Automatic Memory Management

Lecture 17

Lecture Outline

- 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

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

- Until recently they were unpopular outside the Lisp family of languages
  - just like type safety used to be unpopular

The Basic Idea

- When an object that takes memory space is created, unused space is automatically allocated
  - In ChocoPy, new objects are created by X() and other constructs.
- 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 it is impossible to tell
  - We will have to use a heuristic to find many (not all) objects that will never be used again
- Observation: a program can use only the objects that it can find:

  x = X()

  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 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

Reachability is an Approximation

- Consider the program:

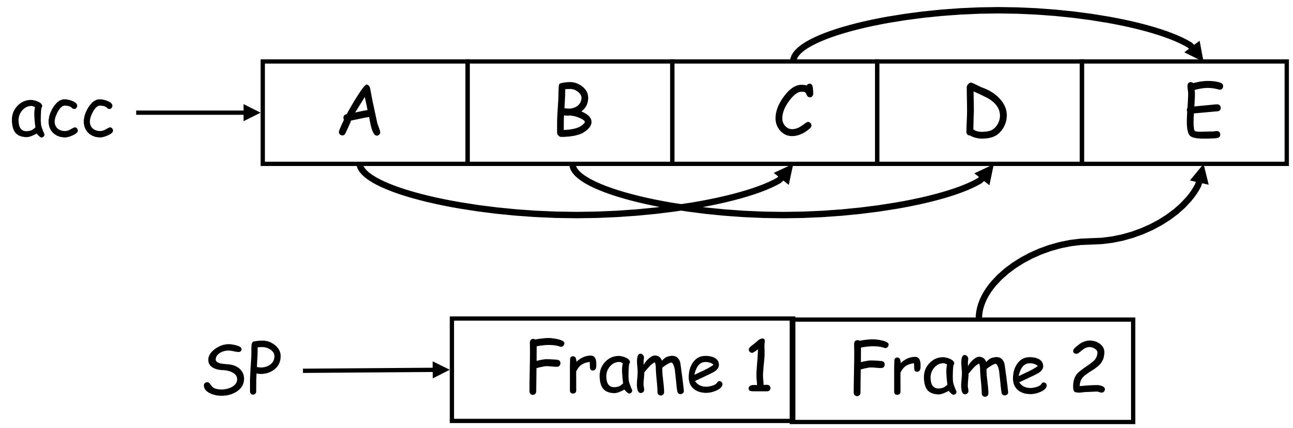
```
x = A()
y = B()
x = y;
if alwaysTrue(): x = A() else: x.foo()
```

- 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 in ChocoPyc

- In ChocoPyc, 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 ChocoPyc we start tracing from acc and stack (and global variables if they are not part of the stack)
  - they are called <u>the roots</u>
- Note that B and D are not reachable from acc or the stack
- Thus we can reuse their storage

Elements of Garbage Collection

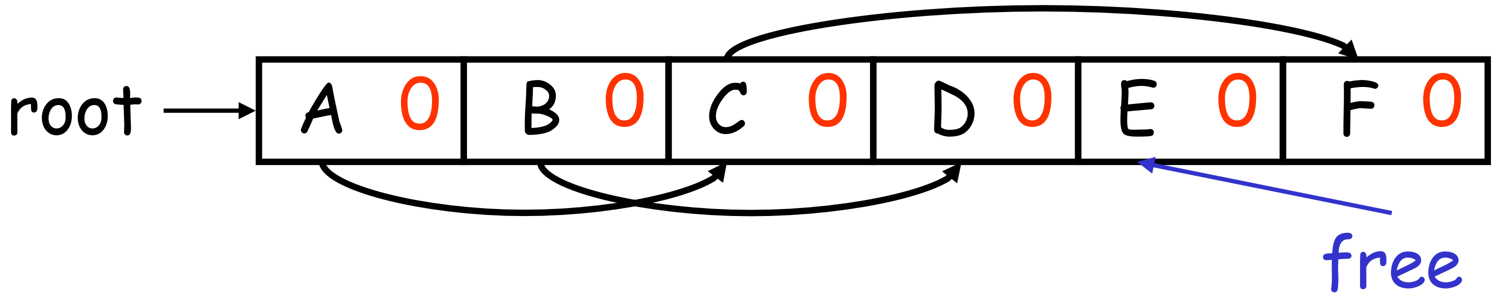
- Every garbage collection scheme has the following steps
  1. Allocate space as needed for new objects
  2. When space runs out:
     a) Compute what objects might be used again (generally by tracing objects reachable from a set of "root" registers)
     b) Free the space used by objects not found in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

First Technique: Mark and Sweep

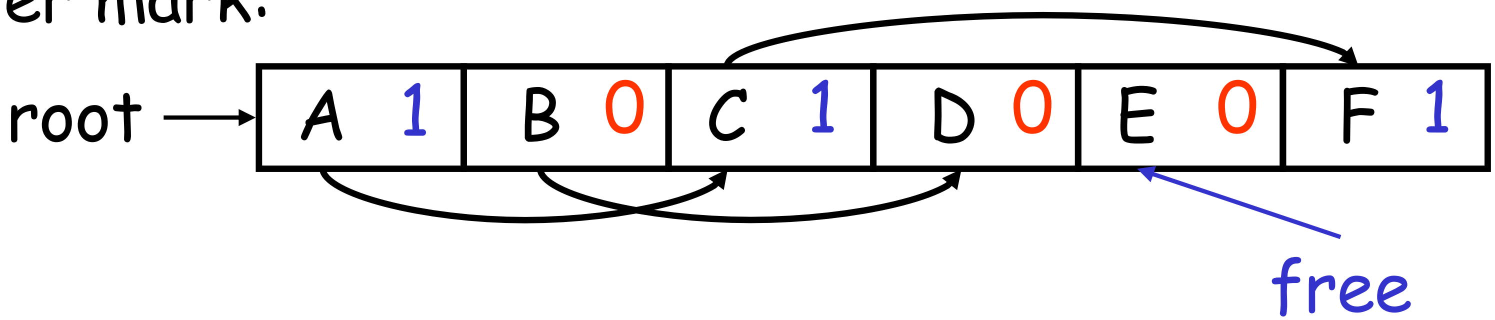
- When memory runs out, GC executes two phases
  - the mark phase: traces reachable objects
  - the sweep phase: collects garbage objects

- 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

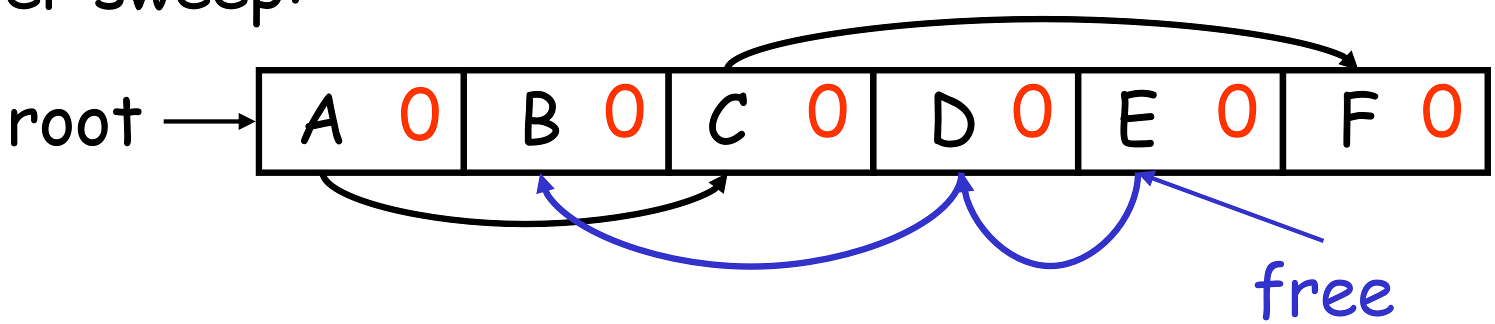
Mark and Sweep Example



After mark:



After sweep:



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 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, 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
  - 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

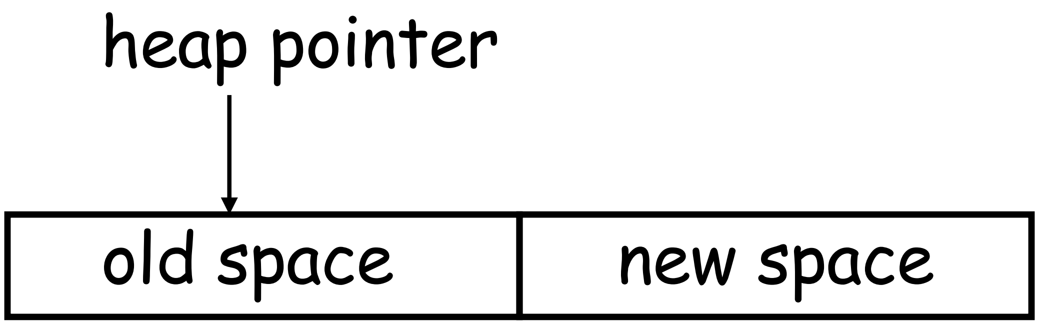
- Similarly, the free list is stored in the free objects themselves

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
- 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



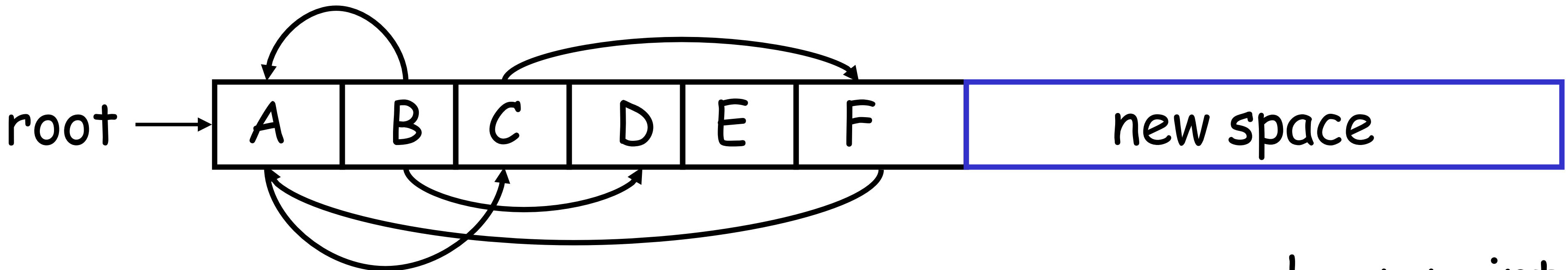
- The heap pointer points to the next free word in the old space
  - Allocation just advances the heap pointer

Stop and Copy Garbage Collection

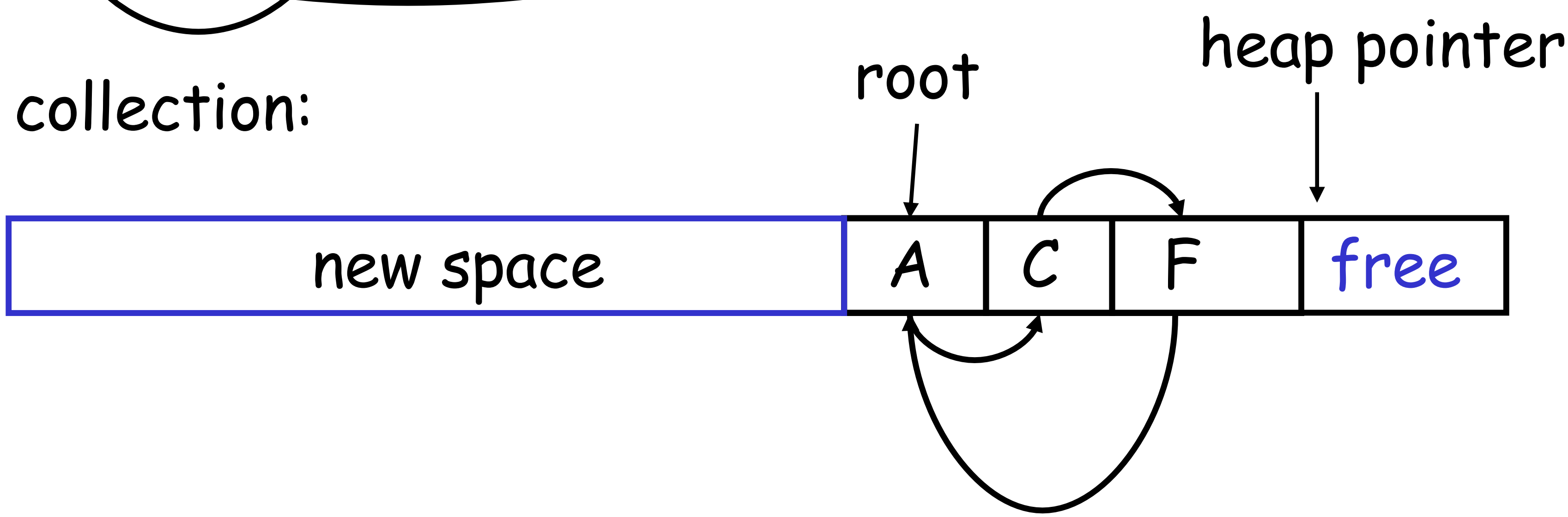
- Starts when the old space is full
- Copies all reachable objects from old space into new space
  - garbage is left behind
  - after the copy phase the new space uses less space than the old one before the collection
- After the copy the roles of the old and new spaces are reversed and the program resumes

Stop and Copy Garbage Collection. Example

Before collection:



After collection:

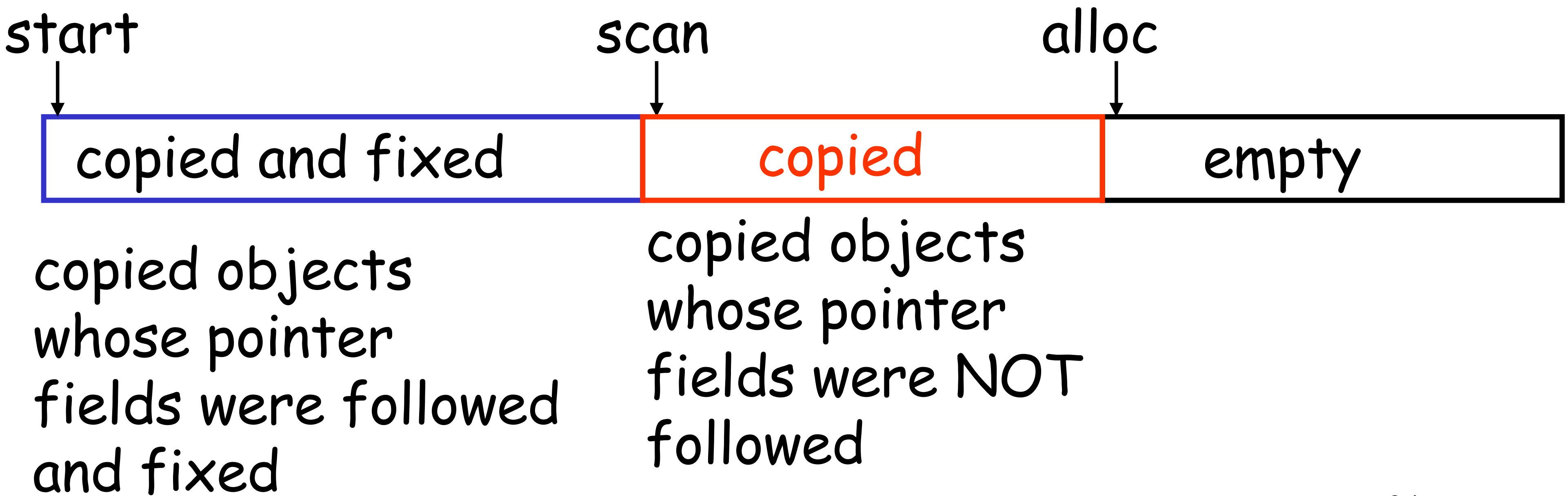


Implementation of Stop and Copy

- 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

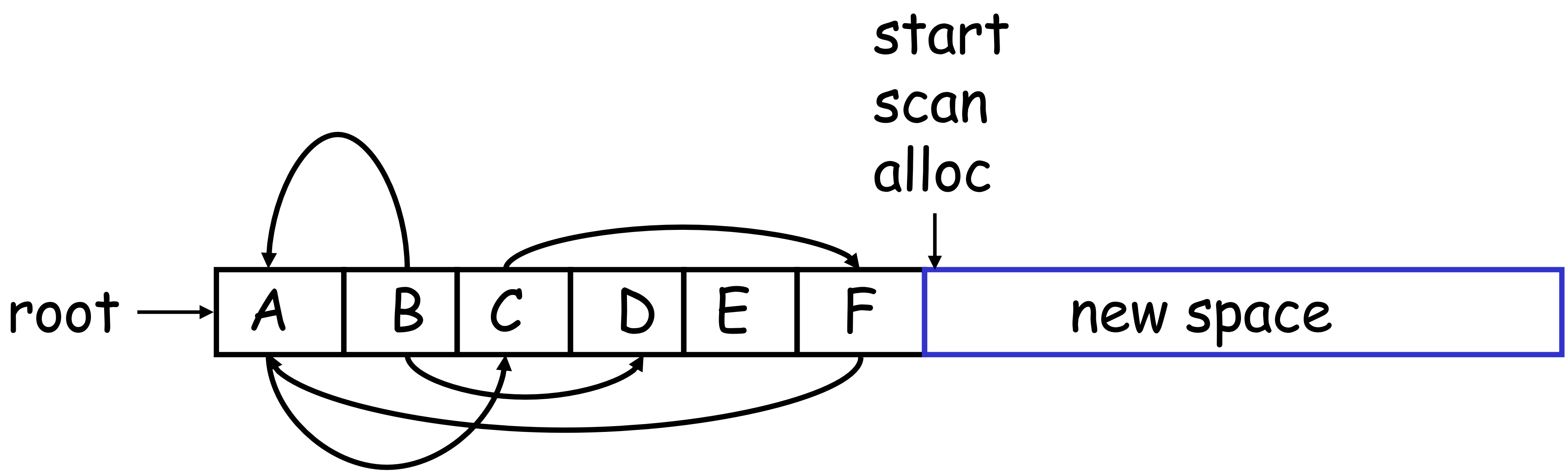
Implementation of Stop and Copy (Cont.)

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



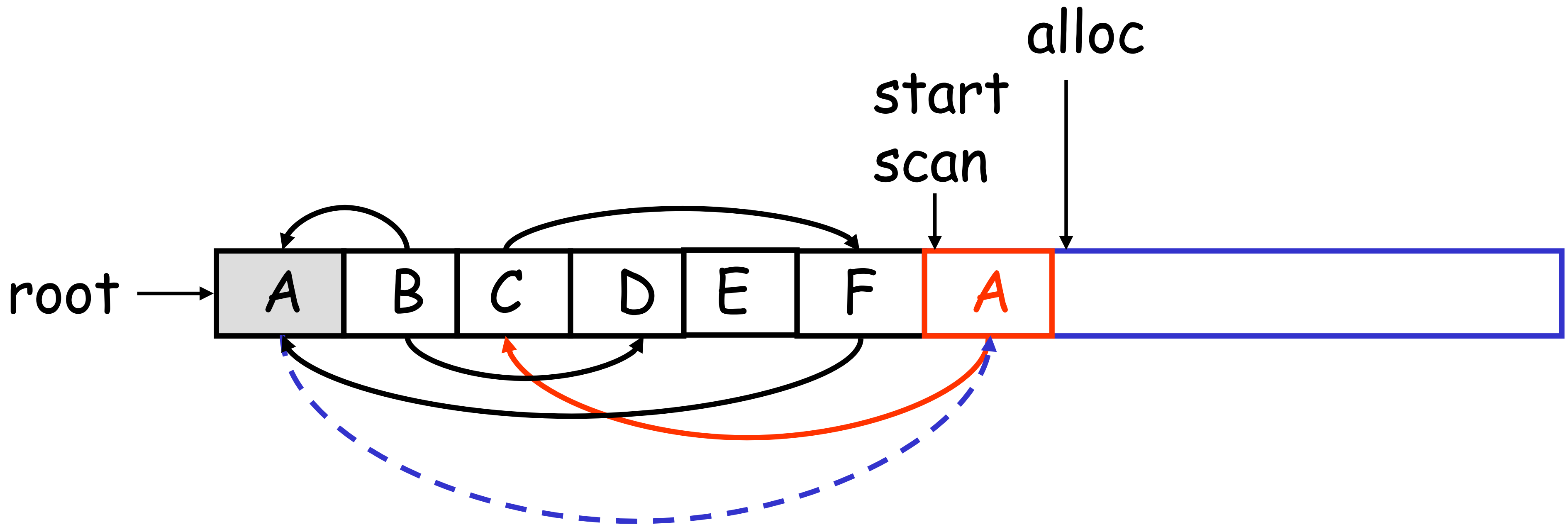
Stop and Copy. Example (1)

- Before garbage collection



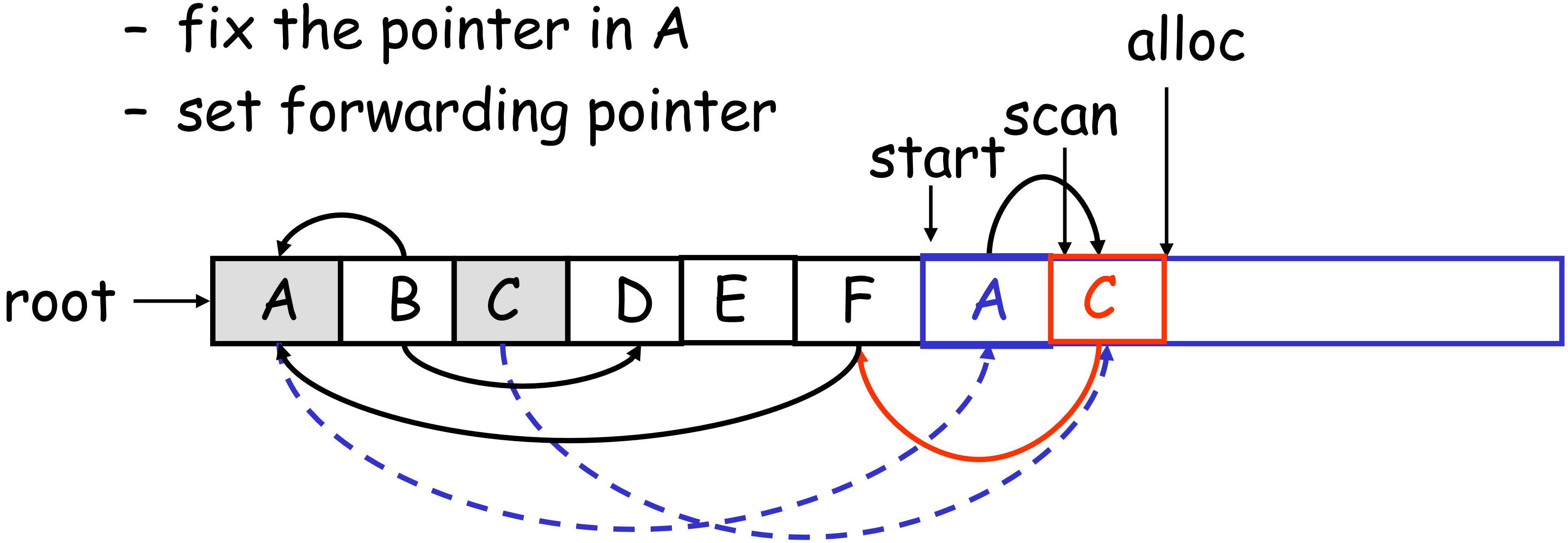
Stop and Copy. Example (3)

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



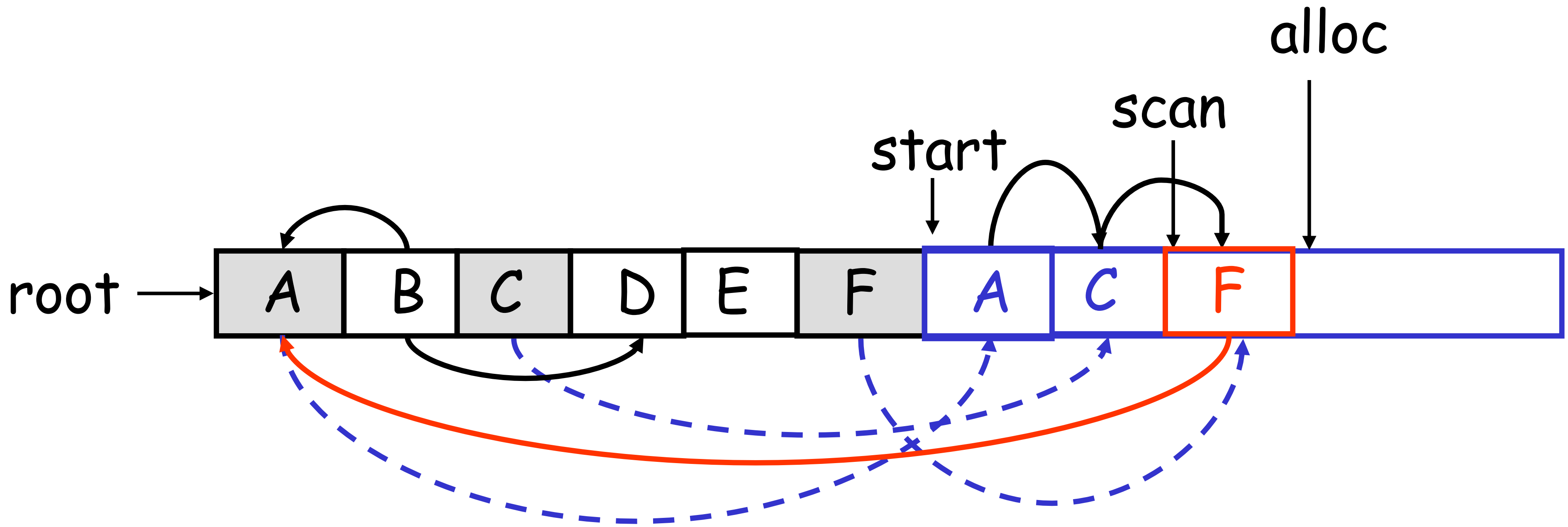
Stop and Copy. Example (3)

- 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



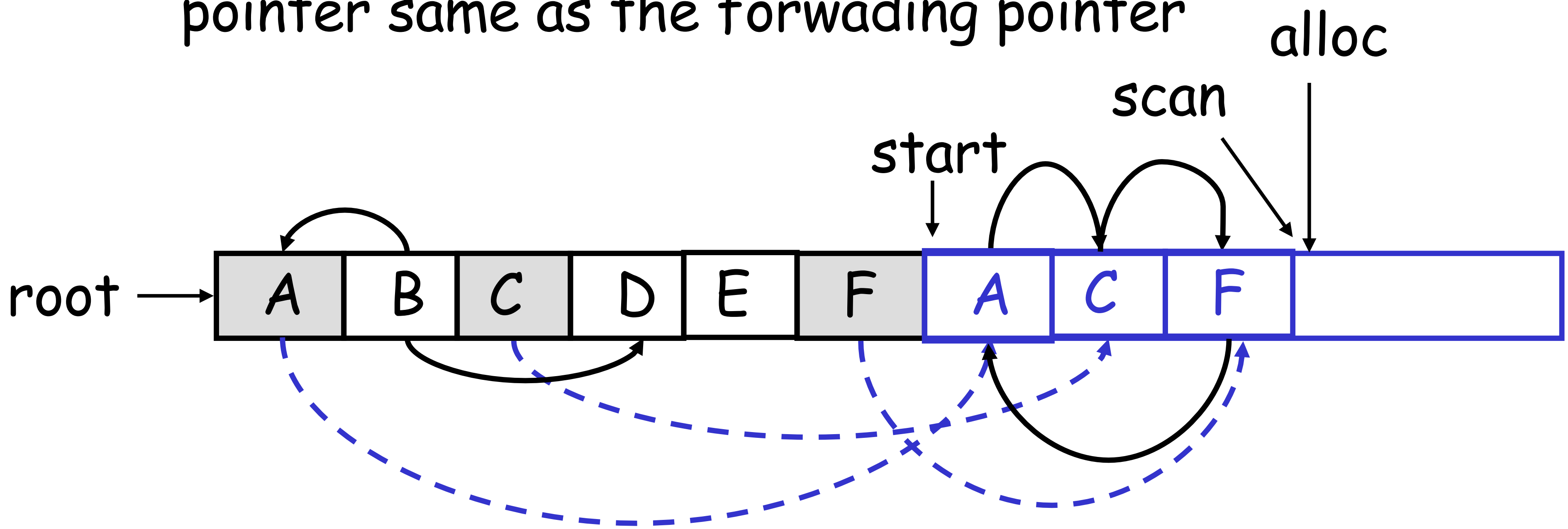
Stop and Copy. Example (4)

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



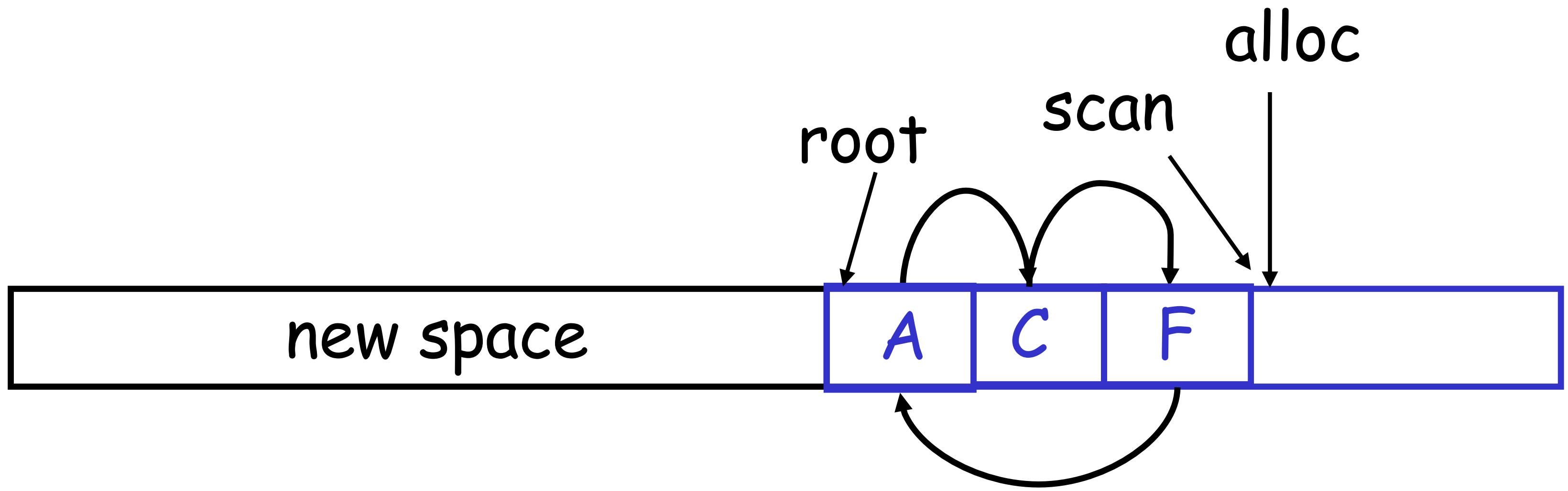
Stop and Copy. Example (5)

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

Stop and Copy. 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

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 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., how can you tell that a memory word is a pointer or somebody's account number ?

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

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

- X() 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

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. Evaluation

- 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
  - Generational: do not scan long-lived objects at every collection
  - Parallel: several collectors working in parallel