Lecture 22

COMPILER DESIGN

Announcements

- **HW5**: OAT v. 2.0
  - Records, function pointers, type checking, array-bounds checks, etc.

- **HW6**: Analysis & Optimizations (the final homework)
  - Alias analysis, constant propagation, dead code elimination, register allocation

Plan

- ~~Next: Register allocation~~

- Upcoming
  - ~~Dataflow analysis (part 2)~~
  - ~~Control-flow analysis & SSA~~
  - **Garbage collection (GC)**
  - Compiler testing & validation
    - How to find thousands of bugs in GCC & LLVM?
  - Compiler verification
    - How to build a fully verified *realistic* compiler?
  - MLIR
  - Guest lecture on GraalVM, PGQL & Green-Marl
  - Summary

AUTOMATIC MEMORY MANAGEMENT (GC)

Plan

- Why Automatic Memory Management (AMM)?
- 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

- C/C++ programs have many storage bugs
  - forgetting to free unused memory
  - dereferencing a dangling pointer
  - overwriting parts of a data structure by accident
  - and so on...

- Storage bugs are hard to find
  - A bug can manifest far away in time & program text from its source

Type Safety and Memory Management

- Some storage bugs can be prevented in a strongly typed language
  - e.g., we cannot overrun the array limits, dereference a null pointer, etc.

- Can types prevent errors with manual memory allocation/deallocation?
  - Some fancy type systems (linear types) were designed for this purpose (Rust)
  - ... but may complicate programming

- If we want type safety, we typically must use AMM (GC)

Automatic Memory Management

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

- Several well-known techniques for performing completely AMM

- For a (long) while, they were unpopular outside Lisp family of languages
  - Just like type safety used to be unpopular

The Basic Idea

- When an object is created, unused space is automatically allocated
  - New objects are created by malloc or new in C/C++

- After a while there is no more unused space

- Some space is occupied by objects that will never be used again

- This space can be freed to be reused later

The Basic Idea (Cont.)

- How to tell whether an object will “never be used again”?
  - In general it is impossible to tell
  - Heuristics to find many (not all) objects that will never be used again

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

  let x : A = new A in { x = y; ... }

  - After x = y there is no way to access the newly allocated object

Garbage

- An object **x** is reachable iff
  - A **register** contains a pointer to **x**, **or**
  - Another reachable object **y** contains a pointer to **x**

- One can find all reachable objects by
  - starting from registers, and
  - following all the pointers

- Unreachable objects can never by referred by the program
  - These objects are called 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
```

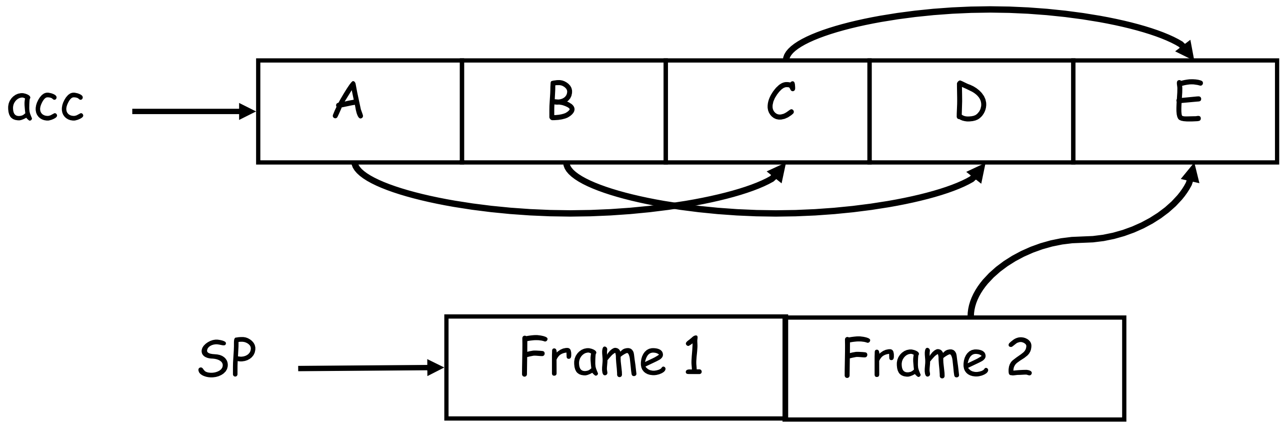
- After `x = y` (assuming y becomes dead there)
  - The object A is not reachable anymore
  - The object B is reachable (through x)
  - Thus, B is not garbage and is not collected
  - But, object B is never going to be used

Tracing Reachable Values

- Assume the only register is the accumulator
  - it points to an object, and
  - this object may point to other objects, etc.

- 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

A Simple Example



- We start tracing from acc and stack
  - they are called the roots

- Note that B and D are not reachable from acc or the stack

- Thus, we can reuse their storage

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)

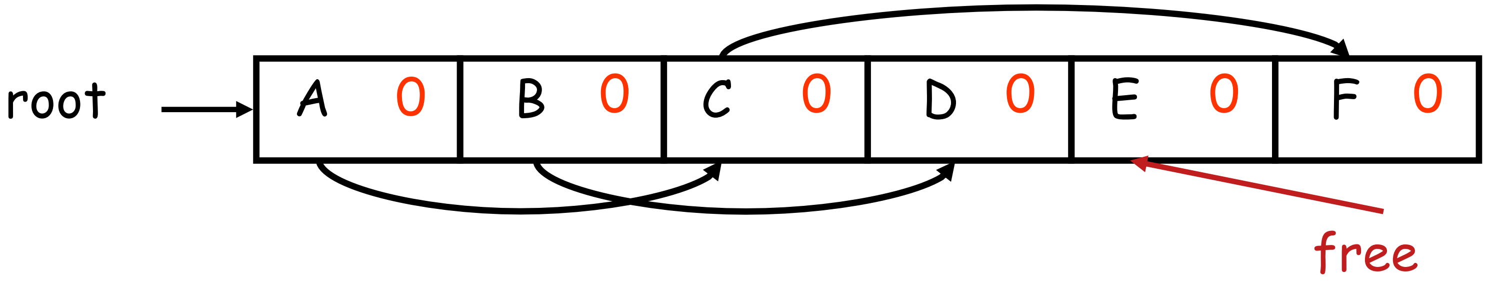
- Some strategies perform GC before space actually runs out

Mark and Sweep [McCarthy 1960]

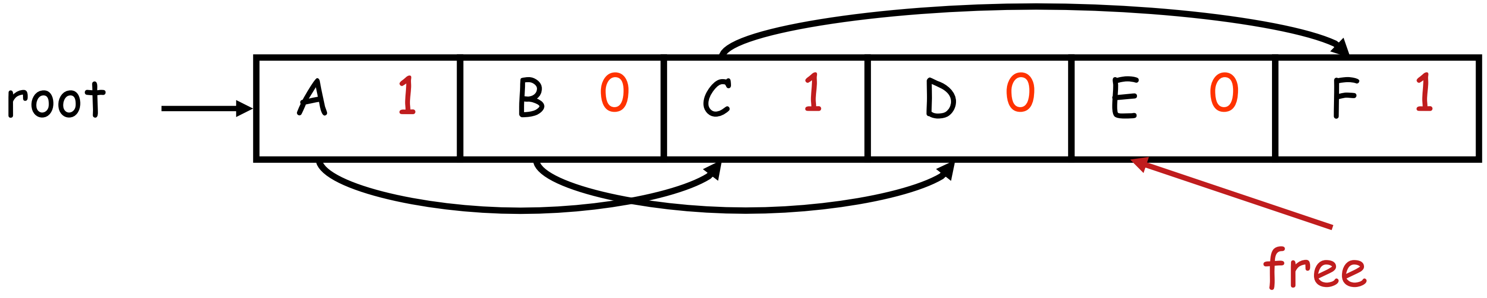
- When memory runs out, GC executes two phases
  - **mark phase**: traces reachable objects
  - **sweep phase**: collects garbage objects

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

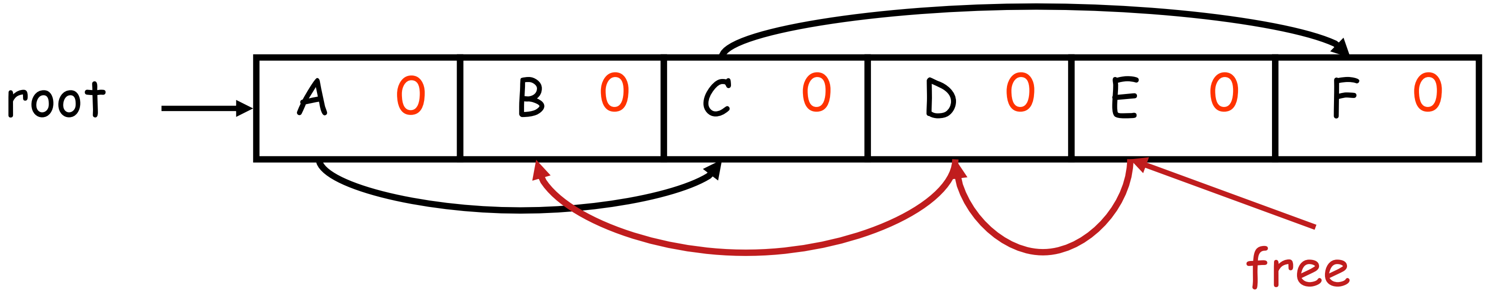
Mark and Sweep Example



After mark:



After sweep:



The Mark Phase

```
let todo = { all roots }
while todo $\neq \emptyset$ do
    pick v $\in$ todo
    todo $\leftarrow$ todo \ { v }
    if mark(v) = 0 then      (* v is unmarked yet *)
        mark(v) $\leftarrow$ 1
        let $v_1,\ldots,v_n$ be the pointers contained in v
        todo $\leftarrow$ todo $\cup$ $\{v_1,\ldots,v_n\}$
    fi
od
```

The Sweep Phase

- The sweep phase scans the heap for objects with mark bit 0
  - these objects have not been 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

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
```

Details

- While conceptually simple, there are some tricky details
  - which is typical of GC algorithms

- A serious problem with the mark phase
  - it is invoked when we are out of space
  - yet it needs space to construct the todo list
  - size of the todo list is unbounded, so cannot reserve space a priori

```
let todo = { all roots }
while todo $\neq \emptyset$ do
    pick v $\in$ todo
    todo $\leftarrow$ todo \ { v }
    if mark(v) = 0 then        (* v is unmarked yet *)
        mark(v) $\leftarrow$ 1
        let $v_1,\ldots,v_n$ be the pointers contained in v
        todo $\leftarrow$ todo $\cup$ $\{v_1,\ldots,v_n\}$
    fi
od
```

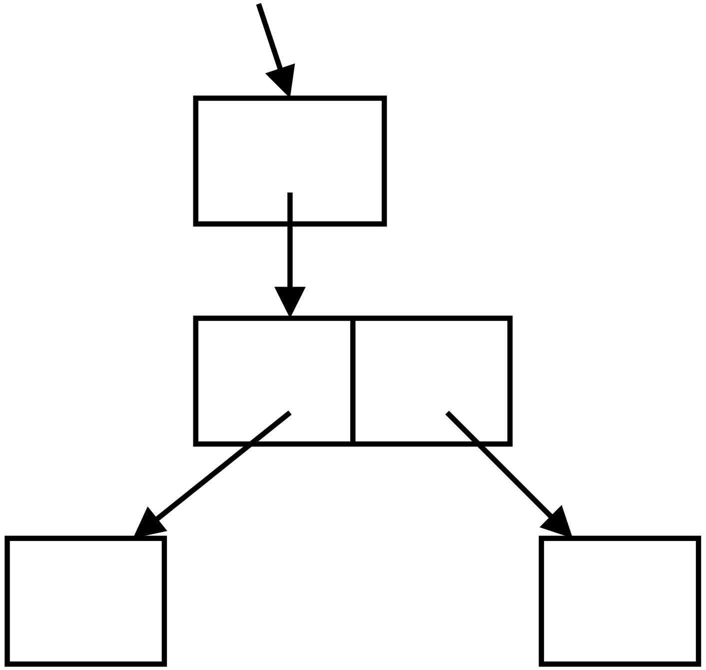
Mark and Sweep: Details

- The todo list is used as auxiliary data structure to perform reachability

- There is a trick to allow the auxiliary data to be stored in the objects
  - **pointer reversal**: when a pointer is followed, reverse it to point to its parent
  - by Deutsch-Schorr-Waite (DSW)

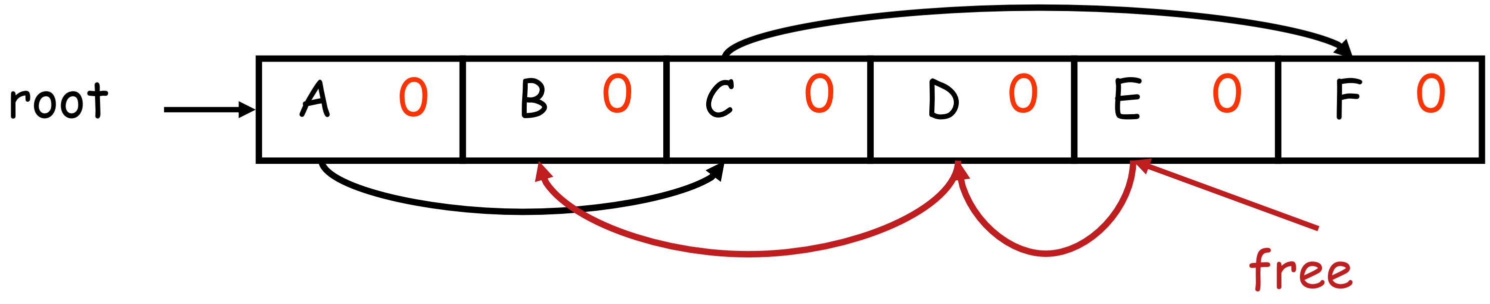
- Similarly, the free list is stored in the free objects themselves

Pointer Reversal



Mark and Sweep: Evaluation

- Space for a new object is allocated from the new list
  - a block large enough is picked
  - an area of the necessary size is allocated from it
  - the left-over is put back in the free list

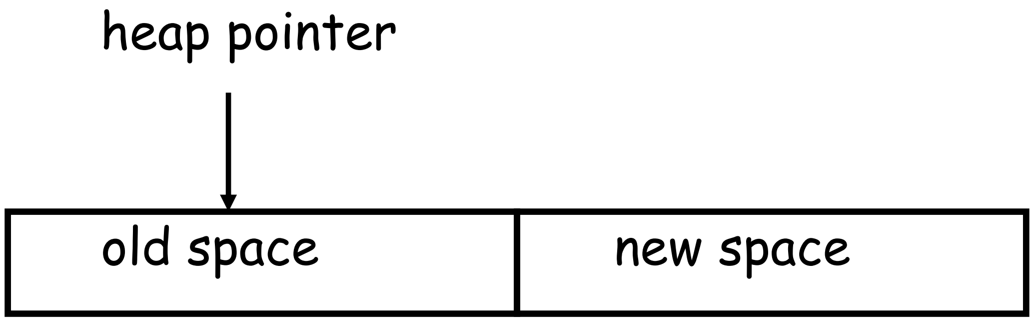


- Mark and sweep can fragment the memory

- Advantages: objects are not moved during GC
  - no need to update the pointers to objects
  - works for languages like C/C++

Another Technique: Stop and Copy

- Memory is organized into two areas
  - **Old space**: used for allocation
  - **New space**: used as a reserve for GC



- The heap pointer points to the next free word in old space
  - Allocation just advances the heap pointer

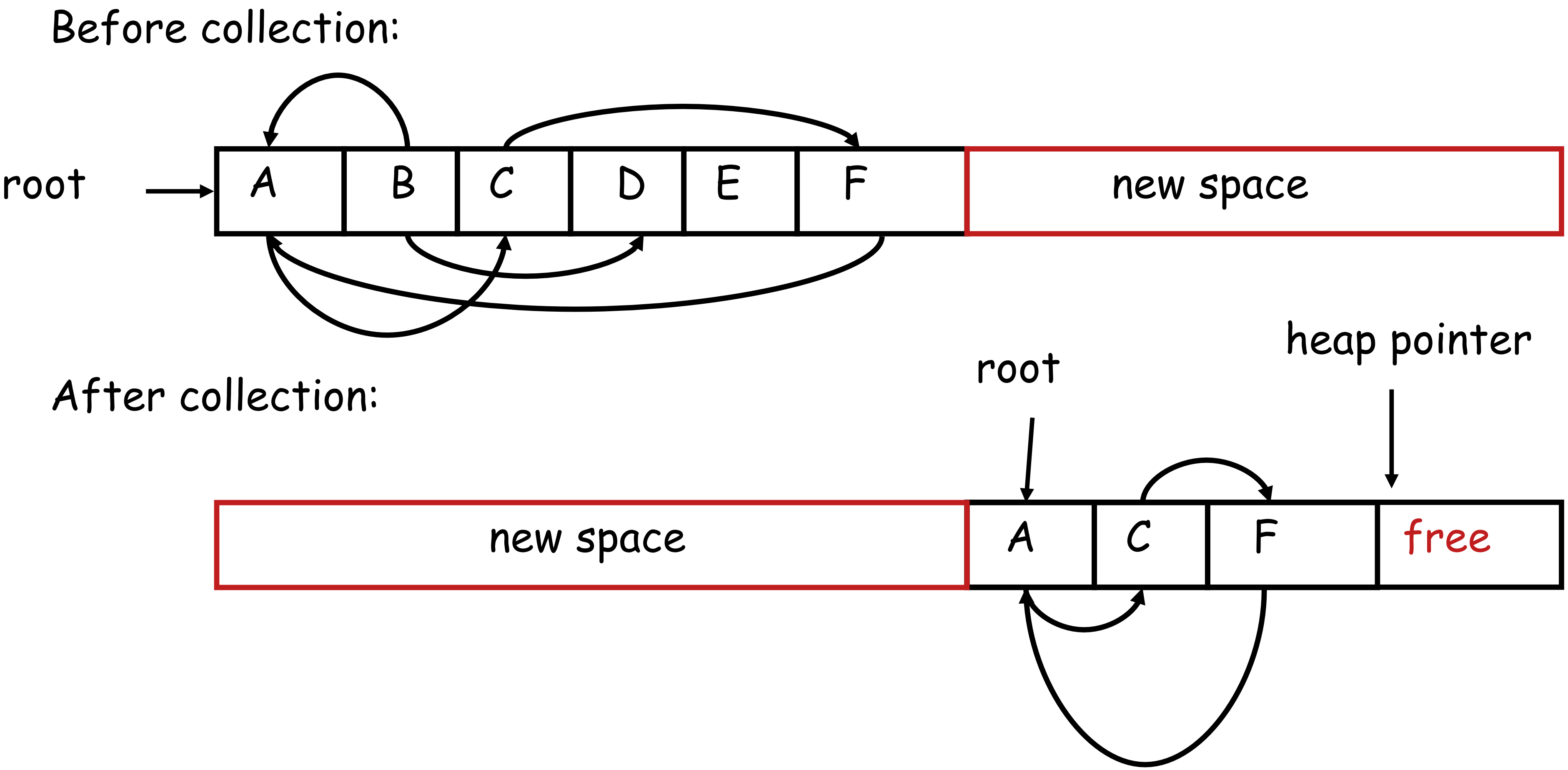
Stop and Copy Garbage Collection

- Starts when the old space is full

- Copies all reachable objects from old space into new space
    - garbage is left behind
    - after copy phase, new space uses less space than old space before GC

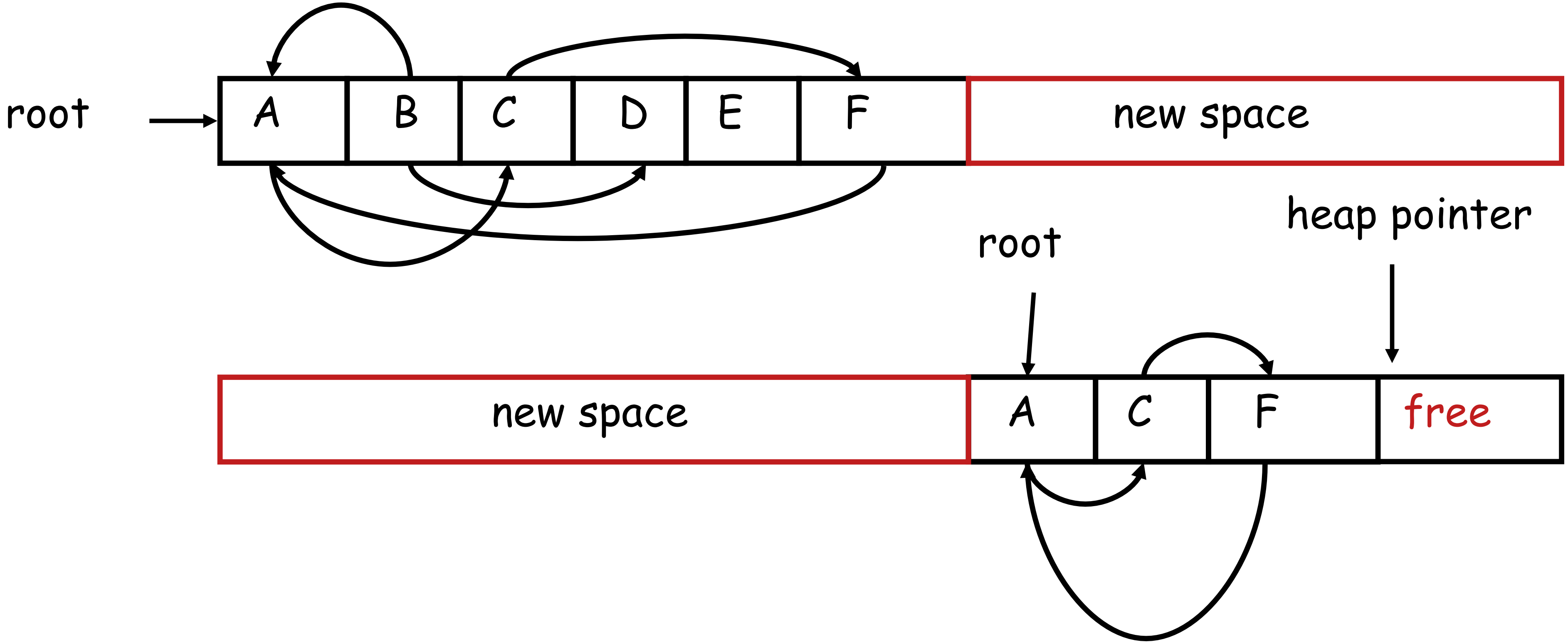
- After the copy
    - The roles of old & new spaces are reversed, and
    - The program resumes

Stop and Copy Garbage Collection: Example



Implementation of Stop and Copy

- Need to find all reachable objects, as for mark and sweep
- As we find a reachable object, copy it into the new space
  - And we have to fix **all** pointers pointing to it!



Implementation of Stop and Copy

- Need to find all reachable objects, as for mark and sweep

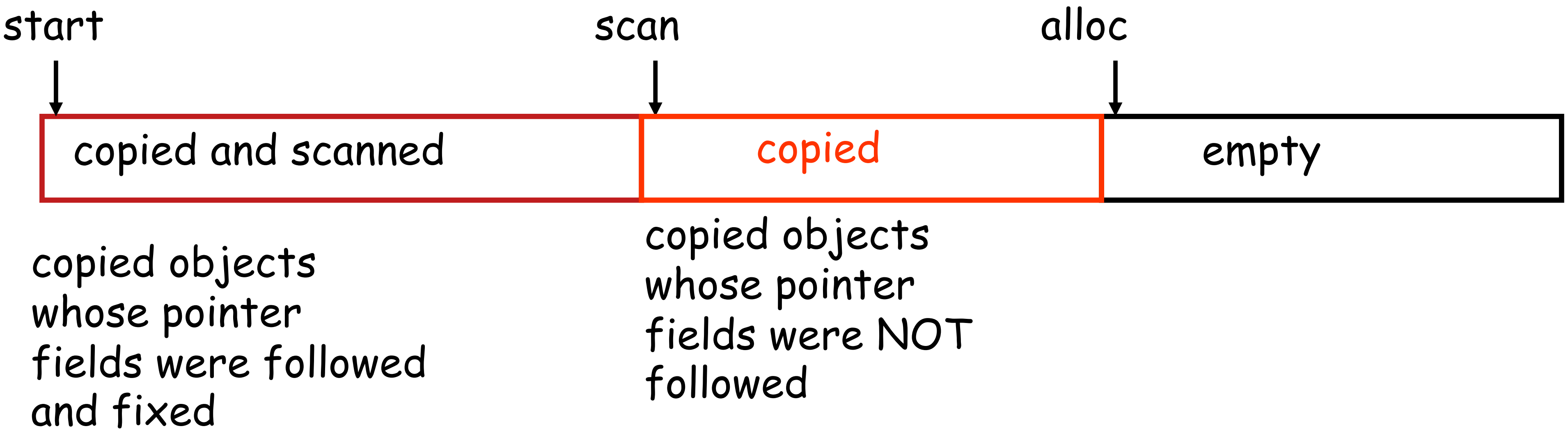
- As we find a reachable object, copy it into the new space
  - And we have to fix **all** pointers pointing to it!

- As we copy an object
  - store in the old copy a **forwarding pointer** to the new copy
  - Any object reached later with a forwarding pointer was already copied

Implementation of Stop and Copy (Cont.)

- Still the issue of how to implement the traversal w/o using extra space

- The following trick solves the problem
  - partition the new space in three contiguous regions



start

scan

alloc

copied and scanned

copied

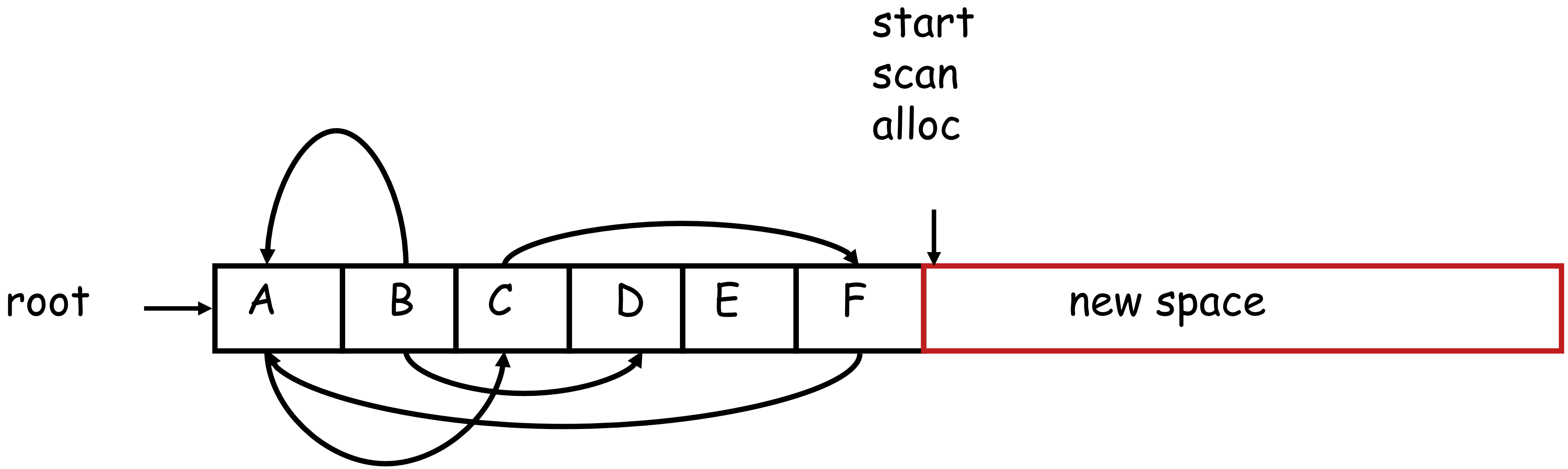
empty

copied objects whose pointer fields were followed and fixed

copied objects whose pointer fields were NOT followed

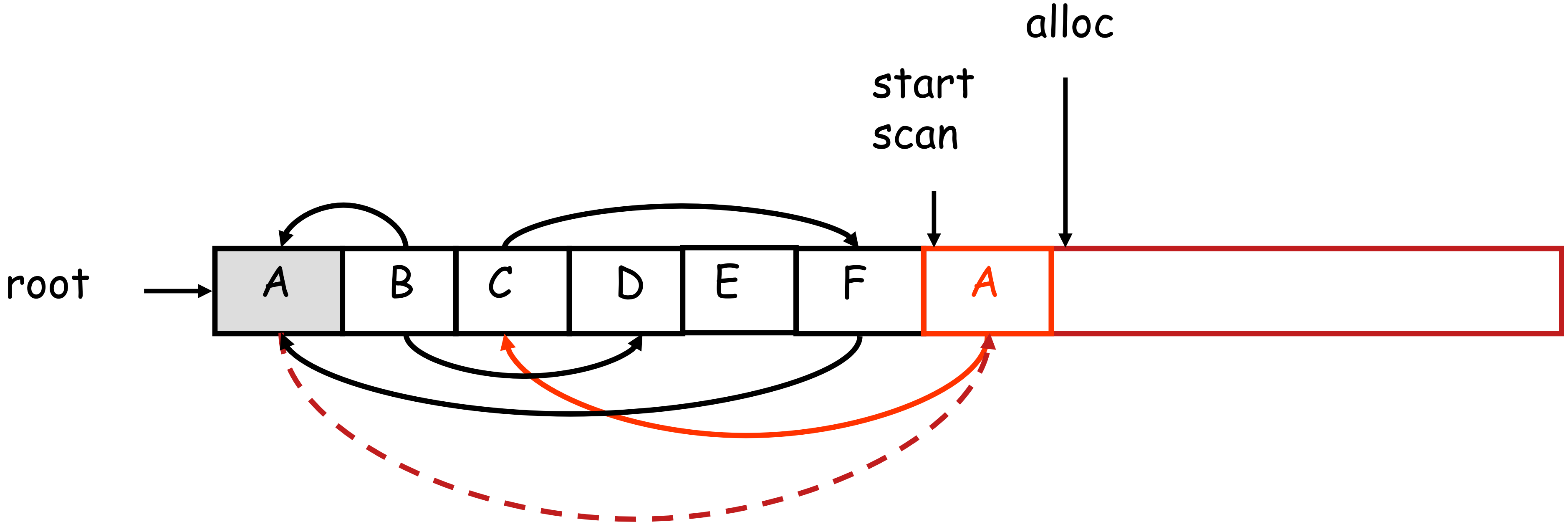
Stop and Copy: Example (1)

- Before garbage collection



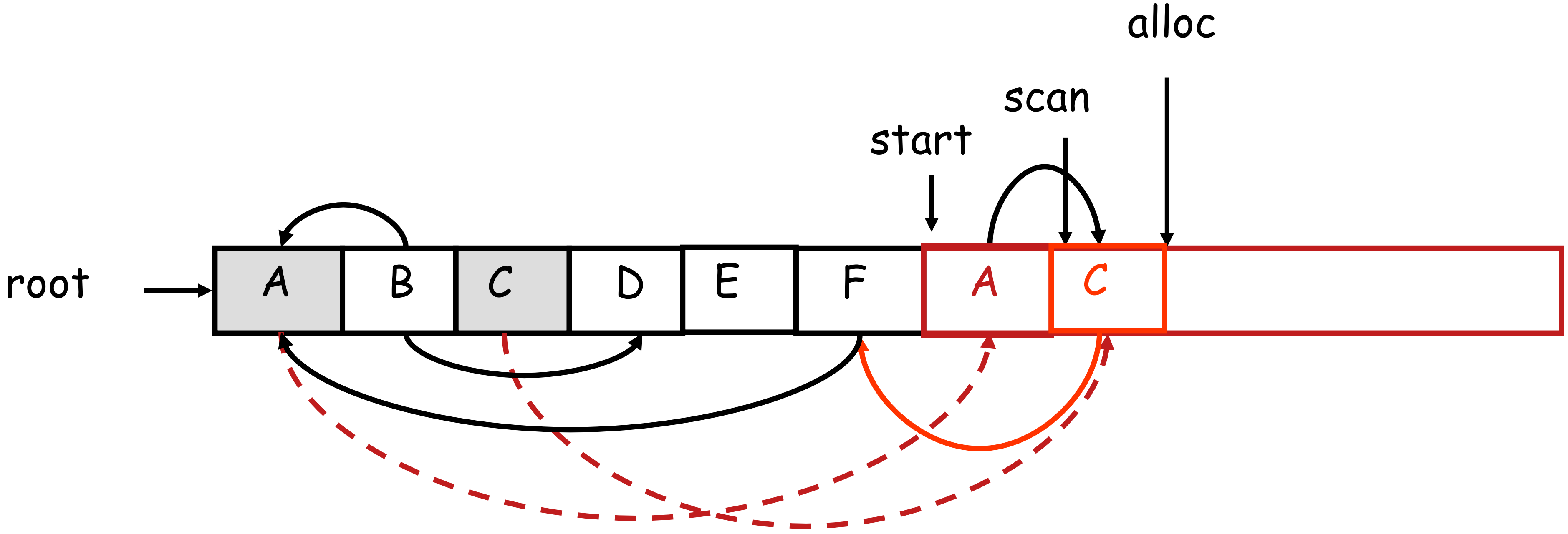
Stop and Copy: Example (3)

- Step 1: Copy objects pointed by roots, set forwarding pointers (dotted arrow)



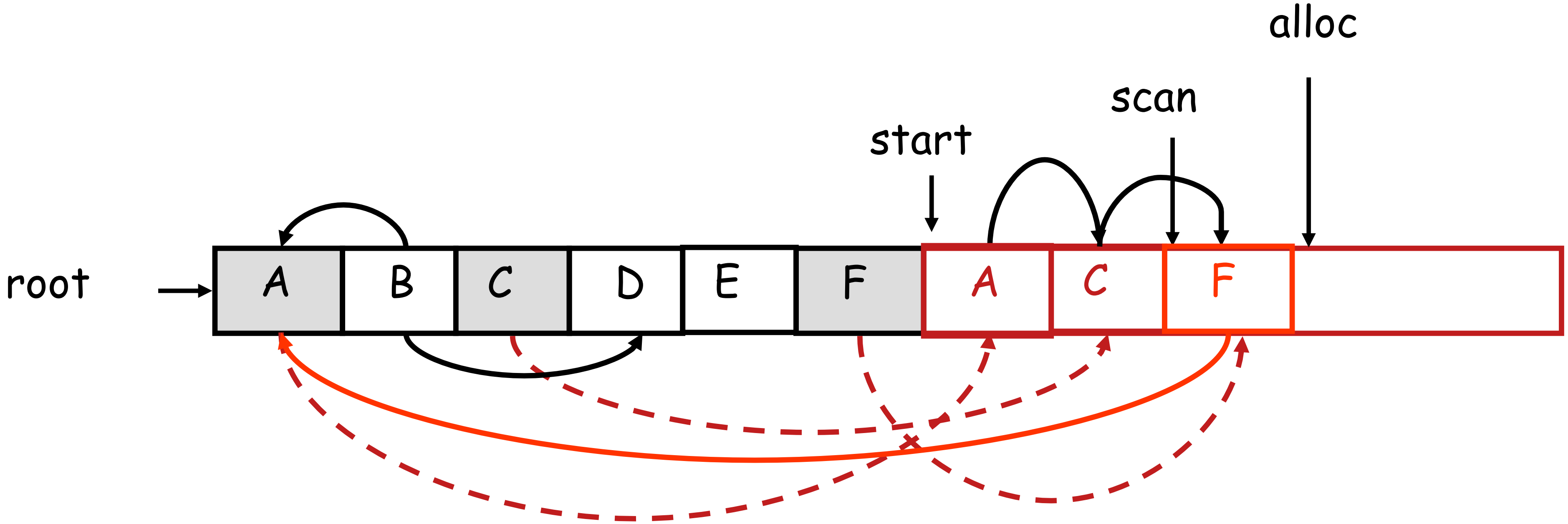
Stop and Copy: Example (3)

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



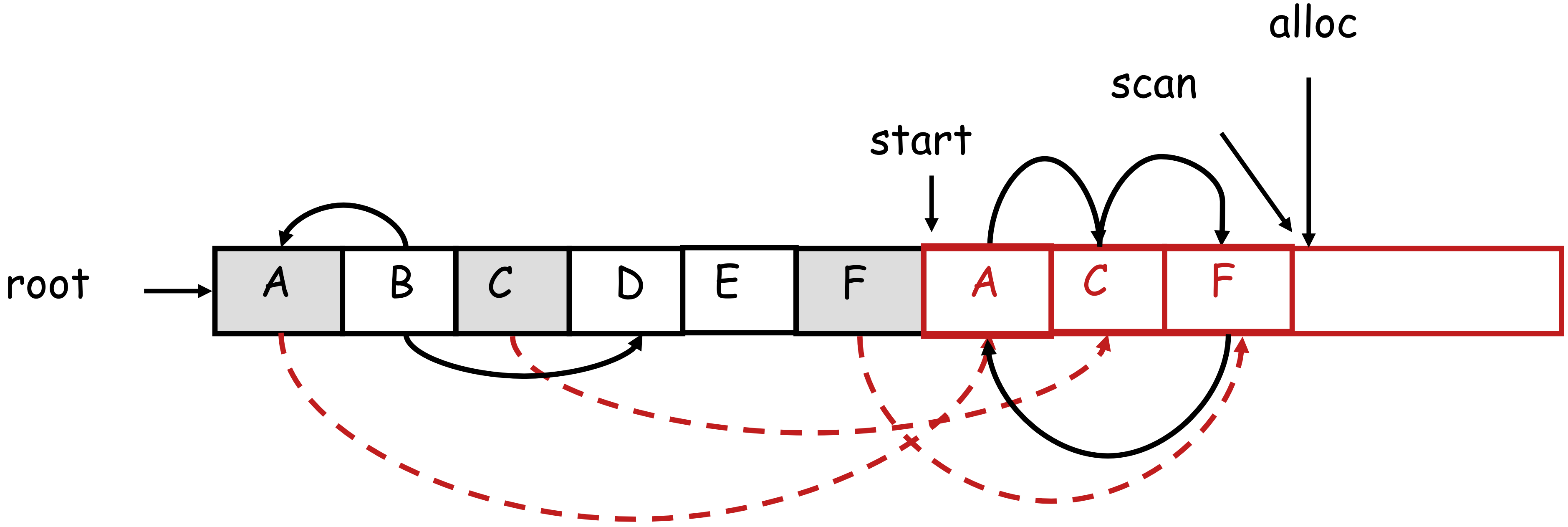
Stop and Copy: Example (4)

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



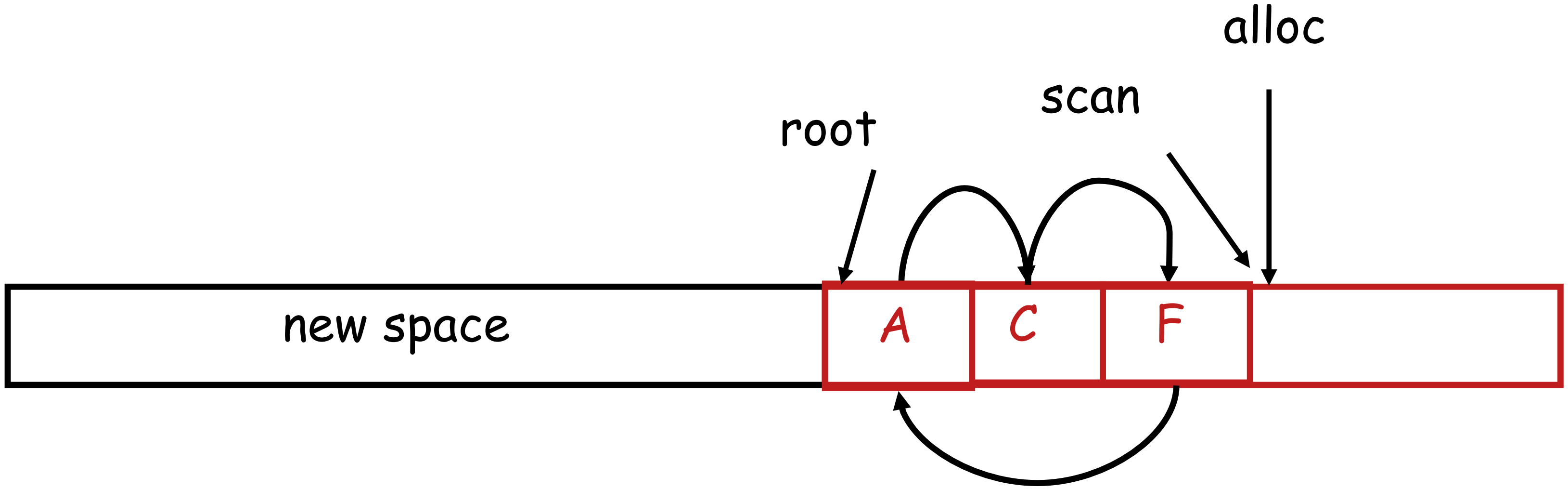
Stop and Copy: Example (5)

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



Stop and Copy: Example (6)

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



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
            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
```

Stop and Copy: Details

- Like mark & sweep, we must tell how large an object is when we scan it
    - And we must also know where are the pointers 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

Stop and Copy: Evaluation

- 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 don't 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/C++, it's impossible to identify the contents of objects in memory
  - E.g., how to tell whether a sequence of two memory words is
    - a list cell (with data and next fields), or
    - a binary tree node (with a left and right fields)
  - Thus we cannot tell where all the pointers are

Conservative Garbage Collection

- But, it is okay/safe to be conservative
  - If a memory word looks like a pointer, it is considered a pointer
    - it must be aligned
    - it must point to a valid address in the data segment
  - All such pointers are followed and we overestimate the reachable objects

- But, we still can’t move objects as we can’t update pointers to them
  - What if what we thought to be a pointer is actually a number?

Reference Counting [Collins 1960]

- Rather than wait for memory to run out, 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 assignment operation has to manipulate the reference count

Implementation of Reference Counting

- **new** returns an object with a reference count of 1

- If x points to an object, let **rc(x)** refer to the object's reference count

- Every assignment **x := y** must be changed

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

Reference Counting: Evaluation

- Advantages
  - Easy to implement
  - Collects garbage incrementally without large pauses in the execution

- Disadvantages
  - Manipulating reference counts at each assignment is very slow
  - Cannot collect circular structures

Garbage Collection: Evaluation

- Automatic memory management avoids some serious storage bugs

- But, it takes away control from the programmer
  - e.g., layout of data in memory
  - e.g., when is memory deallocated

- Most GC implementations stop the execution during collection
  - not acceptable in real-time applications

Garbage Collection: Evaluation

- Garbage collection is going to be around for a while

- There are advanced garbage collection algorithms
  - **Concurrent**: allow the program to run while collection is happening
  - **Generational**: do not scan long-lived objects at every collection
    - JVM uses this kind
  - **Parallel**: several collectors working in parallel