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 (memory leak)
  - dereferencing a dangling pointer (use freed memory, security)
  - free freed memory
  - 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(Rust)

- 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 several well-known techniques for performing completely automatic memory management
- Became mainstream with the popularity of Java
- Programming languages provide this feature
  - Reference counting (runtime): Python, PHP, scripting languages
  - Mark-and-Sweep (runtime): Java, C#, Go
  - Object ownership + lifetime (compile time): Rust, C++11 Smart pointer
  - Region-based (compile time): Cyclone

The Basic Idea

- **Problem**: When an object that takes memory space is created, unused space is automatically allocated
  - In Cool, new objects are created by new X
  - In C++/Java, new objects are created by new X()
  - After a while there is no more unused space
- **Key Insight into Solution:** Some space is occupied by objects that will never be used again
- This space can be freed to be reused later
  - In Cool, you are encouraged to implement an approach
  - In C++, objects can be deleted manually by delete x
  - Java, objects may be automatically deleted

The Basic Idea

- **How can we tell whether an object will "never be used again"?**
  - ✓ In general it is impossible to tell
  - ✓ We will have to use heuristic algorithms to find many (not all) objects that will never be used again
- 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 reachable if and only if:
  – A register/memory 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 referred by the program
  – These objects are called garbage
- Sound?
- Complete?

Reachability is a Safe 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)

  which objects are not reachable?
  which objects are not used?
  which objects are garbage?

  - The object A is not reachable anymore
  - The object B is reachable (through x or y)
  - Thus B is not garbage and is not collected, A is garbage
  - But object B is never going to be used

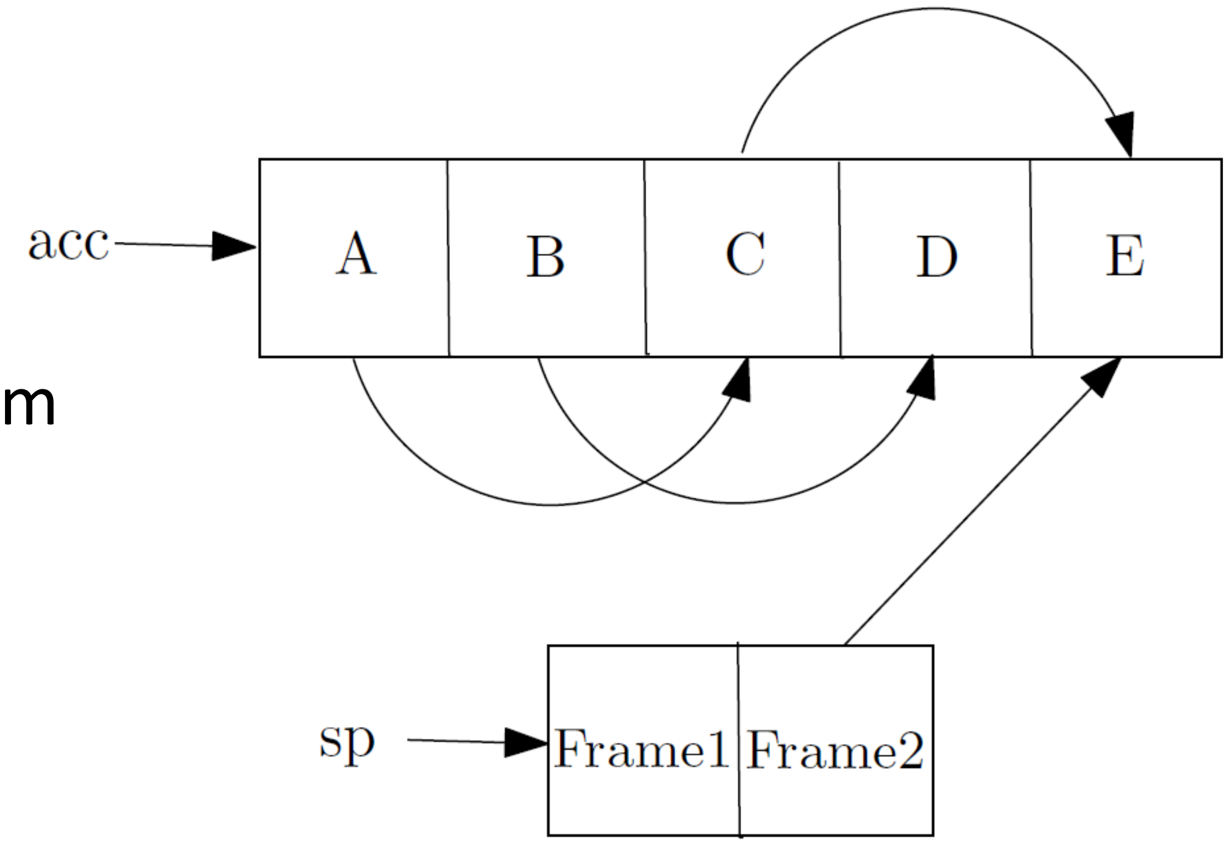
Reachability is a Safe Approximation

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 (or all registers) 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

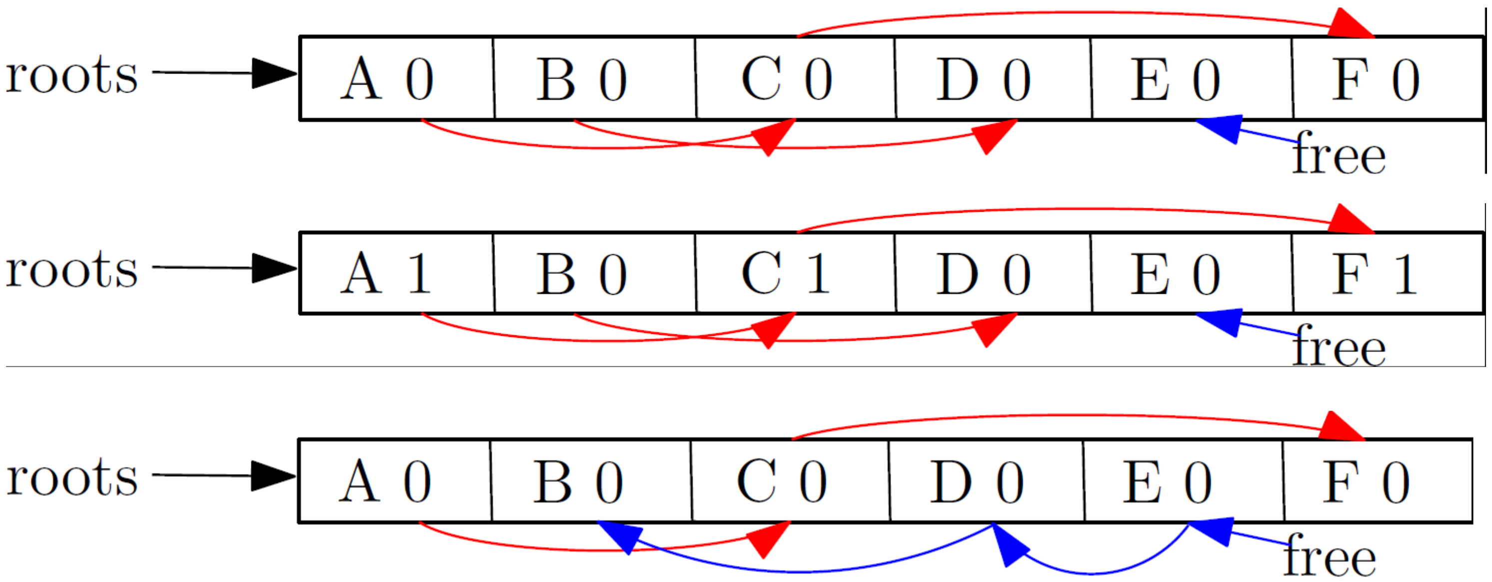


Elements of Garbage Collection

- 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 garbage collection before the space actually runs out
  - ✓ Space: the best use of available memory
  - ✓ Overhead: total time of the program
  - ✓ Pause time: the time cost of garbage collection each time
  - ✓ Program locality: same cache or pages

First Technique: 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



After mark phase

After sweep phase

The Mark Phase

```
todo = { all roots }
while todo ≠ Ø 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 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

```
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 free list
    fi
    p ← p + sizeof(p)
od
```

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

Mark and Sweep: Details

- While conceptually simple, this algorithm has a number of tricky details
  - this 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, an auxiliary data structure to perform the reachability analysis
  - the size of the todo list is unbounded so we cannot reserve space for it a priori

Any idea?

Solution: Encode the auxiliary data into 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

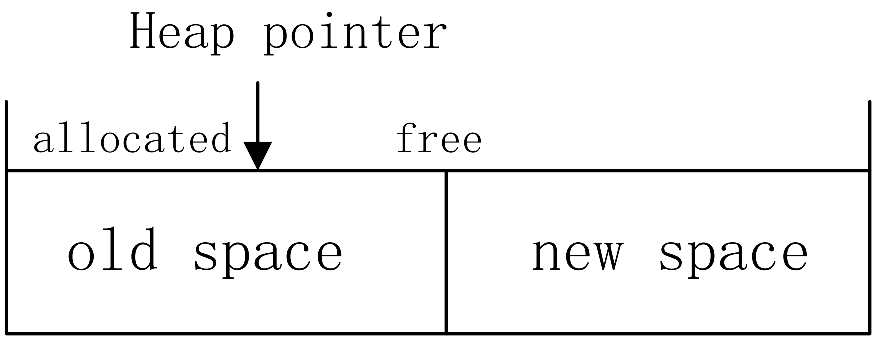
Mark and Sweep. Evaluation

- 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
- Problem?

  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 Java

Another Technique: Stop and Copy

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

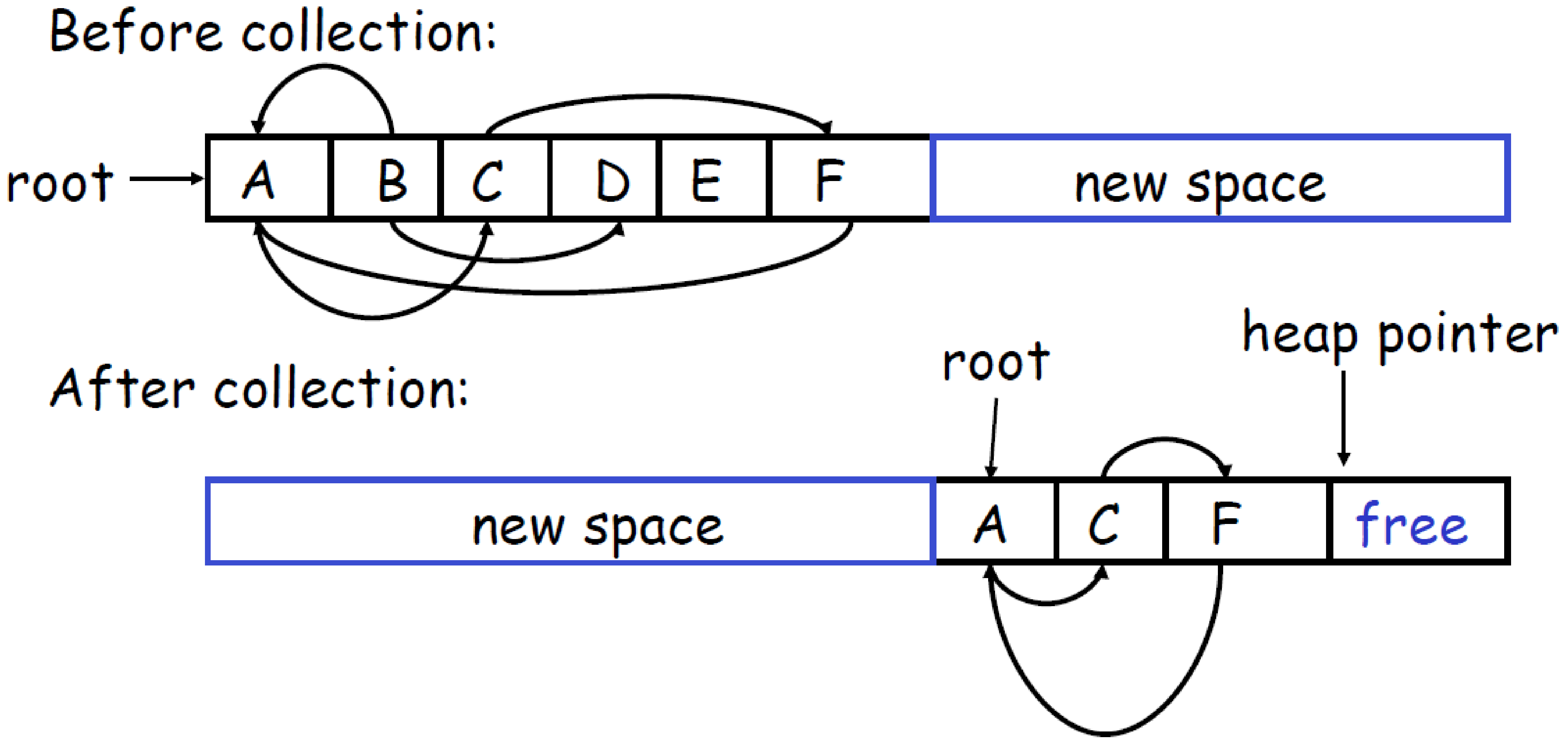


- The heap pointer points to the next free word in the old space
  - Allocation just advances the heap pointer

Stop and Copy Garbage Collection

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

Example



Before collection:
root
A
B
C
D
E
F
new space
After collection:
root
heap pointer
new space
A
C
F
free

Implementation of Stop and Copy

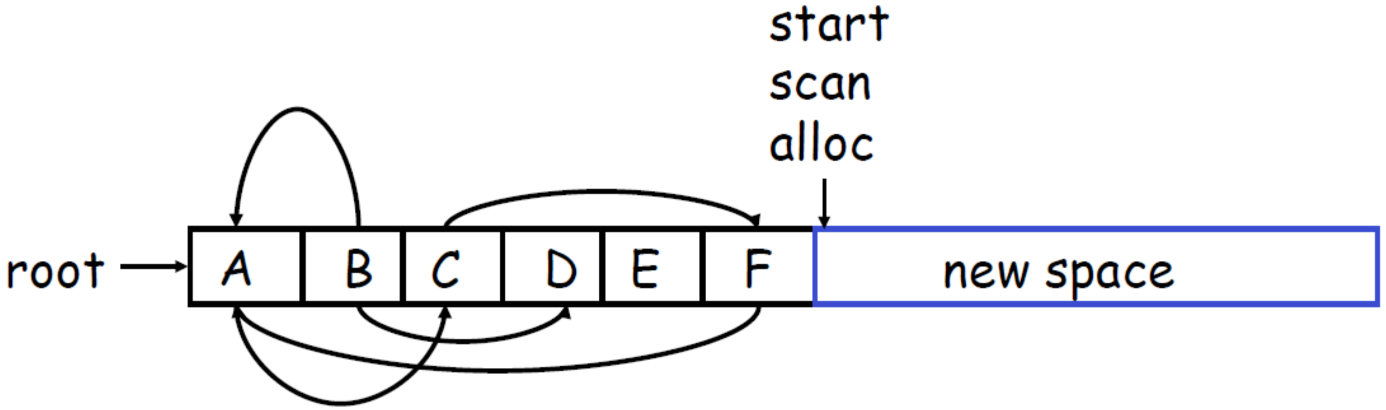
- We need to find all the reachable objects, as for mark and sweep
- As we find a reachable object we copy it into the new space
  - And we have to fix ALL pointers pointing to it!
- As we copy an object we store in the old copy a forwarding pointer to the new copy
  - when we later reach an object with a forwarding pointer we know it was already copied
- 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 and fixed

copied objects whose pointer fields were NOT followed

Example

Before garbage collection



Step 1: Copy the objects pointed by roots and set forwarding pointers (dotted arrow)

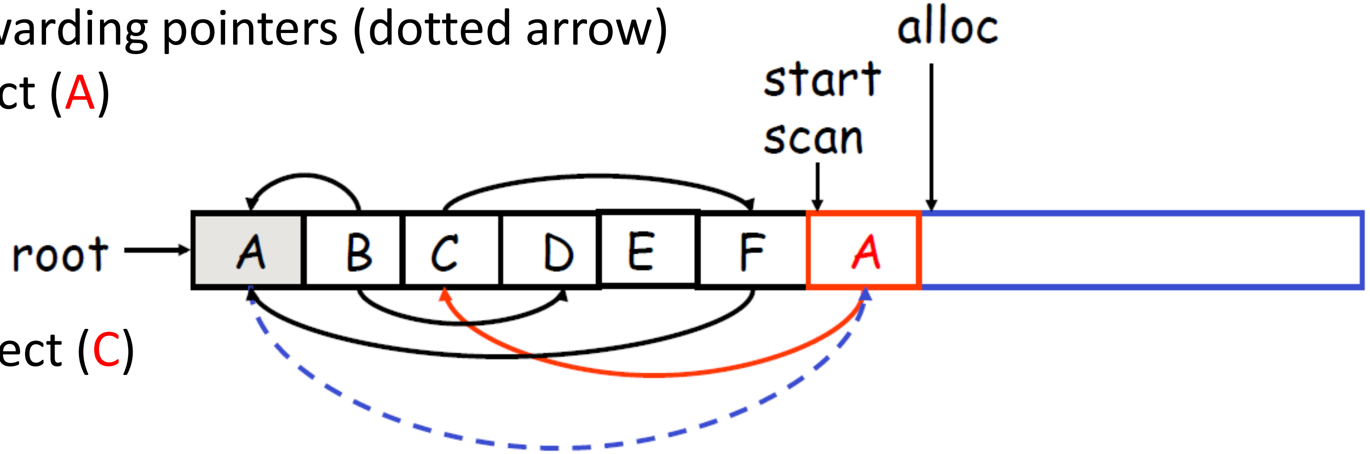
Step 2: Follow the pointer in the next unscanned object (A)

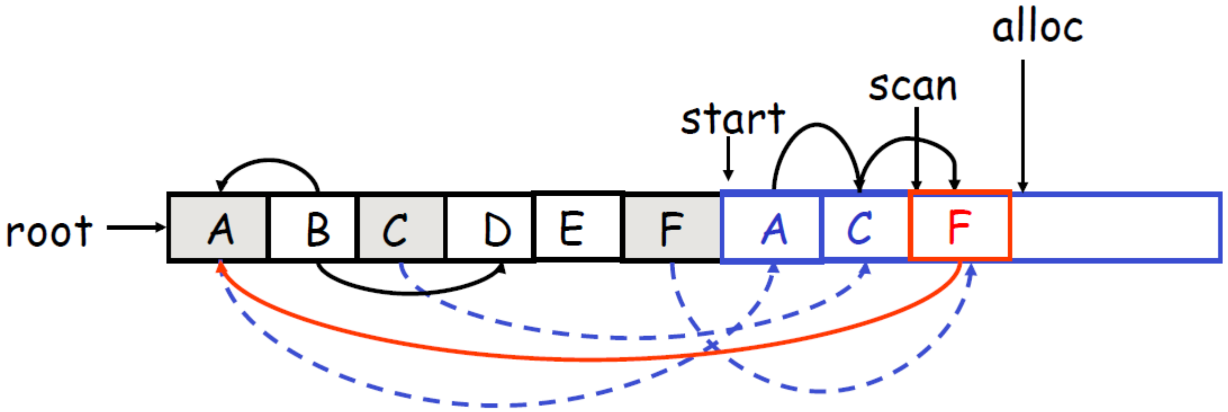
– copy the pointed objects (just C in this case)

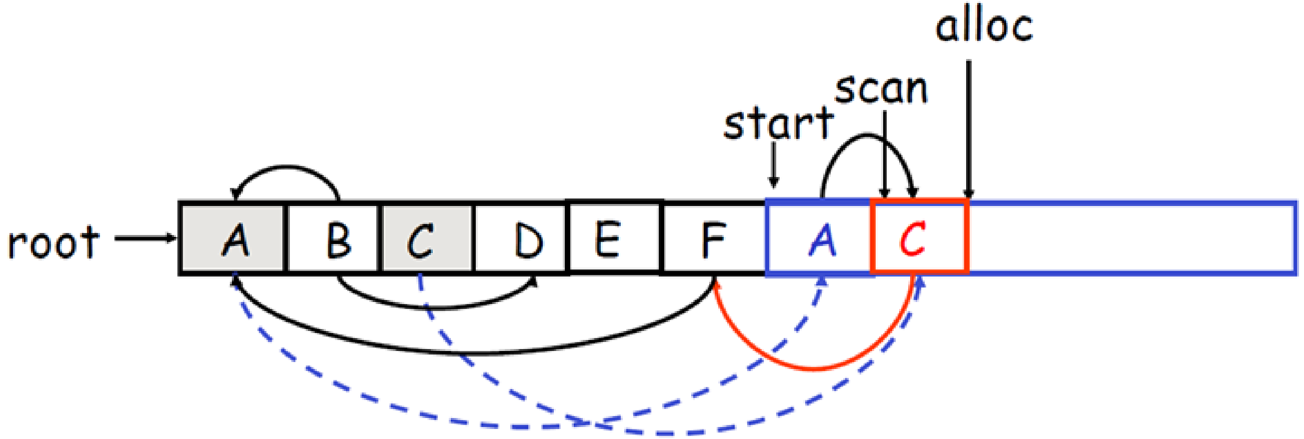
– fix the pointer in A

– set forwarding pointer

Step 3: Follow the pointer in the next unscanned object (C)







Example

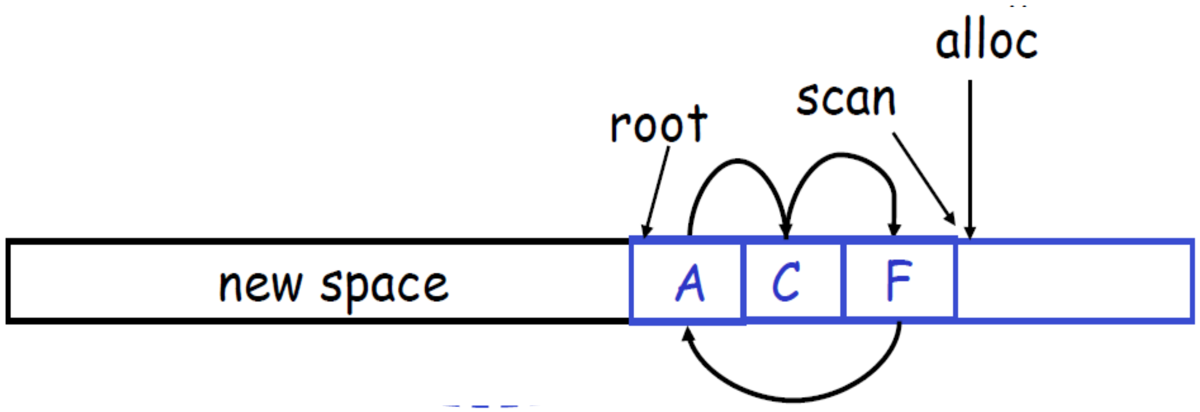
Step 1: Copy the objects pointed by roots and set forwarding pointers (dotted arrow)
Step 2: Follow the pointer in the next unscanned object (A)

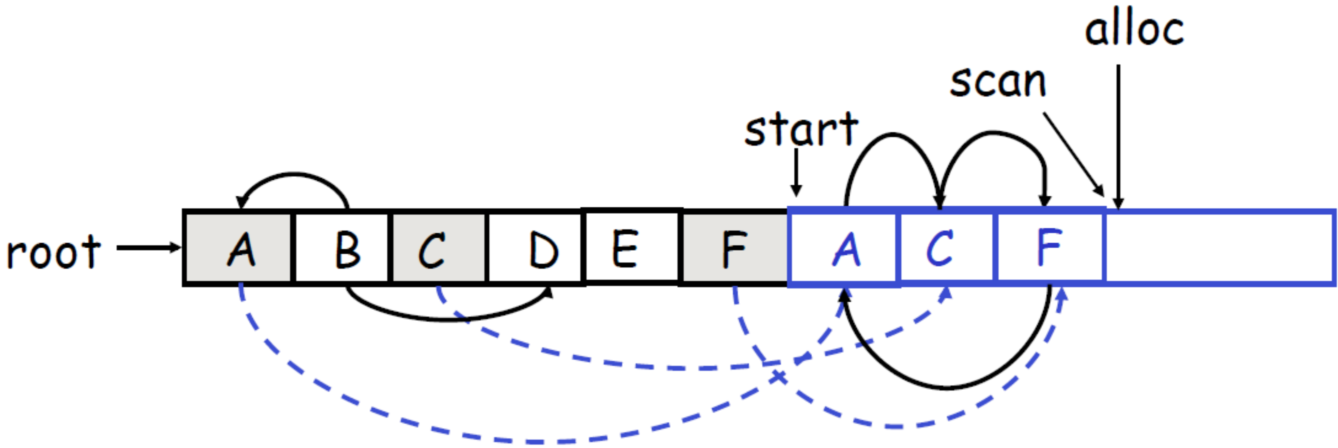
- copy the pointed objects (just C in this case)
- fix the pointer in A
- set forwarding pointer

Step 3: Follow the pointer in the next unscanned object (C)
Step 4: Follow the pointer in the next unscanned object (F)

- the pointed object (A) was already copied. Set the pointer same as the forwarding pointer

Step 5: 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
    O: = * scan
    for each pointer p contained in O do
        O':= *p  // get the object pointed by p
        if O' is without a forwarding pointer
            *alloc := O' // copy O' to new space
            alloc:= alloc + sizeof(O') // update alloc pointer
            set 1st word of old O' to point to the new copy
            *p:= new copy of O'  //p to point to the new copy of O'
        else
            set p in O equal to the forwarding pointer // *p is already copied
        fi
    end for
    scan:= scan + sizeof(O) // scan pointer to the next object
od
```

Details

- As with mark and sweep, we must be able to tell how large is an object 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

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

Advantages:

1. Only touches live data , while mark and sweep touches both live and dead data
2. No fragmentation

Disadvantages:

- Requires (at most) twice the memory space

Stop and Copy for C/C++

- Stop and copy is not suitable for C/C++

Why?

- Garbage collection relies on being able to find all reachable objects
  - it needs to find all pointers in an object and size
- In C or C++ it is impossible to identify the contents of objects in memory
  - E.g., how can you tell that a memory word is a pointer or somebody's account number ?
  - Because of incomplete type information, the use of unsafe casts, etc.
- Conservative Garbage Collection (for C/C++)
  - ✓ Idea: suppose it is a pointer if it looks like one
  - ✓ most pointers are within a certain address range,
  - ✓ they are word aligned, etc.
  - ✓ may retain memory spuriously
- Different styles of conservative collector
- Mostly-copying: can move objects you are sure of

Technique 3: 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 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 then 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, memory leak

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 garbage collection implementation stop the execution during collection
  – not acceptable in real-time applications
- Garbage collection is going to be around for a while
- Researchers are working on advanced garbage collection algorithms:
  – Concurrent: allow the program to run while the collection is happening
  – Incremental: do not scan long-lived objects at every collection
  – Parallel: several collectors working in parallel