

DSA - Mini Project-Report

Aakash Desai - PES1UG24CS006

Aarush Muralidhara - PES1UG24CS010

Abhay Balakrishna - PES1UG24CS012

Problem Statement - Visualize how unreachable objects accumulate and cause memory pressure in managed runtimes.

Functionalities - Create objects, create references , Run GC simulation , visualize unreachable subgraphs, the two additional functionalities were: Force leak scenarios; Export snapshots.

Linear Data Structures - Linked List, Stack

Non-Linear Data Structure - Graph (object reference graph)

Properties-

1. Graph (Object Reference Graph)

- Property 1: It's a Directed Graph. References are one-way. An object A pointing to B does not mean B points back to A. This is the fundamental property that makes "reachability" a one-way street.
- Property 2: It's a Cyclic Graph. It can (and in our test cases, does) have cycles, where $A \rightarrow B \rightarrow C \rightarrow A$. This is a critical property because it's why simple reference counting fails and why a tracing GC (like our mark-and-sweep) is necessary.
- Property 3: It's an Unweighted Graph. The references (edges) just represent a connection; they don't have a "cost" or "weight."
- Property 4: It's Implemented as an Adjacency List. Each object (vertex) holds its own linked list (RefNode *refs) of the objects it points to. This is an implementation-level property.

2. Linked List (heap_head list)

- Property 1: It's a Singly Linked List. Each object points to the next object in the heap, but not to the previous one.
Property 2: It's a Dynamic Data Structure. It can grow and shrink at runtime using malloc() and free(). This is its primary purpose-to manage a collection of objects whose size isn't known at compile time.
- Property 3: It has $O(1)$ Insertion at the Head. Our create_object function adds new objects to the front of the list in constant time, which is very efficient.
- Property 4: It has $O(n)$ Traversal/Search Time. To find a specific object (like in find_object_by_name) or to visit every object (like in g_sweep), you must traverse the list from the beginning.

3. The Stack (Used for GC Marking)

- Property 1: It's a LIFO (Last-In, First-Out) Structure. This is its defining characteristic.
- Property 2: It's Implemented Implicitly. The `g_mark` function is recursive. Recursion uses the program's internal call stack to manage function calls. This call stack is the stack data structure.
- Property 3: It Enables Depth-First Search (DFS). The LIFO property of the call stack is exactly what allows your `go_mark` function to explore one path of references as far as it can go before backtracking, which is the definition of a DFS.

Advantages -

Linked List :

- **Dynamic memory usage:** Nodes are created as needed — similar to object allocation in managed runtimes.
- **Simple visualization of references:** Each node clearly points to the next, making it easy to show how breaking a link causes downstream nodes to become unreachable.
- **Demonstrates GC concept effectively:** If the head (root) reference is lost, the entire list becomes **unreachable** — perfect for illustrating GC reachability.
- **Easy to implement:** Conceptually and programmatically simple — ideal for educational simulations.

Stack :

- **Represents the runtime call stack naturally:** Perfect for simulating **root references** (local variables, active function calls).
- **Simple to manage:** Push and pop operations make it easy to simulate scope creation and destruction.
- **Supports GC root simulation:** When you pop from the stack, the referenced object can become unreachable — illustrating real GC triggers.
- **Efficient memory model:** Constant-time push/pop and minimal overhead

Graph :

- **Realistic representation of memory:** Objects in managed runtimes form complex reference networks — graphs model this perfectly.
- **Reachability analysis:** Using DFS/BFS, you can simulate how GC identifies reachable and unreachable objects.
- **Detects memory leaks and cycles:** Can represent circular references and demonstrate why reference counting alone can fail.
- **Flexible visualization:** Graphs allow arbitrary connections — ideal for visualizing object interdependencies.
- **Dynamic structure:** You can add/remove edges and nodes at runtime to simulate object creation/deletion.

Disadvantages -

Linked List :

- **Complications while Deleting:** Because each object only has a next pointer and no back pointer, removing a node from the heap list requires the complex pointer-to-pointer technique.
- **Not memory efficient for large-scale visualization:** Each node needs extra memory for pointer/reference storage.
- **Limited realism:** Real-world object references rarely form simple chains; they often form **graphs** with multiple incoming/outgoing links.
- **Breaking a link isolates a large subgraph at once:** It can oversimplify GC behavior compared to a more interconnected object reference graph.

Stack :

- **Limited reference flexibility:** Stack elements can only reference what's below or globally reachable — not ideal for representing cross-object relationships.
- **Short-lived references only:** Real programs have both **stack** (short-term) and **heap** (long-term) references, but a stack alone can't show heap object lifetimes accurately.
- **Does not illustrate cyclic references:** Stack references are acyclic — so you need graphs to show memory leaks caused by cycles.

Graph :

- **Complex implementation:** Building and managing a dynamic graph with reference updates is more involved than linked lists or stacks.
- **Harder to visualize clearly:** For many nodes, the visual graph becomes cluttered, making relationships hard to follow.
- **Traversal overhead:** GC simulation (reachability analysis) requires DFS/BFS — higher computational cost than simple linear structures.
- **Memory usage:** Storing adjacency lists/matrices and metadata for visualization consumes additional memory.

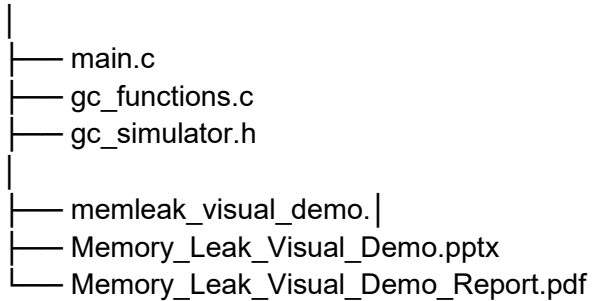
ADT Operations & Algorithm

Core Operations of ObjectGraph:

Operation	Purpose
<u>create_object()</u>	Allocates a new object in the heap (linked list node).
<u>add_reference(from, to)</u>	Adds a directed reference between two objects
<u>gc_mark_all()</u>	Traverses all roots and marks reachable objects recursively.
<u>gc_sweep()</u>	Frees all unmarked (unreachable) objects from memory.
<u>write_dot()</u>	Generates a DOT file to visualize the object graph (Graphviz).
<u>final_cleanup()</u>	Frees all remaining allocated memory at program termination.
<u>force_leak_scenario()</u>	To Simulate and run pre-defined scenarios
<u>reset_simulator_state()</u>	Clears the data/memory values from previous simulations
<u>export_snapshot()</u>	Creates a .txt file with current heap status

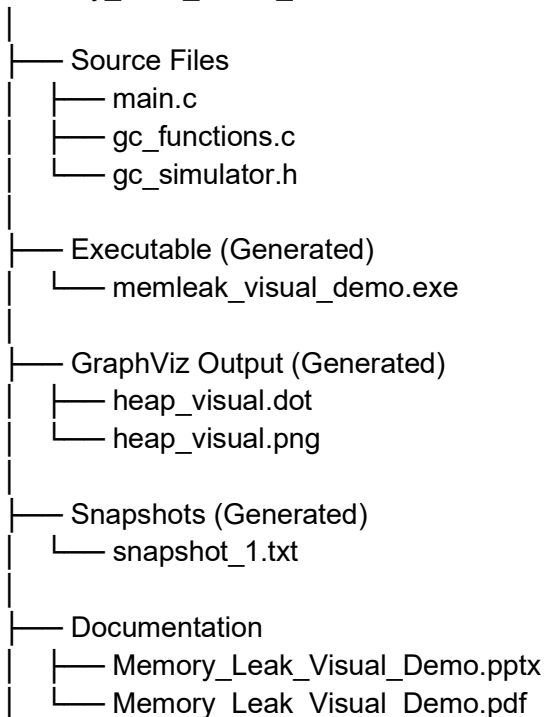
File Hierarchy:

Project/



After Running all the functionalities, the File Hierarchy:

Memory_Leak_Visual_Demo/



Flow Of Code

START



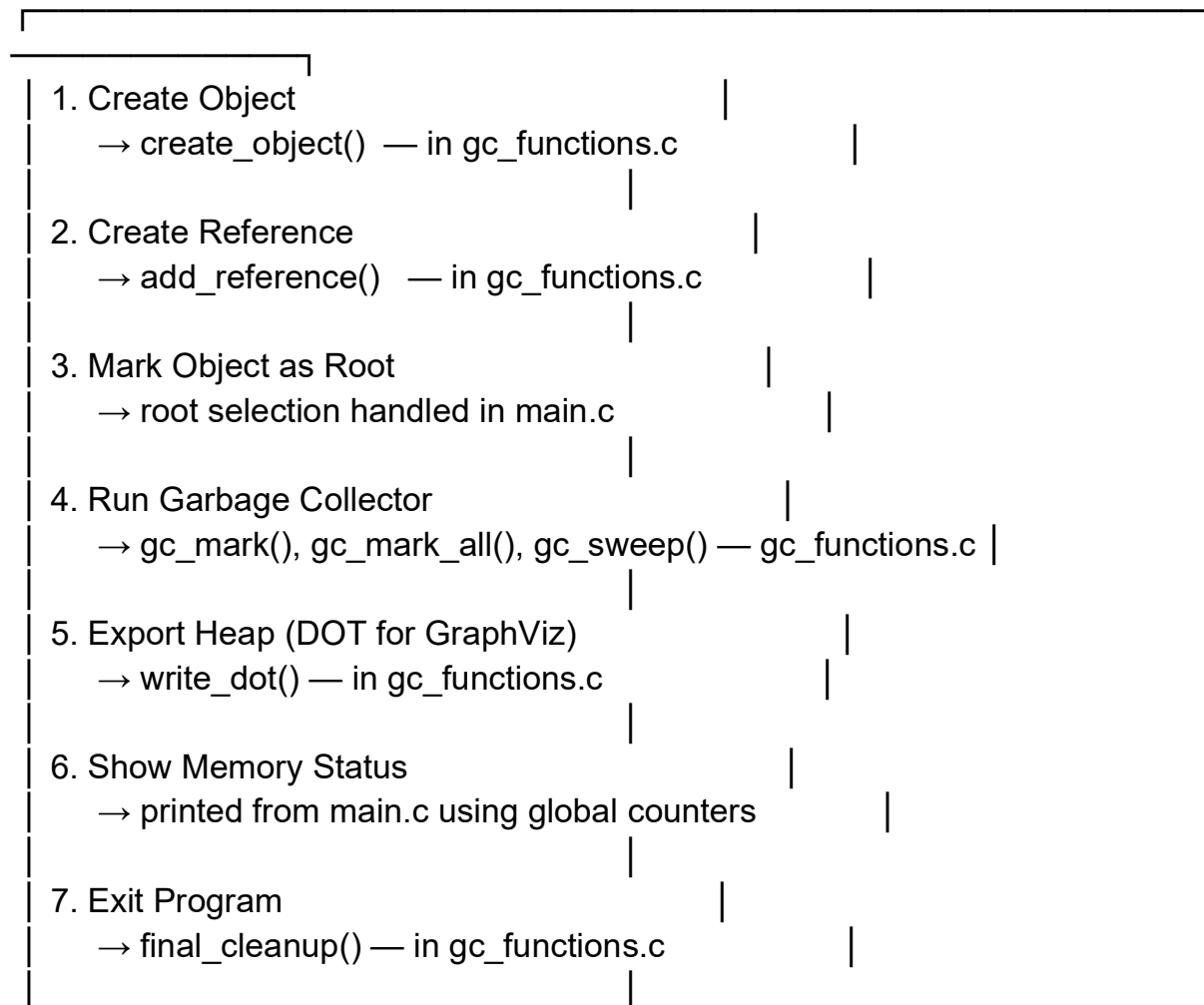
Initialize Globals

(from declarations in gc_simulator.h, definitions in gc_functions.c)



MAIN MENU LOOP

(code inside main.c)



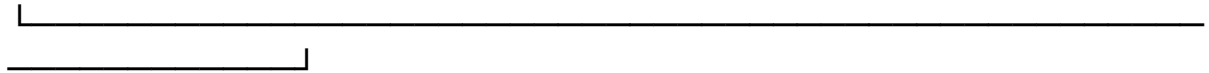
| 8. Force Leak Scenarios |

| → force_leak_scenario() — gc_functions.c |

| → reset_simulator_state() — gc_functions.c |

| 9. Export Snapshot (.txt) |

| → export_snapshot() — gc_functions.c |



↓
Repeat Until User Chooses Exit
(main.c handles loop)

↓
END

Alternative Data Structures - Tree , queue , heap

1. Tree (Hierarchical Structure)

Advantages

- Natural way to show ownership or containment (e.g., GUI hierarchies, object composition).
- Easy to visualize — each parent has well-defined children.
- Reachability is simple (root to leaf traversal).

Disadvantages

- Real programs often have shared references — trees don't allow multiple parents easily.
- Cannot represent cyclic references (important for GC simulation).

Queue

Advantages

- Essential for **BFS-based GC algorithms** (used in mark-sweep).
- Simple FIFO structure for processing reachable objects layer by layer.

Disadvantages

- Only useful internally — not ideal for visualization.

Heap (Binary or Custom Memory Heap Model)

Advantages

- Represents actual **heap memory** structure — adds realism.
- Can show fragmentation or memory usage visually.
- Allows simulation of memory allocation and release events.

Disadvantages

- Implementation complexity is higher.
- Not primarily focused on reference visualization — more on memory usage.

Output Snippet

Objective

This test demonstrates the core functionality of the dynamic mark-and-sweep garbage collector. It intentionally creates a disconnected "island" of objects to simulate a memory leak, and verifies that the GC correctly identifies, collects, and frees unreachable objects while preserving reachable (live) ones.

1. Test Setup (Creation Phase)

Objects Created:

Five objects were created sequentially: A, B, C, D, E.

References Built:

Two distinct reference chains were constructed:

- Main chain: $A \rightarrow B \rightarrow C$
- Disconnected "island": $D \rightarrow E$

Root Designation:

Object A was marked as the single root.

2. Initial State (Before GC - The "Leak")

- Memory Status (Choice 6):
Before running the GC, the memory status showed:
- Total objects created: 5
- Total objects freed: 0
- Total memory allocated: 274 bytes
- Total memory freed: 0 bytes
- Current memory in use: 274 bytes

(Values may vary slightly depending on name lengths due to dynamic allocation.)

- Visualization (Choice 5):

The heap visualization (Graphviz .dot file) correctly displayed:

- Reachable objects (A, B, C) as green nodes.
- Unreachable "island" (D, E) as red nodes.
- Root (A) as a light blue double circle.

This confirmed that only objects in the $A \rightarrow B \rightarrow C$ chain were reachable from the root.

3. Execution (Garbage Collection Phase)

Garbage Collector (Choice 4):

The collector was executed.

Console Output:

The program accurately detected and collected the unreachable objects, printing:

[GC] Collecting unreachable object: E

[GC] Collecting unreachable object: D

[GC] Cycle complete -> 2 object(s) collected.

[GC] Memory freed this cycle: 116 bytes

[GC] Current memory in use: 158 bytes

Result:

- Two unreachable objects (D and E) were successfully freed.
- Freed memory = 116 bytes (reflecting both Object + dynamic name + RefNode allocations).
- The heap now contained only A, B, and C, all reachable from root A.

4. Final State (Verification Phase)

Memory Status (Choice 6):

Total objects created: 5

Total objects freed: 2

Total memory allocated: 274 bytes

Total memory freed: 116 bytes

Current memory in use: 158 bytes

Re-running the GC (Choice 4):

The collector was executed again.

Output confirmed a clean heap:

[GC] Cycle complete -> 0 object(s) collected.

Program Exit (Choice 7):

When the program exited, `final_cleanup()` executed automatically, printing:

[Cleanup] Freeing remaining objects...

Freeing survivor: C

Freeing survivor: B

Freeing survivor: A

All memory freed successfully.

Conclusion

The test was a complete success.

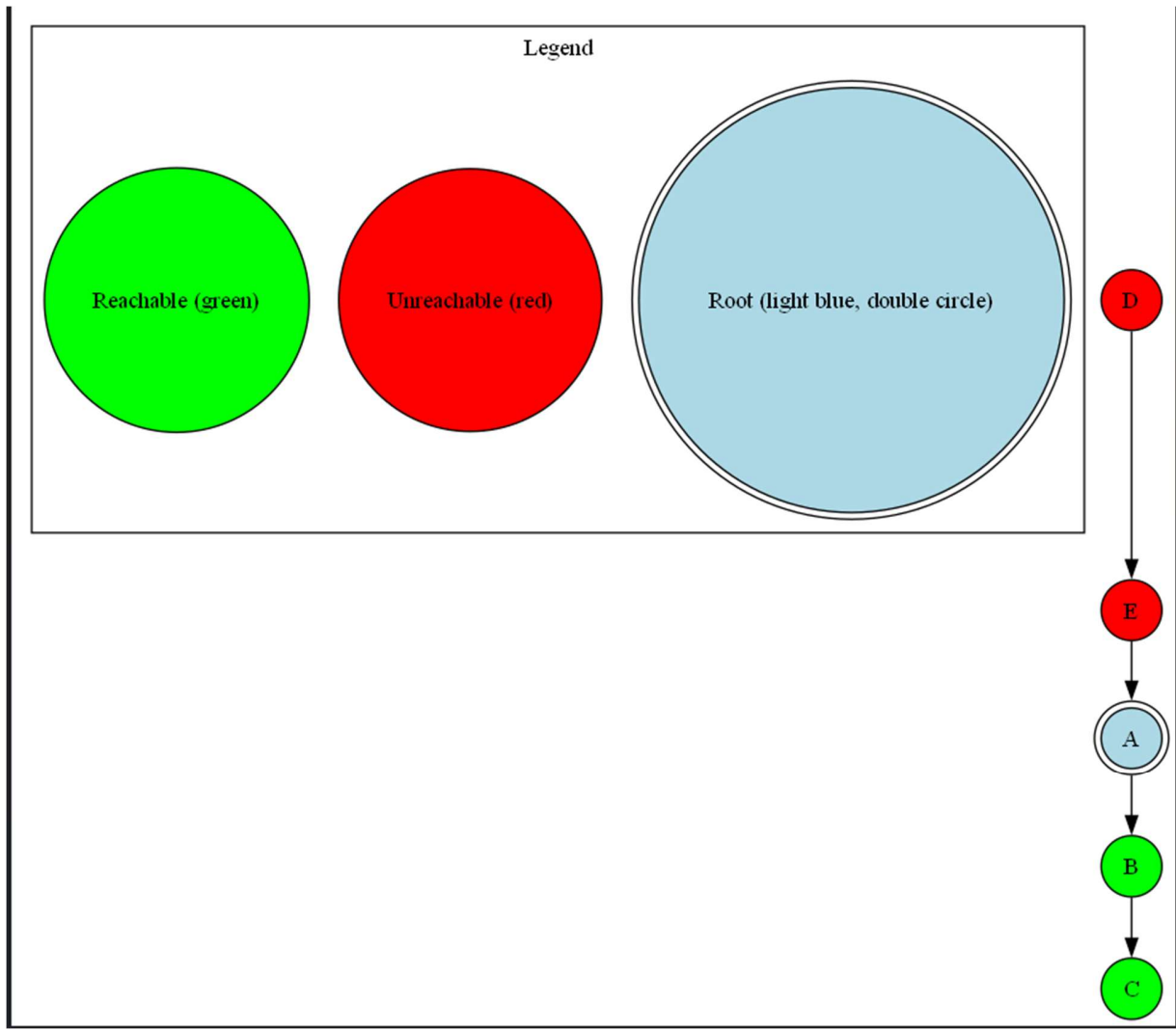
It demonstrated that the garbage collector:

- Accurately detects unreachable objects (D, E).
- Frees all unreferenced memory safely.
- Preserves reachable data (A, B, C).
- Updates memory statistics dynamically using real object sizes.

- Provides correct visualization and reporting at every phase.

This verifies that the updated implementation — with dynamic string allocation, precise memory tracking, and safe recursive marking — works exactly as intended and handles memory cleanup robustly.

Visual Output



Concise Form of the Output

Output (Memory Status + Visualization)

```
===== MEMORY LEAK VISUAL DEMO =====
1. Create Object
2. Create Reference
3. Mark Object as Root
4. Run Garbage Collector
5. Visualize Heap (DOT)
6. Show Memory Status
7. Exit
```

```
Enter choice: 1
Enter object name: A
Created object 'A' (approx 42 bytes)
```

```
Enter choice: 2
Enter source object name: A
Enter target object name: B
Reference created: A -> B
```

```
Enter choice: 3
Enter object name to mark as root: A
Object 'A' marked as root.
```

```
Enter choice: 4
Running garbage collector...
Before GC: 274 bytes in use
[GC] Collecting unreachable object: E
[GC] Collecting unreachable object: D
[GC] Cycle complete -> 2 object(s) collected.
[GC] Memory freed this cycle: 116 bytes
[GC] Current memory in use: 158 bytes

After GC: 158 bytes in use
```

```
Enter choice: 5
Marking heap for visualization...
DOT file generated: heap_visual.dot
```

```
Enter choice: 6

--- Memory Status ---
Total objects created: 5
Total objects freed: 2
Total memory allocated: 274 bytes
Total memory freed: 116 bytes
Current memory in use: 158 bytes
```

Additional Implementations

```
Enter choice: 8

--- Force Leak Scenarios ---
1. Simple chain leak (A -> B -> C and D -> E)
2. Cyclic leak (A -> B -> C -> A and D <-> E)
3. Long chain + garbage nodes
Select scenario: 1
[Force Leak] Creating scenario 1...
Created object 'A' (approx 42 bytes)
Created object 'B' (approx 42 bytes)
Created object 'C' (approx 42 bytes)
Created object 'D' (approx 42 bytes)
Created object 'E' (approx 42 bytes)
Reference created: A -> B
Reference created: B -> C
Reference created: D -> E
[Force Leak] Scenario 1 created.

--- Unreachable Objects (Garbage) ---
- D
- E

Total unreachable = 2 object(s)
Garbage memory (will be freed now) = 84 bytes

[GC] Collecting unreachable object: E
[GC] Collecting unreachable object: D
[GC] Cycle complete -> 2 object(s) collected.
[GC] Memory freed this cycle: 100 bytes
[GC] Current memory in use: 158 bytes

--- Memory Status After Scenario ---
Total objects created: 5
Total objects freed: 2
Total memory allocated: 258 bytes
Total memory freed: 100 bytes
Current memory in use: 158 bytes
```

```
Enter choice: 9
Snapshot exported successfully: snapshot_1.txt
```

```
snapshot_1.txt
File Edit View

===== HEAP SNAPSHOT =====
Timestamp: Thu Nov 13 15:24:12 2025

Total objects created: 5
Total objects freed: 2
Total memory allocated: 258 bytes
Total memory freed: 100 bytes
Current memory in use: 158 bytes

=== Objects in Heap ===
Object: C (ID: 1002)
No references

Object: B (ID: 1001)
-> C

Object: A (ID: 1000)
-> B

===== END OF SNAPSHOT =====
```