

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

Oishani Ganguly

GitHub Repository:

[https://github.com/Oishani/MSCS632\\_Assignment2](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

The screenshot shows a Jupyter Notebook cell with the following code:

```
1 # Python : Calculate the sum of an array
2 def calculate_sum(arr)    # ← missing colon here
3     total = 0
4     for num in arr:
5         total += num
6     return total
7
8 numbers = [1, 2, 3, 4, 5]
9 result = calculate_sum(numbers)
10 print("Sum in Python :", result)
11
```

When executed, the output is:

```
File "/home/cg/root/68a1a6a891bc8/main.py", line 2
def calculate_sum(arr)    # ← missing colon here
                         ^^^^^^^^^^^^^^^^^^^^^^
SyntaxError: expected ':'
```

#### 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

The screenshot shows a code editor window with the following code:

```
// JavaScript : Calculate the sum of an array
function calculateSum(arr) {
  let total = 0;
  for (let num of arr) {
    total += num;
  }
  return total;
}

let numbers = [1, 2, 3, 4, 5];
let result = calculateSum(numbers);
console.log("Sum in JavaScript:", result); // ← missing closing parenthesis
```

On the right, the terminal output shows:

```
/home/cg/root/68a1def1d490e/script.js:12
  console.log("Sum in JavaScript:", result); // ← missing closing parenthesis
                                         ^^^^^^

SyntaxError: missing ) after argument list
  at internalCompileFunction (node:internal/vm:73:18)
  at wrapSafe (node:internal/modules/cjs/loader:1274:20)
  at Module._compile (node:internal/modules/cjs/loader:1320:27)
  at Module._extensions..js (node:internal/modules/cjs/loader:1414:10)
  at Module.load (node:internal/modules/cjs/loader:1197:32)
  at Module._load (node:internal/modules/cjs/loader:1013:12)
  at Function.executeUserEntryPoint [as runMain] (node:internal/modules/run_main:128:12)
  at node:internal/main/run_main_module:28:49

Node.js v18.19.1
```

## 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++

The screenshot shows a code editor window with the following code:

```
// C++ : Calculate the sum of an array
#include <iostream>
using namespace std;

int calculateSum(int arr[], int size) {
  int total = 0;
  for (int i = 0; i < size; i++) {
    total += arr[i];
  }
  return total;
}

int main () {
  int numbers[] = {1, 2, 3, 4, 5};
  int size = sizeof(numbers) / sizeof(numbers[0]);
  int result = calculateSum(numbers, size); // ← missing semicolon here
  cout << "Sum in C++ " << result << endl;
  return 0;
}
```

On the right, the terminal output shows:

```
main.cpp: In function 'int main()':
main.cpp:17:5: error: expected ',' or ';' before 'cout'
  17 |   cout << "Sum in C++ " << result << endl;
      |   ^
      |~~~
```

## 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++)
<b>When detected</b>	Parse time immediately before execution	Parse time before execution	Compile time (separate build step)
<b>Typical error type</b>	<code>SyntaxError</code> with precise expectation (e.g., “expected ‘:’”)	<code>SyntaxError</code> with named construct (e.g., “missing ) after argument list”)	Compiler diagnostic (e.g., “expected ‘;’ before ‘cout’”)
<b>Location accuracy</b>	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)
<b>Runtime impact</b>	Program doesn’t start; fix & re-run quickly	Program doesn’t start; fix & re-run quickly	No executable produced; must recompile after fixes
<b>Message clarity</b>	Friendly and specific; suggests the exact missing symbol	Short, specific grammar hint; concise	More terse, sometimes cryptic; errors can cascade from one typo

## Conclusion

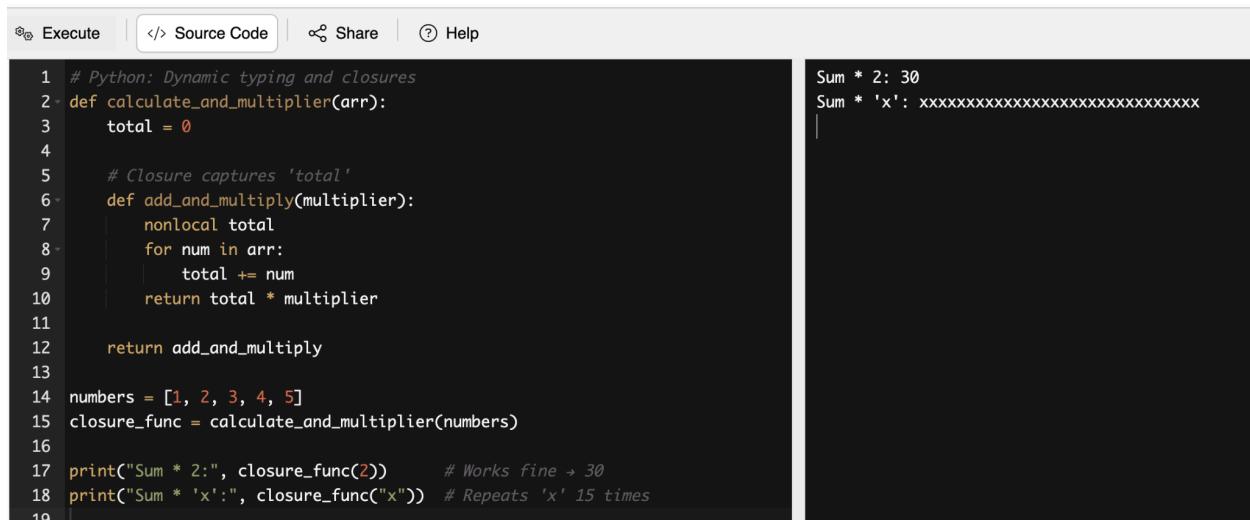
- Python and JavaScript both stop before any code runs; their messages point directly at the broken construct and are fast to iterate on.
- C++ reports errors during compilation; a single missing token can trigger multiple diagnostics, and the first error is often the most informative to fix.
- All three 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)



The screenshot shows a Jupyter Notebook cell with the following code:

```
1 # Python: Dynamic typing and closures
2 def calculate_and_multiplier(arr):
3     total = 0
4
5     # Closure captures 'total'
6     def add_and_multiply(multiplier):
7         nonlocal total
8         for num in arr:
9             total += num
10        return total * multiplier
11
12    return add_and_multiply
13
14 numbers = [1, 2, 3, 4, 5]
15 closure_func = calculate_and_multiplier(numbers)
16
17 print("Sum * 2:", closure_func(2))      # Works fine -> 30
18 print("Sum * 'x':", closure_func("x"))  # Repeats 'x' 15 times
```

The output of the code is:

```
Sum * 2: 30
Sum * 'x':xxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
```

### 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)

The screenshot shows a code editor interface with the following components:

- Top Bar:** Includes buttons for "Execute", "Source Code", "Share", and "Help".
- Code Area:** Displays the following JavaScript code:

```
1 // JavaScript: Dynamic typing, coercion, and closures
2 function calculateAndMultiplier(arr) {
3     let total = 0;
4
5     // Closure captures 'total'
6     return function(multiplier) {
7         for (let num of arr) {
8             total += num;
9         }
10        return total * multiplier;
11    };
12 }
13
14 let numbers = [1, 2, 3, 4, 5];
15 let closureFunc = calculateAndMultiplier(numbers);
16
17 console.log("Sum * 2:", closureFunc(2));      // Works fine → 30
18 console.log("Sum * 'x':", closureFunc("x")); // NaN due to coercion
19
```
- Output Area:** Shows the results of the console logs:

```
Sum * 2: 30
Sum * 'x': NaN
```

### 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](#)

```
1 // C++: Static typing, block scope, and closures
2 #include <iostream>
3 #include <vector>
4 #include <numeric>
5 using namespace std;
6
7 auto calculateAndMultiplier(const vector<int>& arr) {
8     int total = 0;
9
10    // Lambda closure capturing 'total' by reference.
11    // Uses accumulate so the sum is deterministic and easy to verify.
12    auto addAndMultiply = [&](int multiplier) {
13        total = accumulate(arr.begin(), arr.end(), 0);
14        return total * multiplier;
15    };
16
17    return addAndMultiply;
18 }
19
20 int main() {
21     vector<int> numbers = {1, 2, 3, 4, 5};
22     auto closureFunc = calculateAndMultiplier(numbers);
23
24     cout << "DEBUG sum: " << accumulate(numbers.begin(), numbers.end(), 0)
25         << "\n"; // 15
26     cout << "Sum * 2: " << closureFunc(2) << endl; // 30
27     // cout << "Sum * 'x': " << closureFunc("x") << endl; // Compile-time
28     // error: no viable conversion
29     return 0;
30 }
```

DEBUG sum: 15

Sum \* 2: 30

The screenshot shows a C++ code editor with the following code:

```

1 // C++: Static typing, block scope, and closures
2 #include <iostream>
3 #include <vector>
4 #include <numeric>
5 using namespace std;
6
7 auto calculateAndMultiplier(const vector<int>& arr) {
8     int total = 0;
9
10    // Lambda closure capturing 'total' by reference.
11    // Uses accumulate so the sum is deterministic and easy to verify.
12    auto addAndMultiply = [&](int multiplier) {
13        total = accumulate(arr.begin(), arr.end(), 0);
14        return total * multiplier;
15    };
16
17    return addAndMultiply;
18 }
19
20 int main() {
21     vector<int> numbers = {1, 2, 3, 4, 5};
22     auto closureFunc = calculateAndMultiplier(numbers);
23
24     cout << "DEBUG sum: " << accumulate(numbers.begin(), numbers.end(), 0)
25         << "\n"; // 15
26     cout << "Sum * 2: " << closureFunc(2) << endl; // 30
27     cout << "Sum * 'x': " << closureFunc('x') << endl; // Compile-time error
28         // no viable conversion
29     return 0;
}

```

The terminal window shows the following errors:

```

main.cpp: In function 'int main()':
main.cpp:26:41: error: no match for call to '(calculateAndMultiplier(const std::vector<int> &):<lambda(int)>) (const char [2])'
26 |     cout << "Sum * 'x': " << closureFunc("x") << endl; // Compile-time
      |                                         error: no viable conversion
      |                                         ^
main.cpp:12:27: note: candidate: 'calculateAndMultiplier(const std::vector<int> &):<lambda(int)>' (near match)
12 |     auto addAndMultiply = [&](int multiplier) {
      |             ^
main.cpp:12:27: note:   conversion of argument 1 would be ill-formed:
main.cpp:26:42: error: invalid conversion from 'const char*' to 'int' [-fpermissive]
      ://gcc.gnu.org/onlinedocs/gcc/Warning-Options.html#index-fpermissive
      -fpermissive ]8;; ]
26 |     cout << "Sum * 'x': " << closureFunc("x") << endl; // Compile-time
      |                                         error: no viable conversion
      |                                         ^
      |                                         |
      |                                         const char*

```

## 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).

The screenshot shows the RUST playground interface. On the left is the code editor with the following Rust code:

```
1 use std::fs;
2 use std::time::{Duration, Instant};
3 use std::thread;
4
5 // ----- Borrowing helpers -----
6 fn sum_borrowed(xs: &[i32]) -> i32 { xs.iter().sum() }           // immut
7 fn push_owned(xs: &mut Vec<i32>, x: i32) { xs.push(x); }          // muta
8
9 // ----- Tiny RSS "profiler" (works in Linux containers) -----
10 fn rss_kb() -> Option<u64> {
11     // Parse VmRSS from /proc/self/status
12     let status = fs::read_to_string("/proc/self/status").ok()?;
13     for line in status.lines() {
14         if line.starts_with("VmRSS:") {
15             return line.split_whitespace().nth(1)?.parse::<u64>().ok()
16         }
17     }
18     None
19 }
20
21 fn print_rss(label: &str, baseline: u64) {
22     let now = rss_kb().unwrap_or(0);
23     let delta = now as i64 - baseline as i64;
24     println!("{}: {} kB (Δ from baseline: {} kB)", label, now, delta);
25 }
26
27 fn main() {
28     println!("--- Section A: Ownership & Borrowing ---");
29     // Heap-allocated Vec + borrows
30     let mut v = Vec::with_capacity(4);
31     push_owned(&mut v, 10);           // &mut borrow
32     push_owned(&mut v, 20);           // &mut borrow
33     let total = sum_borrowed(&v);    // & borrow (read-only)
34     println!("sum(v) = {}", total);
35
36     // Move semantics (no GC; deterministic drops)
37     let s = String::from("hi"); // heap allocation
```

On the right is the execution output window. It shows the command being run: `Compiling playground v0.0.1 (/playground)`. It indicates the profile used: `dev`. It shows the time taken: `0.81s`. It then displays the results of the RSS profiler:

- Section A: Ownership & Borrowing:
  - sum(v) = 30
  - moved string = hi
- Section B: Runtime memory profiling (RSS):
  - RSS baseline: 1972 kB
  - After alloc (A): 67836 kB (Δ from baseline: 65664 kB)
  - After drop (A): 2220 kB (Δ from baseline: 248 kB)
  - Elapsed (A): 91.976285ms
- After alloc (B): 67500 kB (Δ from baseline: 65528 kB)
- After scope drop (B): 67500 kB (Δ from baseline: 65528 kB)
- Elapsed (B): 83.46456ms

Note: RSS may not fall exactly to baseline due to allocator/OS caching.

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

The screenshot shows the Rust playground interface. On the left, the code editor contains a file named `main.rs` with the following content:

```

15     return line.split_whitespace().nth(1)? .parse::<usize>() .ok()
16 }
17 }
18 None
19 }

20 fn print_rss(label: &str, baseline: usize) {
21     let now = rss_kb() .unwrap_or(0);
22     let delta = now as isize - baseline as isize;
23     println!("{}KB (Δ from baseline: {}KB)", now);
24 }
25 }

26 fn main() {
27     println!("==== Section A: Ownership & Borrowing ====");
28     // Heap-allocated Vec + borrows
29     let mut v = Vec::with_capacity(4);
30     push_owned(&mut v, 10); // &mut borrow
31     push_owned(&mut v, 20);
32     let total = sum_borrowed(&v); // & borrow (read-only)
33     println!("sum(v) = {total}");

34     // Move semantics (no GC; deterministic drops)
35     let s = String::from("hi"); // heap allocation
36     let s2 = s; // MOVE: s -> s2 (s is now invalid)
37     println!("moved string = {s2}");
38     // ---- TOGGLE to show compile-time safety: uncomment one line, then
39     // println!("use after move: {s}"); // ERROR: use of moved value
40     // drop(v); println!("{}:{}", v); // ERROR: use after move
41     // -----
42     println!("n==== Section B: Runtime memory profiling (RSS) ===");
43     let baseline = rss_kb() .unwrap_or(0);
44     println!("RSS baseline: {baseline} kB");

45     // ---- Scenario A: allocate and then free (make it obvious)
46     let t = Instant::now();
47     let mut buf = vec![0u8; 64 * 1024 * 1024]; // ~64 MB
48     buf.fill(0xAA); // touch every page so it
49     print_rss("After alloc (A)", baseline);
50 }

51 println!("nNote: RSS may not fall exactly to baseline due to allocator/OS caching.");
52
53 }
```

The right side of the interface shows the execution results. The "Errors" section displays the following message:

Compiling playground v0.0.1 (/playground)  
**error[E0382]: borrow of moved value: `s`**  
--> src/main.rs:41:32  
37 | let s = String::from("hi"); // heap allocation  
38 | - move occurs because `s` has type `String`, which does not impl  
39 | let s2 = s; // MOVE: s -> s2 (s is now invalid)  
40 | - value moved here  
41 | println!("use after move: {s}"); // ERROR: use of moved value  
42 | ^ value borrowed here after move  
43 |  
44 | = note: this error originates in the macro `\$crate::format\_args\_nl` which  
45 | help: consider cloning the value if the performance cost is acceptable  
46 |  
47 | let s2 = s.clone(); // MOVE: s -> s2 (s is now invalid)  
48 | +++++++  
49 |  
50 | For more information about this error, try `rustc --explain E0382`.  
**error: could not compile `playground` (bin "playground") due to 1 previous error**

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

The screenshot shows the Rust playground interface. On the left, the code editor contains a file named `main.rs` with the following content:

```

40 // ---- TOGGLE to show compile-time safety: uncomment one line, then
41 // println!("use after move: {s}"); // ERROR: use of moved value
42 // drop(v); println!("{}:{}", v); // ERROR: use after move
43
44 println!("n==== Section B: Runtime memory profiling (RSS) ===");
45 let baseline = rss_kb() .unwrap_or(0);
46 println!("RSS baseline: {baseline} kB");

47 // ---- Scenario A: allocate and then free (make it obvious)
48 let t = Instant::now();
49 let mut buf = vec![0u8; 64 * 1024 * 1024]; // ~64 MB
50 buf.fill(0xAA); // touch every page so it
51 print_rss("After alloc (A)", baseline);
52

53 drop(buf); // explicit early free
54 thread::sleep(Duration::from_millis(50)); // tiny pause for the OS
55 print_rss("After drop (A)", baseline);
56 println!("Elapsed (A): {}ms", t.elapsed());
57

58 // ---- Scenario B: compare LEAK vs NO-LEAK using a scope
59 let t2 = Instant::now();
60 {
61     let mut leak = vec![0u8; 64 * 1024 * 1024]; // another ~64 MB
62     leak.fill(0xAA); // commit pages
63     print_rss("After alloc (B)", baseline);
64
65     // TOGGLE: keep this line to LEAK; comment it for the NO-LEAK run
66     std::mem::forget(leak); // intentionally leak (Drop won't run)
67 } // if not forgotten, leak would drop here
68

69 thread::sleep(Duration::from_millis(50)); // small pause
70 print_rss("After scope drop (B)", baseline);
71 println!("Elapsed (B): {}ms", t2.elapsed());
72

73 println!("nNote: RSS may not fall exactly to baseline due to allocator/OS caching.");
74
75 }
76 }
```

The right side of the interface shows the execution results. The "Standard Output" section displays the following log:

Compiling playground v0.0.1 (/playground)  
Finished 'dev' profile [unoptimized + debuginfo] target(s) in 0.98s  
Running 'target/debug/playground'  
  
==== Section A: Ownership & Borrowing ===  
sum(v) = 30  
moved string = hi  
  
==== Section B: Runtime memory profiling (RSS) ===  
RSS baseline: 1776 kB  
After alloc (A): 67312 kB (Δ from baseline: 65536 kB)  
After drop (A): 1872 kB (Δ from baseline: 96 kB)  
Elapsed (A): 89.338741ms  
  
After alloc (B): 67408 kB (Δ from baseline: 65632 kB)  
After scope drop (B): 67408 kB (Δ from baseline: 65632 kB)  
Elapsed (B): 82.781354ms  
  
Note: RSS may not fall exactly to baseline due to allocator/OS caching.

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

The screenshot shows the Rust Playground interface with the following details:

- Toolbar:** RUN ▶, DEBUG ▾, STABLE ▾, ...
- Execution Tab:**
  - Compiling playground v0.0.1 (/playground)
  - Finished dev profile [unoptimized + debuginfo] target(s) in 0.78s
  - Running `target/debug/playground`
- Standard Output:**

```
==> Section A: Ownership & Borrowing ==>
sum(v) = 30
moved string = hi

==> Section B: Runtime memory profiling (RSS) ==>
RSS baseline: 1784 kB
After alloc (A): 67448 kB (Δ from baseline: 65664 kB)
After drop (A): 2080 kB (Δ from baseline: 296 kB)
Elapsed (A): 91.529159ms

==> After alloc (B): 67616 kB (Δ from baseline: 65832 kB)
After scope drop (B): 2080 kB (Δ from baseline: 296 kB)
Elapsed (B): 90.22729ms

Note: RSS may not fall exactly to baseline due to allocator/OS caching.
```
- Code Area:**

```
40 // ---- TOGGLE to show compile-time safety: uncomment one line, then
41 // println!("use after move: {s}");           // ERROR: use of moved
42 // drop(v); println!("{}?", v);             // ERROR: use after move
43
44 println!("\\n==== Section B: Runtime memory profiling (RSS) ===");
45 let baseline = rss_kb().unwrap_or(0);
46 println!("RSS baseline: {}kb", baseline);
47
48 // ---- Scenario A: allocate and then free (make it obvious)
49 let t = Instant::now();
50 let mut buf = vec![0u8; 64 * 1024 * 1024]; // ~64 MB
51 buf.fill(0xA);                           // touch every page so it
52 print_rss("After alloc (A)", baseline);
53
54 drop(buf);                            // explicit early free
55 thread::sleep(Duration::from_millis(50)); // tiny pause for the al
56 print_rss("After drop (A)", baseline);
57 println!("Elapsed (A): {}\\n", t.elapsed());
58
59 // ---- Scenario B: compare LEAK vs NO-LEAK using a scope
60 let t2 = Instant::now();
61 {
62     let mut leak = vec![0u8; 64 * 1024 * 1024]; // another ~64 MB
63     leak.fill(0xAA);                          // commit pages
64     print_rss("After alloc (B)", baseline);
65
66     // TOGGLE: keep this line to LEAK; comment it for the NO-LEAK run
67     // std::mem::forget(&leak); // intentionally leak (Drop won't run
68 } // if not forgotten, leak would drop here
69
70 thread::sleep(Duration::from_millis(50)); // small pause
71 print_rss("After scope drop (B)", baseline);
72 println!("Elapsed (B): {}?", t2.elapsed());
73
74 println!("\\nNote: RSS may not fall exactly to baseline due to allocat
75 }
76
```

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()`.

```

1- import java.util.*;
2 import java.util.concurrent.TimeUnit;
3
4 public class GcSingle {
5     static final boolean LEAK = true;           // set false for NO-LEAK run
6
7     static final int MB_PER_CHUNK = 16;        // 16 MB per array
8     static final int CHUNKS_B = 6;             // Scenario B allocates 6 chunks
9                 // (~96 MB)
10
11    static long usedMB() {
12        Runtime r = Runtime.getRuntime();
13        return (r.totalMemory() - r.freeMemory()) / (1024 * 1024);
14    }
15
16    static void touch(byte[] a) { Arrays.fill(a, (byte)0xAA); } // commit
17
18    static void forceGc() throws InterruptedException {
19        // Stronger GC "nudge" for sandboxed runners
20        for (int i = 0; i < 3; i++) { System.gc(); Thread.sleep(100); }
21    }
22
23    public static void main(String[] args) throws Exception {
24        System.out.println("==== Java: GC demonstration ===");
25        long base = usedMB();
26        System.out.println("Heap baseline: " + base + " MB");
27
28        // ---- Scenario A: allocate then drop ----
29        long tA = System.nanoTime();
30        byte[] a = new byte[MB_PER_CHUNK * 1024 * 1024];
31        touch(a);
32        System.out.println("After alloc (A): " + usedMB() + " MB");
33
34        a = null;                      // drop reference
35        forceGc();                     // stronger than a single System.gc()
36        long afterDropA = usedMB();
37        System.out.println("After drop (A): " + afterDropA + " MB");
38        System.out.println("Elapsed (A): " + (System.nanoTime() - tA));
39
40        // ---- Scenario B: drop then allocate ----
41        System.out.println("==== Java: GC demonstration ===");
42        System.out.println("Heap baseline: " + base + " MB");
43
44        a = null;                      // drop reference
45        forceGc();                     // stronger than a single System.gc()
46        long afterDropB = usedMB();
47        System.out.println("After drop (B): " + afterDropB + " MB");
48        System.out.println("Elapsed (B): " + (System.nanoTime() - tB));
49
50    }
51
52    static void printUsage() {
53        System.out.println("Usage: java GcSingle [NO-LEAK]");
54    }
55}

```

==== Java: GC demonstration ===  
 Heap baseline: 2 MB  
 After alloc (A): 19 MB  
 After drop (A): 1 MB  
 Elapsed (A): 440 ms

After alloc (B): 103 MB  
 After scope drop (B): 103 MB (LEAK=true)  
 Elapsed (B): 449 ms

Note: Java has no dangling pointers; 'leaks' are typically retained references that keep objects reachable.

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

```

1 import java.util.*;
2 import java.util.concurrent.TimeUnit;
3
4 public class GcSingle {
5     static final boolean LEAK = false; // set false for NO-LEAK run
6
7     static final int MB_PER_CHUNK = 16; // 16 MB per array
8     static final int CHUNKS_B = 6; // Scenario B allocates 6 chunks
9         (~96 MB)
10
11    static long usedMB() {
12        Runtime r = Runtime.getRuntime();
13        return (r.totalMemory() - r.freeMemory()) / (1024 * 1024);
14    }
15
16    static void touch(byte[] a) { Arrays.fill(a, (byte)0xAA); } // commit
17        pages
18
19    static void forceGc() throws InterruptedException {
20        // Stronger GC "nudge" for sandboxed runners
21        for (int i = 0; i < 3; i++) { System.gc(); Thread.sleep(120); }
22    }
23
24    public static void main(String[] args) throws Exception {
25        System.out.println("== Java: GC demonstration ==");
26        long base = usedMB();
27        System.out.println("Heap baseline: " + base + " MB");
28
29        // ---- Scenario A: allocate then drop ----
30        long tA = System.nanoTime();
31        byte[] a = new byte[MB_PER_CHUNK * 1024 * 1024];
32        touch(a);
33        System.out.println("After alloc (A): " + usedMB() + " MB");
34
35        a = null; // drop reference
36        forceGc(); // stronger than a single System.gc()
37        long afterDropA = usedMB();
38        System.out.println("After drop (A): " + afterDropA + " MB");
39        System.out.println("Elapsed (A): " + (System.nanoTime() - tA) / 1000000 + " ms");
40
41        // ---- Scenario B: drop then allocate ----
42        System.out.println("== Java: GC demonstration ==");
43        long tB = System.nanoTime();
44        byte[] b = new byte[MB_PER_CHUNK * 1024 * 1024];
45        touch(b);
46        System.out.println("After alloc (B): " + usedMB() + " MB");
47
48        b = null; // drop reference
49        forceGc(); // stronger than a single System.gc()
50        long afterDropB = usedMB();
51        System.out.println("After drop (B): " + afterDropB + " MB");
52        System.out.println("Elapsed (B): " + (System.nanoTime() - tB) / 1000000 + " ms");
53
54        Note: Java has no dangling pointers; 'leaks' are typically retained references
55            that keep objects reachable.

```

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).

```

1 #include <bits/stdc++.h>
2 #include <thread>
3 #include <chrono>
4 using namespace std;
5
6 static bool LEAK = true; // set false for NO-LEAK run
7
8 size_t rss_kb() {
9     ifstream f("/proc/self/status");
10    string s; while (getline(f, s)) {
11        if (s.rfind("VmRSS:", 0) == 0) {
12            istringstream iss(s);
13            string k; size_t kb; string unit; iss >> k >> kb >> unit; return kb;
14        }
15    }
16    return 0;
17 }
18 void print_rss(const string& label, size_t baseline) {
19     size_t now = rss_kb();
20     long long delta = (long long)now - (long long)baseline;
21     cout << label << ":" << now << " kB (\Delta from baseline: " << delta << " kB)\n";
22 }
23
24 int main() {
25     ios::sync_with_stdio(false);
26     cout << "==== C++: manual memory + RSS ===\n";
27     print_rss("Baseline", rss_kb());
28
29     cout << "After alloc (A): " << rss_kb();
30     cout << "After scope (B): " << rss_kb();
31     cout << "Elapsed (A): " << chrono::duration_cast<chrono::milliseconds>(chrono::high_resolution_clock::now() - start).count() << " ms\n";
32
33     cout << "After alloc (B): " << rss_kb();
34     cout << "After scope (B): " << rss_kb();
35     cout << "Elapsed (B): " << chrono::duration_cast<chrono::milliseconds>(chrono::high_resolution_clock::now() - start).count() << " ms\n";
36
37     cout << "Note: Prefer RAII (std::unique_ptr) to make frees automatic.\n";
38
39     ...Program finished with exit code 0
40 Press ENTER to exit console.

```

The terminal output shows the following memory usage statistics:

- Baseline:** RSS: 3328 kB
- After alloc (A):** RSS: 68992 kB (Delta: 65664 kB)
- Elapsed (A):** 77 ms
- Elapsed (B):** 82 ms
- After alloc (B):** RSS: 69096 kB (Delta: 65768 kB)
- After scope (B):** RSS: 69096 kB (Delta: 65768 kB)
- Elapsed (B):** 77 ms

Note: Prefer RAII (`std::unique_ptr`) to make frees automatic.

...Program finished with exit code 0  
Press ENTER to exit console.

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

The screenshot shows a web-based C++ compiler interface. The code in `main.cpp` is as follows:

```
1 #include <bits/stdc++.h>
2 #include <thread>
3 #include <chrono>
4 using namespace std;
5
6 static bool LEAK = false; // set false for NO-LEAK run
7
8 size_t rss_kb() {
9     ifstream f("/proc/self/status");
10    string s; while (getline(f, s)) {
11        if (s.rfind("VmRSS:", 0) == 0) {
12            istringstream iss(s);
13            string k; size_t kb; string unit; iss >> k >> kb >> unit; return kb;
14        }
15    }
16    return 0;
17 }
18 void print_rss(const string& label, size_t baseline) {
19     size_t now = rss_kb();
20     long long delta = (long long)now - (long long)baseline;
21     cout << label << ":" << now << " kB (\u0394 from baseline: " << delta << " kB)\n";
22 }
23
24 int main() {
25     ios::sync_with_stdio(false);
26     cout << "==== C++: manual memory + RSS ===\n";
27     size_t base = rss_kb();
```

The output window shows the following results:

```
==== C++: manual memory + RSS ===
RSS baseline: 3328 kB
After alloc (A): 68992 kB (\u0394 from baseline: 65664 kB)
After delete (A): 3480 kB (\u0394 from baseline: 152 kB)
Elapsed (A): 82 ms

After alloc (B): 69016 kB (\u0394 from baseline: 65688 kB)
After scope (B): 3480 kB (\u0394 from baseline: 152 kB)
Elapsed (B): 79 ms
Note: Prefer RAII (std::unique_ptr) to make frees automatic.
```

...Program finished with exit code 0
Press ENTER to exit console.

**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>