




Garbage Collection

M B Thejesshwar

24AI10035



Why Does Memory Management Matter?



What is Memory Management?

Memory management is all about handling how a program uses memory—allocating it when needed and freeing it up when it's no longer in use. Doing this efficiently is crucial for keeping programs fast and stable.

Why Should We Care?

Better Performance – Proper memory management ensures programs run smoothly without unnecessary slowdowns.

Avoiding Crashes & Leaks – Poor handling can lead to memory leaks, where memory keeps getting used but never released, eventually crashing the program.

Scalability – As programs get bigger, messy memory management can make them harder to maintain and debug.

Why Does Memory Management Matter?



How Do Different Languages Handle It?

Manual Management – Languages like C/C++ require programmers to manually allocate and free memory (malloc/free).

Garbage Collection (GC) – Languages like Java, Python, and Go automatically clean up unused memory.

Ownership Model – Rust takes a different approach by enforcing strict rules on how memory is owned and borrowed, preventing leaks at compile time.

How It Works

In languages like C/C++, memory is manually allocated using malloc() and released using free().

The programmer is responsible for keeping track of allocated memory and ensuring it is properly freed.

```
1  #include <stdlib.h>
2  int main() {
3      int* ptr = malloc( Size: sizeof(int));
4      *ptr = 42;
5      free(ptr);
6  }
7  |
```


Manual Memory Management: malloc & free



Challenges of Manual Memory Management

Memory Leaks – If `free()` is forgotten, memory remains allocated, leading to increased usage over time.

Dangling Pointers – Freeing memory too early can lead to accessing invalid memory, causing crashes.

Double Free Errors – Calling `free()` twice on the same pointer can corrupt memory.

Complexity – Large programs require careful tracking of allocated memory, making debugging difficult.


Why It's a Problem?

Works fine for small programs, but in large-scale applications, manual memory management becomes error-prone.

Bugs related to memory issues can be hard to find and fix.

Modern programming languages offer better solutions like Garbage Collection (GC) or Ownership Models to simplify memory handling.

Garbage Collection (GC): What & Why



What is Garbage Collection?


Garbage Collection (GC) is an automatic memory management technique used in languages like Java, Python, and Go. It helps programs manage memory by automatically reclaiming unused memory, preventing memory leaks.

How Does It Work?


The program allocates memory for objects.

When objects are no longer needed (i.e., they have no references), the garbage collector automatically removes them.

This process happens in the background, freeing developers from manual memory management.



Garbage Collection (GC): What & Why



Why Do We Need GC?

Prevents Memory Leaks – No need to manually track memory allocations.

Simplifies Code – Developers don't have to worry about freeing memory.

Reduces Crashes – No risk of accessing freed memory (dangling pointers).

Improves Security – Reduces the risk of memory-related vulnerabilities like buffer overflows.


But... Is GC Always Good?

Performance Overhead– GC runs periodically, sometimes causing slowdowns.

Less Control– Developers can't precisely decide when memory is freed.

More CPU Usage– Can pause execution when collecting garbage, especially in real-time applications.

Unpredictable Timing – GC can introduce latency spikes, making it unsuitable for real-time applications.



Garbage Collection (GC): What & Why

When to Use GC?

Ideal for high-level applications where ease of development is a priority (e.g., web apps, scripting).

Not suitable for systems programming or real-time applications where precise memory control is required.


Where is GC Used?

Java (JVM) – Uses generational garbage collection.

Python – Uses reference counting + cycle detection.

Go – Uses concurrent garbage collection for performance.

Reference Counting – Used in Python, Swift



What is Reference Counting?

A memory management technique where each object keeps track of how many references (or variables) are pointing to it.


When the reference count drops to zero, the object is automatically deleted.

How Does It Work?

Every time an object is assigned to a variable, its reference count increases.

When a variable no longer needs the object, the count decreases.

If no references remain, the object is removed from memory.



Reference Counting – Used in Python, Swift



Where is Reference Counting Used?

Python – Uses reference counting along with cycle detection to manage memory.

Swift – Uses Automatic Reference Counting (ARC) to optimize memory usage.

Pros & Cons of Reference Counting

Immediate Cleanup – Objects are freed as soon as they're no longer needed.

No GC Pauses – Unlike traditional garbage collection, it doesn't cause random slowdowns.

Cyclic References Can Cause Memory Leaks – If two objects reference each other, they won't be deleted automatically.

Extra Overhead – Maintaining reference counts adds some performance cost.

Reference Counting – Used in Python, Swift

Example and Fix for Cyclic Reference

```
1 import sys
2 a=[]
✓ [10] < 10 ms

1 sys.getrefcount(a)#This adds a temporary reference count
✓ [11] < 10 ms
2

1 b=a
✓ [12] < 10 ms

1 sys.getrefcount(b)
✓ [13] 10ms
3

1 del a
✓ [14] < 10 ms

1 sys.getrefcount(b)
✓ [15] 15ms
2
```

```
▶ 1 #Fix for Cyclic Reference
2 import weakref
3 class Node:
4     def __init__(self, value):
5         self.value = value
6         self.ref = None
7 a = Node(1)
8 b = Node(2)
9 a.ref = weakref.ref(b)
10 b.ref = weakref.ref(a)
✓ [17] 11ms
```


Tracing Garbage Collection – Used in Java, Go

What is Tracing Garbage Collection?

A memory management technique where the system automatically finds and removes objects that are no longer in use. Instead of keeping track of reference counts, it traces which objects are still reachable and clears out the rest.

How It Works

Find Active Objects – The garbage collector starts from root objects (e.g., global variables, stack references).

Trace Reachable Objects – It follows all references to determine which objects are still being used.

Remove Unused Objects – Anything not traced is considered garbage and gets deleted.

Where is Tracing GC Used?

Java (JVM) – Uses generational garbage collection to optimize performance.

Go (Golang) – Uses a concurrent garbage collector to minimize pauses.

Tracing Garbage Collection – Used in Java, Go



Pros & Cons of Tracing Garbage Collection

No Need for Manual Memory Management – Developers don't have to worry about freeing memory.

Handles Cyclic References – Unlike reference counting, tracing GC can clean up objects that reference each other.

Can Cause Performance Overhead – Scanning memory takes time, and the program may pause briefly.

Unpredictable Timing – The GC decides when to run, so developers have less control.

Higher CPU Usage – More complex than simple reference counting, which can impact performance.

Go's Approach: Concurrent GC

Unlike traditional stop-the-world GC, Go's garbage collector runs in the background alongside program execution.

This makes it ideal for low-latency applications like web servers and real-time systems.

Generational Garbage Collection – Used in JVM



What is Generational Garbage Collection?

A memory management technique used in the Java Virtual Machine (JVM) to improve efficiency.

Based on the idea that most objects don't last long—they are created, used briefly, and discarded quickly.

JVM splits memory into generations and optimizes garbage collection for each section.

How It Works?

JVM divides the heap into three sections:

Young Generation – Where new objects are allocated. This space is cleaned frequently.

Old (Tenured) Generation – Objects that survive multiple garbage collections move here.

Permanent Generation (Metaspace in Java 8+) – Stores class metadata and method details.

Generational Garbage Collection – Used in JVM



Garbage Collection Process

Minor GC – Quickly cleans up unused objects in the Young Generation. Surviving objects are moved to the Old Generation.

Major GC (Full GC) – Cleans up the Old Generation, which takes more time and can pause the program.

Metaspace Cleanup – JVM automatically manages class metadata memory.

Why Use Generational GC?

Faster Garbage Collection – Short-lived objects are collected quickly, keeping memory clean.


Reduces Performance Impact – Frequent small cleanups prevent large memory slowdowns.

Efficient Memory Management – Helps Java applications run smoothly by optimizing memory usage.

Full GC Can Be Slow – When the Old Generation fills up, a Full GC may pause the application.

Requires Fine-Tuning – Developers sometimes need to tweak JVM settings for the best performance.

Rust's Ownership Model – Borrow Checker, Lifetimes



Why Does Rust Use an Ownership Model?

Unlike languages that rely on garbage collection, Rust manages memory at compile-time.

The ownership system ensures memory safety without runtime overhead.

It prevents common issues like use-after-free, double frees, and memory leaks.

How Ownership Works in Rust

Every value in Rust has one owner at a time.


When the owner goes out of scope, the value is automatically freed.

Ownership can be moved or borrowed, but copying isn't allowed unless explicitly permitted.

Borrowing & Borrow Checker

Instead of transferring ownership, Rust allows borrowing, meaning a function can use a value without taking ownership.

Rust enforces borrowing rules at compile-time using the borrow checker.



Rust's Ownership Model – Borrow Checker, Lifetimes



Mutable vs. Immutable Borrowing

You can have multiple immutable references.

You can only have one mutable reference at a time.

Lifetimes – Solving Borrowing Issues

Lifetimes ensure references remain valid for as long as needed, preventing dangling references.

The compiler checks lifetimes to ensure references never outlive their data.

Why Rust's Ownership Model Stands Out

Memory Safety Without a Garbage Collector

Prevents Dangling Pointers & Data Races

More Efficient Than Reference Counting or GC

Has a Learning Curve (especially for new programmers)

More Restrictions Than GC-Based Languages

Comparison: Garbage Collection vs Ownership Model

Which One Should You Use?

Garbage Collection Works Well For:

Large-scale applications where ease of development is a priority.

Languages like Python, Java, and Go, where occasional GC pauses are acceptable.

Rust's Ownership Model Is Best For:

Performance-critical applications (e.g., game engines, OS kernels).

Embedded systems where memory usage must be tightly controlled.

Concurrent programs that need safe memory management without GC slowdowns.

Final Takeaways

Garbage Collection makes life easier, but it can add performance overhead.

Rust's Ownership Model requires discipline but eliminates unpredictable GC delays.

The best choice depends on your project's needs!

Performance Examples – Code Snippets Showing Impact

GC runs when the JVM decides, so we have no control over when memory is freed.

Large-scale memory allocations trigger frequent GC cycles, which can slow down execution.

```
1 import java.util.ArrayList;
2 public class GCTest {
3     public static void main(String[] args) {
4         ArrayList<int[]> list = new ArrayList<>();
5         long start = System.nanoTime();
6         for (int i = 0; i < 10_000; i++) {
7             list.add(new int[1000]);
8         }
9         System.gc();
10        long end = System.nanoTime();
11        System.out.println("Time taken: " + (end - start) / 1e6 + " ms");
12    }
13 }
```

Time taken: 112.089071 ms
=== Code Execution Successful

Rust:

Memory is freed immediately once variables go out of scope.

No background GC processes—execution remains predictable and consistent.

```
1 use std::time::Instant;
2 fn main() {
3     let start = Instant::now();
4     let mut vec = Vec::new();
5     for _ in 0..10_000 {
6         vec.push(vec![0; 1000]);
7     }
8     let duration = start.elapsed();
9     println!("Time taken: {:.2?}", duration);
10 }
```

Time taken: 23.39ms
=== Code Execution Successful

Performance Examples – Code Snippets Showing Impact

Python's Reference Counting vs GC Delays

Python uses reference counting for memory management, but it also relies on a garbage collector to clean up objects caught in circular references.

```
1 import time
2 import sys
3 class Test:
4     def __del__(self):
5         print("Object destroyed")
6 start = time.time()
7 a = Test()
8 print(sys.getrefcount(a))
9 a = None
10 print("Execution time:", time.time() - start)
```

✓ [20] 16ms

2

Object destroyed

Execution time: 0.0011932849884033203

If an object's reference count drops to zero, it is immediately deallocated.

But if there are circular references, Python's garbage collector must intervene, which can introduce delays.

Performance Takeaways


Rust (Ownership Model) → Memory is managed at compile time, no GC pauses.

Java (GC-Based) → Easier memory management, but GC can cause unpredictable

Python (Ref Counting + GC) → Fast for simple cases, but circular references require GC be slow.

The best choice depends on how much control you need over memory management!

Closing Thoughts



The Future of Memory Management

Hybrid approaches (e.g., Rust's Ownership + Reference Counting in Swift) may gain traction.

Advances in low-latency GC could make traditional GC techniques more efficient.

Memory safety will remain a crucial factor as software complexity grows.

Final Thought

There's no one-size-fits-all solution! Choosing the right memory management model depends on the specific needs of your application.

Thank You!

