11. Memory Allocation

2025 Fall

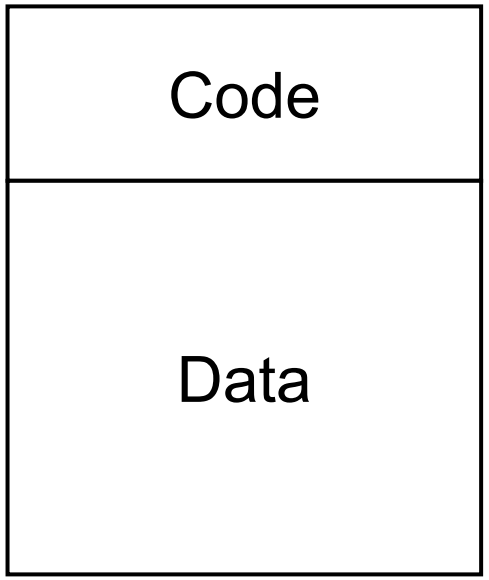
Hunjun Lee

Hanyang University

Review: Storage Organization

- **The compiler is responsible for:**
  - Generating code
  - Orchestrating use of the data area

Storage Organization



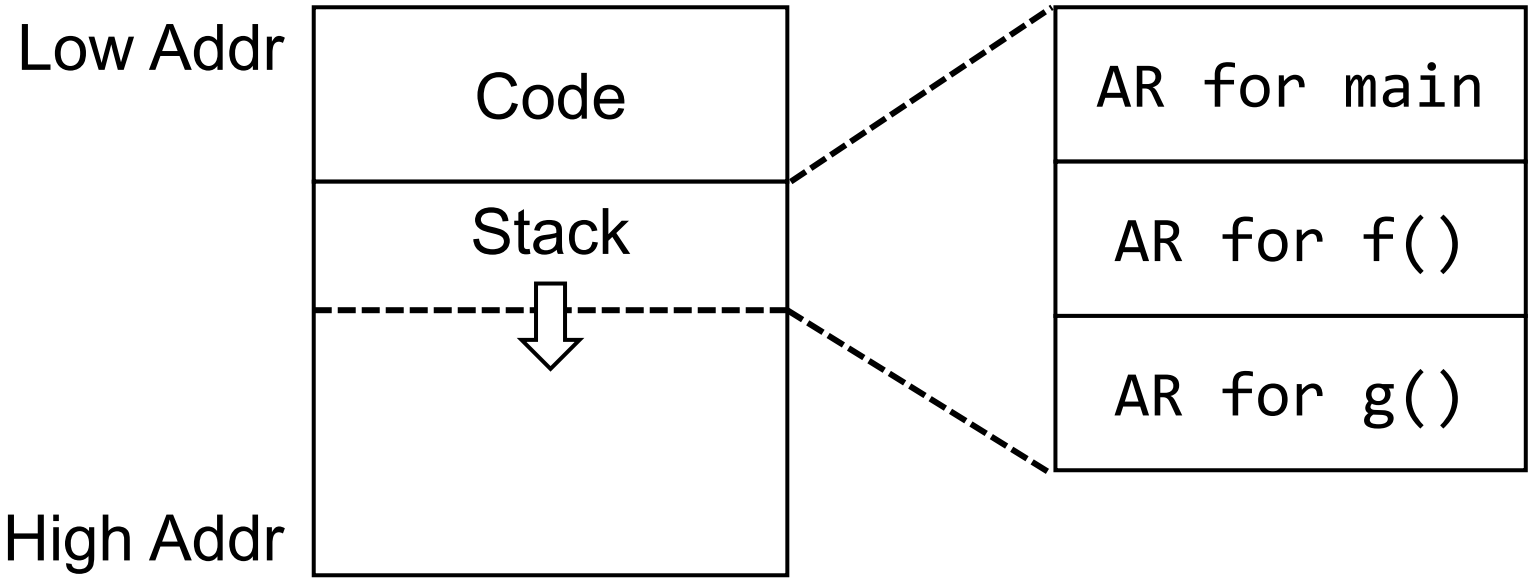
Low Addr

High Addr

The code manipulates the data in memory

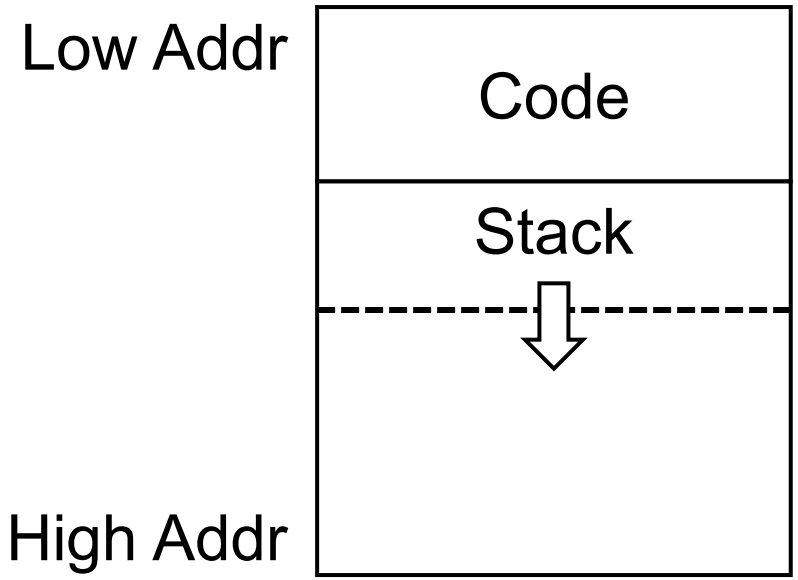
Review: Stack Management

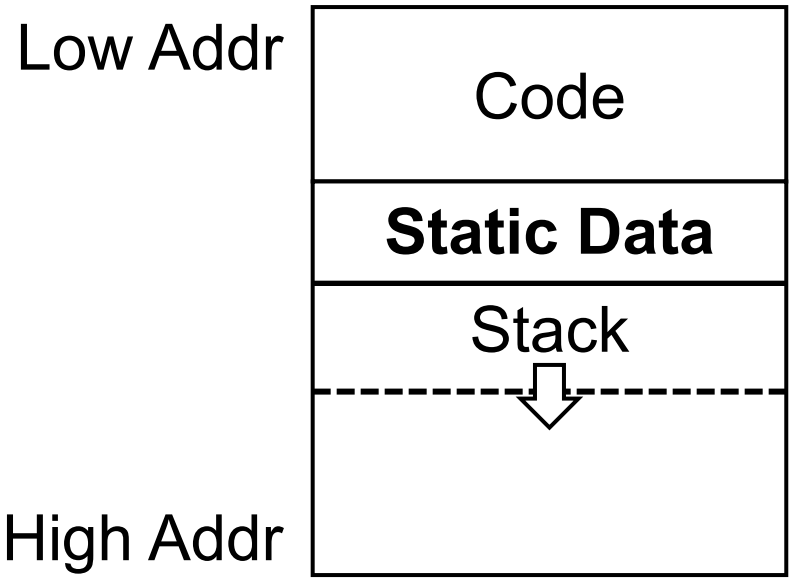
- **Stack data is stored starting from the low address, which grows downwards**
- **The information to manage one activation is called activation record (AR) or frame**



Review: Global Memory

- **We allocate the static variables after the code memory**

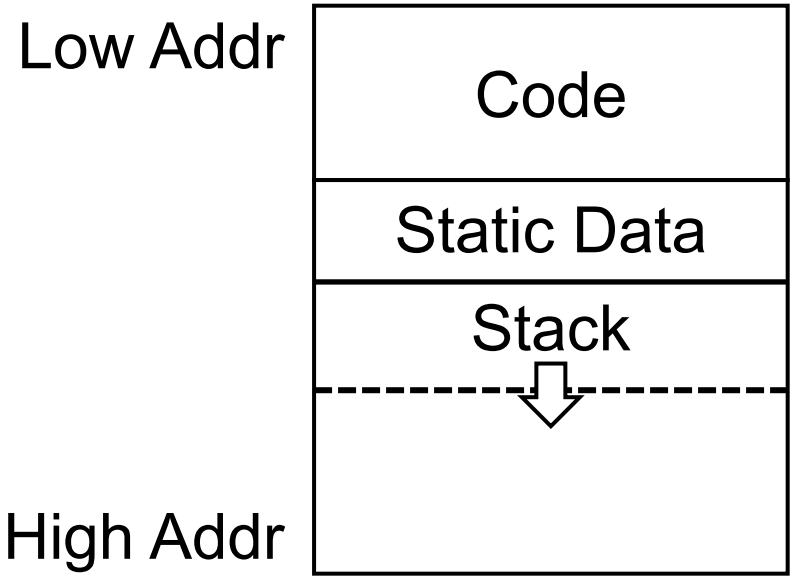


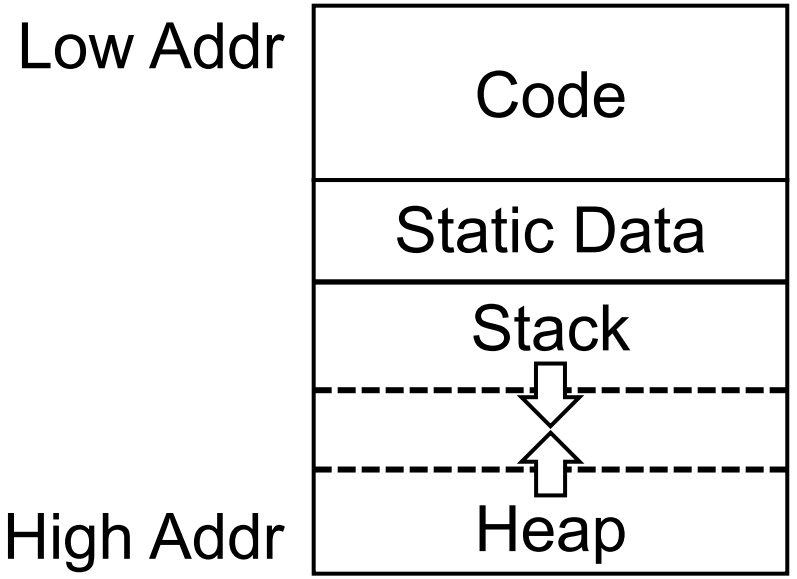


Review: Dynamic Allocation

- **The dynamically allocated value outlives the procedure that creates it (unless deleted by free)**
- **We rely on heap to store the dynamically allocated data**

Question: How to manage memory for heap area?





Manual Memory Management

- **Storage management is still a hard problem in modern programming**

- **C and C++ demand manual storage management that are prone to bugs**
  - Forget to free the unused memory
  - Dereferencing a dangling pointer, etc

- **Storage bugs are hard to find in complex systems**
  - The bugs occur later in time

Automatic Memory Management

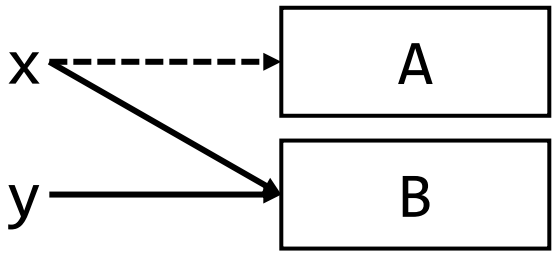
- **Automatic memory management has become popular with the advent of Java (1990s)**

- **The main task in automatic memory management is to "free (deallocate) memory space for the data that will never be used again"**
  - Upon identifying the data that will never be used, the memory manager can simply free them

Object Reachability - 1

- **Reachability: the objects it can find (there is a pointer to that object)**

```
Obj * x = new Obj(A);
Obj * y = new Obj(B);
x = y;
```



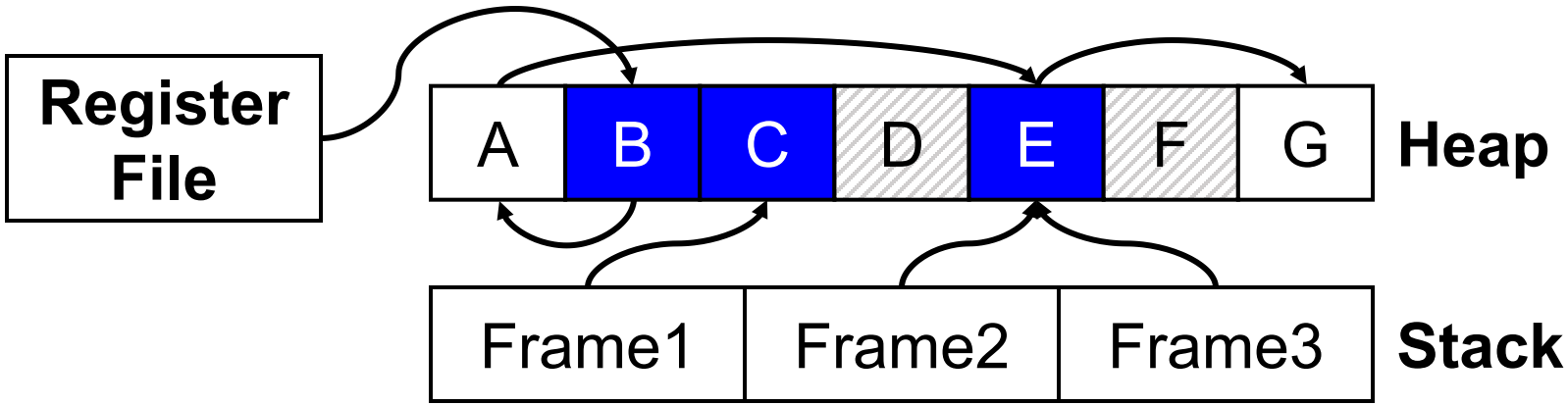
This object is not reachable anymore

  - An object x is reachable if and only if a register (or the pointer in the stack) points to x or another reachable object y contains a pointer to x

  - An unreachable object can never be used ➔ They are deemed as garbage

Object Reachability - 2

- **Object reachability is determined by traversing the data from the root**
  - The root is the **"pointer in the register file, stack, global mem, …"**
  - Traverse the pointer from the reachable objects



Garbage Collection

- **Garbage collection consists of two steps**
  - Allocate space as needed for new objects
  - When the space runs out, free unreachable objects

- **Some strategies collect garbage before the space runs out**

Variants in Garbage Collection

- Mark and Sweep
- Stop and Copy
- Conservative Collection
- Reference Counting

Mark and Sweep

- **The mark and sweep garbage collector consists of two phases**
  - In the mark phase: trace reachable objects and mark them
  - In the sweep phase: collect garbage objects

- **Reserve an extra bit for sweeping**
  - Initially marked as 0 and set to 1 for the reachable objects in the mark phase

Mark Phase Pseudocode

```
set todo = {all pointers accessible from root}
while (todo != empty)
      pop v in todo;
      if mark(v) = 0 then
            mark(v) = 1;
            for v_i for the pointers in v
                  todo = todo + {v_i}
```

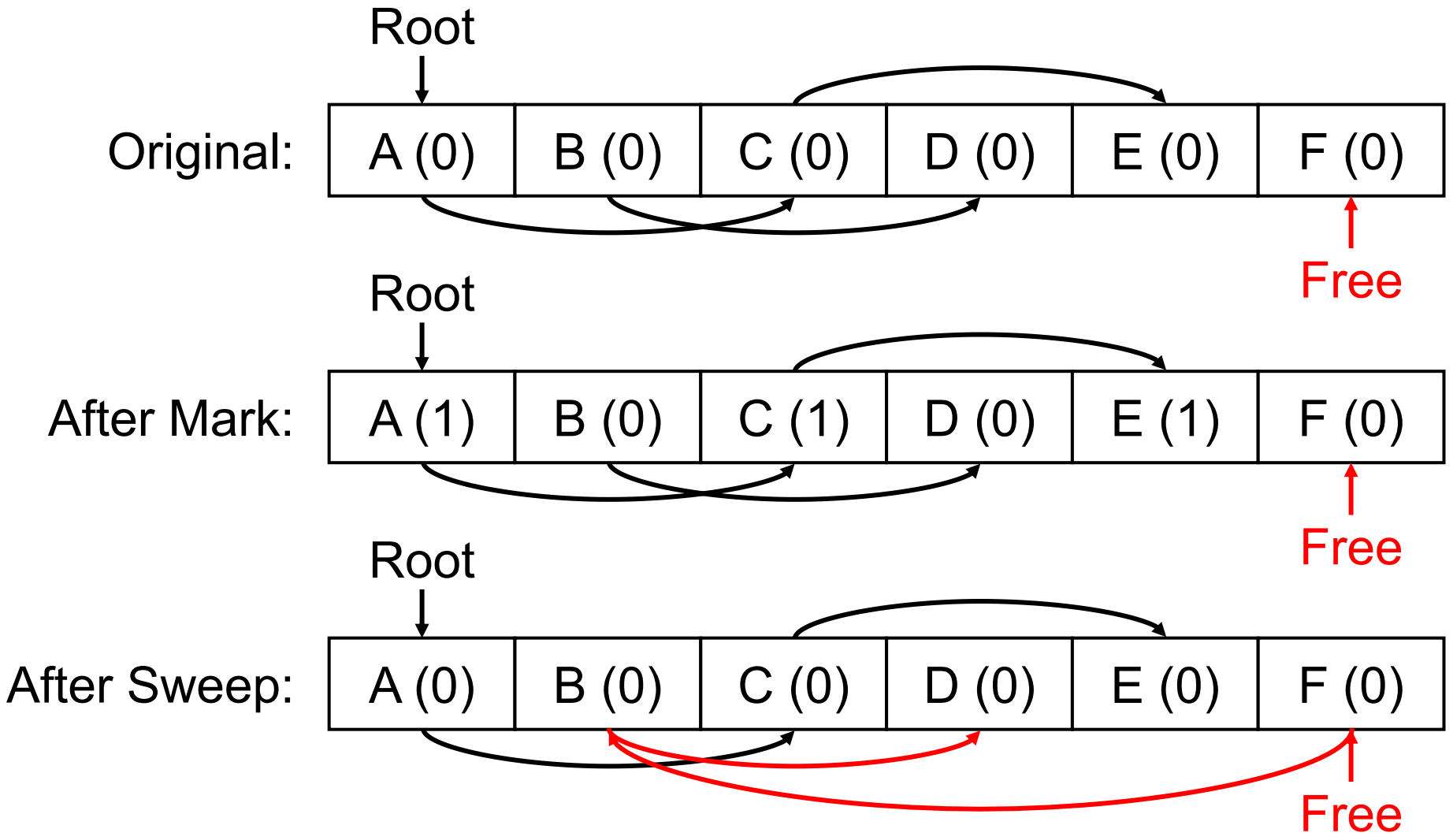
Sweep Phase

- **The sweep phase scans the heap and look for objects marked with 0**
  - They represent unreachable objects and should be freed from the memory
- **These objects (or memory regions) are added to the free list**
  - The free list is later used for memory allocation
- **Objects with mark 1 are reset to mark 0**

Sweep Phase Pseudocode

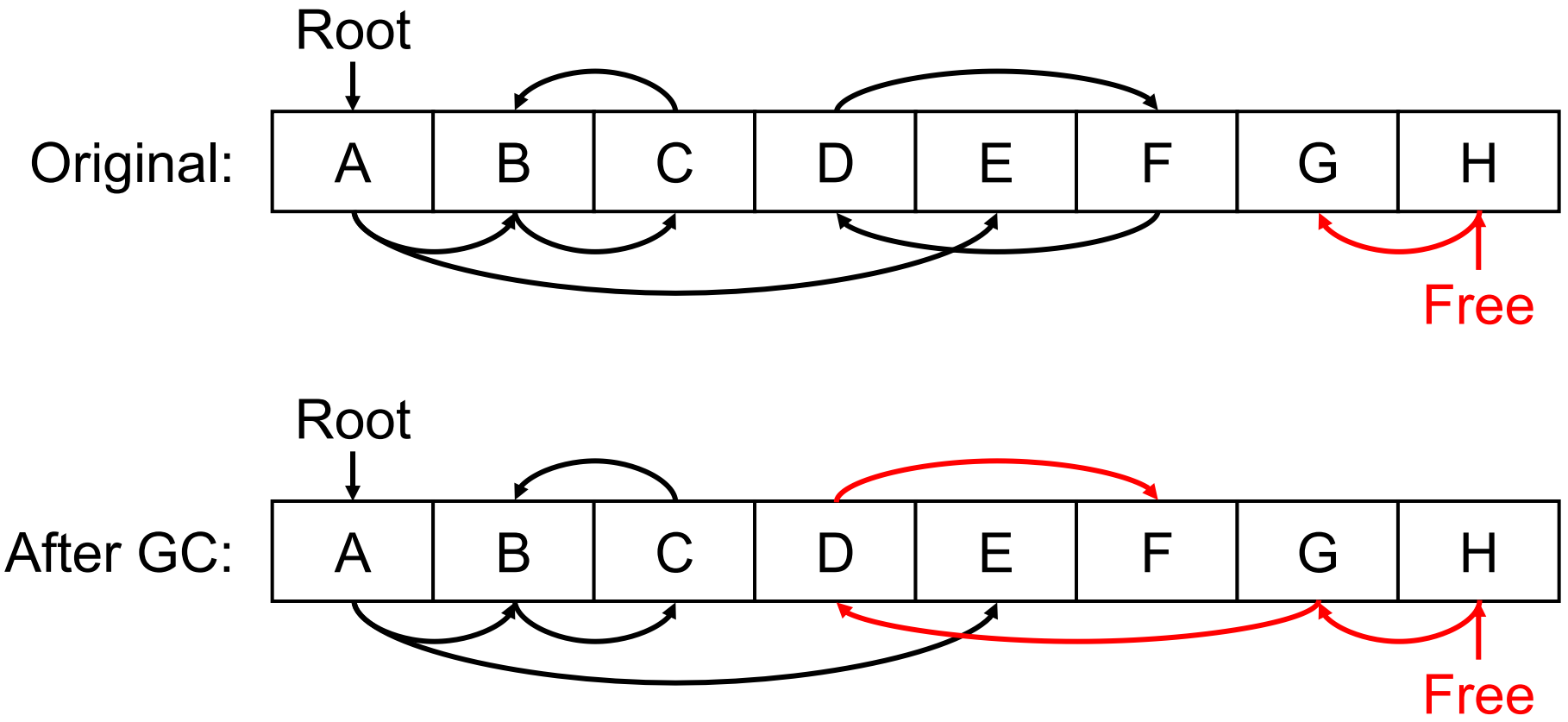
```
// sizeof(p) is the size of the block starting at p

p = bottom(heap)
while p < top(heap)
      if mark(p) = 1
            mark(p) = 0;
      else
            add block p…(p+sizeof(p)-1) to freelist
      p = p + sizeof(p)
```

Example


Root
Original:
A (0)
B (0)
C (0)
D (0)
E (0)
F (0)
Free
Root
After Mark:
A (1)
B (0)
C (1)
D (0)
E (1)
F (0)
Free
Root
After Sweep:
A (0)
B (0)
C (0)
D (0)
E (0)
F (0)
Free

Class Exercise

- **Examine what happens after mark and sweep**



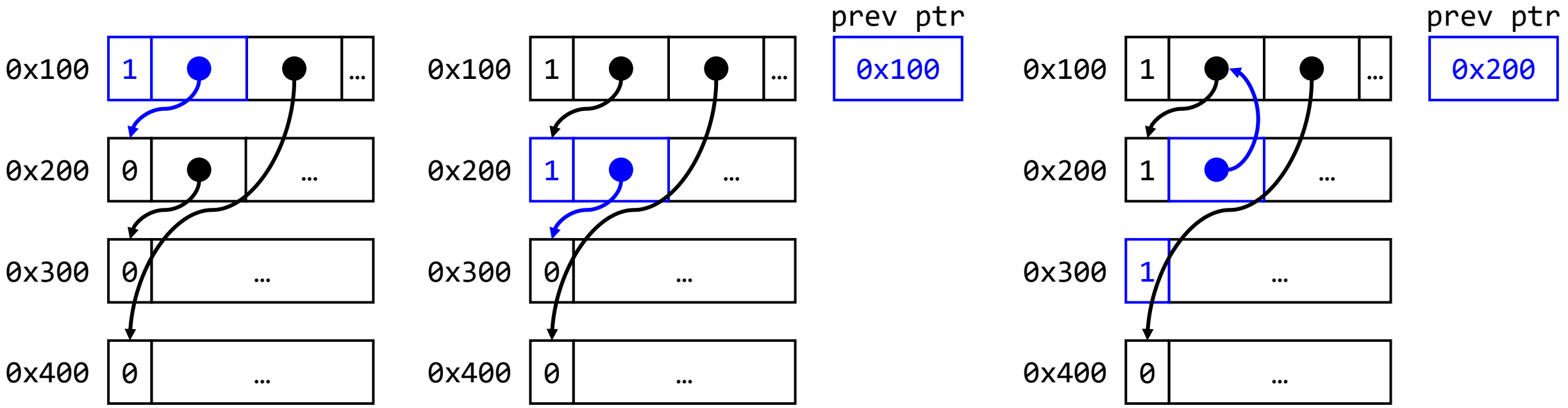
Practical Implementation Issues

- **There are a number of tricky details in implementing a GC**

- **A serious problem with the mark phase:**
  - It is typically invoked when we are out of space (in the storage)
  - Yet, it needs to construct the todo set
  - **Therefore, the todo set should not consume an extra memory**

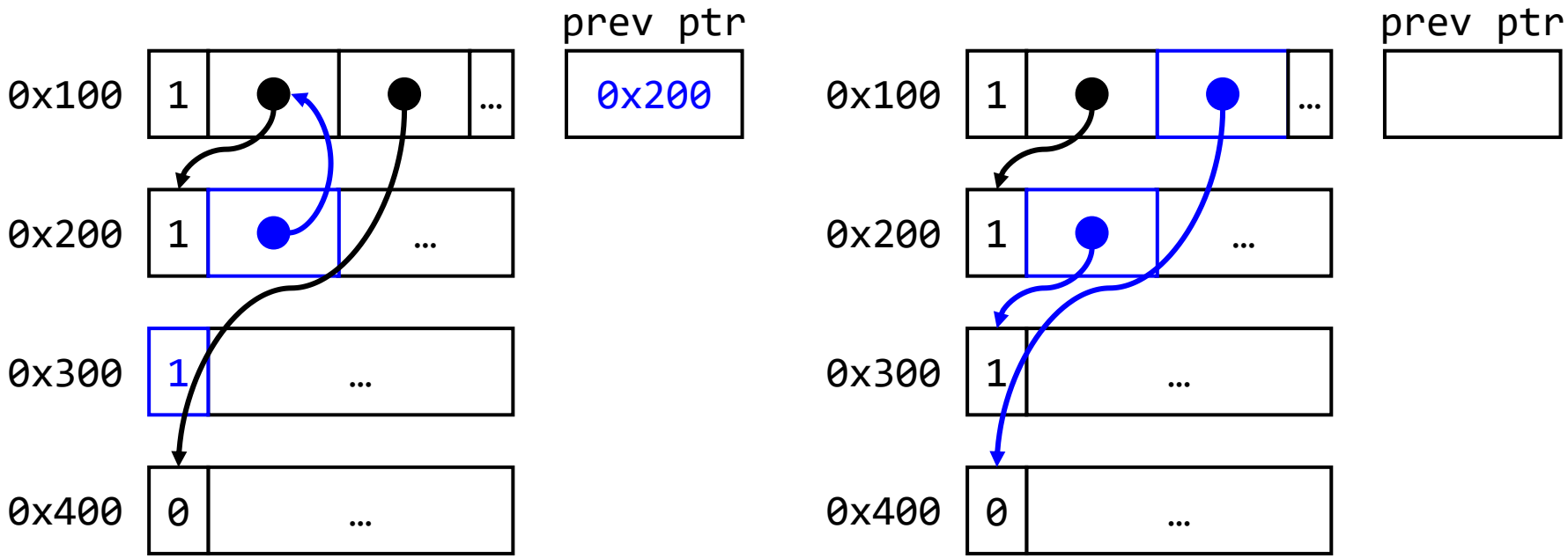
Memory Issues in Mark and Sweep

- **Trick: auxiliary data stored in the objects (pointer reversal)**
  - Make the pointer point to its parent
  - Keep the free list w/o consuming memory: the free list is stored in the free objects themselves



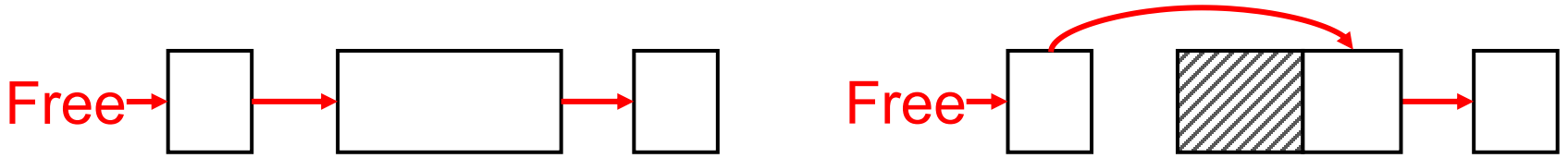
Memory Issues in Mark and Sweep

- **Trick: auxiliary data stored in the objects (pointer reversal)**
  - Make the pointer point to its parent
  - Keep the free list w/o consuming memory: the free list is stored in the free objects themselves



Memory Allocation in Mark and Sweep

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

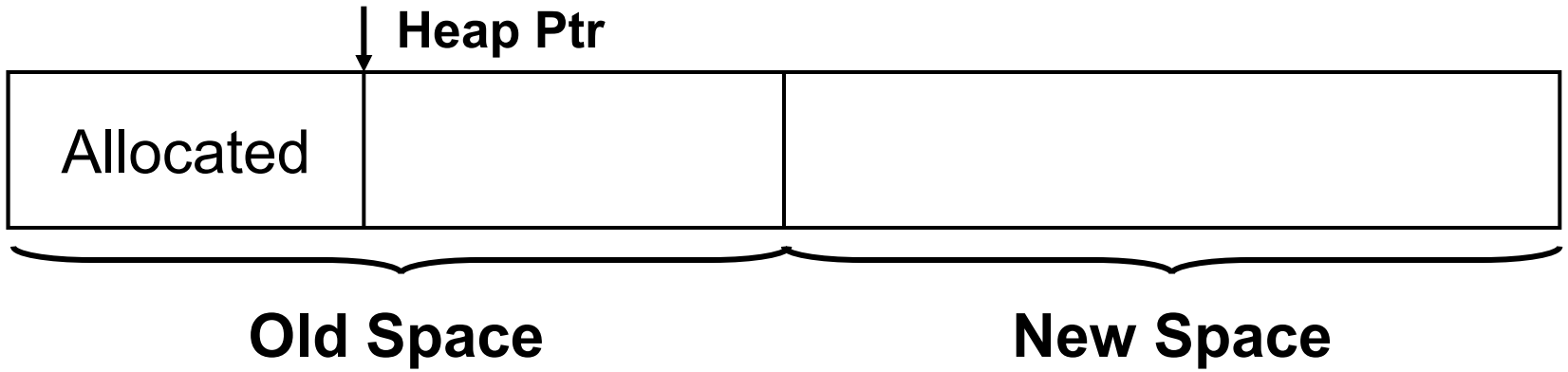


- **Mark and sweep incurs memory fragmentation**
  - We need to merge blocks whenever possible
- **Advantage: objects are not moved during GC**
  - No need to update the pointers to objects
  - Works well for C / C++

Stop and Copy Organization

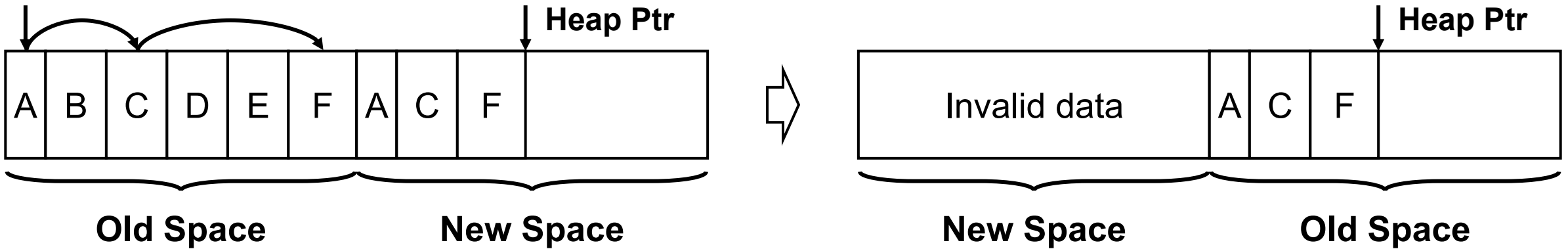
- **In stop and copy, the memory is organized into two areas**
  - Old space: used for allocation
  - New space: used as a reserve for GC

- **The heap pointer points to the next free word in the old space**
  - Allocate advances the heap pointer



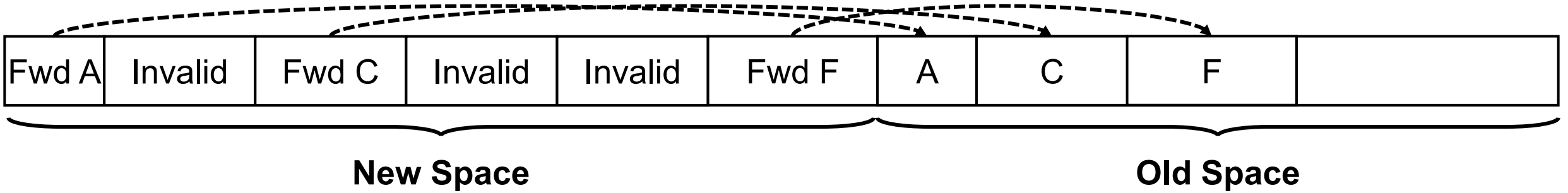
Stop and Copy from Old to New

- **Starts GC when the old space is full**
  - Identifies reachable objects
  - Copies all reachable objects into the new space (contiguously)
  - Garbage is left behind in the old space
  - Reverse the role of old space and new space



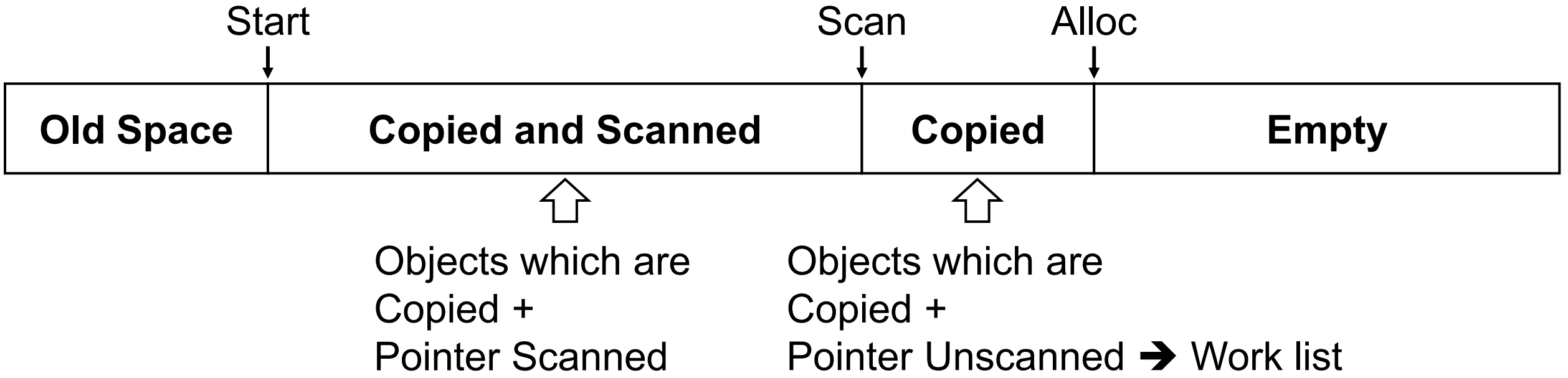
Pointer Adjustment

- **We change the location of the allocated data**
  - We have to fix all pointers pointing to the data ➔ this can be extremely slow
- **We perform a lazy update of the pointer**
  - When copying the data from old to new space, (in the old space) we overwrite where the data is relocated (forwarding pointer)
  - If the pointer reference the new space (which was old space), we update the pointer using the pointer in the new space



Simple Sweep in Stop and Copy

- **We use a special trick to minimize mark phase**
  - We split the new space using three pointers: start, scan, and alloc

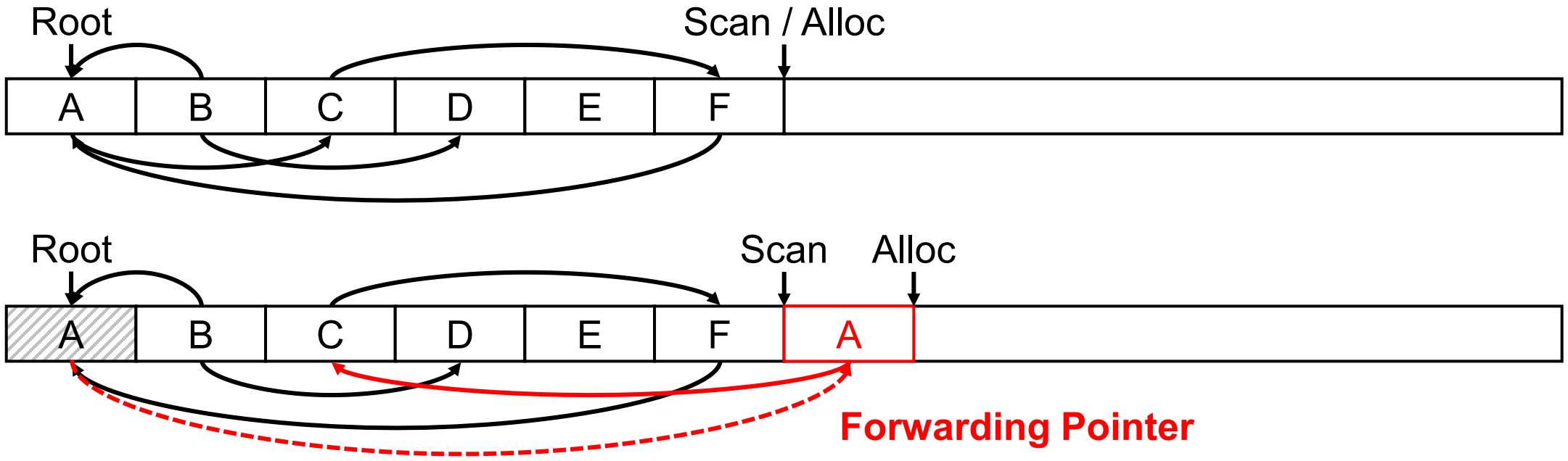


Recall: Mark Phase Pseudocode

```
set todo = {all pointers accessible from root}
while (todo != empty)
      pop v in todo;
      if mark(v) = 0 then
            mark(v) = 1;
            for v_i for the pointers in v
                  todo = todo + {v_i}
```

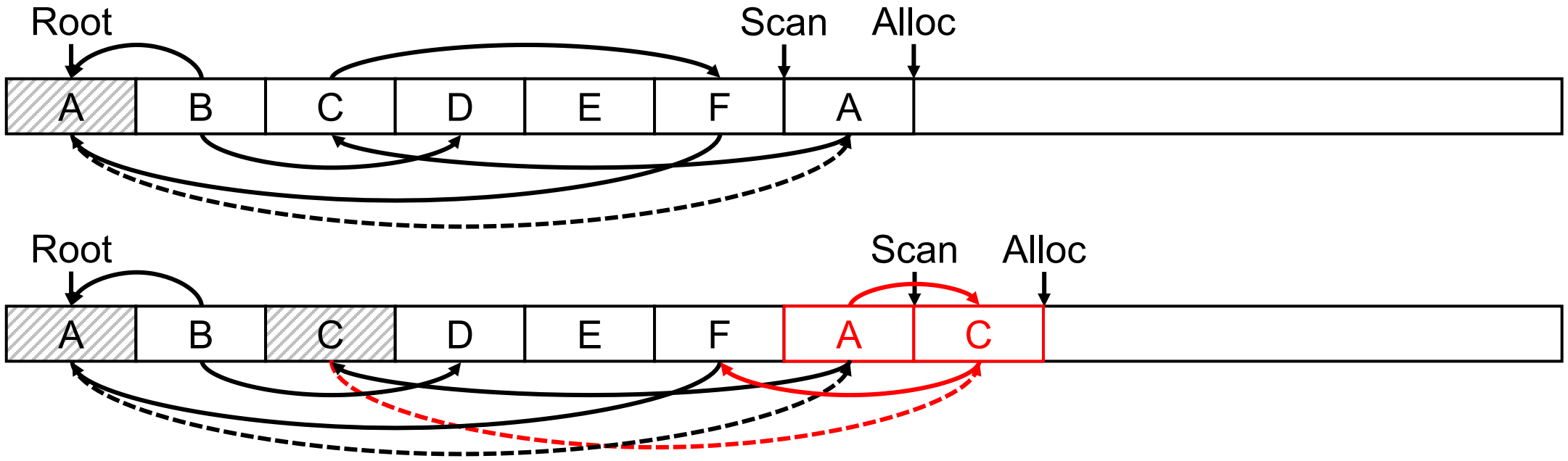
Stop and Copy Implementation - 1

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



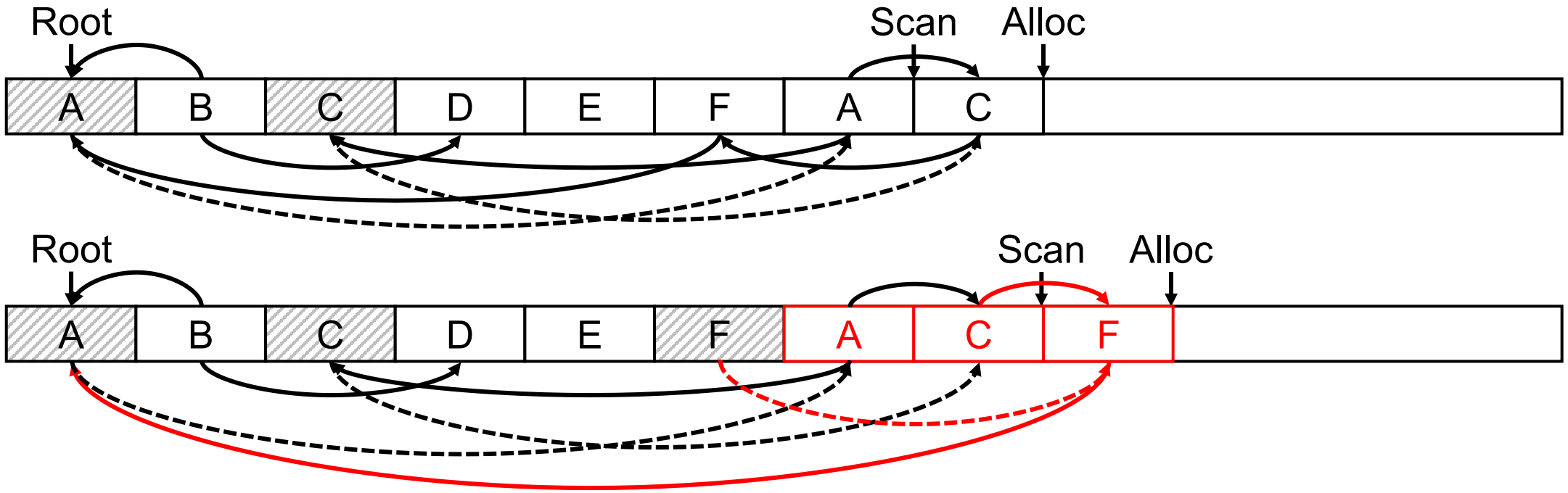
Stop and Copy Implementation - 2

- **Step #2: Follow the pointer in the unscanned object (i.e., A)**
  - Copy the pointed objects and set forwarding pointer
  - Fix the pointer in A



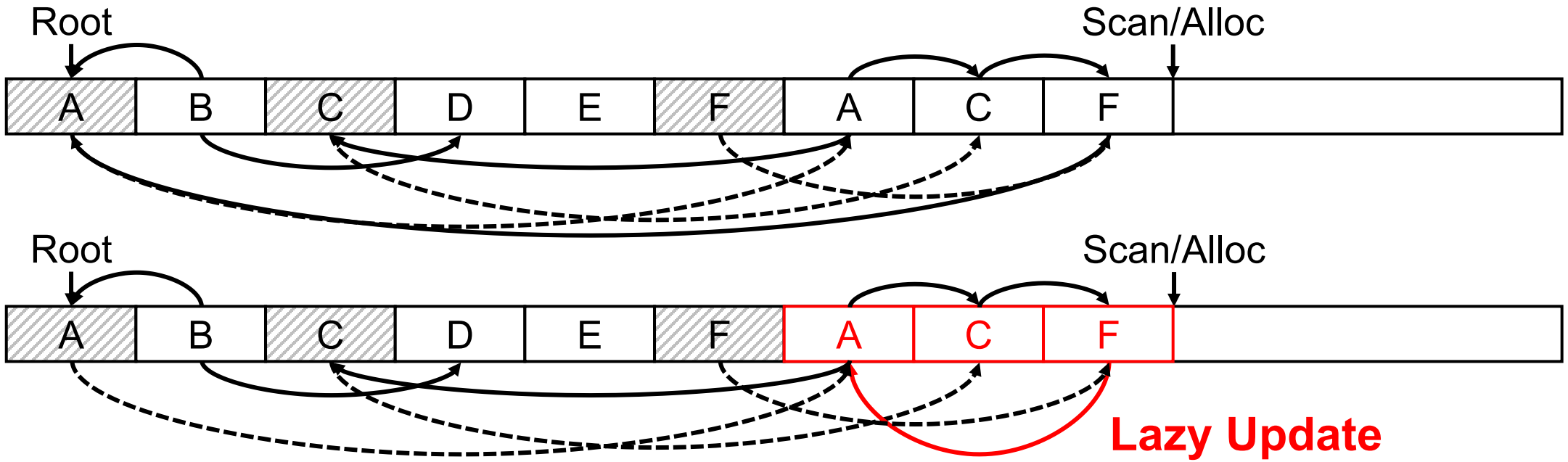
Stop and Copy Implementation - 3

- **Repeat Step #2 until the end**



Stop and Copy Implementation - 4

- **Repeat Step #2 until the end (where the scan is the same as alloc)**



Stop and Copy Implementation - 5

- **We should also copy any objects pointed by the stack and register file (root)**
  - This can be an expensive operation
  - Why not rely on lazy update? ➔ Can cause errors if the GC happens again before the object is referenced

Stop and Copy

- Stop and copy is generally believed to be the fastest GC technique
- Allocation is very cheap (just incrementing the heap pointer)
- Collection is relatively cheap (especially when there is a lot of garbage) as is only touches reachable objects
- C/C++ do not allow copying and changing the location

Garbage Collection and Types

- Finding reachable objects involve finding pointers in an object and scanning the pointed objects

- In some languages (e.g., C/C++), the pointers are dynamically casted (it is hard to know which field indicates pointer)

Conservative Collection

- **Just, make everything conservative**
  - If a memory word (data) looks like a pointer, assume that it is a pointer
    - It must be aligned and must be a valid address
  - This overestimates the reachable objects (may suffer from reduced memory space)

- **But still, conservative collection does not allow the use of "stop and copy" for C/C++**

Reference Counting

- **Rather than waiting for the memory to be exhausted, free object when there are no more pointers to it**
- **In each object, store the number of pointers to that object**
  - This is called a reference count
- **Each assignment operation manipulates the reference count**

Counting Method

- **"New" returns an object with reference count 1**
  - RC(x) indicates the reference count of x

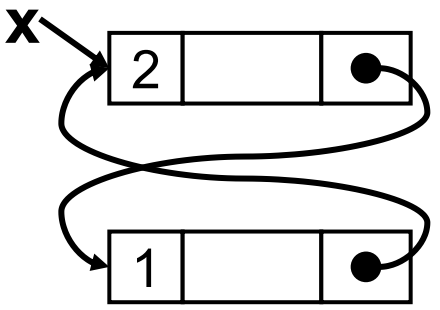
- **Assume x, y point to objects o, p ➔ every assignment x = y; becomes:**
  - rc(p) = rc(p) + 1;
  - rc(o) = rc(o) – 1;
  - if (rc(o) == 0) {free o;}
  - x = y;

Reference Counting Overview

- **Reference counting is easy to implement**

- **Collects garbage incrementally without large pauses in the execution**
  - Better to keep QoS (Quality-of-Service)

- **There are several disadvantages**
  - Cannot collect circular structures
  - Slow as each assignment manipulates reference counts



Needs to set the pointer to NULL before x is not used

Overview of the Automatic Memory Management

- **Automatic memory management prevents serious storage bugs**

- **But, there are several disadvantages**
  - Reduces the programmer control (memory layout and when the memory is deallocated)
  - May suffer from problems in real-time applications
  - Sub-optimal garbage collection (memory leaks)
    - The programmers need to explicitly change the reference of unused objects
      - x = NULL (to make it unreachable)

Optimizing Garbage Collection

- **Garbage collection is very important**

- **There are advanced strategies to enable fast GC**
  - Concurrent: allow program to run during GC
  - Generational: do not scan long-lived objects at every collection
  - Real time: bound the length of the pauses
  - Parallel: several collectors working at once