

CS 536

Introduction to Programming Languages and Compilers

Charles N. Fischer

Lecture 12

Heap Management

A very flexible storage allocation mechanism is *heap allocation*.

Any number of data objects can be allocated and freed in a memory pool, called a *heap*.

Heap allocation is enormously popular. Almost all non-trivial Java and C programs use **new** or **malloc**.

Heap Allocation

A request for heap space may be *explicit* or *implicit*.

An explicit request involves a call to a routine like **new** or **malloc**.

An explicit pointer to the newly allocated space is returned.

Some languages allow the creation of data objects of unknown size. In Java, the + operator is overloaded to represent string catenation.

The expression **Str1 + Str2** creates a new string representing the catenation of strings **Str1** and **Str2**. There is no compile-time bound on the sizes of **Str1** and **Str2**, so heap space must be implicitly allocated to hold the newly created string.

Whether allocation is explicit or implicit, a *heap allocator* is needed. This routine takes a size parameter and examines unused heap space to find space that satisfies the request.

A *heap block* is returned. This block must be big enough to satisfy the space request, but it may well be bigger.

Heaps blocks contain a *header* field that contains the size of the block as well as bookkeeping information.

The complexity of heap allocation depends in large measure on how *deallocation* is done.

Initially, the heap is one large block of unallocated memory. Memory requests can be satisfied by simply modifying an “end of

heap” pointer, very much as a stack is pushed by modifying a stack pointer.

Things get more involved when previously allocated heap objects are deallocated and reused.

Deallocated objects are stored for future reuse on a *free space list*.

When a request for n bytes of heap space is received, the heap allocator must search the free space list for a block of sufficient size. There are many search strategies that might be used:

- **Best Fit**

The free space list is searched for the free block that matches most closely the requested size. This minimizes wasted heap space, the search may be quite slow.

- **First Fit**

The first free heap block of sufficient size is used. Unused space within the block is split off and linked as a smaller free space block. This approach is fast, but may “clutter” the beginning of the free space list with a number of blocks too small to satisfy most requests.

- **Next Fit**

This is a variant of first fit in which succeeding searches of the free space list begin at the position where the last search ended. The idea is to “cycle through” the entire free space list rather than always revisiting free blocks at the head of the list.

- **Segregated Free Space Lists**
There is no reason why we must have only *one* free space list. An alternative is to have several, indexed by the size of the free blocks they contain.

Deallocation Mechanisms

Allocating heap space is fairly easy. But how do we deallocate heap memory no longer in use?

Sometimes we may never need to deallocate! If heaps objects are allocated infrequently or are very long-lived, deallocation is unnecessary. We simply fill heap space with “in use” objects.

Virtual memory & paging may allow us to allocate a very large heap area.

On a 64-bit machine, if we allocate heap space at 1 MB/sec, it will take 500,000 *years* to span the entire address space!

Fragmentation of a very large heap space commonly forces us to include some form of reuse of heap space.

User-controlled Deallocation

Deallocation can be manual or automatic. Manual deallocation involves explicit programmer-initiated calls to routines like **free(p)** or **delete(p)**.

The object is then added to a free-space list for subsequent reallocation.

It is the programmer's responsibility to free unneeded heap space by executing deallocation commands. The heap manager merely keeps track of freed space and makes it available for later reuse.

The really hard decision—when space should be freed—is shifted to the programmer, possibly leading to catastrophic *dangling pointer* errors.

Consider the following C program fragment

```
q = p = malloc(1000);  
free(p);  
/* code containing more malloc's */  
q[100] = 1234;
```

After **p** is freed, **q** is a *dangling pointer*. **q** points to heap space that is no longer considered allocated.

Calls to **malloc** may reassign the space pointed to by **q**.

Assignment through **q** is illegal, but this error is almost never detected.

Such an assignment may change data that is now part of another heap object, leading to very subtle errors. It may even change a header field or a free-space link, causing the heap allocator itself to fail!

Automatic Garbage Collection

The alternative to manual deallocation of heap space is *garbage collection*.

Compiler-generated code tracks pointer usage. When a heap object is no longer pointed to, it is *garbage*, and is *automatically* collected for subsequent reuse.

Many garbage collection techniques exist. Here are some of the most important approaches:

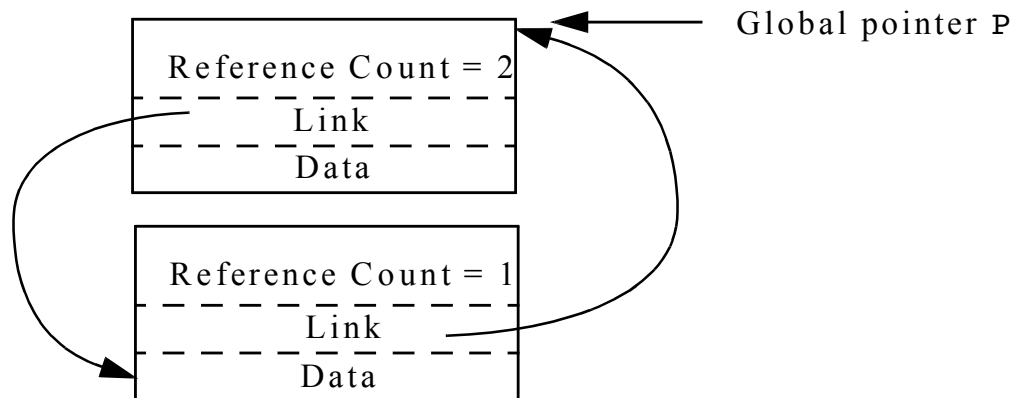
Reference Counting

This is one of the oldest and simplest garbage collection techniques.

A *reference count* field is added to each heap object. It counts how many references to the heap object exist. When an object's reference count reaches zero, it is garbage and may be collected.

The reference count field is updated whenever a reference is created, copied, or destroyed. When a reference count reaches zero and an object is collected, all pointers in the collected object are also followed and corresponding reference counts decremented.

As shown below, reference counting has difficulty with *circular structures*. If pointer **P** is



set to null, the object's reference count is reduced to 1. Both objects have a non-zero count, but neither is accessible through any external pointer. The two objects are garbage, but won't be recognized as such.

If circular structures are common, then an auxiliary technique, like mark-sweep collection, is needed to collect garbage that reference counting misses.

Mark-Sweep Collection

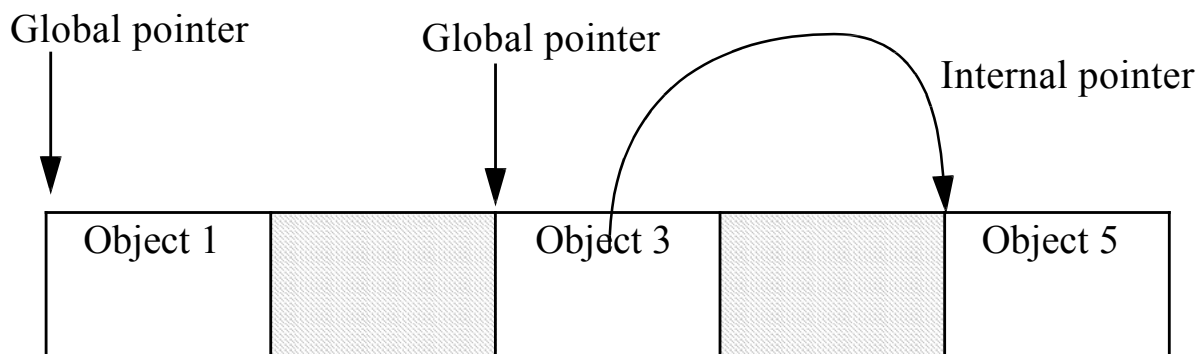
Many collectors, including mark & sweep, do *nothing* until heap space is nearly exhausted.

Then it executes a *marking phase* that identifies all live heap objects.

Starting with global pointers and pointers in stack frames, it marks reachable heap objects. Pointers in marked heap objects are also followed, until all live heap objects are marked.

After the marking phase, any object not marked is garbage that may be freed. We then *sweep* through the heap, collecting all unmarked objects. During the sweep phase we also clear all marks from heap objects found to be still in use.

Mark- sweep garbage collection is illustrated below.



Objects 1 and 3 are marked because they are pointed to by global pointers. Object 5 is marked because it is pointed to by object 3, which is marked. Shaded objects are not marked and will be added to the free- space list.

In any mark- sweep collector, it is vital that we mark *all* accessible heap objects. If we miss a pointer, we may fail to mark a live heap object and later incorrectly free it. Finding all pointers is a bit tricky

in languages like Java, C and C ++ , that have pointers mixed with other types within data structures, implicit pointers to temporaries, and so forth. Considerable information about data structures and frames must be available at run- time for this purpose. In cases where we can't be sure if a value is a pointer or not, we may need to do *conservative garbage collection*.

In mark- sweep garbage collection *all* heap objects must be swept. This is costly if most objects are dead. We'd prefer to examine *only* live objects.

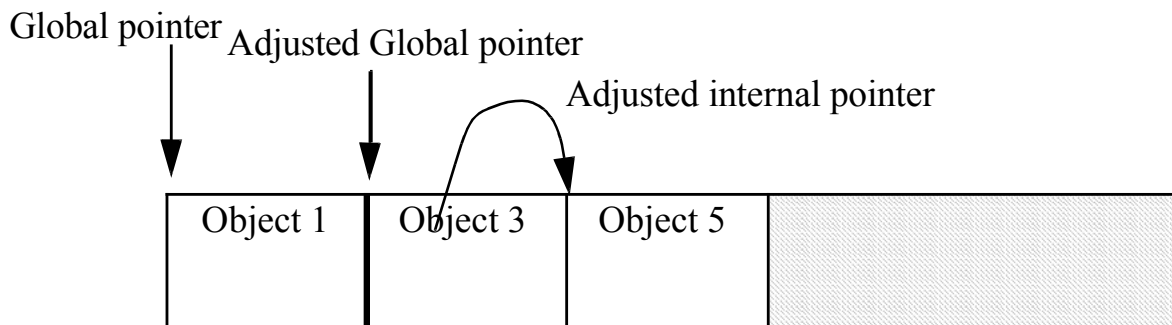
Compaction

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.