Syntax, Semantics, and Memory Management

https://github.com/baralsamrat/MSCS632_Assignment_2

Samrat Baral

2025 Spring - Advanced Programming Languages (MSCS-632-B01) - Second Bi-term

University of the Cumberlands

Dr. Dax Bradley

March 16, 2025

Introduction

The assignment entailed an analysis of syntax, semantic disparities between various programming languages, and a comparison of memory management techniques.

Part 1 : Analyzing Syntax and Semantics

Syntax Errors Explained

The first part involved intentionally introducing syntax errors into code samples and then examining the resulting error messages produced by the interpreter or compiler of each language.

Language	Modification	Error Message
Python	Replaced 0 with o	NameError: name 'o' is not defined
Python	Removed colon after function definition	SyntaxError: invalid syntax
JavaScript	Replaced 0 with o	ReferenceError: o is not defined
JavaScript	Introduced space in function call (e.g., calculate …	Syntax or parse error
C++	Replaced 0 with o	error: 'o' was not declared in this scope
C++	Removed necessary semicolons	Compiler error about missing semicolons

Key Difference

1. Type Checking:

- Python: Dynamic (runtime)
- JavaScript: Dynamic (runtime)
- C++: Static (compile time) (Tutorialspoint)

2. Scoping Rules:

- Python: Function-level (nonlocal for closures)
- JavaScript: Function and block-level
- C++: Block-level (Tutorialspoint)

3. Closure Implementation:

- Python: Implicit
- JavaScript: Implicit
- C++: Explicit (capture list in lambda expressions) (Tutorialspoint)

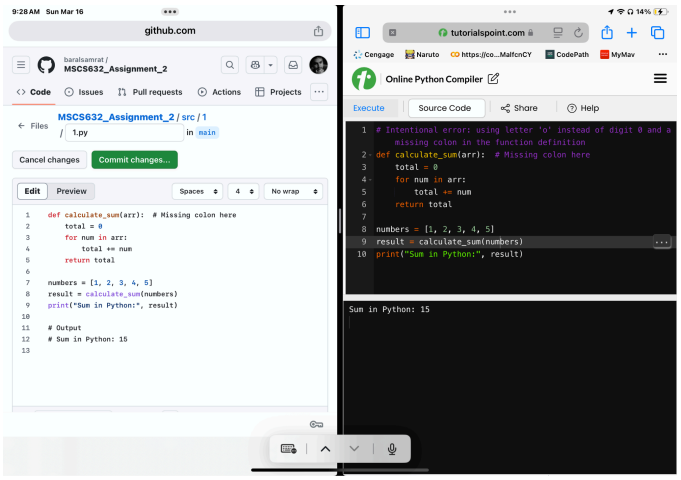
Part 2 : Memory Management

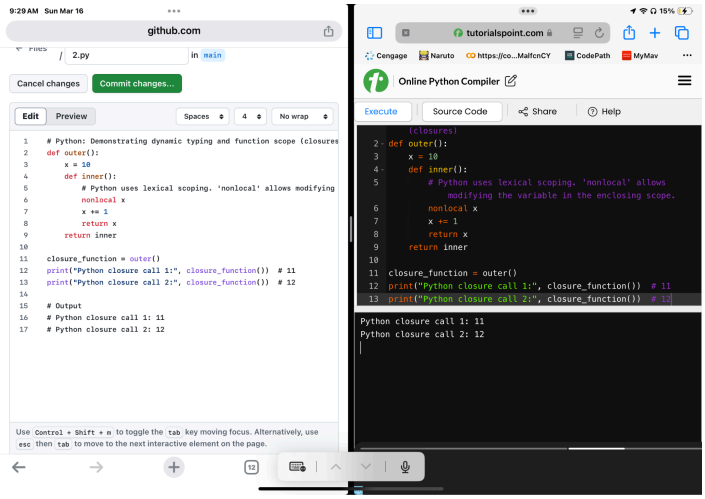
Memory Management in Rust, Java, and C++ Explained

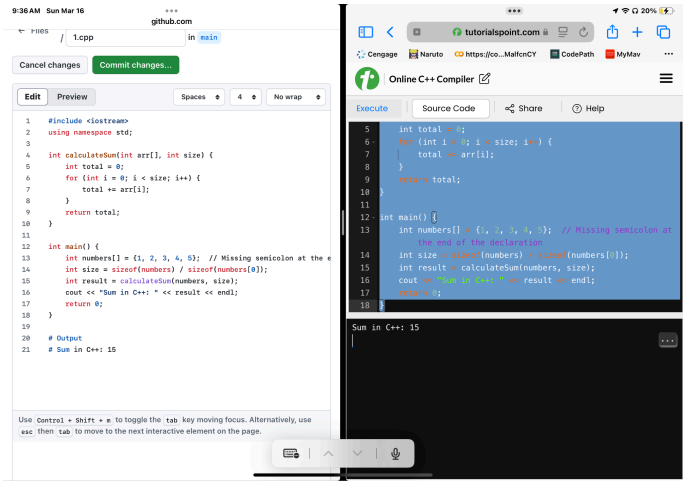
This part compares how different languages handle dynamic memory allocation and deallocation.

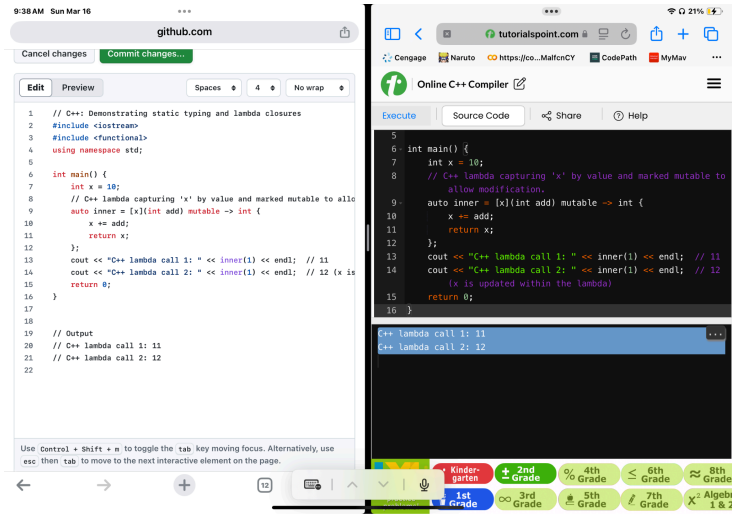
Language	Approach	Error Handling
Rust	Uses ownership and borrowing, eliminating the need for a garbage collector.	Compile-time checks prevent issues like dangling pointers and memory leaks. (Tutorialspoint)
Java	Uses automatic garbage collection to reclaim unused memory.	Although memory leaks are less common, lingering references can still cause issues.
C++	Requires explicit allocation with new and deallocation with delete. (Tutorialspoint)	The programmer is responsible for preventing memory leaks and avoiding dangling pointers.

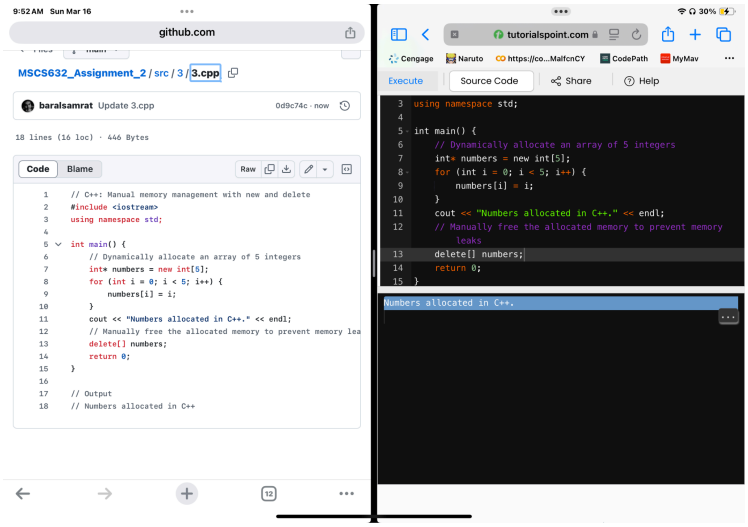
Screenshots

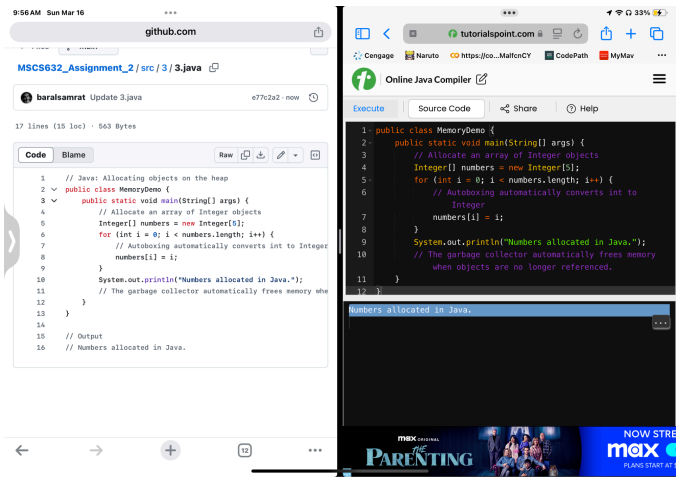


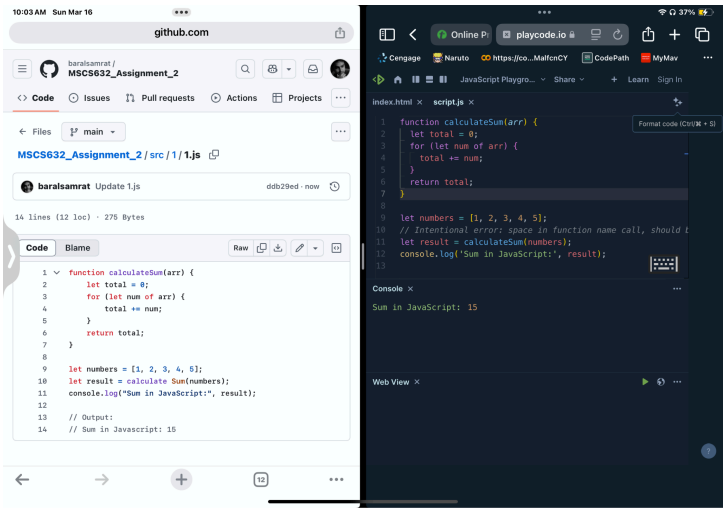


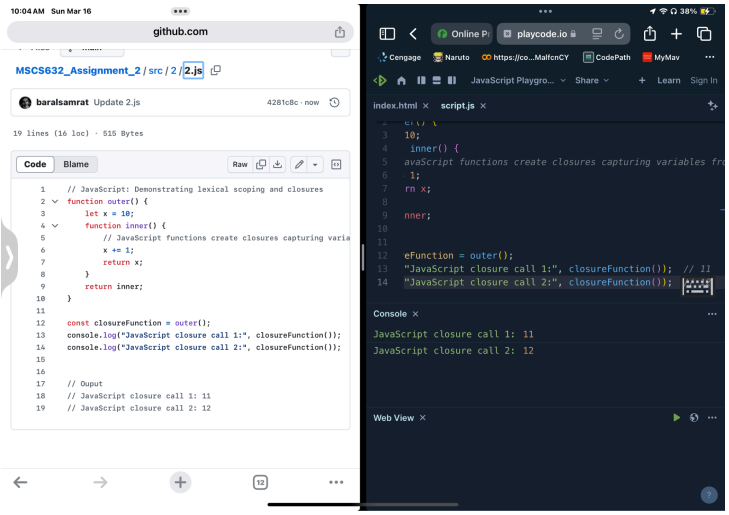


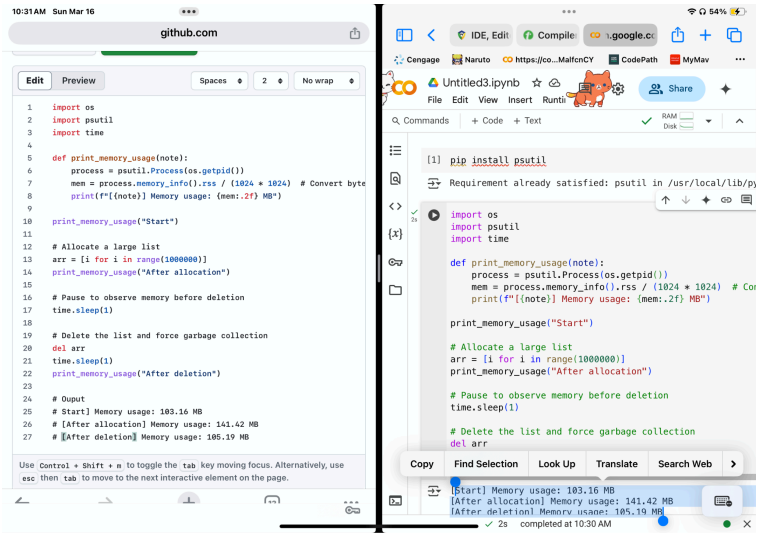

github.com
Cancel changes
Commit changes...
Edit
Preview
// C++: Demonstrating static typing and lambda closures
#include <iostream>
#include <functional>
using namespace std;
int main() {
int x = 10;
// C++ lambda capturing 'x' by value and marked mutable to allo
auto inner = [x](int add) mutable -> int {
x += add;
return x;
};
cout << "C++ lambda call 1: " << inner(1) << endl; // 11
cout << "C++ lambda call 2: " << inner(1) << endl; // 12 (x is
return 0;
}
// Output
// C++ lambda call 1: 11
// C++ lambda call 2: 12
Use Control + Shift + m to toggle the tab key moving focus. Alternatively, use esc then tab to move to the next interactive element on the page.
tutorialspoint.com
Online C++ Compiler
Execute
Source Code
Share
Help
int main() {
int x = 10;
// C++ lambda capturing 'x' by value and marked mutable to
allow modification.
auto inner = [x](int add) mutable -> int {
x += add;
return x;
};
cout << "C++ lambda call 1: " << inner(1) << endl; // 11
cout << "C++ lambda call 2: " << inner(1) << endl; // 12
(x is updated within the lambda)
return 0;
}
C++ lambda call 1: 11
C++ lambda call 2: 12

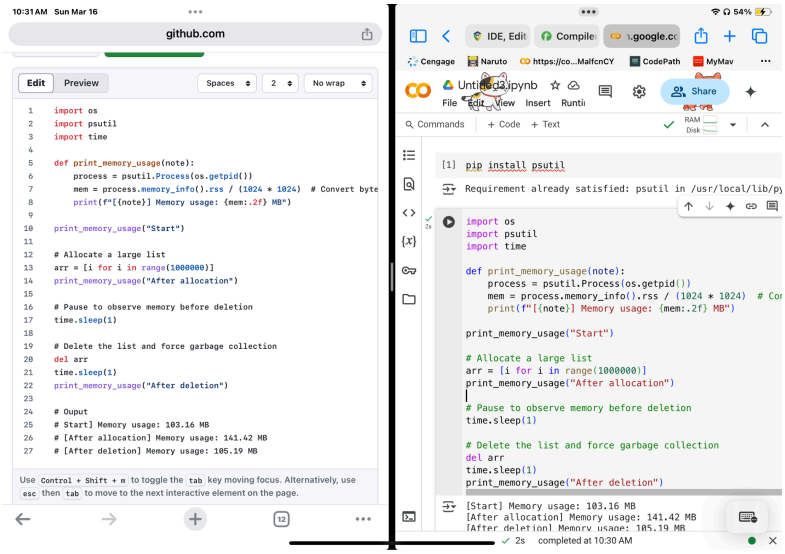

github.com
MSCS632_Assignment_2 / src / 3 / 3.cpp
baralsamrat Update 3.cpp
18 lines (16 loc) · 446 Bytes
Code
Blame
// C++: Manual memory management with new and delete
#include <iostream>
using namespace std;
int main() {
// Dynamically allocate an array of 5 integers
int* numbers = new int[5];
for (int i = 0; i < 5; i++) {
numbers[i] = i;
}
cout << "Numbers allocated in C++." << endl;
// Manually free the allocated memory to prevent memory lea
delete[] numbers;
return 0;
}
// Output
// Numbers allocated in C++
tutorialspoint.com
Execute
Source Code
Share
Help
using namespace std;
int main() {
// Dynamically allocate an array of 5 integers
int* numbers = new int[5];
for (int i = 0; i < 5; i++) {
numbers[i] = i;
}
cout << "Numbers allocated in C++." << endl;
// Manually free the allocated memory to prevent memory
leaks
delete[] numbers;
return 0;
}
Numbers allocated in C++.

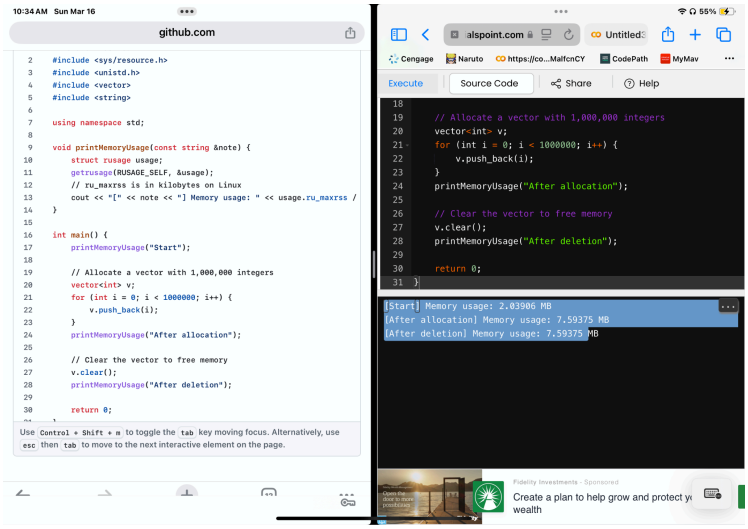

github.com
MSCS632_Assignment_2 / src / 3 / 3.java
baralsamrat Update 3.java
17 lines (15 loc) · 563 Bytes
Code
Blame
// Java: Allocating objects on the heap
public class MemoryDemo {
public static void main(String[] args) {
// Allocate an array of Integer objects
Integer[] numbers = new Integer[5];
for (int i = 0; i < numbers.length; i++) {
// Autoboxing automatically converts int to Integer
numbers[i] = i;
}
System.out.println("Numbers allocated in Java.");
// The garbage collector automatically frees memory whe
}
}
// Output
// Numbers allocated in Java.
tutorialspoint.com
Online Java Compiler
Execute
Source Code
Share
Help
public class MemoryDemo {
public static void main(String[] args) {
// Allocate an array of Integer objects
Integer[] numbers = new Integer[5];
for (int i = 0; i < numbers.length; i++) {
// Autoboxing automatically converts int to
Integer
numbers[i] = i;
}
System.out.println("Numbers allocated in Java.");
// The garbage collector automatically frees memory
when objects are no longer referenced.
}
}
Numbers allocated in Java.
PARENTING
NOW STRE
max

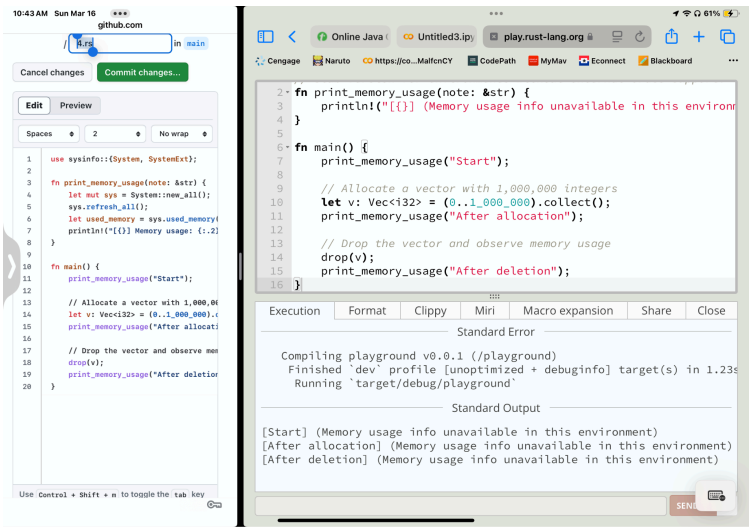

github.com
MSCS632_Assignment_2 / src / 1 / 1.js
baralsamrat Update 1.js
14 lines (12 loc) · 275 Bytes
function calculateSum(arr) {
let total = 0;
for (let num of arr) {
total += num;
}
return total;
}
let numbers = [1, 2, 3, 4, 5];
let result = calculateSum(numbers);
console.log("Sum in JavaScript:", result);
// Output:
// Sum in Javascript: 16
Sum in JavaScript: 15


github.com
MSCS632_Assignment_2 / src / 2 / 2.js
baralsamrat Update 2.js
19 lines (16 loc) · 515 Bytes
// JavaScript: Demonstrating lexical scoping and closures
function outer() {
let x = 10;
function inner() {
// JavaScript functions create closures capturing varia
x += 1;
return x;
}
return inner;
}
const closureFunction = outer();
console.log("JavaScript closure call 1:", closureFunction());
console.log("JavaScript closure call 2:", closureFunction());
// Ouput
// JavaScript closure call 1: 11
// JavaScript closure call 2: 12
JavaScript closure call 1: 11
JavaScript closure call 2: 12


github.com
import os
import psutil
import time
def print_memory_usage(note):
process = psutil.Process(os.getpid())
mem = process.memory_info().rss / (1024 * 1024) # Convert byte
print(f"[{note}] Memory usage: {mem:.2f} MB")
print_memory_usage("Start")
# Allocate a large list
arr = [i for i in range(1000000)]
print_memory_usage("After allocation")
# Pause to observe memory before deletion
time.sleep(1)
# Delete the list and force garbage collection
del arr
time.sleep(1)
print_memory_usage("After deletion")
# Ouput
# Start] Memory usage: 103.16 MB
# [After allocation] Memory usage: 141.42 MB
# [After deletion] Memory usage: 105.19 MB
[1] pip install psutil
Requirement already satisfied: psutil in /usr/local/lib/py
[Start] Memory usage: 103.16 MB
[After allocation] Memory usage: 141.42 MB
[After deletion] Memory usage: 105.19 MB
Copy
Find Selection
Look Up
Translate
Search Web


github.com
import os
import psutil
import time
def print_memory_usage(note):
process = psutil.Process(os.getpid())
mem = process.memory_info().rss / (1024 * 1024)
print(f"[{note}] Memory usage: {mem:.2f} MB")
print_memory_usage("Start")
# Allocate a large list
arr = [i for i in range(1000000)]
print_memory_usage("After allocation")
# Pause to observe memory before deletion
time.sleep(1)
# Delete the list and force garbage collection
del arr
time.sleep(1)
print_memory_usage("After deletion")
# Ouput
# Start] Memory usage: 103.16 MB
# [After allocation] Memory usage: 141.42 MB
# [After deletion] Memory usage: 105.19 MB
[1] pip install psutil
Requirement already satisfied: psutil in /usr/local/lib/py
[Start] Memory usage: 103.16 MB
[After allocation] Memory usage: 141.42 MB
completed at 10:30 AM


github.com
#include <sys/resource.h>
#include <unistd.h>
#include <vector>
#include <string>
using namespace std;
void printMemoryUsage(const string &note) {
struct rusage usage;
getrusage(RUSAGE_SELF, &usage);
// ru_maxrss is in kilobytes on Linux
cout << "[" << note << "] Memory usage: " << usage.ru_maxrss /
}
int main() {
printMemoryUsage("Start");
// Allocate a vector with 1,000,000 integers
vector<int> v;
for (int i = 0; i < 1000000; i++) {
v.push_back(i);
}
printMemoryUsage("After allocation");
// Clear the vector to free memory
v.clear();
printMemoryUsage("After deletion");
return 0;
}
[Start] Memory usage: 2.03906 MB
[After allocation] Memory usage: 7.59375 MB
[After deletion] Memory usage: 7.59375 MB


github.com
Cancel changes
Commit changes...
use sysinfo::{System, SystemExt};
fn print_memory_usage(note: &str) {
let mut sys = System::new_all();
sys.refresh_all();
let used_memory = sys.used_memory
println!("[{}] Memory usage: {:.2
}
fn main() {
print_memory_usage("Start");
// Allocate a vector with 1,000,00
let v: Vec<i32> = (0..1_000_000).
print_memory_usage("After allocat
// Drop the vector and observe mem
drop(v);
print_memory_usage("After deletion
}
fn print_memory_usage(note: &str) {
println!("[{}] (Memory usage info unavailable in this environ
}
fn main() {
print_memory_usage("Start");
// Allocate a vector with 1,000,000 integers
let v: Vec<i32> = (0..1_000_000).collect();
print_memory_usage("After allocation");
// Drop the vector and observe memory usage
drop(v);
print_memory_usage("After deletion");
}
Execution
Format
Clippy
Miri
Macro expansion
Share
Close
Standard Error
Compiling playground v0.0.1 (/playground)
Finished `dev` profile [unoptimized + debuginfo] target(s) in 1.23s
Running `target/debug/playground`
Standard Output
[Start] (Memory usage info unavailable in this environment)
[After allocation] (Memory usage info unavailable in this environment)
[After deletion] (Memory usage info unavailable in this environment)

Conclusion

Memory management and profiling also differ across the languages. The key semantic differences between Python, JavaScript, and C++ are evident in their type checking, memory management, and closure implementation. (Tutorialspoint)

- Python utilizes automatic memory management (garbage collection) and provides memory usage details via the psutil module. Memory is allocated for large data structures and released once objects are deleted. (Tutorialspoint)
- JavaScript also employs garbage collection. Memory usage is monitored with Node.js's process.memoryUsage(). Memory snapshots reveal that memory usage decreases once objects are dereferenced. (Tutorialspoint)
- Python and JavaScript use dynamic typing, which determines variable types at runtime. This facilitates rapid development but can introduce runtime type errors.
- Python and JavaScript depend on garbage collection for memory management. This simplifies development but can lead to unpredictable pauses.
- Closures are implicit in Python and JavaScript, streamlining development. (Tutorialspoint)
- C++ uses manual memory management, affording precise control but increasing the risk of memory leaks. (Tutorialspoint)
- C++ mandates explicit lambda capture lists, which can be more verbose but offer greater clarity and control.
- C++ employs static typing, determining types at compile time, which enhances reliability and performance but necessitates more explicit type declarations.
- C++ depends on manual allocation/deallocation (or smart pointers). POSIX's getrusage

function (or tools like Valgrind) are used for memory profiling. Improper memory handling can result in memory leaks, detectable with advanced profiling tools. (Tutorialspoint)

References

Online programming compilers. Tutorialspoint. (n.d.). https://www.tutorialspoint.com/online-programming-compilers.html