Memory management in programming languages

C/C++

C allows for the direct memory allocation/releasing. The programmer decides whether to use static variables which size is known at compile time or manually allocate memory dynamically (for example if size of object to store is not known before the execution). Here are two variable definitions with the same meaning (creating array) but with different place in memory:

```
int array[100];

int* array = malloc(100 * sizeof(int));
```

C++ allows the same level of control but adds syntax sugar in form of **new** operator:

```
int* array = new int[100];
```

While it makes memory management more powerful it also requires a lot of attention from programmer to avoid memory leaks and free all allocated memory:

```
// C style
free(array);

// C++ style
delete array;
```

JavaScript

JavaScript allocates memory when things (objects, strings, etc.) are created and “automatically” frees it up when they are not used anymore, a process called **garbage collection**. Its job is to

track memory allocation and use in order to find when a piece of allocated memory is not needed any longer in which case, it will automatically free it.

```
var n = 374; // allocates memory for a number
var s = 'sessionstack'; // allocates memory for a string
var o = {
  a: 1,
  b: null
}; // allocates memory for an object and its contained values
```

The main concept garbage collection algorithms rely on is the one of **reference**.
Within the context of memory management, an object is said to **reference** another object if the former has an access to the latter (can be implicit or explicit). **Reference counting** is the simplest garbage collection algorithm. An object is considered “garbage collectible” if there are **zero** references pointing to it.

```
prop = o.a; // prop now referencing property 'a' of object o

o = 1; // object inside o now has zero references to it and can be
garbage-collected. However, property 'a' is still accessible from prop and
cannot be deleted

prop = null;       // now the object what was the 'a' property of the
object originally in
                   // 'o' has zero references to it.
                   // It can be garbage collected.
```

Cycles problem (objects referencing to each other) is handled by **Mark-and-Sweep** algorithm (actually graph traversal with marking each node and removing unreachable nodes).

Although it makes life easier for the programmer, garbage collecting is unpredictable, and even in this case we can have memory leaks. In JS there are four common types:

1) Global variables (creating variable without **var** keyword creates global instead of local one, deprecated behaviour and can be avoided with ‘use strict’ directive)
2) Forgotten timers or callbacks (timers have to be deleted so the objects inside, its dependencies and references to these objects from outside can be collected)
3) Closures (accessing variables from outer scope)

4) Out of DOM references (storing DOM nodes inside data structures that leads to not collecting them in case of deletion with for example **removeChild** since we still have reference to them from our data structure)

Swift

In Swift, memory management is handled by **Automatic Reference Counting** (ARC). Whenever you create a new instance of a class ARC allocates a chunk of memory to store information about the type of instance and values of stored properties of that instance. Every instance of a class also has a property called **reference count**, which keeps track of all the properties, constants, and variables that have a strong reference to itself. Whenever the **reference count** of an object reaches zero that object will be deallocated.

Again, although memory management is handled for us, we still can have memory leaks since ARC is vulnerable to **reference cycles**. You create a strong reference cycle when two instances of a class both hold strong references to each other. So both of them will be stored in memory **until application terminates**.

Java

Pretty much the same as **JavaScript** with broader types of GC. More info in references.

Go

Go memory management is automatically done by the standard library from the allocation of the memory to its collection when it is not used anymore. The memory management is designed to be fast in a concurrent environment and integrated with the garbage collector.

Example:

```
package main

type smallStruct struct {
   a, b int64
   c, d float64
}
```

```
func main() {
   smallAllocation()
}

//go:noinline
func smallAllocation() *smallStruct {
   return &smallStruct{}
}
```

//go:noinline disables in-lining that would optimize the code by removing the function and, therefore, end up with no allocation.

Go has a lot of useful tools, one of which called **escape analysis:**

```
go tool compile "-m"
```

It allows to see that **smallStruct** indeed is allocated at heap:

```
main.go:14:9: &smallStruct literal escapes to heap
```

Go has two types of allocation: **small** (less than 32kb) and **large**.

Small allocation uses a **local cache** called *mcache* and gets memory from it. This cache handles a list of span (memory chunk of 32kb) *mspan* that contains available allocation memory. It is highly optimized to be used with parallel execution (**goroutines**) since each processor has its own **local cache** so using it does not require lock and makes allocation more efficient.

Large allocations are rounded up to the page size and pages are allocated directly to the heap.

The memory allocator is originally based on TCMalloc, a memory allocator optimized for the concurrent environment created by Google.

References

https://www.tutorialspoint.com/cprogramming/c_memory_management.htm

https://blog.sessionstack.com/how-javascript-works-memory-management-how-to-handle-4-common-memory-leaks-3f28b94cfbec
https://medium.com/elements/memory-management-in-swift-31e20f942bbc
https://www.javatpoint.com/memory-management-in-java
https://medium.com/a-journey-with-go/go-memory-management-and-allocation-a7396d430f44