# Dynamic Storage Allocation

Dynamic Storage allocation refers to the run-time allocation of storage for variables, structures, procedure calls, etc.

- For most languages, the total amount of storage required for these of objects cannot be determined at compile time.
    - FORTRAN is the exception.
- Some language features requiring dynamic allocation:
    - Recursion
    - Pointers, explicit allocation (new, malloc, cons)
    - Dynamic data structures (lists, trees, graphs, dynamic arrays)
    - Higher order functions (lambda's, etc.)

# Stack-based vs. Heap Based Allocation

In C, Pascal, and Ada, most objects are allocated on the stack.

- Last-In-First-Out allocation is appropriate for activation records because local variables and parameters do not outlive the procedure calls that created them.
- When a procedure returns, its activation record is removed from the stack.
    - The local variables are automatically "deallocated" i.e. the space they occupy becomes available for reuse.
    - Works fine for local variables that are integers, booleans, characters, floats, records and arrays. The only way to access these structures is via the name of the variable.
- When the scope of the local variables is exited (by procedure return), their values can no longer be accessed and thus deallocation is safe.

What about pointers?

- Pointers are used for aliasing. They allow multiple identifiers to refer to the same object.
- A pointer's value is the address of object that it points to.
- Pointer values may be assigned, passed as parameters, etc.
    - No copying of the object is entailed in these operations.

In general, when the procedure containing a local pointer variable p returns, we cannot assume that the object that p points to can longer be accessed.

- The pointer value may have been assigned to a non-local variable, stored in a global structure, or returned by the procedure.

```
struct foo *f()
{   struct foo *a;
    a = (struct foo *) malloc(sizeof(struct foo));
    return a;
 }
```

```
type foo = record ... end;
var b: ^foo;
 . . .
procedure f;
     var a:^foo;
begin
      new(a);
     b := a
end;
```

```
(define (f g x y)
    (let ((a (cons x y)))
       (g a)))
```

The object that a points to in each case cannot be deallocated when the procedure f exits. Thus, the object cannot be stored in f's activation record on the stack.

- This is because the lifetime (aka *extent*) of the object is greater than the lifetime of the procedure that created it.
    - The lifetime depends on the execution of the program, thus the object is said to have *dynamic extent*.

The area of memory in which objects with dynamic extent are allocated is called the *heap*.

If the program creates a sufficient number of heap-allocated objects, the heap will fill up.

- At this point no more heap space will be available for more objects and the program will crash.

However, some of the objects in the heap may no longer be needed in the program. The storage for these objects should be reclaimed, i.e. made available for subsequent use.

Reclaiming heap-allocated objects can be specified explicitly by the user (via calls to free or delete) or can be performed automatically by the system.

Automatic reclamation of heap allocated objects is generally called *garbage collection*.

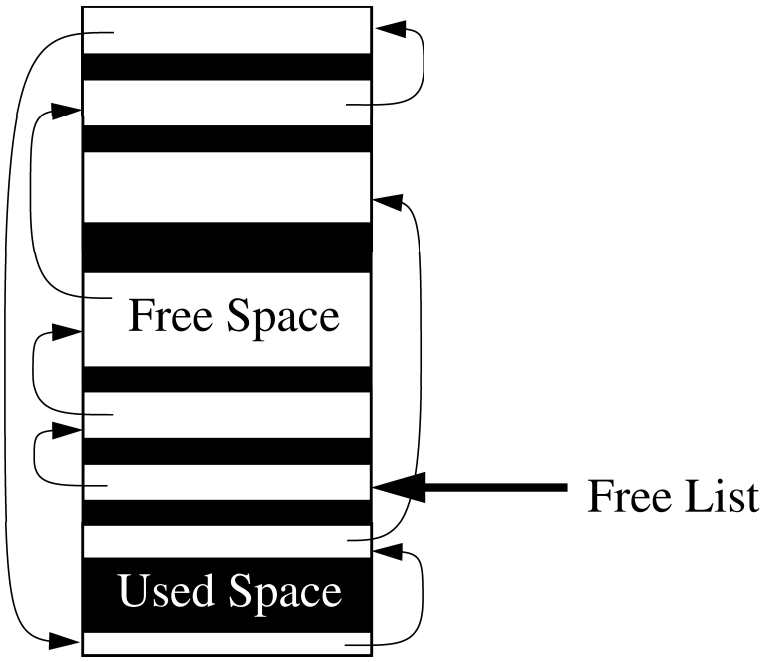
# Heap Storage Allocation: Free-List vs. Heap Pointer

Heap space for objects can be allocated in one of (at least) two ways:

1. Free List Method

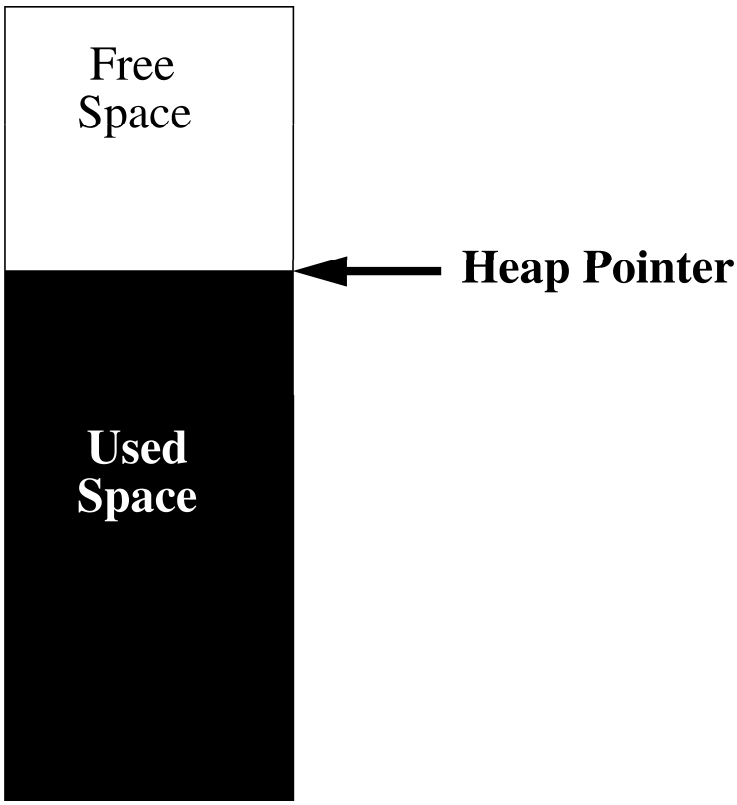
- Available blocks of storage are chained together in a linked list, called the *free list*.
- When a block of storage is required, it is removed from the free list. When a block of storage is reclaimed, it is inserted in the free list.
- Disadvantages:
  - The free list may have to searched for a block of the right size.
    - Leads to fragmented memory (analogous to segmented virtual memory in operating systems).

- Expensive allocation, cheap reclamation



2. Heap Pointer Method:

- Maintain available space as a contiguous block of bytes.
- A heap pointer, analogous to a stack pointer, points to the next available byte.
- When an object is allocated, the heap pointer is bumped up by the appropriate amount.
- Disadvantage:
  - Requires compaction of live objects (pushing them together until they are all adjacent).
- Cheap allocation, expensive reclamation (?)



# Storage Reclamation: Implicit vs Explicit

Programs written in Pascal, C, and Ada generally do not make heavy use of the heap.

- Implementations of these languages typically do not perform garbage collection.
- Instead, functions to reclaim objects are provided to the user (e.g. free, delete).

Very high level languages like LISP, Scheme, ML, Prolog, and Setl make heavy use of the heap and require garbage collection.

- LISP: Lists, S-expressions, Symbols (w/ properties)
- Scheme, ML: Lists, Closures (for first-class functions)
- Prolog: Lists, Substitutions, program points (for backtracking)
- Setl: Sets, Bags, Maps.

# Categories of Garbage Collection (GC)

Simply put, the two tasks that a garbage collector performs are:

1. Find each block of storage occupied by an object that cannot subsequently be referenced during execution (a *dead* object), and
2. Make that storage available for subsequent use.

There are two major categories of garbage collection that perform the first task in fundamentally different ways:

1. *Mark/Sweep Collectors* and *Copying Collectors*
   - When the heap fills up, these collectors traverse memory, determining the set of all *live* objects (i.e. those not dead).
   - The complement of this set is the set of dead objects, whose storage can be reclaimed.
2. *Reference Counting Collectors*
   - During execution, these determine when a particular object has become dead.
   - At that point, the storage for that object is reclaimed.

# Mark/Sweep and Copying Garbage Collection

During execution, the heap-allocated objects can be referenced in a variety of ways: as the values of variables, components in a structure, etc.

In particular, an object x is *live* (i.e. can be referenced) if:

1. It is pointed to directly by a variable, i.e. there is a pointer to x in an activation record (or global data area).
2. There is a register, containing a temporary or intermediate value, that points to x.
3. There is an object y in the heap containing a pointer to x, and y is itself live.

It is clear that all live objects in the heap can be found by a graph traversal.

- The starting points for the traversal are the local variables on the stack, the global variables, and the registers. These are called the *roots* of the traversal.
- Any object that is not reachable from the roots is dead and can be reclaimed.

Mark/Sweep Collectors and Copying Collectors perform a graph traversal. They differ in the actions that are performed during and after the traversal.

# Mark/Sweep Garbage Collection

Basic Idea:

- Each object contains an extra bit called a *mark bit*.
- When the heap fills up, program execution is suspended and the garbage collection process begins.
- The garbage collector traverses the heap, starting at the roots, and sets the mark bit of each object encountered. This traversal is called the *mark phase*.
- When the traversal is complete, all live objects will have their mark bits set. Thus, any object whose mark bit is not set is unreachable. The collector then scans the entire memory, placing each object whose mark bit is not set on the free list. This is called the *sweep phase*.

```
Garbage_collect()
{ for each root pointer p do
      mark(p);
  sweep() ;
}
mark(p)
{ if p->mark <> 1 then
     for each pointer field p->x do
              mark(p->x);
}
sweep()
 { for all objects o in memory do
     if o.mark = 0 then insert(o, free_list);
 }
```

# Cost of Mark/Sweep GC

If L = amount of storage occupied by live objects
M = size of heap
then

$Cost_{M/S} = O(L) + O(M)$
$= O(M)$, since $M > L$.

- Cost of Mark/Sweep is proportional to the size of the heap, no matter how many of live objects there are.

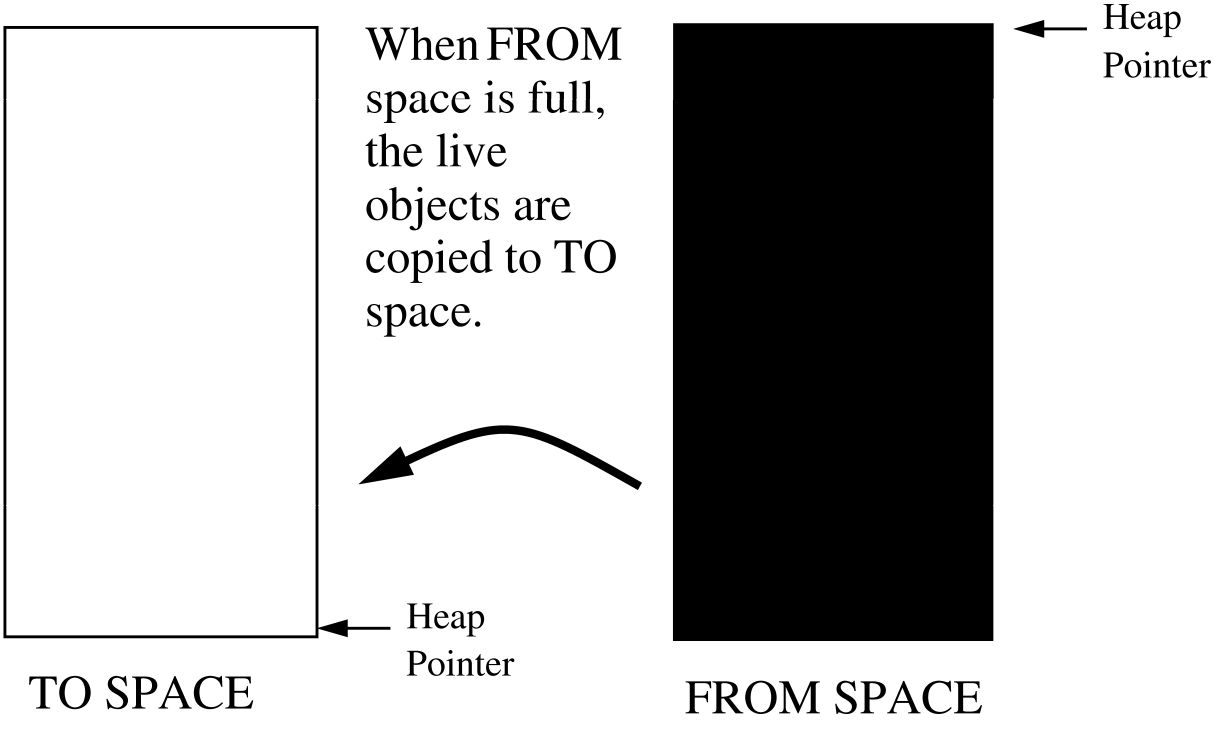
## Disadvantages of Mark/Sweep Collection:

- Non-compacting, so free list must be used.
  - Expensive allocation.
- Program suspends during GC.
  - unsuitable for real-time programs.
  - known as a "stop-and-collect" method.

# Copying Garbage Collection

Basic Idea:

- There are two heaps, only one of which is in use. The one in use is called the FROM space and the other one is called the TO space.
- Objects are allocated in the FROM space. When it fills up, the garbage collection process begins.
- As the collector traverses the graph, it copies each object it encounters from the FROM space to the (empty) TO space.
- When the traversal is complete, all live objects will have been copied into TO space.
    - At this point, the spaces are "flipped", i.e. the TO space becomes the FROM space, the FROM space becomes the TO space, and program execution resumes.



# Properties of copying collection:

Each live object is copied into the first available space in TO space.

- When the traversal is complete, the live storage will have been compacted. Thus, heap allocation is performed via a heap pointer.

Since objects are being moved from FROM space to TO space, pointers to the object must be updated to reflect its new address.

- This is achieved by leaving a "forwarding address" in the old object once it is moved.
- If a pointer to the old object is encountered later on during the traversal, the pointer is updated with the forwarding address.

How can you tell if an address is is a "forwarding address" or an ordinary pointer?

- A forwarding address is an address in TO space. No ordinary pointer can point from FROM space into TO space.

# Copying GC

```
Garbage_collect()
{ for each root pointer p do
     traverse(p);
 }


Traverse(p)
 {  if p-> contains a forwarding address q   then
     p:= q                    /*assume pass by reference*/
   else {
     new_p := copy(p, TO_SPACE);
     write_forward_address(p,new_p);
     for each pointer field p->x do
           traverse(p->x) ;
     }
}
```

# Cost of Copying Collection

If L = amount of storage occupied by live object
M = size of each heap
then

$$Cost_{copy} = O(L)$$

- Unlike mark/sweep GC, the cost is proportional to the amount of live storage, not the size of the heap.

Experimental Data for LISP:

$$L = 0.3 * M$$

Since each heap is half the size of the heap in mark/sweep GC, copying garbage collection will happen twice as frequently.

- But, in a virtual memory system, only the FROM space need occupy physical memory. The TO space will be swapped out to disk.

## Disadvantages

- "Stop-and-copy" is unsuitable for real-time.

# Reference Counting

Reference counting collection is based on a completely different idea.

Each object contains an extra field called a *reference count*.

- During execution, the reference count for an object contains the count of all pointers pointing to the object.
- When a pointer to an object is copied, the object's reference count is incremented.
- When a pointer to an object is destroyed, the object's reference count is decremented.
- When an object's reference count becomes 0, the object can be reclaimed.
    - Any pointer contained in the object is "destroyed",causing more reference count activity
    - The object is placed on the free list.

The code for copying and destroying pointers is:

```
copy(p1,p2)          /* assume pass by reference */
{ p2 := p1;
  p1->ref_count := p1->ref_count + 1;
}


destroy(p)
{ p->refcount = p->refcount - 1;
  if p->refcount := 0 then
       for each pointer field p->x do
                 { destroy(p->x);
                   insert(p-> ,free_list);
                   }
}
```

# Cost of Reference Counting

It is hard to compare the cost of reference counting to mark/sweep or copying GC.

$$\text{Cost}_{RC} = O(\#\text{ pointers created}) + O(\#\text{ pointers destroyed})$$

but this cost is spread over the whole execution.

## Advantage

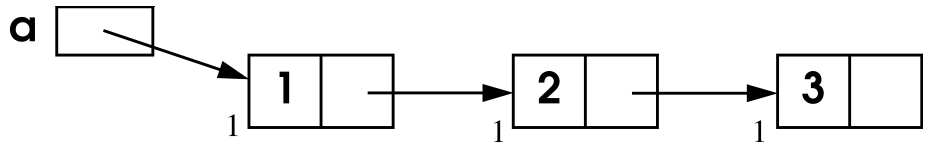
Storage reclamation is incremental,

- happens a little bit at a time.
- The program will not be interrupted for long. (No guarantee, though).
    - Suitable for real-time programs (?)

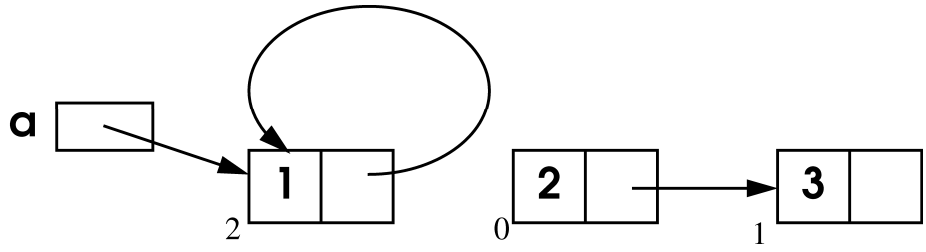
## Disadvantages:

1. Cycles are a big problem! Consider:

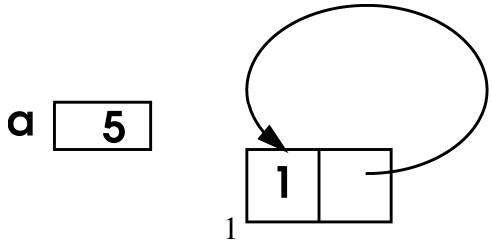
**(set! a '(1 2 3))**




**(set-cdr! a a)**




**(set! a 5)**




- The the first cons cell is dead but will never be reclaimed.
- Have a backup mark/sweep collector for when the heap fills up with cycles.

2. Tricky to implement.
   - It is easy to introduce bugs in reference counting when generating code to create temporaries, pass parameters, etc.
3. Each object must have a reference count. No matter how many bits are allocated, there will always be overflow situations.

## Interesting variant of Reference Counting

*one-bit* reference count:

- When the initial pointer to the object is created, the reference count is set to 0. Thus, a 0 really means that there is only one pointer to the object.
- If a pointer is destroyed and the reference count of the object is 0, then the object can be reclaimed.
- Once a reference count = 1, it is never decremented.
    - know as a *sticky* reference count

Shared objects will not be reclaimed!

- Experimental data: 97% of objects are not shared.
- Again, have a backup mark/sweep collector.

The one-bit reference count can also be put in the pointer, rather than the object

# GC Implementation, Design, and Research Issues

# Compacting Mark/Sweep Garbage Collection

Advantages:

- Fast allocation using heap pointer.
- Larger heap (thus fewer collections) than copying collectors.

Disadvantage:

- Tricky algorithms, most require several passes.

Basic Idea: Sweep phase pushes all live objects together.

- Build a table indicating how much each object will move, used to to modify pointers to that object.
- Each object will move n bytes, where n is the total storage of the dead objects below it.

First sweep pass:

- Build table

Second sweep pass:

- Move objects and modify pointers.

The table can be constructed in the heap by overwriting dead objects.

- Very tricky, though.

# Non-recursive Mark/Sweep and Copying Collectors

Major problem with the simple Mark/Sweep and Copying Collectors as described here:

- The traversals are recursive
- Maximum stack size could concievably be proportional to the number of live objects!

Solution:

- Perform non-recursive graph traversals.

Two popular methods (only first works for mark/sweep):

- Pointer reversal
- Trace from objects already copied into TO space.
    - Breadth-first traversal.

# Generational Copying Garbage Collection

Experimental Observation:

- The older an object gets, the longer it can be expected to live.

This makes sense for the following reasons:

- Many objects, representing temporary or intermediate values, live for only a short time (i.e. become garbage soon after creation).
- Those objects that live longer tend to correspond to the important objects (globally accessible variables, central data structure of the program, etc.)

*Generational Copying GC* exploits this property in the following way:

- Instead of two heaps, there are many heaps.
- Each heap contains objects of similar age.
    - These heaps are called generations, for obvious reasons.
    - There are young generations, middle-aged generations, and old generations.
- The younger generations are garbage collected much more frequently than the older generations, since a higher percentage of the objects will be garbage in the younger generations.
    - Thus, the cost of GC is reduced because only a fraction of the live objects are traversed each time.
- Objects in one generation that are traversed during garbage collection are copied into the next older generation.
    - If the older generation fills up, then that generation will be garbage collected as well, etc.

What are the roots of a generational copying collector?

It most cases, older objects will not point to younger objects.

- In purely functional languages, they *cannot*.
- Thus (in FL's), when traversing the youngest generation, the roots are simply those registers, local variables, and global variables that point into the youngest generation.

An object in an older generation can point to an object in a younger generation only as the result of an assignment.

- If the younger object is only reachable via the older object, then it will be incorrectly reclaimed when the younger generation is collected.
- Thus, upon the assignment, the address of the younger object is stored in a list of *incoming references*. During collection, the elements of the list serve as roots of the traversal. Each generation has its own incoming reference list.

# Incremental Copying Garbage Collection

Idea:

- Allocate objects in TO space, and copy a few objects from FROM space to TO space every time an allocation routine (cons, etc) is called.
- By the time TO space fills up, all live objects in FROM space will have been copied into TO space.
    - Must choose right number of objects to copy during each allocation.
- At this point, the spaces can be flipped.

How is the (incremental) traversal performed?

- By tracing from objects already copied into TO space (as described earlier).

# Tagged data vs. untagged data

# Parallel and/or Distributed GC