Compaction

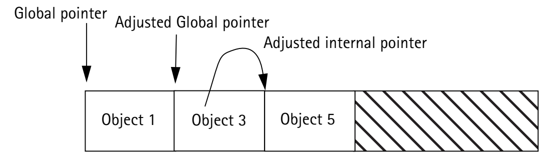
After the sweep phase, live heap objects are distributed throughout the heap space. This can lead to poor locality. If live objects span many memory pages, paging overhead may be increased. Cache locality may be degraded too.

We can add a *compaction phase* to mark-sweep garbage collection.

After live objects are identified, they are placed together at one end of the heap. This involves another tracing phase in which global, local and internal heap pointers are found and adjusted to reflect the object's new location.

Pointers are adjusted by the total size of all garbage objects

between the start of the heap and the current object. This is illustrated below:



Compaction merges together freed objects into one large block of free heap space. Fragments are no longer a problem.

Moreover, heap allocation is greatly simplified. Using an "end of heap" pointer, whenever a heap request is received, the end of heap pointer is adjusted, making heap allocation no more complex than stack allocation.

Because pointers are adjusted, compaction may not be suitable for languages like C and C++, in which it is difficult to unambiguously identify pointers.

Copying Collectors

Compaction provides many valuable benefits. Heap allocation is simple end efficient. There is no fragmentation problem, and because live objects are adjacent, paging and cache behavior is improved.

An entire family of garbage collection techniques, called *copying collectors* are designed to integrate copying with recognition of live heap objects. Copying collectors are very popular and are widely used.

Consider a simple copying collector that uses *semispaces*. We start with the heap divided into two halves—the *from* and *to spaces*.

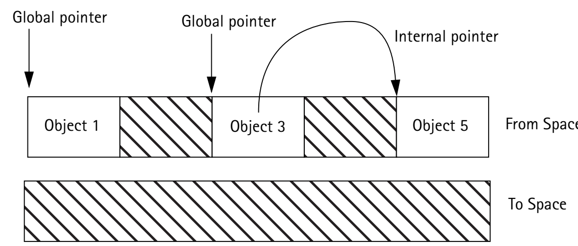
Initially, we allocate heap requests from the from space, using a simple “end of heap” pointer. When the from space is exhausted, we stop and do garbage collection.

Actually, though we *don’t* collect garbage. We collect live heap objects—garbage is never touched.

We trace through global and local pointers, finding live objects. As each object is found, it is moved from its current position in the from space to the next available position in the to space.

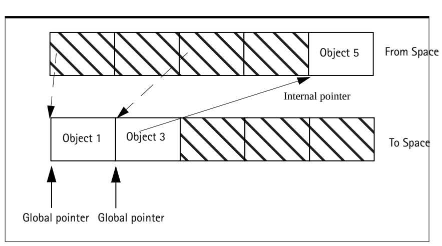
The pointer is updated to reflect the object’s new location. A “forwarding pointer” is left in the object’s old location in case there are multiple pointers to the same object.

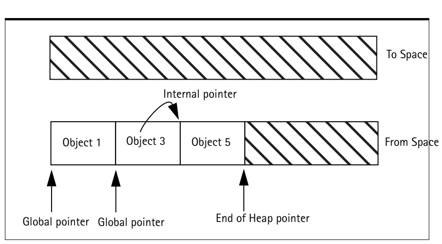
This is illustrated below:



The from space is completely filled. We trace global and local pointers, moving live objects to the to space and updating pointers. This is illustrated below. (Dashed arrows are forwarding pointers). We have yet to handle pointers internal to copied heap objects. All copied heap objects are traversed. Objects referenced are copied and internal pointers

are updated. Finally, the to and from spaces are interchanged, and heap allocation resumes just beyond the last copied object. This is illustrated in the lower figure.





The biggest advantage of copying collectors is their speed. Only live objects are copied; deallocation of dead objects is essentially free. In fact, garbage collection can be made, on average, as fast as you wish—simply make the heap bigger. As the heap gets bigger, the time between collections increases, reducing the number of times a live object must be copied. In the limit, objects are never copied, so garbage collection becomes free!

Of course, we can’t increase the size of heap memory to infinity. In fact, we don’t want to make the heap so large that paging is required, since swapping pages to disk is dreadfully slow. If we can make the heap large enough that the lifetime of most heap objects is less than the time between collections, then deallocation of short-lived objects will appear to be free, though longer-lived objects will still exact a cost.

Aren't copying collectors terribly wasteful of space? After all, at most only half of the heap space is actually used. The reason for this apparent inefficiency is that *any* garbage collector that does compaction must have an area to copy live objects to. Since in the worst case *all* heap objects could be live, the target area must be as large as the heap itself. To avoid copying objects more than once, copying collectors reserve a to space as big as the from space. This is essentially a space-time trade-off, making such collectors

very fast at the expense of possibly wasted space.

If we have reason to believe that the time between garbage collections will be greater than the average lifetime of most heaps objects, we can improve our use of heap space. Assume that 50% or more of the heap will be garbage when the collector is called. We can then divide the heap into 3 segments, which we'll call A, B and C. Initially, A and B will be used as the from space, utilizing 2/3 of the heap. When we copy live objects, we'll copy them into segment C, which will be big enough if half or more of the heap objects are garbage. Then we treat C and A as the from space, using B as the to space for the next collection. If we are

unlucky and more than 1/2 the heap contains live objects, we can still get by. Excess objects are copied onto an auxiliary data space (perhaps the stack), then copied into A after all live objects in A have been moved. This slows collection down, but only rarely (if our estimate of 50% garbage per collection is sound). Of course, this idea generalizes to more than 3 segments. Thus if 2/3 of the heap were garbage (on average), we could use 3 of 4 segments as from space and the last segment as to space.

Generational Techniques

The great strength of copying collectors is that they do no work for objects that are born and die between collections. However, not all heaps objects are so short-lived. In fact, some heap objects are very long-lived. For example, many programs create a dynamic data structure at their start, and utilize that structure throughout the program. Copying collectors handle long-lived objects poorly. They are repeatedly traced and moved between semispaces without any real benefit.

Generational garbage collection techniques were developed to better handle objects with varying lifetimes. The heap is divided into two or more *generations*, each with its own to and from space. New objects are allocated in the youngest generation, which is collected most frequently. If an object survives across one or more collections of the youngest generation, it is “promoted” to the next older generation, which is collected less often. Objects that survive one or more collections of this generation are then moved to the next older generation. This continues until very long-lived objects reach the oldest generation, which is collected very infrequently (perhaps even never).

The advantage of this approach is that long-lived objects are “filtered out,” greatly reducing the cost of repeatedly processing them. Of course, some long-lived

objects will die and these will be caught when their generation is eventually collected.

An unfortunate complication of generational techniques is that although we collect older generations infrequently, we must still trace their pointers in case they reference an object in a newer generation. If we don’t do this, we may mistake a live object for a dead one. When an object is promoted to an older generation, we can check to see if it contains a pointer into a younger generation. If it does, we record its address so that we can trace and update its pointer. We must also detect when an existing pointer inside an object is changed. Sometimes we can do this by checking “dirty bits” on

heap pages to see which have been updated. We then trace all objects on a page that is dirty. Otherwise, whenever we assign to a pointer that already has a value, we record the address of the pointer that is changed. This information then allows us to only trace those objects in older generations that might point to younger objects.

Experience shows that a carefully designed generational garbage collectors can be very effective. They focus on objects most likely to become garbage, and spend little overhead on long-lived objects. Generational garbage collectors are widely used in practice.

Conservative Garbage Collection

The garbage collection techniques we’ve studied all require that we identify pointers to heap objects accurately. In strongly typed languages like Java or ML, this can be done. We can table the addresses of all global pointers. We can include a code value in a frame (or use the return address stored in a frame) to determine the routine a frame corresponds to. This allows us to then determine what offsets in the frame contain pointers. When heap objects are allocated, we can include a type code in the object’s header, again allowing us to identify pointers internal to the object.

Languages like C and C++ are weakly typed, and this makes identification of pointers much harder. Pointers may be type-cast into integers and then back into pointers. Pointer arithmetic allows pointers into the middle of an object. Pointers in frames and heap objects need not be initialized, and may contain random values. Pointers may overlay integers in unions, making the current type a dynamic property.

As a result of these complications, C and C++ have the reputation of being incompatible with garbage collection. Surprisingly, this belief is false. Using *conservative garbage collection*, C and C++ programs *can* be garbage collected.

The basic idea is simple—if we can't be sure whether a value is a pointer or not, we'll be conservative and assume it is a pointer. If what we think is a pointer isn't, we may retain an object that's really dead, but we'll find all valid pointers, and never incorrectly collect a live object. We may mistake an integer (or a floating value, or even a string) as an pointer, so compaction in any form *can't* be done. However, mark-sweep collection will work.

Garbage collectors that work with ordinary C programs have been developed. User programs need not be modified. They simply are linked to different library routines, so that **`malloc`** and **`free`** properly support the garbage collector. When new heap space is

required, dead heap objects may be automatically collected, rather than relying entirely on explicit **`free`** commands (though **`free`**s are allowed; they sometimes simplify or speed heap reuse).

With garbage collection available, C programmers need not worry about explicit heap management. This reduces programming effort and eliminates errors in which objects are prematurely freed, or perhaps never freed. In fact, experiments have shown that conservative garbage collection is very competitive in performance with application-specific manual heap management.