2.6 GARBAGE COLLECTION

In general layman's terms, Garbage collection (GC) is nothing but collecting or gaining memory back which has been allocated to objects but which is not currently in use in any part of our program. Let's get into more detail. Garbage collection is the process in which programs try to free up memory space that is no longer used by objects.

Garbage collection is implemented differently for every language. Most high-level programming languages have some sort of garbage collection built in. Low-level programming languages may add garbage collection through libraries.

As said above, every programming language has their own way of performing GC. In C programming, developers need to take care of memory allocation and deallocation using malloc() and dealloc() functions. But, in the case of C# developers don't need to take care of GC and it’s not recommended either.

How does memory allocation happen?

In C#, memory allocation of objects happens in a managed heap, which is taken care of by CLR (common language runtime). Memory allocation for the heap is done through win32 dll in OS and similarly in C.

But, in C objects are placed in memory wherever there is free space that fits the size of the object. Also, memory mapping works based on Linkedlist concepts. In C#, memory allocation for the heap happens in a linear manner, one after another.

Whenever a new object is being created, memory is allocated in the heap and the pointer is moved to the next memory address. Memory allocation in C# is faster than in C. This is because in C the memory needs to search and allocate for the object. So it will take a bit more time than C#.

Generations in C# GC

In .net programming, the heap has three generations called generations 0, 1, and 2. Generation 0 gets filled first whenever a new object is created. Then the garbage collector runs when Generation 0 gets filled. Newly created objects are placed in Generation 0.

While performing garbage collection all the unwanted objects are destroyed, and so memory gets freed and compacted. GC takes care of pointing the pointers of freed memory once GC happens.

Generations 1 and 2 contain objects which have longer lifetimes. GC on generations 1 and 2 will not happen until generations 0 has sufficient memory to allocate.

You shouldn't invoke GC programmatically. It’s good to let it happen on its own. GC gets call whenever generation 0 gets filled.

Pros and Cons of GC

Garbage collection is a tool that saves time for programmers. For example it replaces the need for functions such as malloc() and free() which are found in C. It can also help in preventing memory leaks.

The downside of garbage collection is that it has a negative impact on performance. GC has to regularly run though the program, checking object references and cleaning out memory. This takes up resources and often requires the program to pause.

When to do it

If an object has no references (is no longer reachable) then it is eligible for garbage collection.

For example in the Java code below, the Thing object originally referenced by ‘thing1’ has its one and only reference redirected to another object on the heap. This means it is then unreachable and will have its memory unallocated by the garbage collector.

```
class Useless {
  public static void main (String[] args) {
  Thing thing1 = new Thing();
  Thing thing2 = new Thing();
  thing2 = thing1; // direct thing2's reference towards thing1
```

```
        // no references access thing2
} }
```

One example of garbage collection is ARC, short for automatic reference counting. This is used in Swift, for example. ARC boils down to keeping track of the references to all objects that are created. If the amount of references drops to 0, the object will be marked for deallocation.