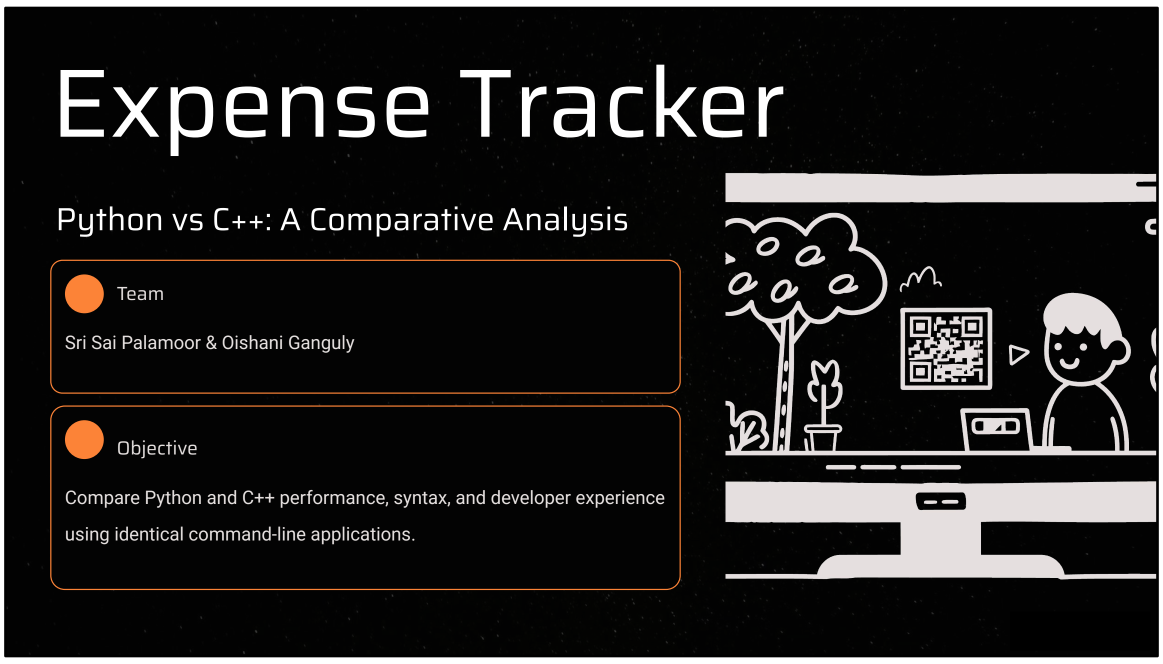


Expense Tracker

Python vs C++: A Comparative Analysis

Team

Sri Sai Palamoor & Oishani Ganguly

Objective

Compare Python and C++ performance, syntax, and developer experience using identical command-line applications.

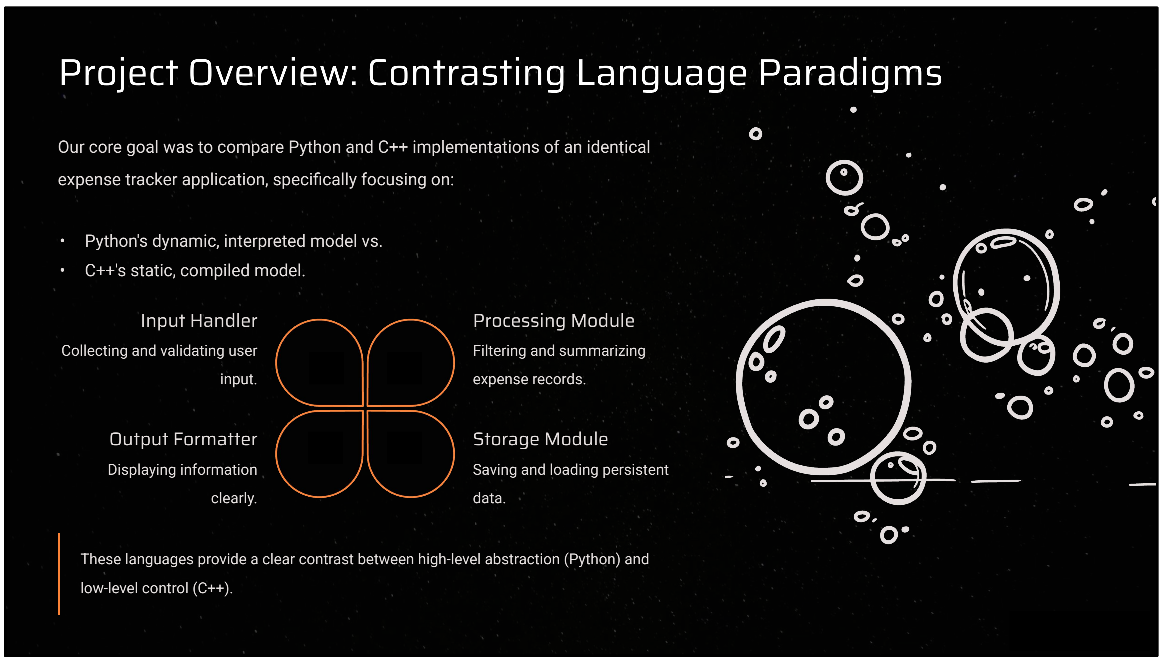
Good morning, everyone!

We're Sri Sai Palamoor and Oishani Ganguly, and today we'll be presenting our comparative analysis of Python and C++ implementations for an identical Expense Tracker application.

The goal of this project was to evaluate how language paradigms influence software design, syntax, performance, and developer experience.

We built two command-line applications with the exact same features, one in Python and one in C++, and then analyzed their differences in implementation and behavior.

We specifically avoided building out a GUI to keep the focus on the language characteristics. So while a CLI app for this is not ideal, it achieves the goal of this project!



Project Overview: Contrasting Language Paradigms

Our core goal was to compare Python and C++ implementations of an identical expense tracker application, specifically focusing on:

- Python's dynamic, interpreted model vs.
- C++'s static, compiled model.

Input Handler
Collecting and validating user input.

Processing Module
Filtering and summarizing expense records.

Output Formatter
Displaying information clearly.

Storage Module
Saving and loading persistent data.

These languages provide a clear contrast between high-level abstraction (Python) and low-level control (C++).

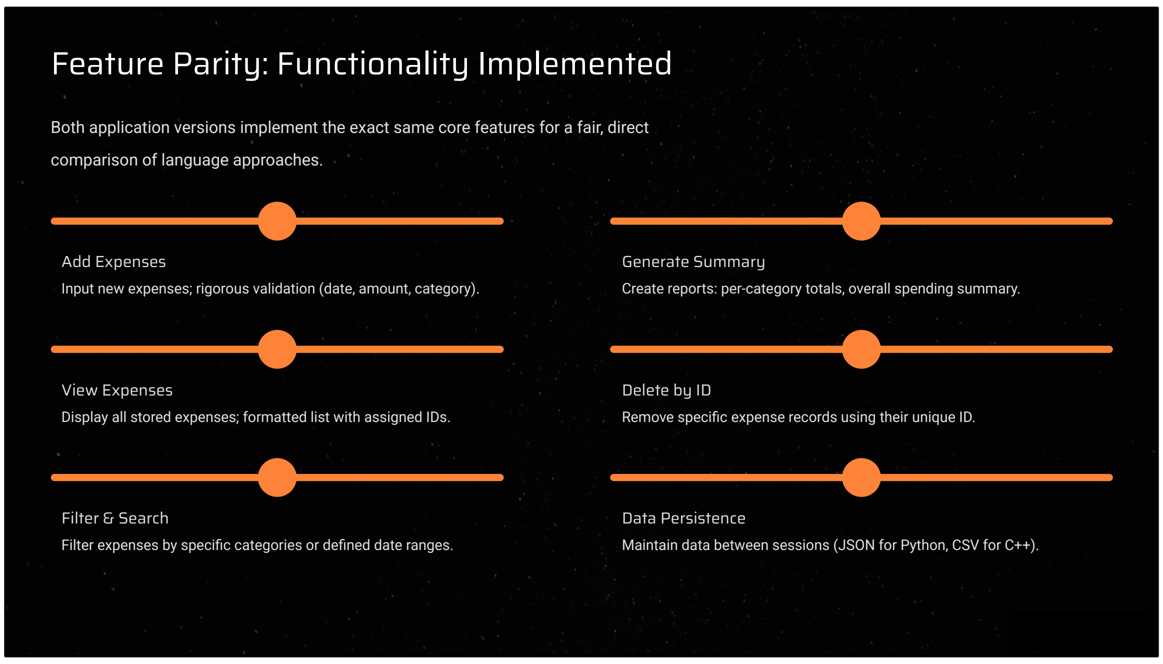
Our project focused on comparing the fundamental paradigms of the two languages:

Python’s dynamic, interpreted model versus C++’s static, compiled model.

To ensure a fair comparison, we used the same modular architecture across both implementations.

- The **Input Handler** collects and validates user inputs.
- The **Processing Module** handles filtering and summarization logic.
- The **Storage Module** manages data persistence.
- And the **Output Formatter** handles how information is displayed in the terminal.

Together, these modules let us isolate how each language handles the same set of operations highlighting Python’s high-level abstraction against C++’s low-level control.



Feature Parity: Functionality Implemented

Both application versions implement the exact same core features for a fair, direct comparison of language approaches.

Add Expenses
Input new expenses; rigorous validation (date, amount, category).

View Expenses
Display all stored expenses; formatted list with assigned IDs.

Filter & Search
Filter expenses by specific categories or defined date ranges.

Generate Summary
Create reports: per-category totals, overall spending summary.

Delete by ID
Remove specific expense records using their unique ID.

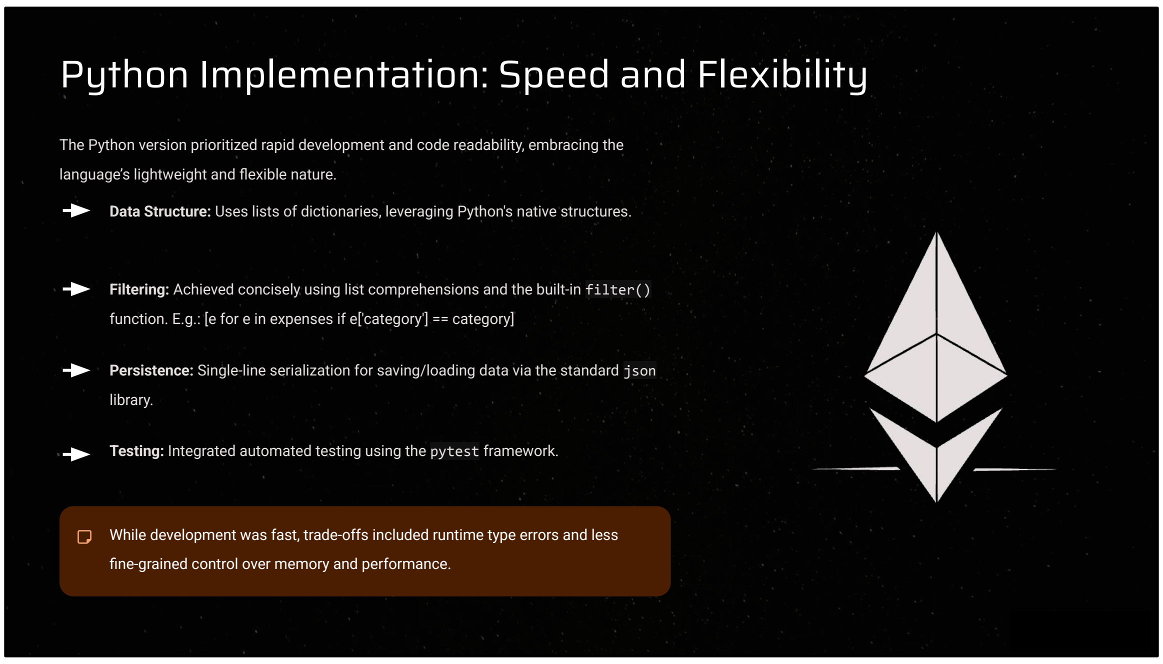
Data Persistence
Maintain data between sessions (JSON for Python, CSV for C++).

To compare languages effectively, we made sure both applications deliver the exact same functionality.

Both versions allow you to:

- Add new expenses with validation for date, amount, and category.
- View expenses in a formatted list with unique IDs.
- Filter and search by category or date range.
- Generate summary reports showing totals by category and overall spending.
- Delete expenses by ID.
- And finally, persist data between sessions.

By keeping the features identical, we could focus our comparison on how each language’s syntax, typing model, and runtime affected the implementation, not the functionality itself.



Python Implementation: Speed and Flexibility

The Python version prioritized rapid development and code readability, embracing the language's lightweight and flexible nature.

- **Data Structure:** Uses lists of dictionaries, leveraging Python's native structures.
- **Filtering:** Achieved concisely using list comprehensions and the built-in `filter()` function. E.g.: [e for e in expenses if e['category'] == category]
- **Persistence:** Single-line serialization for saving/loading data via the standard `json` library.
- **Testing:** Integrated automated testing using the `pytest` framework.

While development was fast, trade-offs included runtime type errors and less fine-grained control over memory and performance.

The Python version prioritized readability and development speed.

This version uses function composition and built-in data structures.

Expenses are dictionaries stored in a list, requiring no schema definition.

Filtering is achieved in a single line using list comprehensions: `[e for e in expenses if e['category'] == category]`.

Aggregation is handled by looping through expenses and building totals dynamically with dictionaries.

Persistence leverages JSON serialization, eliminating manual parsing logic.

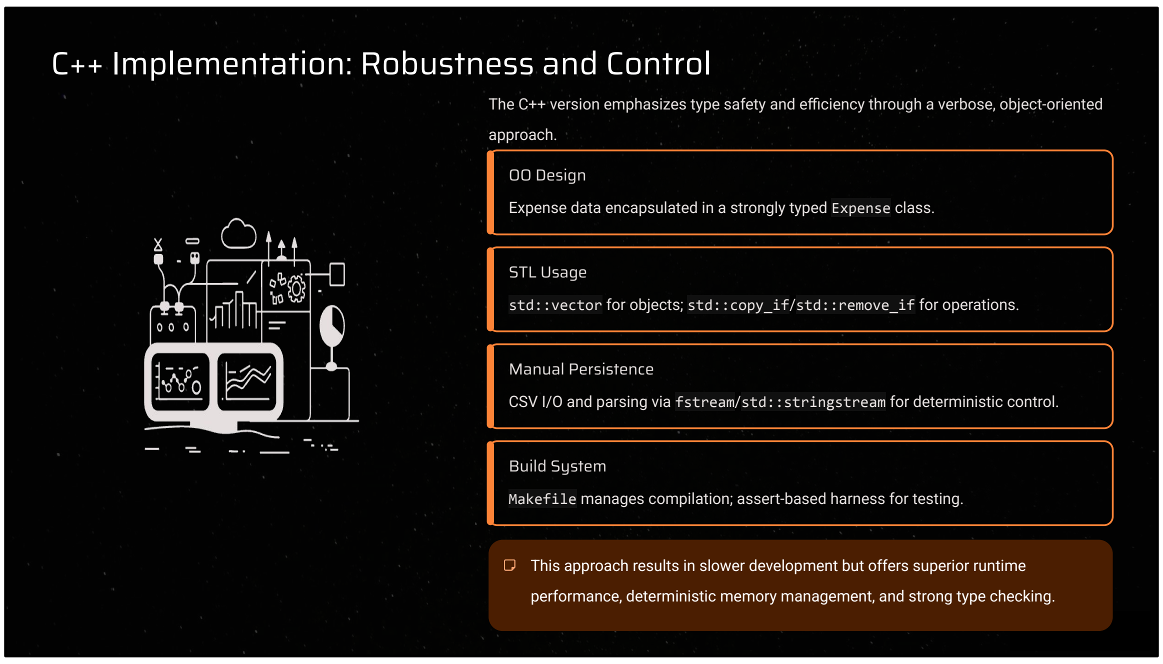
Error handling relies on Python's `try/except`, and testing is automated via `pytest` to verify core logic and edge cases.

This pattern demonstrates how Python's dynamic typing and functional-style programming make it ideal for rapid prototyping and iterative refinement.

Overall, the development process was fast and intuitive.

The trade-off, however, was the risk of runtime type errors and slightly less control over performance and memory which are common characteristics of dynamically typed languages.

But Python excelled at rapid prototyping and ease of debugging.



C++ Implementation: Robustness and Control

The C++ version emphasizes type safety and efficiency through a verbose, object-oriented approach.

OO Design

Expense data encapsulated in a strongly typed `Expense` class.

STL Usage

`std::vector` for objects; `std::copy_if`/`std::remove_if` for operations.

Manual Persistence

CSV I/O and parsing via `fstream`/`std::stringstream` for deterministic control.

Build System

`Makefile` manages compilation; assert-based harness for testing.

This approach results in slower development but offers superior runtime performance, deterministic memory management, and strong type checking.

The C++ version represents the opposite end of the spectrum emphasizing precision, type safety, and efficiency.

In C++, we followed an object-oriented approach built around an `Expense` class.

Each object encapsulates data and provides getter methods.

Expenses are stored in a `std::vector`, enabling indexed access and iteration.

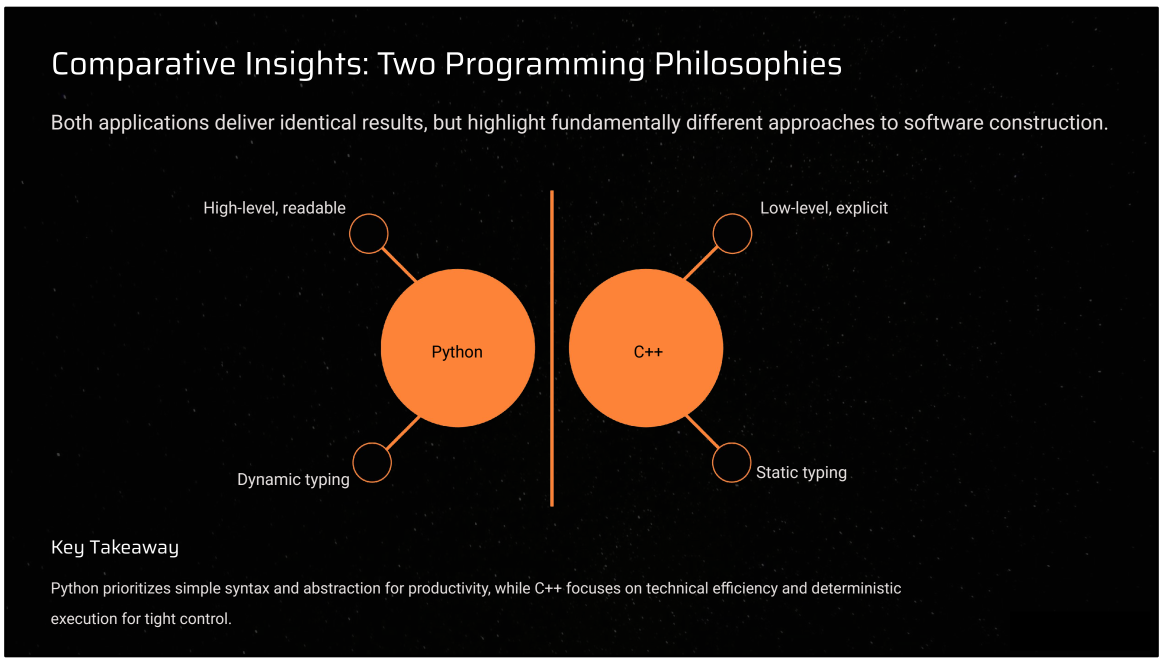
Filtering uses STL algorithms like `std::copy_if` combined with lambda expressions to produce filtered lists.

For summaries, an `std::unordered_map` accumulates totals by category, mirroring Python’s dictionaries but with static typing.

Persistence is implemented manually using `std::ofstream` for writing and `std::stringstream` for reading CSV lines ensuring deterministic control over I/O.

C++ enforces compile-time type checking, and errors are managed through explicit `try/catch` blocks, demonstrating disciplined programming patterns with fine-grained memory and performance control.

While this required more time to build, it provided excellent runtime efficiency, deterministic memory management, and compile-time error checking demonstrating C++'s strength in control and performance.



Comparative Insights: Two Programming Philosophies

Both applications deliver identical results, but highlight fundamentally different approaches to software construction.

High-level, readable
Low-level, explicit
Python
C++
Dynamic typing
Static typing

Key Takeaway

Python prioritizes simple syntax and abstraction for productivity, while C++ focuses on technical efficiency and deterministic execution for tight control.

If we compare both implementations side by side, we see distinct programming philosophies emerge.

Python’s approach is functional and data-centric relying heavily on dynamic structures and built-in libraries for operations like filtering and serialization.

It favors simplicity, abstraction, and developer productivity. You can implement and test features quickly with minimal code.

C++’s implementation is structural and type-driven, where classes, iterators, and templates define behavior explicitly.

It requires deliberate design and explicit control, but rewards that effort with speed, efficiency, and strong type guarantees

And so, Python code is concise and easier to modify, while C++ code is verbose but highly efficient and predictable.

The Python version prioritizes developer experience; the C++ version prioritizes runtime performance.

Both approaches illustrate that language design directly shapes how developers architect and reason about software.

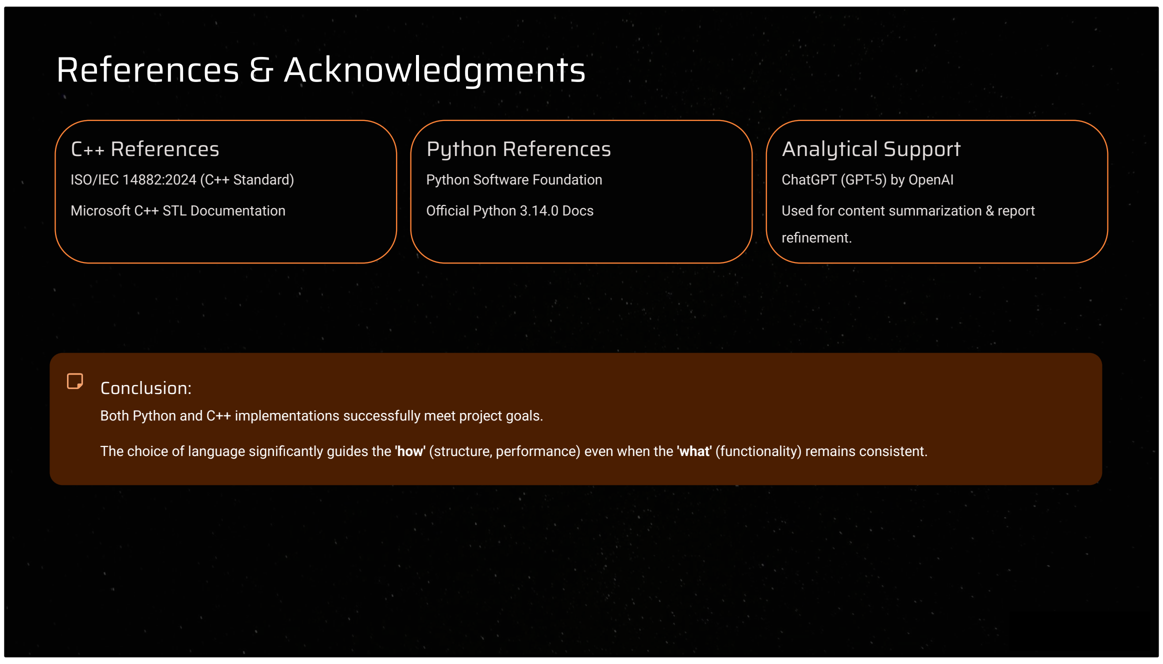
Demonstration

A live demo will highlight the practical differences between Python and C++ command-line applications.

Python Demo

C++ Demo

The two implementations offer the same functionality but vividly illustrate how distinct programming language paradigms shape core software engineering choices.



References & Acknowledgments

C++ References

ISO/IEC 14882:2024 (C++ Standard)

Microsoft C++ STL Documentation

Python References

Python Software Foundation

Official Python 3.14.0 Docs

Analytical Support

ChatGPT (GPT-5) by OpenAI

Used for content summarization & report refinement.

Conclusion:

Both Python and C++ implementations successfully meet project goals.

The choice of language significantly guides the **'how'** (structure, performance) even when the **'what'** (functionality) remains consistent.

Thank you!

Questions?