# GC Vs. ARC

In the early days of programming, programmers were faced with many memory problems with their computers: slow processing speed with little memory space to work with. When an object is created, a variable then is assigned to it. Upon termination, only the variable is destroyed, this means only a reference to the object is killed, but the actual object still lives in memory. Roughly describing, if there happens to be too many of these rogue objects in memory, the computer will be resource starved, and will be unable to perform any more instructions.

Programmers came up with ways to solve this problem, and the most common way is the manual memory management(MMM): to keep track of each object and release them at the right time. (This is also what we have been doing with our objects.)

Nowadays, we have automatic memory management(AMM) in modern language. The two most commonly used is the Garbage Collection(GC) and the Automatic Reference Counting(ARC). While GC is used by languages such as Java, JavaScript, C# and etc., ARC is used by Objective C, Swift, C++ smart pointers, and etc.

GC is used at runtime in the background, which detects inactive objects and removes them from memory. Object is considered inactive when it is unreachable, or simply put, there are no reference to the object. ARC on the other hand, is deployed automatically as a compile mechanism at compile-time; ARC injects memory management calls such as retain and release on objects as reference points fluctuate. A reference count indicates when an object can be destroyed: as more variables is referenced to the object, the reference count goes up; as all the variables are being nullified, the reference count goes down, and finally when the reference count reaches zero, the object is deallocated.

These two methods both have their strengths and weaknesses:

| | Strengths | Weaknesses |
|---|---|---|
| ARC | + Faster than GC<br>+ reclaims memory immediately | - unable to handle retained cycles(circular object referencing) |
| GC | + Don’t need to worry about adding “free” or “delete” statements to code<br>+ Never have bugs related to dangling references/pointers (double frees, etc.) | - Random(indeterminate) deallocation<br>- Cannot implement own class destructor<br>- Takes a huge chunk of memory(RAM) for the CG algorithm(responsible for “awkward pauses”) |

Performance wise, there is no clear winner. They both have their own advantages and disadvantages.