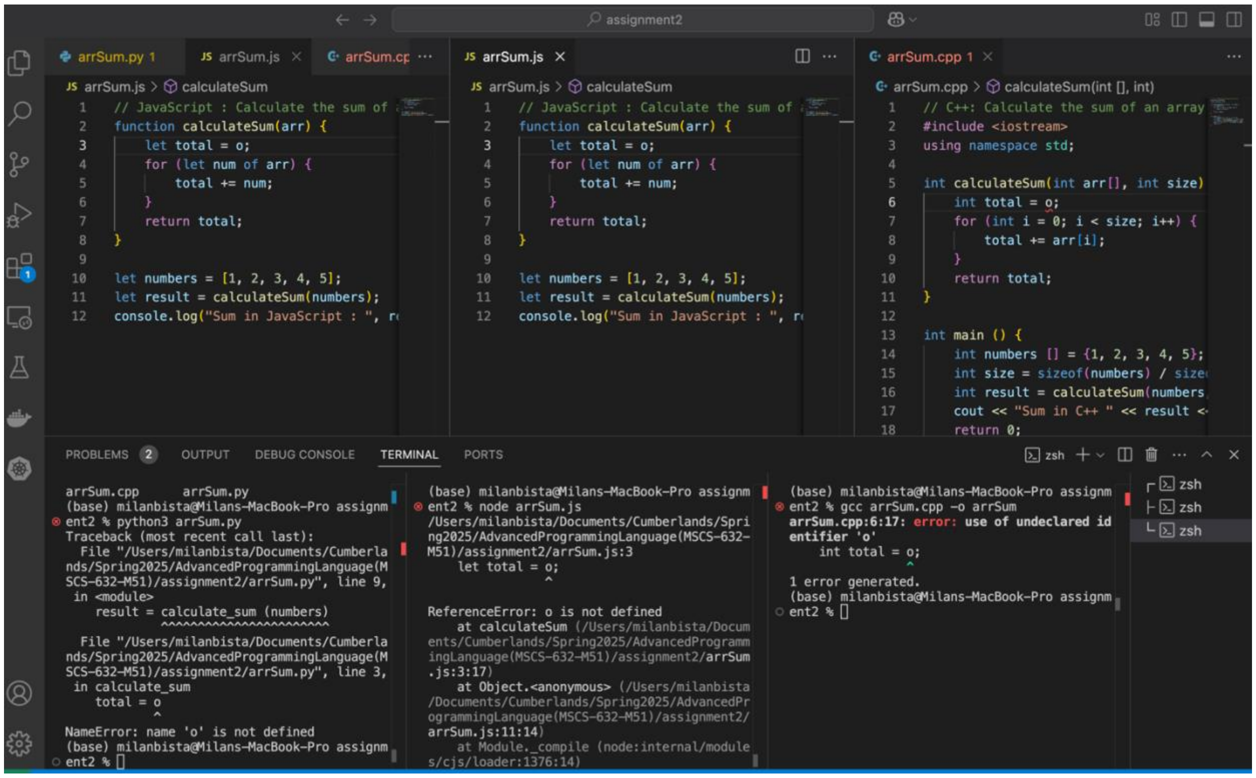
Assignment 2: Syntax, Semantics, and Memory Management

Milan Bista

2025 Spring – Advanced Programming Languages (MSCS-632-M51) – Full Term

Instructores: Vanessa Cooper / machica Mcclain

Syntax errors occur when a program does not conform to the grammatical rules of a programming language. These errors prevent execution in compiled languages like C++ and cause runtime failures in interpreted languages like Python and JavaScript. The screenshot provided captures code snippets in Python, JavaScript, and C++ along with their respective error outputs. Below is an analysis based on the captured errors.



```javascript
// JavaScript : Calculate the sum of
function calculateSum(arr) {
    let total = o;
    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 : ", r
```

```javascript
// JavaScript : Calculate the sum of
function calculateSum(arr) {
    let total = o;
    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 : ", r
```

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

int calculateSum(int arr[], int size)
    int total = o;
    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) / size
    int result = calculateSum(numbers
    cout << "Sum in C++ " << result <
    return 0;
```

```
arrSum.cpp      arrSum.py
(base) milanbista@Milans-MacBook-Pro assignm
ent2 % python3 arrSum.py
Traceback (most recent call last):
  File "/Users/milanbista/Documents/Cumberla
nds/Spring2025/AdvancedProgrammingLanguage(M
SCS-632-M51)/assignment2/arrSum.py", line 9,
 in <module>
    result = calculate_sum (numbers)
             ^^^^^^^^^^^^^^^^^^^^^^^

  File "/Users/milanbista/Documents/Cumberla
nds/Spring2025/AdvancedProgrammingLanguage(M
SCS-632-M51)/assignment2/arrSum.py", line 3,
 in calculate_sum
    total = o
            ^
NameError: name 'o' is not defined
(base) milanbista@Milans-MacBook-Pro assignm
ent2 % 
```

```
(base) milanbista@Milans-MacBook-Pro assignm
ent2 % node arrSum.js
/Users/milanbista/Documents/Cumberlands/Spri
ng2025/AdvancedProgrammingLanguage(MSCS-632-
M51)/assignment2/arrSum.js:3
    let total = o;
                ^

ReferenceError: o is not defined
    at calculateSum (/Users/milanbista/Docum
ents/Cumberlands/Spring2025/AdvancedProgramm
ingLanguage(MSCS-632-M51)/assignment2/arrSum
.js:3:17)
    at Object.<anonymous> (/Users/milanbista
/Documents/Cumberlands/Spring2025/AdvancedPr
ogrammingLanguage(MSCS-632-M51)/assignment2/
arrSum.js:11:14)
    at Module._compile (node:internal/module
s/cjs/loader:1376:14)
```

```
(base) milanbista@Milans-MacBook-Pro assignm
ent2 % gcc arrSum.cpp -o arrSum
arrSum.cpp:6:17: error: use of undeclared id
entifier 'o'
    int total = o;
                ^
1 error generated.
(base) milanbista@Milans-MacBook-Pro assignm
ent2 % 
```

In the Python code, the error arises because of the use of o instead of 0 to initialize the total variable. Python is an interpreted language, so the error is detected at runtime when the code attempts to use o, which has not been defined anywhere. This results in a NameError, with the message "name 'o' is not defined." Since Python stops execution at this point, the program does not proceed beyond the initialization step, making the error easy to spot and fix.

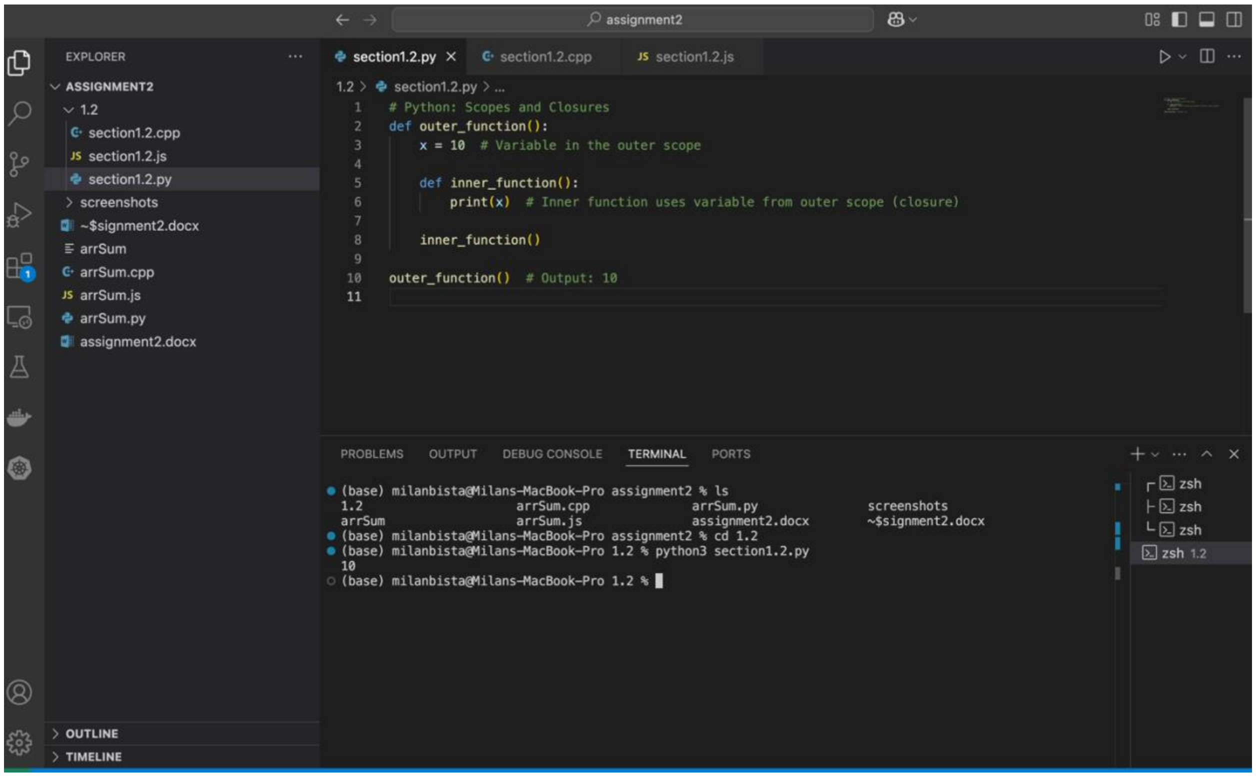
In the JavaScript code, the error occurs due to the use of an undefined variable o instead of 0. This results in a runtime ReferenceError with the message "o is not defined." The error is detected when the code is executed, and the JavaScript interpreter halts execution immediately at that point. This error is different from Python's NameError in that it specifically indicates that the variable o was never declared, making it clear that the issue is a missing definition in the scope.

Since C++ is a statically typed and compiled language, the compiler checks the code for errors before execution. It does not allow the use of variables that have not been declared. The compiler catches this issue during the compilation process and halts, preventing the program from being compiled into an executable. The error message clearly indicates that the identifier o has not been declared, and thus, the code cannot proceed until this issue is resolved.

1.2 Section 2: Analysis of Type Systems, Scopes, and Closures in Python, JavaScript, and C++

In this section, I have written three simple programs in **Python**, **JavaScript**, and **C++** to analyze how each language handles a specific feature such as **Type Systems** and **Scopes & Closures**. Additionally, I will identify three key semantic differences between the languages and explain how these differences affect program behavior and performance.

1. **Python Example**

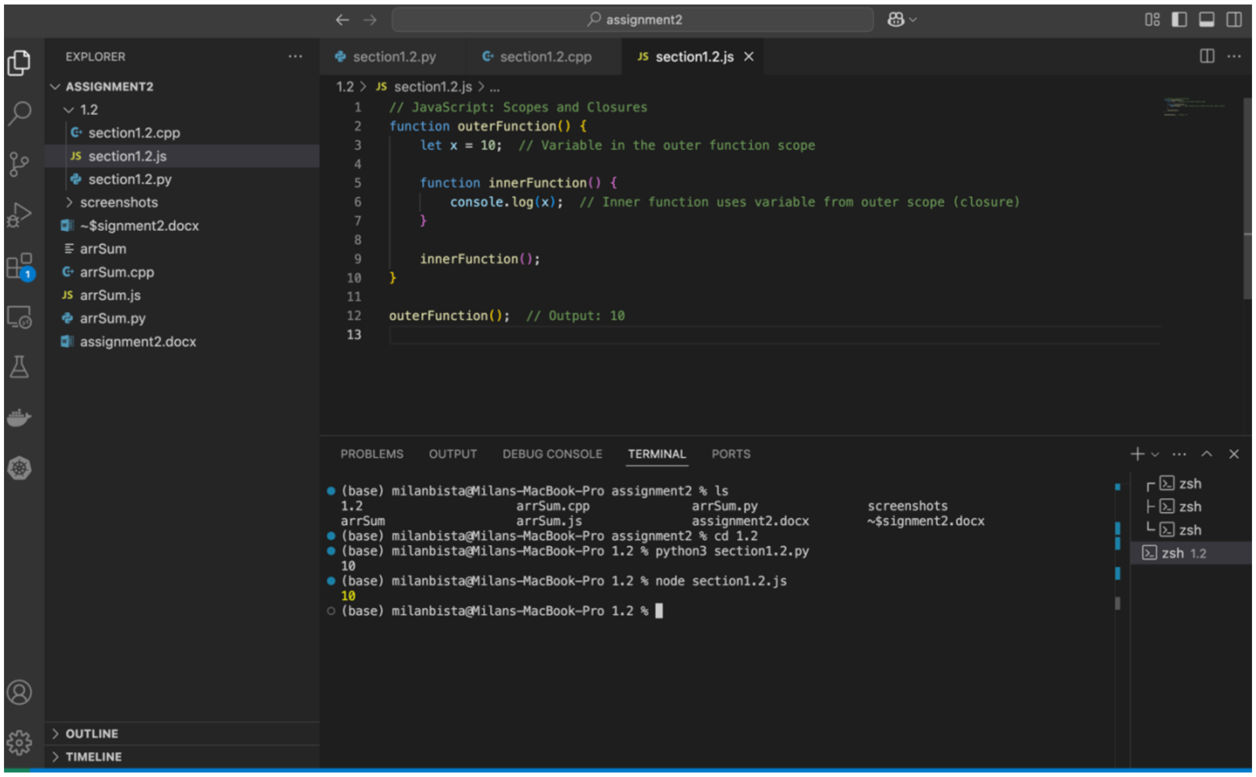


```python
# Python: Scopes and Closures
def outer_function():
    x = 10  # Variable in the outer scope

    def inner_function():
        print(x)  # Inner function uses variable from outer scope (closure)

    inner_function()

outer_function()  # Output: 10
```

```
(base) milanbista@Milans-MacBook-Pro assignment2 % ls
1.2                     arrSum.cpp              arrSum.py               screenshots
arrSum                  arrSum.js               assignment2.docx        ~$signment2.docx
(base) milanbista@Milans-MacBook-Pro assignment2 % cd 1.2
(base) milanbista@Milans-MacBook-Pro 1.2 % python3 section1.2.py
10
(base) milanbista@Milans-MacBook-Pro 1.2 %
```

- **Type System**: Python is a dynamically typed language, meaning variables are not bound to a specific type. The type of a variable is inferred at runtime.
- **Scopes and Closures**: In Python, variables from an outer scope can be accessed by inner functions, forming closures. The inner function remembers the environment in which it was created, thus having access to outer variables even after the outer function has returned.

2. JavaScript Example:



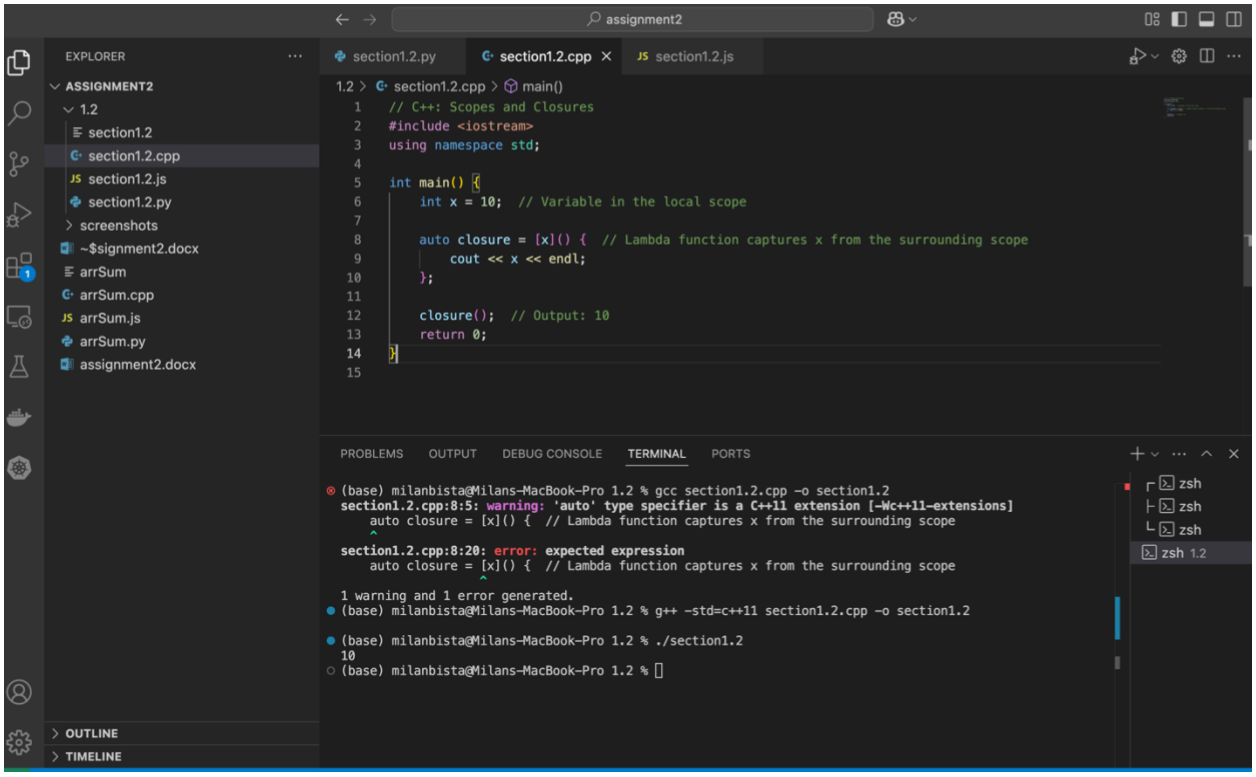
```
// JavaScript: Scopes and Closures
function outerFunction() {
    let x = 10;  // Variable in the outer function scope

    function innerFunction() {
        console.log(x);  // Inner function uses variable from outer scope (closure)
    }

    innerFunction();
}

outerFunction();  // Output: 10
```

```
(base) milanbista@Milans-MacBook-Pro assignment2 % ls
1.2                     arrSum.cpp              arrSum.py               screenshots
arrSum                  arrSum.js               assignment2.docx        ~$signment2.docx
(base) milanbista@Milans-MacBook-Pro assignment2 % cd 1.2
(base) milanbista@Milans-MacBook-Pro 1.2 % python3 section1.2.py
10
(base) milanbista@Milans-MacBook-Pro 1.2 % node section1.2.js
10
(base) milanbista@Milans-MacBook-Pro 1.2 %
```

- **Type System**: JavaScript is also a dynamically typed language, and variables are assigned types at runtime.
- **Scopes and Closures**: JavaScript functions create closures in a similar manner as Python. Inner functions can access variables from their enclosing scope, even after the outer function has executed, due to closures.

---

3. C++ Example:



```cpp
// C++: Scopes and Closures
#include <iostream>
using namespace std;

int main() {
    int x = 10;  // Variable in the local scope

    auto closure = [x]() {  // Lambda function captures x from the surrounding scope
        cout << x << endl;
    };

    closure();  // Output: 10
    return 0;
}
```

```
(base) milanbista@Milans-MacBook-Pro 1.2 % gcc section1.2.cpp -o section1.2
section1.2.cpp:8:5: warning: 'auto' type specifier is a C++11 extension [-Wc++11-extensions]
    auto closure = [x]() {  // Lambda function captures x from the surrounding scope
    ^
section1.2.cpp:8:20: error: expected expression
    auto closure = [x]() {  // Lambda function captures x from the surrounding scope
                   ^
1 warning and 1 error generated.
(base) milanbista@Milans-MacBook-Pro 1.2 % g++ -std=c++11 section1.2.cpp -o section1.2

(base) milanbista@Milans-MacBook-Pro 1.2 % ./section1.2
10
(base) milanbista@Milans-MacBook-Pro 1.2 % 
```

- **Type System**: C++ is a statically typed language. All variables must be explicitly declared with their types. This improves type safety and performance during compilation but requires more code for type handling.
- **Scopes and Closures**: C++ allows closures through lambda functions. Lambdas can capture variables from the surrounding scope by value or by reference. This is a key feature that allows lambda functions to behave similarly to closures in Python and JavaScript.

Key Semantic Differences Across the Languages:

1. **Type Systems**:
    - **Python** and **JavaScript** are dynamically typed, meaning variables can hold any type and types are determined at runtime. This provides flexibility but can lead to runtime errors if type mismatches occur.
    - **C++**, being statically typed, requires variables to be defined with specific types. This provides better type safety and optimization during compilation but reduces flexibility and increases verbosity in code.
2. **Scopes**:

    - **Python** and **JavaScript** share a similar scoping mechanism in which functions create new scopes, and inner functions can access variables from outer scopes. Python uses indentation for block scoping, while JavaScript uses curly braces {}.
    - **C++** uses block scoping, and variables are limited to their block scope unless explicitly passed to lambdas or closures.
3. **Closures**:
    - **Python** and **JavaScript** support closures naturally with functions. Inner functions can access variables from their enclosing scope, which is the core feature of closures.
    - **C++** provides closures through lambdas. Unlike Python and JavaScript, C++ lambdas require the explicit capture of variables from the outer scope, either by value or by reference.

Performance and Behavior:

- **Python** and **JavaScript** rely on **dynamic typing**, which leads to runtime overhead for type checking and can affect performance when handling large data or computationally intensive tasks. However, this allows for greater flexibility in code writing.
- **C++**, with its **static type system**, compiles to machine code that runs faster and more efficiently. However, the verbosity in type declarations and the need for explicit memory management can increase the complexity of the code.

In terms of **closures**:

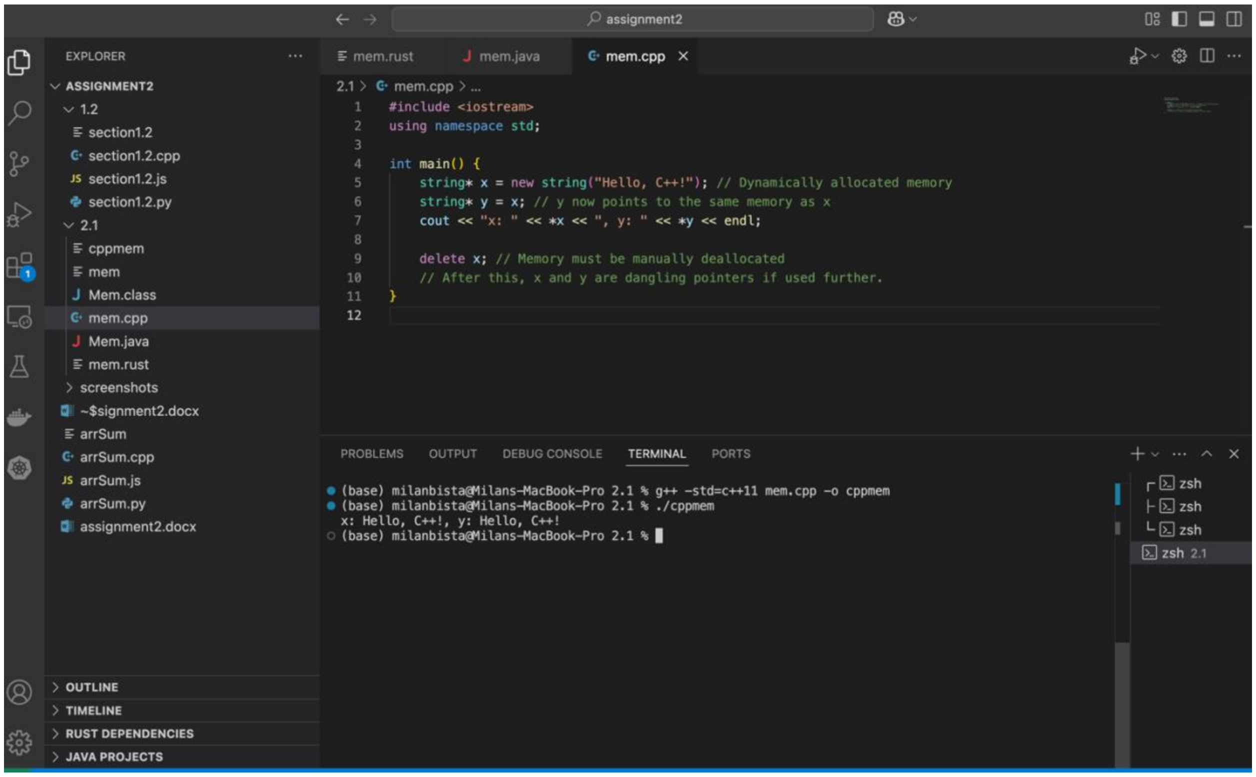
- **Python** and **JavaScript** handle closures, similarly, allowing variables to be captured by inner functions automatically.
- **C++**'s lambda closures require more explicit management of captured variables, either by value or by reference, providing more control but also introducing complexity.

Thus, the choice of language can have significant performance implications based on the type of system and how closures are managed.

2 Part 2: Memory Management

2.1 Section 3

Rust:



```cpp
#include <iostream>
using namespace std;

int main() {
    string* x = new string("Hello, C++!"); // Dynamically allocated memory
    string* y = x; // y now points to the same memory as x
    cout << "x: " << *x << ", y: " << *y << endl;

    delete x; // Memory must be manually deallocated
    // After this, x and y are dangling pointers if used further.
}
```

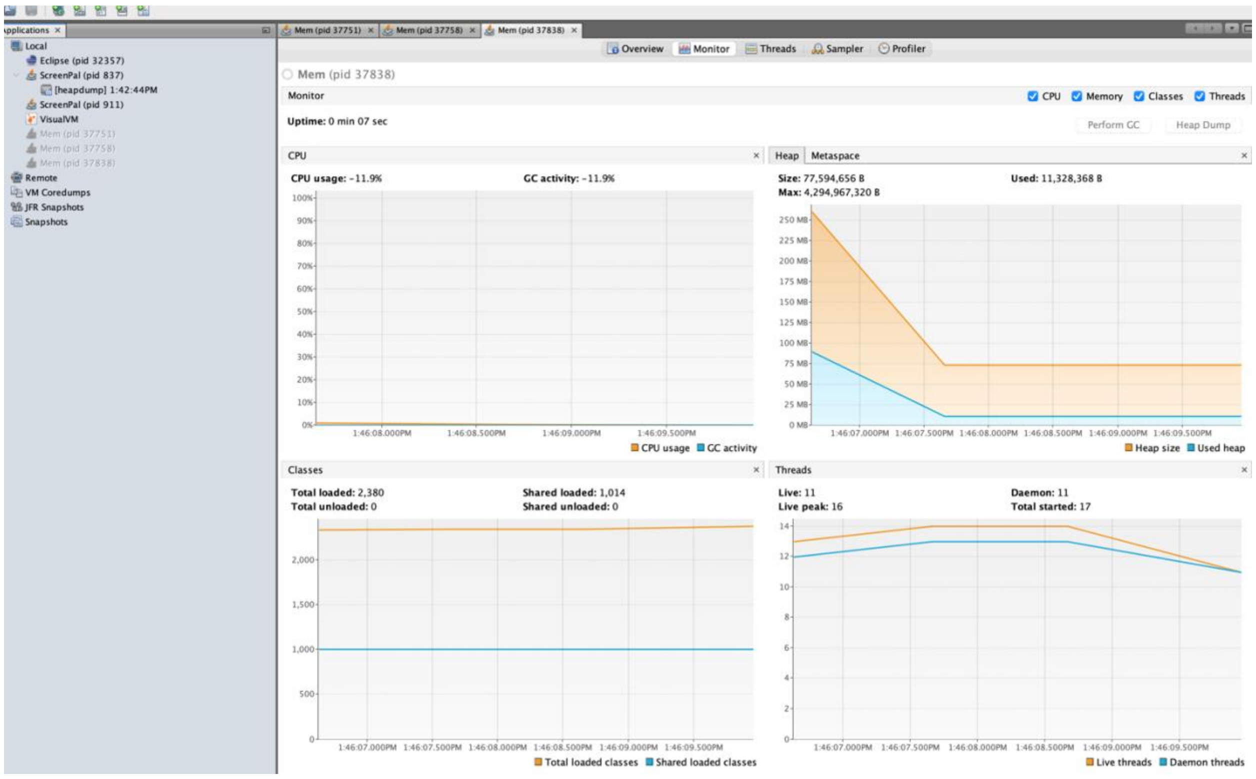
```
(base) milanbista@Milans-MacBook-Pro 2.1 % g++ -std=c++11 mem.cpp -o cppmem
(base) milanbista@Milans-MacBook-Pro 2.1 % ./cppmem
x: Hello, C++!, y: Hello, C++!
(base) milanbista@Milans-MacBook-Pro 2.1 %
```

In this Rust program, memory is dynamically allocated for the string "Hello, Rust!" through the String::from function. Rust's memory management is governed by its ownership and borrowing model. The string is owned by the variable x, and y borrows it immutably. This model ensures memory safety without the need for manual memory management. When x goes out of scope, the memory is automatically freed due to Rust's ownership rules, preventing memory leaks and dangling pointers.

The DHAT profiler tracks memory allocation and deallocation during the program's execution. Initially, 1,100 bytes are allocated in three blocks for the String object. At the point of maximum memory usage (t-gmax), this remains unchanged. By the program's end (t-end), the allocated memory reduces to 1,088 bytes in two blocks, reflecting the automatic memory deallocation when x is dropped. This data is saved in the dhat-heap.json file for further analysis.

Rust's approach to memory management through ownership and borrowing guarantees that memory is managed efficiently and safely without manual intervention, avoiding issues like memory leaks or dangling pointers. The DHAT tool provides valuable insight into memory usage patterns, showing how Rust automatically handles memory cleanup when variables go out of scope.

Java:



Mem (pid 37838)

Monitor ✓ CPU ✓ Memory ✓ Classes ✓ Threads

Uptime: 0 min 07 sec

CPU usage: ~11.9% GC activity: ~11.9%

Size: 77,594,656 B Used: 11,328,368 B
Max: 4,294,967,320 B

Total loaded: 2,380 Shared loaded: 1,014
Total unloaded: 0 Shared unloaded: 0

Live: 11 Daemon: 11
Live peak: 16 Total started: 17

In Java, memory management is handled automatically through garbage collection, which eliminates the need for manual memory allocation and deallocation. The JVM manages memory by dynamically allocating memory on the **heap** and periodically reclaiming unused memory through garbage collection. Here's how Java handles memory in the provided program:

Dynamic Memory Allocation and Garbage Collection

In the Java program, new String("Hello, Java!") dynamically allocates memory for each string object on the heap. The ArrayList holds references to these string objects. Once the list is cleared using stringList.clear(), the objects become eligible for garbage collection. Calling System.gc() requests the JVM to run garbage collection, but it’s not guaranteed to happen immediately.

How Garbage Collection Works

The **garbage collector** (GC) uses a **mark-and-sweep** algorithm:

1. **Marking**: It identifies objects still in use (i.e., referenced).
2. **Sweeping**: It removes objects that are no longer referenced, freeing up memory.

In this example, after clearing the list, the string objects are no longer referenced and are eligible for GC, which will reclaim their memory during the next collection cycle.

Memory Profiling with VisualVM

Using **VisualVM**, you can monitor memory usage as the program runs. The **Monitor** tab shows heap memory usage, which increases as new objects are allocated and decreases when garbage collection occurs. The **GC** tab shows garbage collection cycles and memory reclamation. This allows you to observe how Java manages memory dynamically and efficiently.

Handling Memory Issues

Java handles memory issues differently from languages like C++:

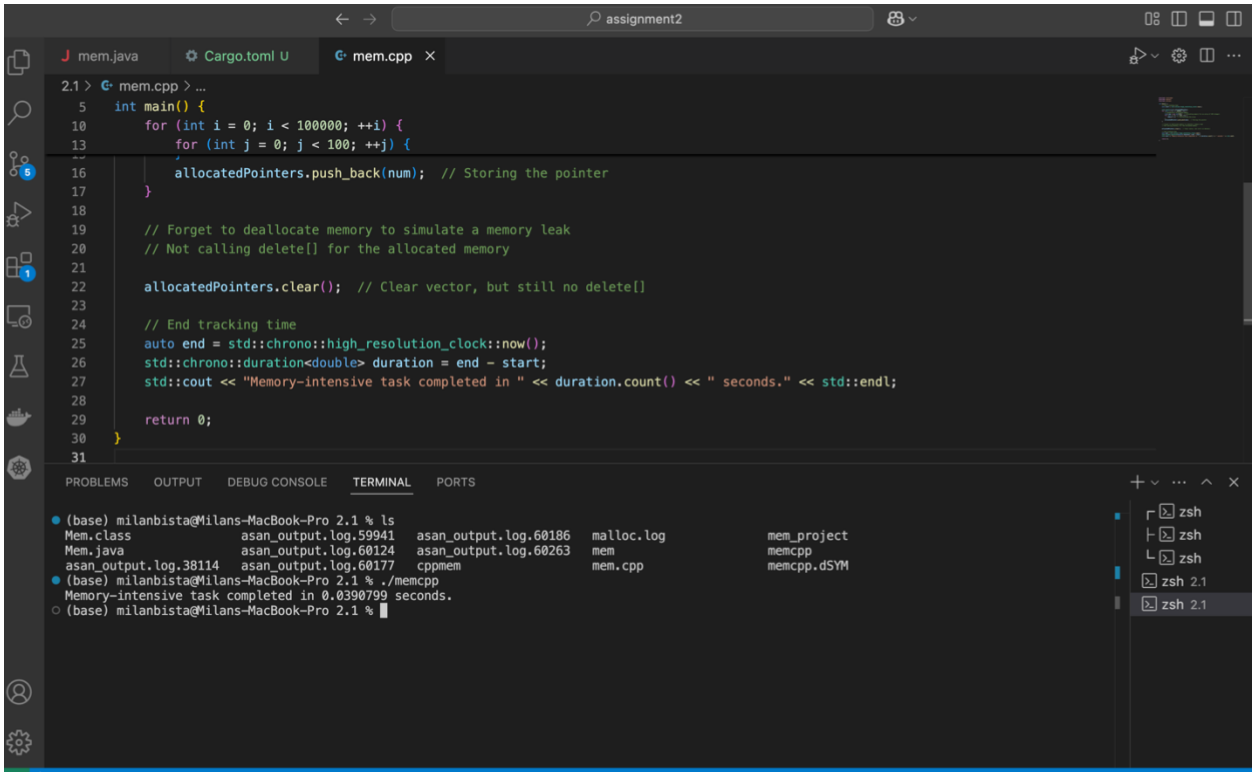
- **Memory Leaks**: These occur if objects are unintentionally kept in memory. This is prevented here by clearing references.
- **Dangling Pointers**: Java avoids these because it manages memory automatically, eliminating the need for direct memory access.

Performance Considerations

While Java handles memory management automatically, frequent garbage collection cycles can add overhead. Large programs may face performance issues due to GC pauses, especially if the heap size is not properly managed. You can adjust heap size using JVM flags like -Xms and -Xmx.

In summary, Java abstracts memory management using garbage collection, which automatically allocates and frees memory. Profiling tools like VisualVM help observe the allocation and deallocation of memory, giving insights into how Java efficiently manages resources compared to languages with manual memory management.

C++



```cpp
int main() {
    for (int i = 0; i < 100000; ++i) {
        for (int j = 0; j < 100; ++j) {
            allocatedPointers.push_back(num);  // Storing the pointer
        }

    // Forget to deallocate memory to simulate a memory leak
    // Not calling delete[] for the allocated memory

    allocatedPointers.clear();  // Clear vector, but still no delete[]

    // End tracking time
    auto end = std::chrono::high_resolution_clock::now();
    std::chrono::duration<double> duration = end - start;
    std::cout << "Memory-intensive task completed in " << duration.count() << " seconds." << std::endl;

    return 0;
}
```

```
(base) milanbista@Milans-MacBook-Pro 2.1 % ls
Mem.class               asan_output.log.59941   asan_output.log.60186   malloc.log              mem_project
Mem.java                asan_output.log.60124   asan_output.log.60263   mem                     memcpp
asan_output.log.38114   asan_output.log.60177   cppmem                  mem.cpp                 memcpp.dSYM
(base) milanbista@Milans-MacBook-Pro 2.1 % ./memcpp
Memory-intensive task completed in 0.0390799 seconds.
(base) milanbista@Milans-MacBook-Pro 2.1 %
```

C++ Memory Management

In C++, memory management is manual, meaning that the programmer is responsible for allocating and deallocating memory. This is typically done using the new and delete operators. When a program allocates memory dynamically (for example, for an object or array), the programmer must ensure that memory is properly freed when it is no longer needed. If the delete operator is not called after new, the allocated memory is never released, resulting in a **memory leak**.

C++ does not have automatic garbage collection, so it's crucial for developers to keep track of memory allocations and deallocations to avoid memory issues. One common error is **dangling pointers**, which occur when memory is freed (using delete), but a pointer still refers to that memory. Accessing such a pointer leads to undefined behavior, often causing crashes or incorrect program results. Therefore, managing memory in C++ requires careful attention and discipline to avoid such issues.

Despite these risks, C++ gives developers full control over memory, which can lead to more optimized and efficient applications. However, with this power comes the responsibility of managing resources correctly. Tools like **Valgrind** or **AddressSanitizer** are often used to detect memory leaks, dangling pointers, and other memory-related errors.

Conclusion

In this assignment, we explored key concepts in programming languages, including syntax and semantics, as well as memory management. The task involved modifying code snippets in Python, JavaScript, and C++ to introduce syntax errors and analyze how each language handles these errors. We observed that while Python and JavaScript provide descriptive error messages that are helpful for debugging, C++ requires more careful handling of compilation errors due to its statically typed nature.

Additionally, we examined the differences in type systems, scopes, and closures across Python, JavaScript, and C++. These languages handle semantic concepts in unique ways, with Python being dynamically typed and offering simpler scoping rules, JavaScript providing closures and higher-order functions, and C++ featuring stricter type enforcement and more manual memory management.

In Part 2, we delved into memory management in Rust, Java, and C++, focusing on their distinct approaches. Rust's ownership and borrowing mechanism ensures memory safety without a garbage collector, preventing issues like dangling pointers. In contrast, Java uses automatic garbage collection, which alleviates the programmer from manually managing memory but can introduce performance overhead. C++, however, relies on manual memory management, which provides more control but requires the programmer to be vigilant about memory leaks and dangling pointers.

By using memory profiling tools, we were able to observe the impact of each language's memory management approach on program performance and efficiency. Overall, this assignment highlighted the importance of understanding both syntax and semantics across languages as well as the critical role of memory management in optimizing program performance and preventing errors.