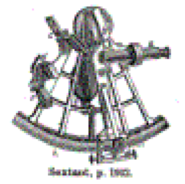

Conman Laboratories

Better living through software ...

Thinking about Memory Management

Mark Grosberg

A common question I get asked is what to do about storage management, in particular reclaiming memory from used objects. There are so many solutions, each with their own quirks, that its hard to decide what to do in a large and complex project. Some of the techniques are:

- Manual storage management
- Lifetime pools
- Reference counting
- “Mark and sweep” garbage collection
- “Stop and copy” garbage collection

The most common technique, manual storage management, is also the most limited and difficult to implement. The only advantage of this technique is for small software there is no code space wasted for any kind of management code. This is only true until the complexity of all the error paths results in more code in calls to free the storage than an automatic method would have wasted. Most programmers also feel that this approach is the most efficient (it isn't).

For any kind of complex software storage management in the face of errors and complex structures really becomes too difficult and can cause major maintenance headaches. Some types of software (especially request-oriented systems) can divide allocations into “pools” of storage. In fact, the Apache webserver uses this approach. In fact, pools can be allocated inside of other pools forming a hierarchy of storage.

This results in a very efficient way of maintaining storage. A single operation can free an many allocations in a single operation. The downside to this approach is that it requires great programmer discipline to make sure pools get destroyed when appropriate and that objects are allocated from pools with the correct lifetime.

Reference counting is a common approach in C++ programs because operator overloading allows it to be implemented easily (but not efficiently). Each object holds a counter that indicates how many other objects depend on this ones existence. When the counter is decremented to zero, the object can be freed. Programs that make use of a generic data structure libraries can implement the majority of the reference counting logic in common code.

Reference counting and pooled allocation are not necessarily mutually exclusive. Pools can be used for allocations where the reference counting model doesn't fit or the cost is too high. In these cases the pool object can be reference counted. When no more objects are using the pool, its entire contents can be eliminated.

Another approach to combining reference counting and pools is to use pools as a mechanism to correct object counts in the face of errors. In this case the pool doesn't handle the low level aspects of storage allocation but instead functions more like a container for objects. When an error happens the pool will de-reference all of the temporary objects. Unless they are referenced by longer-lived data structures they will all be reclaimed.

This approach is quite powerful but has a flaw. Reference counting only works for non-circular data structures. As soon as data structures have a cycle (two nodes depending on each other) it becomes nearly impossible to release the storage of the nodes, efficiently. It is possible to mark the state of an object and trace pointers to determine cycles during a reference decrement, but it is very inefficient to do on every reference.

Garbage collection uses the idea of tracing pointers to determine live objects from the above reference counting problem. However, it does not do this every time an object is no longer used by another. Rather, the system runs until it is too low on free memory to continue. When the system needs to have storage reclaimed the main task of the software is stopped and all of the active objects are traversed. Any unreachable objects are discarded.

There are two main strategies for garbage collection. The first, mark and sweep, works the obvious way. Each object posses a one bit flag. During garbage collection each object that is reachable is marked. Any objects that are not marked after all the reachable objects are traversed are unreachable and therefore garbage. The sweep phase frees each node.

Mark and sweep has a problem in that it requires two passes over all the active objects in the system. Furthermore it allows the heap to become very fragmented. Another approach to garbage collection, stop and copy, does not have some of these problems (but if suffers from others).

In stop and copy garbage collection the free space is divided exactly in half. A global flag indicates which half of the free space is used for allocations. When the current free space is exhausted the system must collect garbage. Each object is traveresed and copied into the second half of the free space. Any unreachable object is not copied during the traversal. The end result is that the reachable objects are transferred to the other half of the heap. After each cycle the meaning of the halves is swapped.

The neat thing about stop and copy garbage collection is that as a side effect of collecting garbage the heap is also compacted (eliminating fragmentation). This means that allocations become very efficient because there is no free list to be searched. The down side is that only half of available memory can be used at any one time.

Many programmers are quick to believe that garbage collection is a panacea that solves all their allocation difficulties. This couldn't be further from the truth. Garbage collection has several disadvantages. The biggest disadvantage is that garbage collection disturbs “locality of reference.” Techniques like caches and virtual memory depend on the fact that not all data is accessed with the same frequency. During a garbage collection cycle the majority of memory is not only examined but changed.

The next major problem with garbage collection is that it makes the response time of a software system very unpredictable. For real-time systems or user interfaces this is not acceptable. In fact, in the case of real-time systems it can result in the failure of the system to meet a deadline.

A commonly stated problem of garbage collection is the so called “finalization problem.” When an object is to be destroyed it is often necessary to perform some action. For example, an object may contain an operating system resource or represent some hardware. This means that cleanup must be accompanied by a cleanup routine (called destructor in C++). This is very difficult because the cleanup routine can change the pointers while the garbage collector is in the midst of traversing them.

One of the most obvious, but often ignored, problems is the need to be able to trace all the pointers in the system. This isn't as easy as it first appears because garbage collection can happen at any time (in the case of multitasking). This means that it is not sufficient to trace just the pointers in global variables (which can be easily identified). Rather, pointers on the stack(s) must also be identified. What if, for example, an object is pointed to only by a local variable as it is being moved (no globally reachable pointers point to the object)? The object could disappear while it is still being used.

Another problem with garbage collection is that it makes programmers to complacent about memory allocation. Garbage collection makes programmers believe that it is impossible for their code to have memory leaks. Of course if code forgets to release a reference to an object, that object will leak. I have seen this very bug in a Java application that a co-worker was developing. The programmer kept adding an object that pointed to a huge tree structure (around a megabyte) to a list. The object in question was a small widget that kept getting added to a focus list. Of course, that meant the structure was never freed.

The picture is not as bleak as I make it out to be. Newer garbage collection algorithms can reduce the locality of reference problem. These newer algorithms (called “generational garbage collection”) even allow garbage collection to be used in some (soft) real-time systems. Most actual implementation of garbage collection are “conservative.” This means that they make a best guess, erring on the side of caution, about what values are pointers. Meaning that some garbage can end up never being collected. In systems demanding high uptimes this is clearly unacceptable.

So what can be done about circular data structures? Thankfully circular data structures are very rare. In fact, a lot of cases, all of the nodes that may contain circular pointers can be grouped into a single data structure where the nodes are managed internally using garbage collection. Compartmental garbage collection like this is more efficient than garbage collection of the entire system. It is also more easily implemented.

The most important point to all this is that there is no one memory management solution that can meet all the requirements of a complex software system. What is evident is that software shouldn't rely on a completely manual approach, it is tedious and easy to make mistakes. Fully automatic systems often have performance problems and can't handle complex object cleanups.

A complex system has many data structures, some very specific to the application and others being generic (linked lists, trees, hash tables, etc). Different techniques are applicable to different data structures. Memory management should be built into the foundation of the system as much as possible. Codifying the memory management logic into generic libraries of techniques as much as possible.