Automatic Memory Management
(a.k.a. Garbage Collection)

Lecture 17

Lecture Outine

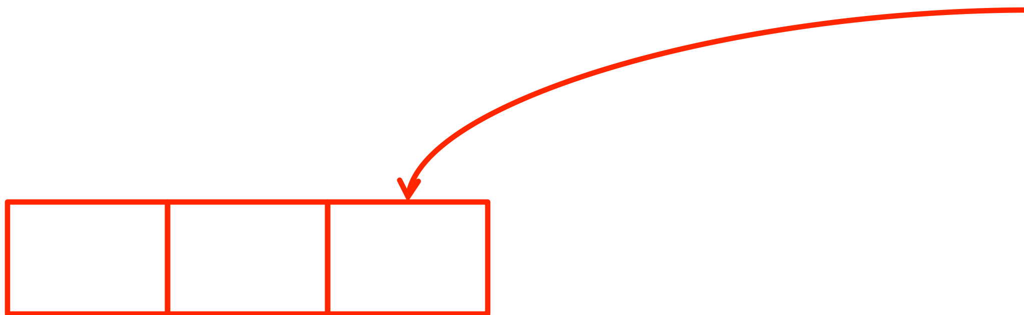
- Why Automatic Memory Management?
- Garbage Collection
- Three Techniques
  - Mark and Sweep
  - Stop and Copy
  - Reference Counting

Why Automatic Memory Management?

- Storage management is still a hard problem in modern programming
  - Manual allocation and deallocation is difficult…
- C and C++ programs have many storage bugs
  - forgetting to free unused memory (memory leak)
  - dereferencing a dangling pointer
  - overwriting parts of a data structure by accident
  - and so on...
- Storage bugs are hard to find
  - Often the last bugs found, and sometimes only found long after production code has shipped
  - Why? a bug can lead to a visible effect far away in time and program text from the source

Why Automatic Memory Management?

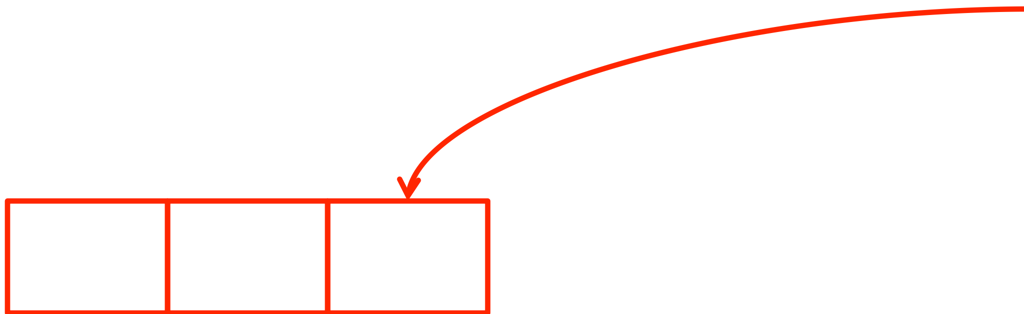
- Storage bugs are hard to find
  - Often the last bugs found, and sometimes only found long after production code has shipped
  - Why? a bug can lead to a visible effect far away in time and program text from the source
- How does this happen?



Assume we have an object of "red" class, and somewhere in the program there is a reference to some attribute of the object

Why Automatic Memory Management?

- Storage bugs are hard to find
  - Often the last bugs found, and sometimes only found long after production code has shipped
  - Why? a bug can lead to a visible effect far away in time and program text from the source
- How does this happen?



At some point the programmer frees the memory for the object, but forgets about the reference

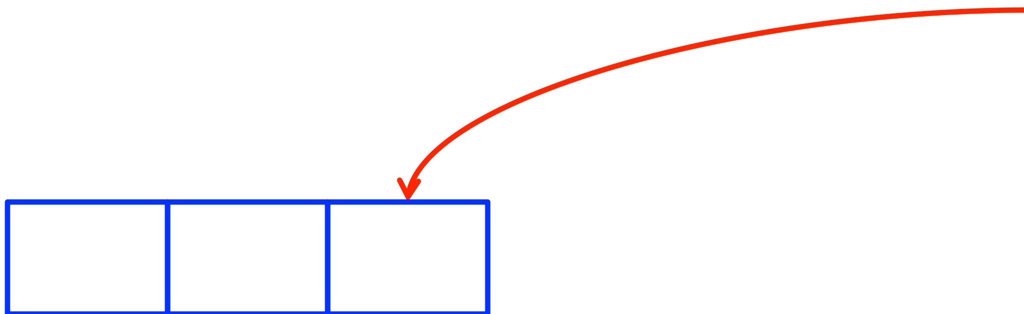
Why Automatic Memory Management?

- Storage bugs are hard to find
  - Often the last bugs found, and sometimes only found long after production code has shipped
  - Why? a bug can lead to a visible effect far away in time and program text from the source
- How does this happen?

At some point the programmer frees the memory for the object, but forgets about the reference

Why Automatic Memory Management?

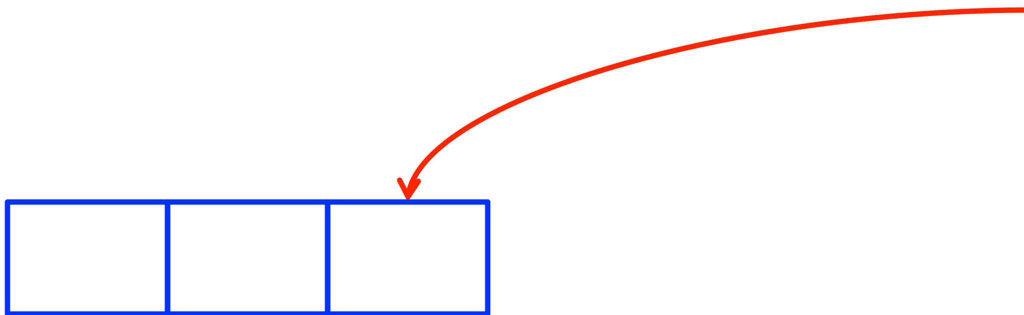
- Storage bugs are hard to find
  - Often the last bugs found, and sometimes only found long after production code has shipped
  - Why? a bug can lead to a visible effect far away in time and program text from the source
- How does this happen?



Some time later, the programmer requests memory allocation and is assigned the same memory, this time for an object of "blue" class

Why Automatic Memory Management?

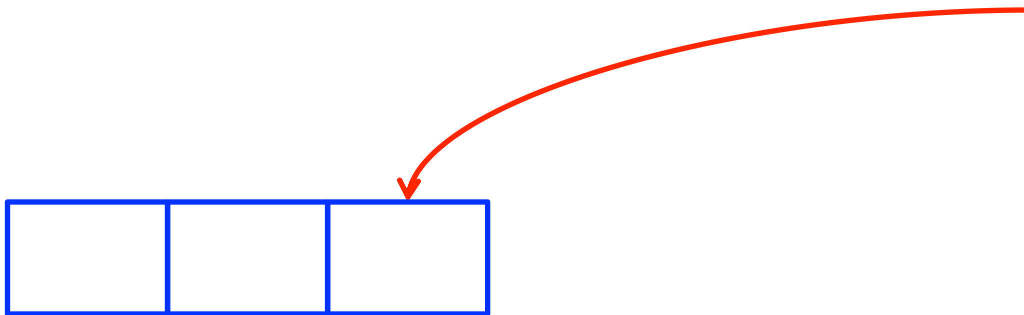
- Storage bugs are hard to find
  - Often the last bugs found, and sometimes only found long after production code has shipped
  - Why? a bug can lead to a visible effect far away in time and program text from the source
- How does this happen?



The piece of code that holds the pointer thinks it's still pointing to an object of type red. If it writes anything using the pointer, it is writing garbage into the blue object

Why Automatic Memory Management?

- Storage bugs are hard to find
  - Often the last bugs found, and sometimes only found long after production code has shipped
  - Why? a bug can lead to a visible effect far away in time and program text from the source
- How does this happen?



Later (possibly very much later) when information is read out of the blue object, there will be garbage there, that will likely cause my program to crash. Not good.

Automatic Memory Management History

- This is an old problem
  - Studied since the 1950s for LISP

- There are well-known techniques for completely automatic memory management

- Only became mainstream in the 1990s with popularity of Java
  - Prior to that no mainstream language that used automatic memory managment

The Basic Idea

- When an object is created, unused space is automatically allocated (by run-time system)
  - In Cool, new objects are created by new X
- After a while there is no more unused space
  - At which point must reclaim space to allow allocation of new objects
- Some space is occupied by objects that will never be used again
  - Of course this may not be the case, but it's likely that it is
  - This space can be freed to be reused later

Type Safety and Memory Management

- Can types prevent errors in programs with manual allocation and deallocation of memory?
  - some fancy type systems (linear types) were designed for this purpose but they complicate programming significantly
- Currently, if you want type safety then you must use automatic memory management

Automatic Memory Management

- This is an old problem:
  - studied since the 1950s for LISP

- There are well-known techniques for completely automatic memory management

- Became mainstream with the popularity of Java

The Basic Idea (Cont.)

- How can we tell whether an object will "never be used again"?
  - in general, impossible to tell
  - we will use heuristics

- Observation: a program can use only the objects that it can find:
- Ex.

$$\text{let } x : A \leftarrow \text{new } A \text{ in } \{ x \leftarrow y; \ldots \}$$

  - After $x \leftarrow y$ there is no way to access the newly allocated object (Why?)

Garbage

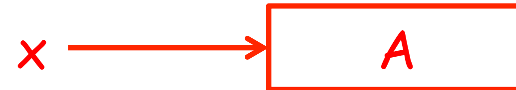
- An object x is *reachable* if and only if:
  - a register contains a pointer to x, or
  - another reachable object y contains a pointer to x

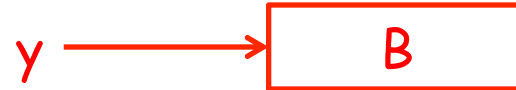
- Thus you can find all reachable objects by starting from registers and following all the pointers
  - And the pointers in objects that are pointed to, etc.
  - Note many things can be reachable by multiple paths
- Any unreachable object can never be used
  - such objects are *garbage*

Reachability is an Approximation

- Consider the program:

  x ← new A;
  y ← new B
  x ← y;
  if alwaysTrue() then x ← new A else x.foo() fi





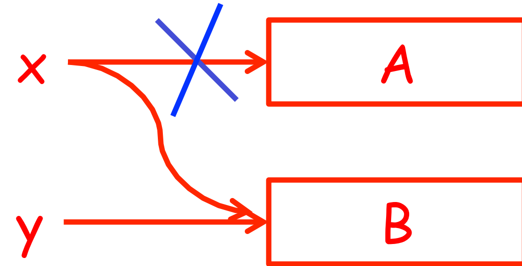
- Just after x ← y, but before if statement
  - assuming y becomes dead …
  - the first object A is unreachable
  - the object B is reachable (through x)
  - thus B is not garbage and is not collected
    - but object B is never going to be used

Reachability is an Approximation

- Consider the program:

```
x ← new A;
y ← new B
x ← y;
if alwaysTrue() then x ← new A else x.foo() fi
```

assume garbage collected here

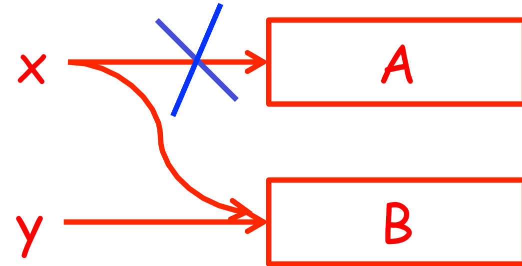


- Just after x ← y, but before if statement
  - assuming y becomes dead …
    - never used after this use of y
  - the first object A is unreachable
  - the object B is reachable (through x)
  - thus B is not garbage and is not collected
    - but object B is never going to be used
    - Because if assigns new pointer to x, and y is dead

Reachability is an Approximation

- Consider the program:

```
x ← new A;
y ← new B
x ← y;
if alwaysTrue() then x ← new A else x.foo() fi
```



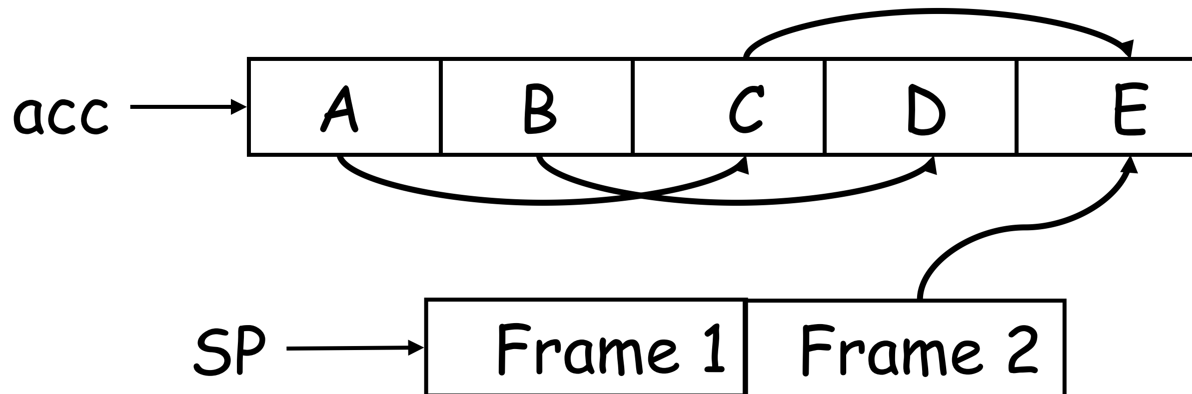
- Thus reachability is an approximation
  - What we're really interested in is objects that will not be used again
  - Some objects deemed reachable will actually never be used again

Tracing Reachable Values in coolc

- In coolc, the only register is the accumulator
  - it points to an object
  - and this object may point to other objects, etc.
  - So we have to trace through all of these

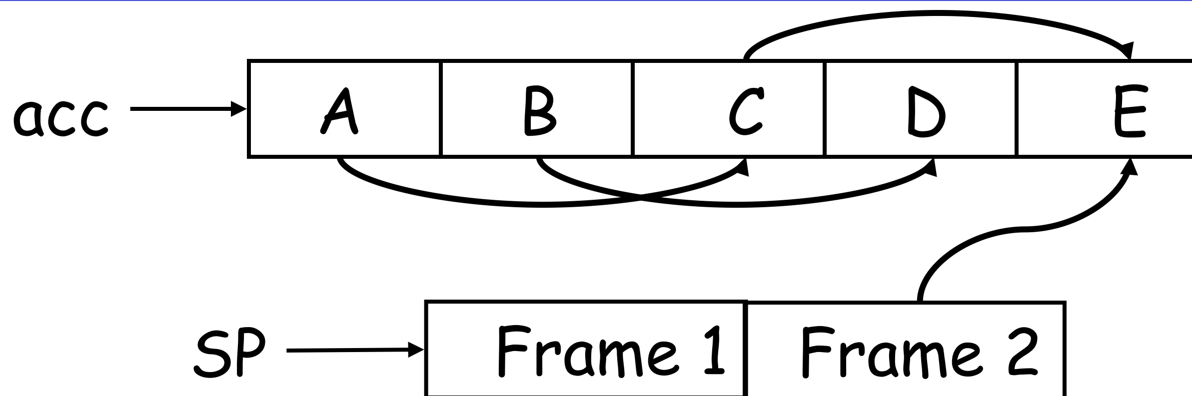
- The stack is more complex
  - each stack frame contains pointers
    - e.g., method parameters
  - each stack frame also contains non-pointers
    - e.g., return address
  - if we know the layout of the frame we can find the pointers in it
    - And of course compiler decides on the layout of the frame, so it does know the layout
    - It needs to keep track of which frame entries are pointers

A Simple Example



- In coolc we start tracing from acc and SP
  - These are called the *roots*: in garbage collection terminology, the registers from which you begin tracing out all the reachable objects
- Note B and D are unreachable from acc and stack
  - Thus we can reuse their storage

A Simple Example



- In coolc we start tracing from acc and SP
  - These are called the *roots*: in garbage collection terminology, the registers from which you begin tracing out all the reachable objects
- Note B and D are unreachable from acc and stack
  - Note also that just because object has a pointer to it, doesn't mean it's reachable (e.g., D). Only pointers to it could be from unreachable objects

Elements of Garbage Collection

- Every garbage collection scheme has the following steps
  1. Allocate space as needed for new objects
  2. When space runs out:
     a) Compute what objects might be used again (generally by tracing objects reachable from a set of "root" registers)
     b) Free the space used by objects not found in (a)

- Some strategies perform garbage collection before the space actually runs out

Three General Garbage Collection Techniques

- Mark and sweep
- Stop and copy
- Reference counting

Mark and Sweep

- When memory runs out, GC executes two phases
  - the mark phase: traces reachable objects
  - the sweep phase: collects garbage objects

- Every object has an extra bit: the *mark* bit
  - reserved for memory management
    - used only by garbage collector
  - initially the mark bit for each object is set to 0
  - set to 1 for the reachable objects in the mark phase

The Mark Phase

```
let todo = { all roots }
while todo ≠ ∅ do
    pick v ∈ todo
    todo ← todo - { v }
    if mark(v) = 0 then        (* v is unmarked yet *)
        mark(v) ← 1
        let v1,...,vn be the pointers contained in v
        todo ← todo ∪ {v1,...,vn}
    fi
od
```

"worklist-based" algorithm, with worklist initially the set of all the roots

The Mark Phase

```
let todo = { all roots }
while todo ≠ ∅ do
    pick v ∈ todo
    todo ← todo - { v }
    if mark(v) = 0 then        (* v is unmarked yet *)
        mark(v) ← 1
        let v1,...,vn be the pointers contained in v
        todo ← todo ∪ {v1,...,vn}
    fi
od
```

note if v is already marked we do nothing except drop it from todo list

The Sweep Phase

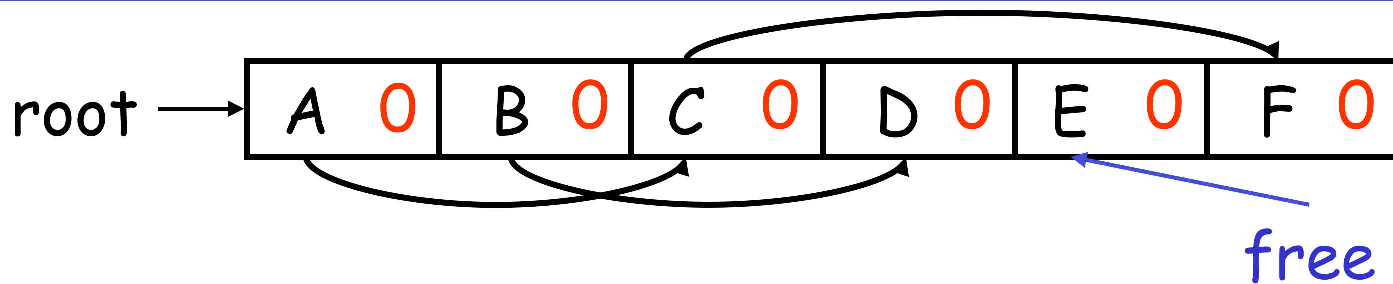
- The sweep phase scans the heap looking for objects with mark bit 0
  - these objects were not visited in the mark phase
  - they are garbage
- Any such object is added to the free list
- The objects with a mark bit 1 have their mark bit reset to 0
  - So they are ready for the next round of garbage collection

The Sweep Phase (Cont.)

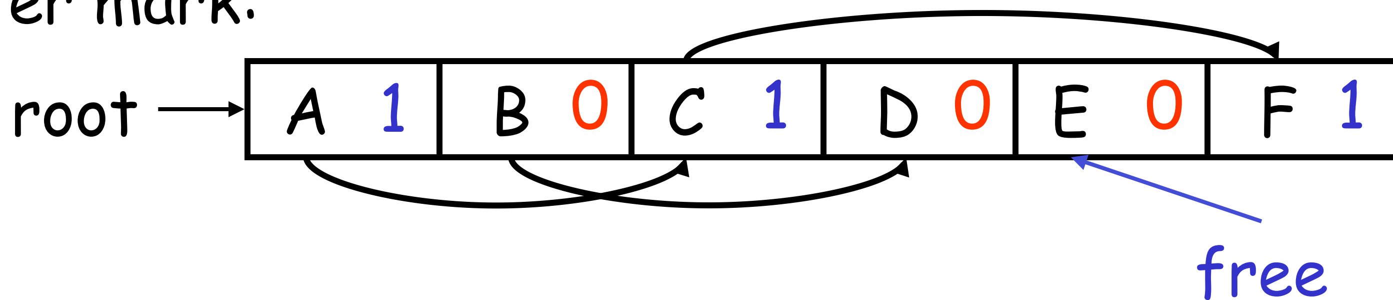
```
(* sizeof(p) is the size of block starting at p *)
p ← bottom of heap
while p < top of heap do
   if mark(p) = 1 then
      mark(p) ← 0
   else
      add block p...(p+sizeof(p)-1) to freelist
   fi
   p ← p + sizeof(p)
od
```

This is the purpose of the size field in cool objects: so garbage collector can know size of object during sweep phase

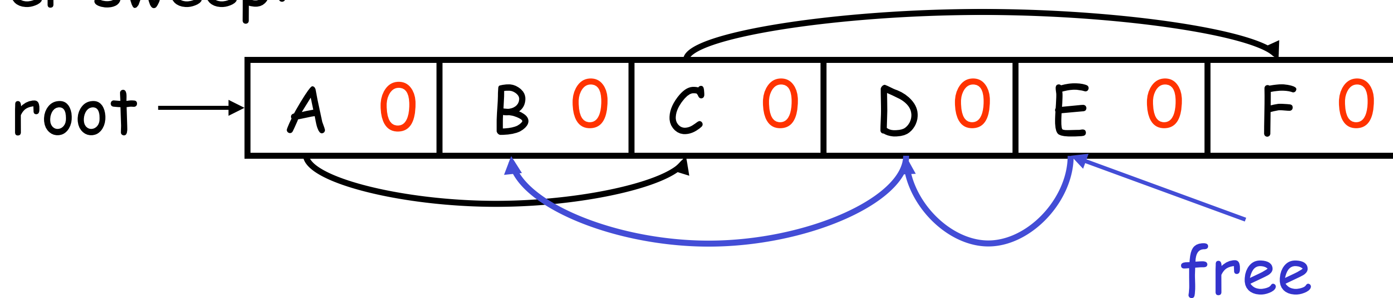
Mark and Sweep Example



After mark:



After sweep:



assume only a single root, and free list a linked list of available space (here, only one object on it at start)

Simple, right?

Well, maybe not...

Details

- While conceptually simple, this algorithm has a number of tricky details
  - typical of GC algorithms

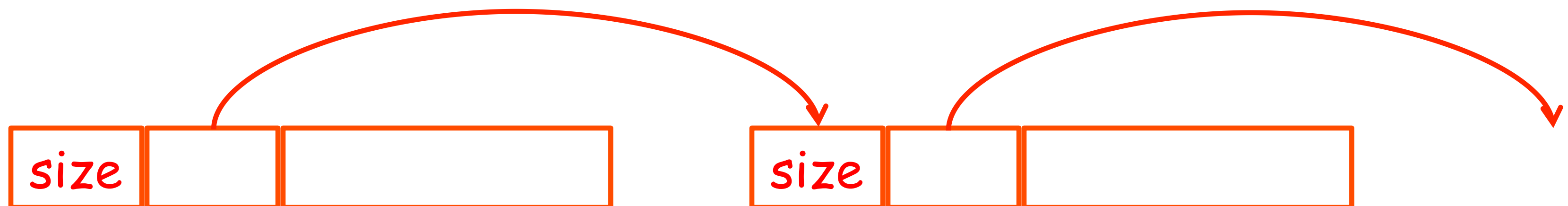
- A serious problem with the mark phase (also typical of GC algorithms)
  - it is invoked when we are out of space
  - yet it needs space to construct the todo list
  - the size of the todo list is unbounded so we cannot reserve space for it a priori
    - and in practice it can be fairly large

Mark and Sweep: Details

- The todo list is used as an auxiliary data structure to perform the reachability analysis
- There is a trick that allows the auxiliary data to be stored in the objects themselves
  - *pointer reversal*: when a pointer is followed it is reversed to point to its parent
  - Allows us to track which objects in the heap still need to be processed without having to use any extra space

Mark and Sweep: Details

- Similarly, the free list is stored in the free objects themselves
  - E.g., the first part of a block of memory stores the size of the block of memory, the second part stores a pointer to the next block of free space on the list



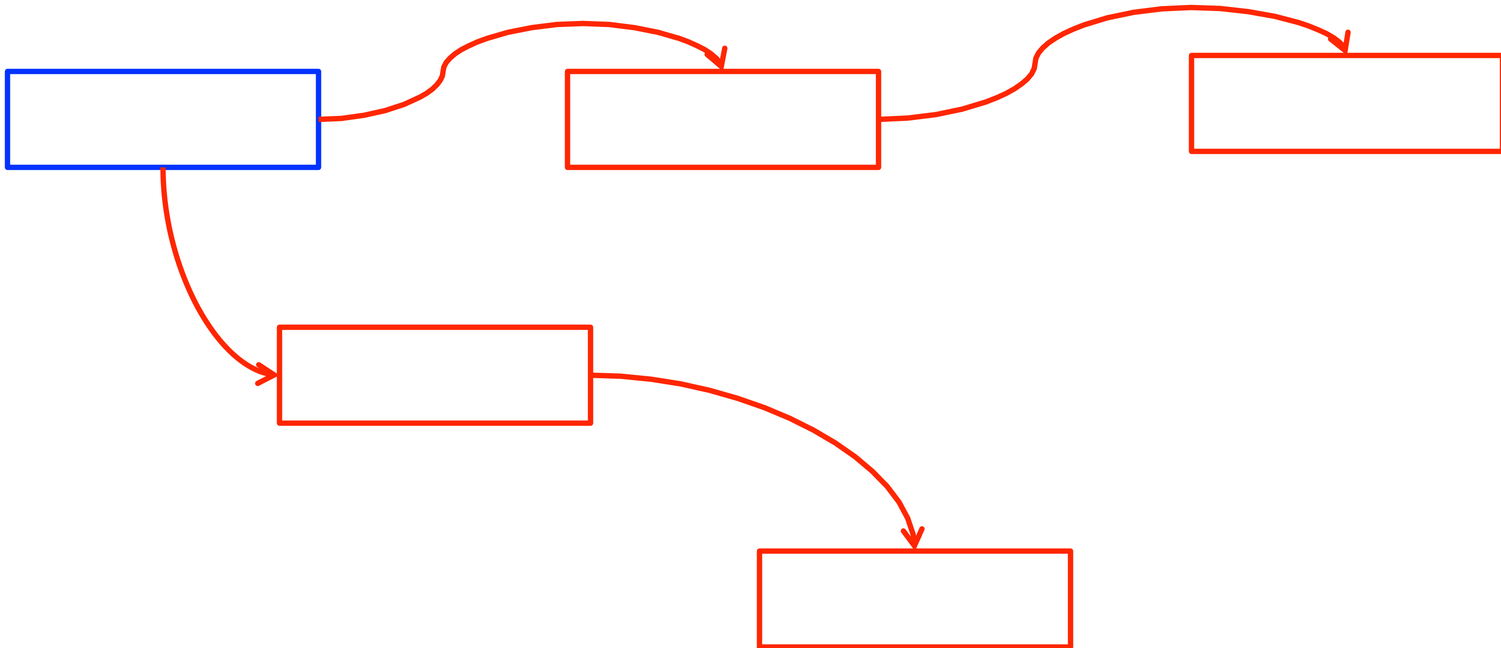
Pointer Reversal Example

- So assume we have some objects, and we want to track reachability, but without keeping a todo list in a separate data structure

root

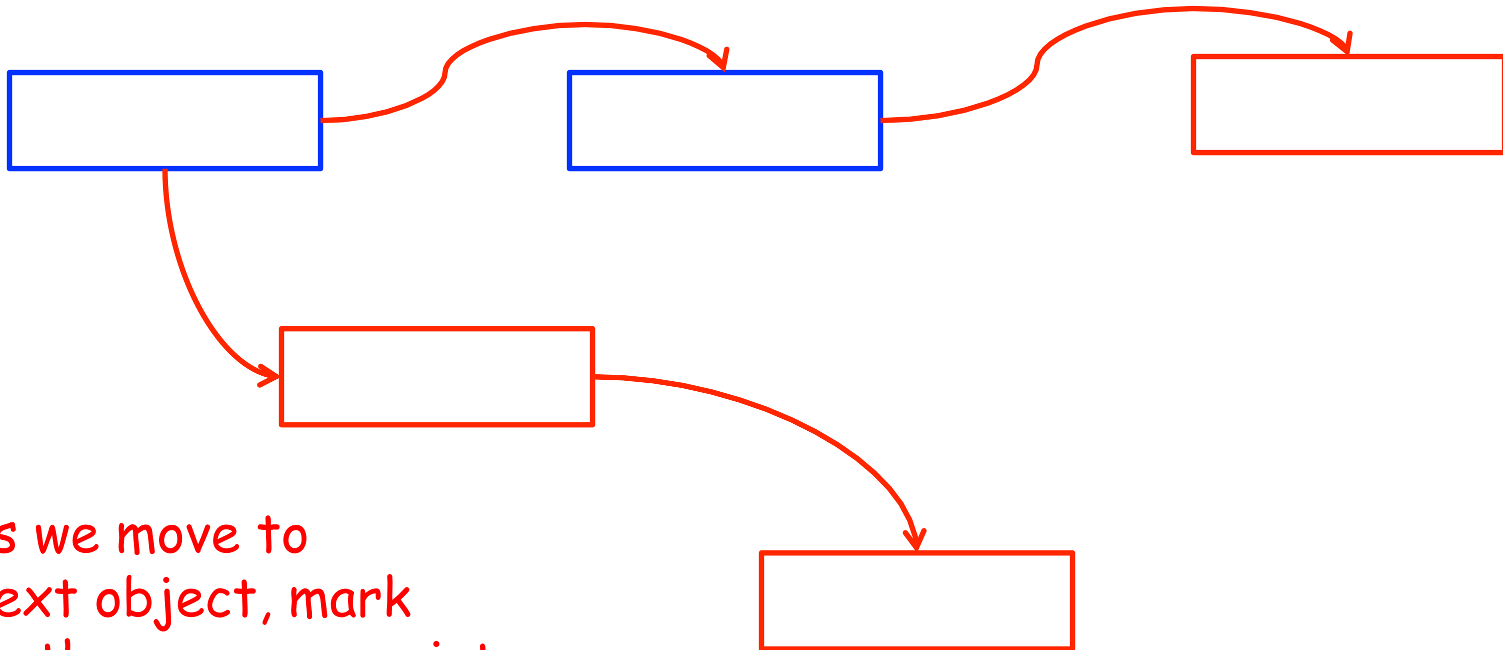
Pointer Reversal Example

- So assume we have some objects, and we want to track reachability, but without keeping a todo list in a separate data structure



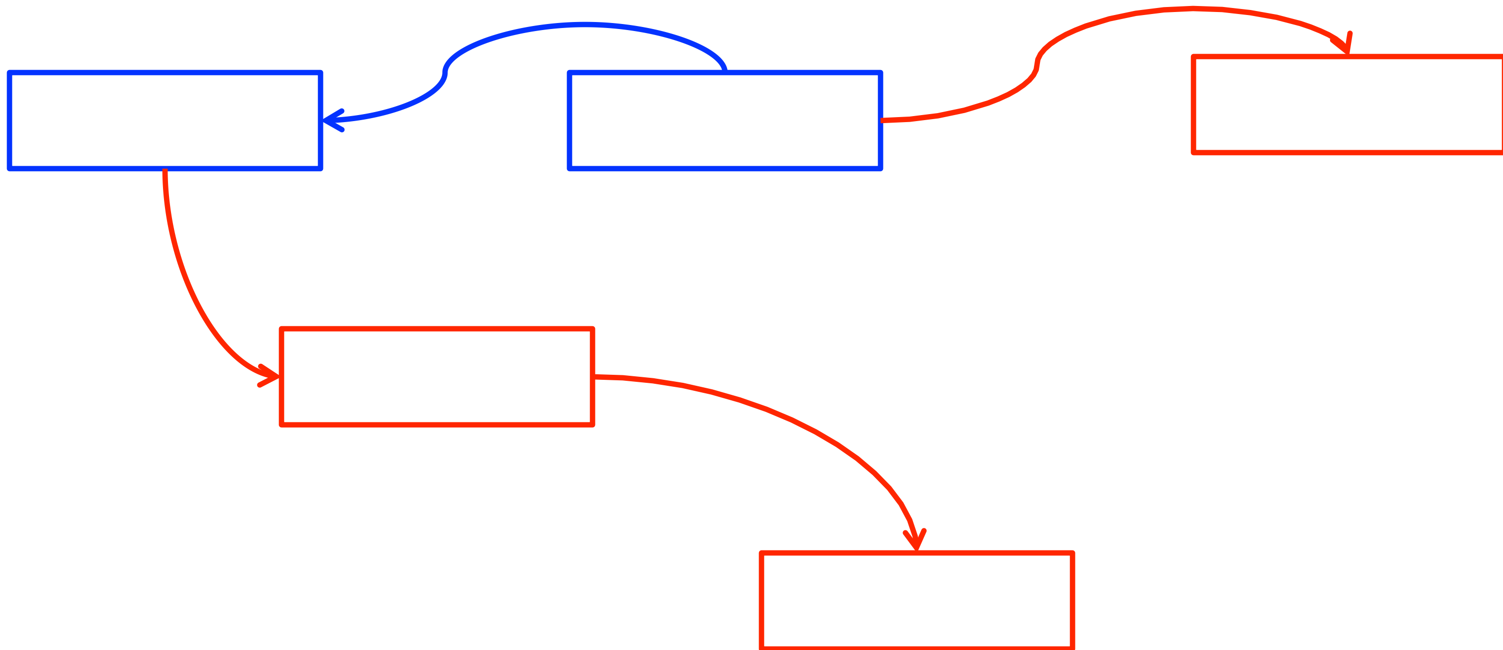
Pointer Reversal Example

- So assume we have some objects, and we want to track reachability, but without keeping a todo list in a separate data structure



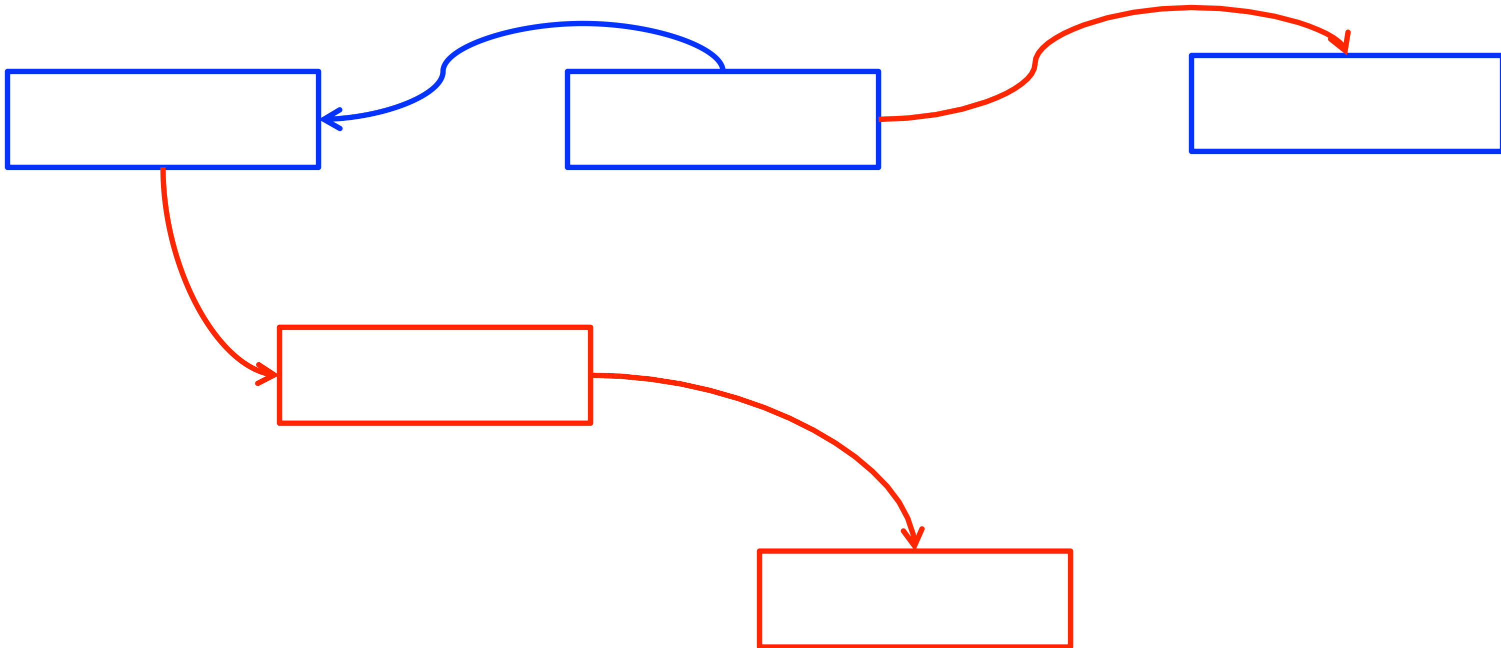
Pointer Reversal Example

- So assume we have some objects, and we want to track reachability, but without keeping a todo list in a separate data structure



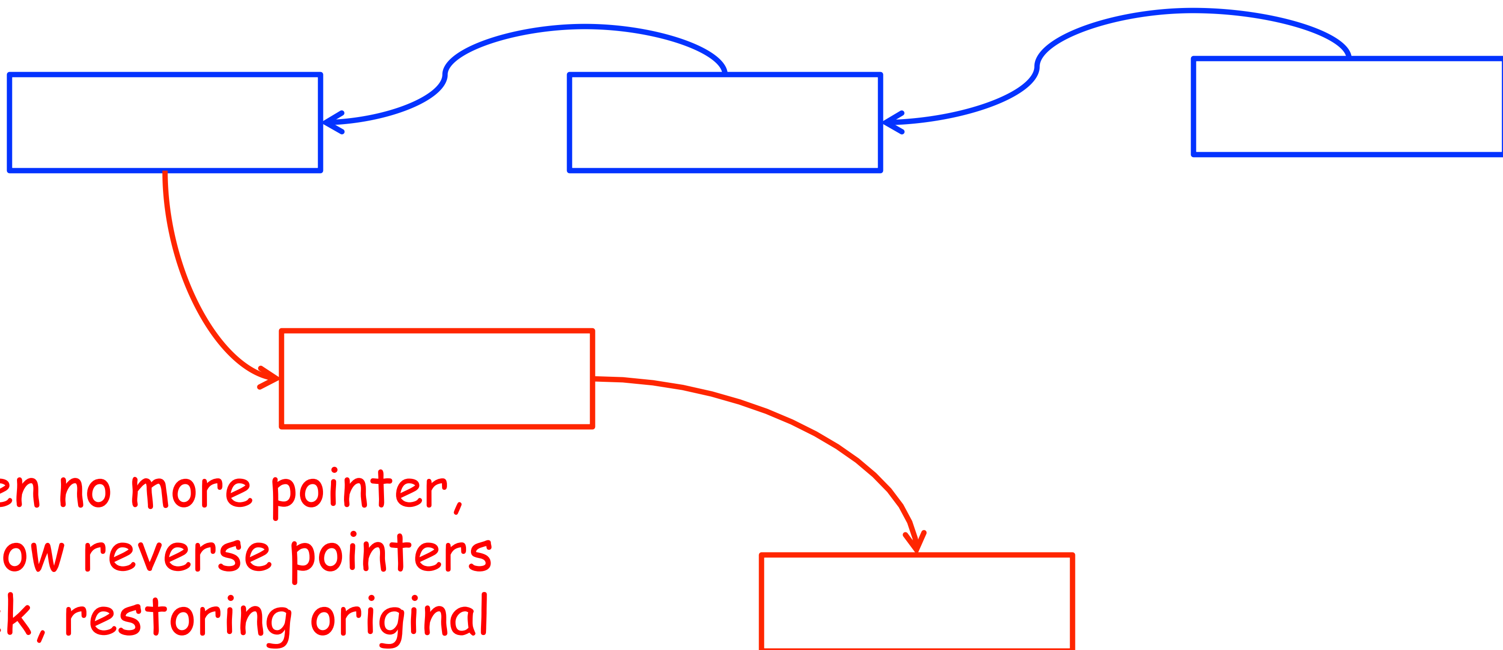
Pointer Reversal Example

- So assume we have some objects, and we want to track reachability, but without keeping a todo list in a separate data structure



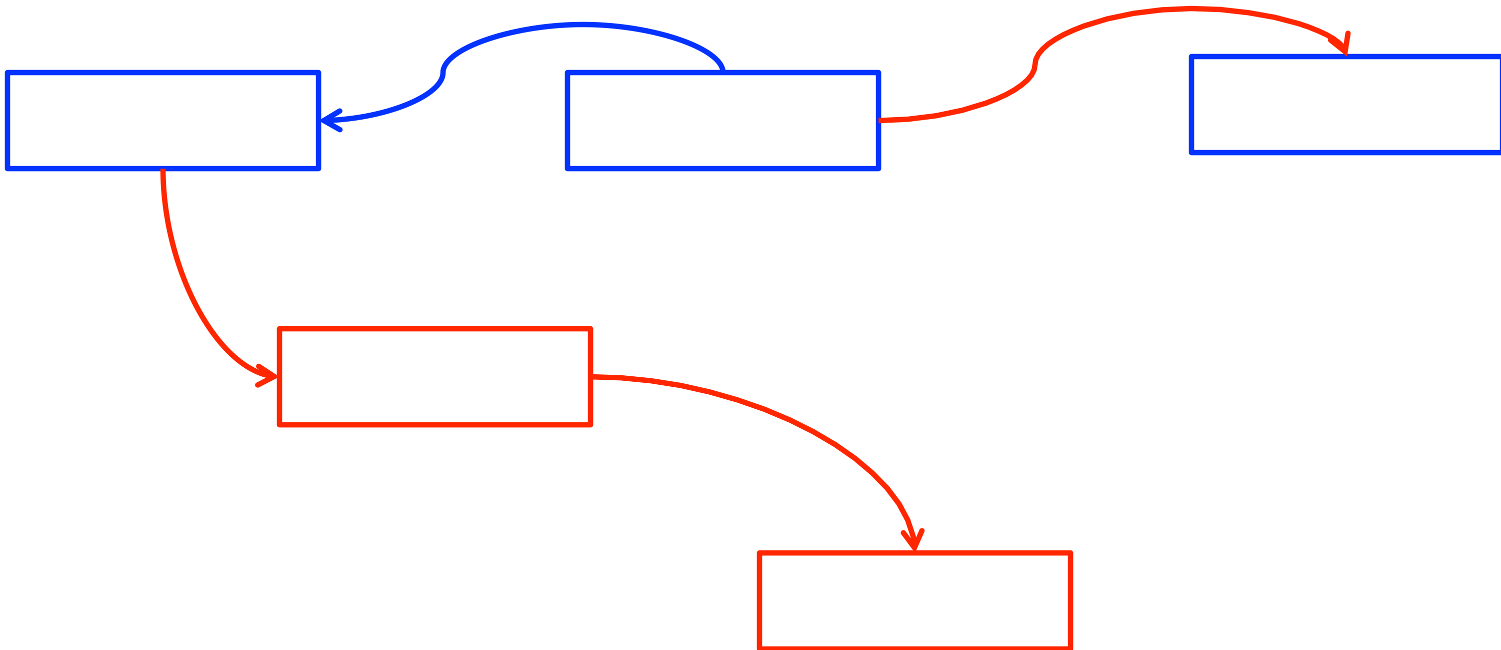
Pointer Reversal Example

- So assume we have some objects, and we want to track reachability, but without keeping a todo list in a separate data structure



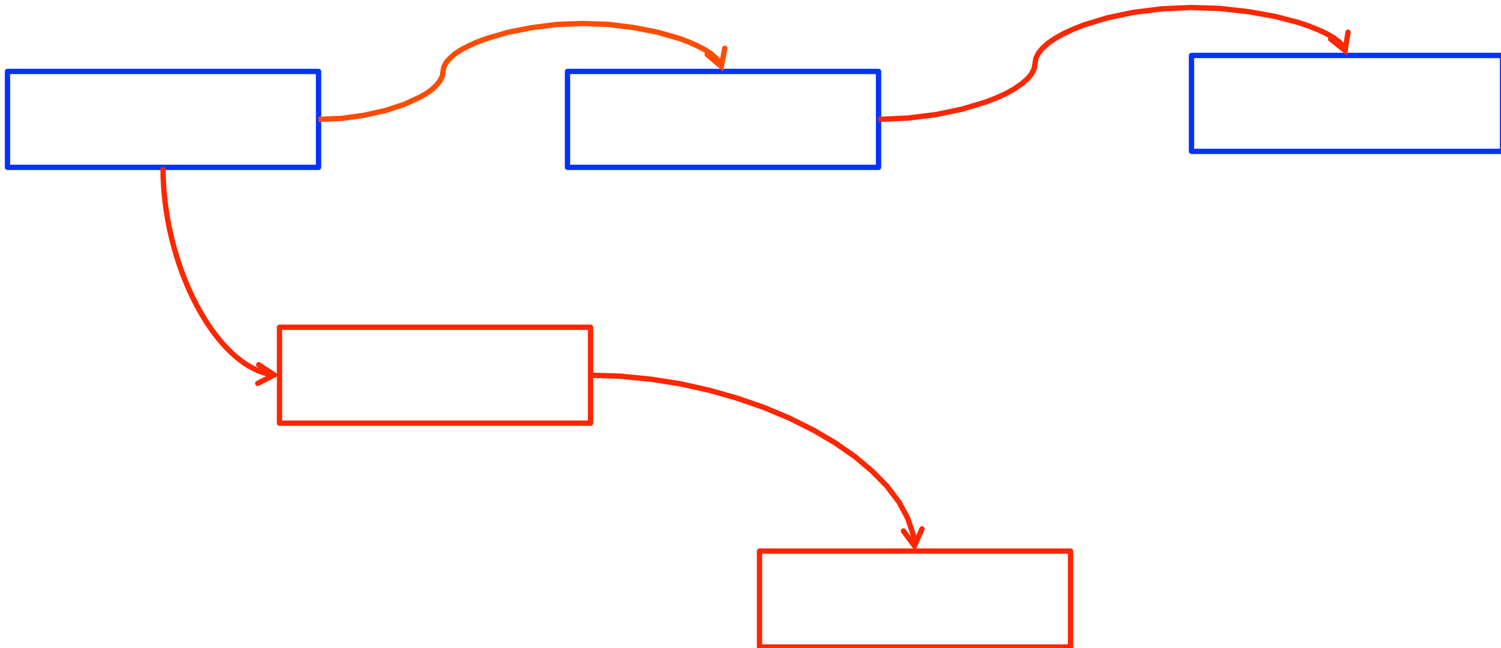
Pointer Reversal Example

- So assume we have some objects, and we want to track reachability, but without keeping a todo list in a separate data structure



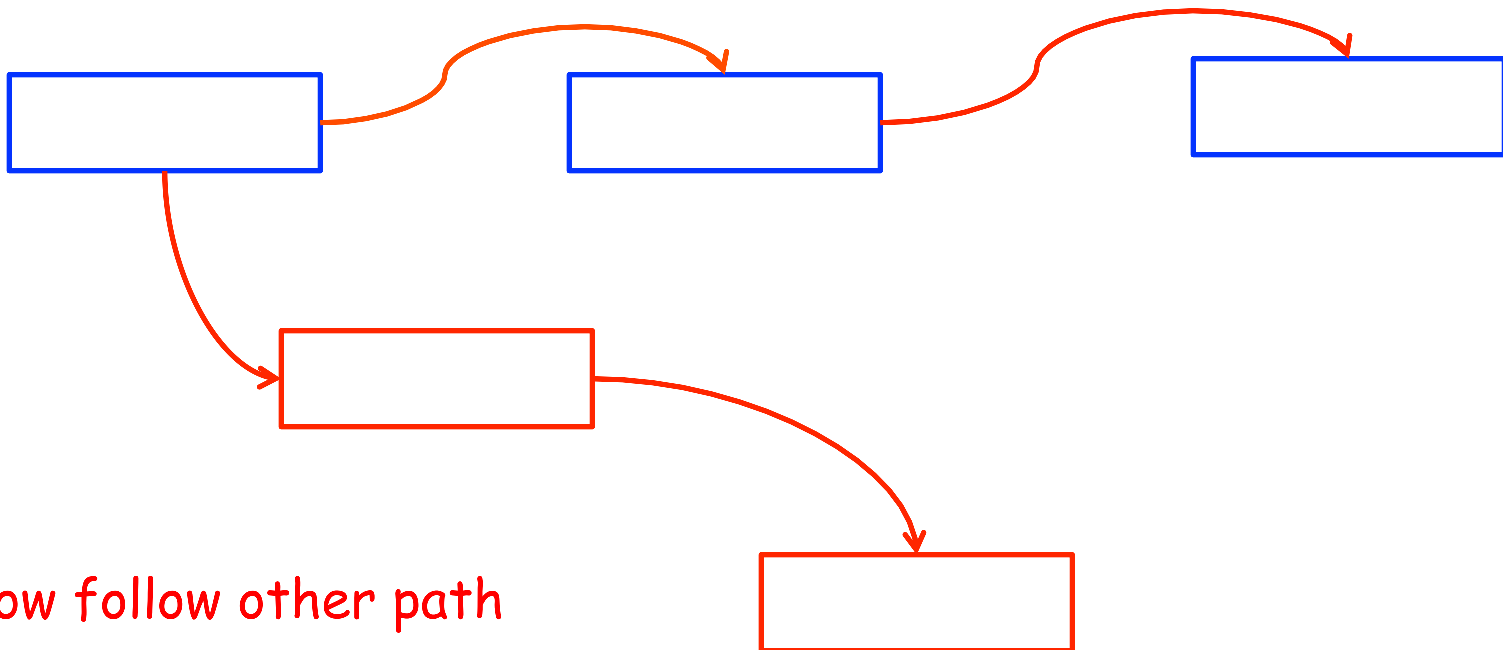
Pointer Reversal Example

- So assume we have some objects, and we want to track reachability, but without keeping a todo list in a separate data structure



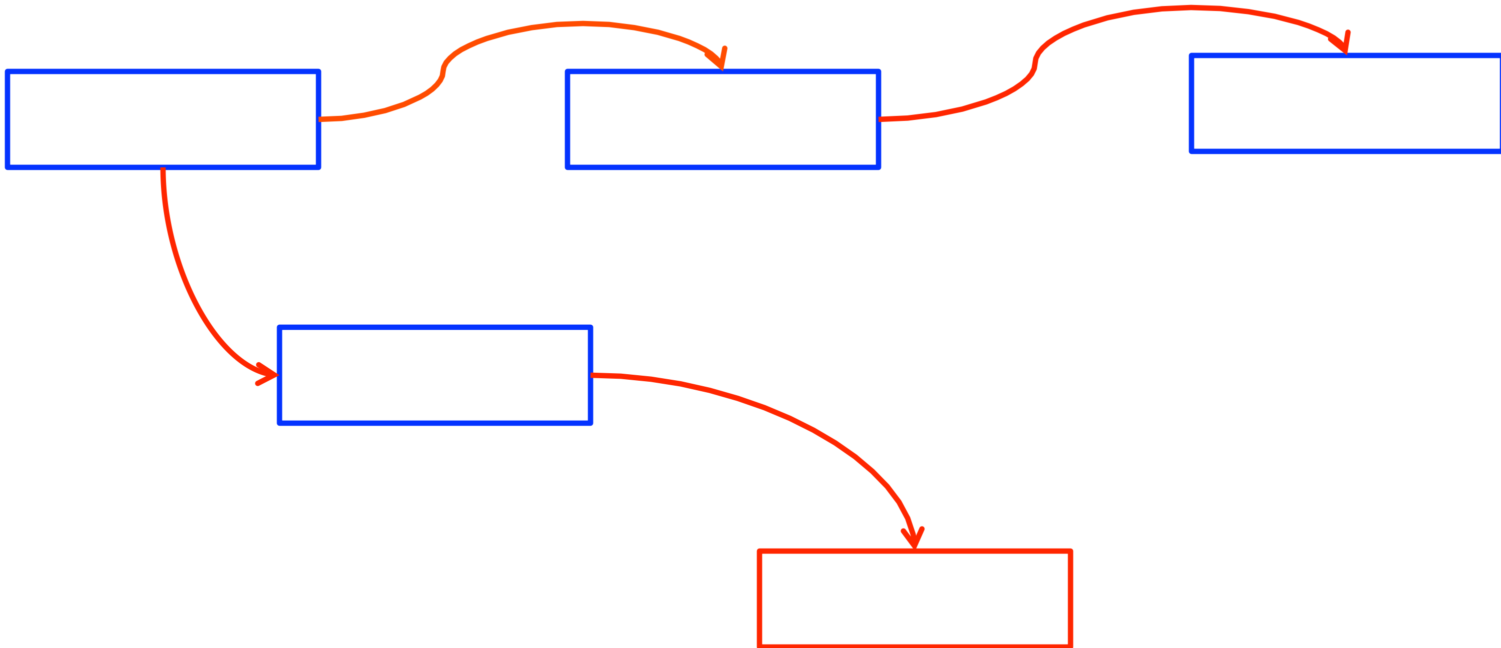
Pointer Reversal Example

- So assume we have some objects, and we want to track reachability, but without keeping a todo list in a separate data structure



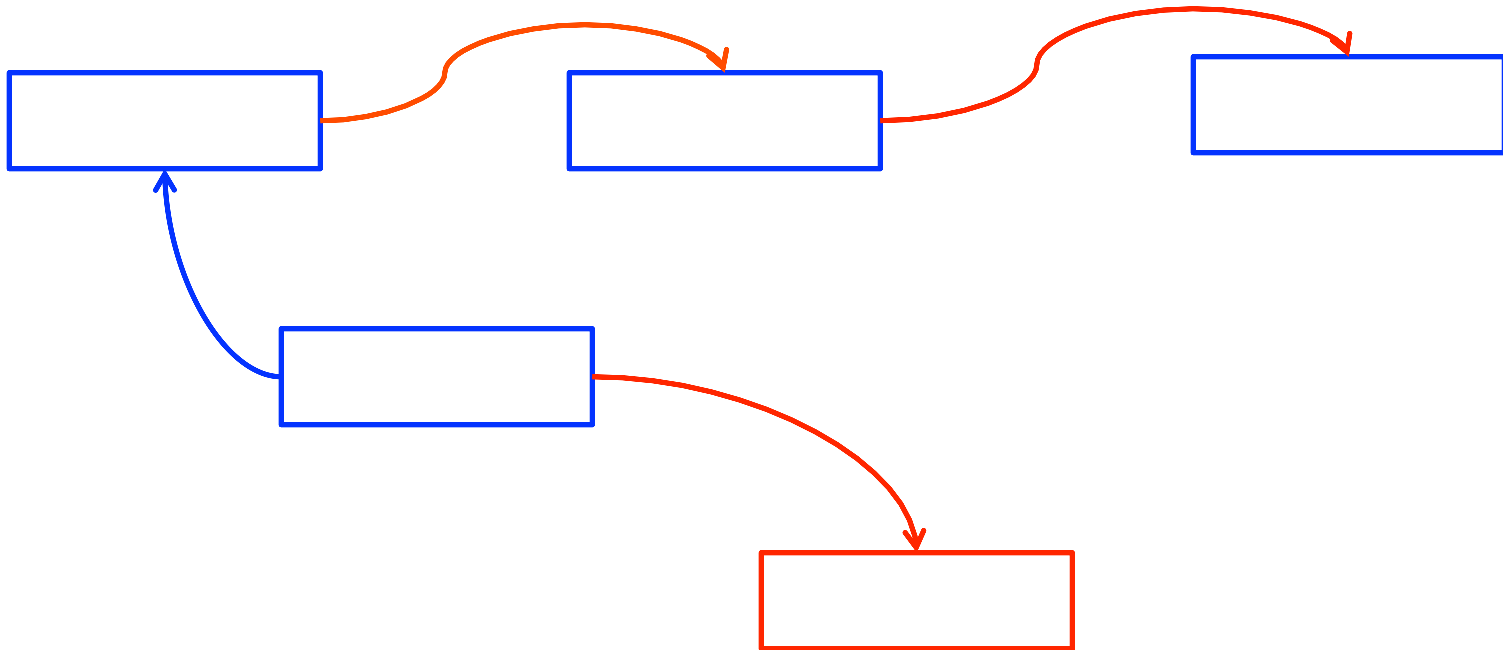
Pointer Reversal Example

- So assume we have some objects, and we want to track reachability, but without keeping a todo list in a separate data structure



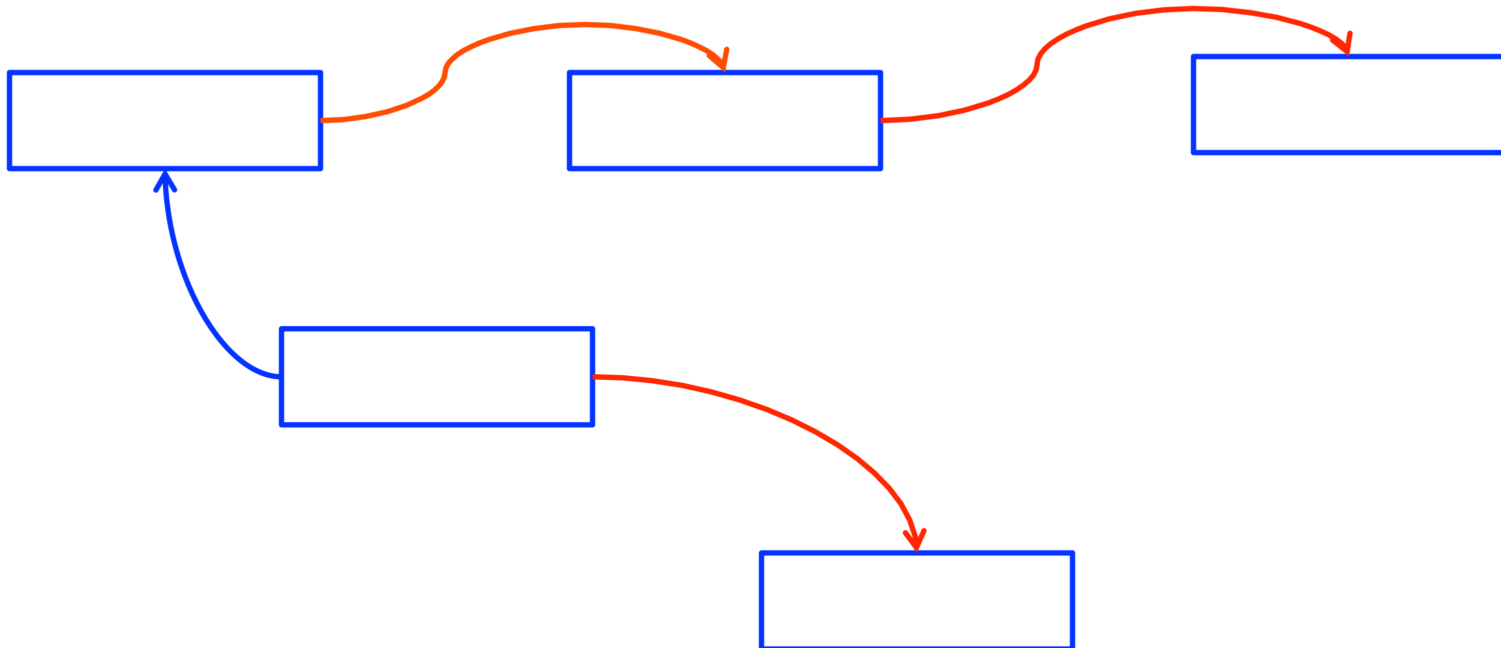
Pointer Reversal Example

- So assume we have some objects, and we want to track reachability, but without keeping a todo list in a separate data structure



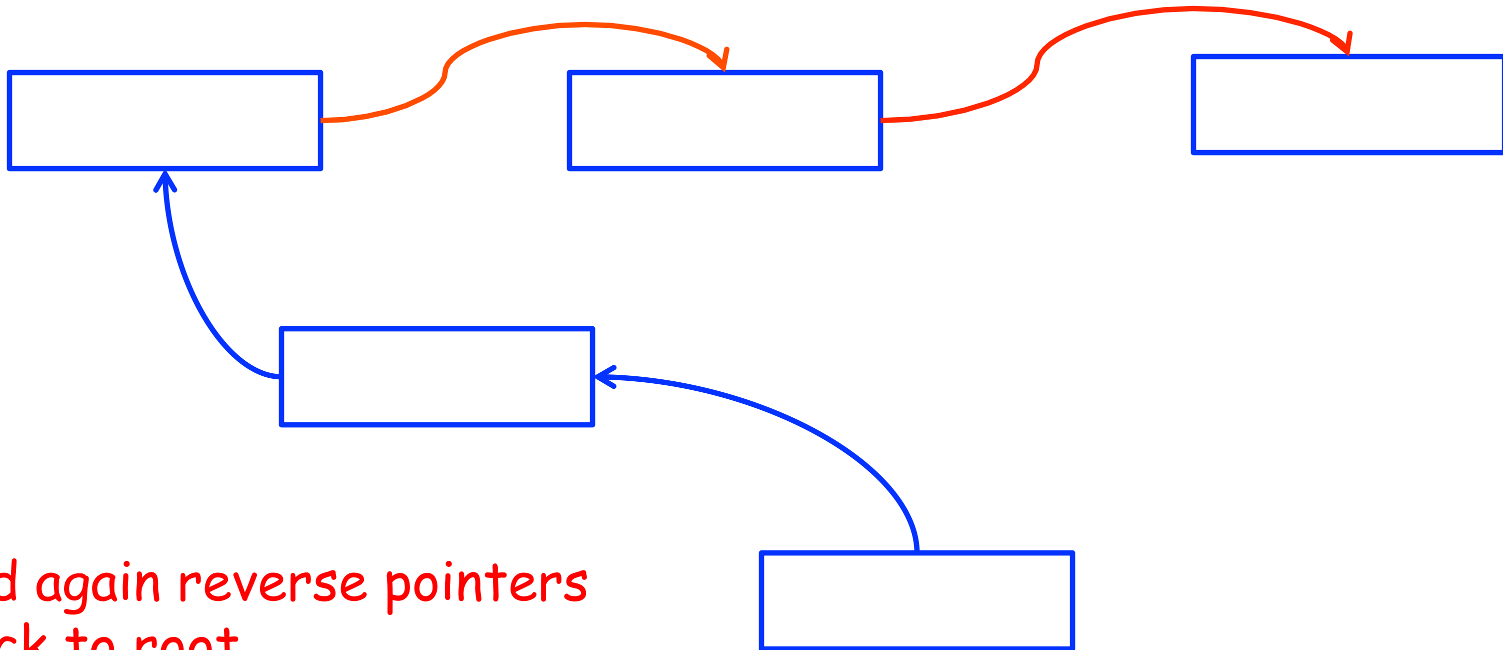
Pointer Reversal Example

- So assume we have some objects, and we want to track reachability, but without keeping a todo list in a separate data structure



Pointer Reversal Example

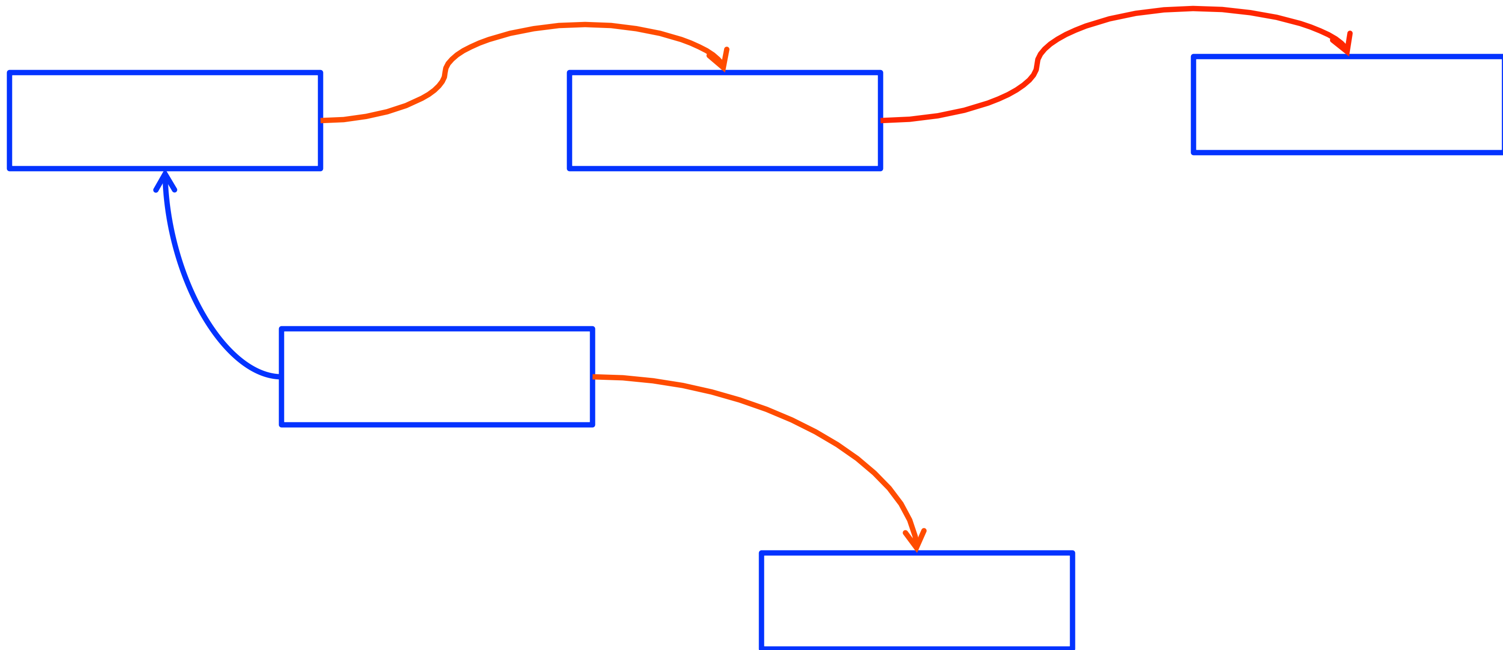
- So assume we have some objects, and we want to track reachability, but without keeping a todo list in a separate data structure



and again reverse pointers back to root

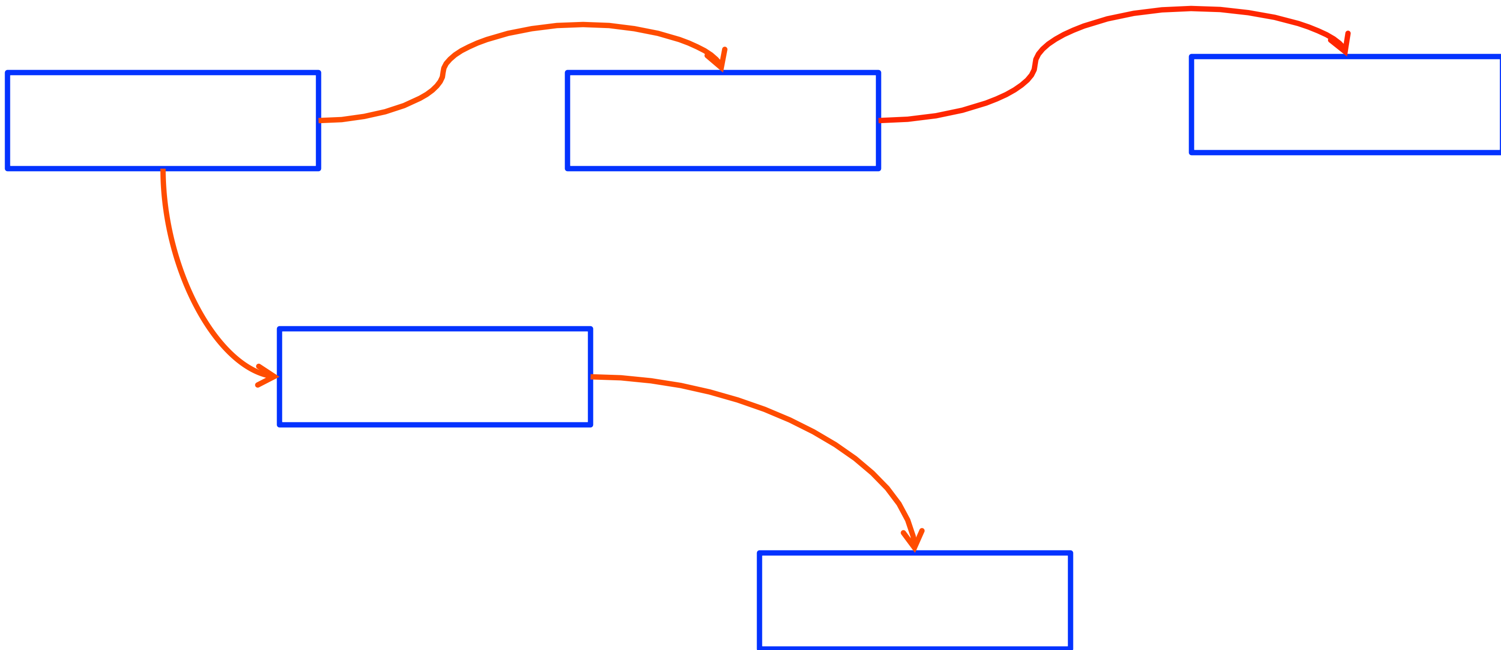
Pointer Reversal Example

- So assume we have some objects, and we want to track reachability, but without keeping a todo list in a separate data structure



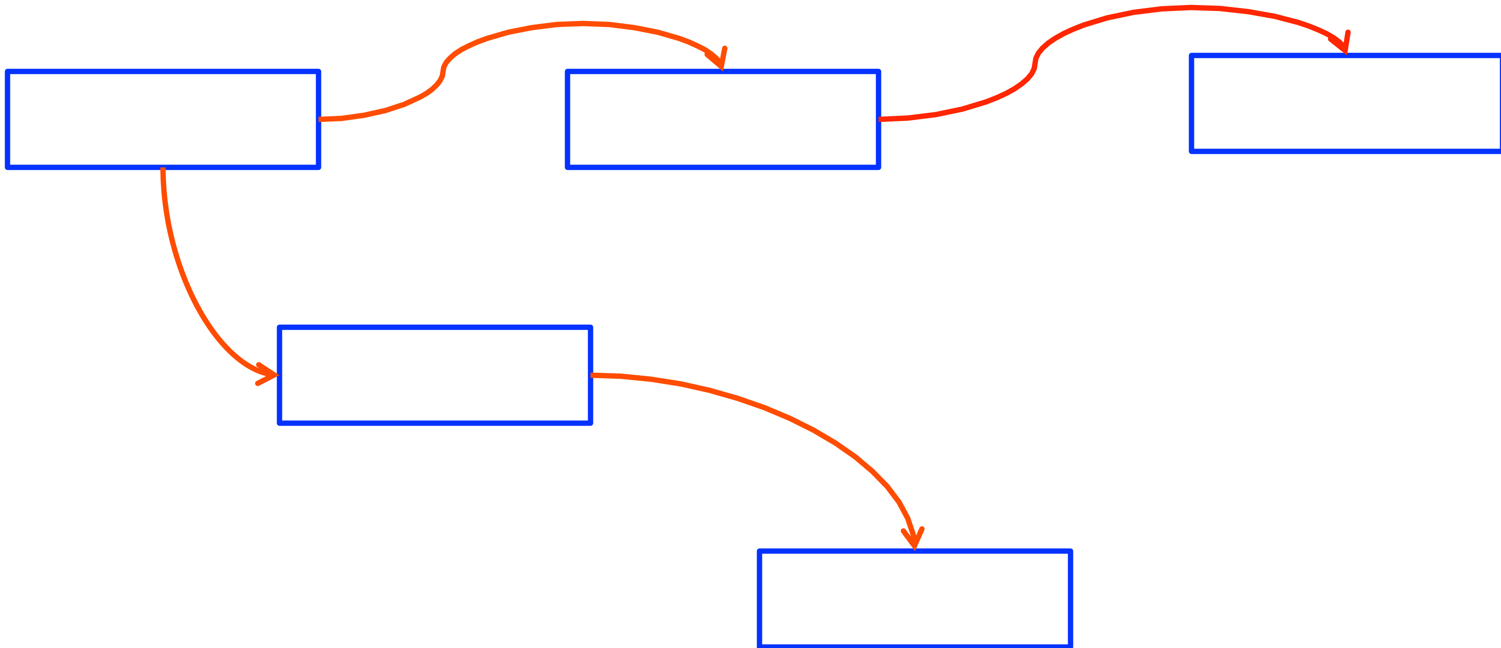
Pointer Reversal Example

- So assume we have some objects, and we want to track reachability, but without keeping a todo list in a separate data structure



Pointer Reversal Example

- Essentially, what pointer reversal does is help us maintain the stack for a depth-first search of the graph
  - Reverse pointers allow us to do the backtracking



Evaluation of Mark and Sweep

- Space for a new object is allocated from the free list
  - have to be sure to pick a block (from the free list) that is large enough to hold the object that we're allocating
  - an area of the necessary size is allocated from that block
    - the left-over is put back in the free list
- Because of this, mark and sweep can fragment the memory
  - Might end up with lots of little bits of left over memory, maybe none of which are big enough to actually hold an object
  - So important for mark and sweep to merge free blocks whenever possible (e.g., if two adjacent blocks end up on free list)

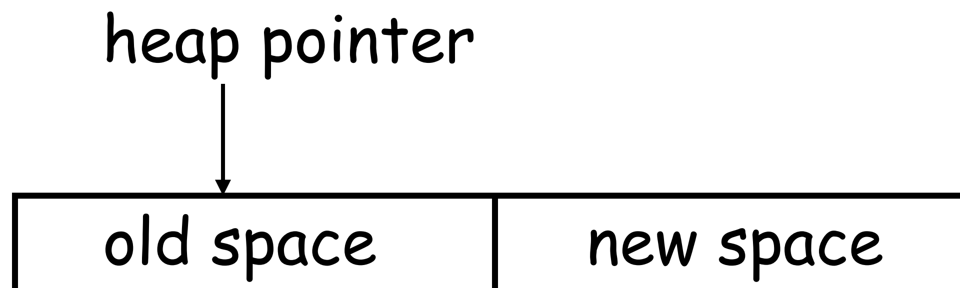
Evaluation of Mark and Sweep

- Space for a new object is allocated from the free list
  - have to be sure to pick a block (from the free list) that is large enough to hold the object that we're allocating
  - an area of the necessary size is allocated from that block
    - the left-over is put back in the free list

- Advantage (perhaps biggest advantage of mark and sweep): objects are not moved during GC
  - no need to update the pointers to objects
  - thus it's actually possible to adapt mark and sweep for languages like C and C++
    - Can't move objects in C and C++ because pointer address is part of their semantics
    - And people have built conservative mark and sweep garbage collectors for C and C++

Another Technique: Stop and Copy

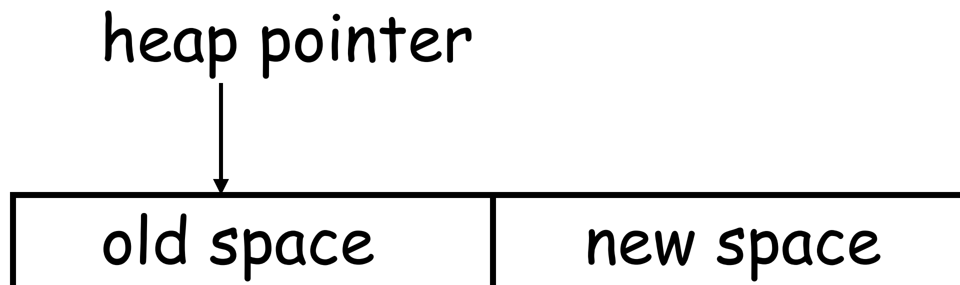
- In this technique, memory is organized into two areas
  - old space: used for allocation
  - new space: only used as a reserve for GC



- So first issue: program can only use half the space
  - Some more sophisticated S & C methods allow more, but still significant reduction

Another Technique: Stop and Copy

- The way allocation works:
  - heap pointer points to the next free word in the old space
    - allocation just advances the heap pointer
    - Everything to the left of the heap pointer is already in use
  - pointer progresses through old space as objects are allocated
    - So one advantage of S & C is a very simple and fast allocation strategy

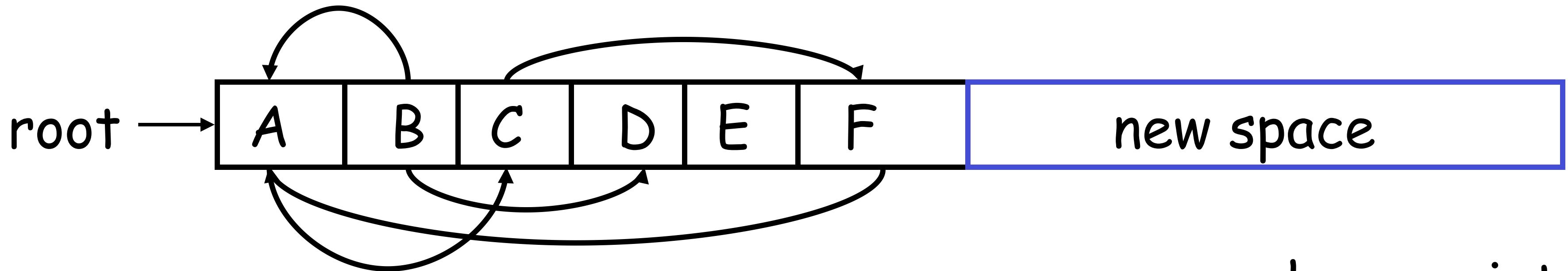


Stop and Copy Garbage Collection

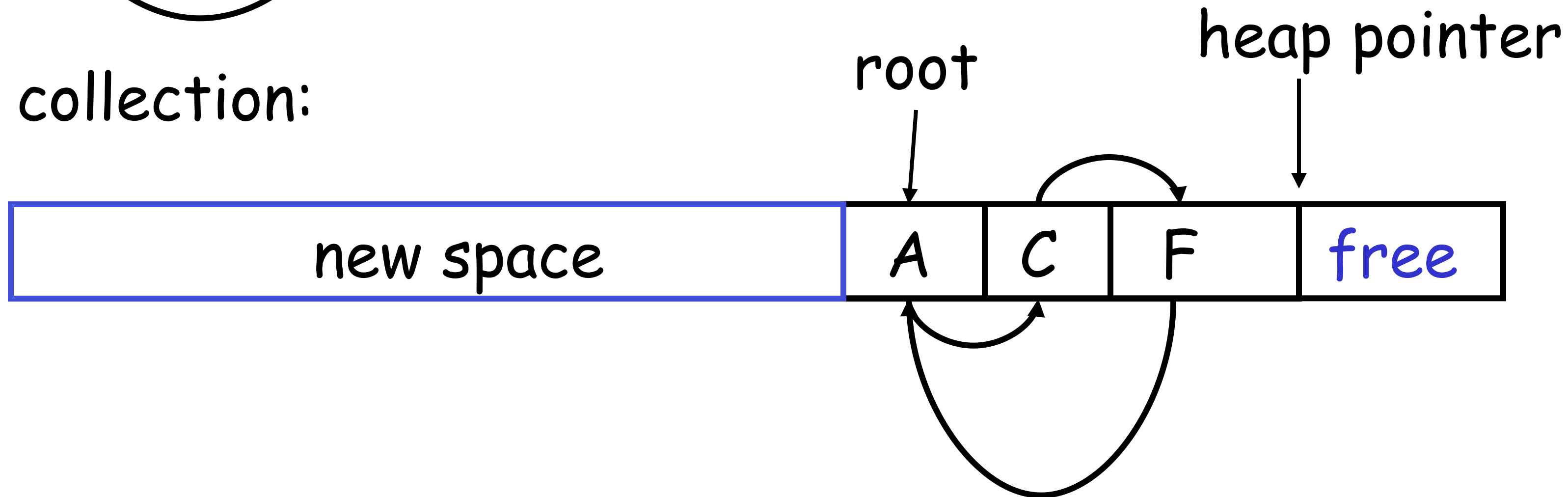
- GC starts when the old space is full
- Copies all reachable objects from old space into new space
  - beauty of this: *garbage is left behind*
  - after the copy phase the new space uses less space than the old one before the collection
    - because you left the garbage behind
- After the copy the roles of the old and new spaces are reversed and the program resumes

Example of Stop and Copy Garbage Collection

Before collection:



After collection:



Note that when copying, we must also change pointers, so, e.g., the copy of A points to the copy of C.

Implementation of Stop and Copy

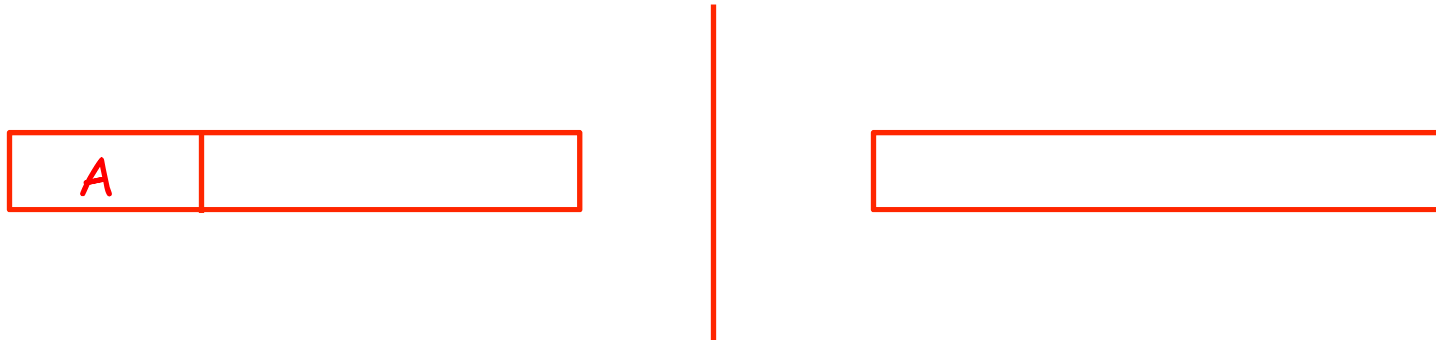
- We need to find all the reachable objects, as for mark and sweep

- As we find a reachable object we copy it into the new space
  - And we have to find and fix ALL pointers pointing to it!
  - It's not obvious how to do this, because when you find an object, you can't see all the pointers that point into that object!

- Idea: As we copy an object we store in the old copy a *forwarding pointer* to the new copy
  - when we later reach an object with a forwarding pointer we know it was already copied

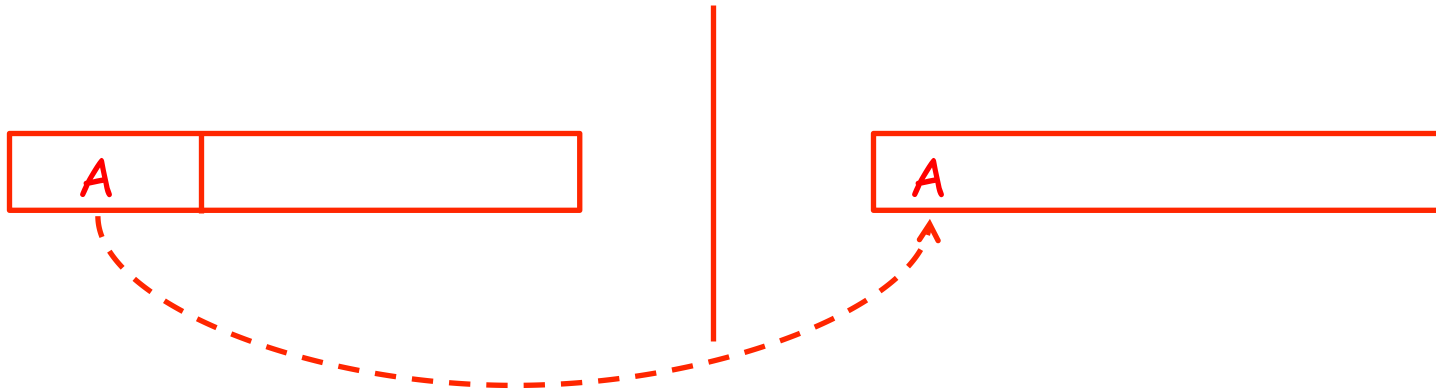
Implementation of Stop and Copy

- Idea: As we copy an object we store in the old copy a *forwarding pointer* to the new copy
  - when we later reach an object with a forwarding pointer we know it was already copied



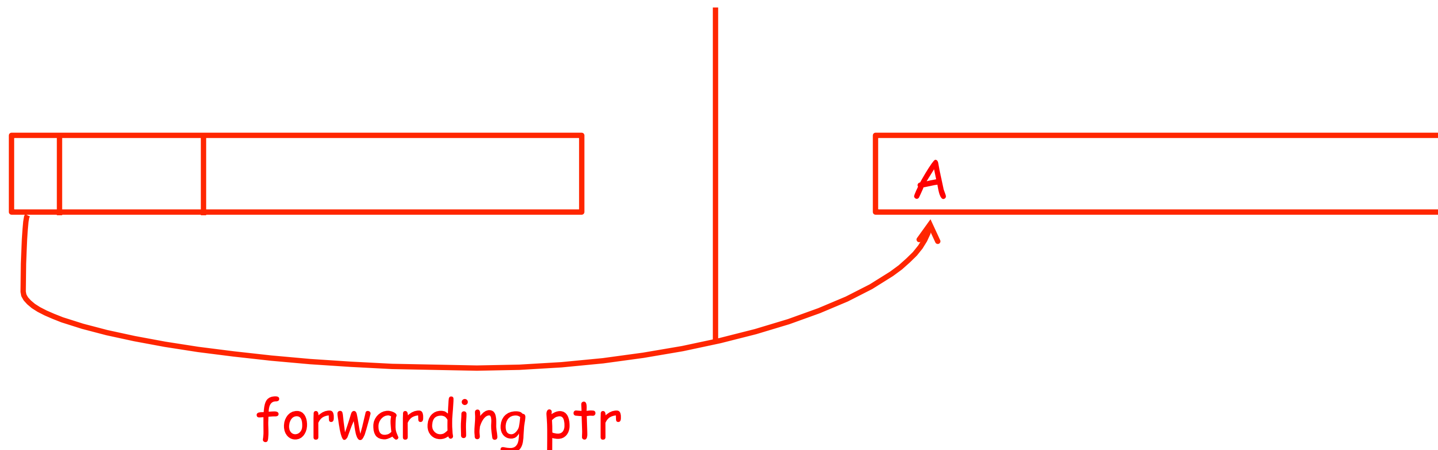
Implementation of Stop and Copy

- Idea: As we copy an object we store in the old copy a *forwarding pointer* to the new copy
  - when we later reach an object with a forwarding pointer we know it was already copied



Implementation of Stop and Copy

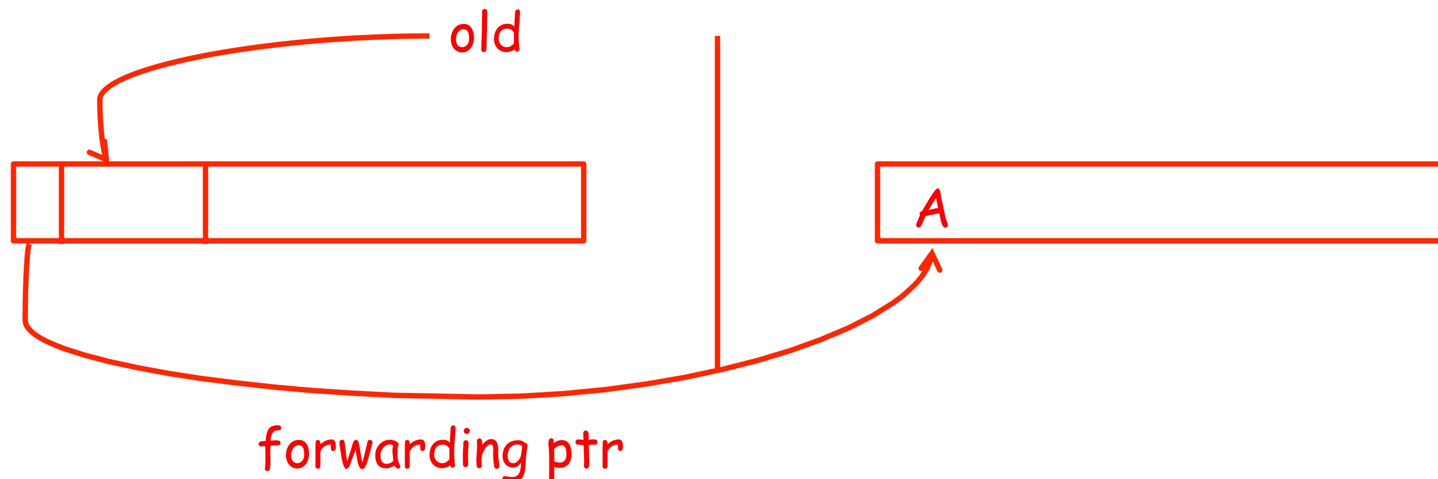
- Idea: As we copy an object we store in the old copy a *forwarding pointer* to the new copy
  - when we later reach an object with a forwarding pointer we know it was already copied



We place a forwarding pointer in a part of the old object (always same part of the object), along with a mark that indicates the object has been copied

Implementation of Stop and Copy

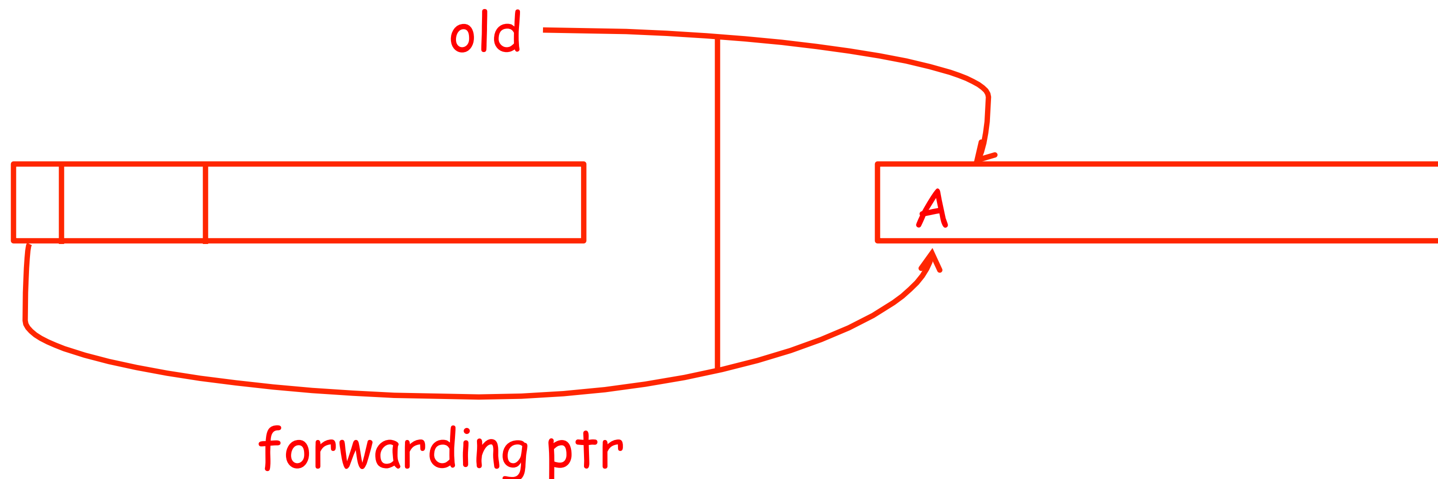
- Idea: As we copy an object we store in the old copy a *forwarding pointer* to the new copy
  - when we later reach an object with a forwarding pointer we know it was already copied



If later in GC we find a pointer to the old object, we follow the forwarding pointer, the new address to update the old pointer

Implementation of Stop and Copy

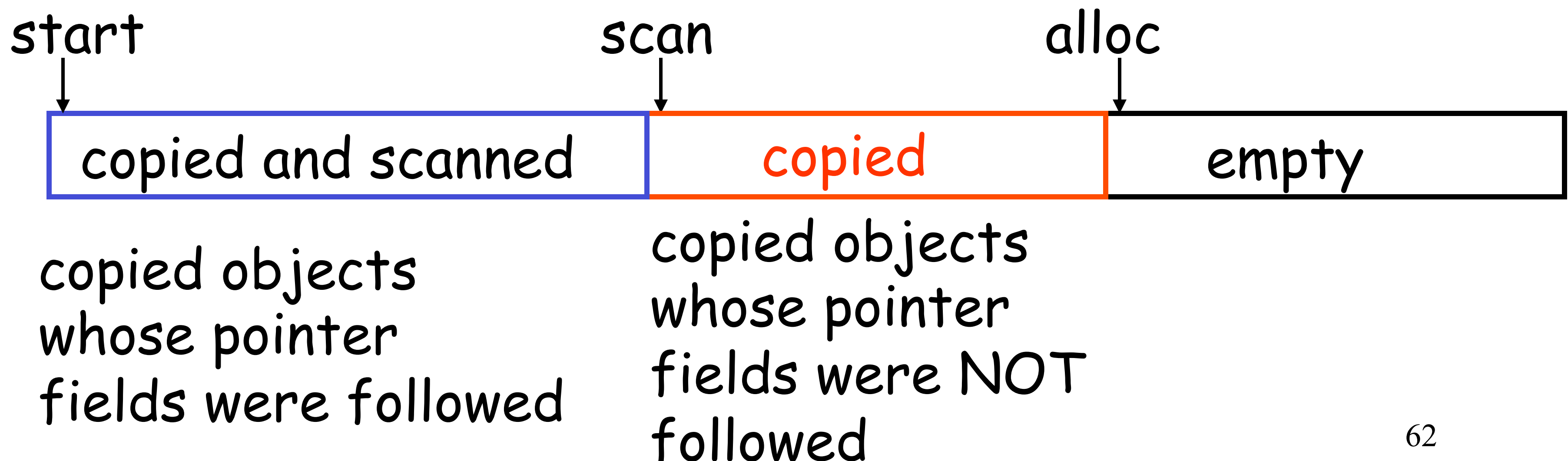
- Idea: As we copy an object we store in the old copy a *forwarding pointer* to the new copy
  - when we later reach an object with a forwarding pointer we know it was already copied



If later in GC we find a pointer to the old object, we follow the forwarding pointer, the new address to update the old pointer

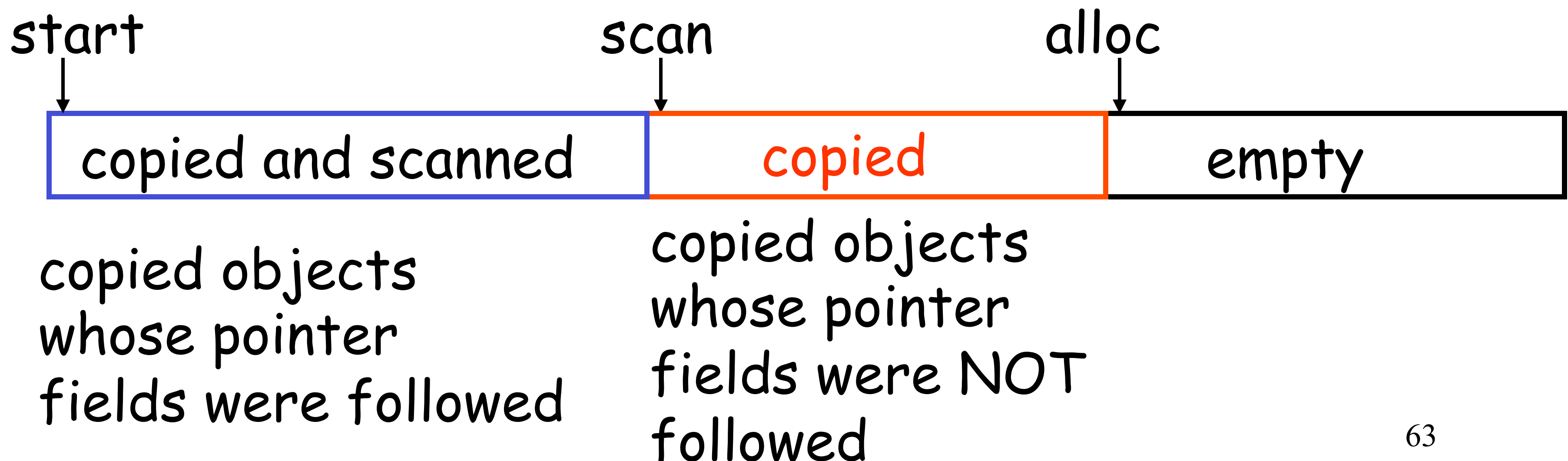
Implementation of Stop and Copy (Cont.)

- Just as with mark and sweep we have the issue of how to implement the traversal without using extra space
  - Because these GC algorithms only get used in low memory situations, so you can't assume memory to build GC data structures



Implementation of Stop and Copy (Cont.)

- Just as with mark and sweep we have the issue of how to implement the traversal without using extra space
  - Really need these algorithms to work in small constant amount of space



Implementation of Stop and Copy (Cont.)

- Just as with mark and sweep we have the issue of how to implement the traversal without using extra space
- The following trick solves the problem:
  - partition the new space in three contiguous regions

start → | copied and scanned | scan → | copied | alloc → | empty |

copied objects whose pointer fields were followed ("copied and scanned")

copied objects whose pointer fields were NOT (yet) followed (not "scanned")

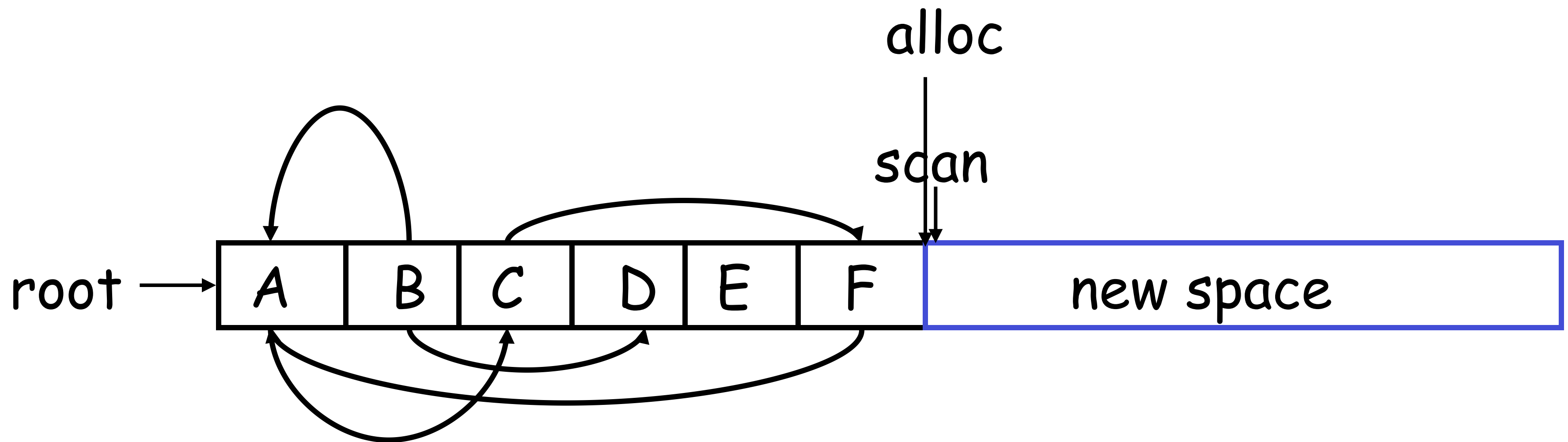
Implementation of Stop and Copy (Cont.)

- Just as with mark and sweep we have the issue of how to implement the traversal without using extra space
- The following trick solves the problem:
  - partition the new space in three contiguous regions

so this middle section is really the "work list" (these objects might point to objects that have not yet been copied)

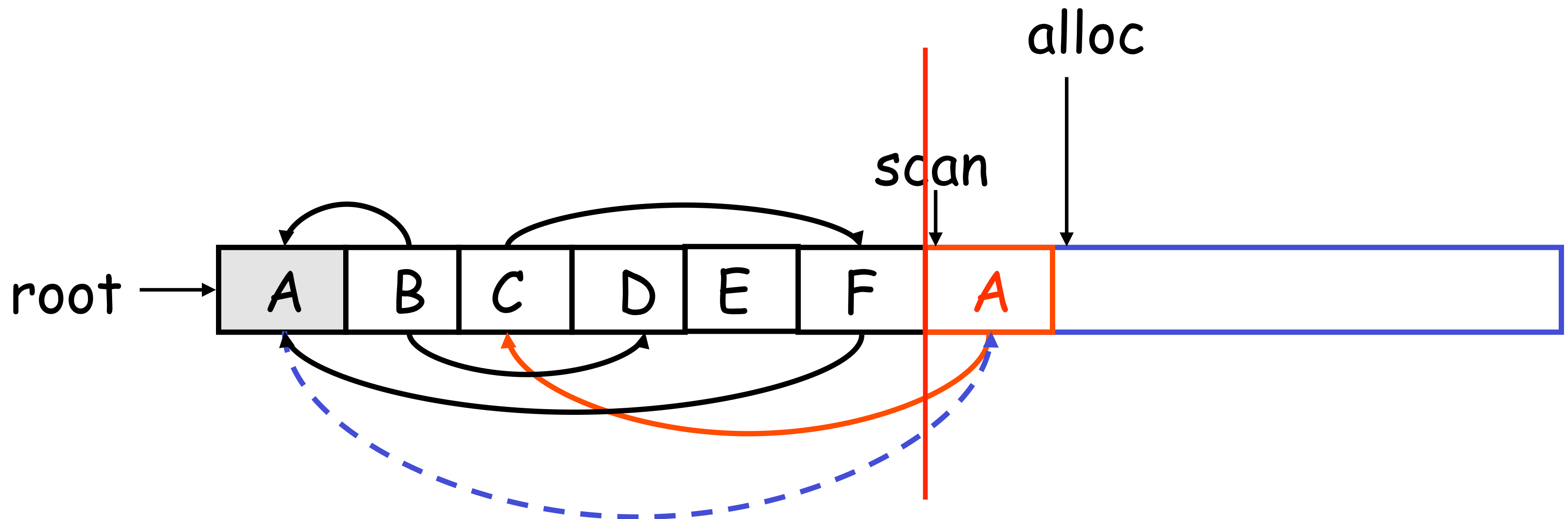
Stop and Copy. Step-by-step Example (1)

- Before garbage collection



Stop and Copy. Example (Step 2)

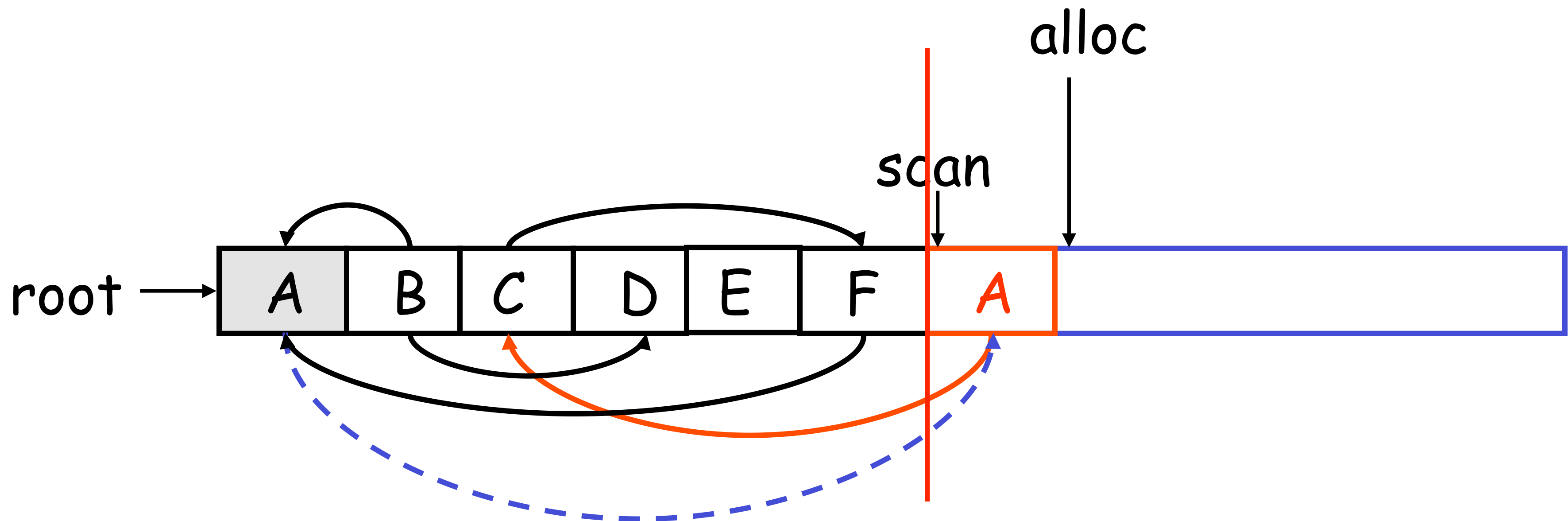
- Step 1: Copy the objects (bit-by-bit copy) pointed to by roots and set forwarding pointers



Note that the copy of A points to C in the old space

Stop and Copy. Example (Step 2)

- Step 1: Copy the objects (bit-by-bit copy) pointed to by roots and set forwarding pointers



Note also that we've marked A (it's grayed out) and left a forwarding pointer (dashed line)

Stop and Copy. Example (Step 3)

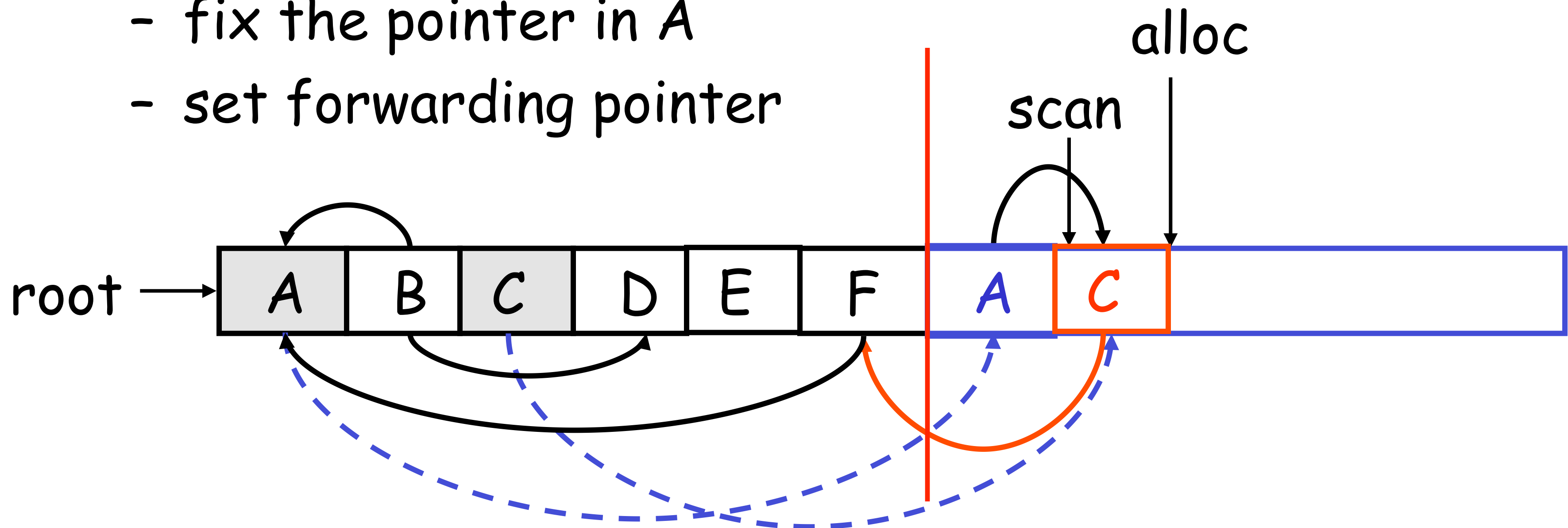
- Step 2: Follow the pointer in the next unscanned object (*A*)
  - copy the pointed-to objects (just *C* in this case)
  - fix the pointer in *A*
  - set forwarding pointer

alloc
scan
root
A B C D E F A C

How do we know there is more to do? If scan and alloc are different, there is work to be done (whatever is between them)

Stop and Copy. Example (Step 3)

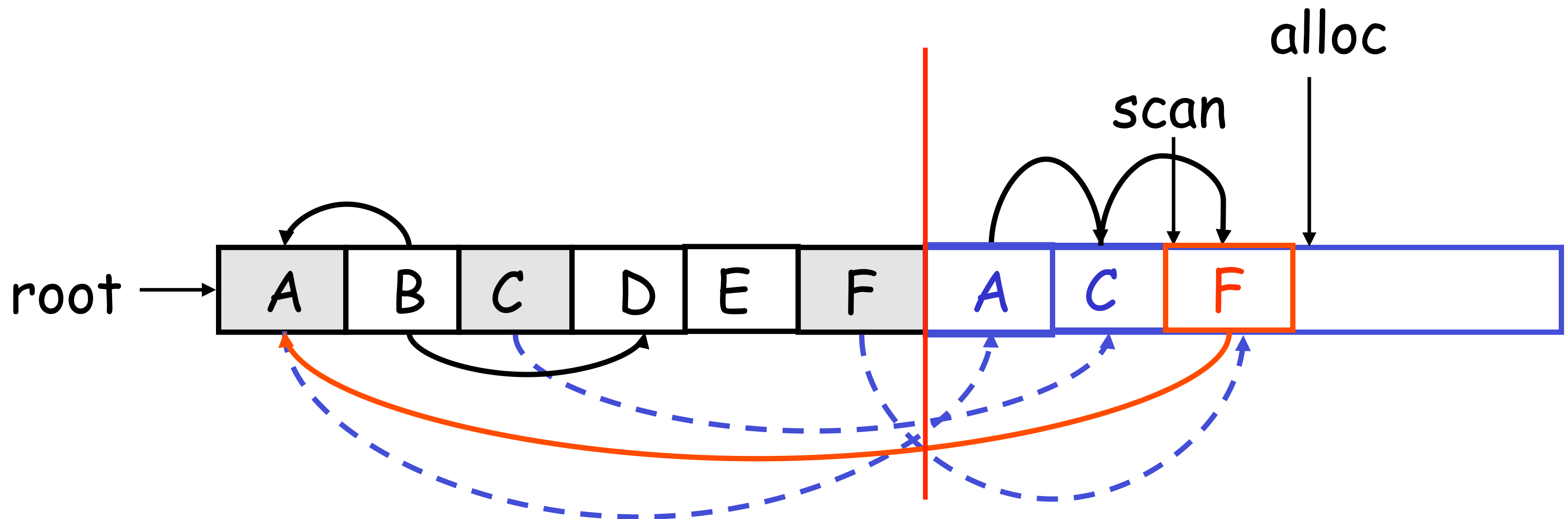
- Step 2: Follow the pointer in the next unscanned object (A)
  - copy the pointed-to objects (just C in this case)
  - fix the pointer in A
  - set forwarding pointer



Note movement of scan and alloc pointers

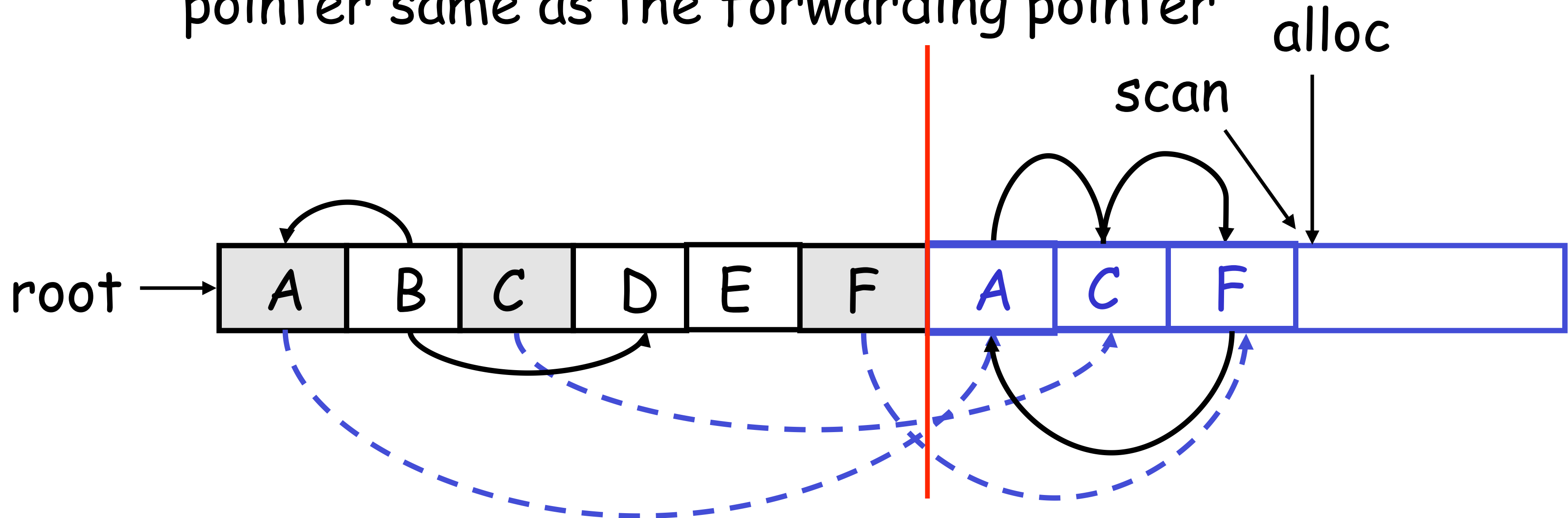
Stop and Copy. Example (Step 4)

- Follow the pointer in the next unscanned object (C)
  - copy the pointed objects (F in this case)



Stop and Copy. Example (Step 5)

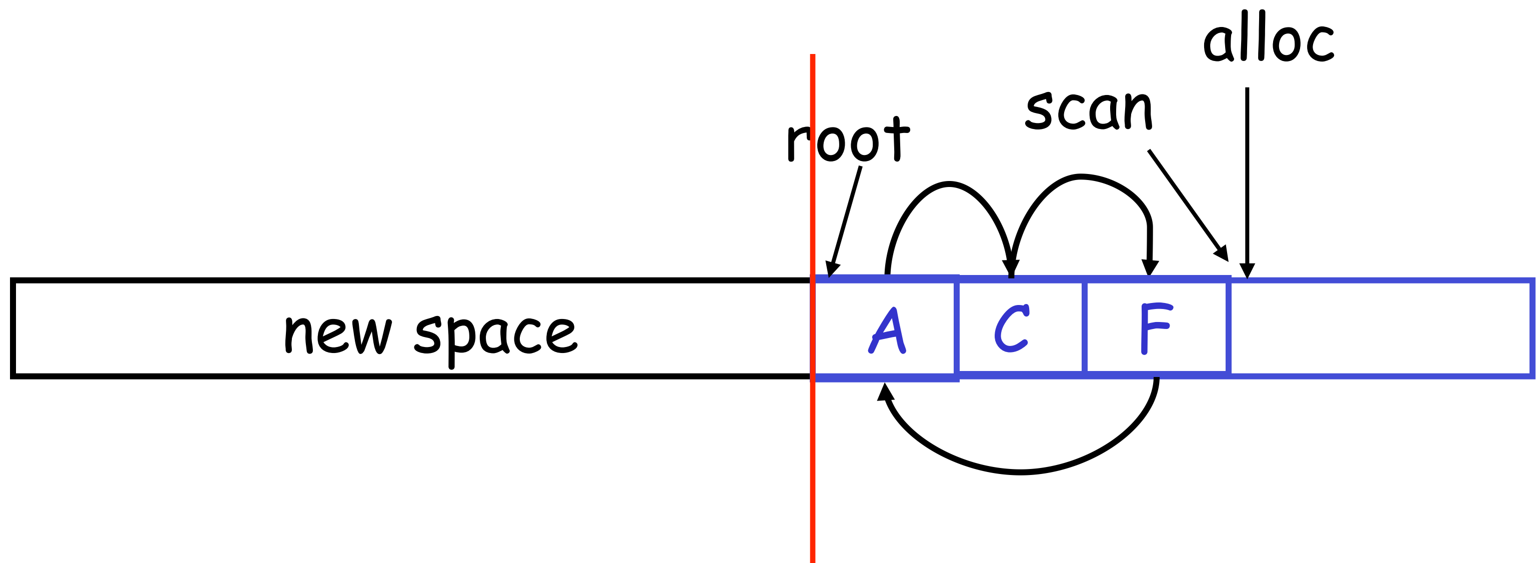
- Follow the pointer in the next unscanned object (F)
  - the pointed object (A) was already copied. Set the pointer same as the forwarding pointer



That is, instead of copying A again, we just set the pointer from F to the old location of A to the value of the forwarding pointer of A

Stop and Copy. Example (Step 6)

- Since scan caught up with alloc we are done
- Swap the role of the spaces and resume the program



We now have a complete copy of the old reachability graph but using less space.

The Stop and Copy Algorithm

```
while scan ≠ alloc do
    let O be the object at scan pointer
    for each pointer p contained in O do
        find O' that p points to
        if O' is without a forwarding pointer
            copy O' to new space (update alloc pointer)
            set 1st word of old O' to point to the new copy
            mark old object as copied
            change p to point to the new copy of O'
        else
            set p in O equal to the forwarding pointer
        fi
    end for
    increment scan pointer to the next object
od
```

Details of Stop and Copy

- As with mark and sweep, we must be able to tell how large an object is when we scan it
  - and we must also know where the pointers are inside the object
- We must also copy any objects pointed to by the stack and update pointers in the stack
  - this can be an expensive operation

Evauation of Stop and Copy

- Stop and copy is generally believed to be the fastest GC technique
- Allocation is very cheap
  - just increment the heap pointer
- Collection is relatively cheap
  - especially if there is a lot of garbage
  - only touch reachable objects
- But some languages do not allow copying
  - C, C++

Why Doesn't C Allow Copying?

- Garbage collection relies on being able to find all reachable objects
  - and it needs to find all pointers in an object

- In C or C++ it is impossible to identify the contents of objects in memory with 100% reliability
  - E.g., a sequence of two memory words might be
    - A list cell (with data and next fields)
    - A binary tree node (with left and right fields)
  - Thus we cannot tell where all the pointers are
    - Effectively this problem is a result of the weakness of the C, C++ type system

Conservative Garbage Collection

- Insight: it is Ok to be conservative (which allows us to extend the method to languages like C, C++)
  - if a memory word looks like a pointer it is OK to consider it a pointer
    - it must be aligned (e.g., address a multiple of 4)
    - it must point to a valid address in the data segment
    - these two requirements rule out most stuff in memory
  - all such pointers are followed and thus we overestimate the set of reachable objects
  - And recall that in GC, reachability is an approximation and not exact
  - So we're going to likely keep around some things that aren't necessary, but that's OK

Conservative Garbage Collection

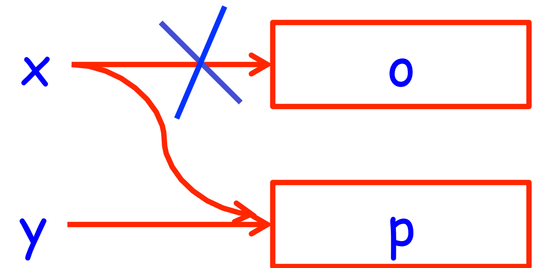
- Insight: it is Ok to be conservative (which allows us to extend the method to languages like C, C++)
  - if a memory word looks like a pointer it is OK to consider it a pointer
    - it must be aligned (e.g., address a multiple of 4)
    - it must point to a valid address in the data segment
    - these two requirements rule out most stuff in memory
- But with this trick, we cannot move objects because we cannot update pointers to them
  - we're not exactly sure whether what we think is a pointer actually IS a pointer. what if what we think is a pointer is actually an account number? And we change it! That's bad
  - So this trick can't be used with stop and copy. Can only be used with mark and sweep

Reference Counting

- Rather that wait for memory to be exhausted, try to collect an object when there are no more pointers to it
- Store in each object the number of pointers to that object
  - this is the reference count
  - Each object has a dedicated field for this count
- Each assignment operation manipulates the reference count
  - If the count drops to zero, then object can be freed

Implementation of Reference Counting

- new returns an object with reference count 1
- Let rc(x) be the reference count of x
- Assume x, y point to objects o, p



- Every assignment x ← y must be changed to:

  rc(p) ← rc(p) + 1
  rc(o) ← rc(o) - 1
  if(rc(o) == 0) then mark o as free
  x ← y

check o because we dropped its reference count

Evaluation of Reference Counting

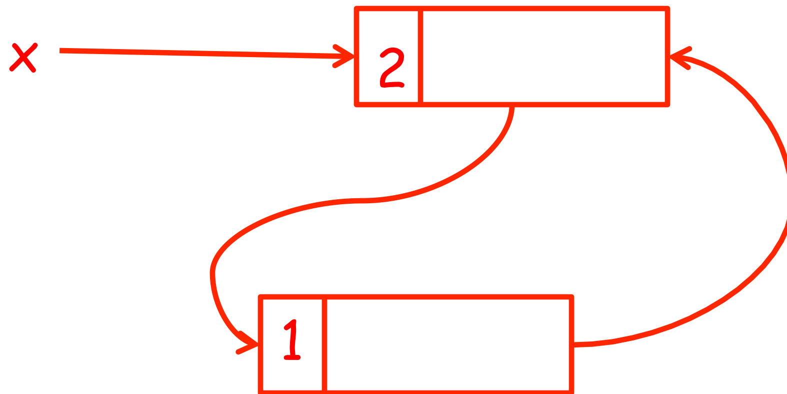
- Advantages:
  - easy to implement
    - Modifications to code are clear. Moreover code generation can simply be modified to generate the appropriate reference counting code
  - collects garbage incrementally without large pauses in the execution
    - So for applications where large pauses are problematic (e.g., real-time apps, interactive apps), reference counting can really help (it minimizes the length of the longest pause for GC)

Evaluation of Reference Counting

- Disadvantages:
  - manipulating reference counts at each assignment is very slow
    - two updates to reference counts, a conditional (effectively introducing conditional into every assignment in the program), then the assignment itself
    - Overhead: taking every single assignment in program and blowing up its cost by about a 4-5 times – this has a very noticeable effect on the performance of many programs
      - Overhead can be slightly eased by optimizing updates to the reference counters (e.g., compiler combines two updates into a single one if they both occur in same basic block) though tricky to get this right (and trickier the more overhead you're trying to save)

Evaluation of Reference Counting

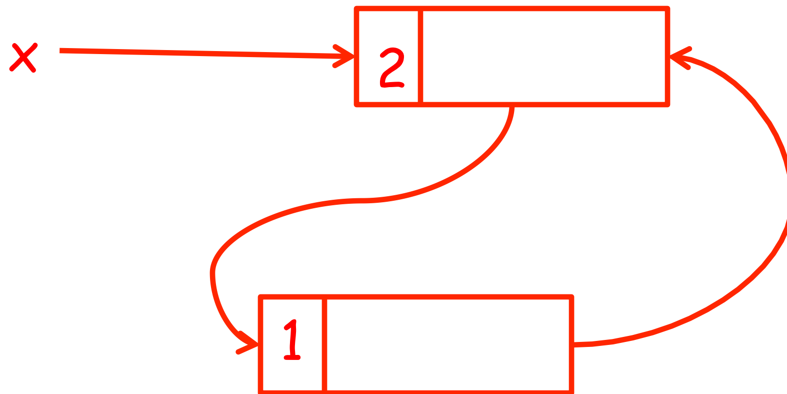
- Disadvantages:
  - cannot directly collect circular structures:



Numbers are reference counts

Evaluation of Reference Counting

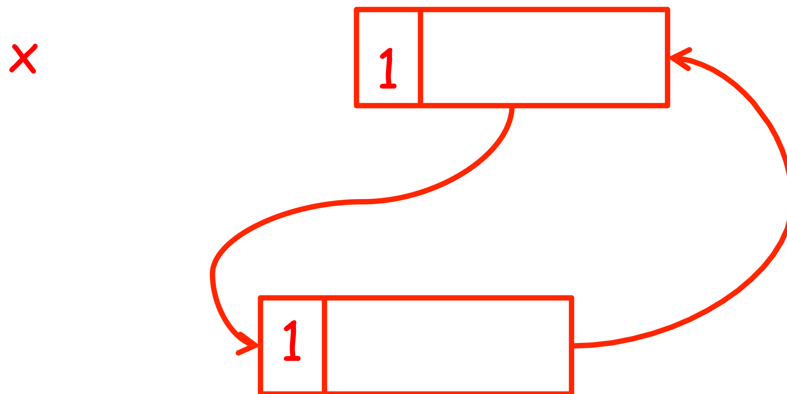
- Disadvantages:
  - cannot directly collect circular structures:



Consider now assignment x ← null

Evaluation of Reference Counting

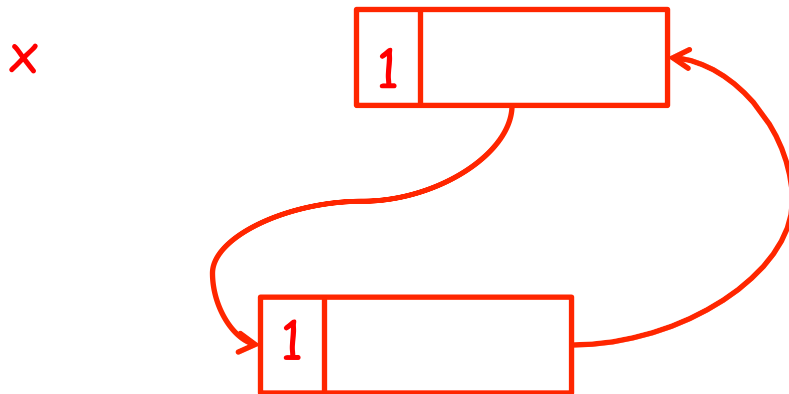
- Disadvantages:
  - cannot directly collect circular structures:



The two objects are unreachable, but GC can't collect them because both have reference count of 1

Evaluation of Reference Counting

- Disadvantages:
  - cannot directly collect circular structures:



What GC can't see is that both of the references are from unreachable objects!

Evaluation of Reference Counting

- Disadvantages:
  - cannot directly collect circular structures
  - So, how to deal with this?
    - Programmer has to remember this realize that if a circular structure is going to become unreachable, try to somehow break the circularity (e.g., by assigning null to bottom object before assigning null to x) OR
    - Back reference counting with some other GC technique that CAN collect cycles
      - So in many reference counting GC systems, a system is set up where every once in a while a mark and sweep cycle is run to collect all the unreachable circular data structures

Evaluation of Garbage Collection

- Automatic memory management prevents serious storage bugs
  - So overall it's a great thing – prevents some of the most difficult bugs in programming
  - So when writing in a garbage collected language, there are whole classes of problems that the programmer does not have to worry about
    - which is a boost to productivity
  - Basically, if your program is a good fit for GC, then you'd be crazy not to use that kind of support

Evaluation of Garbage Collection

- But downside is that GC reduces programmer control
  - e.g., no control of layout of data in memory
  - e.g., not control of when memory is deallocated
    - So no control over how much memory your program is using
  - Really problematic for high-end data processing and scientific applications
    - Use a lot of data and need to make very efficient use of memory
    - People in these domains usually still use manual memory management

Evaluation of Garbage Collection

- Also pauses problematic in real-time applications
  - E.g., embedded systems controlling dangerous machinery have to have guaranteed response times
  - This being said, there has been much progress in the last few years on "real-time" garbage collectors
- Memory leaks possible (even likely)
  - GC prevents you from corrupting your memory, but really not from hanging onto too much memory (and possibly affecting the performance of your program dramatically)
  - Leaks likely because programmer not as aware of how memory being used

Example

- In Java, suppose variable x in a compiler points to the AST of a program
  - This is a huge data structure
  - But once intermediate code is generated, compiler no longer requires AST
  - But AST won't be collected, because x points to it!
  - What programmer should do (but many don't) is assign null to x once the AST is no longer needed
  - It's very common in production Java programs to have this kind of memory leak because programmer just forgot about it

Evaluation of Garbage Collection

- Garbage collection is very important
  - Every programmer should be aware of its benefits and costs
  - It's also a very interesting aspect of the implementation of programming languages

Evaluation of Garbage Collection

- There are much more advanced garbage collection algorithms than what we have discussed:
  - concurrent: allow the program to run while the collection is happening
  - generational: do not scan long-lived objects at every collection
    - Once we've seen such an object in a few iterations of GC, assume it's going to be around a while and skip it for a few iterations (they are put in a separate area that is collected less frequently)
  - real time: bound the length of pauses
  - parallel: several collectors working in parallel