# Reference Counts

In languages like C or C++, the programmer is responsible for dynamic allocation and deallocation of memory on the heap.

In C, this is done using the functions malloc() and free().

In C++, the operators new and delete are used with essentially the same meaning; they are actually implemented using malloc() and free().

Every block of memory allocated with malloc() should eventually be returned to the pool of available memory by exactly one call to free().

It is important to call free() at the right time.

If a block's address is forgotten but free() is not called for it, the memory it occupies cannot be reused until the program terminates. This is called a *memory leak*.

On the other hand, if a program calls free() for a block and then continues to use the block, it creates a conflict with re-use of the block through another malloc() call. This is called *using freed memory*.

It has the same bad consequences as referencing uninitialized data -- core dumps, wrong results, mysterious crashes.

Common causes of memory leaks are unusual paths through the code.

For instance, a function may allocate a block of memory, do some calculation, and then free the block again. Now a change in the requirements for the function may add a test to the calculation that detects an error condition and can return prematurely from the function. It's easy to forget to free the allocated memory block when taking this premature exit, especially when it is added later to the code. Such leaks, once introduced, often go undetected for a long time: the error exit is taken only in a small fraction of all calls, and most modern machines have plenty of virtual memory, so the leak only becomes apparent in a long-running process that uses the leaking function frequently. Therefore, it's important to prevent leaks from happening by having a coding convention or strategy that minimizes this kind of errors.

Since Python makes heavy use of malloc() and free(), it needs a strategy to avoid memory leaks as well as the use of freed memory. The chosen method is called *reference counting*.

The principle is simple: every object contains a counter, which is incremented when a reference to the object is stored somewhere, and which is decremented when a reference to it is deleted. When the counter reaches zero, the last reference to the object has been deleted and the object is freed.

An alternative strategy is called *automatic garbage collection*. (Sometimes, reference counting is also referred to as a garbage collection strategy, hence my use of ``automatic'' to distinguish the two.)

The big advantage of automatic garbage collection is that the user doesn't need to call free() explicitly. (Another claimed advantage is an improvement in speed or memory usage -- this is no hard fact however.)

The disadvantage is that for C, there is no truly portable automatic garbage collector, while reference counting can be implemented portably (as long as the functions malloc() and free() are available -- which the C Standard guarantees). Maybe some day a sufficiently portable automatic garbage collector will be available for C. Until then, we'll have to live with reference counts.