# Phases

## Marking Phase

Collector checks roots into Thread’s stack. If root found it turns on a bit on the object’s sync block index field. This is called marking. Object is marked referenced and collector will not clean or dereference it.

Example: lets there are objects in heap from A, B, C till J.

Let’s assume that A, B, C, D and F object has roots available. Here D object contains a field that refers object H.

Now when collector goes through each object’s stack and checks all roots available for other objects. Collector marks each and every object of which root is found.

So, A, B, C, D, F and H are the object which collector has marked.

The garbage collector continues to walk through all reachable objects recursively. After a root and the objects referenced by its fields are marked, garbage collector checks the next root and continues marking objects/ if the garbage collector is going to mark an object that it previously marked, it can stop walking down that path.

This behavior serves two purposes:

1. Performance: the performance is enhanced significantly because the garbage collector doesn’t walk through a set of objects more than once.
2. Infinite loops are prevented if you have any circular linked lists of objects.

Once all of the roots have been checked, the heap contains a set of marked and unmarked objects. The marked objects are reachable via the application’s code but the unmarked objects are unreachable.

The unreachable objects are considered garbage, and the memory that they occupy can be reclaimed.

## Compact Phase

The garbage collector now starts what is called the compact phase of the collection. This is when the collector traverses the heap linearly looking for contiguous blocks of unmarked (garbage) objects. It works in following steps:

1. If small block is found, the garbage collector leaves the blocks alone.
2. If large free contiguous blocks are found, the garbage collector shifts the non-garbage objects down in memory to compact the heap.
3. Moving object in memory invalidates all variables and CPU registers that contain pointers to the objects. So the garbage collector must revisit all of the application’s roots and modify them so that each root’s value points to the object’s new memory location.
4. In addition, if any object contains a field that refers to another moved object, the garbage collector is responsible for correcting there fields as well.
5. After the heap memory is compacted, the managed heap’s “NextObjPtr” pointer is set to point to a location just after the last non-garbage object.

# Why there is no garbage collector in ANSI C++?

The reason is that a garbage collector must be able to identify an application’s roots and must also be able to find all object pointers. The problem with unmanaged C++ is that it allows casting a pointer from one type to another, and there’s no way to know what a pointer refers to. In the CLR, the managed heap always knows the actual type of an object and uses the metadata information to determine which members of an object refer to other objects.

# What is finalization in .Net C#?

Finalization is to release native resources.

Finalization is a mechanism offered by the CLR that allows an object to perform some graceful cleanup prior to the garbage collector reclaiming the object’s memory. Any type that wraps a native resource, such as file, network connection, socket, mutex or other type, must support finalization. Basically, the type implements a method named **Finalize**.

When the garbage collector determines that an object is garbage, it calls the object’s finalize method (if exists). In other words, any type that implements the **Finalize** method is in effect stating that all of its object want a last meal before they are killed.