**Assignment 2: Syntax, Semantics, and Memory Management**

Shrisan Kapali

Advanced Programming Languages (MSCS 632 M20)

Ulrich Vouama  
June 1, 2025

**Part 1: Analyzing Syntax and Semantics**

**Python Code Snippet**

When running the provided code snippet for the Python code, the compiler gave the following error.A screenshot of a computer

AI-generated content may be incorrect.

In Python, code is interpreted line by line during execution (Rizwan, 2023). While executing the calculate\_sum function, as the lowercase “o” is used instead of the digit zero while assigning the value to the total variable, the interpreter encounters an undefined variable “o” and throws a NameError. As Python interprets code line by line, it stops at the first error found during the code run.

**JavaScript** **Code Snippet**

The compiler gave an “Unexpected identifier” error for the provided JavaScript code snippet. A screenshot of a computer screen

AI-generated content may be incorrect.

While executing the line “let result = calculate Sum (numbers),” the JavaScript interpreter identifies “calculate” as one identifier and then “Sum” as another identifier. This is not a valid expression, and as “Sum” is neither defined nor a keyword, it throws an unexpected identifier error. The correct expression should have been “calculateSum(numbers)” which has no space between “calculate” and “Sum”. Additionally, while defining the total, letter “o” is not defined, so if we fix the “calculateSum(numbers)”, next, the JavaScript compiler throws a ReferenceError, as “o” has not been defined before. A screenshot of a computer screen

AI-generated content may be incorrect.

**C++** **Code Snippet**

In C++, all syntax issues are reported during compile time. As the "cout << "Sum in C++" << result << endl;" has an extra quotation, it causes a syntax error. Additionally, the digit "0" must have been used instead of the letter "o", or the letter "o" must have been defined as "int o=0" for the code to execute properly. In C++, until all the issues are resolved during the compile time, the Code fails to run.

**A screenshot of a computer screen

AI-generated content may be incorrect.**

**Section 2**

A program that calculates the factorial of a given number was written in Python, JavaScript, and C++.

**Python Factorial Code**

**A screenshot of a computer program

AI-generated content may be incorrect.**

**Output**

**A computer code with black text

AI-generated content may be incorrect.**

In Python, explicit type declaration is not required, and we do not declare the variable type. The type of variable is assigned dynamically and provides high flexibility for programmers. In the factorial program, the “result” variable type is a number if the “n” value is greater than or equal to 0, and a string type when the value of n is less than 0.

Python uses lexical scoping, and its scope can be classified into local, global, and nonlocal variables and follows the LEGB rule (“Python Variable Scope,” n.d.). A local scope is where the variable can only be accessed within the function or block that defines it. The variable can be accessed from any part of the program in a global scope, and the nonlocal variables are used in nested functions (“Python Variable Scope,” n.d.).

Additionally, closures are fully supported in Python. When a closure is created in Python, reference to the nested function in its enclosing scope is automatically stored; that way, the inner function can access those variables.

**JavaScript Factorial Code**

While declaring or defining a variable, JavaScript does not require programmers to define the variable type explicitly. Variables can be defined using the keywords let, const, or var, and their type is determined at the runtime. In JavaScript, implicit type coercion is allowed, and strict operators (===) are required to avoid unintended type coercion.

JavaScript also uses lexical scoping, which determines the scope of a variable by its declaration position within the code. Its scoping can be distinguished into the global scope, the variable declared outside any function or block (“JavaScript Scope,” n.d.). Function Scope in JS is where variables declared inside a function are accessible anywhere inside the function, and block scope in JS is where variables declared are only accessible inside a {} block (“JavaScript Scope,” n.d.).

JavaScript strongly supports closures and is very common in async/event code. In JavaScript, a function forms a closure over its lexical environment and allows access to the variables from outer functions. A screenshot of a computer program

AI-generated content may be incorrect.

Output

A white background with blue and yellow text

AI-generated content may be incorrect.

**C++ Factorial Code**

            C++ requires explicit type declarations for all the variables, function parameters, and return values. The type checking in C++ occurs at the compile time, and the program fails to compile until all the type errors are resolved. In C++, implicit type conversions are allowed but are generally restricted to safe conversions.

            In C++, the scope of an identifier is determined by its position in the source code. It has a global scope where variables declared outside the functions or class can be used anywhere after the declaration. Local scope in C++ limits the variable use within the defined function and naming scope where the same variable name is present inside and outside a function but is treated as separate variables (“C++ Variable Scope,” n.d.).

            Closures are not common in C++. However, lambda expressions in C++ allow for anonymous functions and specify variables from their surrounding scope.

A screenshot of a computer program

AI-generated content may be incorrect.

**Key Semantic Differences between Python, JavaScript, and C++**

**Block Codes**

Indentation is used to define a block in Python, while in JavaScript and C++, curly braces are used for code blocks and semicolons to end the statements. The indentation for code blocks in Python makes it simpler and comparatively more straightforward to read the code, while if the code is not formatted correctly in JavaScript and C++ using a formatter such as Prettier or using appropriate indentation, it might make it challenging for developers to read the code.

**Type Systems**

JavaScript and Python dynamically set the type of variable during the runtime. In contrast, in C++, the variable type needs to be explicitly set and checked during the compile time. The static type in C++ helps prevent unwanted program behavior during the runtime because of reduced flexibility and longer development time for specific tasks. In contrast, dynamic typing in JavaScript and Python allows for faster prototyping and more concise code but may lead to runtime errors, such as type mismatches or undefined behavior due to implicit conversions.

**Memory Management**

            In Python and JavaScript, memory is automatically managed through garbage collection, while in C++, developers need to allocate and deallocate memory explicitly. The standard memory errors, such as memory leaks and dangling pointers, are prevented in Python and JS due to the automatic garbage collection. However, it comes with additional performance overhead. In C++, as it allows for manual memory management, users can highly optimize memory usage, and it has no runtime overhead. However, if the memory is not managed correctly, issues such as dangling pointers and leaks can cause the program to crash.

**Part 2: Memory Management**

To understand memory management across Rust, Java, and C++, programs have been written to test memory usage and performance across the programming languages.

**Rust Program demonstrating ownership and borrowing**

**A screenshot of a computer program

AI-generated content may be incorrect.**

            This Rust program demonstrates ownership and borrowing concepts by creating a string and assigning it to the variables. We first create the ownership and then process the string ownership of the variable my\_string. If we try to access the my\_string after its ownership has been transferred, the compiler throws a “borrow of moved value: `my\_string`” error.A screenshot of a computer program

AI-generated content may be incorrect.

            If the variable only borrows the variable using the reference, the variable can still be reused. Finally, the values are automatically dropped when the program ends, and the memory is freed.

            The memory in Rust is automatically allocated and deallocated. In the program, String, a standard library that manages its heap memory, automatically handles deallocation when the owner goes out of scope. As the ownership model ensures that memory is automatically freed, memory leaks are hard to create. Additionally, the borrowing and lifetime rules help prevent the dangling pointers as these rules prevent references from outliving the data they point to.

**Rust Program Output**

**A screenshot of a computer program

AI-generated content may be incorrect.**

**Using Valgrind to analyze program memory**

**A screenshot of a computer program

AI-generated content may be incorrect.**

For this simple Rust program, using Valgrind, 10 heap memory were allocated and freed and no leaks were identified.

The rust program was compiled using the code

**“rustc -g RustMemoryManagement.rs -o rust\_memory\_debug”**

And after installing Valgrind,

**“valgrind --leak-check=full --track-origins=yes ./rust\_memory\_debug”**

command was used to print the debug output on the terminal.

**Java Program demonstrating garbage collection**

            We create an array of integers to understand automatic garbage collection in Java. We define a class called DataObjects that initializes an array of integers and holds a name, provides constructors, and overrides finalize method to include the print statement.

            We define a scope such that when the DataObject is initialized within the scope and when the scope is finished, the garbage collection automatically runs. We also indicate that the JVM will run the garbage collection using the command System.gc().

A screenshot of a computer program

AI-generated content may be incorrect.

A screenshot of a computer program

AI-generated content may be incorrect.

**Output**

A screenshot of a computer program

AI-generated content may be incorrect.

            In the Java program, we allocate heap memory using the “new” keyword. As in Java, objects no longer reachable become eligible for garbage collection once the scope of the DataObjects defined within the {} braces were executed. The system identified that the memory allocated is eligible for garbage collection. Thus, the System.gc() command indicated to the JVM that this memory is ready for deallocation. Thus, as in the terminal, we can see that the JVM deallocated the allocated memory. Additionally, in the program the “longLivedObject” shows that the objects remain in the memory as long as they are referenced.

The GC automatically handles memory allocation and deallocation in Java. However, logical memory leaks can still occur if the variables holds onto references to objects that are not needed.

**Observing Memory Management using JVM flags**

The Java program was compiled using the command,

**“javac JavaMemoryManagement.java”**

The following command was used to observe the garbage collection logging.

**“java -Xms128m -Xmx512m -Xlog:gc\* JavaMemoryManagement”**

**A white background with black text

AI-generated content may be incorrect.**

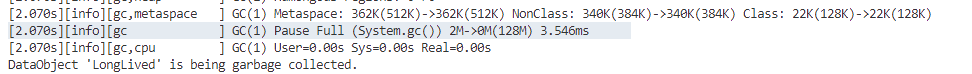
**A screenshot of a computer program

AI-generated content may be incorrect.**

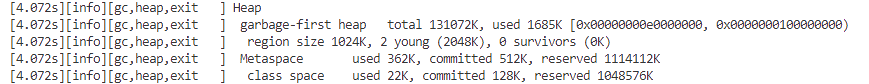
From the logs, we can observe that for the first garbage collection triggered by Sytem.gc() command, the allocated heap memory size of 2M before GC was changed to 0M after GC which happened in .302ms.



For the second garbage collection triggered by Sytem.gc() command, the allocated heap memory size of 2M before GC was changed to 0M after GC which happened in .3546ms.



The overall heap size was 131072K and the program used 1685K.



**C++ Program demonstrating manual memory management**

In C++, we have to allocate and deallocate memory manually. In the program, we illustrate memory allocation using “new” and deallocation using “delete” and showcase problems such as potential memory leaks and dangling pointers. In addition, we also demonstrate modern C++ using smart pointers by using functions such as “unique\_ptr,” “weak\_ptr,” and “shared\_ptr.” For allocating and deallocating arrays in C++, we use “new[]” and “delete[]” keywords.

A screenshot of a computer program

AI-generated content may be incorrect.

A screen shot of a computer program

AI-generated content may be incorrect.

A screenshot of a computer code

AI-generated content may be incorrect.

A computer screen shot of a program

AI-generated content may be incorrect.

**Output**

**A screenshot of a computer program

AI-generated content may be incorrect.**

            In the C++ program, we demonstrate the memory allocation with “new” or “new[]” and deallocation with “delete” or “delete[].” If the delete is not called after the object is no longer in use, the allocated memory is not returned to the system, causing a memory leak. To prevent dangling pointers, once the object memory has been deallocated, we need to set the object as a null pointer; otherwise, accessing the object leads to undefined behavior, causing the program to crash.

            The smart pointer in C++ automatically handles deallocation when the object's scope goes out of scope. This mitigates the risks of manual memory management.

**Using Valgrind Massif to visualize memory usage**

Using the following command, the C++ file was compiled using gcc.

**“g++ -g C++MemoryManagement.cpp -o cpp\_memory\_debug”**

We then execute the output file using Massif.

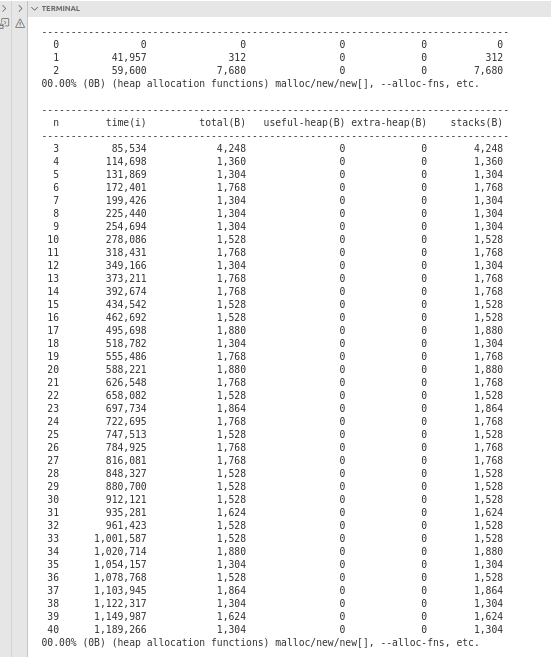
**“valgrind --tool=massif --stacks=yes --massif-out-file=massif.out ./cpp\_memory\_debug”**

**A screenshot of a computer program

AI-generated content may be incorrect.**

The above memory profiling data was generated.

Using the **“ms\_print massif.out”** command, we got a detailed output of the program memory usage.



A screenshot of a computer

AI-generated content may be incorrect. A screenshot of a computer screen

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect. A screenshot of a computer

AI-generated content may be incorrect.

A screenshot of a computer code

AI-generated content may be incorrect.

Using massif-visualizer to get a better understanding of memory usage.

**A screenshot of a computer

AI-generated content may be incorrect.**

From the report we saw that the peak memory usage occurred at snapshot 86 which was 74.6 KiB. This indicates that maximum heap memory of 74.6 KiB was used by the program during the total execution.

**GitHub Link**

https://github.com/ShrisanKapali-Cumberlands/MSCS-632-Assignment2

**References**

*C++ Variable Scope*. W3Schools Online Web Tutorials. (n.d.-a). https://www.w3schools.com/cpp/cpp\_scope.asp

*JavaScript Scope*. W3Schools Online Web Tutorials. (n.d.-b). https://www.w3schools.com/js/js\_scope.asp

Rizwan, A. (2023, December 22). *Python: Interpreted or compiled? A dive into Python’s execution*. Medium. https://medium.com/@abdulrehmanrizwan81/python-interpreted-or-compiled-a-dive-into-pythons-execution-5315ab1450d6

*Understanding python variables scope*. Tutorialspoint. (n.d.). https://www.tutorialspoint.com/python/python\_variables\_scope.htm