On Memory Management and Garbage Collection: Appendix

C’s Memory Model

3 Types of variables → Static, automatic, dynamic.

Static-duration variables → Allocated in memory along with the executable code of the program and persists for the entire lifetime of the program. Global/static local variables.

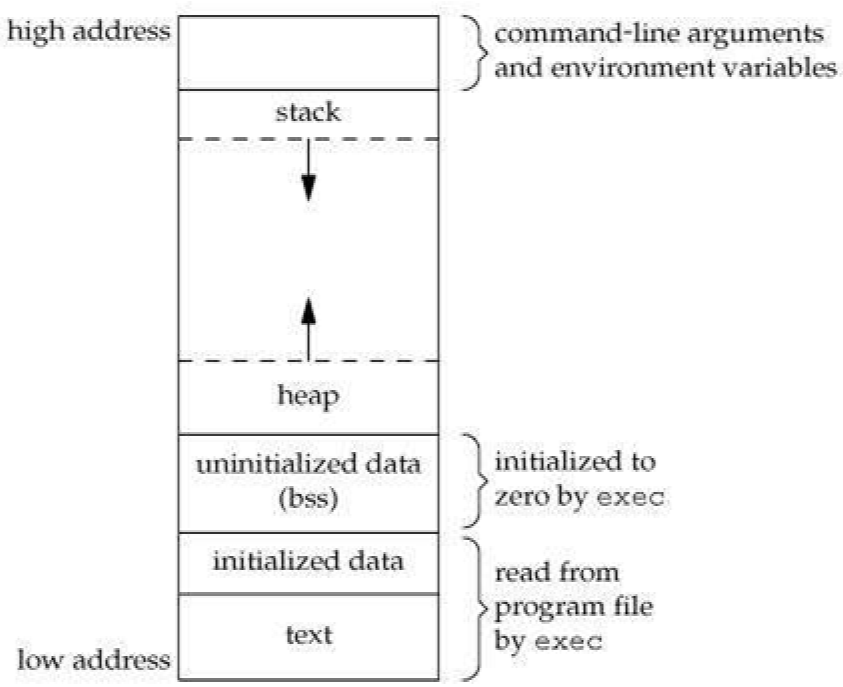
Automatic-duration variables → Local variables and function parameters, allocated on the stack and removed as functions are called and returned.

Size of allocation must be compile time constants for these 2 types*. Further, automatic-duration variables cannot persist across function calls while static-duration variables persist for the entire runtime whether they’re needed or not. Clearly, this is inadequate for all use cases and some form of dynamic management is needed.

Dynamic variables → Flexible, but have to be managed by the user using malloc, free, etc.

*except for variable-length automatic arrays

Memory Structure



Text Segment: Machine code of the compiled program, generally read-only to prevent a program from modifying its source.

Initialized Data: Global, Static, and Constant variables. Has read-only and read-write areas.
Uninitialized Data: Uninitialized global and static variables, automatically initialized to 0 before program execution.

Heap: Where dynamic memory allocation happens. Malloc, calloc, etc assign memory by growing the heap upwards.
Stack: Contains stack frames which have function local variables, arguments, and the return address of the caller.

Examples

```
global int i;
```

Uninitialized segment

```
global char s[] = “hi world”;
```

Initialized read-write segment

```
static char* string = "hello world";
```

Initialized segment. String literal "hello world" stored in the read-only area and a pointer to it stored in the read-write area.

Issues with this approach (Explicit/Manual Memory Management)

1. **Allocation Failure**: Malloc isn't guaranteed to succeed, and may return a null pointer. Using the returned pointer without checking can lead to a null pointer dereference which may or may not crash the program due to a segfault. Undefined behavior.

2. **Memory Leaks**: Not freeing up memory can lead to a build-up of unused space and can lead to allocation failures.

3. **Logical Errors**: Memory usage after calling free (dangling pointer) or before a call to malloc (wild pointer), calling free twice etc can also lead to segmentation faults. These errors are especially hard to debug as the OS may not immediately be reclaimed by the OS, causing the dangling pointer to appear to work in some cases and not in the others.

4. **Issues due to no bounds-checking**: Trying to read out of bounds (can segfault if not permitted by OS), writing out of bounds (Buffer overflow or segfault), etc.

```
int a[1] = {0};
a[1] = 0;
```

This is undefined behavior and sometimes leads to segmentation fault (When the program tries to write to a location it doesn’t have permission to), other times it doesn’t and becomes incredibly hard to debug as low reproducibility.

Memory Safety and Garbage Collection

A programming language is said to be memory-safe when it is protected from these errors and vulnerabilities associated with memory access.

Most common approach involves two components:

1. Automatic Memory Management System

2. Runtime checks on memory access

Runtime Checks → Checking for accessing valid and permitted locations, index bounds, etc.

Automatic memory management system → 2 almost independent parts
1. Allocate space for new objects (Dynamic memory allocation)
2. Reclaim space from dead objects (Garbage collection)

Garbage Collection

Want to reclaim "garbage" objects, objects that won't ever be needed. However, this "liveness" is incredibly difficult to assess and may not even be possible in some cases (eg: if the system supports hot patching). To overcome this, GCs use reachability as a proxy to liveness. An object is reachable if there is some direct or indirect chain of reachable references that reference it.

Desirable properties: a) Correctness b) Performance (latency and throughput, 0 latency is meaningless if 0% of garbage objects are collected) and c) Memory overhead

Most garbage collection schemes are based on one of 4 fundamental approaches:

Mark Sweep Collection
Mark Compact Collection
Copying Collection
Reference Counting

Mark-Sweep Collection Algorithm

Typically GC invoked when heap is exhausted and there isn't sufficient memory to satisfy an allocation request. (Alternative approach: Generational)

1. Mark Phase
Collector iterates over all roots (global/local variables, stack frames, registers, etc) and marks every object that it encounters by setting a bit. Marking can be done using BFS/DFS, but DFS is generally used with an explicit stack (and not recursion), and most real world workloads produce shallow stacks.

2. Sweep Phase
Collector iterates over all objects in the heap and reclaims memory from all the unmarked objects.

A variation of this approach called Concurrent Mark-Sweep (CMS) is one of the most popular approaches and is used by languages such as Go.
The approach is mostly the same, but uses a concept of Tri-Color Marking, which differentiates objects into 3 sets instead of 2. Objects are marked in parallel as references are created instead of all at the end, this doesn’t require the Stop-The-World pause that naive MS does.

Mark Compact

Mark step is the same, difference is that compacting step relocates marked objects to the beginning of the heap area, essentially defragmenting free space which speeds up future allocations.

Mark and Copy
Very similar to Mark Compact, except objects are copied from one region to another. This has the advantage that marking and copying can be done in parallel, at the cost of needing more reserved memory to serve as the target region.

Oracle's HotSpot JVM implementation uses a combination of multiple algorithms. The primary method is Mark-Sweep-Compact, which is Mark Sweep along with a Compacting step which relocates objects to defragment free memory. This is used as-needed and requires a STW pause. During normal execution, JVM uses a CMS alongside application processes.

Reference Counting

Idea: Counting the number of pointer references to each allocated object.
A reference count field is attached to every object and the manager maintains the invariant that at all times the reference count of an object is equal to the number of direct pointer references to that object.

Pros: Simple to implement and has low cost, the overhead is also well-distributed throughout the program. Plays well with cache (as GC doesn't need to trace all live objects), also results in better temporal locality for cache as memory can be immediately reused (unlike other GCs that have to wait for collector execution)
Cons: Can't deal with cyclic references, will leak memory

Because of its inability to deal with cyclic references leading to memory leaks, ref counting is insufficient for memory safety. Tracing garbage collection is required.
CPython uses a combination of ref counting and generational GC.

Generational Garbage Collection

Generational is essentially a heuristic to speed up tracing garbage collection.
2 key concepts:
1. Concept of generation
2. Concept of threshold

New object starts its life in the first generation of the GC. If a GC process is executed and the object survives, it moves up a generation.
The Python GC has 3 generations in total, and each generation is assigned a threshold. If the number of objects in a generation exceeds its threshold, then the GC will trigger the collection process.

Essentially a heuristic, where we assume the newer objects have a higher chance of being collected and that objects that have lived longer are likely to live longer.

Python's Tracing Garbage Collection

Steps of the algorithm explained with an example. Consider chains of references: var1 → A → B → C and D → D.

Step 1: Add all objects in the generation to an "objects to scan" list
Step 2: Create an extra gc_ref reference counting field for each object initialized to their reference count

A, B, C, and D are currently in "to scan" list with gc_refs: 1 1 1 1

Step 3: Iterate over all objects and decrement the gc_ref of all objects referenced. Essentially uncounting all internal references - now each object's gc_ref represents the number of external references.
Realize that any object with non-0 gc_ref has at least one external ref and thus should never be considered for GC. All other objects are moved to a tentative list.

gc_refs changed to: 1 0 0 0
Therefore B, C, and D are moved to tentative list

Step 4: Now perform BFSs starting at each reachable object and bring back any reachable objects from the tentative list. BFS ensures that any indirectly reachable objects are also covered.

BFS starts at A which reaches B and then C, so B and C are brought back to the first list.

Step 5: Objects in the tentative list now collected

D is remaining so gets deleted.

Why doesn’t C have this?

"In general, C++ implementations obey the zero-overhead principle: What you don’t use, you don’t pay for. And further: What you do use, you couldn’t hand code any better." ~ Stroustrup

The idea is that code cost should be easy to anticipate, and the way to do this is to have a very close relationship between the primitive operations provided by the language and the operations of the machine itself. The language shouldn't add overheads that aren't needed

Rust’s Approach

Memory Safety guarantee without needing GC.
Intuition: If you want the benefits without the runtime cost, then you probably need to do some work at compile time.

Dynamic memory allocation needed here is the same as other languages, the cleanup step is what's different.

2 major concepts: Ownership and Borrowing

Pre-req: Stack and Heap
Circle back to stack and heap. Stack faster, but size must be fixed and known at compile time.

Like C, string literals are immutable and stored on the stack (since size known), dynamic strings are mutable and allocated on the heap.

```
    let s1 = “hello”;
    let mut s2 = String::from("hello");
```

Ownership

Rules:
1. Each value in Rust has a variable that's called its owner
2. There can only be one owner at a time.
3. When the owner goes out of scope, the value is dropped

The idea is that memory is reclaimed when the variable that "owns" it goes out of scope.

```
{
    let s = "hello";
    // Do stuff
}
```

Intuitively, this is what we do in other approaches as well, we want a value to be dropped once the _variables_ that refer to it are gone.

```
    let original = String::from("hello");
    let other = original;
    println!("{}", original);  // Error!
```

Instead of a shallow copy or alias, Rust performs what's called a _move_. We say original was moved into other and original so original is now invalid.

Note: Only applies to values on the heap (size unknown). Values on the stack are copied. The following is perfectly fine:

```
    let x = 5;
    let y = x;
    println!("x = {}, y = {}", x, y);
```

Important consideration → Ownership also passed when calling functions

```
    fn takes_ownership(some_string: String) {
        println!("{}", some_string);
    }

    let s = String::from("hello");
    takes_ownership(s);  // s is no longer valid
```

Can return the value, which passes back ownership.

References and Borrowing
Instead of returning every value and changing ownership all the time (quite tedious), we can instead pass references.

```
    takes_ownership(&s);  // s is still valid
```

Since ownership doesn't change, s is still valid after function return.

Note: By default, references are immutable but we can do "&mut s" to pass and receive immutable values. Within a scope, there can be only one mutable reference. Can't have multiple mutable or mutable + immutable. Can have multiple immutable.

Where's the magic?
1. It's easy to identify the owner, and easy to check all references have been dropped before the owner (at compile time).
2. Rust is passing the buck to the programmer - your job to identify which is the last reference and pass ownership to it.

Caveat: Possible (but not easy) to create ref cycles that leak memory. Have to explicitly use weak references to prevent this from happening.

Misc.

Buffer Overflow → Stack Protector
Adds a guard variable to each stack frame and checks before return if it has been modified. An overwritten guard variable indicates a buffer overflow and the checker alerts the run-time environment.

Sources

C’s Memory Model, Common Issues:
https://en.wikipedia.org/wiki/C_dynamic_memory_allocation
C’s Memory Layout: https://www.hackerearth.com/practice/notes/memory-layout-of-c-program/

Intro to GC & Mark Sweep:
https://www.educative.io/courses/a-quick-primer-on-garbage-collection-algorithms/jy6v,
https://en.wikipedia.org/wiki/Garbage_collection_(computer_science)
JVM’s Garbage Collection Strategy: https://stackify.com/what-is-java-garbage-collection/,
https://stackoverflow.com/questions/16695874/why-does-the-jvm-full-gc-need-to-stop-the-world

Reference Counting: https://stackify.com/python-garbage-collection/

Cyclic References, Island of Isolation:
https://stackoverflow.com/questions/792831/island-of-isolation-of-garbage-collection
Python's Tracing Garbage Collection:
https://devguide.python.org/garbage_collector/#identifying-reference-cycles

Rust & C Comparison Overview:
https://www.oreilly.com/library/view/programming-rust/9781491927274/ch01.html
Rust Ownership and Borrowing:
https://doc.rust-lang.org/book/ch04-00-understanding-ownership.html
Rust Reference Cycles: https://doc.rust-lang.org/book/ch15-06-reference-cycles.html