# 

# Syntax, Semantics, and Memory Management

Sri Sai Palamoor

MSCS-632-M80

Advanced Programming Languages

Sep 20 2025

# Part 1: Analyzing Syntax and Semantics

### **Section 1: Syntax Errors**

Below are three programs (Python, JavaScript, C++) intentionally modified to include syntax or semantic errors.

**python\_syntax\_error.py**

# Intentional error: using 'o' instead of 0 or initializing variable

def calculate\_sum(arr) :

total = o # <-- 'o' is not defined; will raise NameError at runtime

for num in arr:

total += num

return total

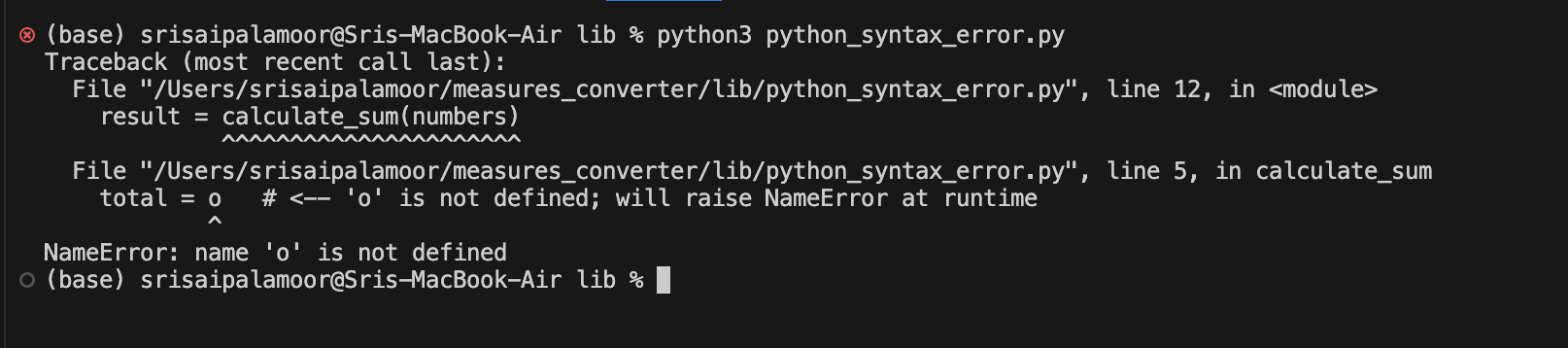
if \_\_name\_\_ == "\_\_main\_\_":

numbers = [1, 2, 3, 4, 5]

result = calculate\_sum(numbers)

print("Sum in Python :", result)

**Error produced:**

****

NameError: name 'o' is not defined

**Analysis:**

* Python detects this only at **runtime** (when executing the function).
* Error messages are descriptive and point to the exact line.

**Javascript\_syntax\_error.js**

// Intentional errors: 'o' undefined and malformed function call token

function calculateSum(arr) {

let total = o; // ReferenceError when executed: o is not defined

for (let num of arr) {

total += num;

}

return total;

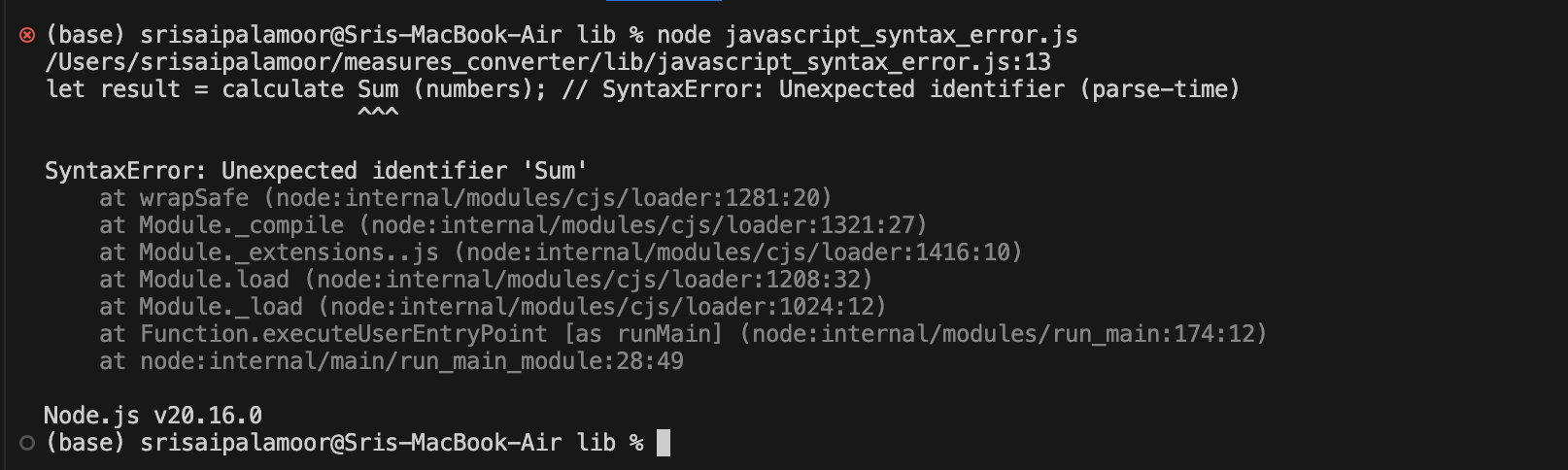
}

let numbers = [1, 2, 3, 4, 5];

let result = calculate Sum (numbers); // SyntaxError: Unexpected identifier (parse-time)

console.log("Sum in JavaScript:", result);

**Error produced:**

****

SyntaxError: Unexpected identifier 'Sum'

**Analysis:**

* JavaScript throws a SyntaxError before execution (parse-time).
* Unlike Python, execution won’t even start if syntax is invalid.

**Cpp\_syntax\_error.cpp**

// Intentional errors for demonstration: unknown identifier 'o' and stream operator misuse

#include <iostream>

using namespace std;

int calculateSum(int arr[], int size) {

int total = o; // 'o' not declared -> compile-time error

for (int i = o; i < size; i++) { // 'o' again

total += arr[i];

}

return total;

}

int main () {

int numbers [] = {1, 2, 3, 4, 5};

int size = sizeof(numbers) / sizeof( numbers [o]); // 'o' used inside sizeof

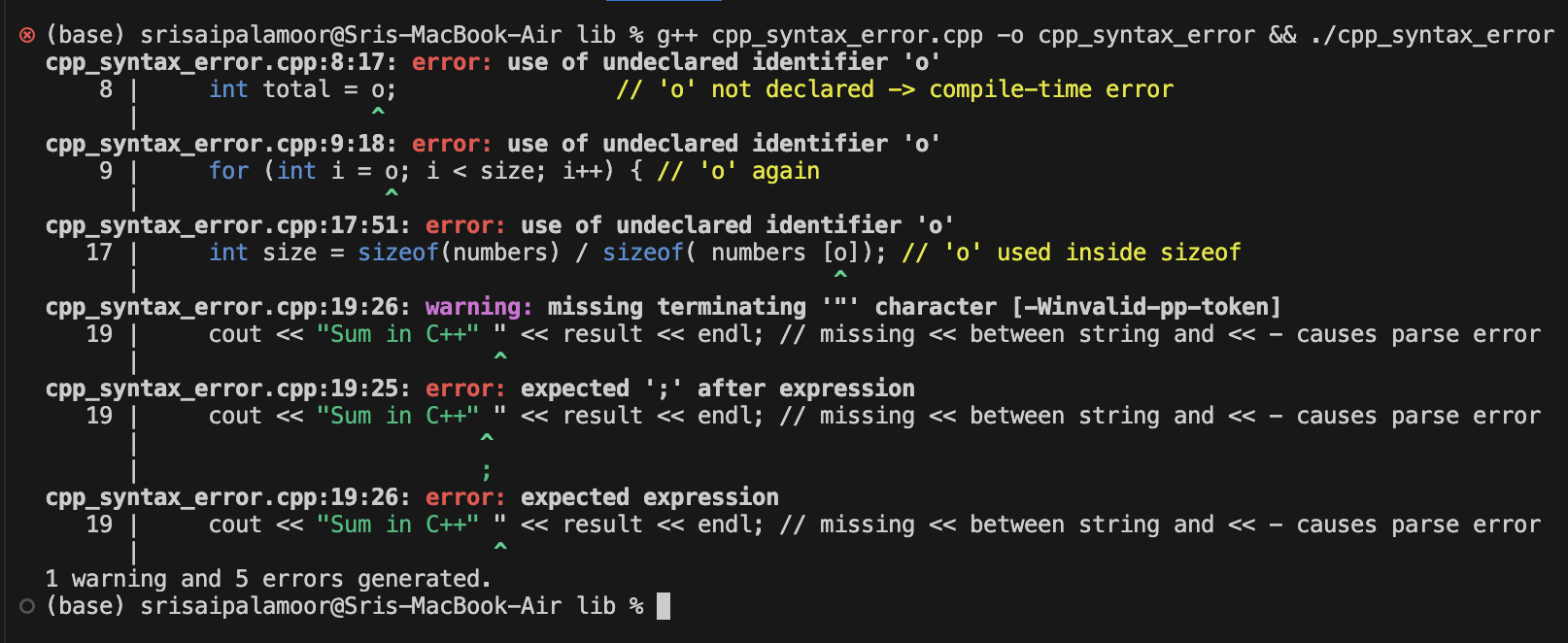
int result = calculateSum(numbers, size);

cout << "Sum in C++" " << result << endl; // missing << between string and << - causes parse error

return o; // returning 'o'

}

**Error produced:**

****

cpp\_syntax\_error.cpp:8:17: error: use of undeclared identifier 'o'

8 | int total = o; // 'o' not declared -> compile-time error

| ^

cpp\_syntax\_error.cpp:9:18: error: use of undeclared identifier 'o'

9 | for (int i = o; i < size; i++) { // 'o' again

| ^

cpp\_syntax\_error.cpp:17:51: error: use of undeclared identifier 'o'

17 | int size = sizeof(numbers) / sizeof( numbers [o]); // 'o' used inside sizeof

| ^

cpp\_syntax\_error.cpp:19:26: warning: missing terminating '"' character [-Winvalid-pp-token]

19 | cout << "Sum in C++" " << result << endl; // missing << between string and << - causes parse error

| ^

cpp\_syntax\_error.cpp:19:25: error: expected ';' after expression

19 | cout << "Sum in C++" " << result << endl; // missing << between string and << - causes parse error

| ^

| ;

cpp\_syntax\_error.cpp:19:26: error: expected expression

19 | cout << "Sum in C++" " << result << endl; // missing << between string and << - causes parse error

**Analysis:**

* C++ reports multiple compile-time errors before producing any executable.
* Messages are less beginner-friendly but very strict.

### **Comparison of Syntax Handling**

* **Python:** Errors appear at runtime; messages are friendly.
* **JavaScript:** Syntax errors caught at parse-time; execution doesn’t start.
* **C++:** Compile-time strict checking; no program runs unless fixed.

### 

### **Section 2: Language Features**

Here we examine **scoping/closures** and semantic differences.

**Python\_closure.py**

def make\_multiplier(factor):

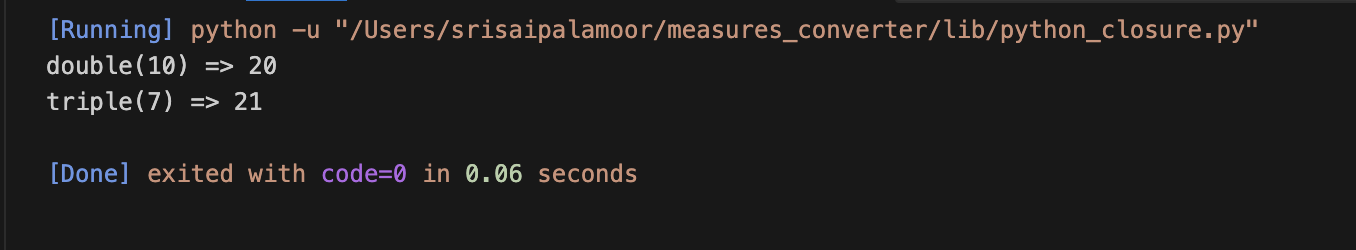
def multiply(x):

return x \* factor # closure captures variable

return multiply

double = make\_multiplier(2)

print(double(10)) # 20



**Js\_closure.js**

function makeAdder(a) {

return function(b) {

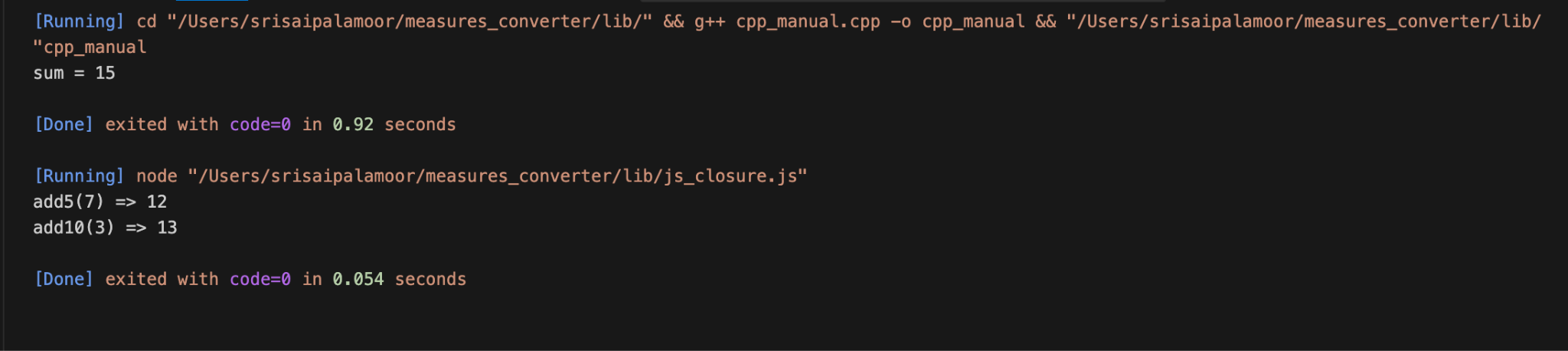
return a + b; // closure

};

}

const add5 = makeAdder(5);

console.log(add5(7)); // 12



**Cpp\_closure.cpp**

#include <iostream>

#include <functional>

using namespace std;

function<int(int)> make\_multiplier(int factor) {

return [factor](int x) { return x \* factor; };

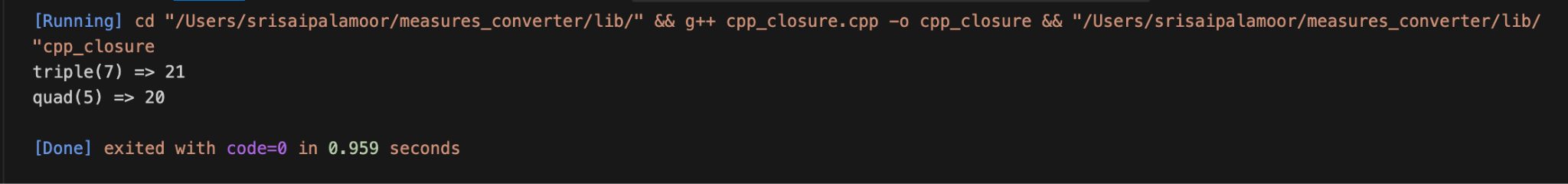
}

int main() {

auto triple = make\_multiplier(3);

cout << triple(7) << endl; // 21

}



### **Semantic Differences**

1. **Type Systems:**
   * Python: dynamically typed → types resolved at runtime.
   * JavaScript: loosely typed (type coercion).
   * C++: strongly & statically typed; errors detected at compile-time.
2. **Closures and Scopes:**
   * Python/JavaScript: closures are first-class, simple.
   * C++: closures use lambdas with explicit capture syntax.
3. **Error Handling Philosophy:**
   * Python favors readability, runtime detection.
   * JavaScript prioritizes execution in browsers, forgiving with coercion.
   * C++ prioritizes performance and safety, strict compilation.

# Part 2: Memory Management

**Rust\_owned.rs**

fn make\_vec() -> Vec<i32> {

let v = vec![1, 2, 3];

v // ownership moved

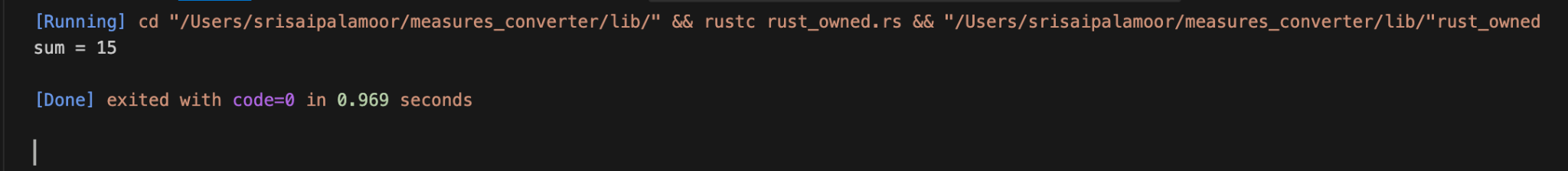
}

fn main() {

let v = make\_vec();

println!("sum = {}", v.iter().sum::<i32>());

}



* Rust uses ownership & borrowing.
* Memory is automatically freed when variables go out of scope.
* Prevents leaks & dangling pointers at compile-time.

**JavaGCExample.java**

public class JavaGCExample {

static class Node { int[] data = new int[100000]; }

public static void main(String[] args) {

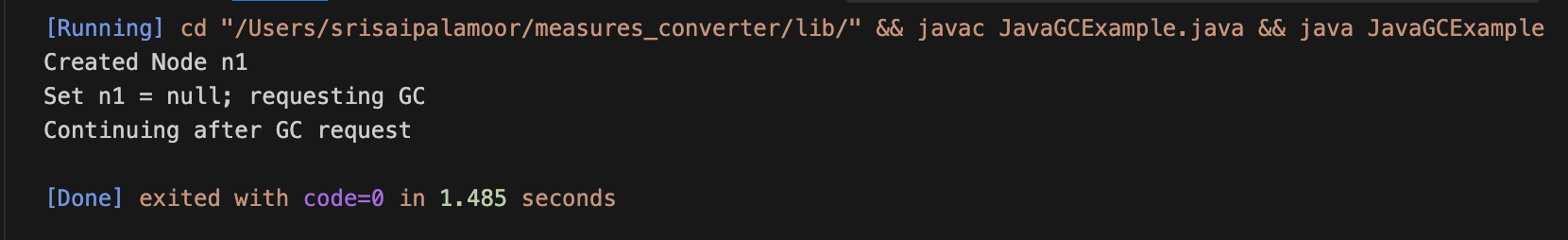
Node n1 = new Node();

n1 = null; // eligible for GC

System.gc(); // request GC

}

}



* Java uses Garbage Collection (GC).
* Developer doesn’t manually free memory.
* GC runs automatically, can cause pauses.

**Cpp\_manual.cpp**

#include <iostream>

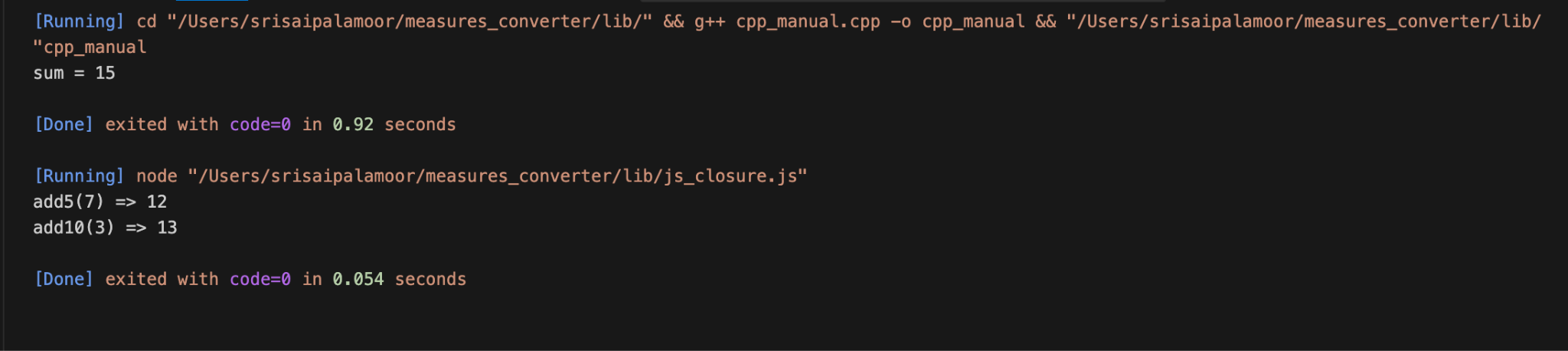
int main() {

int \*arr = new int[5];

for (int i = 0; i < 5; ++i) arr[i] = i + 1;

delete[] arr; // manual free

return 0;}



* C++ requires manual memory management.
* Forgetting delete causes leaks; deleting twice causes errors.
* Tools like Valgrind help detect problems.

### **Memory Management Comparison**

| **Language** | **Management Style** | **Pros** | **Cons** |
| --- | --- | --- | --- |
| **Rust** | Ownership & borrowing | Memory safety without GC | Learning curve |
| **Java** | Garbage Collection | Easy for developers | Performance overhead |
| **C++** | Manual new/delete | Full control, high performance | Risk of leaks/dangling pointers |

# GitHub Repository

<https://github.com/spalamoor39148/MSCS_632_Assignment2>

# Conclusion

This assignment highlights how different programming languages handle **syntax, semantics, and memory management**. Python and JavaScript prioritize ease of use and flexibility but handle errors at runtime or parse-time. C++ is strict and optimized for performance but requires careful developer attention. Rust introduces a unique memory model balancing safety and efficiency, while Java relies on garbage collection.

# References

* Bilge, L., & Dumitras, T. (2012). *Before we knew it: An empirical study of zero-day attacks in the real world*. Proceedings of the 2012 ACM Conference on Computer and Communications Security, 833–844. https://doi.org/10.1145/2382196.2382284
* Matsakis, N. D., & Klock, F. S. (2014). *The Rust language*. ACM SIGAda Ada Letters, 34(3), 103–104.
* Stroustrup, B. (2013). *The C++ Programming Language (4th ed.)*. Addison-Wesley.