Unique Karanjit

Department of Computer Science, University of the Cumberlands

MSCS-632-M51: Advanced Programming Language

Dr. Vanessa Cooper

Feb 2, 2025

Part 1: Analyzing Syntax and Semantics

1.1 Section 1

For the Python code upon running, we see the error in the below screenshot.

```python
# Python : 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("Sumin Python :", result)
```

```
Traceback (most recent call last):
  File "/home/cg/root/679ec9ff12f91/main.py", line 9, in <module
      >
    result = calculate_sum (numbers)
^^^^^^^^^^^^^^^^^^^^^^^^
  File "/home/cg/root/679ec9ff12f91/main.py", line 3, in
      calculate_sum
    total = o
            ^
NameError: name 'o' is not defined
```

Error Messages:

a. NameError: name ‘o’ is not defined – The lowercase letter ‘o’ is used instead of the number ‘0’. Python treats ‘o’ as an undefined variable.

b. There is no syntax error for *print ("Sumin Python :", result),* but the output seems to be a typo because there is a missing space between Sum and in.

For the JavaScript code upon running, we see the error in the below screenshot.

```javascript
// JavaScript : 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 = calculate Sum (numbers);
console.log("Sum in JavaScript:", result);
```

```
/home/cg/root/679eca7ecb53c/script.js:11
let result = calculate Sum (numbers);
                 ^^^

SyntaxError: Unexpected identifier
    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
```

```javascript
// JavaScript : 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);
```

```
/home/cg/root/679eca7ecb53c/script.js:3
    let total = o;
            ^

ReferenceError: o is not defined
    at calculateSum (/home/cg/root/679eca7ecb53c/script.js:3:17)
    at Object.<anonymous> (/home/cg/root/679eca7ecb53c/script.js
        :11:14)
    at Module._compile (node:internal/modules/cjs/loader:1356:14
        )
    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
```

Error Messages:

a. Uncaught ReferenceError: o is not defined - JavaScript treats o as an undefined variable. It should be a number since we are doing addition operation.

b. Uncaught SyntaxError: Unexpected identifier 'Sum' - Function name calculate Sum contains a space instead of calculateSum, which is invalid syntax.

For the C++ code upon running, we see the error in the below screenshot.

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

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

int main () {
    int numbers [] = {1, 2, 3, 4, 5};
    int size = sizeof(numbers) / sizeof(numbers[o]); // 'o'
        should be '0'
    int result = calculateSum(numbers, size);
    cout << "Sum in C++" " << result << endl; // Extra "

    return o; // 'o' should be '0'
}
```

```
main.cpp: In function 'int calculateSum(int*, int)':
main.cpp:5:17: error: 'o' was not declared in this scope
    5 |     int total = o;  // 'o' should be '0'
      |                 ^
main.cpp: In function 'int main()':
main.cpp:14:49: error: 'o' was not declared in this scope
   14 |     int size = sizeof(numbers) / sizeof(numbers[o]); //
       'o' should be '0'
      |                                                 ^
main.cpp:16:57: error: expected ';' before 'return'
   16 |     cout << "Sum in C++" " << result << endl; // Extra "
      |
          ^
      |
          ;
   17 |
   18 |     return o; // 'o' should be '0'
      |     ~~~~~~
```

Error Messages:

a. error: ‘o’ was not declared in this scope → C++ treats o as an undefined variable.

b. error: missing ‘ ” ’ before ‘<<’ token → cout << "Sum in C++" " << result << endl; is missing a concatenation operator (<<).

1.2

Type System Comparison

Each programming language has a different approach to type systems.

Python is a dynamically typed language which means that the types are determined during the runtime. JavaScript is loosly typed which means that the implicit types can happen .C++ is a statically typed language where types are checked during the compile time.

Analysis on Dynamically Typed Example in Python

tutorialspoint | Online Python Compiler

Execute | Source Code | Share | Help

```python
def add(a, b):
    return a + b

print(add(5, 3))   # Outputs: 8 (integer addition)
print(add("5", "3"))  # Outputs: "53" (string concatenation)
```

```
8
53
```

- Python allows dynamic typing so that we can pass string and integer to the same function where the type of variable is not defined.
- The add function works with both integers and strings without explicit type definitions.

Analysis on Loosely typed Example in JavaScript

tutorialspoint | Online Javascript Editor

Execute | Source Code | Share | Help

```javascript
1 function add(a, b) {
2     return a + b;
3 }
4
5 console.log(add(5, 3));   // Outputs: 8
6 console.log(add("5", 3)); // Outputs: "53" (string
      concatenation due to implicit type coercion)
7
```

```
8
53
```

- JavaScript has a loose typing system which allows the implicit type conversion.
- The output statement in line 6 concatenates “5” and 3 into “53” because JavaScript converts 3 into a string.

Analysis on Statically Typed Example in C++

Execute | Source Code | Share | Help

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

int add(int a, int b) {
    return a + b;
}

int main() {
    cout << add(5, 3) << endl; // Outputs: 8
    // cout << add("5", "3") << endl; // Error: Cannot add strings in C++
    return 0;
}
```

```
8
```

- C++ requires explicit type declarations like int a and int b.
- C++ enforces strict type checking at the compile time, preventing implicit type coercion.

Scopes and Closure Comparison

Scope defines where a variable can be accessed from. Closure is the helper that allows the functions or methods to remember variables from their parent scope. Different programming languages have different scopes and closures which are analyzed below.

tutorialspoint | Online Python Compiler

Execute | Source Code | Share | Help

```python
def outer():
    x = 10
    def inner():
        print(x)  # Inner function has access to x
    return inner

closure_func = outer()
closure_func()  # Outputs: 10
```

```
10
```

Analysis of Python Scope:

- Python supports closures, where inner() remembers x even after outer() finishes execution.
- Since x is defined with parent closure outer, it is visible inside inner() function as well.

tutorialspoint | Online Javascript Editor

Execute | Source Code | Share | Help

```javascript
function outer() {
    let x = 10;
    return function inner() {
        console.log(x);  // Inner function can access outer variable
    };
}

let closureFunc = outer();
closureFunc();  // Outputs: 10
```

```
10
```

Analysis of JavaScript Closure

- JavaScript also supports closures. Block scope (let) prevents variable leakage.

Execute | Source Code | Share | Help

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

void outer() {
    int x = 10;
}

int main() {
    cout << x; // Error: x is not accessible here
    return 0;
}
```

```
main.cpp: In function 'int main()':
main.cpp:9:14: error: 'x' was not declared in this scope
    9 |     cout << x; // Error: x is not accessible here
      |              ^
```

Analysis of C++ Scope

- C++ has a block-level scope but no built-in closure mechanism. A workaround involves using lambda functions with captures.

As conclusion, three key semantic differences can be summarized as below.

1. Type System
    a. Python and JavaScript are dynamically typed whereas C++ is statistically typed.
    b. C++ is faster than Python and JavaScript because type checking is done during compile time in C++, whereas in Python and JavaScript, it is done during runtime.
2. Memory Management
    a. Python and JavaScript use automatic garbage collection whereas C++ requires manual memory management.
    b. C++ has more control over memory but also it has a high risk of memory leakages.
    c. Python and JavaScript may have overhead during runtime because of automatic garbage collection.

3. Scopes and Closures

    a. Python and JavaScript Support closures (functions can access outer variables).

    b. C++ has no direct closures so that we must use lambda expressions.

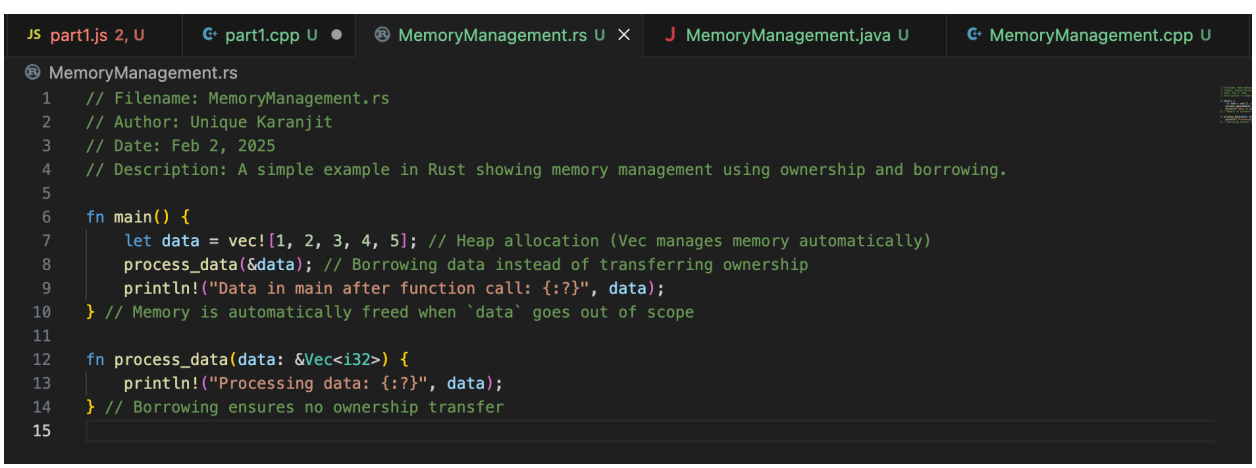
    c. Closures help in functional programming, whereas C++ relies mostly on object-oriented approach.

The pro and cons of Python, JavaScript and C++ can be summarized as below.

- Python is flexible but slower due to dynamic typing and garbage collection.
- JavaScript is like Python but has implicit type coercion which can cause unexpected results.
- C++ is fast and memory-efficient but requires manual memory management and strict typing.

2.1 Section 3

Rust implements memory safety using ownership, borrowing, and lifetime to prevent general issues like memory leaks and dangling pointers.




```rust
// Filename: MemoryManagement.rs
// Author: Unique Karanjit
// Date: Feb 2, 2025
// Description: A simple example in Rust showing memory management using ownership and borrowing.

fn main() {
    let data = vec![1, 2, 3, 4, 5]; // Heap allocation (Vec manages memory automatically)
    process_data(&data); // Borrowing data instead of transferring ownership
    println!("Data in main after function call: {:?}", data);
} // Memory is automatically freed when `data` goes out of scope

fn process_data(data: &Vec<i32>) {
    println!("Processing data: {:?}", data);
} // Borrowing ensures no ownership transfer
```

Memory Management in Rust

- Ownership ensures that when something goes out of scope, the memory is automatically freed.
- Borrowing prevents dangling pointers.
- No explicit free() calls.
- Rust ensures memory safety at compile-time, so there is no garbage collector.

Java relies on **automatic garbage collection (GC)** to manage memory, meaning unused objects are deallocated automatically.

```
JS part1.js 2, U   part1.cpp U   MemoryManagement.rs U   MemoryManagement.java U   MemoryManagement.cpp U
MemoryManagement.java > MemoryManagement
1   // Filename: MemoryManagement.java
2   // Author: Unique Karanjit
3   // Date: Feb 2, 2025
4   // Description: A simple example in Java demonstrating garbage collection.
5
6   public class MemoryManagement {
        Run | Debug
7       public static void main(String[] args) {
8           int[] data = new int[]{1, 2, 3, 4, 5}; // Heap allocation
9           processData(data);
10          System.out.println("Data in main after function call: " + java.util.Arrays.toString(data));
11          // No need to manually free memory—GC will clean up
12      }
13
14      static void processData(int[] data) {
15          System.out.println("Processing data: " + java.util.Arrays.toString(data));
16      }
17  }
18
```

Memory Management in Java

- The **JVM garbage collector (GC)** automatically frees unused memory.
- No manual memory deallocation is needed, since this is already managed.
- **GC overhead** can lead to performance unpredictability.

C++: Manual Memory Management

C++ requires **explicit allocation (new) and deallocation (delete)**, which increases flexibility but also the risk of memory leaks and dangling pointers.

MemoryManagement.cpp

```cpp
// Filename: MemoryManagement.cpp
// Author: Unique Karanjit
// Date: Feb 2, 2025
// Description: A simple example in C++ demonstrating manual memory management with `new` and `delete`.

#include <iostream>

void processData(int* data, int size) {
    std::cout << "Processing data: ";
    for (int i = 0; i < size; i++) {
        std::cout << data[i] << " ";
    }
    std::cout << std::endl;
}

int main() {
    int* data = new int[5]{1, 2, 3, 4, 5}; // Explicit heap allocation
    processData(data, 5);
    delete[] data; // Manual deallocation to avoid memory leaks
    return 0;
}
```

Memory Management in C++

- **Manual memory allocation and deallocation** via new and delete and must be developed by the developer.
- There is a high risk of **memory leaks** in case if delete is forgotten.
- Possible **dangling pointers** if delete is called but the pointer is still accessed.