Memory Management in Python, JavaScript, C++, Rust, and Java

Report by: Carter Nelson
Course: Advanced Programming Languages (MSCS-632-B01)
Instructor: Dax Bradley
Date: 3/16/2025

---

1. Introduction

The goal of this assignment is to analyze and compare the memory management strategies in five programming languages: Python, JavaScript, C++, Rust, and Java. The analysis will explore syntax errors, type systems, memory management approaches, and performance trade-offs in these languages. Through experimentation with different types of code and the handling of memory, we aim to understand how each language's memory management system affects safety, performance, and developer experience.

---

2. Part 1: Analyzing Syntax and Semantics

2.1 Python: Syntax Errors

In the following Python code, we attempt to calculate the sum of an array:

```
def calculate_sum(arr):
    total = o
    for num in arr:
        total += num
    return total

numbers = [1, 2, 3, 4, 5]
result = calculate_sum(numbers)
print("Sum in Python:", result)
```

Error: The variable `o` is not defined. The correct initialization should be `total = 0`.

Error Message:

```
NameError: name 'o' is not defined
```

Explanation: In Python, the interpreter expects a valid value or variable. Since `o` was not declared or initialized, it raises a `NameError`.

2.2 JavaScript: Syntax Errors

In the following JavaScript code, we again calculate the sum of an array:

```
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:", result);
```

Error: Similar to the Python error, `o` is not defined and should be initialized as `0`.

Error Message:

```
Uncaught ReferenceError: o is not defined at calculateSum
```

Explanation: JavaScript raises a `ReferenceError` when an undefined variable is used. The interpreter cannot find the variable `o`, which results in a crash.

2.3 C++: Syntax Errors

The following C++ code also calculates the sum of an array:

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

int calculateSum(int arr[], int size) {
    int total = o;
    for (int i = o; 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);
    cout << "Sum in C++: " << result << endl;
    return 0;
}
```

Error: `o` is used instead of `0`, which causes an issue when trying to initialize `total`.

Error Message:
`error: 'o' was not declared in this scope`

Explanation: The C++ compiler issues a `scope error` when it encounters `o`, as this variable is undefined. It cannot proceed with the calculation until the correct initialization value (`0`) is used.

2.4 Error Comparison

Each language handles errors differently:

- **Python**: Raises a `NameError` for undefined variables. It is a runtime error, making it easy to debug.
- **JavaScript**: Raises a `ReferenceError` when an undefined variable is accessed.
- **C++**: Throws a compile-time error for undeclared variables. This requires fixing the issue before compilation, making it less flexible but safer in terms of early error detection.

---

3. Part 2: Memory Management

3.1 Rust: Ownership and Borrowing

Rust uses an ownership model for memory management, where each value has a single owner, and when the owner goes out of scope, the memory is freed. Here's an example:

```rust
fn calculate_sum(arr: Vec<i32>) -> i32 {
    let mut total = 0;
    for num in arr {
        total += num;
    }
    total
}

fn main() {
    let numbers = vec![1, 2, 3, 4, 5];
    let result = calculate_sum(numbers);
    println!("Sum in Rust: {}", result);
}
```

In Rust, ownership is transferred when `arr` is passed to `calculate_sum`. This prevents issues like dangling pointers and memory leaks.

Memory Management:

- Rust ensures memory safety through ownership and borrowing.
- No garbage collection overhead.
- Prevents memory leaks and dangling pointers through compile-time checks.

3.2 Java: Garbage Collection

Java uses automatic garbage collection, meaning memory is automatically managed and reclaimed when objects are no longer in use. Here's a simple Java program:

```java
public class MemoryManagement {
    public static void main(String[] args) {
        int[] numbers = {1, 2, 3, 4, 5};
        int result = calculateSum(numbers);
        System.out.println("Sum in Java: " + result);
    }

    public static int calculateSum(int[] arr) {
        int total = 0;
        for (int num : arr) {
```

```
            total += num;
        }
        return total;
    }
}
```

Memory Management:

- Garbage collection in Java ensures that unused objects are removed from memory automatically.
- May introduce latency due to collection pauses, but helps developers avoid manual memory management.

3.3 C++: Manual Memory Management

C++ gives developers full control over memory allocation and deallocation. Here's a simple C++ program:

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

int* create_array(int size) {
    int* arr = new int[size];
    return arr;
}

void delete_array(int* arr) {
    delete[] arr;
}

int main() {
    int* numbers = create_array(5);
    numbers[0] = 1;
    numbers[1] = 2;
    numbers[2] = 3;
    numbers[3] = 4;
    numbers[4] = 5;
```

```
    // Use numbers array
    delete_array(numbers);  // Manually free memory
    return 0;
}
```

Memory Management:

- Developers must manually allocate and free memory using `new` and `delete`.
- Risks of memory leaks or dangling pointers if `delete` is not properly used.

3.4 Memory Management Comparison

- **Rust**: Prevents errors like memory leaks with its ownership system.
- **Java**: Easier for developers with automatic garbage collection but may suffer from pauses during collection.
- **C++**: Offers the most control but also the most responsibility, requiring careful management of memory.

---

4. Conclusion

In conclusion, the three languages—Rust, Java, and C++—each have distinct approaches to memory management and syntax. Rust's ownership model guarantees memory safety, Java simplifies development with garbage collection, and C++ offers fine-grained control at the cost of complexity. The choice of language depends on the specific needs of the application, balancing performance, safety, and developer productivity.

References

Matsakis, N., & Turon, A. (2018). *The Rust programming language*. No Starch Press.

Oracle. (2024). Java SE documentation. Retrieved from https://docs.oracle.com/en/java/

Stroustrup, B. (2013). *The C++ programming language* (4th ed.). Addison-Wesley.