Assignment 2: Syntax, Semantics, and Memory Management

Shrisan Kapali

Advanced Programming Languages (MSCS 632 M20)

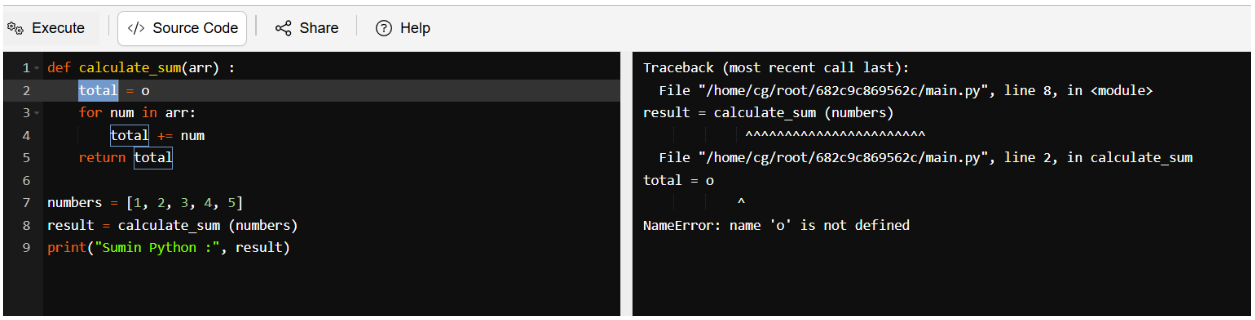
Ulrich Vouama

June 1, 2025

Part 1: Analyzing Syntax and Semantics

Python Code Snippet

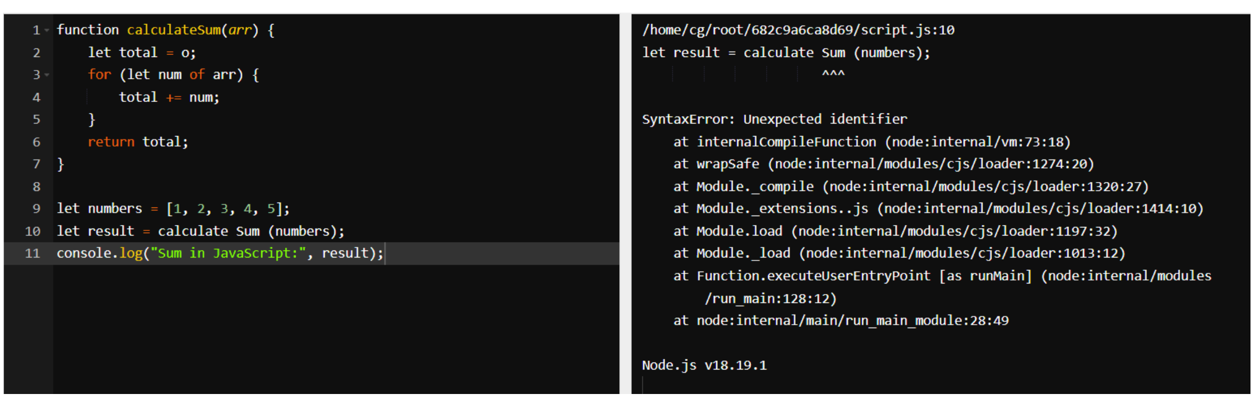
When running the provided code snippet for the Python code, the compiler gave the following error.



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.



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.

```
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/682c9a6ca8d69/script.js:2
    let total = o;
                ^

ReferenceError: o is not defined
    at calculateSum (/home/cg/root/682c9a6ca8d69/script.js:2:17)
    at Object.<anonymous> (/home/cg/root/682c9a6ca8d69/script.js:10:14)
    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
```

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.

```cpp
#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 [o]);
    int result = calculateSum(numbers, size);
    cout << "Sum in C++" " << result << endl;
    return o;
}··
```

```
main.cpp:16:26: warning: missing terminating " character
   16 |     cout << "Sum in C++" " << result << endl;
      |                          ^
main.cpp:16:26: error: missing terminating " character
   16 |     cout << "Sum in C++" " << result << endl;
      |                          ^~~~~~~~~~~~~~~~~~~~
main.cpp: In function 'int calculateSum(int*, int)':
main.cpp:5:17: error: 'o' was not declared in this scope
    5 |     int total = o;
      |                 ^
main.cpp:8:6: error: '\U0000feff' was not declared in this scope
    8 |     }·
      |
main.cpp:10:1: warning: no return statement in function returning non-void
    [·]8;;https://gcc.gnu.org/onlinedocs/gcc/Warning-Options.html#index
    -Wreturn-type·-Wreturn-type·]8;;·]
   10 | }
      | ^
main.cpp: In function 'int main()':
main.cpp:14:51: error: 'o' was not declared in this scope
   14 |     int size = sizeof(numbers) / sizeof( numbers [o]);
      |                                                   ^
main.cpp:16:25: error: expected ';' before 'return'
   16 |     cout << "Sum in C++" " << result << endl;
      |                         ^
      |                         ;
   17 |     return o;
      |     ~~~~~~
```

Section 2

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

Python Factorial Code

README.md | Section2_Python_Factorial.py | Section2_JavaScript_Factorial.js | Section2_C++_Factorial.cpp

Section2_Python_Factorial.py > ...

```python
# A simple Python program to calculate the factorial of a number using recursion.
def factorial(n):
    # If the number is negative, return an error message
    if n < 0:
        return "Factorial is not defined for negative numbers"
    # Base case: factorial of 0 is 1
    elif n == 0:
        return 1
    # Recursive case: n! = n * (n-1)!
    else:
        return n * factorial(n - 1)


# Demonstration of the factorial function
number = 6
result = factorial(number)
print(f"The factorial of {number} is {result}")

# Providing additional test cases for negative
number_negative = -6
result_negative = factorial(number_negative)
print(f"The factorial of {number_negative} is {result_negative}")

# Providing a test case for zero
number_zero = 0
result_zero = factorial(number_zero)
print(f"The factorial of {number_zero} is {result_zero}")
```

Output

```
25/MSCS-632/Assignments/Assignment2/Section2_Python_Factorial.py"
The factorial of 6 is 720
The factorial of -6 is Factorial is not defined for negative numbers
The factorial of 0 is 1
```

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.

JS Section2_JavaScript_Factorial.js > ...

```
// A function to calculate the factorial of a number using recursion
function factorial(n) {
  // Check if the input is a negative number, zero, or a positive integer
  // If negative, return a message indicating factorial is not defined
  if (n < 0) {
    return "Factorial is not defined for negative numbers";
  }
  // If zero, return 1 (0! = 1)
  else if (n === 0) {
    return 1;
  }
  // Else, calculate factorial recursively
  else {
    return n * factorial(n - 1);
  }
}

// Test the factorial function with different inputs
let number = 6;
let result = factorial(number);
console.log(`The factorial of ${number} is ${result}`);

// Test for negative scenario
let numberNegative = -6;
let resultNegative = factorial(numberNegative);
console.log(`The factorial of ${numberNegative} is ${resultNegative}`);

// Test for zero scenario
let numberZero = 0;
let resultZero = factorial(numberZero);
console.log(`The factorial of ${numberZero} is ${resultZero}`);
```

Output

```
C:\Program Files\nodejs\node.exe .\Section2_JavaScript_Factorial.js
The factorial of 6 is 720
The factorial of -6 is Factorial is not defined for negative numbers
The factorial of 0 is 1
```

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.

Section2_C++_Factorial.cpp > main()

```cpp
// C++ Code to determine the factorial of a number
#include <iostream>

// Function declaration
long long factorial(int n);

int main()
{
    // Example usage of the factorial function
    int number = 6;
    long long result = factorial(number);
    std::cout << "The factorial of " << number << " is " << result << std::endl;

    // Testing with a negative number
    int numberNegative = -6;
    long long resultNegative = factorial(numberNegative);
    std::cout << "The factorial of " << numberNegative << " is " << resultNegative << std::endl;

    // Testing with zero
    int numberZero = 0;
    long long resultZero = factorial(numberZero);
    std::cout << "The factorial of " << numberZero << " is " << resultZero << std::endl;

    return 0;
}

// Defining the factorial function
long long factorial(int n)
{
    // Handle negative input gracefully
    if (n < 0)
    {
        return 0;
    }
    // If the number is 0, return 1 (0! = 1)
    else if (n == 0)
    {
        return 1;
    }
    // For all other positive integers, calculate factorial recursively
    else
    {
        return (long long)n * factorial(n - 1);
    }
}
```

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

README.md | Section2_Python_Factorial.py | Section2_JavaScript_Factorial.js | Section2_C++_Factorial.cpp | RustMemoryManagement.rs U

RustMemoryManagement.rs

```rust
// This function takes ownership of a String.
// When 's' goes out of scope, its memory will be automatically freed.
fn process_string_ownership(s: String) {
    println!("Processing string (owned): {}", s);
    // 's' is dropped here, and its memory is reclaimed.
}

// This function borrows a String. It does not take ownership.
// The caller retains ownership, and the borrowed reference must be valid
fn process_string_borrowed(s: &str) {
    println!("Processing string (borrowed): {}", s);
}

// This function creates a String and returns ownership.
fn create_string() -> String {
    let new_string = String::from("Hello World!");
    println!("Created string: {}", new_string);
    new_string // Ownership is moved out of this function
}

fn main() {
    println!("--- Assignment Part 2 : Rust Memory Management ---");

    // 1. Transfer of Ownership to my_string
    let my_string = create_string(); // my_string now owns the String
    println!("Original string owner my_string: {}", my_string);

    // After this call, my_string will no longer own the String.
    // It's moved into process_string_ownership.
    process_string_ownership(my_string);
    // println!("Attempting to use my_string after move: {}", my_string);
    // ^ This commented line if uncommented would cause a compile-time error: "borrow of moved value: `my_string`"

    println!("\n--- Demonstrating Borrowing ---");
    let another_string = String::from("Borrowing the string!");
    println!("Another string in main: {}", another_string);

    // We pass a reference (&<variable_name>) to the function.
    // main function still owns 'another_string'.
    process_string_borrowed(&another_string);
    println!(
        "After Borrow Another string : {}",
        another_string
    );

    // Rust automatically drops 'another_string' when it goes out of scope here.
    println!("--- End of Rust program ---");
}
```

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.

```
   Compiling playground v0.0.1 (/playground)
error[E0382]: borrow of moved value: `my_string`
  --> src/main.rs:31:61
   |
25 |     let my_string = create_string(); // my_string now owns the String
   |         --------- move occurs because `my_string` has type `String`, which does
...
30 |     process_string_ownership(my_string);
   |                              --------- value moved here
31 |      println!("Attempting to use my_string after move: {}", my_string);
   |                                                              ^^^^^^^^^ value bor
   |
note: consider changing this parameter type in function `process_string_ownership` t
  --> src/main.rs:3:32
   |
3  | fn process_string_ownership(s: String) {
   |    ------------------------    ^^^^^^ this parameter takes ownership of the val
   |    |
   |    in this function
   = note: this error originates in the macro `$crate::format_args_nl` which comes f
help: consider cloning the value if the performance cost is acceptable
   |
30 |     process_string_ownership(my_string.clone());
   |                                       ++++++++

For more information about this error, try `rustc --explain E0382`.
error: could not compile `playground` (bin "playground") due to 1 previous error
```

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.

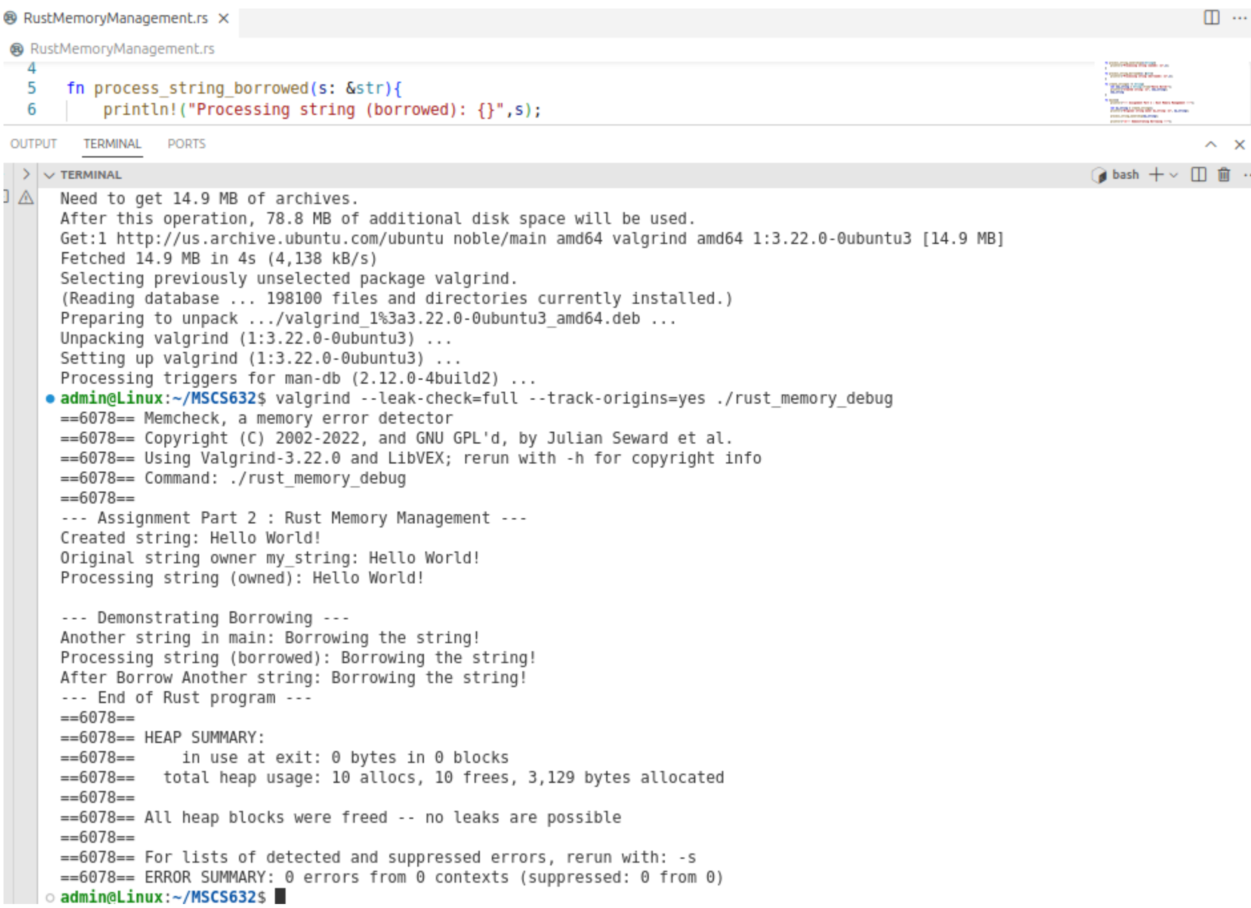
The memory in Rust is automatically allocated and deallocated. In the program, String, a standard library that manages its heap memory, automatically handles deallocation when the owner goes out of scope. As the ownership model ensures that memory is automatically freed, memory leaks are hard to create. 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

Standard Output

```
--- Assignment Part 2 : Rust Memory Management ---
Created string: Hello World!
Original string owner my_string: Hello World!
Processing string (owned): Hello World!

--- Demonstrating Borrowing ---
Another string in main: Borrowing the string!
Processing string (borrowed): Borrowing the string!
After Borrow Another string : Borrowing the string!
--- End of Rust program ---
```

Using Valgrind to analyze program memory



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().

README.md | JavaMemoryManagement.java 2, U

JavaMemoryManagement.java > JavaMemoryManagement > main(String[])

```java
// Creating a DataObject class to demonstrate memory management in Java
// This example illustrates how Java's garbage collector works with heap memory
class DataObject {
    private int[] largeArray;
    private String name;

    // Constructor to initialize the DataObject with a name and a large array
    public DataObject(String name, int size) {
        this.name = name;
        this.largeArray = new int[size]; // Allocating a large array on the heap
        System.out.println("DataObject '" + name + "' created with array of size " + size);
    }

    // This method will be called by the garbage collector before an object is
    // removed
    @Override
    protected void finalize() throws Throwable {
        System.out.println("DataObject '" + name + "' is being garbage collected.");
    }
}
```

README.md   JavaMemoryManagement.java 2, U ×

JavaMemoryManagement.java > JavaMemoryManagement

```
public class JavaMemoryManagement {

    Run | Debug
    public static void main(String[] args) {
        System.out.println(x:"--- Java Memory Management ---");

        // Create a scope to let objects become eligible for GC
        {
            System.out.println(x:"Creating 10 DataObjects...");
            for (int i = 0; i < 10; i++) {
                // Each DataObject and its largeArray are allocated on the heap
                new DataObject("Object_" + i, size:10); // Creating objects without holding references
            }
            System.out.println(x:"Finished creating DataObjects in inner scope.");
            // At this point, many DataObject instances become unreachable.
            // The JVM's garbage collector will eventually reclaim their memory.
        } // End of inner scope

        System.out.println(x:"\nSuggesting garbage collection...");
        // Hint to the JVM to run GC, but not guaranteed
        System.gc(); // This is just a suggestion, not a command.

        System.out.println(x:"\nCreating a long-lived object:");
        DataObject longLivedObject = new DataObject(name:"LongLived", size:15); // This object will persist

        // Let the program run for a bit to allow GC to occur
        try {
            Thread.sleep(millis:2000);
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        }

        System.out.println(x:"Long-lived object still active.");

        // Nullifying the reference to the long-lived object makes it eligible for GC
        System.out.println(x:"Nullifying long-lived object reference.");
        longLivedObject = null;
        System.gc(); // Another suggestion for GC

        try {
            Thread.sleep(millis:2000); // Give GC time to run
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        }

        System.out.println(x:"--- End of Java program ---");
    }
}
```

Output

```
PS S:\University of Cumberlands\Summer 2025\MSCS-632\Assignments\Assignment2>  s:; cd 's:'
a.exe' '-XX:+ShowCodeDetailsInExceptionMessages' '-cp' 'C:\Users\shris\AppData\Roaming\Co
yManagement'
--- Java Memory Management ---
Creating 10 DataObjects...
DataObject 'Object_0' created with array of size 10
DataObject 'Object_1' created with array of size 10
DataObject 'Object_2' created with array of size 10
DataObject 'Object_3' created with array of size 10
DataObject 'Object_4' created with array of size 10
DataObject 'Object_5' created with array of size 10
DataObject 'Object_6' created with array of size 10
DataObject 'Object_7' created with array of size 10
DataObject 'Object_8' created with array of size 10
DataObject 'Object_9' created with array of size 10
Finished creating DataObjects in inner scope.

Suggesting garbage collection...

Creating a long-lived object:
DataObject 'LongLived' created with array of size 15
DataObject 'Object_9' is being garbage collected.
DataObject 'Object_8' is being garbage collected.
DataObject 'Object_7' is being garbage collected.
DataObject 'Object_6' is being garbage collected.
DataObject 'Object_5' is being garbage collected.
DataObject 'Object_4' is being garbage collected.
DataObject 'Object_3' is being garbage collected.
DataObject 'Object_2' is being garbage collected.
DataObject 'Object_1' is being garbage collected.
DataObject 'Object_0' is being garbage collected.
Long-lived object still active.
Nullifying long-lived object reference.
DataObject 'LongLived' is being garbage collected.
--- End of Java program ---
PS S:\University of Cumberlands\Summer 2025\MSCS-632\Assignments\Assignment2>
```

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”

```
Suggesting garbage collection...
[0.061s][info][gc,start    ] GC(0) Pause Full (System.gc())
[0.061s][info][gc,task     ] GC(0) Using 3 workers of 13 for full compaction
[0.062s][info][gc,phases,start] GC(0) Phase 1: Mark live objects
[0.062s][info][gc,phases      ] GC(0) Phase 1: Mark live objects 0.734ms
[0.062s][info][gc,phases,start] GC(0) Phase 2: Prepare compaction
[0.063s][info][gc,phases      ] GC(0) Phase 2: Prepare compaction 0.246ms
[0.063s][info][gc,phases,start] GC(0) Phase 3: Adjust pointers
[0.063s][info][gc,phases      ] GC(0) Phase 3: Adjust pointers 0.439ms
[0.063s][info][gc,phases,start] GC(0) Phase 4: Compact heap
[0.064s][info][gc,phases      ] GC(0) Phase 4: Compact heap 0.307ms
[0.064s][info][gc,phases,start] GC(0) Phase 5: Reset Metadata
[0.064s][info][gc,phases      ] GC(0) Phase 5: Reset Metadata 0.179ms
[0.064s][info][gc,heap        ] GC(0) Eden regions: 3->0(23)
[0.065s][info][gc,heap        ] GC(0) Survivor regions: 0->0(0)
[0.065s][info][gc,heap        ] GC(0) Old regions: 0->3
[0.065s][info][gc,heap        ] GC(0) Humongous regions: 0->0
[0.065s][info][gc,metaspace   ] GC(0) Metaspace: 320K(512K)->320K(512K) NonClass: 303K(384K)->303K(384K) Class: 16K(128K)->16K(128K)
[0.065s][info][gc             ] GC(0) Pause Full (System.gc()) 2M->0M(128M) 4.302ms
[0.065s][info][gc,cpu         ] GC(0) User=0.00s Sys=0.00s Real=0.01s


Long-lived object still active.
Nullifying long-lived object reference.
[2.066s][info][gc,start       ] GC(1) Pause Full (System.gc())
[2.066s][info][gc,task        ] GC(1) Using 3 workers of 13 for full compaction
[2.067s][info][gc,phases,start] GC(1) Phase 1: Mark live objects
[2.067s][info][gc,phases      ] GC(1) Phase 1: Mark live objects 0.734ms
[2.067s][info][gc,phases,start] GC(1) Phase 2: Prepare compaction
[2.068s][info][gc,phases      ] GC(1) Phase 2: Prepare compaction 0.256ms
[2.068s][info][gc,phases,start] GC(1) Phase 3: Adjust pointers
[2.068s][info][gc,phases      ] GC(1) Phase 3: Adjust pointers 0.320ms
[2.068s][info][gc,phases,start] GC(1) Phase 4: Compact heap
[2.068s][info][gc,phases      ] GC(1) Phase 4: Compact heap 0.215ms
[2.068s][info][gc,phases,start] GC(1) Phase 5: Reset Metadata
[2.069s][info][gc,phases      ] GC(1) Phase 5: Reset Metadata 0.164ms
[2.069s][info][gc,heap        ] GC(1) Eden regions: 2->0(23)
[2.069s][info][gc,heap        ] GC(1) Survivor regions: 0->0(0)
[2.069s][info][gc,heap        ] GC(1) Old regions: 3->3
[2.070s][info][gc,heap        ] GC(1) Humongous regions: 0->0
[2.070s][info][gc,metaspace   ] GC(1) Metaspace: 362K(512K)->362K(512K) NonClass: 340K(384K)->340K(384K) Class: 22K(128K)->22K(128K)
[2.070s][info][gc             ] GC(1) Pause Full (System.gc()) 2M->0M(128M) 3.546ms
[2.070s][info][gc,cpu         ] GC(1) User=0.00s Sys=0.00s Real=0.00s
DataObject 'LongLived' is being garbage collected.
--- End of Java program ---
[4.072s][info][gc,heap,exit   ] Heap
[4.072s][info][gc,heap,exit   ]  garbage-first heap   total 131072K, used 1685K [0x00000000e0000000, 0x0000000100000000)
[4.072s][info][gc,heap,exit   ]   region size 1024K, 2 young (2048K), 0 survivors (0K)
[4.072s][info][gc,heap,exit   ]  Metaspace       used 362K, committed 512K, reserved 1114112K
[4.072s][info][gc,heap,exit   ]   class space    used 22K, committed 128K, reserved 1048576K
PS S:\University of Cumberlands\Summer 2025\MSCS-632\Assignments\Assignment2>
```

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.

```
[0.065s][info][gc,metaspace   ] GC(0) Metaspace: 320K(512K)->320K(512K) NonClass: 303K(384K)->303K(384K) Class: 16K(128K)->16K(128K)
[0.065s][info][gc             ] GC(0) Pause Full (System.gc()) 2M->0M(128M) 4.302ms
[0.065s][info][gc,cpu         ] GC(0) User=0.00s Sys=0.00s Real=0.01s
```

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.

```
[2.070s][info][gc,metaspace   ] GC(1) Metaspace: 362K(512K)->362K(512K) NonClass: 340K(384K)->340K(384K) Class: 22K(128K)->22K(128K)
[2.070s][info][gc             ] GC(1) Pause Full (System.gc()) 2M->0M(128M) 3.546ms
[2.070s][info][gc,cpu         ] GC(1) User=0.00s Sys=0.00s Real=0.00s
DataObject 'LongLived' is being garbage collected.
```

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

```
[4.072s][info][gc,heap,exit   ] Heap
[4.072s][info][gc,heap,exit   ]  garbage-first heap   total 131072K, used 1685K [0x00000000e0000000, 0x0000000100000000)
[4.072s][info][gc,heap,exit   ]   region size 1024K, 2 young (2048K), 0 survivors (0K)
[4.072s][info][gc,heap,exit   ]  Metaspace       used 362K, committed 512K, reserved 1114112K
[4.072s][info][gc,heap,exit   ]   class space    used 22K, committed 128K, reserved 1048576K
```

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.

README.md   C++MemoryManagement.cpp U ×

C++MemoryManagement.cpp > MyObject > ~MyObject()

```cpp
// C++ Code to manage memory manually and with smart pointers
#include <iostream> // For std::cout, std::endl
#include <vector>   // For std::vector (to avoid raw array issues for multiple objects)
#include <memory>   // For smart pointers (to show modern C++ approach)

// A simple class to demonstrate memory allocation/deallocation
class MyObject
{
public:
    int value;
    // Constructor
    MyObject(int val) : value(val)
    {
        std::cout << "MyObject(" << value << ") created." << std::endl;
    }
    // Destructor
    ~MyObject()
    {
        std::cout << "MyObject(" << value << ") destroyed." << std::endl;
    }
};
```

```cpp
// Function demonstrating manual allocation and deallocation
void manualMemoryFunction()
{
    std::cout << "\n--- Manual Memory Management ---" << std::endl;

    // 1. Basic Allocation and Deallocation
    // Allocate a single MyObject on the heap
    MyObject *obj1 = new MyObject(10);
    std::cout << "obj1 value: " << obj1->value << std::endl;
    // Deallocate the object when no longer needed
    delete obj1;
    obj1 = nullptr; // Set to nullptr to avoid dangling pointer

    // 2. Demonstrating a potential memory leak
    std::cout << "\n--- Potential Memory Leak Scenario ---" << std::endl;
    MyObject *leakObj = new MyObject(20);
    // If we forget to call 'delete leakObj' here, this memory will be leaked.
    std::cout << "Leak object created. Forgetting to delete it..." << std::endl;
    // The pointer 'leakObj' goes out of scope, but the allocated memory is not freed.

    // 3. Demonstrating a dangling pointer
    std::cout << "\n--- Dangling Pointer Scenario ---" << std::endl;
    MyObject *danglePtr = new MyObject(30);
    delete danglePtr; // Memory is freed
    // danglePtr is now a dangling pointer: it points to freed memory.
    // Using it is undefined behavior.
    // std::cout << "Attempting to use dangling pointer: " << danglePtr->value << std::endl;

    // 4. Allocating an array
    std::cout << "\n--- Allocating an Array ---" << std::endl;
    MyObject *objArray = new MyObject[3]{MyObject(40), MyObject(41), MyObject(42)};
    std::cout << "objArray[0] value: " << objArray[0].value << std::endl;
    // Must use 'delete[]' for arrays
    delete[] objArray;
    objArray = nullptr;

    std::cout << "--- End of Manual Memory Management Section ---" << std::endl;
}
```

```
// Function demonstrating modern C++ with smart pointers
void smartPointerFunction()
{
    std::cout << "\n--- Smart Pointer Memory Management ---" << std::endl;

    // 1. std::unique_ptr: Exclusive ownership
    std::cout << "std::unique_ptr demo:" << std::endl;
    std::unique_ptr<MyObject> u_ptr = std::make_unique<MyObject>(50); // Allocated on heap
    std::cout << "u_ptr value: " << u_ptr->value << std::endl;
    // No 'delete' needed. Memory automatically freed when u_ptr goes out of scope.

    // Transfer ownership
    std::unique_ptr<MyObject> u_ptr_moved = std::move(u_ptr);
    // std::cout << "u_ptr after move: " << u_ptr->value << std::endl; // Compile error: u_ptr is moved
    std::cout << "u_ptr_moved value: " << u_ptr_moved->value << std::endl;

    // 2. std::shared_ptr: Shared ownership (reference counting)
    std::cout << "\nstd::shared_ptr demo:" << std::endl;
    std::shared_ptr<MyObject> s_ptr1 = std::make_shared<MyObject>(60);
    std::cout << "s_ptr1 value: " <<
    std::cout << "s_ptr1 use count: "                                    ;

    std::shared_ptr<MyObject> s_ptr2                                     the same object
    std::cout << "s_ptr2 value: " << s_ptr2->value << std::endl;
    std::cout << "s_ptr1 use count after s_ptr2: " << s_ptr1.use_count() << std::endl;

    // Object destroyed when last shared_ptr goes out of scope.
    std::cout << "s_ptr1 and s_ptr2 will go out of scope at end of function." << std::endl;
    std::cout << "--- End of Smart Pointer Memory Management Section ---" << std::endl;
}
```

std::shared_ptr<MyObject> s_ptr2

s_ptr2 also points to the same object

Generate Copilot summary

```
// Main function to demonstrate memory management
int main()
{
    std::cout << "--- C++ Memory Management ---" << std::endl;

    manualMemoryFunction();
    smartPointerFunction();

    // The leaked object from manualMemoryFunction will only be truly reclaimed
    // when the program exits. This highlights the leak.

    std::cout << "--- End of C++ program ---" << std::endl;
    return 0;
}
```

Output

```
--- C++ Memory Management ---

--- Manual Memory Management ---
MyObject(10) created.
obj1 value: 10
MyObject(10) destroyed.

--- Potential Memory Leak Scenario ---
MyObject(20) created.
Leak object created. Forgetting to delete it...

--- Dangling Pointer Scenario ---
MyObject(30) created.
MyObject(30) destroyed.

--- Allocating an Array ---
MyObject(40) created.
MyObject(41) created.
MyObject(42) created.
objArray[0] value: 40
MyObject(42) destroyed.
MyObject(41) destroyed.
MyObject(40) destroyed.
--- End of Manual Memory Management Section ---

--- Smart Pointer Memory Management ---
std::unique_ptr demo:
MyObject(50) created.
u_ptr value: 50
u_ptr_moved value: 50

std::shared_ptr demo:
MyObject(60) created.
s_ptr1 value: 60
s_ptr1 use count: 1
s_ptr2 value: 60
s_ptr1 use count after s_ptr2: 2
s_ptr1 and s_ptr2 will go out of scope at end of function.
--- End of Smart Pointer Memory Management Section ---
MyObject(60) destroyed.
MyObject(50) destroyed.
--- End of C++ program ---
```

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"

The above memory profiling data was generated.

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

```
TERMINAL
--------------------------------------------------------------------------------
  0              0                0                0             0            0
  1         41,957              312                0             0          312
  2         59,600            7,680                0             0        7,680
00.00% (0B) (heap allocation functions) malloc/new/new[], --alloc-fns, etc.

--------------------------------------------------------------------------------
  n        time(i)         total(B)   useful-heap(B) extra-heap(B)    stacks(B)
--------------------------------------------------------------------------------
  3         85,534            4,248                0             0        4,248
  4        114,698            1,360                0             0        1,360
  5        131,869            1,304                0             0        1,304
  6        172,401            1,768                0             0        1,768
  7        199,426            1,304                0             0        1,304
  8        225,440            1,304                0             0        1,304
  9        254,694            1,304                0             0        1,304
 10        278,086            1,528                0             0        1,528
 11        318,431            1,768                0             0        1,768
 12        349,166            1,304                0             0        1,304
 13        373,211            1,768                0             0        1,768
 14        392,674            1,768                0             0        1,768
 15        434,542            1,528                0             0        1,528
 16        462,692            1,528                0             0        1,528
 17        495,698            1,880                0             0        1,880
 18        518,782            1,304                0             0        1,304
 19        555,486            1,768                0             0        1,768
 20        588,221            1,880                0             0        1,880
 21        626,548            1,768                0             0        1,768
 22        658,082            1,528                0             0        1,528
 23        697,734            1,864                0             0        1,864
 24        722,695            1,768                0             0        1,768
 25        747,513            1,528                0             0        1,528
 26        784,925            1,768                0             0        1,768
 27        816,081            1,768                0             0        1,768
 28        848,327            1,528                0             0        1,528
 29        880,700            1,528                0             0        1,528
 30        912,121            1,528                0             0        1,528
 31        935,281            1,624                0             0        1,624
 32        961,423            1,528                0             0        1,528
 33      1,001,587            1,528                0             0        1,528
 34      1,020,714            1,880                0             0        1,880
 35      1,054,157            1,304                0             0        1,304
 36      1,078,768            1,528                0             0        1,528
 37      1,103,945            1,864                0             0        1,864
 38      1,122,317            1,304                0             0        1,304
 39      1,149,987            1,624                0             0        1,624
 40      1,189,266            1,304                0             0        1,304
00.00% (0B) (heap allocation functions) malloc/new/new[], --alloc-fns, etc.
```

```
admin@Linux:~/MSCS632$ ms_print massif.out
--------------------------------------------------------------------------------
Command:            ./cpp_memory_debug
Massif arguments:   --stacks=yes --massif-out-file=massif.out
ms_print arguments: massif.out
--------------------------------------------------------------------------------


    KB
74.57^                                                                       #
     |                                                                      @#::
     |                                                                      @#::
     |                                                                      @#::
     |                                                                      @#::
     |                                                                      @#::
     |                                                                      @#::
     |                                                                      @#::
     |                                                                      @#::
     |                                                                      @#::
     |                                                                      @#::
     |                                                                      @#::
     |                                                                      @#::
     |                                                                      @#::
     |                                                                      @#::
     |                                                                      @#::
     |                                                                      @#::
     |                                                                      @#::
     |  @                                                                   @#::
     |  @                                                                   @#::
   0 +----------------------------------------------------------------------->Mi
     0                                                                     1.971

Number of snapshots: 92
 Detailed snapshots: [2, 40, 41, 57, 67, 77, 84, 85, 86 (peak)]
```

TERMINAL

```
--------------------------------------------------------------------------------
  0              0                0                0             0            0
  1         41,957              312                0             0          312
  2         59,600            7,680                0             0        7,680
00.00% (0B) (heap allocation functions) malloc/new/new[], --alloc-fns, etc.

--------------------------------------------------------------------------------
  n        time(i)         total(B)   useful-heap(B) extra-heap(B)    stacks(B)
--------------------------------------------------------------------------------
  3         85,534            4,248                0             0        4,248
  4        114,698            1,360                0             0        1,360
  5        131,869            1,304                0             0        1,304
  6        172,401            1,768                0             0        1,768
  7        199,426            1,304                0             0        1,304
  8        225,440            1,304                0             0        1,304
  9        254,694            1,304                0             0        1,304
 10        278,086            1,528                0             0        1,528
 11        318,431            1,768                0             0        1,768
 12        349,166            1,304                0             0        1,304
 13        373,211            1,768                0             0        1,768
 14        392,674            1,768                0             0        1,768
 15        434,542            1,528                0             0        1,528
 16        462,692            1,528                0             0        1,528
 17        495,698            1,880                0             0        1,880
 18        518,782            1,304                0             0        1,304
 19        555,486            1,768                0             0        1,768
 20        588,221            1,880                0             0        1,880
 21        626,548            1,768                0             0        1,768
 22        658,082            1,528                0             0        1,528
 23        697,734            1,864                0             0        1,864
 24        722,695            1,768                0             0        1,768
 25        747,513            1,528                0             0        1,528
 26        784,925            1,768                0             0        1,768
 27        816,081            1,768                0             0        1,768
 28        848,327            1,528                0             0        1,528
 29        880,700            1,528                0             0        1,528
 30        912,121            1,528                0             0        1,528
 31        935,281            1,624                0             0        1,624
 32        961,423            1,528                0             0        1,528
 33      1,001,587            1,528                0             0        1,528
 34      1,020,714            1,880                0             0        1,880
 35      1,054,157            1,304                0             0        1,304
 36      1,078,768            1,528                0             0        1,528
 37      1,103,945            1,864                0             0        1,864
 38      1,122,317            1,304                0             0        1,304
 39      1,149,987            1,624                0             0        1,624
 40      1,189,266            1,304                0             0        1,304
00.00% (0B) (heap allocation functions) malloc/new/new[], --alloc-fns, etc.
```

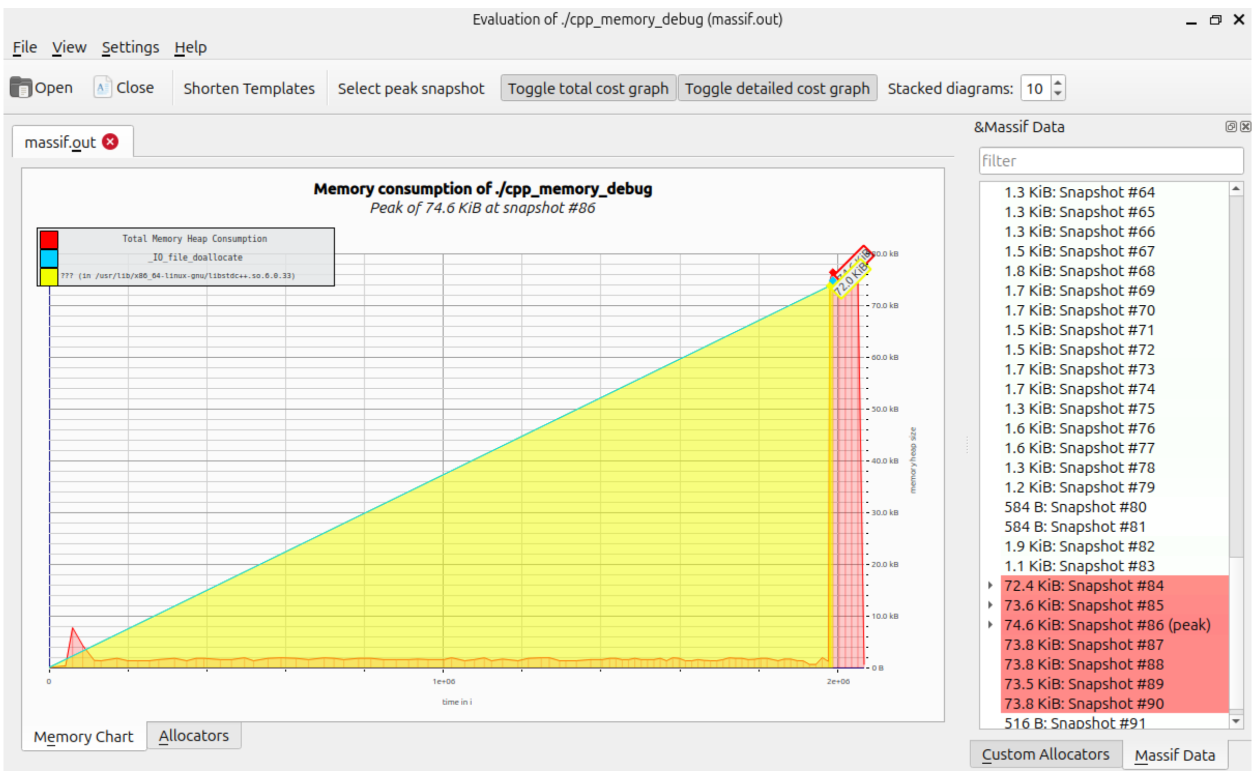
TERMINAL

```
--------------------------------------------------------------------------------
  n        time(i)         total(B)   useful-heap(B) extra-heap(B)    stacks(B)
--------------------------------------------------------------------------------
 41      1,215,599            1,768                0             0        1,768
00.00% (0B) (heap allocation functions) malloc/new/new[], --alloc-fns, etc.

--------------------------------------------------------------------------------
  n        time(i)         total(B)   useful-heap(B) extra-heap(B)    stacks(B)
--------------------------------------------------------------------------------
 42      1,248,429            1,880                0             0        1,880
 43      1,268,673            1,880                0             0        1,880
 44      1,294,794            1,304                0             0        1,304
 45      1,310,658            1,304                0             0        1,304
 46      1,337,531            1,304                0             0        1,304
 47      1,373,759            1,528                0             0        1,528
 48      1,402,813            1,528                0             0        1,528
 49      1,421,773            1,768                0             0        1,768
 50      1,437,729            1,768                0             0        1,768
 51      1,453,593            1,520                0             0        1,520
 52      1,469,633            1,528                0             0        1,528
 53      1,485,561            1,768                0             0        1,768
 54      1,501,885            1,528                0             0        1,528
 55      1,517,817            1,528                0             0        1,528
 56      1,533,902            1,528                0             0        1,528
 57      1,549,971            1,768                0             0        1,768
00.00% (0B) (heap allocation functions) malloc/new/new[], --alloc-fns, etc.

--------------------------------------------------------------------------------
  n        time(i)         total(B)   useful-heap(B) extra-heap(B)    stacks(B)
--------------------------------------------------------------------------------
 58      1,565,972            1,304                0             0        1,304
 59      1,581,861            1,880                0             0        1,880
 60      1,597,926            1,768                0             0        1,768
 61      1,613,973            1,304                0             0        1,304
 62      1,629,938            1,304                0             0        1,304
 63      1,645,826            1,528                0             0        1,528
 64      1,661,713            1,368                0             0        1,368
 65      1,677,646            1,304                0             0        1,304
 66      1,693,523            1,304                0             0        1,304
 67      1,709,423            1,528                0             0        1,528
00.00% (0B) (heap allocation functions) malloc/new/new[], --alloc-fns, etc.
```

```
--------------------------------------------------------------------------------
  n        time(i)         total(B)   useful-heap(B) extra-heap(B)    stacks(B)
--------------------------------------------------------------------------------
 68      1,725,302            1,872                0             0        1,872
 69      1,741,319            1,768                0             0        1,768
 70      1,757,187            1,768                0             0        1,768
 71      1,773,180            1,528                0             0        1,528
 72      1,789,066            1,584                0             0        1,584
 73      1,805,023            1,768                0             0        1,768
 74      1,820,887            1,760                0             0        1,760
 75      1,849,000            1,304                0             0        1,304
 76      1,864,879            1,624                0             0        1,624
 77      1,881,070            1,624                0             0        1,624
00.00% (0B) (heap allocation functions) malloc/new/new[], --alloc-fns, etc.

--------------------------------------------------------------------------------
  n        time(i)         total(B)   useful-heap(B) extra-heap(B)    stacks(B)
--------------------------------------------------------------------------------
 78      1,896,987            1,320                0             0        1,320
 79      1,912,862            1,272                0             0        1,272
 80      1,928,730              584                0             0          584
 81      1,944,887              584                0             0          584
 82      1,960,756            1,912                0             0        1,912
 83      1,976,634            1,176                0             0        1,176
 84      1,979,903           74,136           73,728             8          400
99.45% (73,728B) (heap allocation functions) malloc/new/new[], --alloc-fns, etc.
->99.45% (73,728B) 0x491B38E: ??? (in /usr/lib/x86_64-linux-gnu/libstdc++.so.6.0.33)
  ->99.45% (73,728B) 0x400571E: call_init.part.0 (dl-init.c:74)
    ->99.45% (73,728B) 0x4005823: call_init (dl-init.c:120)
      ->99.45% (73,728B) 0x4005823: _dl_init (dl-init.c:121)
        ->99.45% (73,728B) 0x401F59F: ??? (in /usr/lib/x86_64-linux-gnu/ld-linux-x86-64.so.2)

--------------------------------------------------------------------------------
  n        time(i)         total(B)   useful-heap(B) extra-heap(B)    stacks(B)
--------------------------------------------------------------------------------
 85      1,982,055           75,328           73,728             8        1,592
97.88% (73,728B) (heap allocation functions) malloc/new/new[], --alloc-fns, etc.
->97.88% (73,728B) 0x491B38E: ??? (in /usr/lib/x86_64-linux-gnu/libstdc++.so.6.0.33)
  ->97.88% (73,728B) 0x400571E: call_init.part.0 (dl-init.c:74)
    ->97.88% (73,728B) 0x4005823: call_init (dl-init.c:120)
      ->97.88% (73,728B) 0x4005823: _dl_init (dl-init.c:121)
        ->97.88% (73,728B) 0x401F59F: ??? (in /usr/lib/x86_64-linux-gnu/ld-linux-x86-64.so.2)
```

```
--------------------------------------------------------------------------------
  n        time(i)         total(B)   useful-heap(B) extra-heap(B)    stacks(B)
--------------------------------------------------------------------------------
 86      1,987,803           76,360           74,752           16        1,592
97.89% (74,752B) (heap allocation functions) malloc/new/new[], --alloc-fns, etc.
->96.55% (73,728B) 0x491B38E: ??? (in /usr/lib/x86_64-linux-gnu/libstdc++.so.6.0.33)
| ->96.55% (73,728B) 0x400571E: call_init.part.0 (dl-init.c:74)
|   ->96.55% (73,728B) 0x4005823: call_init (dl-init.c:120)
|     ->96.55% (73,728B) 0x4005823: _dl_init (dl-init.c:121)
|       ->96.55% (73,728B) 0x401F59F: ??? (in /usr/lib/x86_64-linux-gnu/ld-linux-x86-64.so.2)
|
->01.34% (1,024B) 0x4B951B4: _IO_file_doallocate (filedoalloc.c:101)
  ->01.34% (1,024B) 0x4BA5523: _IO_doallocbuf (genops.c:347)
    ->01.34% (1,024B) 0x4BA2F8F: _IO_file_overflow@@GLIBC_2.2.5 (fileops.c:745)
      ->01.34% (1,024B) 0x4BA3AAE: _IO_new_file_xsputn (fileops.c:1244)
        ->01.34% (1,024B) 0x4BA3AAE: _IO_file_xsputn@@GLIBC_2.2.5 (fileops.c:1197)
          ->01.34% (1,024B) 0x4B96A11: fwrite (iofwrite.c:39)
            ->01.34% (1,024B) 0x49BADC3: std::basic_ostream<char, std::char_traits<char> >& std::__ostream_insert<char, std::char_traits<char> >(std::basic_ostrea
m<char, std::char_traits<char> >&, char const*, long) (in /usr/lib/x86_64-linux-gnu/libstdc++.so.6.0.33)
              ->01.34% (1,024B) 0x49BB13B: std::basic_ostream<char, std::char_traits<char> >& std::operator<< <std::char_traits<char> >(std::basic_ostream<char, s
td::char_traits<char> >&, char const*) (in /usr/lib/x86_64-linux-gnu/libstdc++.so.6.0.33)
                ->01.34% (1,024B) 0x109A7E: main (C++MemoryManagement.cpp:96)

--------------------------------------------------------------------------------
  n        time(i)         total(B)   useful-h Open file in editor (ctrl + click)  stacks(B)
--------------------------------------------------------------------------------
 87      2,003,676           75,608           74,752           16          840
 88      2,019,560           75,584           74,776           56          752
 89      2,035,429           75,328           74,760           56          512
 90      2,051,308           75,576           74,760           56          760
 91      2,067,174              536                4           20          512
```

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



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.

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