⬅️ | HOME

Group Project: Demonstration of Expense Tracker Implementations

Date: April 20, 2025

- **Term:** Spring 2025 Second Bi-term
- **Class:** Advanced Programming Languages -MSCS-632-B01
- **Instructor:** Dr. Dax Bradley
- **University of the Cumberlands**

This report demonstration two implementations of an Expense Tracker Application—one in C++ and one in Python. Both versions support the same core functions: adding expenses, filtering expenses by date or category, and displaying expense summaries. However, they leverage language-specific features that have different impacts on design, performance, and readability.

The GitHub repository, https://github.com/baralsamrat/MSCS632_Project_Group, contains the source code for the C++ and Python expense tracker applications, along with the metrics analysis script and this report. The repository's commit history reflects the individual contributions of each team member.

REPORT : - 📒 PDF

Team Contributions: Group 1

Name	Email	Contributions
Prasanna Adhikari	padhikari34605@ucumberlands.edu	Led the C++ implementation, focusing on memory management with smart pointers and efficient use of STL containers. Video Demonstartion of the Application
Shashwat Baral	sbaral29114@ucumberlands.edu	Developed the Python implementation, emphasizing the use of dynamic typing and the simplicity of Python's data structures.
Samrat Baral	sbaral30064@ucumberlands.edu	Implemented the metrics and visualization components, providing side-by-side comparisons of code metrics between the two languages. Created PowerPoint
Sahithi Bontha	sbontha35464@ucumberlands.edu	Coordinated integration, documentation, and testing, ensuring all contributions were reflected in the GitHub repository.

Video Demonstration

- Powerpoint Presentaion - PDF Expense_Tracker_Presentation.pdf
- Demonstration Video Presentaion https://somup.com/cTfq1As50K

Demonstration

Now it's time to see both versions of the Expense Tracker running. For this demonstration, I'll use a shell script provided in our project that automates running the C++ and Python programs sequentially. This script first compiles the C++ code and executes it, then sets up a Python virtual environment and runs the Python script (as well as the optional analysis tools). By doing this, we can ensure both implementations run with the same sample data and we can observe their outputs one after the other.

Figure: Running the unified demo script compiles and executes the C++ tracker (top output) and then runs the Python tracker (bottom output), followed by a metrics analysis. The console output shows filtered expenses and summaries from each implementation, which are nearly identical.

Running the C++ Tracker: After compiling the C++ program (using `g++` with C++14 standard), the script runs the C++ expense tracker. On the terminal, we see the output of the C++ program. It first prints a heading **"Running C++ Expense Tracker:"** to indicate the C++ output section. Then we see:

- **Filtered by Category (Food):** Under this, the program lists all expenses in the "Food" category. In our sample data, there are two such expenses. The output shows two lines: one for the expense on `2023-10-10` with amount `$50.00` (Food – Lunch at a restaurant), and one for `2023-10-20` with amount `$20.00` (Food – Snacks). The table columns are nicely aligned. If you look at the figure, in the top half under C++, you can see those two entries listed under the "Food" filter output. This confirms that the C++ filtering by category works correctly, picking up only the relevant entries.
- **Filtered by Date (2023-10-10 to 2023-10-15):** Next, the C++ program outputs a section for date filtering. It prints all expenses between October 10, 2023 and October 15, 2023 inclusive. According to our data, that range would include the expenses on 2023-10-10 and 2023-10-15. Indeed, the output shows two entries: one on `2023-10-10` ($50.00, Food) and one on `2023-10-15` ($30.00, Transport – Bus fare). Expenses outside this range (like the one on 2023-10-20) are not shown. This demonstrates the date filter logic is working. The output format again is a table with those entries. The C++ code used string comparisons to filter these, but as a user observing the output, it looks seamless and correct.
- **Expense Summary:** Finally, the C++ program prints the summary of expenses. In the output, we see a section labeled **"Expense Summary:"** followed by the totals per category and overall total. It shows **Food: $70.00**, **Transport: $30.00**, and **Total Expenses: $100.00**. These numbers correspond to the sum of the two food expenses (50 + 20 = 70) and the one transport expense (30), with a grand total of 100.00. The formatting is to two decimal places, giving it a professional look. This confirms that our summary function aggregated the data correctly. After this, the C++ program ends.

All of the above C++ results appeared in the terminal in one go, since the sample data was hardcoded and processed immediately. If this were an interactive program, we could have entered expenses and commands to filter, but here it's scripted for demonstration.

Running the Python Tracker: Immediately after the C++ output, the script proceeds to run the Python version. It activates the Python environment (ensuring required libraries like matplotlib and numpy are installed, though those are more for the visualization step) and then executes `expense_tracker.py`. The terminal then shows the output of the **Python expense tracker**, labeled by the script as **"Running Python Expense Tracker:"**. We observe that the Python output is very similar to the C++ output, which is great because it means both implementations are producing the same results for the same data. Specifically:

- The Python output begins with **"Filtered by Category (Food):"** and lists the expenses in the Food category. Just like the C++ version, it shows the 2023-10-10 expense ($50.00 Food – Lunch at a restaurant) and the 2023-10-20 expense ($20.00 Food – Snacks). These are formatted in a table with columns. You might notice in the figure (bottom half for Python output) that the amounts are displayed as `$50.00` and `$20.00` (with two decimal places). Our Python code formatted the output to always show two decimals, whereas the C++ printed `$50` and `$20` without the trailing `.00` in the category filter section. This is a minor formatting difference, but it doesn't affect the content of the information.

- Next, the Python output shows **"Filtered by Date (2023-10-10 to 2023-10-15):"**. Under this, it lists the expenses that fall in that date range, which again are the entries on 2023-10-10 and 2023-10-15. We see those two lines in the Python output as well (one for the $50 Food expense on the 10th, and one for the $30 Transport expense on the 15th). This confirms the Python date filtering worked. Internally, Python used actual date objects to compare, but as a user we simply see the correct filtered list printed out.

- Finally, the Python script prints the **Expense Summary**. It shows the same totals: **Food: $70.00**, **Transport: $30.00**, **Total Expenses: $100.00**. The formatting is virtually identical to the C++ summary (both show two decimal places here). This part of the output verifies that the Python aggregation of totals matches the C++ aggregation, which we would expect since they processed the same data.

At this point, we have seen both the C++ and Python implementations produce the same outputs. Despite differences in code structure and language syntax, the functionality for the end-user is consistent. This is an important validation for our project: it means we didn't inadvertently add or lose any features when porting the logic from one language to the other.

Metrics Analysis and Visualization: In addition to the core application, our project includes a small metrics analysis component (which I developed) to compare the code of the two implementations. After running the Python tracker, the script automatically runs `metrics.py` (our analysis script) and then `visualize.py`. The metrics script counts things like total lines of code and number of function definitions in each codebase. The console output (visible in the figure above after the Python results) shows, for example, that the **C++ code has about 96 total lines**, with **75 lines of actual code (excluding blanks/comments)** and **7 function definitions**, whereas the **Python code has about 60 total lines**, with **48 lines of code** and **5 function definitions**. This quantitative comparison highlights that the Python implementation is shorter in terms of code (which matches our expectation that Python is more concise). To make this comparison more visual, the `visualize.py` script then generates a bar chart showing these metrics side by side.

image

Figure: A bar chart generated by our visualization script, comparing code metrics between the C++ and Python implementations. The blue bars show the total number of lines of code (C++ has more lines than Python), and the orange bars show the number of function definitions in each (C++ slightly more than Python). This illustrates the difference in code verbosity and structure between the two languages for the same application.

Looking at the chart above, you can see the **Code Metrics Comparison** for C++ vs Python. The C++ bar on the left is taller for total lines of code, indicating the C++ version required more lines (due to extra headers, declarations, and verbose syntax). The Python bar is shorter, reflecting fewer lines needed. The function count (orange bars) difference is small – C++ has a couple more functions (we wrote separate functions for each task, whereas Python’s simpler syntax meant slightly fewer distinct function definitions were needed). This visual reinforces the idea that Python was more compact, though both were reasonably small programs. It’s a nice way to wrap up our demonstration by not only showing that the applications work, but also analyzing how they were built.