University of Glasgow | School of Computing Science

Garbage Collection (1)

Advanced Operating Systems
Lecture 8

Lecture Outline

- Introduction
- Reference counting
- Garbage collection
  - Mark-sweep
  - Mark-compact
  - Copying collectors
  - …

Introduction

- Wide distrust of automatic memory management in real-time, embedded, and systems programming
  - Perception of high processor and memory overheads, unpredictable poor timing behaviour
  - But, memory management problems are common in code with manual memory management!
    - Memory leaks and unpredictable memory allocation performance (calls to `malloc()` can vary in execution time by several orders of magnitude)
    - Memory corruption and buffer overflows
- Performance of automatic memory management is much better than in the past
  - Not all problems solved, but there *are* garbage collectors with predictable timing, suitable for real-time applications
  - Moore's law makes the overheads more acceptable

Automatic Memory Management

- Memory/object allocation and deallocation may be manual or automatic
  - Automatic allocation/deallocation of variables on the stack is common
    - In the example code, memory for `di` is automatically allocated when the function executes, and freed when it completes
    - Extremely simple and efficient memory management for languages like C/C++ that have complex value types
    - Useless for Java-like languages, where objects are allocated on the heap

```
int saveDataForKey(char *key, FILE *outf)
{
   struct DataItem di;

   if (findData(&di, key)) {
       saveData(&di, outf);
       return 1;
   }
   return 0;
}
```

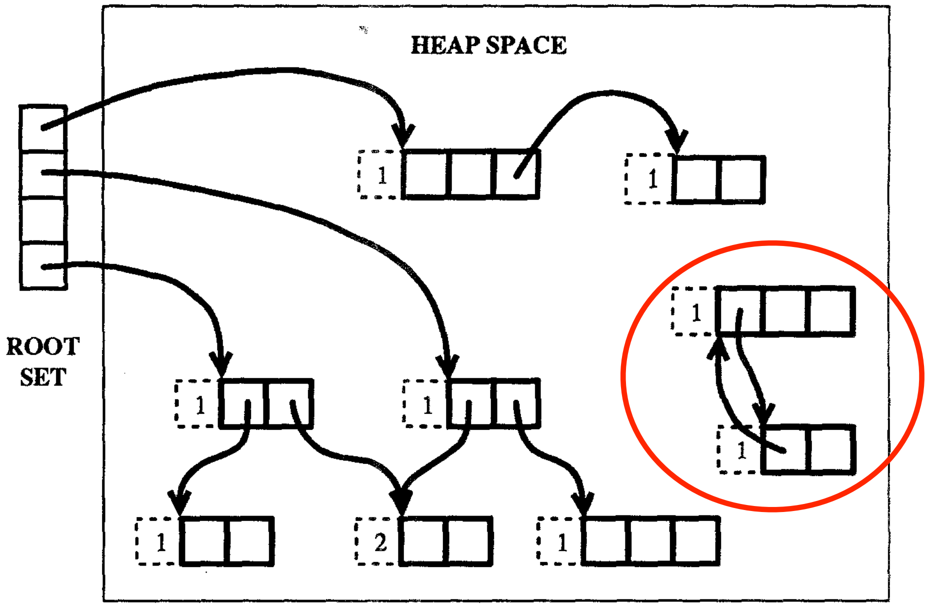
  - Memory allocated on the heap is allocated explicitly (e.g., using `malloc`)
  - Heap memory may be explicitly freed, or automatically reclaimed when no longer referenced
    - Automatic reclamation doesn't remove the need to manage object life-cycles, and doesn't prevent memory leaks

Automatic Heap Management

- Aim is to find objects that are no longer used, and make their space available for reuse
  - An object is no longer used (ready for reclamation) if it is not reachable by the running program via any path of pointer traversals
  - Any object that is *potentially* reachable is preserved – is better to waste memory if unsure about reachability, than to deallocate an object that is used, leading to a dangling pointer and later program crash

Reference Counting

- Simple automatic heap management scheme
  - Each object is augmented with a count of the number of references to that object
  - Incremented each time a reference to the object is created; decremented when a reference is destroyed
  - When the count reaches zero, there are no references to the object, and it may be reclaimed
  - Reclaiming an object may remove references to other objects, causing their count to become zero, triggering further reclamation
- Incremental operation: collection occurs in many small bursts
- Cycles are problematic and must be explicitly broken by the programmer
- Per-object overhead to store reference count is inefficient if many small objects are used
- Short-lived objects: high processor overhead, due to cost of managing reference counts



Source: P. Wilson, "Uniprocessor garbage collection techniques", Proc IWMM'92, DOI 10.1007/BFb0017182

Widely used by scripting languages (e.g., Perl and Python), and in the MacOS X Objective-C runtime

Garbage Collection

- Avoid problems of reference counting via *tracing algorithms*
  - Explicitly trace through the allocated objects, recording which are in use, rather than continually maintaining reference counts; dispose of unused objects
  - This moves *garbage collection* to be a separate phase of the program's execution, rather than an integrated part of an objects lifecycle
    - A garbage collector runs and *disposes of* objects
    - An object is reclaimed when *its* reference count becomes zero

- Many tracing garbage collection algorithms exist:
  - Mark-sweep, mark-compact, copying
  - Generational algorithms

Mark-Sweep Collectors

- Simplest automatic garbage collection scheme
- Two phase algorithm
  - Distinguish live objects from garbage (*mark*)
  - Reclaim the garbage (*sweep*)

- Non-incremental algorithm: program is paused to perform collection when memory becomes tight

- Will collect all garbage, whether or not there are cycles

Distinguishing Live Objects

- Find the *root set* of objects
  - Global and stack variables
- Traverse the object relationship graph staring at the root set to find all other reachable, live, objects
  - Breadth-first or depth-first search
  - Must read every pointer in every object in the system to systematically find all reachable objects
- Mark reachable objects
  - Stop traversal at previously seen objects to avoid following cycles
  - Either set a bit in the object header, or in some separate table of live objects

Reclaiming the Garbage

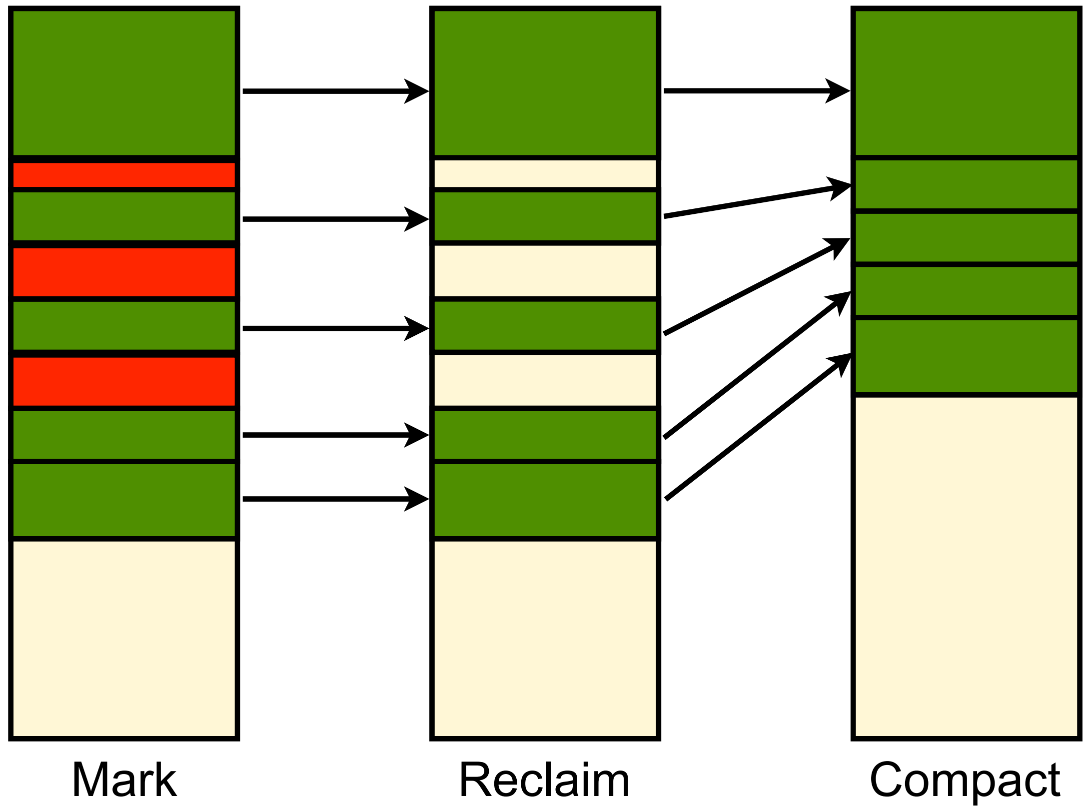
- Sweep through the entire heap, examining every object for liveness in turn
  - If marked as alive, keep it, otherwise reclaim the object's space
  - Space occupied by reclaimed objects is marked as free: the system must maintain one or more free lists to track available space
  - New objects are allocated in the space previously reclaimed
- No problem with collecting cycles, since the mark phase will not reach unreferenced cycles

Problems with Mark-Sweep Collectors

- Cost proportional to size of heap
  - Program is stopped with the collector runs; unpredictable collection time
  - All live objects must be marked, and all garbage must be reclaimed
  - Unlike reference counting, mark-sweep garbage collection is slower if the program has lots of memory allocated
- Fragmentation
  - Since objects are not moved, space may become fragmented, making it difficult to allocate large objects (even though space available overall)
- Locality of reference
  - Passing through the entire heap in unpredictable order disrupts operation of cache and virtual memory subsystem
  - Objects located where they fit (due to fragmentation), rather than where it makes sense from a locality of reference viewpoint

Mark-Compact Collectors

- Traverse the object graph, and mark live objects
- Reclaim unreachable objects, then *compact* the live objects, moving them to leave a single contiguous free space
  - Reclaiming and compacting memory can be done in a single pass, but still touches the entire address space



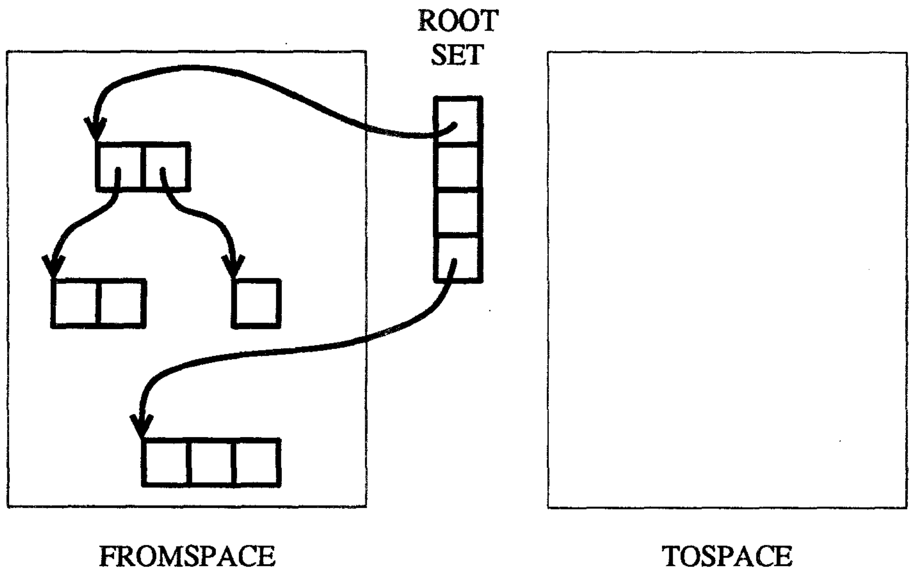
- Advantages:
  - Solves fragmentation problems
  - Allocation is very quick (increment pointer to next free space, return previous value)
- Disadvantages:
  - Collection is slow, due to moving objects in memory, and time taken is unpredictable
  - Collection has poor locality of reference

Copying Collectors

- Copying collectors integrate the traversal (marking) and copying phases into one pass
  - All the live data is copied into one region of memory
  - All the remaining memory contains garbage, or has not yet been used
- Similar to mark-compact, but more efficient
- Time taken to collect is proportional to the number of live objects

Stop-and-copy Using Semispaces (1)

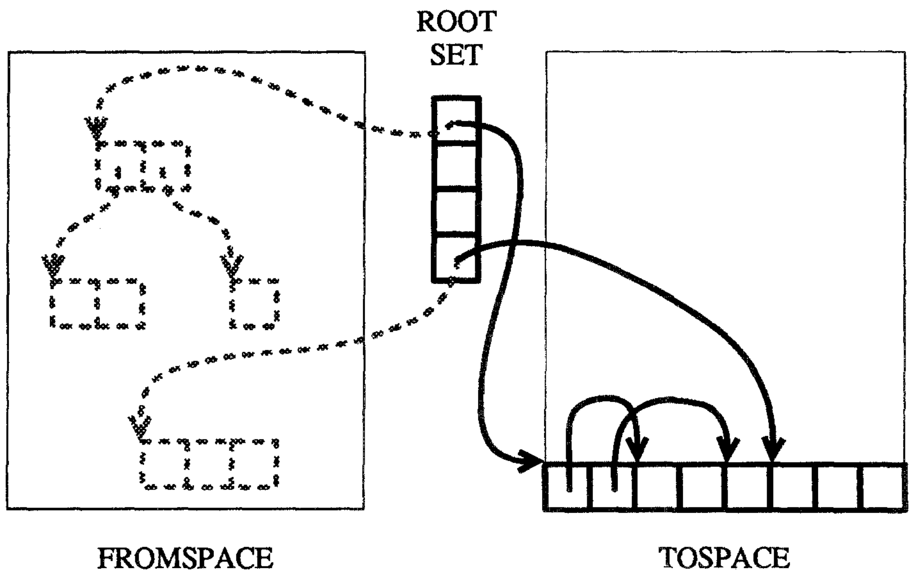
- Standard approach: a semispace collector, that uses the Cheney algorithm for copying traversal
- Divide the heap into two halves, each one a contiguous block of memory
- Allocations made linearly from one half of the heap only
  - Memory is allocated contiguously, so allocation is fast (as in the mark-compact collector)
  - No problems with fragmentation due to allocating data of different sizes
- When an allocation is requested that won't fit into the active half of the heap, a collection is triggered



Source: P. Wilson, "Uniprocessor garbage collection techniques", Proc IWMM'92, DOI 10.1007/BFb0017182

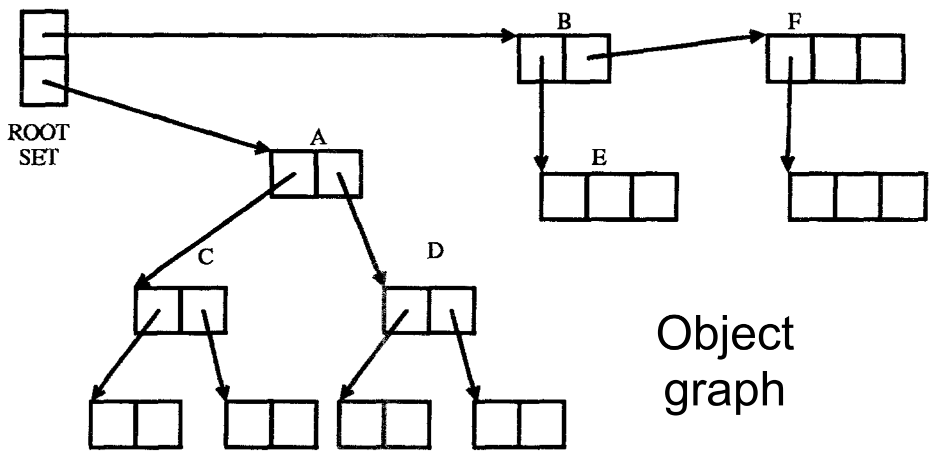
Stop-and-copy Using Semispaces (2)

- Collection stops execution of the program
- A pass is made through the active space, and all live objects are copied to the other half of the heap
  - The Cheney algorithm is commonly used to make the copy in a single pass
  - Anything not copied is unreachable, and is simply ignored (and will eventually be overwritten by a later allocation phase)
- The program is then restarted, using the other half of the heap as the active allocation region
- The role of the two parts of the heap (the two semispaces) reverses each time a collection is triggered

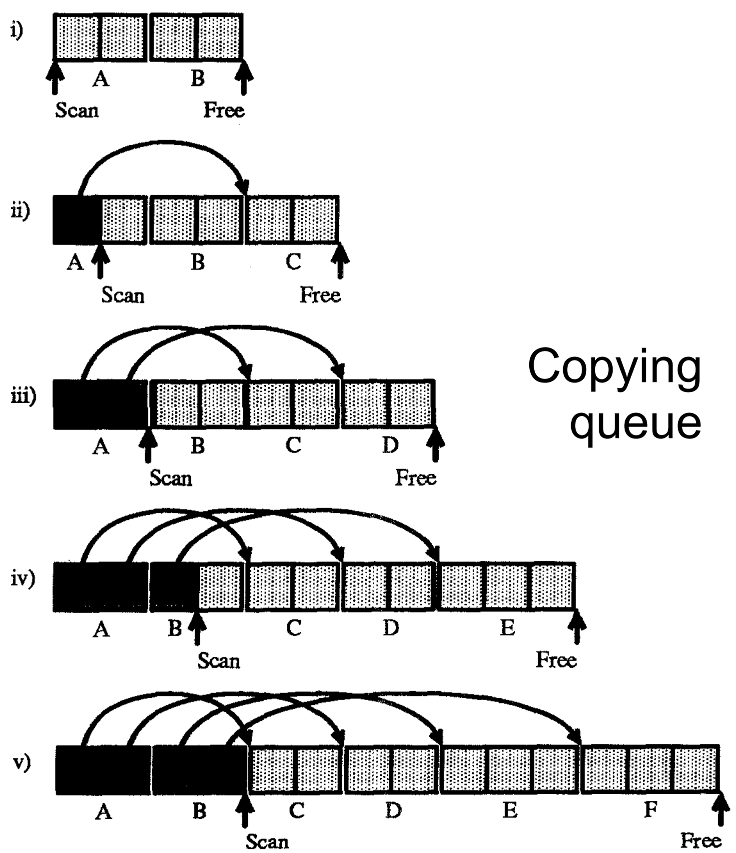


Source: P. Wilson, "Uniprocessor garbage collection techniques", Proc IWMM'92, DOI 10.1007/BFb0017182

Breadth-first Copying: Cheney Algorithm



Object graph



Copying queue

Source: P. Wilson, “Uniprocessor garbage collection techniques”, Proc IWMM’92, DOI 10.1007/BFb0017182

- The *root set* of objects is identified, and forms the initial queue of live objects to be copied
- Objects in the queue are examined in turn:
  - Each unprocessed object directly referenced by the object in the queue is itself added to the end of the queue
  - The object in the queue is copied to the other space, and the original is marked as having been processed (pointers are updated as the copy is made)
- Once the end of the queue is reached, all live objects have been copied

Efficiency of Copying Collectors

- Time taken for collection depends on the amount of data copied, which depends on the number of live objects
- Collection only happens when the semispace is full
- *If most objects die young*, then can reduce the data to be copied by increasing the size of the heap
  - Increasing the size of the heap increases the age to which objects need to live in order to be copied; most don’t live that long, and so aren’t copied
  - Trade-off memory for collection time: more memory used, less fraction of time spent copying data

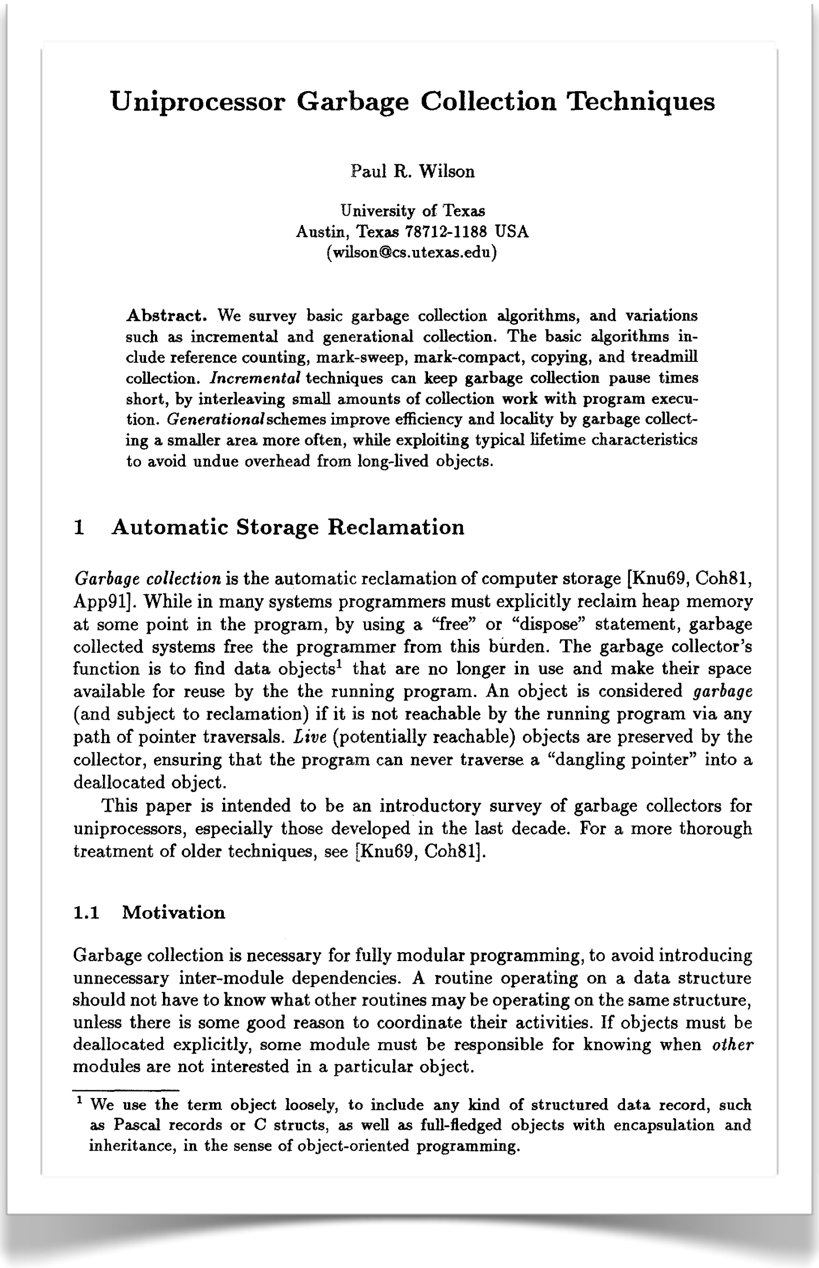
Concluding Remarks

- These approaches have broadly similar costs
  - But they move where the cost is paid: on allocation or collection; in terms of memory or processing time
  - Considering efficiency of copying collectors, and object lifetimes, leads to a possible optimisation: generational collectors (next lecture)

- Mark-sweep and reference counting don't move data, and so can work with weakly-typed data
  - In languages like C and C++, with casting and pointer arithmetic, it's hard to identify all possible pointers, but can usually identify values that might be pointers and be conservative in what's collected
  - But – can't move an object, if you can't be sure all pointers to it have been updated

Further Reading

- P. R. Wilson, “Uniprocessor garbage collection techniques”, In Proc. IWMM’92, St. Malo, France, DOI 10.1007/BFb0017182



Uniprocessor Garbage Collection Techniques

Paul R. Wilson

University of Texas
Austin, Texas 78712-1188 USA
(wilson@cs.utexas.edu)

Abstract. We survey basic garbage collection algorithms, and variations such as incremental and generational collection. The basic algorithms include reference counting, mark-sweep, mark-compact, copying, and treadmill collection. *Incremental* techniques can keep garbage collection pause times short, by interleaving small amounts of collection work with program execution. *Generational* schemes improve efficiency and locality by garbage collecting a smaller area more often, while exploiting typical lifetime characteristics to avoid undue overhead from long-lived objects.

1 Automatic Storage Reclamation

Garbage collection is the automatic reclamation of computer storage [Knu69, Coh81, App91]. While in many systems programmers must explicitly reclaim heap memory at some point in the program, by using a “free” or “dispose” statement, garbage collected systems free the programmer from this burden. The garbage collector’s function is to find data objects[1] that are no longer in use and make their space available for reuse by the the running program. An object is considered *garbage* (and subject to reclamation) if it is not reachable by the running program via any path of pointer traversals. *Live* (potentially reachable) objects are preserved by the collector, ensuring that the program can never traverse a “dangling pointer” into a deallocated object.

This paper is intended to be an introductory survey of garbage collectors for uniprocessors, especially those developed in the last decade. For a more thorough treatment of older techniques, see [Knu69, Coh81].

1.1 Motivation

Garbage collection is necessary for fully modular programming, to avoid introducing unnecessary inter-module dependencies. A routine operating on a data structure should not have to know what other routines may be operating on the same structure, unless there is some good reason to coordinate their activities. If objects must be deallocated explicitly, some module must be responsible for knowing when *other* modules are not interested in a particular object.

[1] We use the term object loosely, to include any kind of structured data record, such as Pascal records or C structs, as well as full-fledged objects with encapsulation and inheritance, in the sense of object-oriented programming.