5.5. GARBAGE COLLECTION LOGIC

Garbage collection (GC) automatically manages memory by identifying and reclaiming unused memory blocks, preventing memory leaks. It was first implemented in Lisp in 1958. Other garbage collection languages include Java, Perl, ML, Modula-3, Prolog, and Smalltalk.

Garbage collection refers to the process of automatically reclaiming memory occupied by objects that are no longer in use, typically performed by the runtime environment of a programming language.

What is Garbage Collection in Data Structure?

Garbage collection is the process of restoring chunks of storage space that a program can no longer access. We assume that objects have a type that the garbage collector can determine at runtime. Based on the type information, we can determine the size of the object and which components of the object have references (pointers) to other objects. We also assume that all references to objects correspond to the address of the object's beginning rather than pointers to specific locations within the object. Therefore, all objects' references have the same value and are easily identifiable.

We'll refer to a user application as the mutator, which modifies the heap's collection of objects. The mutator generates objects by obtaining space from the memory manager, and it can add and remove references to existing objects. When the mutator can't "reach" an object, it becomes garbage.

The garbage collector locates these inaccessible objects and returns them to the memory manager, which keeps track of the available space.

Why Use Garbage Collection?

Garbage collection (GC) is used to automatically manage memory in programming languages like Java, reducing the risk of **memory leaks** and **dangling pointers**. It helps by:

- **Automatically reclaiming unused objects**, improving memory efficiency.
- **Preventing memory leaks** by removing unreachable objects.
- **Enhancing application stability** by avoiding manual memory management errors.
- **Improving developer productivity**, allowing focus on business logic rather than memory allocation.

How to Choose a Garbage Collector?

Choosing the right garbage collector depends on factors like **application performance needs, heap size, latency requirements, and throughput expectations**. Common GC options in Java:

- **Serial GC** – Best for **small applications** with single-threaded environments.
- **Parallel GC** – Suitable for **high-throughput applications** with multi-core processors.
- **G1 GC** – Ideal for **low-latency applications** requiring balanced performance.

- **ZGC/Shenandoah GC** – Designed for **large heap applications** with **minimal pause times**.

Selecting the right GC requires testing and profiling the application's memory usage and performance.

Types of Garbage Collection in Data Structure

Mark-and-Sweep

The mark-and-sweep algorithm involves two phases: marking and sweeping. During the marking phase, the algorithm traverses the object graph starting from the root, marking all reachable objects. In the sweeping phase, it scans the heap for unmarked objects and reclaims their memory, making it available for future allocations.

Copying

The copying garbage collection algorithm divides the heap into two halves. It allocates objects in one half, and when it runs out of space, it copies live objects to the other half, leaving behind unused space, which gets reclaimed in bulk. This process reduces fragmentation and is efficient for memory allocation but can be less space-efficient due to the division of the heap.

Reference Counting

Reference counting maintains a count of references to each object. When an object's reference count drops to zero, it is considered unreachable and can be immediately reclaimed. While simple and providing immediate reclamation, reference counting has drawbacks, such as handling cyclic references, where objects reference each other, preventing their reference counts from reaching zero.

Generational

Generational garbage collection divides objects into generations based on their lifespan. It assumes that most objects die young, so it focuses on collecting young objects more frequently. The heap is divided into young and old generations, with minor collections occurring frequently in the young generation and less frequently in the old generation. This approach optimizes performance by reducing the overhead of frequent garbage collection.

Incremental

Incremental garbage collection breaks the work of garbage collection into smaller chunks, interleaving it with the program's execution. This reduces pause times and improves the application's responsiveness. The collector incrementally marks and sweeps objects, allowing the program to run concurrently with the garbage collection process.

Concurrent

Concurrent garbage collection runs alongside the application, minimizing pause times and improving responsiveness. It allows the program to continue executing while the garbage collector identifies and reclaims unreachable objects. This

approach is complex to implement but provides significant performance benefits, particularly for interactive applications.

Requirements of Garbage Collection in Data Structure

Type Safety

Not all languages are suitable for automatic garbage collection. A garbage collector must determine whether every given data element or component of a data element is or may be used as a pointer to a piece of allocated memory space to function correctly. A type-safe language is one in which any data component can be determined.

Type-safe languages, such as ML, allow us to decide types at compile time. These are called statically typed languages. Other type safe languages, such as Java, have types that can be decided during runtime rather than compile time. These are called dynamically typed languages. If a language is neither statically nor dynamically type-safe, it is unsafe.

1. Garbage collection is important for avoiding fragmentation and making the most available memory.
2. Garbage collectors are known for causing programs — the mutators — to halt for an unusually long time when garbage collection occurs without warning. As a result, in addition to reducing total execution time, it is preferable to reduce the maximum pause duration.
3. Real-time applications are specific scenarios in which certain computations must be finished within a certain amount of time. We must either disable garbage collection while executing real-time tasks or limit the pause time. As a result, garbage collection is only employed in real-time systems.
4. We can't evaluate a garbage collector's speed based on its running time. The garbage collector is in charge of data placement, impacting the mutator program's data locality. It can increase the temporal locality by freeing up space and reusing it; it can improve the geographical locality by moving data used together in the same cache or pages.

Characteristics of Garbage Collection

1. Many programmers argue that automatic garbage collection is inefficient compared to explicit storage reclamation. However, several studies have shown that well-implemented garbage collectors outperform systems with explicit deallocation.
2. Automatic deallocation relieves a programmer of memory management concerns, improving system writability and reducing development time and costs.
3. Garbage collection is required for functional languages that cannot use a stack-based environment due to unpredictable execution patterns.
4. Garbage collection is generally so expensive that, even though it was established decades ago and effectively avoids memory leaks, many prominent programming languages have yet to implement it.

5. Garbage collection can take a long time. It does not significantly increase an application's total execution time. Since the garbage collector must deal with a large amount of data, using the memory subsystem significantly impacts its speed.

Algorithms

Many algorithms can be used to implement automatic garbage collection in computer software. Some of them are given below.

Classical Algorithms:

1. Reference Counting
2. Mark-Sweep
3. Mark-Compact
4. Copying Collection
5. Non-Copying implicit collection

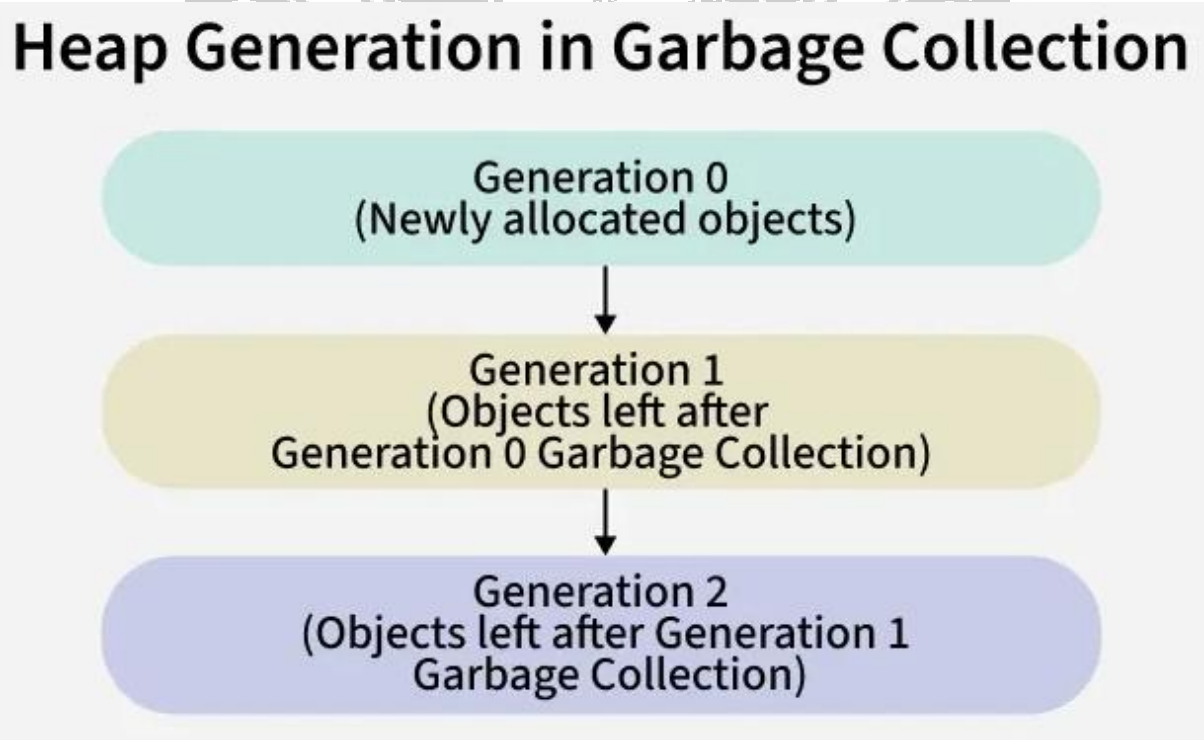
More Creative Algorithms

1. Incremental Tracing Collection
2. Generational Collection

We will read them in detail in the later articles.

Generations in Garbage Collection

To improve performance, the Garbage Collector (GC) organizes objects in the heap into generations. This strategy is based on the observation that most objects are short-lived, while a smaller portion live longer.



Generation 0 (Newly Allocated Objects)

- This is where all new objects are created.
- Examples include local variables, temporary objects like strings created inside a loop or lightweight helper objects.
- Since most objects die quickly, GC frequently collects Generation 0, reclaiming memory immediately.
- Objects that survive this collection are promoted to Generation 1.

Generation 1 (Survivors of Gen 0 Collection)

- Acts as a buffer zone between short-lived and long-lived objects.
- If an object survives one or more GC cycles in Generation 0, it is promoted here.
- This generation is collected less often than Generation 0, balancing performance and memory use.
- Typical objects in Gen 1 are those used beyond a single method call but not throughout the application's lifetime.

Generation 2 (Long-Lived Objects)

- Contains objects that survived multiple collections.
- These are usually long-lived objects, such as static data, application caches or large objects that persist for the lifetime of the program.
- Collected least frequently because scanning this region is expensive.
- The runtime delays cleaning Gen 2 objects unless memory pressure is high.
- Phases of Garbage Collection in C#

1. Marking Phase

- The GC scans objects in memory and identifies live (reachable) objects.
- A list of all live objects is created.
- Unreferenced objects are marked as garbage.

2. Relocating Phase (sometimes included within Mark & Compact)

- References of live objects are updated.
- This ensures that if objects are moved in memory later, all pointers still remain valid.

3. Compacting Phase

- The GC frees memory from dead objects.
- Remaining live objects are compacted (moved together) to eliminate gaps.
- This reduces fragmentation and makes new memory allocation faster.

Important Garbage Collection Methods

The System.GC class provides methods to interact with the garbage collector.

1. GC.Collect()

Forces garbage collection of all generations.

```
using System;

class Program {
    static void Main() {
        for (int i = 0; i < 1000; i++) {
            var obj = new object();
        }

        GC.Collect(); // Forces garbage collection
        GC.WaitForPendingFinalizers();

        Console.WriteLine("Forced garbage collection
completed.");
    }
}
```

2. GC.GetTotalMemory()

Returns the number of bytes currently allocated in managed memory.

```
using System;

class Program {
    static void Main() {
        long before = GC.GetTotalMemory(false);
        int[] arr = new int[1000];
        long after = GC.GetTotalMemory(false);

        Console.WriteLine($"Memory before: {before}");
        Console.WriteLine($"Memory after: {after}");
    }
}
```

3. `GC.MaxGeneration`
Returns the maximum generation supported by the system (usually 2).

```
Console.WriteLine("Maximum Generation: " + GC.MaxGeneration);
```

4. GC.GetGeneration(object obj)

Returns the generation in which a given object resides.

```
string name = "Hello";
Console.WriteLine("Generation of name: " +
GC.GetGeneration(name));
```

5. GC.WaitForPendingFinalizers()

```
GC.Collect();
GC.WaitForPendingFinalizers();
```

Advantages of Garbage Collection

- Eliminates manual memory management.
- Protects against memory leaks.
- Optimizes memory allocation.
- Reduces fragmentation with compaction.

Limitations of Garbage Collection

- GC execution is non-deterministic, you can't control exactly when it runs.
- Introducing GC cycles can cause performance overhead if objects are created and discarded frequently.
- Finalizers may delay object cleanup since they run on a separate thread.