Automatic Memory Management

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

- C and 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 lead to a visible effect far away in time and program text from the source

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

- When an object is created, unused space is automatically allocated
  - In Cool, new objects are created by new X
- 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 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:

  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 <u>reachable</u> if and only if:
  - a register contains a pointer to x, or
  - another reachable object y contains a pointer to x

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

- An unreachable object can never be used
  - such objects are <u>garbage</u>

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
```

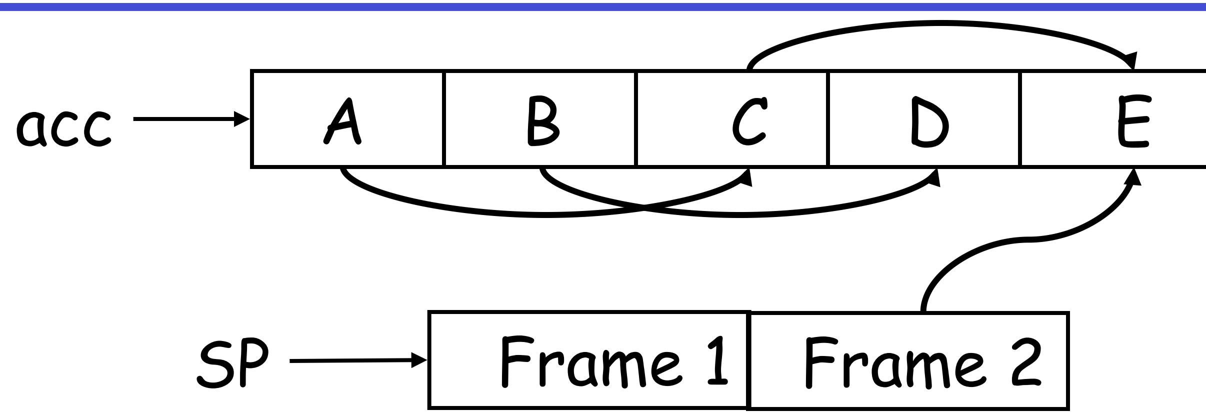
- After x ← y
  - assuming y becomes dead …
  - the first object A is unreachable
  - the object B is reachable (through x)
  - thus B is not garbage and is not collected
    - but object B is never going to be used

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.

- 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



- In Coolc we start tracing from acc and stack
  - These are the *roots*
- Note B and D are unreachable from acc and 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 garbage collection before the space actually runs out

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
  - initially the mark bit is 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
```

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

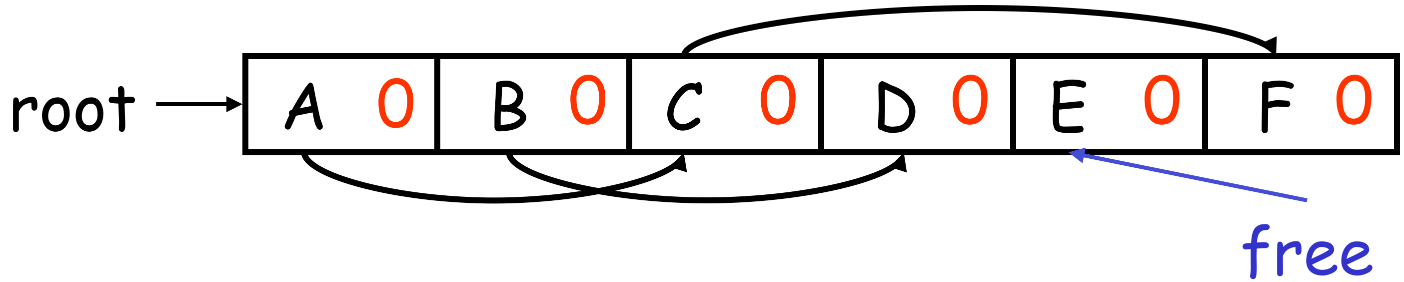
- Any such object is added to the free list

- The objects with a mark bit 1 have their mark bit reset to 0

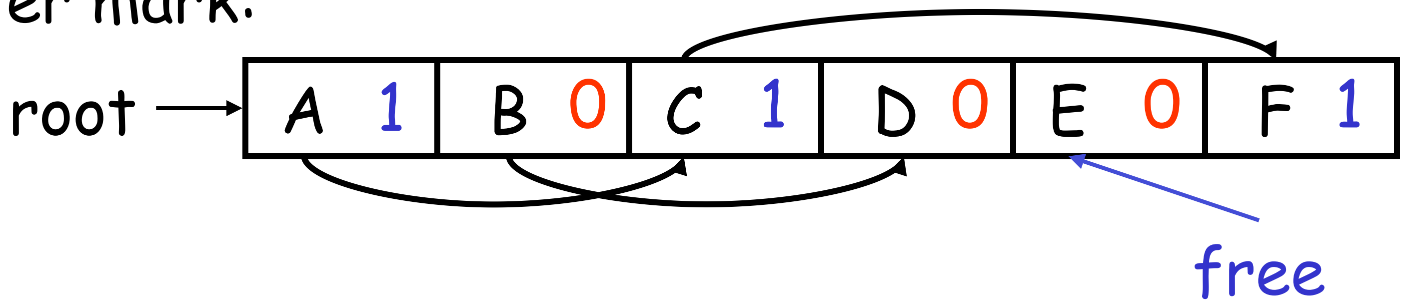
The Sweep Phase (Cont.)

```
(* 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
```

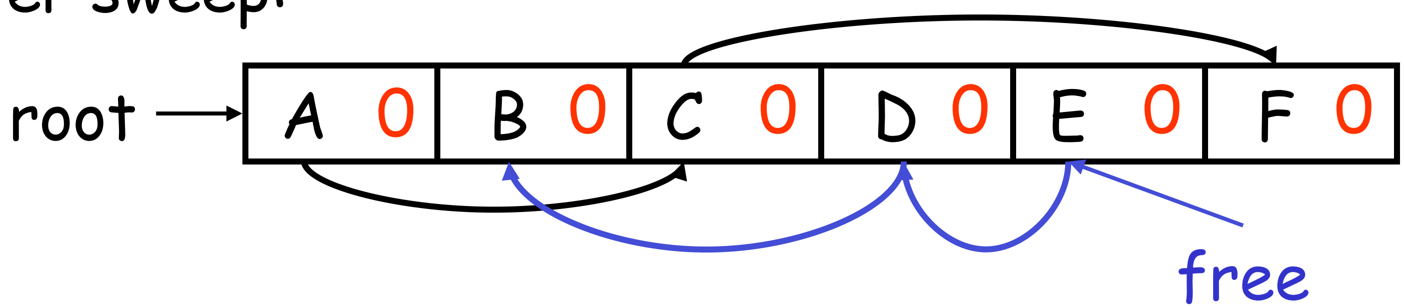
Mark and Sweep Example



After mark:



After sweep:



Details

- While conceptually simple, this algorithm has a number of tricky details
  - 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
  - the size of the todo list is unbounded so we cannot reserve space for it a priori

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

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

Evaluation of Mark and Sweep

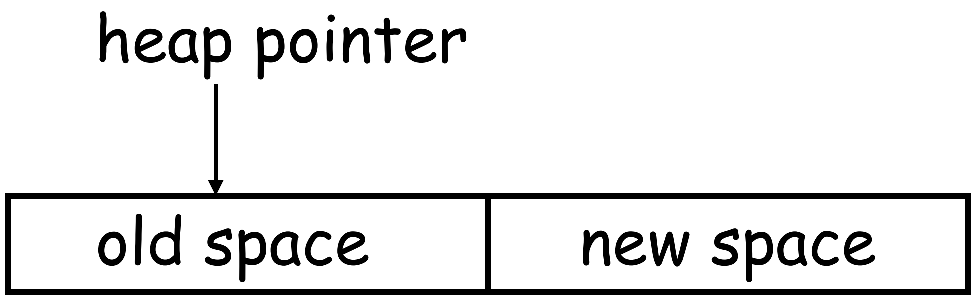
- Space for a new object is allocated from the new list
  - a block large enough is picked
  - an area of the necessary size is allocated from it
  - the left-over is put back in the free list

- Mark and sweep can fragment the memory

- Advantage: objects are not moved during GC
  - no need to update the pointers to objects
  - works for languages like C and 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



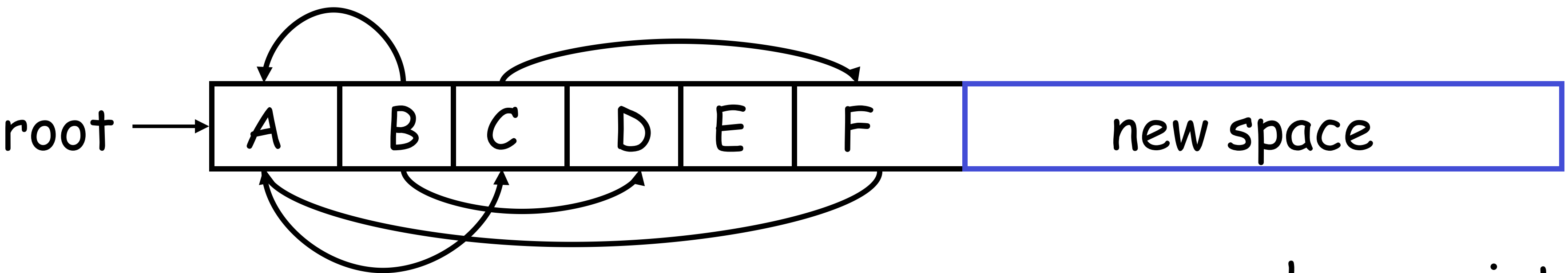
- The heap pointer points to the next free word in the old space
  - allocation just advances the heap pointer

Stop and Copy Garbage Collection

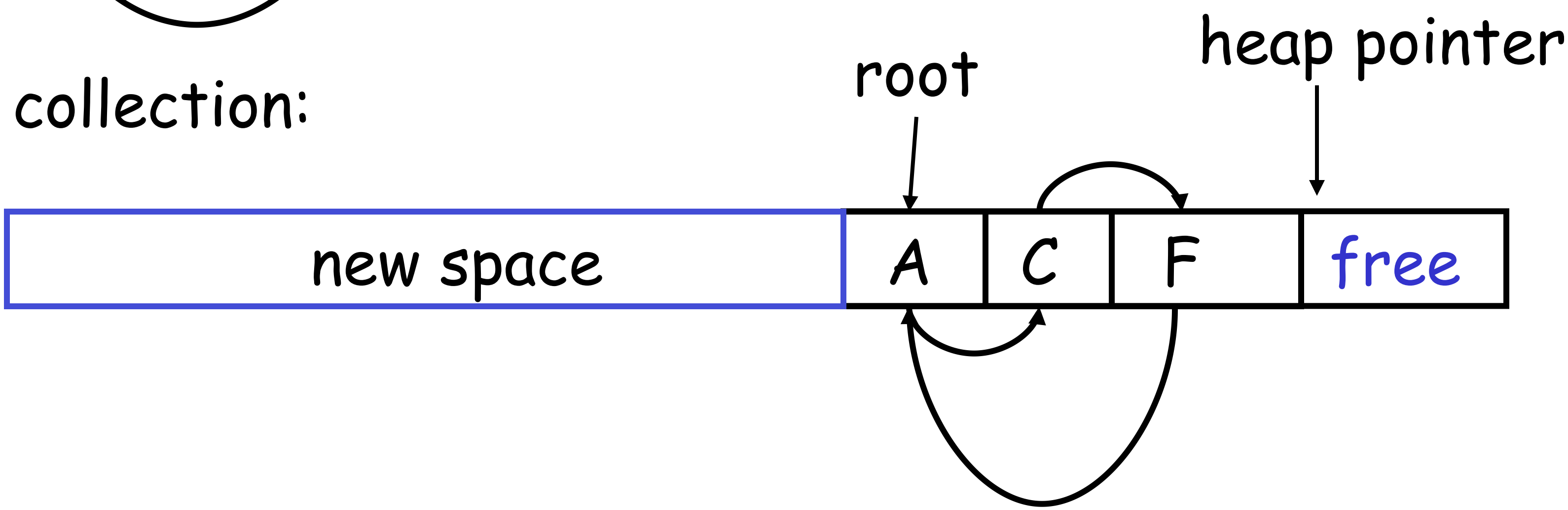
- Starts when the old space is full
- Copies all reachable objects from old space into new space
  - garbage is left behind
  - after the copy phase the new space uses less space than the old one before the collection
- After the copy the roles of the old and new spaces are reversed and the program resumes

Example of Stop and Copy Garbage Collection

Before collection:



After collection:



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 fix ALL pointers pointing to it!

- As we copy an object we store in the old copy a <u>forwarding pointer</u> 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 (Cont.)

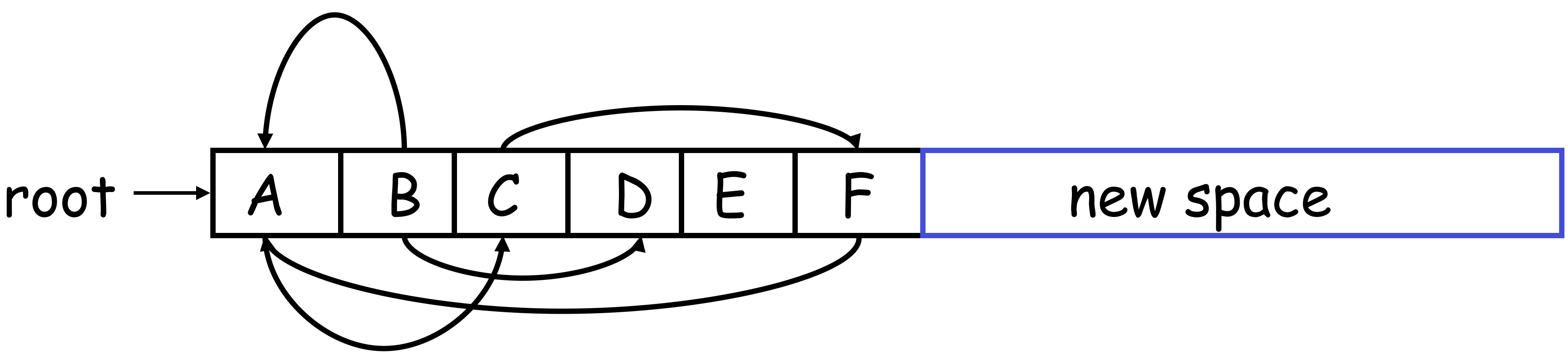
- We still have the issue of how to implement the traversal without using extra space
- The following trick solves the problem:
  - partition the new space in three contiguous regions

copied objects whose pointer fields were followed

copied objects whose pointer fields were NOT followed

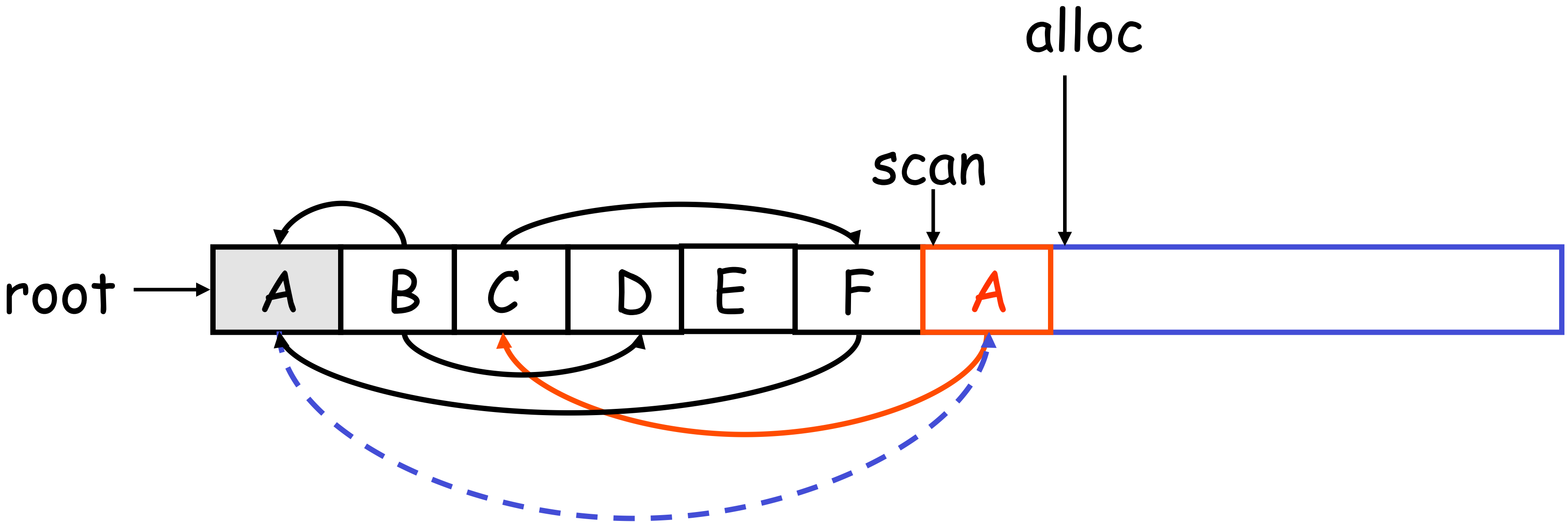
Stop and Copy. Example (1)

- Before garbage collection



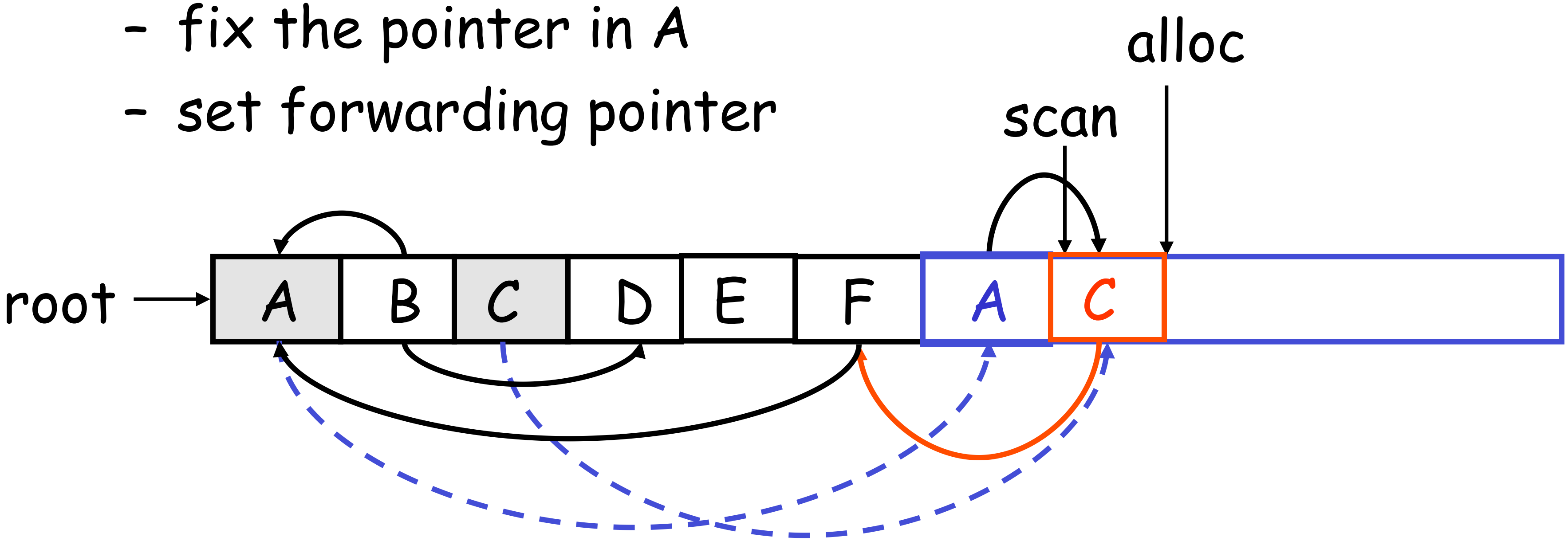
Stop and Copy. Example (2)

- Step 1: Copy the objects pointed to by roots and set forwarding pointers



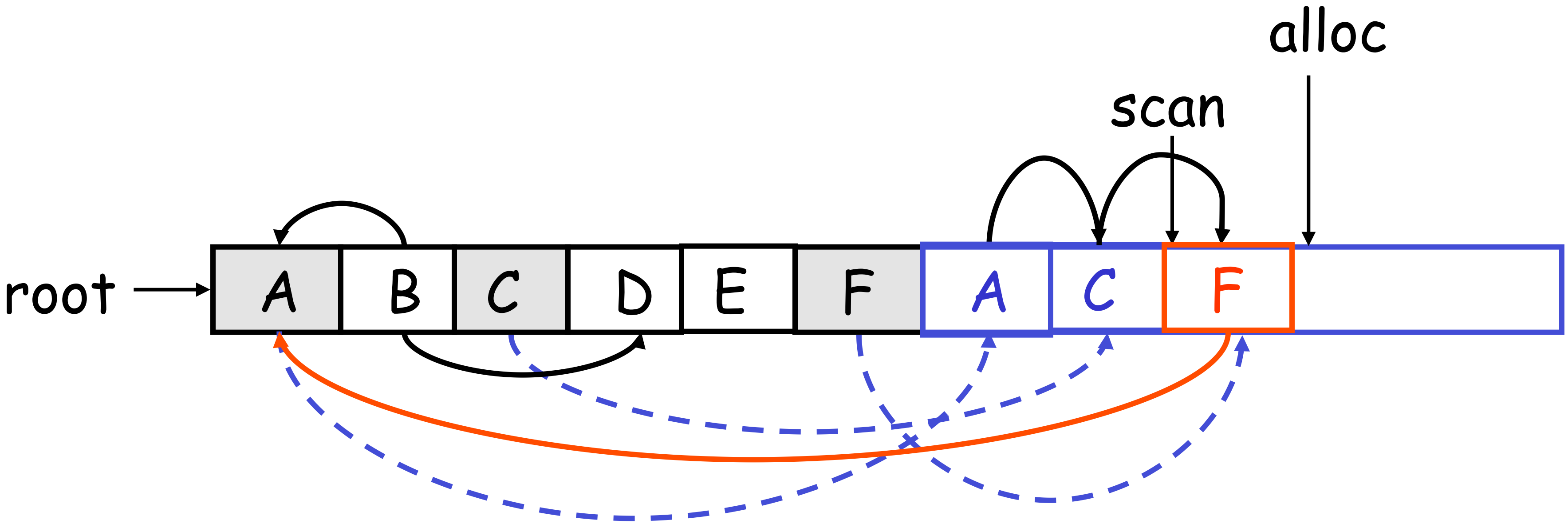
Stop and Copy. Example (3)

- 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



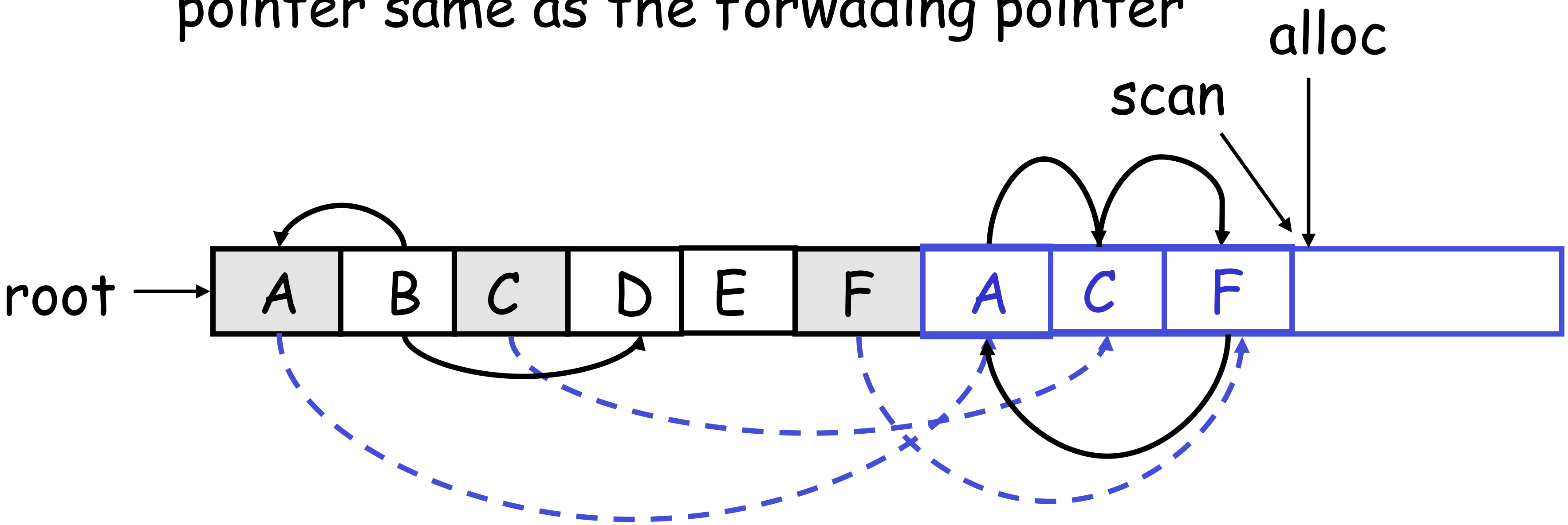
Stop and Copy. Example (4)

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



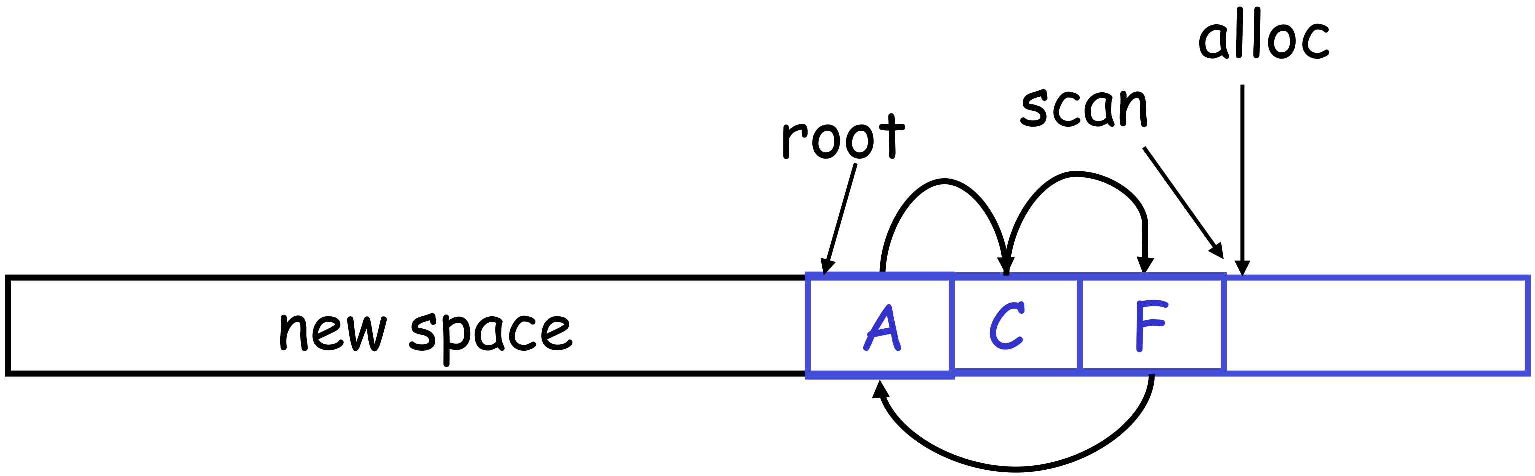
Stop and Copy. Example (5)

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

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
  - 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

Conservative Garbage Collection

- But it is Ok to be <u>conservative</u>:
  - 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 set of reachable objects

- But we still cannot move objects because we cannot update pointers to them
  - what if what we thought is a pointer is actually an account number?

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 assignment operation manipulates the reference count

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:

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

Evaluation of Reference Counting

- Advantages:
  - easy to implement
  - collects garbage incrementally without large pauses in the execution
- Disadvantages:
  - cannot collect circular structures
  - manipulating reference counts at each assignment is very slow

Evaluation of Garbage Collection

- Automatic memory management prevents serious storage bugs

- But reduces programmer control
  - e.g., layout of data in memory
  - e.g., when is memory deallocated

- Pauses problematic in real-time applications
- Memory leaks possible (even likely)

Evaluation of Garbage Collection

- Garbage collection is very important

- Researchers are working on advanced garbage collection algorithms:
  - concurrent: allow the program to run while the collection is happening
  - generational: do not scan long-lived objects at every collection
  - parallel: several collectors working in parallel