Garbage Collection

When we know that a block of allocated memory is no longer needed, it is easy to free it up by marking it as unused in the Heap Table. The trick is discovering when it is no longer needed.

There are two standard ways of identifying unneeded memory:

- Relying on the user to indicate the lack of need, using a construct such as C's free() function.
- Having a system running in the background that automatically identifies memory that is no longer addressable and hence can be deallocated. This is called a *garbage collection* system. Object-oriented languages and functional languages both tend to use garbage collection.

There are several different approaches to garbage collection. We will discuss 3:

- Reference counts
- Mark and Sweep
- Copy Collectors

Reference Counts

With each chunk of allocated memory keep a count of how many references the program has to locations within that chunk. This count is usually maintained in the Heap Table.

For example, consider an object-oriented language. Objects are allocated on the heap, so if you construct a new object of class Foobar with

```
Foobar p = new Foobar()
```

the value of p will be an address on the heap. The result of this line is that the chunk of memory containing this address gets a reference count of 1. If we then say

```
Foobar q = p
```

we increment the reference count to 2.

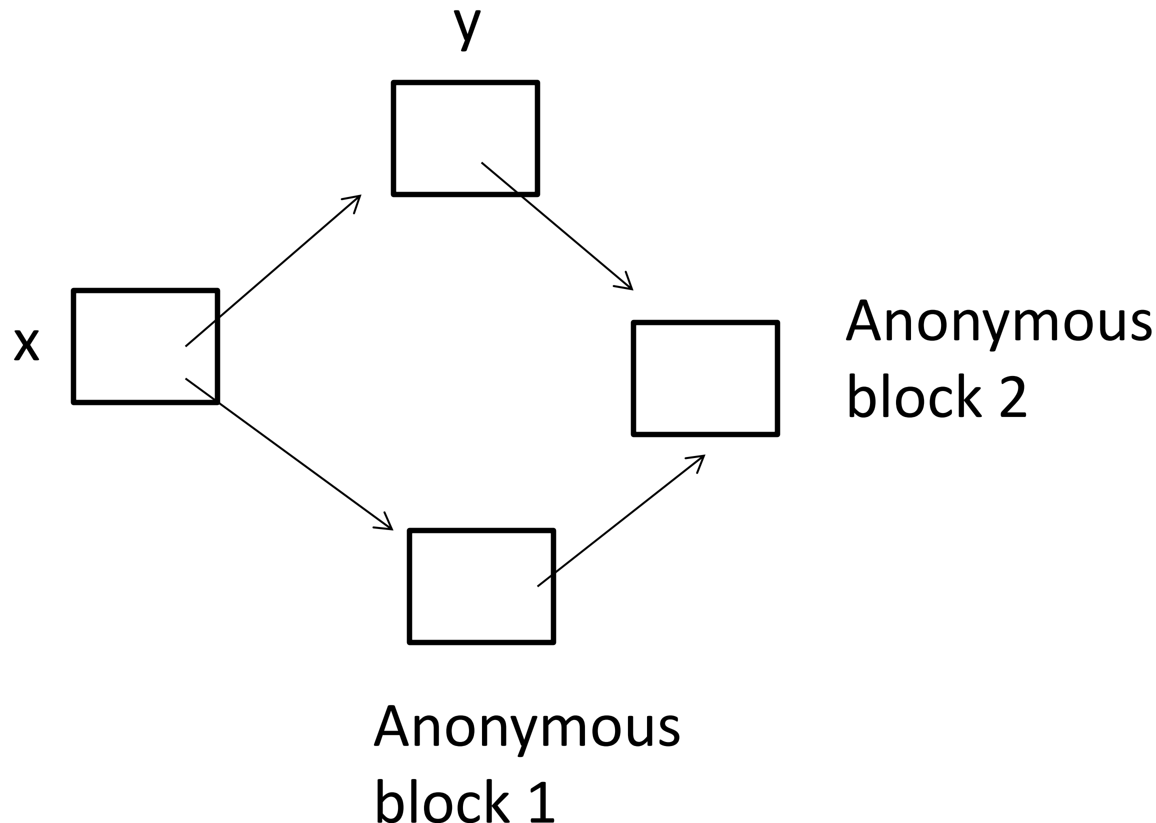
If this happens within function fun( ) which returns p and we call the function with

    Foobar x = fun( )

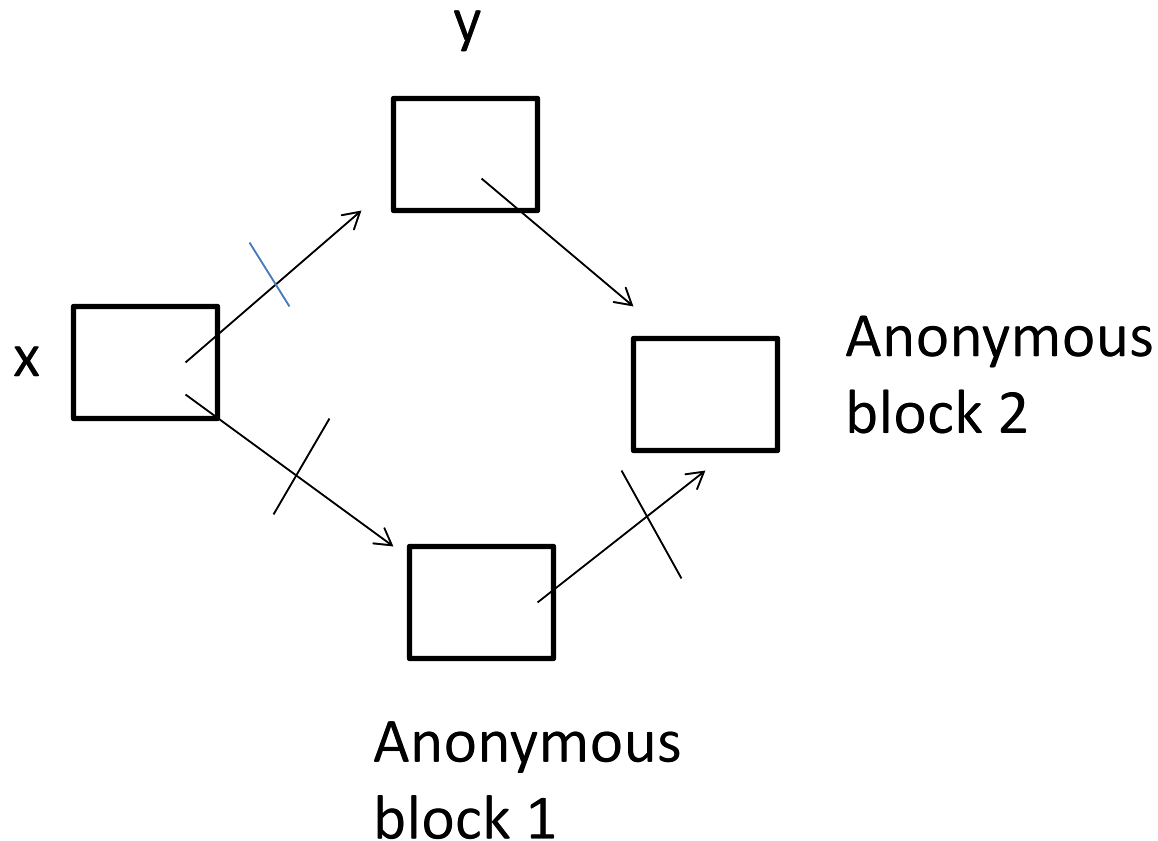
then the reference count is increased by 1 for this assignment and decremented by 2 since p and q are both deallocated when we return from fun( ).

When a reference count for a chunk of memory reaches 0, the chunk can be deallocated. This means that anything pointed to by the chunk has its reference count decremented.

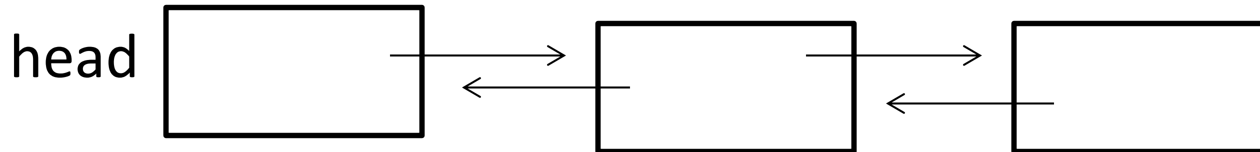
For example, in the following picture x and y are variables holding pointers to their locations, which point to other locations. The reference counts are x:1, y:2, block1: 1, and block2: 2



If x is dealloated the reference counts become x:0, y:1, block1: 0 and block2: 1. Both x and block 1 will be garbage collected.



Note that reference counts have a problem with cyclic list structures:



The reference count for the first block is 2, so even if we deallocate the *head* variable we can't garbage collect any of the blocks.

The standard release of Python is said to run a cycle detection algorithm and so it can garbage-collect cyclic structures not referenced externally.

Mark & Sweep

Reference counting is an incremental garbage collection method -- you can collect unneeded memory at any point after it becomes unneeded.

Mark & Sweep is a batch approach. When something needs to be done the execution of the program is halted, memory is reclaimed, and execution resumes.

The idea of Mark & Sweep is that we make a pass through the Heap Table, marking any chunk into which there is a live reference. We then make a second pass, deallocating any unmarked memory.

This is expensive, so it is not usually done until the heap is almost exhausted.

The difficulty with Mark & Sweep is the marking algorithm. This usually starts with a live variable analysis -- which variables are currently live. Any memory pointed to by live variables is marked, and recursively anything that it points to is also marked.

Advantages of Mark & Sweep:

- Allocation of the heap is fast, since there are no reference counts to worry about.
- Actual deallocation is fast since there are no pointers to follow.
- Assignment is fast, since there are no reference counts to analyze.
- Cyclic data structures are not a problem since any structure that cannot be reached by some live variable will be garbage collected.
- Most programs don't need garbage collection, and will run faster under Mark & Sweep.

Disadvantages of Mark & Sweep:

- The program has to halt while garbage collection takes place.

Copy Collectors

These divide the heap into two halves -- the *from-space* and the *to-space*. Initially all allocations are from the from-space. When this is nearly full a Mark&Sweep pass is executed and the live parts of the heap are copied to the to-space. The roles of the from-space and to-space are then reversed.'

In other words, Copy Collectors combine Mark&Sweep with a defragmentation algorithm.

The big cost of copy collecting over other methods is that it reduces the effective heap size by a factor of 2. But then, memory is cheap, right?

Note that the garbage is never "collected"; it just isn't copied.

An object that is created and then is deallocated before the marking pass is handled for free.

Copy collectors handle ephemeral data well. They are much less efficient for long-lived data.