY): 2025

Enter month (1-12): 4

Entries for 2025-04:

ID	Date	Type	Category	Description	Amount
110	2025-04-23	expense	Wants	Movie	\$23.00

Budget Tracking System

=====

1. Display all entries
2. Expense Distribution
3. Sort Entries
4. Add Income/Expense Entry
5. Modify Entry
6. Filter by Month
7. Exit

Choice: 7

Goodbye and thanks for using our budget tracker app!!!