MSCS-632: Assignment 2: Syntax, Semantics, and Memory Management

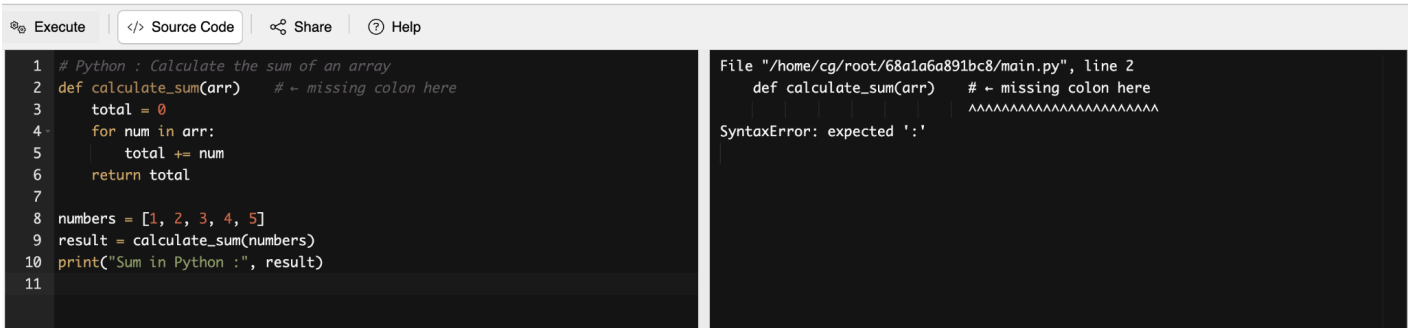
Oishani Ganguly

GitHub Repository:
https://github.com/Oishani/MSCS632_Assignment2

Part 1: Analyzing Syntax and Semantics

1.1 Section 1: Syntax Errors in Python vs. JavaScript vs. C++

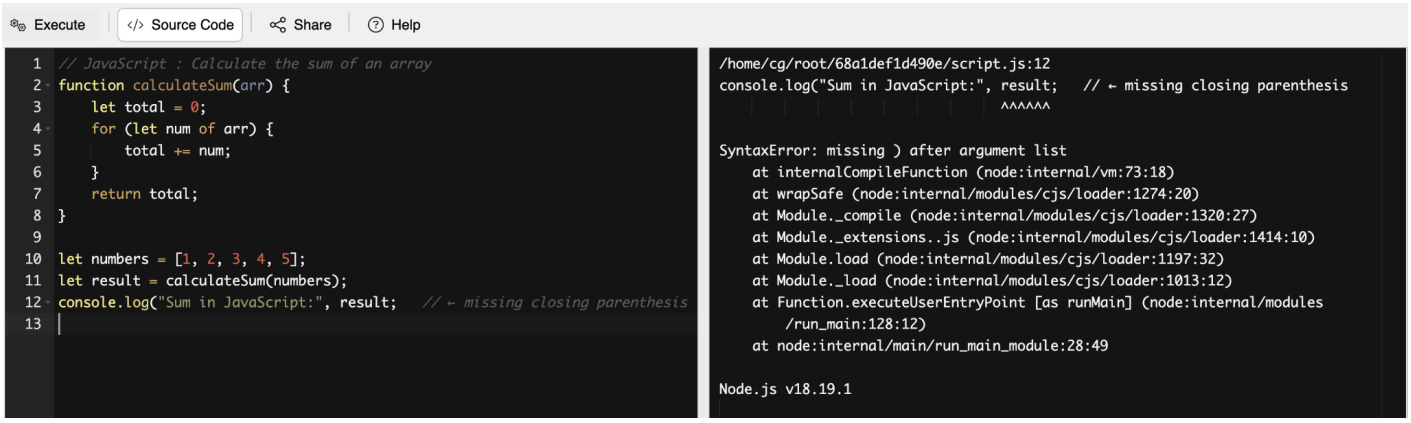
Python



What the message means:

- Python parses code just-in-time before executing.
- It pinpoints the exact location (file, line, caret under)) and states the expectation “expected ':'” - the colon that starts a function suite is missing.

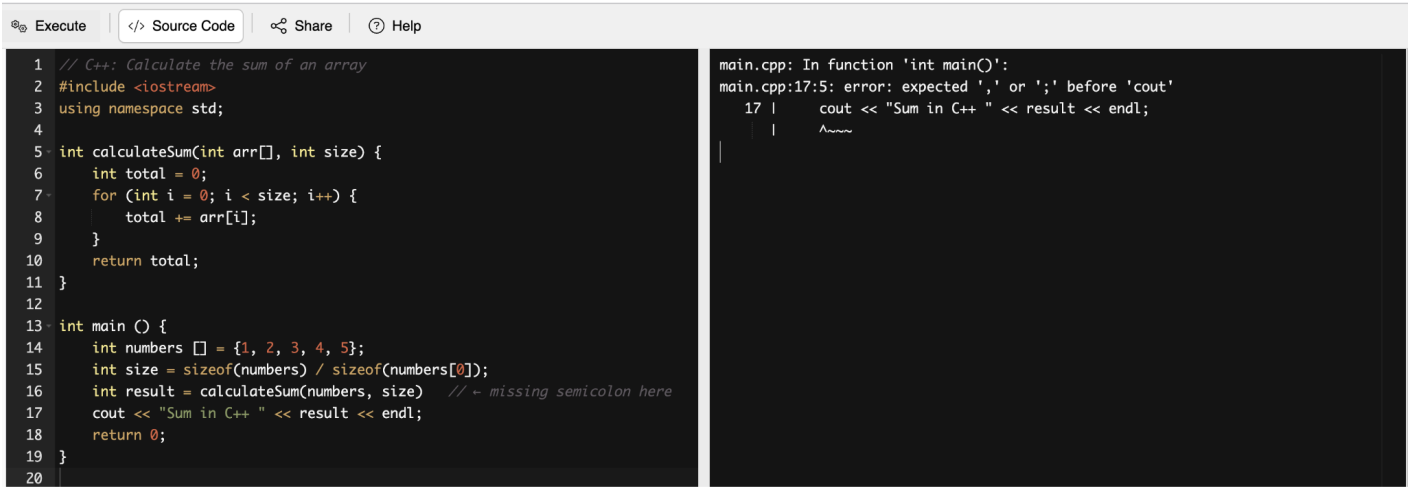
JavaScript



What the message means:

- JavaScript engines parse the whole file before running.
- The parser can't complete the function call, so it throws a `SyntaxError` with a specific hint "missing) after argument list."
- Stack lines give the file and character position; execution doesn't begin.

C++



What the message means:

- C++ compilation stops because a semicolon is required after the function call.

- Compilers often report the error at the next token (`cout`) where the parse failed and tell us what was expected (‘,’ or ‘;’). No binary is produced until all such errors are fixed.

Cross-Language Comparison: How Syntax Errors Are Handled

Aspect	Python	JavaScript (Node/Browser)	C++ (g++)
When detected	Parse time immediately before execution	Parse time before execution	Compile time (separate build step)
Typical error type	`SyntaxError` with precise expectation (e.g., “expected ':'”)	`SyntaxError` with named construct (e.g., “missing) after argument list”)	Compiler diagnostic (e.g., “expected ‘;’ before ‘cout’”)
Location accuracy	Very precise line/column + caret under offending token	Precise line/column; often shows the problematic token; stack shows file	May flag the token after the real mistake (cascading errors possible)
Runtime impact	Program doesn’t start; fix & re-run quickly	Program doesn’t start; fix & re-run quickly	No executable produced; must recompile after fixes
Message clarity	Friendly and specific; suggests the exact missing symbol	Short, specific grammar hint; concise	More terse, sometimes cryptic; errors can cascade from one typo

Conclusion

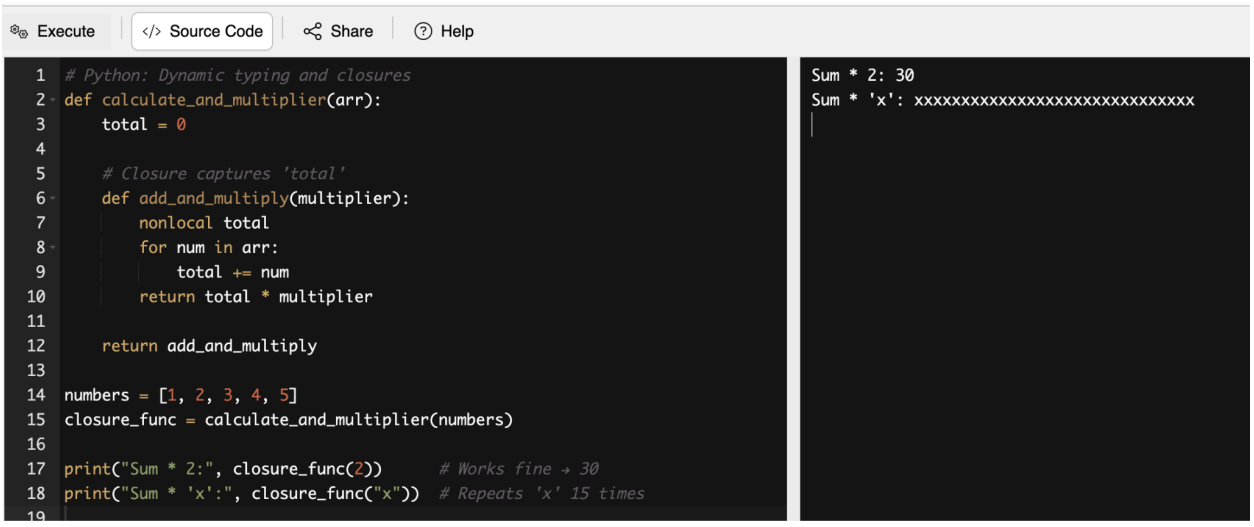
- Python and JavaScript both stop before any code runs; their messages point directly at the broken construct and are fast to iterate on.
- C++ reports errors during compilation; a single missing token can trigger multiple diagnostics, and the first error is often the most informative to fix.
- All three give line numbers; Python and JS usually highlight the exact expectation, while C++ sometimes highlights the point of failure right after the true mistake.

1.2 Section 2: Type Systems, Scopes, and Semantic Differences in Python, JavaScript, and C++

Overview

This section examines how Python, JavaScript, and C++ handle type systems, scope rules, and closures through custom programs. Each program calculates a sum, applies a multiplier via a closure, and demonstrates how type handling differs across the three languages.

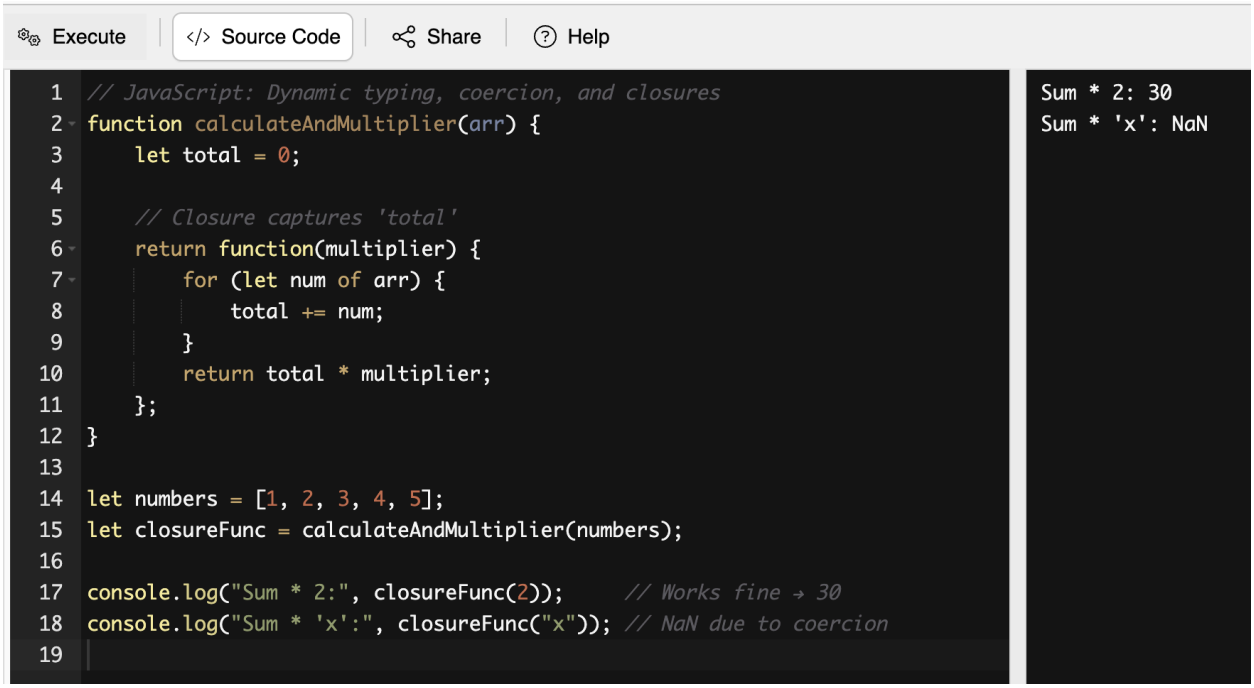
Python (dynamic typing, closures)



Analysis

- **Dynamic typing**: Python allows int * str, interpreting it as string repetition.
- **Closures**: The inner function add_and_multiply retains access to total and updates it each call.
- **Scope**: The nonlocal keyword is required to modify total from the outer scope.

JavaScript (dynamic typing, coercion, closures)



Analysis

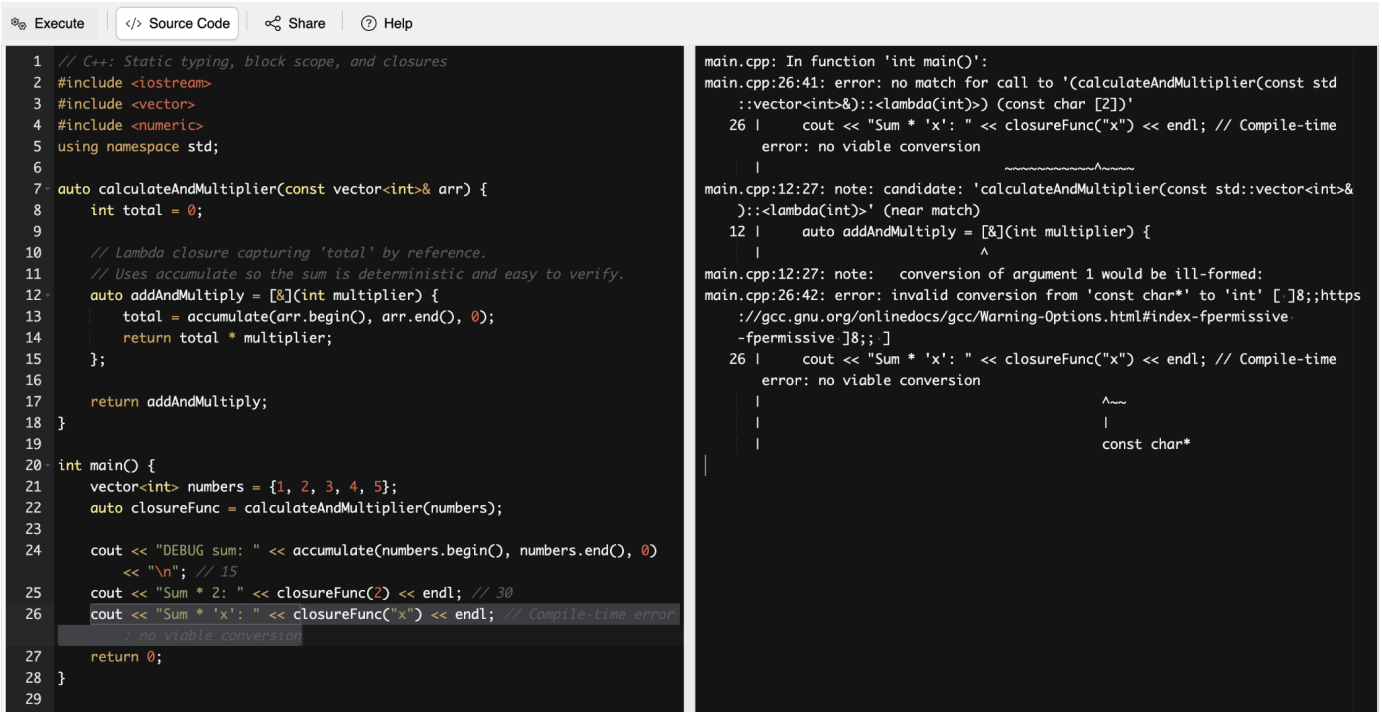
- **Dynamic typing with coercion:** JavaScript attempts numeric multiplication, fails, and returns `NaN`.
- **Closures:** Functions naturally retain variables from their lexical scope.
- **Scope:** `let` ensures block scope for `num` and `total`.

C++ (static typing, explicit lambda capture)

Execute | Source Code | Share | Help

```cpp
// C++: Static typing, block scope, and closures
#include <iostream>
#include <vector>
#include <numeric>
using namespace std;

auto calculateAndMultiplier(const vector<int>& arr) {
    int total = 0;

    // Lambda closure capturing 'total' by reference.
    // Uses accumulate so the sum is deterministic and easy to verify.
    auto addAndMultiply = [&](int multiplier) {
        total = accumulate(arr.begin(), arr.end(), 0);
        return total * multiplier;
    };

    return addAndMultiply;
}

int main() {
    vector<int> numbers = {1, 2, 3, 4, 5};
    auto closureFunc = calculateAndMultiplier(numbers);

    cout << "DEBUG sum: " << accumulate(numbers.begin(), numbers.end(), 0)
        << "\n"; // 15
    cout << "Sum * 2: " << closureFunc(2) << endl; // 30
    // cout << "Sum * 'x': " << closureFunc("x") << endl; // Compile-time
        error: no viable conversion
    return 0;
}

```

```
DEBUG sum: 15
Sum * 2: 30
```



Analysis

- **Static typing & compile-time safety:** The lambda's parameter is `int`. Passing a string (e.g., `"x"`) triggers a **compile-time error** due to type mismatch.
- **Closures with explicit capture:** The lambda uses `[&]` to capture `total` by reference; mutation is deliberate and visible.
- **Block scope:** `total` and `numbers` live within their respective `{}` blocks; lifetimes are clear.
- **Deterministic behavior & performance:** `std::accumulate` performs a typed reduction over `int`, enabling the optimizer and avoiding hidden coercions found in dynamic languages (Stroustrup & Sutter 2024).

Three Key Semantic Differences and Their Effects

1. Type System & Error Timing

- **Python:** Dynamic, strong. `int * str` is valid as sequence repetition; invalid mixes (e.g., `int += str`) raise runtime exceptions.
- **JavaScript:** Dynamic with coercion. Invalid numeric ops typically yield `NaN` rather than throwing.

- **C++:** Static, strong. The lambda parameter is `int`; passing a string would be rejected at compile time. Using `std::accumulate` enforces a typed reduction (`int`) and prevents implicit coercions.

Effect: Python/JS enable rapid iteration but can hide issues (repetition/`NaN`). C++ front-loads errors and enables stronger optimization thanks to compile-time type knowledge.

2. Closures & Scope Manipulation

- **Python:** Lexical scope; must use `nonlocal` (or `global`) to rebind outer names captured by closures.
- **JavaScript:** Lexical closures are frictionless and pervasive (callbacks/promises); `let`/`const` provide block scoping.
- **C++:** Closures via lambdas with explicit capture. We used `[&]` to capture `total` by reference. Mutation is deliberate and visible; capture lists can be fine-grained (`[&total]`, `[=]`, etc.).

Effect: JS is the most ergonomic for closure-heavy patterns; Python is clear but requires `nonlocal`; C++ is explicit and safer in large codebases, making mutation intent and lifetimes explicit.

3. Runtime/Performance Model

- **Python/JS:** Interpreted/JIT with Garbage Collector (GC) - excellent for velocity; compute-heavy loops often need library help (NumPy) or engine optimizations (V8 JIT).
- **C++:** Ahead-of-time compiled, RAII, no GC by default. `std::accumulate` on `int` enables predictable, optimized machine code.

Effect: For tight numeric loops, idiomatic C++ generally outperforms Python/JS; Python/JS shine for rapid prototyping and integration.

Conclusion

The finalized experiment underscores how type semantics, closure mechanics, and scope rules shape real behavior:

- Python favors expressiveness: operators like `*` are overloaded for sequences, making `"x" * 15` valid; errors surface at runtime for truly incompatible operations.
- JavaScript leans on coercion: invalid numeric combinations quietly become `NaN`, which can propagate if unchecked.
- C++ enforces static correctness and predictable performance: the lambda's typed interface and `std::accumulate` guarantee deterministic numeric behavior; incompatible arguments fail at compile time.

Together, these differences illustrate the trade-offs between flexibility, ergonomics, and performance/safety that drive language choice and program design.

Part 2: Memory Management

2.1 Section 3: Dynamic Memory Across Rust, Java, and C++

Programs & Profiling Setup

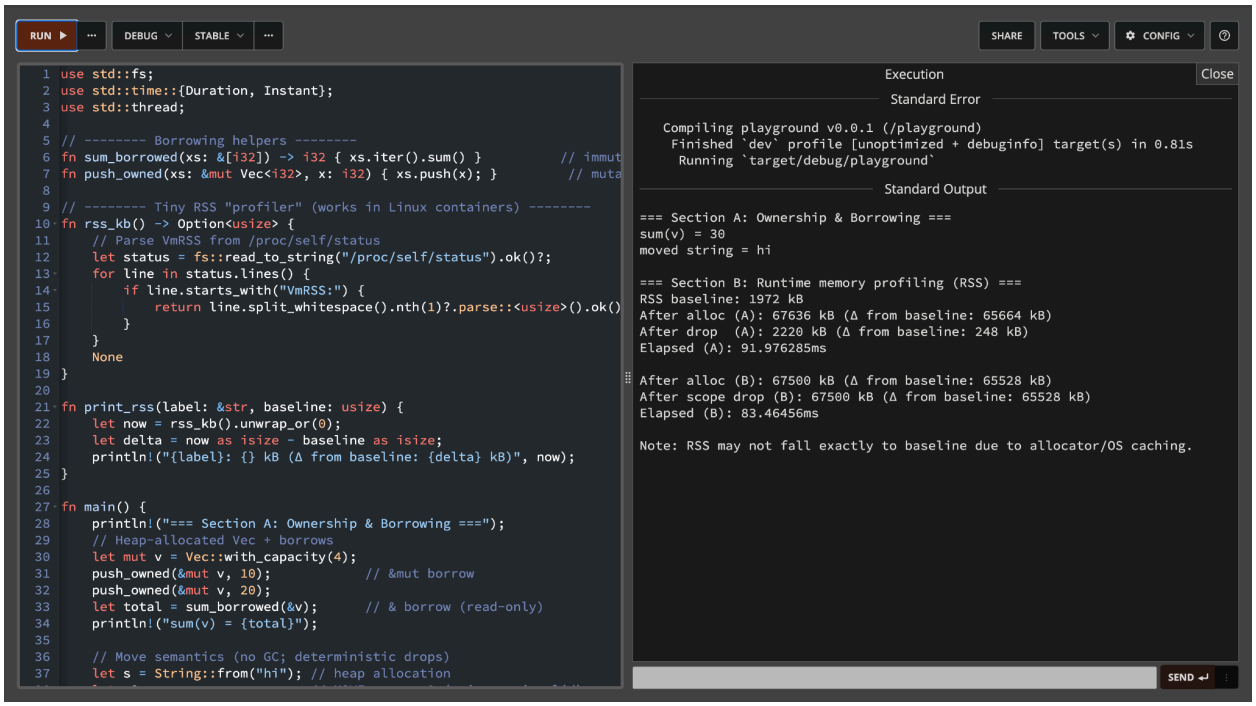
- **Rust**: Program demonstrates ownership & borrowing, deterministic frees via `drop`, and an intentional leak using `std::mem::forget`. A small in-process profiler reads RSS from `/proc/self/status` to show memory deltas and prints simple timings.
- **Java**: Program allocates arrays; scenario A drops the reference to show GC reclamation; scenario B toggles a "logical leak" by retaining references in a list. We measure heap via `Runtime.totalMemory() - freeMemory()` and nudge GC with a small `forceGc()` helper.
- **C++**: Program uses `new[]`/`delete[]` for a large buffer to show manual frees. scenario B toggles a leak by omitting `delete[]`. We read RSS like Rust for qualitative deltas.

  We intentionally touch each buffer so pages are committed, and we add a short sleep after frees so RSS/heap changes are visible even in online sandboxes.

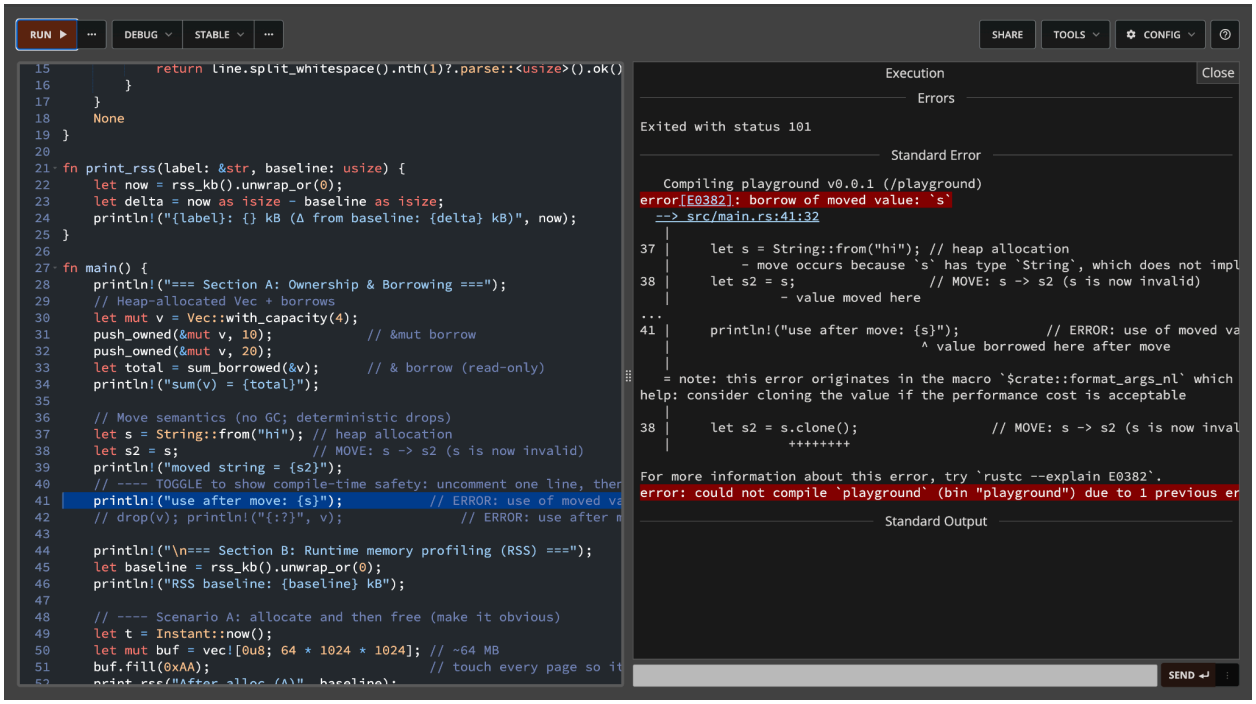
Rust - Ownership, Borrowing, and Deterministic Free

What the program does:

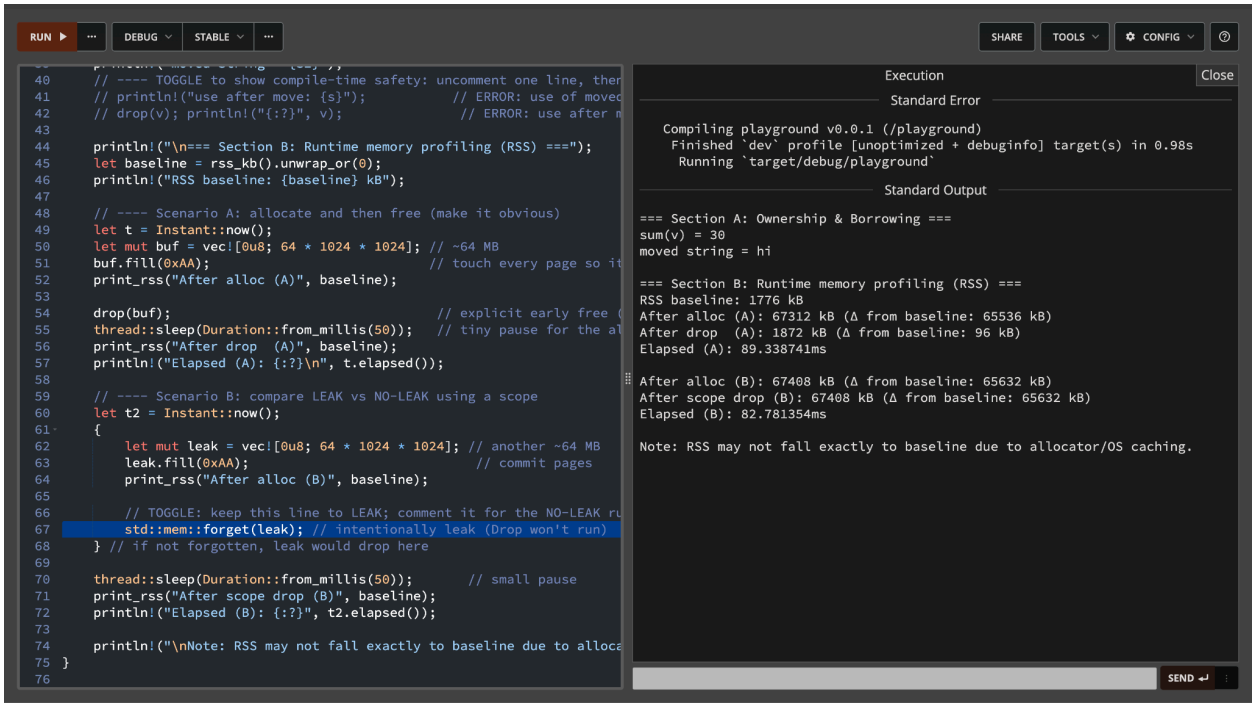
- Pushes into a `Vec<i32>` via a mutable borrow; reads via an immutable borrow (`&[i32]`).
- Demonstrates move semantics (`let s2 = s;`) and shows a compile-time error if `s` is used afterward.
- Scenario A: allocate ~64 MB `Vec<u8>`, touch pages, `drop` it → RSS falls toward baseline.
- Scenario B: allocate another ~64 MB in a scope; *leak* via `std::mem::forget` (or comment it for no-leak) → RSS stays high (leak) vs returns toward baseline (no-leak).



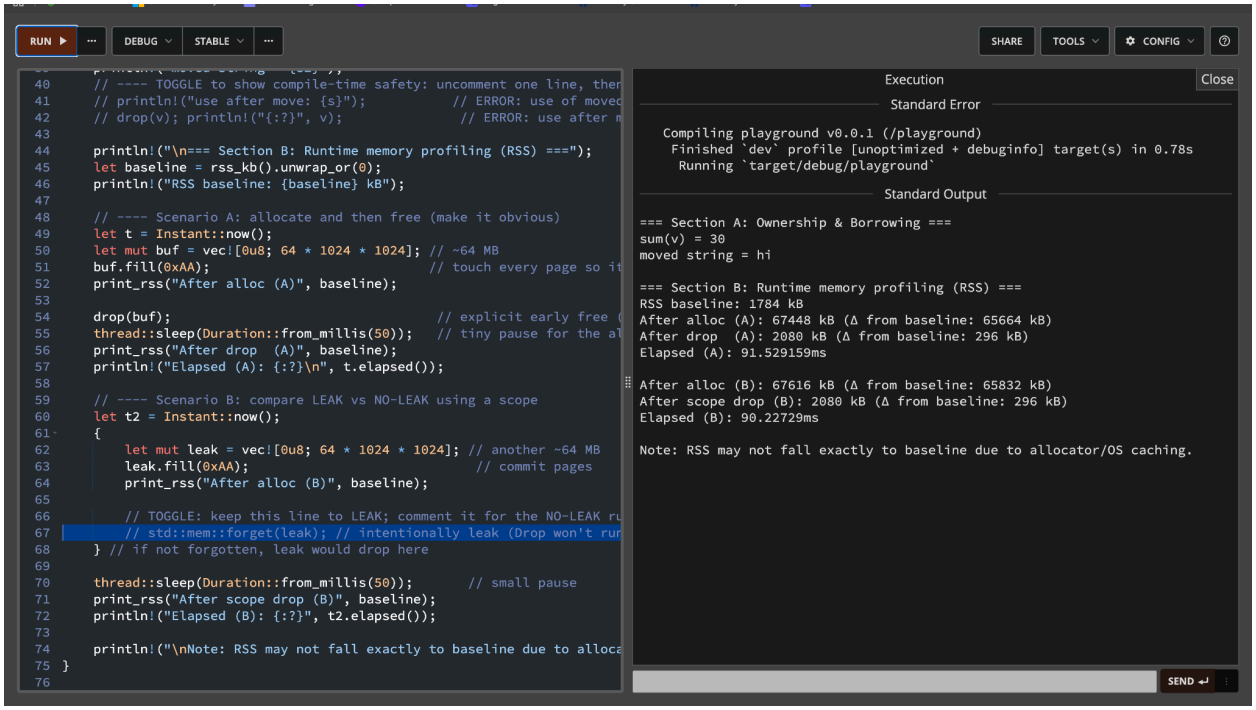
Normal run: baseline, alloc/drop A, alloc B, leak B.



E0382 compile error (“borrow/use of moved value”).



Leak: “After scope drop (B)” remains high.



No-leak: “After scope drop (B)” near baseline.

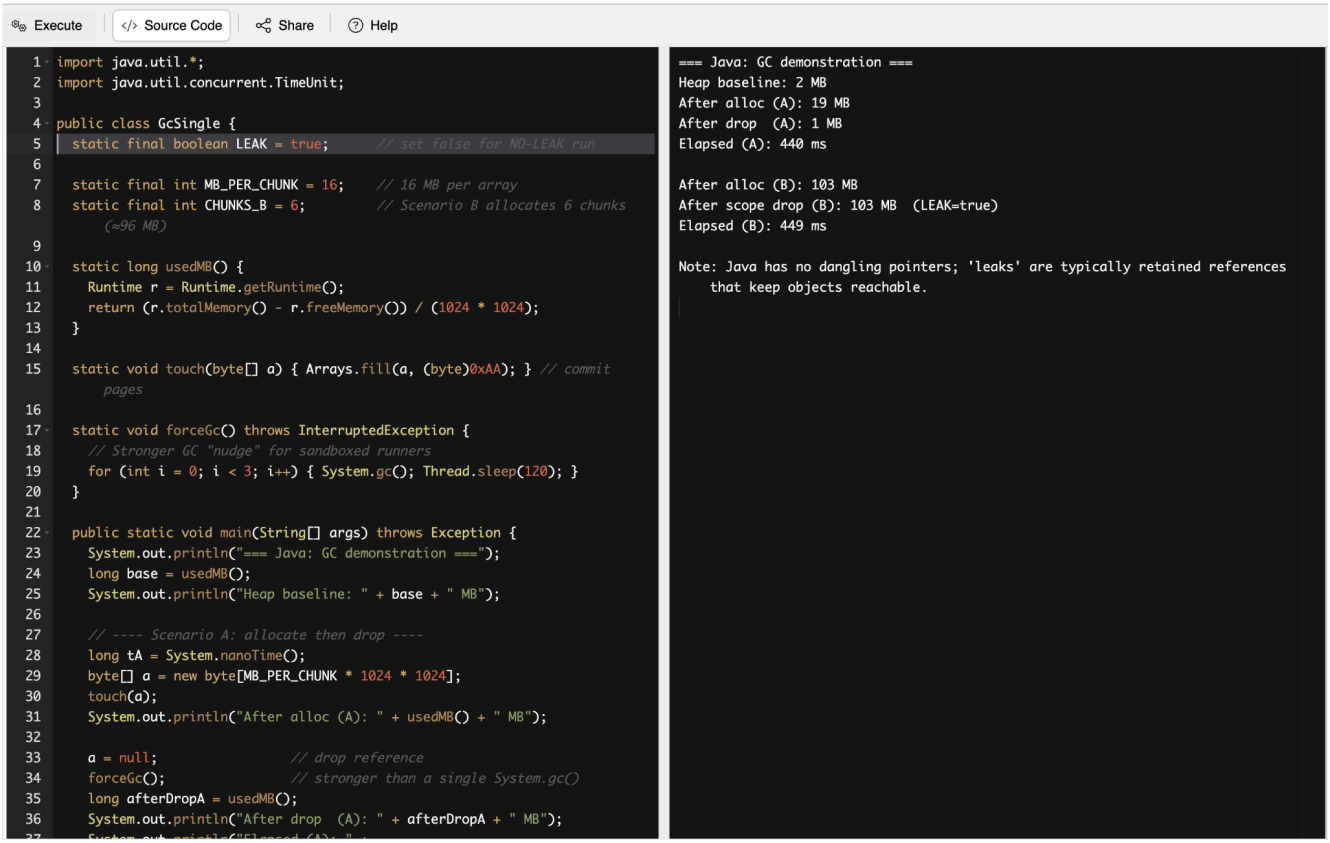
Takeaways:

- Safety by construction: use-after-move is blocked at compile time; safe Rust prevents dangling pointers/UAF.
- Deterministic deallocation: memory releases at scope end or `drop`, visible as a downward RSS change.
- Leaks require intent (`mem::forget`); otherwise they’re rare in safe Rust (Klabnik & Nichols, 2018).

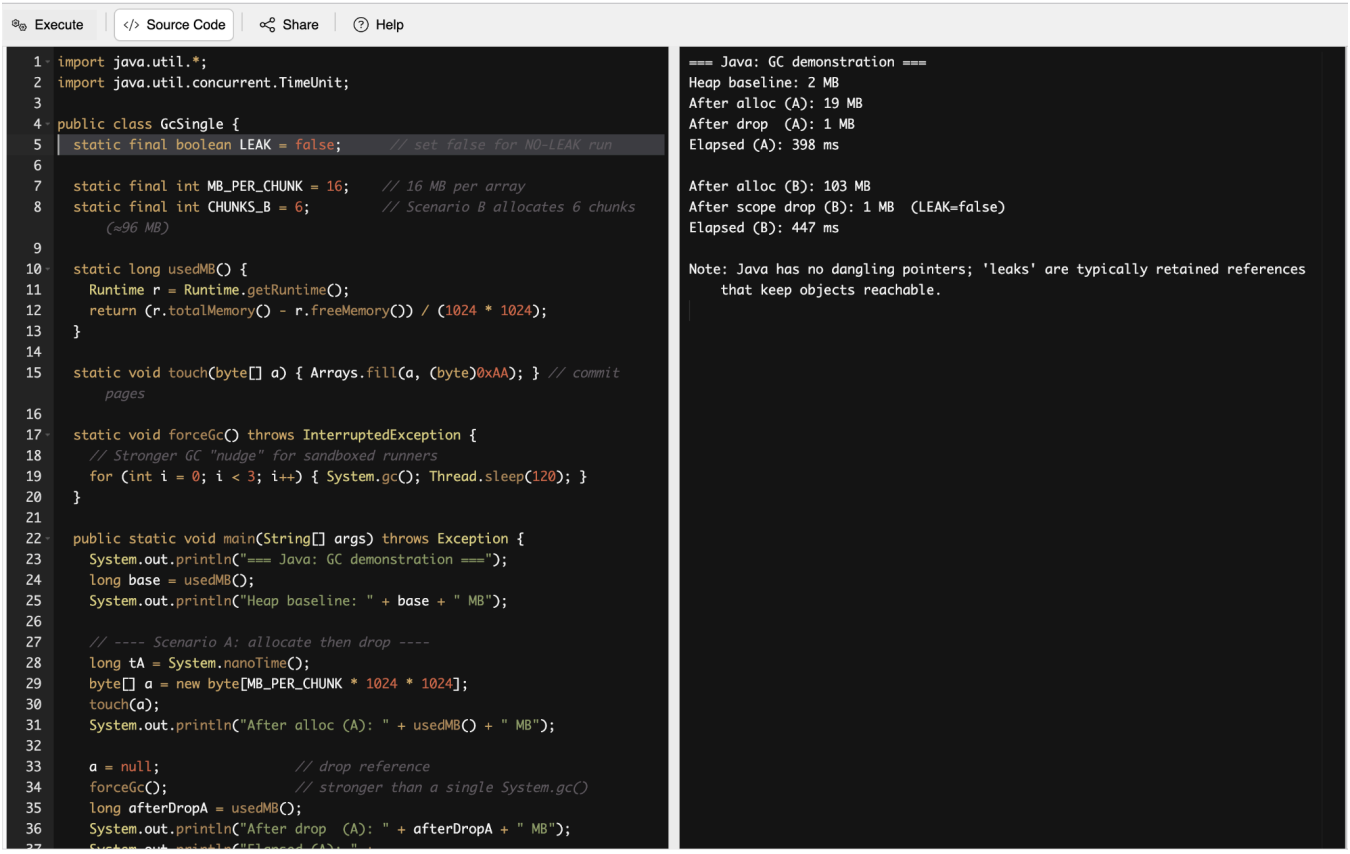
Java - Garbage Collection and Retained-Reference “Leaks”

What the program does:

- Scenario A: allocate ~16–32 MB; null the reference; call `forceGc()` → heap usage decreases.
- Scenario B: allocate multiple chunks; LEAK=true retains them in a list (objects remain reachable); LEAK=false clears references before `forceGc()`.



LEAK=true: “After scope drop (B)” remains high.



LEAK=false: “After scope drop (B)” drops after GC.

Takeaways:

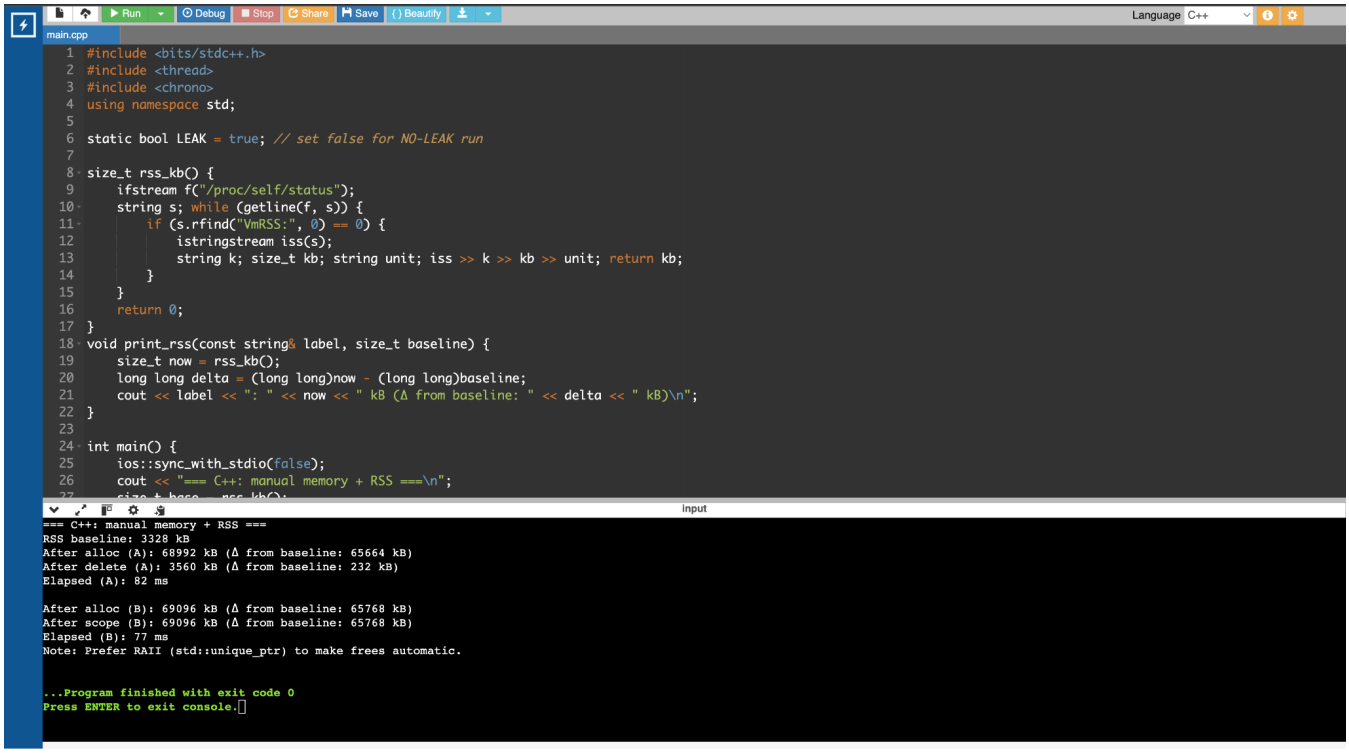
- Automatic reclamation: no explicit `free`; GC reclaims once objects are unreachable.
- No dangling pointers in user code; the typical failure mode is a logical leak - keeping references so GC can’t collect.
- Heap often remains above the initial baseline because the JVM keeps memory committed; the direction (down vs stuck) is the signal (Gosling et al., 2023).

C++ - Manual Allocation/Free and Leak Toggle

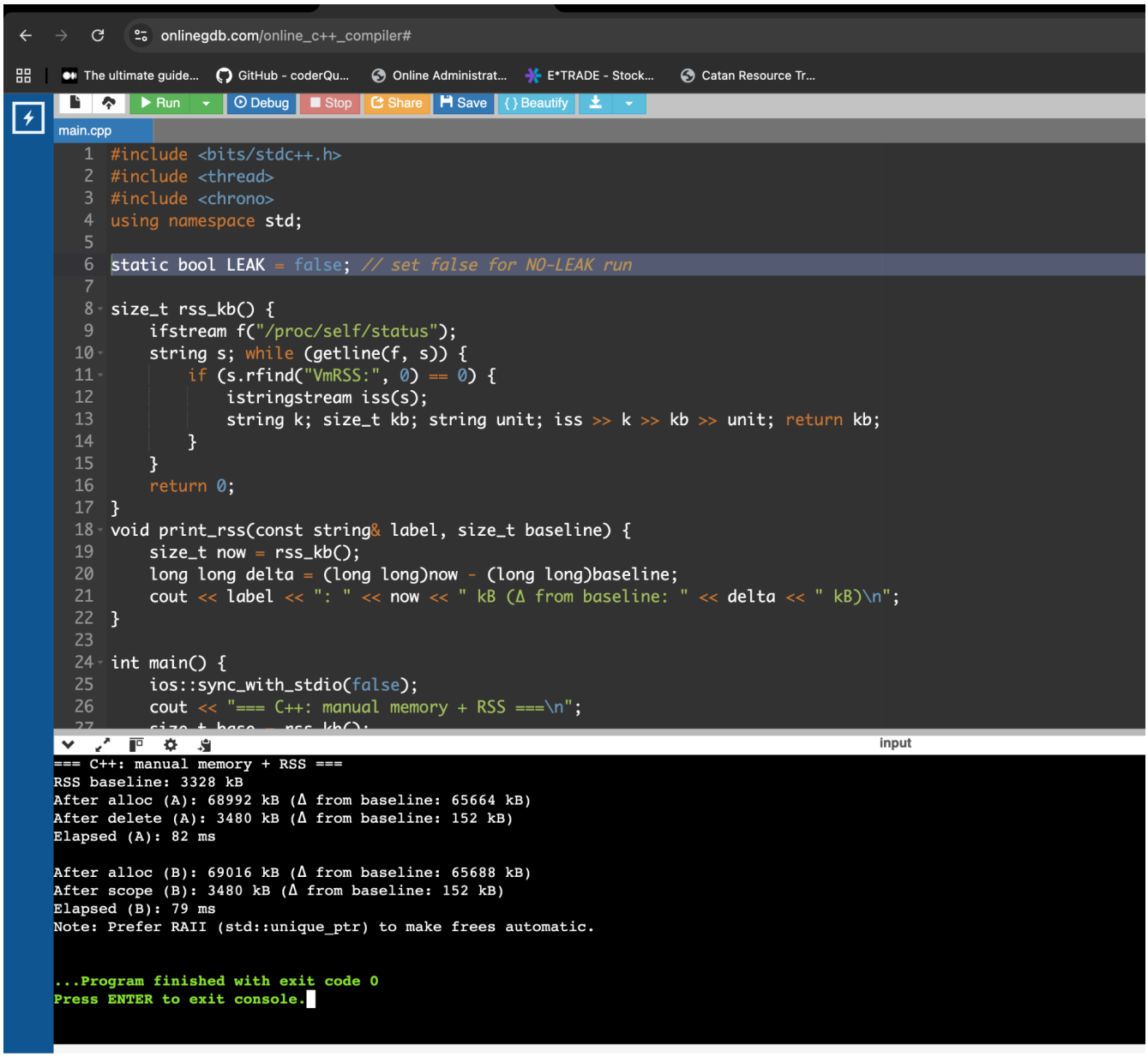
What the program does:

- Scenario A: `char* buf = new char[N];` → touch pages → `delete[] buf;` → RSS falls toward baseline.

- Scenario B: allocate another `new[]` in a scope; LEAK=true omits `delete[]` (leak); LEAK=false calls `delete[]` (no-leak).



LEAK=true: “After scope (B)” remains high.



LEAK=false: “After scope (B)” drops near baseline.

Takeaways:

- Full control, full responsibility: correctness depends on calling the right `delete`/`delete[]`.
- Easy to create true leaks or UAF without RAII; profiling shows retained RSS when leaks occur.
- Using RAII (`std::unique_ptr`, containers) eliminates most manual-free mistakes (Meyers, 2014).

Comparative Analysis

- **Who frees memory?**
  - **Rust**: Programmer writes the scopes; compiler inserts deterministic drops; safe Rust forbids UAF/dangling pointers.
  - **Java**: GC frees unreachable objects; programmer influences reachability and GC pressure, not the exact timing.
  - **C++**: Programmer frees explicitly (or uses RAII). Forgetting to free leaks; freeing too early risks UAF.

- **Primary failure mode**
  - **Rust**: Compile-time errors prevent typical runtime memory bugs; leaks require intent (e.g., `mem::forget`) or special cases (cycles).
  - **Java**: Retained references → "logical leaks"; no dangling pointers in normal code.
  - **C++**: Leaks and dangling pointers/UAF if `delete`/lifetimes are mishandled; sanitizers help at runtime.

- **Determinism vs. latency**
  - **Rust/C++**: Deallocation is deterministic (on scope/`delete`), which is predictable for latency-sensitive code.
  - **Java**: Reclamation is non-deterministic; pauses/throughput trade-offs are managed by the GC.

- **Profiling signal**
  - **Rust/C++ (RSS)**: Visible drop after free; stays high when leaking.
  - **Java (used heap)**: Drops when references cleared and GC runs; stays high when references retained (Oracle, 2024).
  - Minor residual over baseline in Rust/C++ is normal allocator/OS caching; Java often keeps the heap committed.

Conclusion

The three programs clearly showcase the distinct memory models:

- **Rust**: Ownership/borrowing provide compile-time safety and deterministic frees; leaks require intent.
- **Java**: Garbage collection automatically reclaims memory; the main risk is retaining references.

- **C++**: Manual allocation/free offers control but risks leaks and UAF without RAII; profiling surfaces mistakes.

References

Gosling, J., Joy, B., Steele, G., Bracha, G., Buckley, A., Smith, D., & Venners, B. (2023). Java Language and Virtual Machine Specifications. *Oracle*. https://docs.oracle.com/javase/specs/

Klabnik, S., & Nichols, C. (2018). The Rust Programming Language. *No Starch Press*. https://doc.rust-lang.org/book/

Meyers, S. (2014). *Effective Modern C++*. O'Reilly Media.

Oracle. (2024). *JDK 24 Documentation*. https://docs.oracle.com/javase/

Stroustrup, B., & Sutter, H. (2024). C++ Core Guidelines. *ISO C++ Foundation*. https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines