# Garbage Collection

15-745 Optimizing Compilers

*Spring 2006*

## Garbage collection

In L3, the storage for some values does not follow a stack discipline for its allocation and deallocation

```
struct intlist {
   val: int;
   next: intlist*;
 };
```

```
intlist* f(int x) {
  var l1: intlist;
  var l2: intlist*;
 …
  l2 = alloc(…);
 …
  return l2;
}
```

## Reclaiming heap storage

If we do nothing to reclaim heap storage after it is used, then eventually some programs will (needlessly) exhaust memory

Several partial solutions:

- explicit deallocation ("**`free(l)`**")
- reference counting
- tracing garbage collector
- conservative garbage collector
- memory mgmt based on region analysis

## Explicit deallocation

Explicit deallocation is

- a dangerous source of bizarre bugs
- highly prone to error
- slow

As such, it is a stupid feature for the vast majority of programming tasks. We will not consider it further.

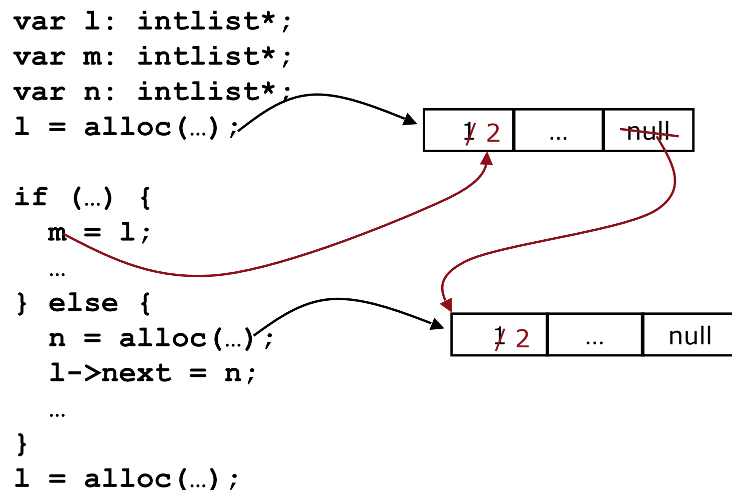
## *Reference Counting*

# Reference counting

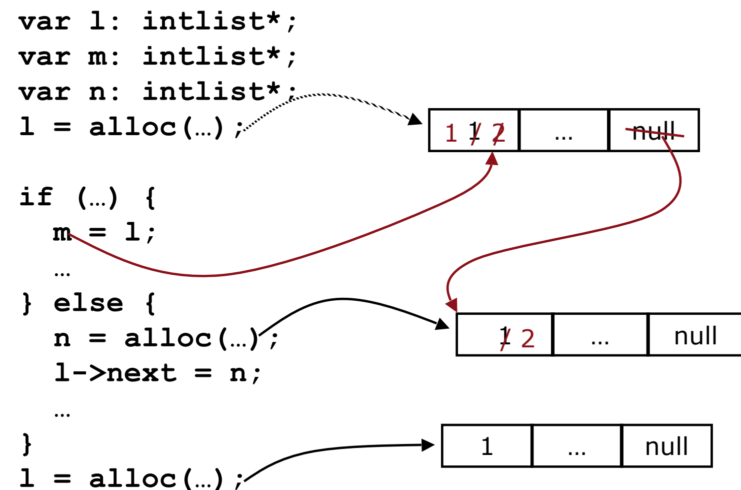
Basic idea:

- add an extra integer (the "*reference count*") to every heap-allocated data structure
- when first allocated, initialize the reference count to 1
- whenever a new reference to the data structure is established, increment the reference count
- whenever a reference disappears, decrement the reference count and check if it is zero
  - if it is zero, then reclaim the storage

# Reference counting example




# Reference counting example



# RC is naïve

Reference counting has been (re)invented many, many times

It is often the first idea that comes to mind for dynamic memory management and has the advantage of simplicity

- also real-time behavior
- also, objects stay in one place

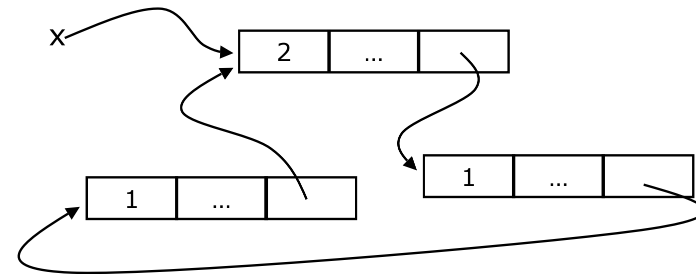
However, it is inefficient

- in storage, because of the reference counts, and
- in speed, because of the need always to maintain the counts

# RC is broken

More importantly, for most programming language applications, reference counting doesn't work well for cyclic structures.

- several techniques for this, but hard and messy




*Mark and Sweep*

# Mark and Sweep

Basic idea:

- maintain a linked list, called the *free list*, that contains all of the heap blocks that can be allocated
- if the program requests a heap block but the free list is empty, invoke the *garbage collector*
  - starting from the roots, trace and mark all reachable heap blocks
  - sweep and collect all unmarked heap blocks, putting each one back on the free list
  - sweep again, unmarking all heap blocks
- first developed by McCarthy in 1960

# Free lists

Suppose that every allocated heap block is two words long

Then a free list is simply a linked list of these blocks

In practice, can have separate free lists for 2, 4, 8, … -word blocks

# Roots

We need to find all heap blocks that might still be used by the program

- if one of the program's live variables points to a heap block, then that heap block *plus all of the heap blocks that it (directly or indirectly) points to* could (conservatively) be used in the future

To find all of these blocks, we must start with the heap pointers that currently exist in the registers and run-time stack

- these are called the *root pointers*
- the root pointers point to the *pointer graph*

# Finding roots

Which registers and stack slots contain heap pointers?

- not a trivial question, because pointers and non-pointers are indistinguishable

Note that we only need to know this at each heap-allocation point. Why?

Essentially, findings roots comes down to deciding whether a word value is or isn't a pointer

# Pointers vs non-pointers

Two main approaches:

- Every word value uses a tag bit
  - eg: low-order bit = 0 means pointer, low-order bit = 1 means non-pointer
  - Each heap block uses its first word to record the length of the block

or…

- For each call to _alloc(), the compiler emits a bit string that tells which registers contain heap pointers
  - a bit string is also emitted for each procedure, to declare which stack slots contain pointers

## Mark and Sweep

Basic idea:

- maintain a linked list, called the *free list*, that contains all of the heap blocks that can be allocated
- if the program requests a heap block but the free list is empty, invoke the *garbage collector*
  - starting from the roots, trace and mark all reachable heap blocks
  - sweep and collect all unmarked heap blocks, putting each one back on the free list
  - sweep again, unmarking all heap blocks

## Tracing

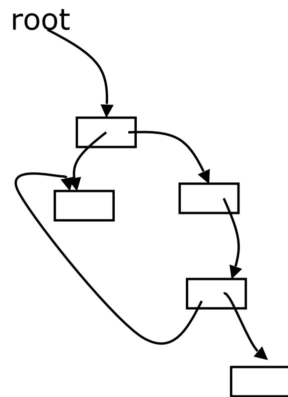
How to trace the pointer graph?

- could do a depth-first or breadth-first traversal

But if we are truly out of memory, we will not have space for the stack/queue!

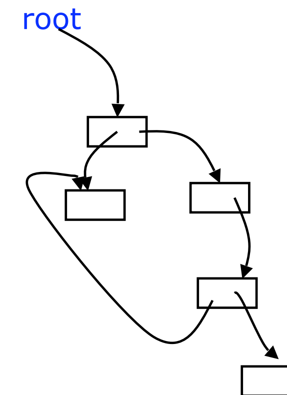
- how large could the stack/queue get?

## Pointer reversal

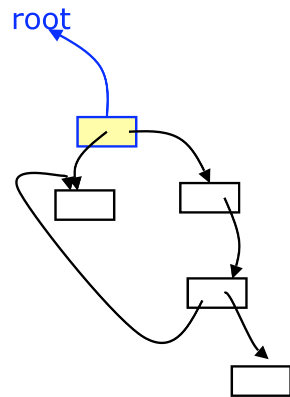



The basic idea is to do a depth-first traversal, and encode the stack in the pointer graph itself, by reversing pointers as we go

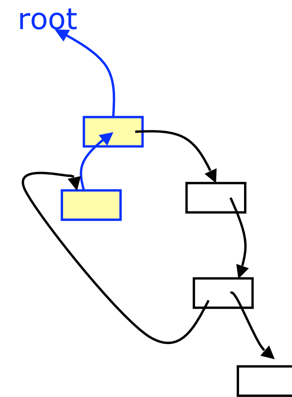
## Pointer reversal example



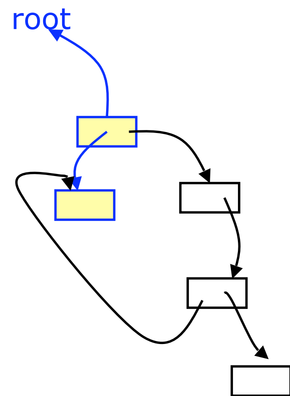
## Pointer reversal example



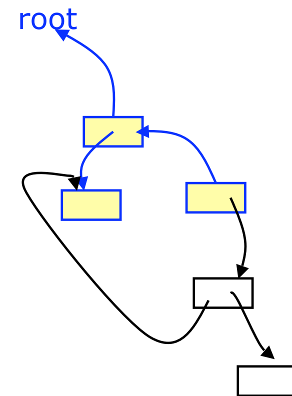

## Pointer reversal example



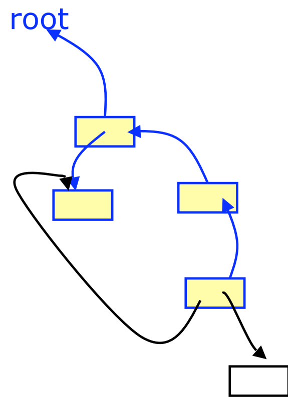

## Pointer reversal example



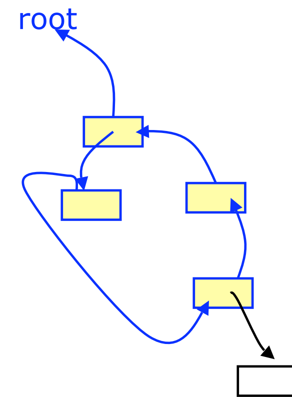

## Pointer reversal example



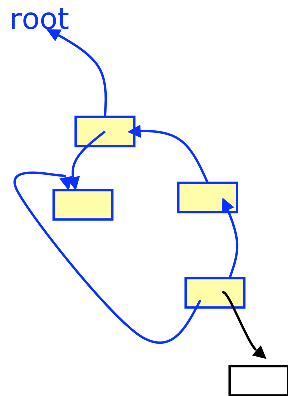
# Pointer reversal example



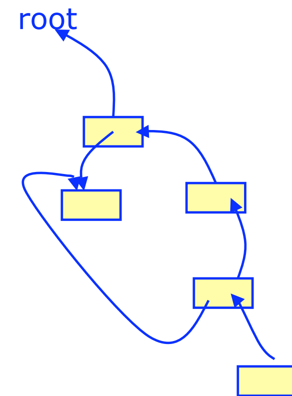

# Pointer reversal example



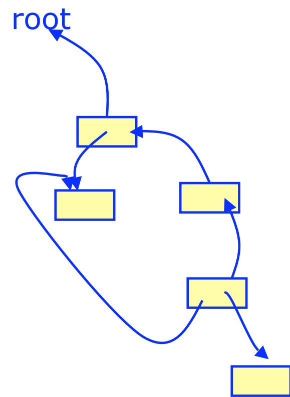

# Pointer reversal example



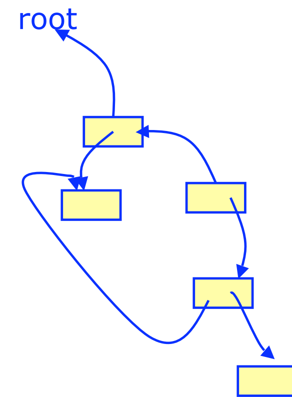

# Pointer reversal example



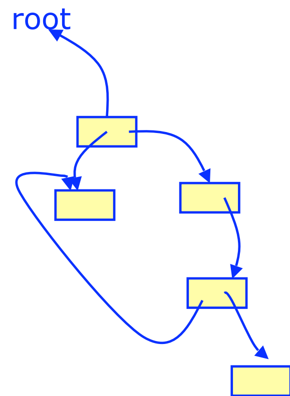
## Pointer reversal example



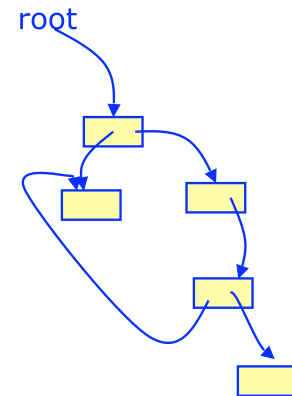

## Pointer reversal example




## Pointer reversal example




## Pointer reversal example



## Marking

Where to put the mark bits?

If we use a header word for every heap block, then can use a bit in the header word

Can also use a table of mark bits

## How fast is mark-and-sweep?

Assume all heap blocks are the same size. Let

- $c_1$ = the cost to trace and mark a heap block
- $c_2$ = the cost to put a heap block back on the free list
- H = the size of the heap, in terms of number of heap blocks
- R = the number of reachable heap blocks

Then the cost to reclaim a heap block is

$$\frac{c_1 R + c_2 H}{H - R}$$

*Stop and Copy*

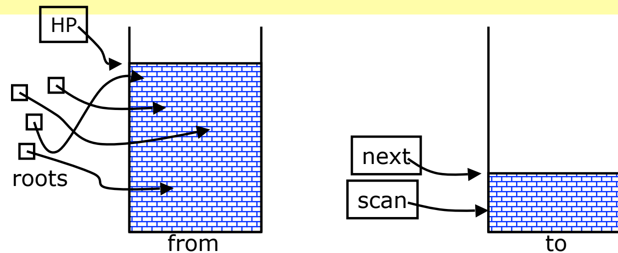
## Copying collection

Basic idea:

- two heaps, called *from-space* and *to-space*
- allocate heap blocks in the from-space, from low memory to high memory
- when from-space is exhausted, trace the pointer graph (starting from the roots) and *copy* all reachable data to to-space
  - leave a forwarding address in the from-space
- when all done copying, *flip*, so that to-space is now from-space and vice-versa

First proposed by Fenichel & Yochelson in 1969

# Classical copying algorithm




```
GC() {
  scan = next = start of to-space
  for each root p {
    forward(p);
  }
  while (scan < next) {
    *scan = forward (scan++);
  }
  flip();
}
```

```
forward(p) {
  if (p in from-space)
    if (*p in to-space)
      return *p
    else {
      *next = *p;
      *p = next++;
      return *p;
    }
  else return p;
}
```

# Copying collection is fast

Note that heap allocation is simply:

- increment the heap pointer, HP
- check whether it is in bounds, and if not, invoke GC

What is the cost to reclaim a heap block?

- assume cost c to trace a reachable block

$$\frac{cR}{H-R}$$

*What is the limit, as $H \rightarrow \infty$?*

# Notes

Like mark-and-sweep, we still need to find roots and distinguish pointers from non-pointers

How to do the heap check?

- overflow method
  - let M=max heap address, B=base heap address
  - then $HP_i$=maxint-(M-B)
- page-protection method
  - usually not precise for most OS's

*Generation Scavenging*

# Generational GC

Two observations by Hewitt in 1987:

- most heap-allocated objects die young
- newer objects usually point to older objects

Basic idea of generational GC:

- a small pair of heaps, called the *nursery*
- if copying collection in the nursery is ineffective, then copy to a bigger pair of heaps, called the *tenured space*

# Nursery and tenured spaces

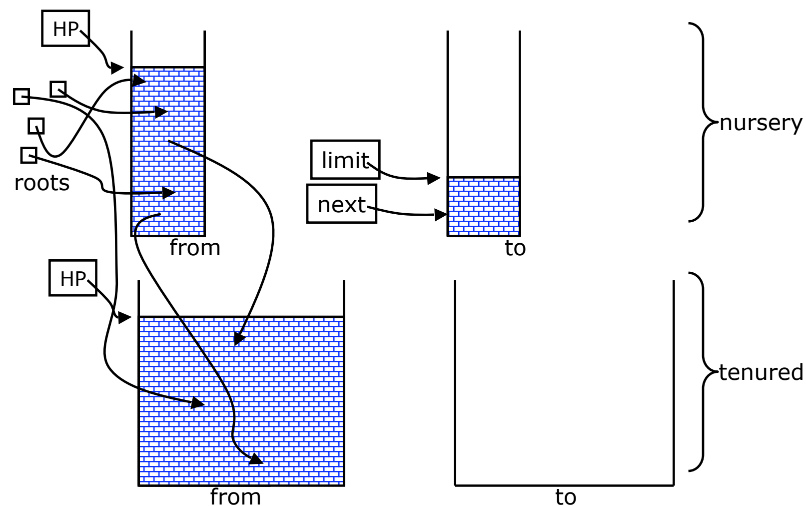
The nursery is very small (<1MB), and mostly garbage (because objects die young)

- therefore, copying collection is fast

If objects get promoted to the tenured space, they are unlikely to point to nursery objects

- therefore, nursery collection can (mostly) ignore objects in the tenured space

# Nursery and tenured spaces




# Some details

GC in the nursery is called "minor"; in the tenured space it is called "major"

Can set a percentage limit for the tenure decision:

- when nursery is X% full after minor GC, then copy to tenured space

It is also possible to have more than two generations, though tends not to be profitable

## The write barrier

The main problem with generational GC occurs when/if pointers in the tenured space point to objects in the nursery

- how can this happen?

We need to trace these pointers during minor collections

- maintain a *store list*, which contains the addresses of every heap block that has been modified
- this means that every write to the heap must update the store list

## In practice

In practice, generational GC is incredibly effective, even for imperative languages

# *Concurrent GC*

## Concurrent GC

Baker in 1981 devised an extension to copying collection to make it concurrent, in the sense that every call to alloc() would be guaranteed to execute less than a (small) constant number of instructions

Basic idea:

- every time a pointer is dereferenced, copy its heap block, plus a few more heap blocks, to the to-space
  - flip if there is nothing left in from-space

## Read barrier

Baker’s algorithm requires extra work for every read of a heap pointer

This is incredibly expensive

But we avoid the long pause times of the simple copying collection algorithm

Later improvements eliminate the read barrier, but with some complications

- Nettles and O’Toole
- Cheng and Blelloch

# *Conservative GC*

## Conservative GC

“Conservative GC” is a misnomer, as all garbage collection algorithms are conservative

But some algorithms are inexact in the sense that they don’t try to do an accurate trace of the pointer graph

The Boehm-Weiser collector that we have provided for your use is just such a garbage collector

## Boehm-Weiser GC

Basic idea:

- Put the heap in high memory
  - so, few program integers are valid heap addresses
- Assume all values that could be aligned heap addresses are in fact heap addresses
- Instrument malloc() to keep track of what has been allocated
- Then, do mark-and-sweep
  - why not do copying collection?

## Boehm-Weiser

As long as no live heap block is pointed to only by an interior pointer, the mark-and-sweep will safely reclaim only garbage

Furthermore, this has the advantage that it will work with most C programs

However, it is fairly slow because of the use of mark-and-sweep

## Region-based mem mgmt

To be discussed in readings…