**Synthax, Semantics and Memory Management**

Shyam Nath

University of Cumberland

Advanced Programming Languages (MSCS-632-M80)

Student ID: 005032491

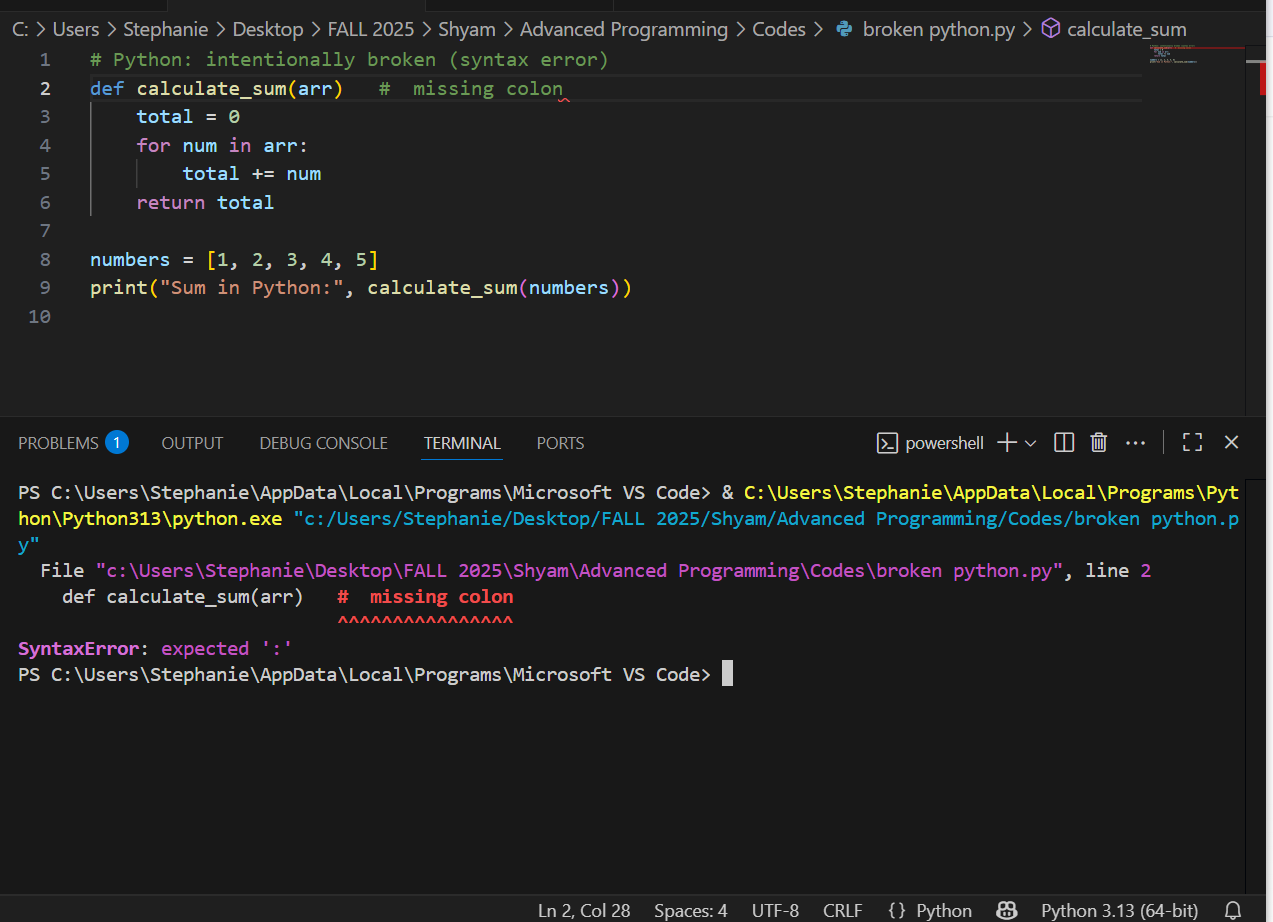
Assignment 2

September 21, 2025

**Synthax, Semantics and Memory Management**

**Part 1: Analyzing Syntax and Semantics**

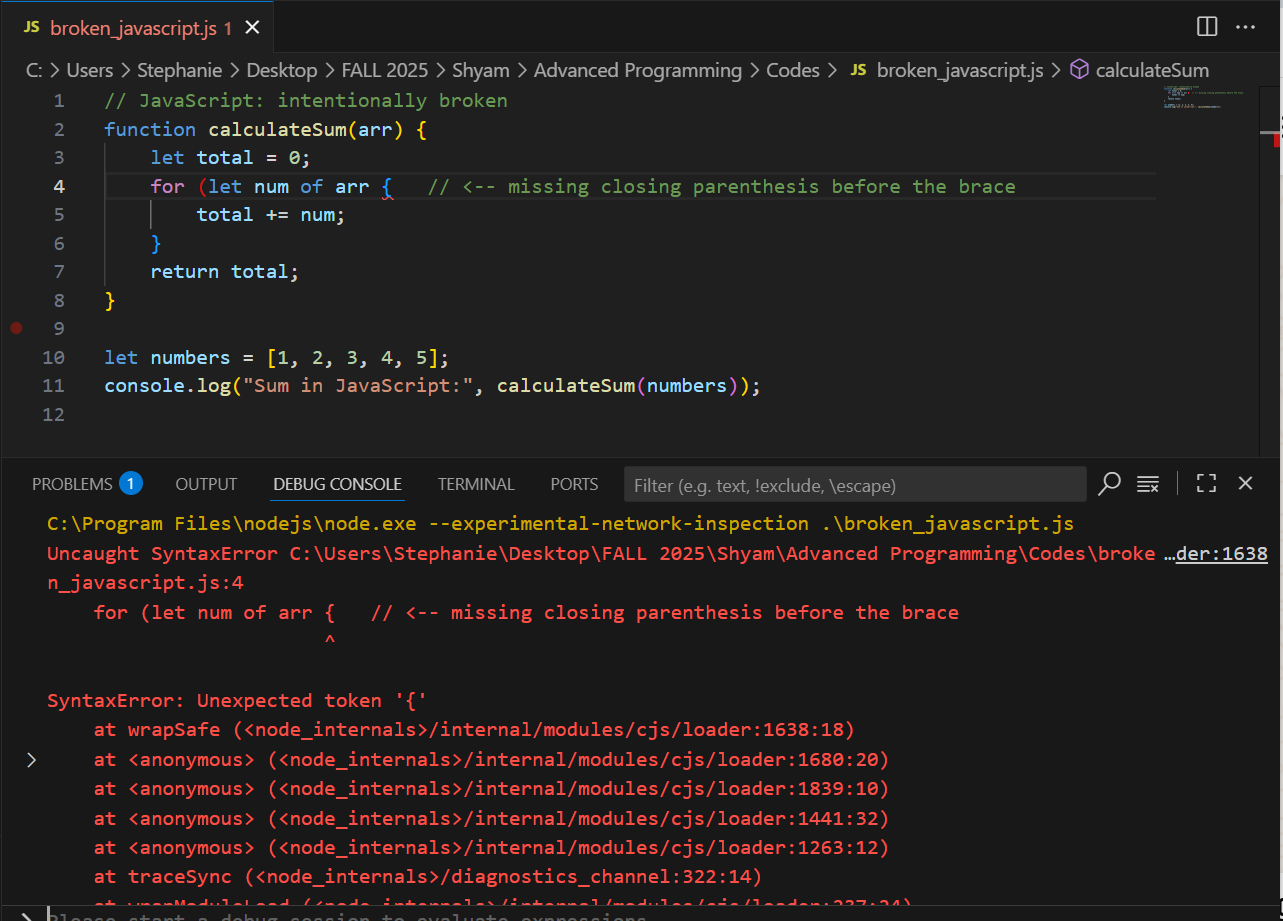
**Broken python code**



**Explanation**

The failure of the parser during the reading of the def line is due to the need to include a colon (:) between the functional parameter list and the list. Python can identify syntax errors during parsing and raises an exception of SyntaxError and provides the location of the problem. The program never runs.

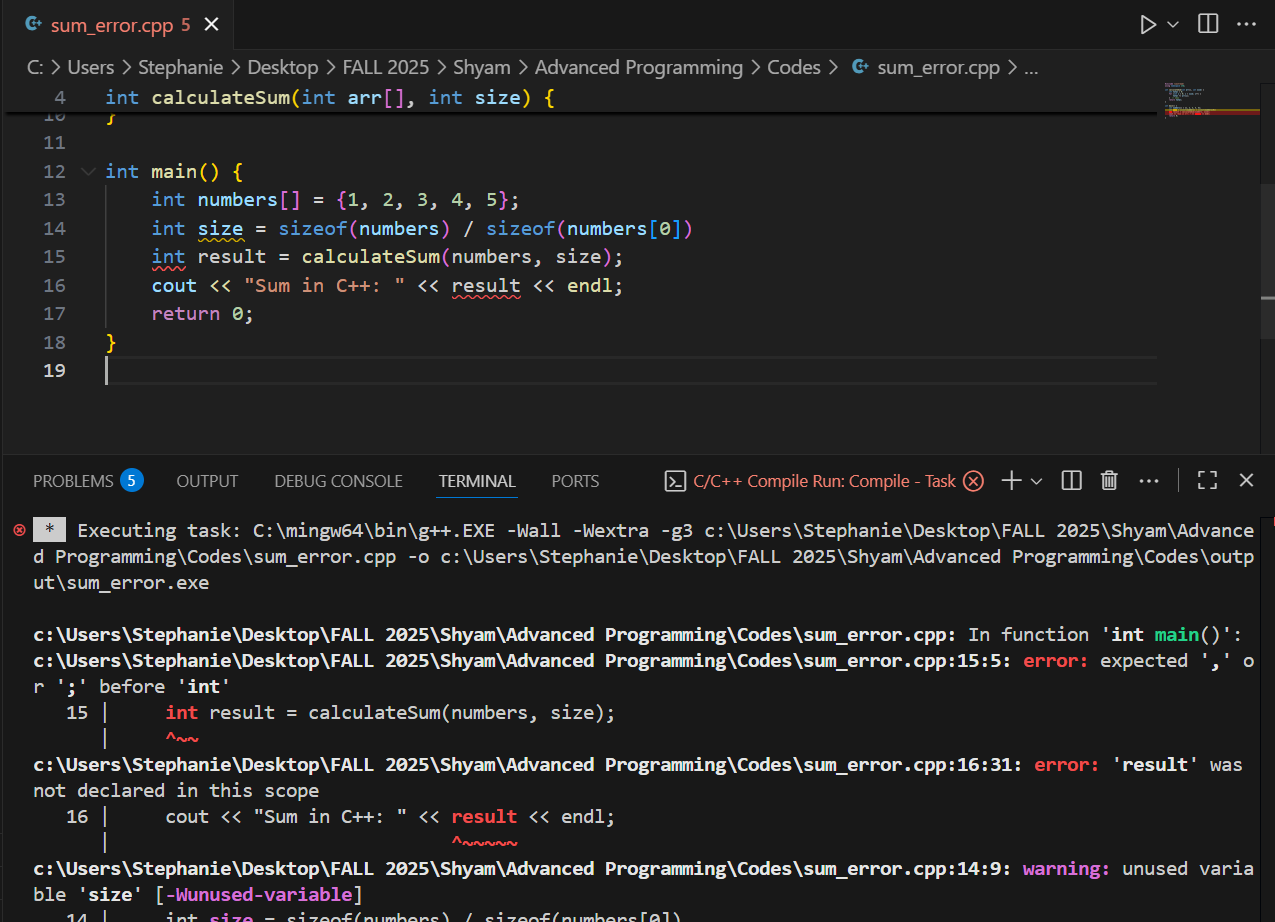
**Broken JavaScript code**



**Explanation**

JavaScript parser anticipated that the for (...) header would be closed by ) but instead it perceives that it is enclosed by {. Parsing syntax error and Node/V8 throws the SyntaxError (Unexpected token) at the point of the offending token. Similar to Python, syntax errors in JS do not start executing until the time of parsing.

**Broken C++ code**



**Explanation**

C++ is compiler based; the lack of semicolon at the end of the int size = line is where the size of the int is defined leaves the first compile error. Since the compiler halts once the initial issue is identified, you usually end up with cascading errors that seem unrelated (e.g., result not declared) - they are the spill over effects of the initial syntax error. The compiler will tell the file name, line and often column and many compilers provide numerous messages to assist in identifying the main error.

**Brief comparison of syntax-error handling.**

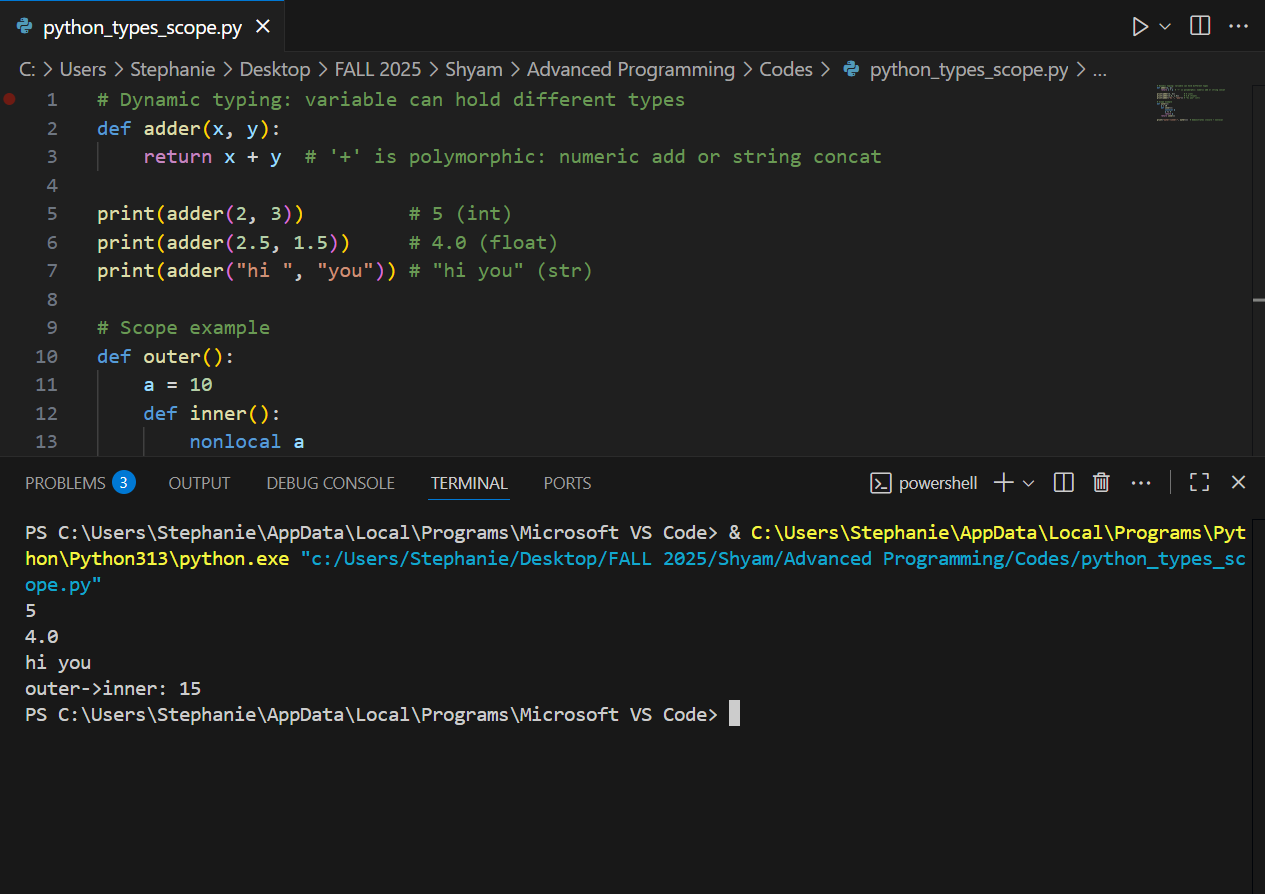
**Python (interpreted):** Parser checks the source prior to execution; SyntaxError is an exception, and there is a caret at the point of error. Direct and immediate; program is not initiated.

**JavaScript (interpreted / JIT):** Like Python - a failure to (safe) successfully parse results in an error (a SyntaxError e.g. Unexpected token) and execution halts.

**C++ (compiled):** A compile-time error with line/column, and in many cases numerous chain messages. Compilers are more verbose in their diagnostics, and often have more than one error since the parser becomes confused on the first syntax problem. Fix the first error first.

**1.2 Section 2**

**Python: Dynamic typing and function scope example**



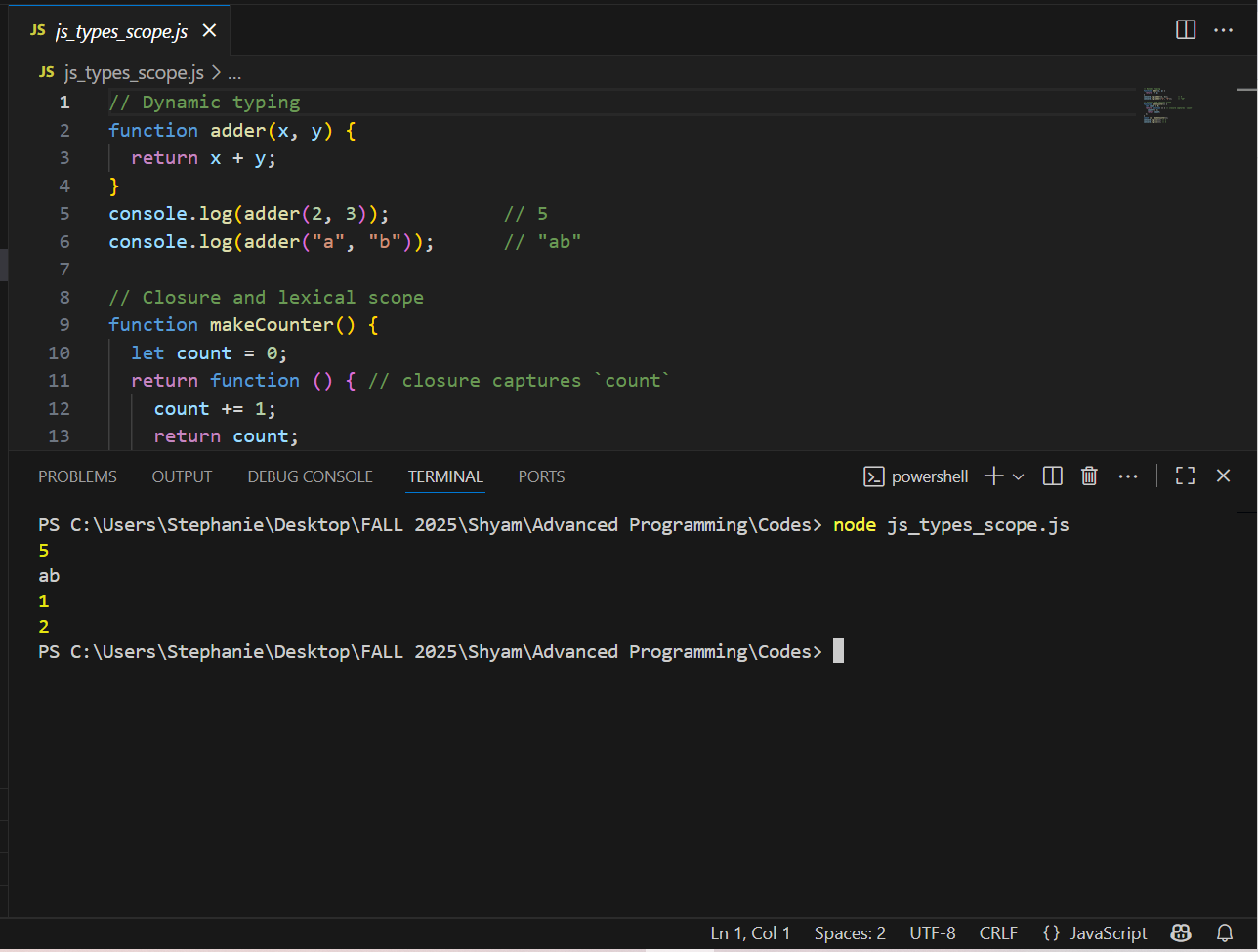
**Notes:**

Python is duck typed and dynamically typed. You can use the adder function for numbers and strings; type checking is done at the execution time.

Nonlocal: captures (and mutates) a variable from an outer scope, and is a closure. Python closures are reference-counted (name-binding), but to be able to assign something you have to declare nonlocal (Hasan et al., 2025).

Performance: dynamic typing incurs runtime cost (type checks, method dispatch).

**JavaScript : dynamic typing and closures**



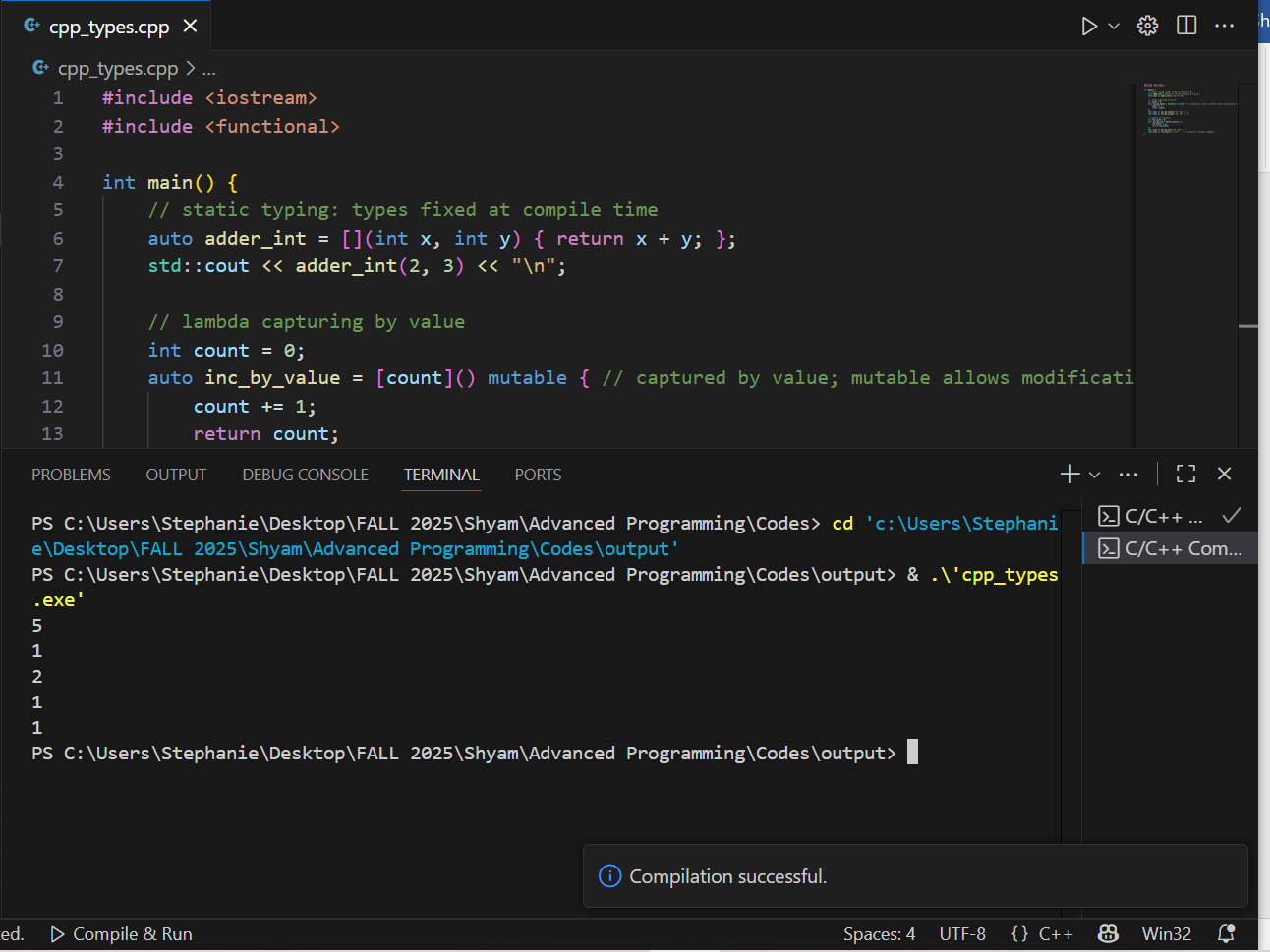
**Behavior & notes:**

JavaScript is a dynamically typed language. Coercions may happen implicitly, such as 1 + "2" to give "12" (and this may lead to gotchas).

JavaScript closures are lexically captured, functions have references to immediate variables. This is the focus of numerous JS patterns (factories, modules).

Performance: dynamic checks, and possible JIT optimizations; closures can retain memory (closed-over variables) until it is collected by GC

**C++ — static typing and lambdas (closures)**



**Behavior & notes:**

* C++ is statically typed: types are checked at compile time — more safety and performance (no runtime type dispatch for basic operations).
* Lambdas can capture by value or reference; mutable lets you modify the captured-by-value copy. C++ closures are implemented with generated function objects; capturing by reference can lead to dangling references if the captured variable goes out of scope.

**Performance:** static typing + ahead-of-time compilation leads to better raw performance, lower runtime overhead.

**Three key semantic differences and their effects**

**Type system: static (C++): dynamic (Python/JS)**

* Impact on behavior The C++ compiler can identify and reject a significant number of type errors during compilation, whereas Python/JS can do it during runtime.
* Impact on performance: Static typing normally permits a more effective machine code and fewer run time checks.

**Lifetime management (manual/RAII vs GC vs ownership) Memory model.**

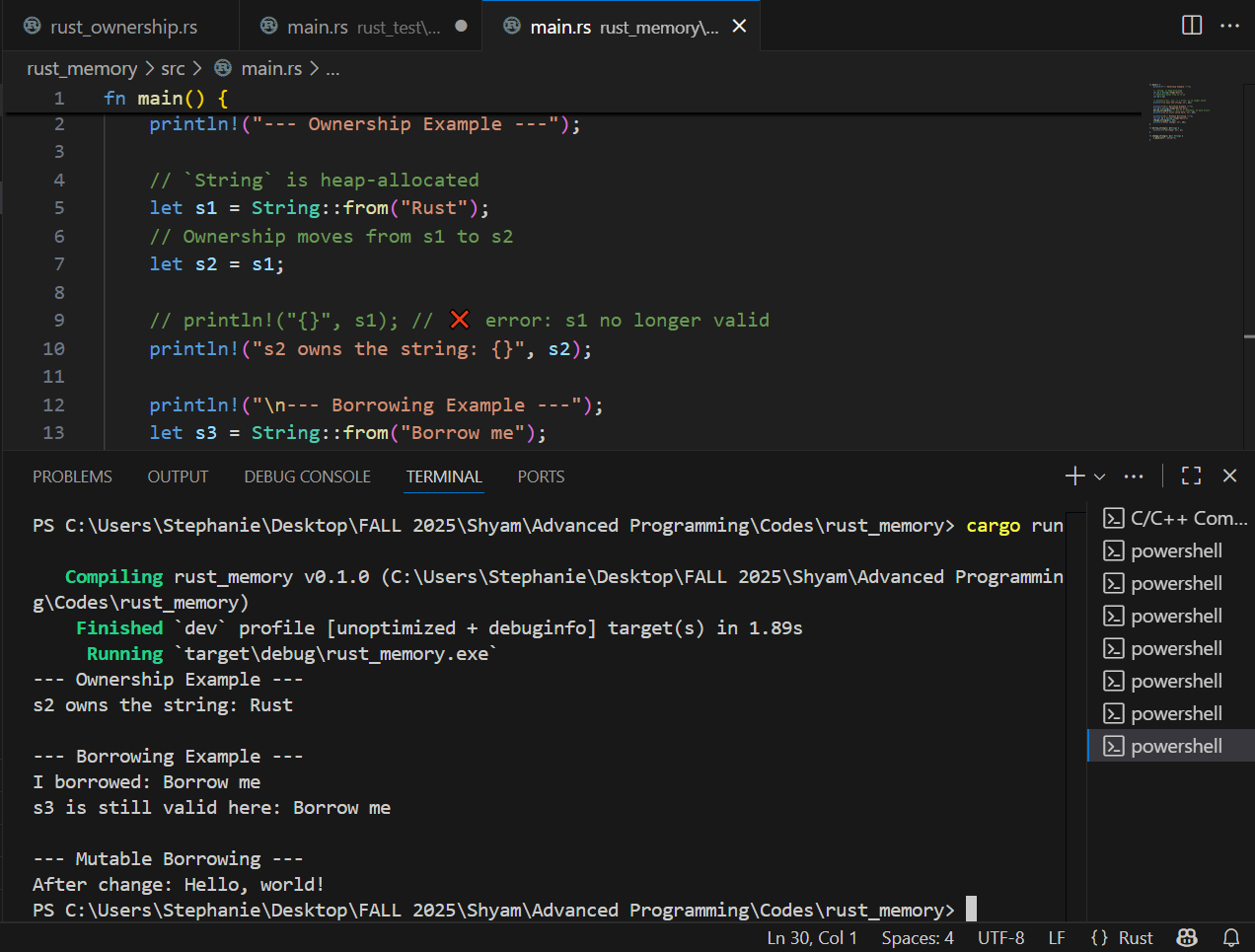
* Impact on behavior C++ operates on RAII/managed memory (destructors) and can trivial to create dangling pointers or leaks when used incompetently. Java/JS are based on garbage collection to free memory when unreachable, in Python free memory is through refcount + cycle GC.
* Performance impact: GC introduces pause/overhead, RAII deterministic destruction and usually reduced latency.

**Representation of scoping and closures.**

* Behavioral impact Since JS/Python closures are lexically captured variables, and C++ closures can be captured by implicit reference/value (explicit), capturing by value or reference may introduce dangling reference bugs in programs that use closures without understanding this fact.
* Impact on performance: Closures are able to cache memory and result in longer lifetime of captured variables; C++ value-capture is able to eliminate indirections and become faster.

**Part 2: Memory Management**

**Rust — ownership & borrowing example**

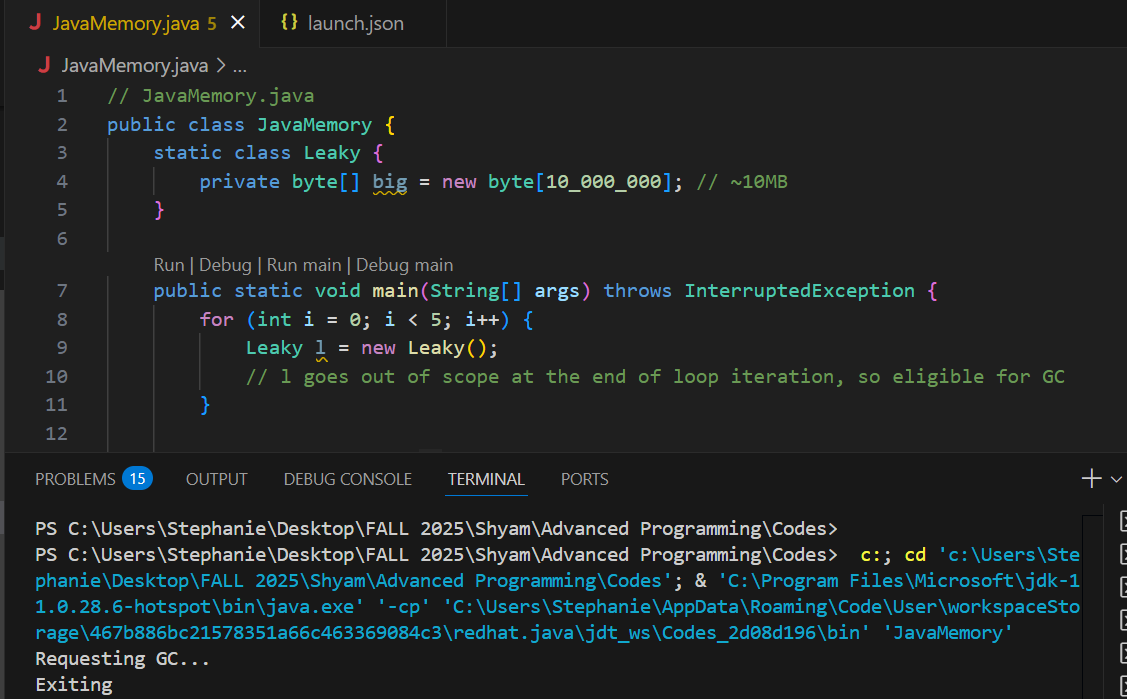


**Key points & behavior:**

* Ownership rules enforce single owner by default; moving ownership invalidates the original binding (prevents use-after-free).
* Borrowing (&T) allows temporary access without transferring ownership; mutable borrows (&mut T) are exclusive. The borrow-checker rejects code that would cause data races or dangling pointers at compile-time.

**Typical compile-time safety:** the borrow checker prevents dangling references and data races. No runtime GC; memory freed deterministically when owners go out of scope (Prabakar & Kiran, 2024).

**Java — garbage collection example**



**Key points & behavior:**

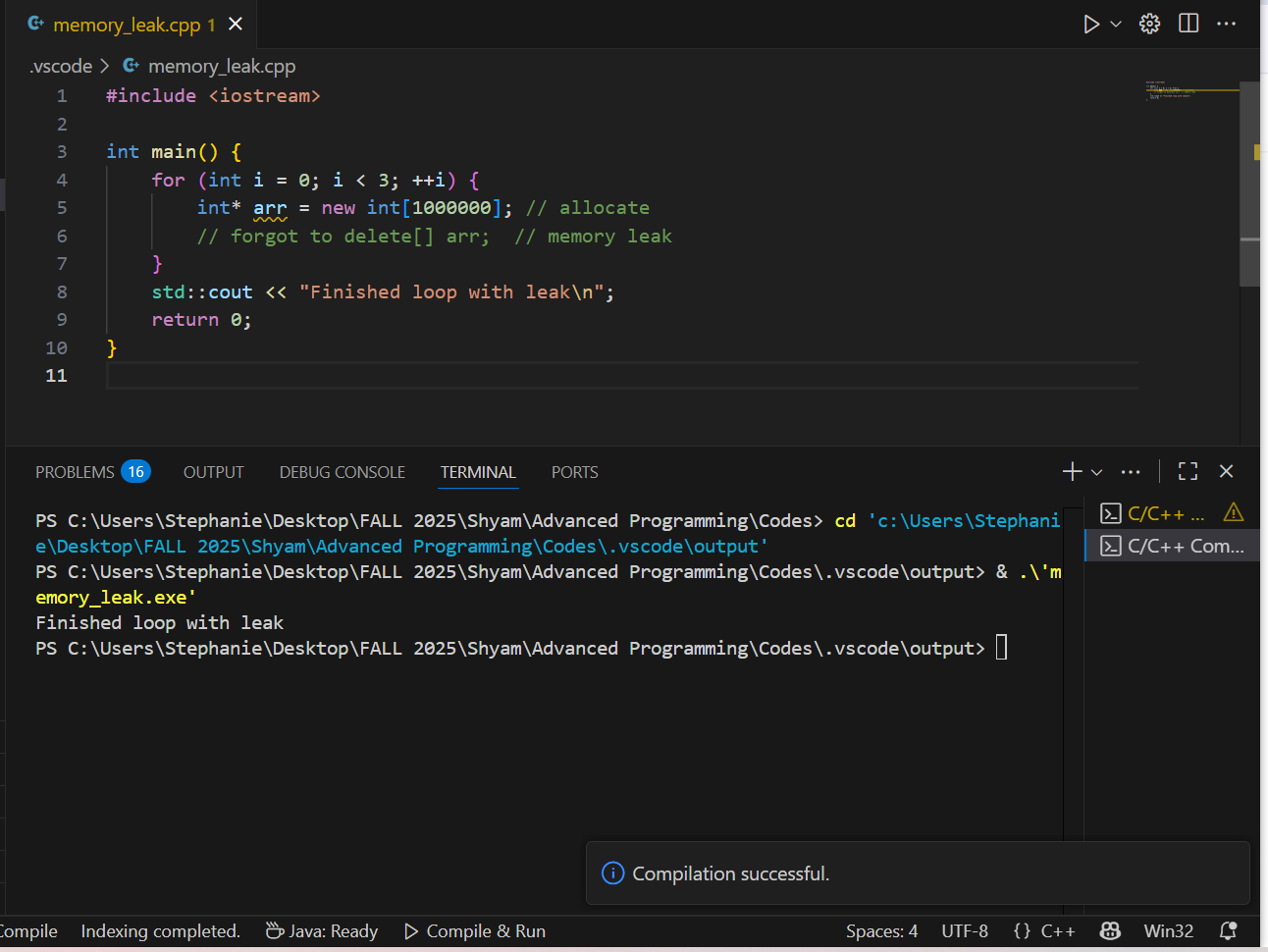
Java Memory Java places memory on the heap (new) and the garbage collector removes unreachable objects.

There is no free memory (you cannot deterministically free memory), but GC makes it easier to bookkeep programs. System.gc() and other calls of the kind are hints. Heap can be profiled using tools such as VisualVM, Java Flight Recorder or jcmd.

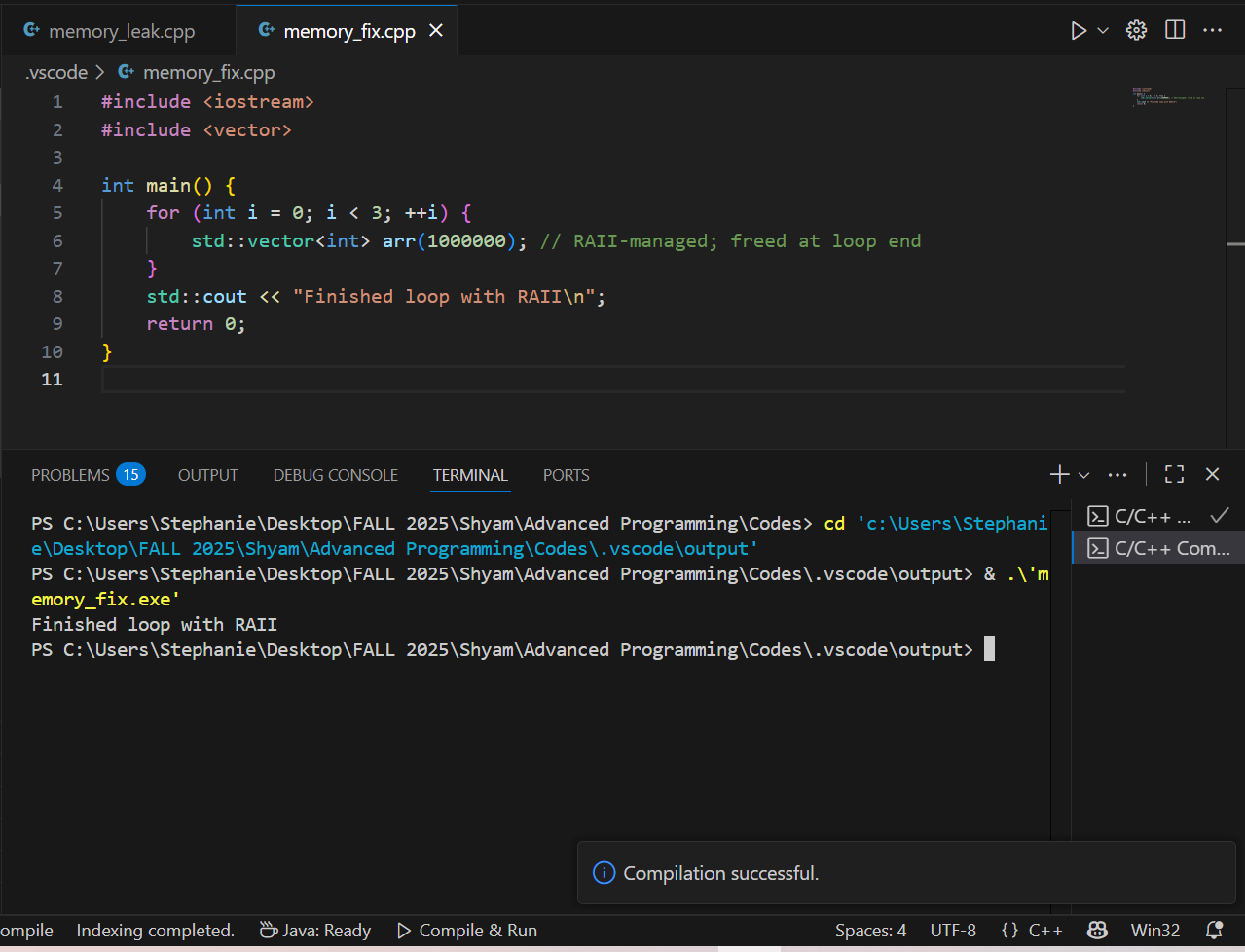
Memory problems: Java memory leaks normally occur when references are accidentally held on to (e.g. large objects being stored in static caches), rather than forgetting to free memory.

**C++ — manual memory management example (leak and fix)**

**Memory leak**



**Memory fix**



**Key points & behavior:**

In C++ handbook, it is quite risky using new[]/delete[] because delete is not mandatory. Dangling pointer occurs when memory is deallocated and the pointer is reused after deletion of memory and if the pointer is dereferenced.

By using RAI types (std::vector, smart pointers such as std::unique\_ptr, std::shared\_ptr), a lot of errors that need to be made manually are avoided.

**Memory errors: Leak, dangling pointers, double-free**

Key points to be investigated for solving the problem were as follows:

* In Java the GC makes forgetting to free (i.e. memory leaks caused by missing close or free) impossible but memory leaks caused by indirectly maintaining references are still possible. C++ memory leak - if programmer forgets delete then use Valgrind/Memcheck to locate them Rust has no such leaks unless std::mem::forget or Rc cycles are introduced (Hasan et al., 2025).
* Dangling pointer: Pointer/ reference which refers to a freed memory. Browsing the Internet to try to find the answer, I realized that what I was trying to do can be done with pointers in C++ (which I do not know how to do in Rust), but what I want to do is not possible at compile time (and I think I see why) because of Rust's borrow checker.
* Double-free - freeing memory twice (C++) is undefined behavior - often crashes.

**Link to GitHub Repository**

<https://github.com/ShyamNath8/MSCS_632_Assignment_2.git>

All the actual codes are in the GitHub Repository

**References**

Hasan, A. F., Saadya, F. J., & Raheem, F. S. (2025). Comparative Analysis of Four Programming Languages for Machine Learning. *Ingenierie des Systemes d'Information*, *30*(6), 1437.

Prabakar, A., & Kiran, R. (2024). WebAssembly Performance Analysis: A Comparative Study of C++ and Rust Implementations.