Reference Counting & Garbage Collection
**Reference counting** is a form of automatic memory management where each object has a count of the number of references to it. An object's reference count is incremented when a reference to it is created, and decremented when a reference is destroyed. The object's memory is reclaimed when the count reaches zero.

There are two major disadvantages to reference counting:

- If two or more objects refer to each other, they can create a cycle whereby neither will be collected as their mutual references never let their reference counts become zero. Some garbage collection systems (like the one in CPython) using reference counting use specific cycle-detecting algorithms to deal with this issue.
- In naïve implementations, each assignment of a reference and each reference falling out of scope often require modifications of one or more reference counters. However, optimizations to this are described in the literature. When used in a multithreaded environment, these modifications (increment and decrement) may need to be interlocked. This may be an expensive operation for processors without atomic operations such as Compare-and-swap.

One important advantage of reference counting is that it provides deterministic garbage collection (as opposed to tracing GC).

In computer science, **garbage collection (GC)** is a form of automatic memory management. The **garbage collector**, or just **collector**, attempts to reclaim **garbage**, or memory used by objects that are no longer in use by the application. Garbage collection was invented by John McCarthy around 1959 to solve the problems of manual memory management in lisp.

Garbage collection is often portrayed as the opposite of manual memory management, which requires the programmer to specify which objects to deallocate and return to the memory system. However, many systems use a combination of the two approaches, and there are other techniques being studied (such as region inference) to solve the same fundamental problem.

**Description**

The basic principles of garbage collection are:

1. Find data objects in a program that cannot be accessed in the future
2. Reclaim the resources used by those objects

By making manual memory deallocation unnecessary (and often forbidding it), garbage collection frees the programmer from having to worry about releasing objects that are no longer needed, which can otherwise consume a significant amount of design effort. It also aids programmers in their efforts to make programs more stable, because it prevents
several classes of runtime errors. For example, it prevents dangling pointer errors, where a reference to a deallocated object is used. (The pointer still points to the location in memory where the object or data was, even though the object or data has since been deleted and the memory may now be used for other purposes, creating a dangling pointer. This can, and often does, lead to storage violation errors that are extremely difficult to detect.)

Many computer languages require garbage collection, either as part of the language specification (e.g., Java, C#, and most scripting languages) or effectively for practical implementation (e.g., formal languages like lambda calculus); these are said to be garbage collected languages. Other languages were designed for use with manual memory management, but have garbage collected implementations available (e.g., C, C++). Some languages, like Ada and Modula-3, allow both garbage collection and manual memory management to co-exist in the same application by using separate heaps for collected and manually managed objects; others, like D, are garbage collected but allow the user to manually delete objects and also entirely disable garbage collection when speed is required. In any case, it is far easier to implement garbage collection as part of the language's compiler and runtime system, but post hoc GC systems exist, including ones that do not require recompilation. The garbage collector will almost always be closely integrated with the memory allocator.

**Benefits**

Garbage collection frees the programmer from manually dealing with memory allocation and deallocation. As a result, certain categories of bugs are eliminated or substantially reduced:

- **Dangling pointer bugs**, which occur when a piece of memory is freed while there are still pointers to it, and one of those pointers is used.
- **Double free bugs**, which occur when the program attempts to free a region of memory that is already free.
- Certain kinds of **memory leaks**, in which a program fails to free memory that is no longer referenced by any variable, leading, over time, to memory exhaustion.

Researchers draw a distinction between "physical" and "logical" memory leaks. In a physical memory leak, the last pointer to a region of allocated memory is removed, but the memory is not freed. In a logical memory leak, a region of memory is still referenced by a pointer, but is never actually used. Garbage collectors generally can do nothing about logical memory leaks. Novice programmers sometimes believe that garbage collection makes memory leaks impossible, not realizing that logical leaks are still possible.

In languages that provide dynamic allocation, garbage collection is crucial to memory safety and often to the associated property of type safety. This in turn improves the security of the language by preventing a wide class of security vulnerabilities based on over-writing memory in unexpected ways.
Reference Counting Garbage Collection

The difficulty in garbage collection is not the actual process of collecting the garbage—it is the problem of finding the garbage in the first place. An object is considered to be garbage when no references to that object exist. But how can we tell when no references to an object exist?

A simple expedient is to keep track in each object of the total number of references to that object. That is, we add a special field to each object called a **reference count**. The idea is that the reference count field is not accessible to the Java program. Instead, the reference count field is updated by the Java virtual machine itself.

Consider the statement

```
Object p = new Integer (57);
```

which creates a new instance of the `Integer` class. Only a single variable, `p`, refers to the object. Thus, its reference count should be one.

```
Figure: Objects with reference counters.
```

Now consider the following sequence of statements:

```
Object p = new Integer (57);
Object q = p;
```

This sequence creates a single `Integer` instance. Both `p` and `q` refer to the same object. Therefore, its reference count should be two.

In general, every time one reference variable is assigned to another, it may be necessary to update several reference counts. Suppose `p` and `q` are both reference variables. The assignment

```
p = q;
```

would be implemented by the Java virtual machine as follows:

```
if (p != q) {
    if (p != null)  --p.refCount;
    p = q;
    if (p != null)  ++p.refCount;
}
```
For example suppose \( p \) and \( q \) are initialized as follows:

```
Object p = new Integer (57);
Object q = new Integer (99);
```

As shown in Figure 1(a), two \texttt{Integer} objects are created, each with a reference count of one. Now, suppose we assign \( q \) to \( p \) using the code sequence given above. Figure 1(b) shows that after the assignment, both \( p \) and \( q \) refer to the same object--its reference count is two. And the reference count on \texttt{Integer}(57) has gone to zero which indicates that it is garbage.

![Reference counts before and after the assignment \( p = q \).](image)

The costs of using reference counts are twofold: First, every object requires the special reference count field. Typically, this means an extra word of storage must be allocated in each object. Second, every time one reference is assigned to another, the reference counts must be adjusted as above. This increases significantly the time taken by assignment statements.

The advantage of using reference counts is that garbage is easily identified. When it becomes necessary to reclaim the storage from unused objects, the garbage collector needs only to examine the reference count fields of all the objects that have been created by the program. If the reference count is zero, the object is garbage.

It is not necessary to wait until there is insufficient memory before initiating the garbage collection process. We can reclaim memory used by an object immediately when its reference goes to zero. Consider what happens if we implement the Java assignment \( p = q \) in the Java virtual machine as follows:

```
if (p != q) {
    if (p != null)
        if (--p.refCount == 0)
            heap.release (p);
    p = q;
    if (p != null)
        ++p.refCount;
}
```

**BCA NOTES** For more details [mail me](mailto:) or follow this link [www.bcaraiupur.blogspot.in](http://www.bcaraiupur.blogspot.in)
Notice that the `release` method is invoked immediately when the reference count of an object goes to zero, i.e., when it becomes garbage. In this way, garbage may be collected incrementally as it is created.