Automatic Memory Management

Martin Kellogg

Course Announcements

- As some of you have noticed, the PA4 leaderboard results are subject to **timing variance**
  - In particular, sometimes you “get lucky” and your compiler’s code appears to be much, much faster than the reference
  - There’s not much I can do about this, unfortunately.
  - We will run each final submission several times and take the **median** Q score (not the best), so if you see a big speedup without doing anything, you should assume it is ephemeral
- PA4c1 is due on Monday
- I will not hold office hours next Wednesday

Agenda

- Why Automatic Memory Management?
- Garbage Collection
- Three Techniques
  - Mark and Sweep
  - Stop and Copy
  - Reference Counting

Motivation: Why Automatic Mem. Mgmt.?

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

Motivation: Why Automatic Mem. Mgmt.?

- ***Storage management*** is still a hard problem in modern programming
- Programs in languages like C and C++ have many storage bugs

Motivation: Why Automatic Mem. Mgmt.?

- ***Storage management*** is still a hard problem in modern programming
- Programs in languages like C and C++ have many storage bugs
  - forgetting to free unused memory

Motivation: Why Automatic Mem. Mgmt.?

- ***Storage management*** is still a hard problem in modern programming
- Programs in languages like C and C++ have many storage bugs
  - forgetting to free unused memory
  - dereferencing a dangling pointer

Motivation: Why Automatic Mem. Mgmt.?

- ***Storage management*** is still a hard problem in modern programming
- Programs in languages like C and C++ have many storage bugs
  - forgetting to free unused memory
  - dereferencing a dangling pointer
  - overwriting parts of a data structure by accident and so on... (can be big **security** problems)

Motivation: Why Automatic Mem. Mgmt.?

- ***Storage management*** is still a hard problem in modern programming
- Programs in languages like C and C++ have many storage bugs
  - forgetting to free unused memory
  - dereferencing a dangling pointer
  - overwriting parts of a data structure by accident and so on... (can be big **security** problems)
- Storage bugs are **hard to find**

Motivation: Why Automatic Mem. Mgmt.?

- ***Storage management*** is still a hard problem in modern programming
- Programs in languages like C and C++ have many storage bugs
  - forgetting to free unused memory
  - dereferencing a dangling pointer
  - overwriting parts of a data structure by accident and so on... (can be big **security** problems)
- Storage bugs are **hard to find**
  - a bug can lead to a visible effect far away in time and program text from the source

What about Types?

- **Some** storage bugs can be prevented in a strongly-typed language

What about Types?

- **Some** storage bugs can be prevented in a strongly-typed language
  - e.g., most type systems guarantee no random access into some other object's private data

What about Types?

- **Some** storage bugs can be prevented in a strongly-typed language
  - e.g., most type systems guarantee no random access into some other object's private data
- Can **types** prevent errors in programs with manual allocation and deallocation of memory?

What about Types?

- **Some** storage bugs can be prevented in a strongly-typed language
  - e.g., most type systems guarantee no random access into some other object's private data
- 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

What about Types?

- **Some** storage bugs can be prevented in a strongly-typed language
  - e.g., most type systems guarantee no random access into some other object's private data
- 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
  - So, **in theory**, yes

What about Types?

- **Some** storage bugs can be prevented in a strongly-typed language
  - e.g., most type systems guarantee no random access into some other object's private data
- 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
  - So, **in theory**, yes
- If you want type safety **in practice** then you must use ***automatic memory management***

Automatic Memory Management

Automatic Memory Management

- This is an old problem
  - studied since the 1950s for Lisp

Automatic Memory Management

- This is an old problem
  - studied since the 1950s for Lisp
- There are several well-known techniques for **completely automatic** memory management

Automatic Memory Management

- This is an old problem
  - studied since the 1950s for Lisp
- There are several well-known techniques for **completely automatic** memory management
  - These are ***dynamic analyses***

Automatic Memory Management

- This is an old problem
  - studied since the 1950s for Lisp
- There are several well-known techniques for **completely automatic** memory management
  - These are ***dynamic analyses***
    - That is, they involve **instrumenting** the program so that its behavior at run time is different

Automatic Memory Management

- This is an old problem
  - studied since the 1950s for Lisp
- There are several well-known techniques for **completely automatic** memory management
  - These are ***dynamic analyses***
    - That is, they involve **instrumenting** the program so that its behavior at run time is different

You are probably familiar with some other dynamic analysis techniques like:

- testing
- code coverage

Automatic Memory Management

- This is an old problem
  - studied since the 1950s for Lisp
- There are several well-known techniques for **completely automatic** memory management
  - These are ***dynamic analyses***
    - That is, they involve **instrumenting** the program so that its behavior at run time is different
    - In particular, we want it to automatically free memory :)

Automatic Memory Management

- This is an old problem
  - studied since the 1950s for Lisp
- There are several well-known techniques for **completely automatic** memory management
  - These are ***dynamic analyses***
    - That is, they involve **instrumenting** the program so that its behavior at run time is different
    - In particular, we want it to automatically free memory :)
- Until relatively recently (Java), these techniques were not popular outside the Lisp family of languages
  - Just like static type safety used to be unpopular…

Automatic Memory Management: Basic Idea

Automatic Memory Management: Basic Idea

- When an object that takes memory space is created, unused space is automatically **allocated**
  - In Cool, new objects are created by `new X`

Automatic Memory Management: Basic Idea

- When an object that takes memory space 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

Automatic Memory Management: Basic Idea

- When an object that takes memory space 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 (= **dead** objects?)

Automatic Memory Management: Basic Idea

- When an object that takes memory space 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 (= **dead** objects?)
- This space can be **freed** to be reused later

Dead Again?

- How can we tell whether an object will “**never be used again**”?

Dead Again?

- How can we tell whether an object will “**never be used again**”?
  - In general it is impossible (**undecidable**) to tell (cf. liveness)

Dead Again?

- How can we tell whether an object will “**never be used again**”?
  - In general it is impossible (**undecidable**) to tell (cf. liveness)
  - We will have to use a **heuristic** to find many (not all) objects that will never be used again

Dead Again?

- How can we tell whether an object will "**never be used again**"?
  - In general it is impossible (**undecidable**) to tell (cf. liveness)
  - 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**.

Dead Again?

- How can we tell whether an object will "**never be used again**"?
  - In general it is impossible (**undecidable**) to tell (cf. liveness)
  - 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**.
- For example:

```
let x : A <- new A in { x <- y; ... }
```

Dead Again?

- How can we tell whether an object will "**never be used again**"?
  - In general it is impossible (**undecidable**) to tell (cf. liveness)
  - 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**.
- For example:

```
let x : A <- new A in { x <- y; ... }
```

- After `x <- y` there is **no way** to access the newly allocated object

Garbage

- **Definition:** An object *x* is ***reachable*** if and only if:

Garbage

- **Definition:** An object *x* is ***reachable*** if and only if:
  - A local variable (or register) contains a pointer to *x*, **or**

Garbage

- **Definition:** An object *x* is ***reachable*** if and only if:
  - A local variable (or register) contains a pointer to *x*, **or**
  - Another reachable object *y* contains a pointer to *x*

Garbage

- **Definition:** An object *x* is ***reachable*** if and only if:
  - A local variable (or register) contains a pointer to *x*, **or**
  - Another reachable object *y* contains a pointer to *x*
    - (Note that `self` is a local variable in Cool)

Garbage

- **Definition:** An object *x* is ***reachable*** if and only if:
  - A local variable (or register) contains a pointer to *x*, **or**
  - Another reachable object *y* contains a pointer to *x*
    - (Note that `self` is a local variable in Cool)
- You can find all reachable objects by starting from local variables and following all the pointers ("**transitive**")

Garbage

- **Definition:** An object *x* is ***reachable*** if and only if:
  - A local variable (or register) contains a pointer to *x*, **or**
  - Another reachable object *y* contains a pointer to *x*
    - (Note that `self` is a local variable in Cool)
- You can find all reachable objects by starting from local variables and following all the pointers ("**transitive**")
- An unreachable object can never be referred to by the program

Garbage

- **Definition:** An object *x* is ***reachable*** if and only if:
  - A local variable (or register) contains a pointer to *x*, **or**
  - Another reachable object *y* contains a pointer to *x*
    - (Note that `self` is a local variable in Cool)
- You can find all reachable objects by starting from local variables and following all the pointers ("**transitive**")
- An unreachable object can never be referred to by the program
  - Such objects are called ***garbage***

Reachability is an Approximation

Reachability is an Approximation

- Consider the program:

```
x <- new Ant;
y <- new Bat;
x <- y;
if alwaysTrue() then x <- new Cow else x.eat() fi
```

Reachability is an Approximation

- Consider the program:

```
x <- new Ant;
y <- new Bat;
x <- y;
if alwaysTrue() then x <- new Cow else x.eat() fi
```

- After `x <- y` (assuming `y` becomes dead there):

Reachability is an Approximation

- Consider the program:

```
x <- new Ant;
y <- new Bat;
x <- y;
if alwaysTrue() then x <- new Cow else x.eat() fi
```

- After `x <- y` (assuming `y` becomes dead there):
  - The `Ant` object is not reachable anymore

Reachability is an Approximation

- Consider the program:

```
x <- new Ant;
y <- new Bat;
x <- y;
if alwaysTrue() then x <- new Cow else x.eat() fi
```

- After `x <- y` (assuming `y` becomes dead there):
  - The `Ant` object is not reachable anymore
  - The `Bat` object is reachable (through `x`)

Reachability is an Approximation

- Consider the program:

```
x <- new Ant;
y <- new Bat;
x <- y;
if alwaysTrue() then x <- new Cow else x.eat() fi
```

- After `x <- y` (assuming `y` becomes dead there):
  - The `Ant` object is not reachable anymore
  - The `Bat` object is reachable (through `x`)
  - Thus the `Bat` is not garbage and is ***not*** collected

Reachability is an Approximation

- Consider the program:

```
x <- new Ant;
y <- new Bat;
x <- y;
if alwaysTrue() then x <- new Cow else x.eat() fi
```

- After `x <- y` (assuming `y` becomes dead there):
  - The `Ant` object is not reachable anymore
  - The `Bat` object is reachable (through `x`)
  - Thus the `Bat` is not garbage and is ***not*** collected
  - But the `Bat` object is never going to be used

Cool Garbage

Cool Garbage

- Recall that at run-time, a Cool interpreter has two mappings:

Cool Garbage

Operational Semantics!

- Recall that at run-time, a Cool interpreter has two mappings:
  - Environment **E** maps variable identifiers to locations
  - Store **S** maps locations to values

Cool Garbage

- Recall that at run-time, a Cool interpreter has two mappings:
  - Environment **E** maps variable identifiers to locations
  - Store **S** maps locations to values
- Proposed Cool garbage collector algorithm:

Cool Garbage

- Recall that at run-time, a Cool interpreter has two mappings:
  - Environment **E** maps variable identifiers to locations
  - Store **S** maps locations to values
- Proposed Cool garbage collector algorithm:
  - for each location l ∈ domain(**S**)

Cool Garbage

- Recall that at run-time, a Cool interpreter has two mappings:
  - Environment **E** maps variable identifiers to locations
  - Store **S** maps locations to values
- Proposed Cool garbage collector algorithm:
  - for each location l ∈ domain(**S**)
    - let can_reach = false

Cool Garbage

- Recall that at run-time, a Cool interpreter has two mappings:
  - Environment **E** maps variable identifiers to locations
  - Store **S** maps locations to values
- Proposed Cool garbage collector algorithm:
  - for each location l ∈ domain(**S**)
    - let can_reach = false
    - for each $(v, l_2) \in$ **E**

Cool Garbage

- Recall that at run-time, a Cool interpreter has two mappings:
  - Environment **E** maps variable identifiers to locations
  - Store **S** maps locations to values
- Proposed Cool garbage collector algorithm:
  - for each location l ∈ domain(**S**)
    - let can_reach = false
    - for each $(v, l_2) \in$ **E**
      - if $l = l_2$ then can_reach = true

Cool Garbage

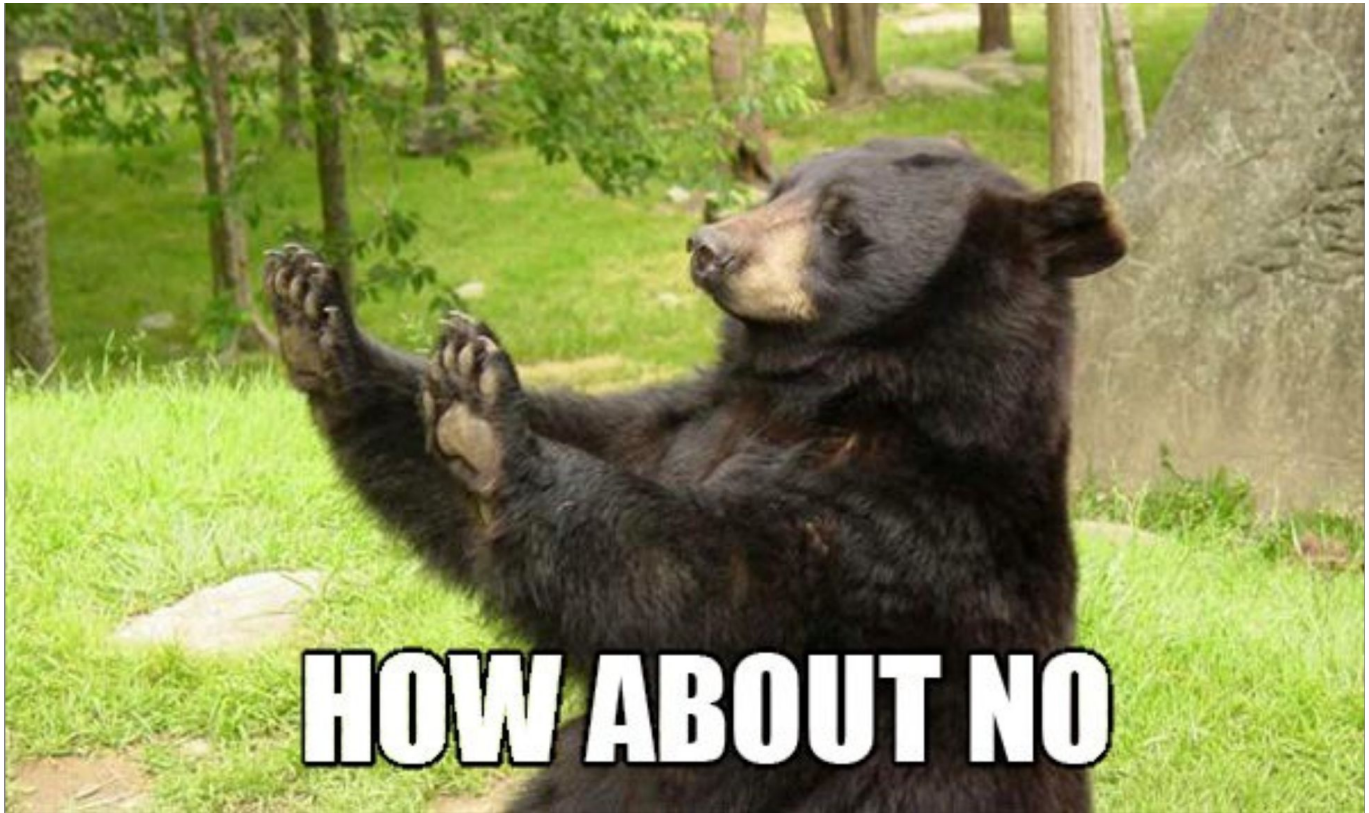
- Recall that at run-time, a Cool interpreter has two mappings:
  - Environment **E** maps variable identifiers to locations
  - Store **S** maps locations to values
- Proposed Cool garbage collector algorithm:
  - for each location l ∈ domain(**S**)
    - let can_reach = false
    - for each $(v, l_2) \in$ **E**
      - if $l = l_2$ then can_reach = true
    - if not can_reach then reclaim_location(l)

Cool Garbage

- Recall that at run-time, a Cool interpreter has two mappings:
  - Environment **E** maps variable identifiers to locations
  - Store **S** maps locations to values
- Proposed Cool garbage collector algorithm:
  - for each location l ∈ domain(**S**)
    - let can_reach = false
    - for each $(v, l_2) \in$ **E**
      - if $l = l_2$ then can_reach = true
    - if not can_reach then reclaim_location(l)

Does this work?

Does That Work?



Cooler Garbage

- Recall that at run-time, a Cool interpreter has two mappings:
  - Environment **E** maps variable identifiers to locations
  - Store **S** maps locations to values
- Proposed Cool garbage collector algorithm:
  - for each location l ∈ domain(**S**)
    - let can_reach = false
    - for each $(v, l_2) \in$ **E**
      - if $l = l_2$ then can_reach = true

Cooler Garbage

- Recall that at run-time, a Cool interpreter has two mappings:
  - Environment **E** maps variable identifiers to locations
  - Store **S** maps locations to values
- Proposed Cool garbage collector algorithm:
  - for each location $l \in$ domain(**S**)
    - let can_reach = false
    - for each $(v, l_2) \in$ **E**
      - if $l = l_2$ then can_reach = true
      - for each $l_3 \in v$

Cooler Garbage

- Recall that at run-time, a Cool interpreter has two mappings:
  - Environment **E** maps variable identifiers to locations
  - Store **S** maps locations to values
- Proposed Cool garbage collector algorithm:
  - for each location l ∈ domain(**S**)
    - let can_reach = false
    - for each $(v, l_2) \in$ **E**
      - if $l = l_2$ then can_reach = true
      - for each $l_3 \in v$ *// v is X(..., $a_i = l_i$, ...)*

Cooler Garbage

- Recall that at run-time, a Cool interpreter has two mappings:
  - Environment **E** maps variable identifiers to locations
  - Store **S** maps locations to values
- Proposed Cool garbage collector algorithm:
  - for each location l ∈ domain(**S**)
    - let can_reach = false
    - for each $(v, l_2) \in$ **E**
      - if $l = l_2$ then can_reach = true
      - for each $l_3 \in v$ *// v is X(..., $a_i = l_i$, ...)*
        - if $l = l_3$ then can_reach = true

Cooler Garbage

- Recall that at run-time, a Cool interpreter has two mappings:
  - Environment **E** maps variable identifiers to locations
  - Store **S** maps locations to values
- Proposed Cool garbage collector algorithm:
  - for each location l ∈ domain(**S**)
    - let can_reach = false
    - for each $(v, l_2) \in$ **E**
      - if $l = l_2$ then can_reach = true
      - for each $l_3 \in v$ *// v is X(..., $a_i = l_i$, ...)*
        - if $l = l_3$ then can_reach = true
    - if not can_reach then reclaim_location(l)

Garbage Analysis

- Could we use this proposed Cooler Garbage Collector **in real life**?

Garbage Analysis

- Could we use this proposed Cooler Garbage Collector **in real life**?
  - How long would it take?
  - How much space would it take?
  - Are we forgetting anything?

Garbage Analysis

- Could we use this proposed Cooler Garbage Collector **in real life**?
  - How long would it take?
  - How much space would it take?
  - Are we forgetting anything?
    - Hint: Yes. It's **still wrong**

Tracing Reachable Values

Tracing Reachable Values

- In Cool, **local variables** are easy to find:

Tracing Reachable Values

- In Cool, **local variables** are easy to find:
  - Use the environment mapping **E**

Tracing Reachable Values

- In Cool, **local variables** are easy to find:
  - Use the environment mapping **E**
    - and one object may point to other objects, etc.

Tracing Reachable Values

- In Cool, **local variables** are easy to find:
  - Use the environment mapping **E**
    - and one object may point to other objects, etc.
- The **stack** is more complex:

Tracing Reachable Values

- In Cool, **local variables** are easy to find:
  - Use the environment mapping **E**
    - and one object may point to other objects, etc.
- The **stack** is more complex:
  - each stack frame (activation record) contains:

Tracing Reachable Values

- In Cool, **local variables** are easy to find:
  - Use the environment mapping **E**
    - and one object may point to other objects, etc.
- The **stack** is more complex:
  - each stack frame (activation record) contains:
    - method parameters! (which are other objects…)

Tracing Reachable Values

- In Cool, **local variables** are easy to find:
  - Use the environment mapping **E**
    - and one object may point to other objects, etc.
- The **stack** is more complex:
  - each stack frame (activation record) contains:
    - method parameters! (which are other objects…)
- If we know the layout of a stack frame we can find the pointers (objects) in it

Tracing Reachable Values

- In Cool, **local variables** are easy to find:
  - Use the environment mapping **E**
    - and one object may point to other objects, etc.
- The **stack** is more co
  - each stack frame
    - method para…)
- If we know the layou… pointers (objects) in it

Reachability can be **tricky**!

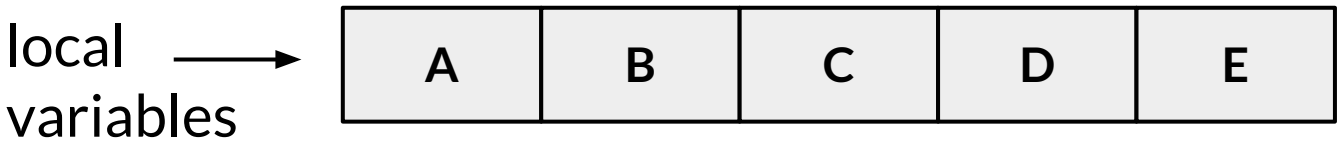
- Many things may look legitimate and reachable but will turn out not to be
- How can we figure this out systematically?

A Simple Example

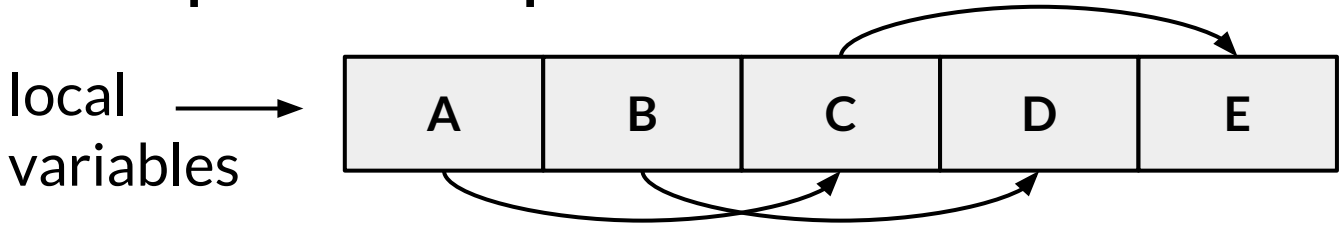
A Simple Example

local variables 

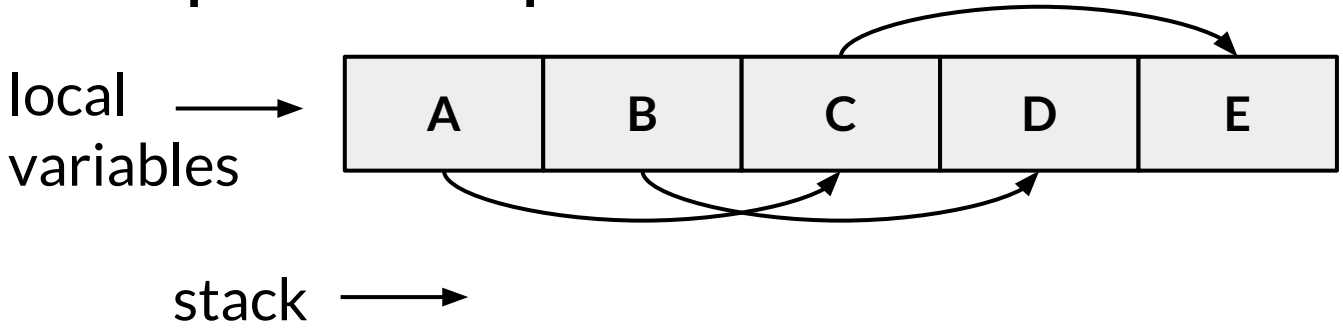
A Simple Example



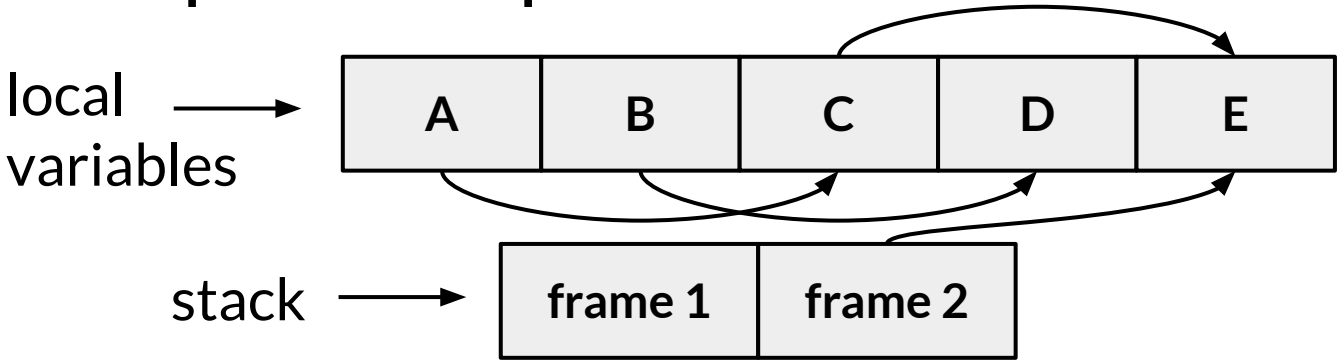
A Simple Example



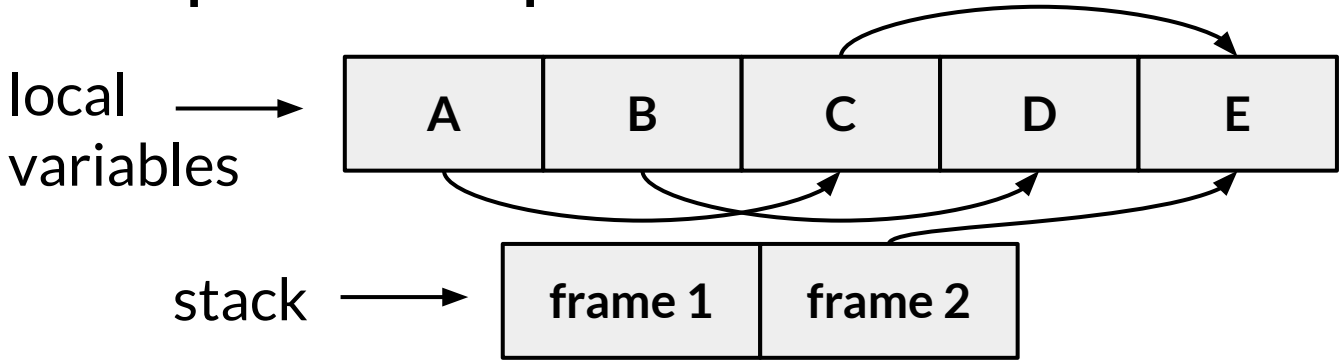
A Simple Example



A Simple Example

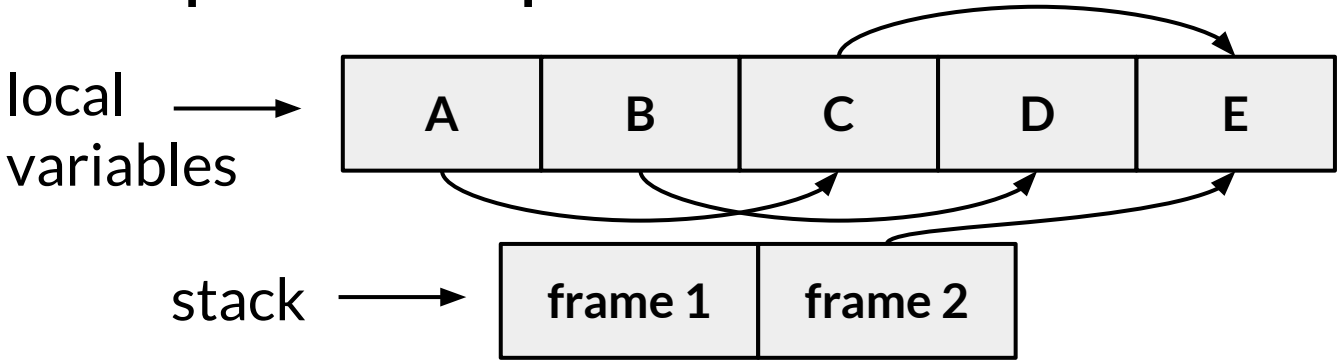


A Simple Example



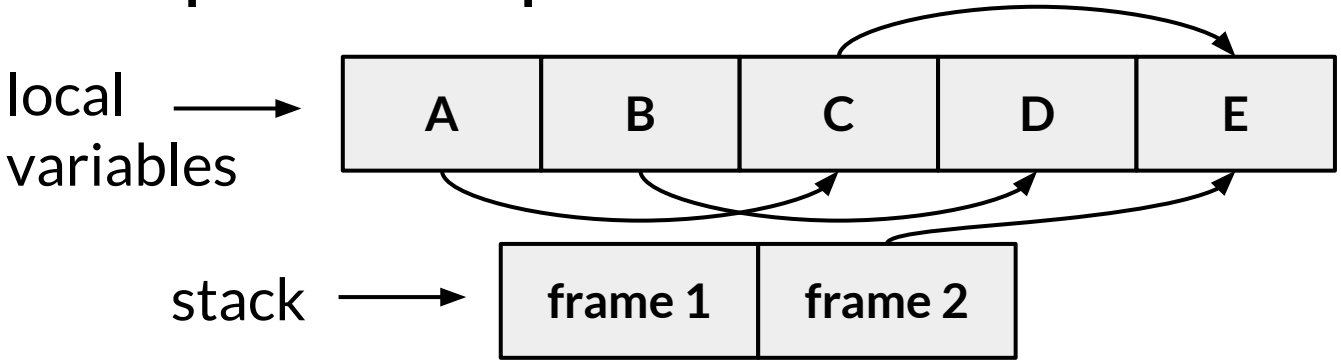
- Start tracing from local vars and the stack

A Simple Example



- Start tracing from local vars and the stack
  - They are called the ***roots***

A Simple Example



- Start tracing from local vars and the stack
  - They are called the ***roots***
- Note that B and D are not reachable from other local vars or the stack
  - Thus we can reuse their storage **when they go out of scope**

Elements of Garbage Collection

- Every garbage collection scheme has the following steps

Elements of Garbage Collection

- Every garbage collection scheme has the following steps
  - **Allocate** space as needed for new objects

Elements of Garbage Collection

- Every garbage collection scheme has the following steps
  - **Allocate** space as needed for new objects
  - When space runs out:

Elements of Garbage Collection

- Every garbage collection scheme has the following steps
  - **Allocate** space as needed for new objects
  - When space runs out:
    - Compute what objects might be used again

Elements of Garbage Collection

- Every garbage collection scheme has the following steps
  - **Allocate** space as needed for new objects
  - When space runs out:
    - Compute what objects might be used again
      - generally by **tracing** objects reachable from a set of roots

Elements of Garbage Collection

- Every garbage collection scheme has the following steps
  - **Allocate** space as needed for new objects
  - When space runs out:
    - Compute what objects might be used again
      - generally by **tracing** objects reachable from a set of roots
    - Free space used by objects not found in the previous step

Elements of Garbage Collection

- Every garbage collection scheme has the following steps
  - **Allocate** space as needed for new objects
  - When space runs out:
    - Compute what objects might be used again
      - generally by **tracing** objects reachable from a set of roots
    - Free space used by objects not found in the previous step
- Some strategies perform garbage collection **before** the space actually runs out

Elements of Garbage Collection

- Every garbage collection scheme has the following steps
  - **Allocate** space as needed for new objects
  - When space runs out:
    - Compute what objects might be used again
      - generally by **tracing** objects reachable from a set of roots
    - Free space used by objects not found in the previous step
- Some strategies perform garbage collection **before** the space actually runs out
  - Why might this be useful?

Elements of Garbage Collection

- Every garbage collection scheme has the following steps
  - **Allocate** space as needed for new objects
  - When space runs out:
    - Compute what objects might be used again
      - generally by **tracing** objects reachable from a set of roots
    - Free space used by objects not found in the previous step
- Some strategies perform garbage collection **before** the space actually runs out
  - Why might this be useful?
    - Hint: will we need any space to run our garbage collector?

Elements of Garbage Collection

- Every garbage collection scheme has the following steps
  - **Allocate** space as needed for new objects
  - When space runs out:
    - Compute wh
      - generall
        roots
    - Free space u
- Some strategies per
  actually runs out
  - Why might this be useful?
    - Hint: will we need any space to run our garbage collector?

m a set of

revious step

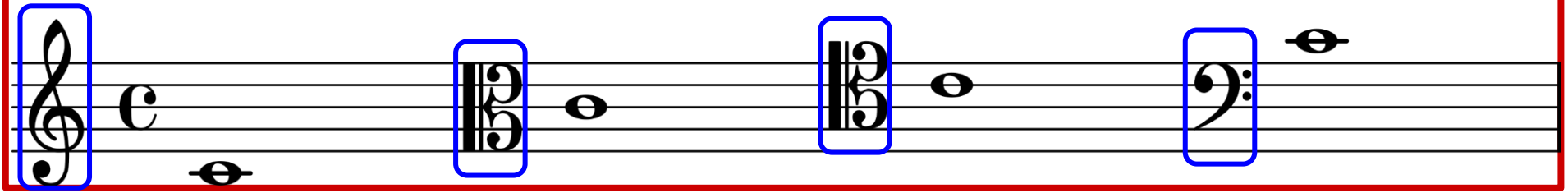
ne space

After trivia, we will see three specific garbage collection algorithms:

- mark and sweep
- stop and copy
- reference counting

Trivia Break: Music Theory

This musical symbol (examples highlighted in **blue** below) indicates which notes are represented by the lines and spaces on a musical staff. Its position on a staff assigns a particular pitch to one of the five lines or four spaces, which defines the pitches on the remaining lines and spaces. The modern symbols derive from the medieval practice of annotating the reference line of a staff with the name of the note it was intended to bear; over time the shapes of these letters became stylised, leading to their current versions.



Trivia Break: Programming Languages

This general-purpose high-level programming language supports multiple paradigms. Its features include automatic memory management, a strong type system, and good support for internationalization and portability. Its creators originally released it in 2000 as a closed-source language, aligning with their business goals at the time. However, in the decades since, the company responsible for the language has changed its attitude towards open-source, and in 2014 the open-source Roslyn compiler for this language was released; it has been the primary compiler for the language since. The language is famously used in game development (e.g., it is the default scripting language in the Unity game engine).

Mark and Sweep

- Our first garbage collection algorithm

Mark and Sweep

- Our first garbage collection algorithm
- When memory runs out, GC executes two phases:

Mark and Sweep

- Our first garbage collection algorithm
- When memory runs out, GC executes two phases:
  - the ***mark*** phase: traces reachable objects

Mark and Sweep

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

Mark and Sweep

- Our first garbage collection algorithm
- 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***

Mark and Sweep

- Our first garbage collection algorithm
- 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

Mark and Sweep

- Our first garbage collection algorithm
- 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

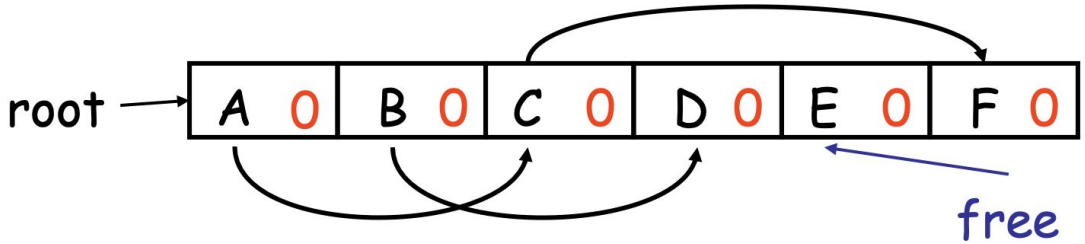
- Our first garbage collection algorithm
- When memory runs out, GC executes two phases:
  - the ***mark*** phase: traces reachable objects
  - the ***sweep*** phase: collects garbage objects
- Every object has an extra bit: the ***mark bit***
  - reserved for memory management
  - initially the mark bit is 0
  - set to 1 for the **reachable** objects in the mark phase
- In the sweep phase, **free** all objects whose mark bit is still 0

Mark and Sweep

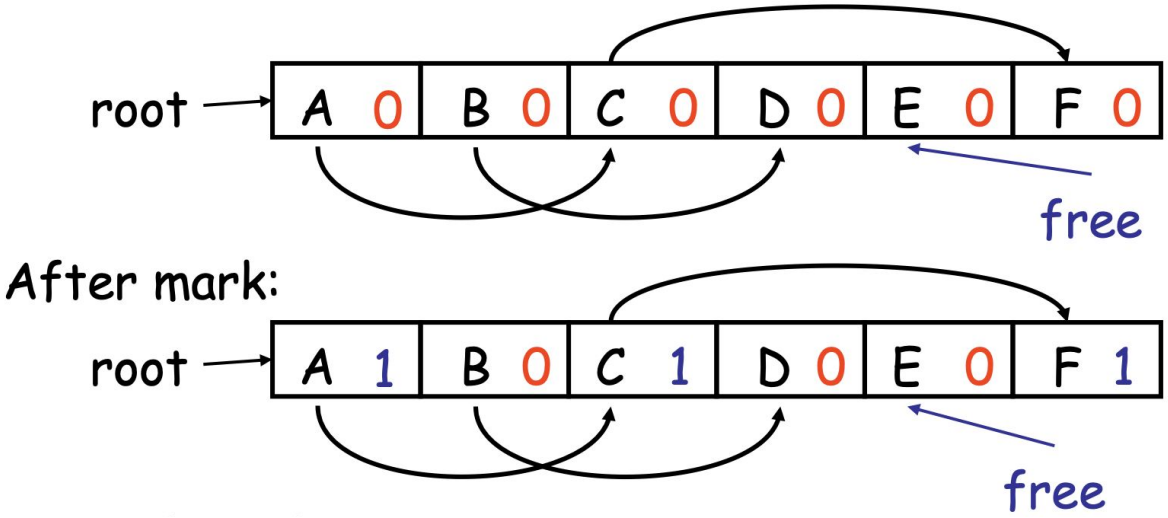
- Our first garbage collection algorithm
- When memory runs out, GC executes two phases:
  - the ***mark*** phase: traces reachable objects
  - the ***sweep*** phase: collects garbage objects
- Every object has an extra bit: the ***mark bit***
  - reserved for memory management
  - initially the mark bit is 0
  - set to 1 for the **reachable** objects in the mark phase
- In the sweep phase, **free** all objects whose mark bit is still 0
  - creating a ***free list*** of garbage that can be reused

Mark and Sweep: Example

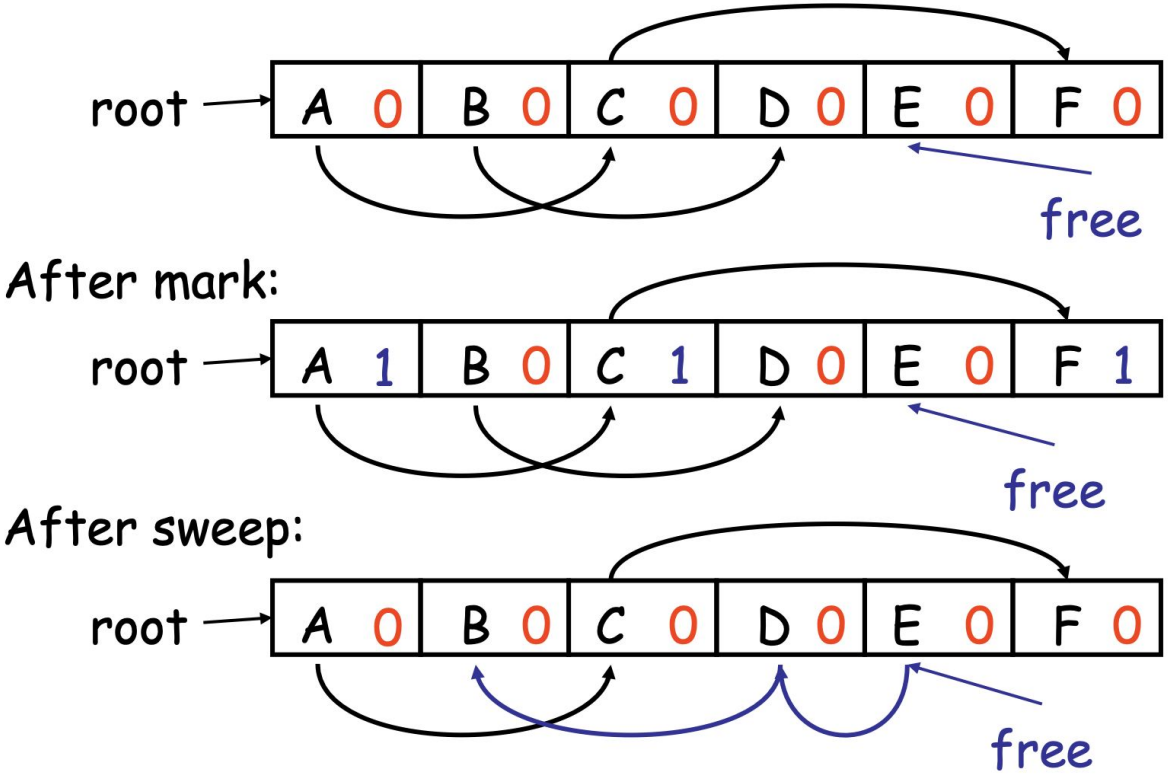
Mark and Sweep: Example



Mark and Sweep: Example



Mark and Sweep: Example



The Mark Phase: Algorithm

The Mark Phase: Algorithm

```
let todo = { all roots }          (* worklist *)
```

The Mark Phase: Algorithm

```
let todo = { all roots }          (* worklist *)
while todo is non-empty ; do
```

The Mark Phase: Algorithm

```
let todo = { all roots }          (* worklist *)
while todo is non-empty ; do
    pick v ∈ todo
    todo <- todo - { v }
```

The Mark Phase: Algorithm

```
let todo = { all roots }          (* worklist *)
while todo is non-empty ; do
    pick v ∈ todo
    todo <- todo - { v }
    if mark(v) = 0 then           (* v is unmarked so far *)
```

The Mark Phase: Algorithm

```
let todo = { all roots }          (* worklist *)
while todo is non-empty ; do
    pick v ∈ todo
    todo <- todo - { v }
    if mark(v) = 0 then           (* v is unmarked so far *)
        mark(v) <- 1
```

The Mark Phase: Algorithm

```
let todo = { all roots }          (* worklist *)
while todo is non-empty ; do
    pick v ∈ todo
    todo <- todo - { v }
    if mark(v) = 0 then           (* v is unmarked so far *)
        mark(v) <- 1
        let v_1, ..., v_n be the pointers contained in v
```

The Mark Phase: Algorithm

```
let todo = { all roots }          (* worklist *)
while todo is non-empty ; do
    pick v ∈ todo
    todo <- todo - { v }
    if mark(v) = 0 then           (* v is unmarked so far *)
        mark(v) <- 1
        let v_1, ..., v_n be the pointers contained in v
        todo <- todo U {v_1, ..., v_n}
```

The Mark Phase: Algorithm

```
let todo = { all roots }          (* worklist *)
while todo is non-empty ; do
    pick v ∈ todo
    todo <- todo - { v }
    if mark(v) = 0 then           (* v is unmarked so far *)
        mark(v) <- 1
        let v_1, ..., v_n be the pointers contained in v
        todo <- todo U {v_1, ..., v_n}
    fi
od
```

The Sweep Phase

The Sweep Phase

- The sweep phase **scans the entire heap** looking for objects with mark bit 0

The Sweep Phase

- The sweep phase **scans the entire heap** looking for objects with mark bit 0
  - these objects have not been visited in the mark phase

The Sweep Phase

- The sweep phase **scans the entire heap** looking for objects with mark bit 0
  - these objects have not been visited in the mark phase
  - and so they are garbage

The Sweep Phase

- The sweep phase **scans the entire heap** looking for objects with mark bit 0
  - these objects have not been visited in the mark phase
  - and so they are garbage
- Any such object is added to the **free list**

The Sweep Phase

- The sweep phase **scans the entire heap** looking for objects with mark bit 0
  - these objects have not been visited in the mark phase
  - and so they are garbage
- Any such object is added to the **free list**
- The objects with a mark bit of 1 have their mark bit reset to 0

The Sweep Phase: Algorithm

The Sweep Phase: Algorithm

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

The Sweep Phase: Algorithm

```
/* sizeof(p) is size of block starting at p */
p <- bottom of heap
```

The Sweep Phase: Algorithm

```
/* sizeof(p) is size of block starting at p */
p <- bottom of heap
while p < top of heap do
```

The Sweep Phase: Algorithm

```
/* sizeof(p) is size of block starting at p */
p <- bottom of heap
while p < top of heap do
        if mark(p) = 1 then
                mark(p) <- 0
```

The Sweep Phase: Algorithm

```
/* sizeof(p) is 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
```

The Sweep Phase: Algorithm

```
/* sizeof(p) is 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: Analysis

Mark and Sweep: Analysis

- While conceptually simple, this algorithm has a number of **tricky details**

Mark and Sweep: Analysis

- While conceptually simple, this algorithm has a number of **tricky details**
  - this is typical of GC algorithms

Mark and Sweep: Analysis

- While conceptually simple, this algorithm has a number of **tricky details**
  - this is typical of GC algorithms
- There is a **serious problem** with the mark phase:

Mark and Sweep: Analysis

- While conceptually simple, this algorithm has a number of **tricky details**
  - this is typical of GC algorithms
- There is a **serious problem** with the mark phase:
  - it is invoked when we are out of space

Mark and Sweep: Analysis

- While conceptually simple, this algorithm has a number of **tricky details**
  - this is typical of GC algorithms
- There is 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**

Mark and Sweep: Analysis

- While conceptually simple, this algorithm has a number of **tricky details**
  - this is typical of GC algorithms
- There is 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

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

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

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

Mark and Sweep: Evaluation

- Space for each new object is allocated from the free list

Mark and Sweep: Evaluation

- Space for each new object is allocated from the free list
  - a block large enough is picked

Mark and Sweep: Evaluation

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

Mark and Sweep: Evaluation

- Space for each new object is allocated from the free list
  - a block large enough is picked
  - an area of the necessary size is allocated from it
    - the leftover is put back in the free list
- Disadvantage: mark and sweep can **fragment** memory
  - why is this a problem?

Mark and Sweep: Evaluation

- Space for each new object is allocated from the free list
  - a block large enough is picked
  - an area of the necessary size is allocated from it
    - the leftover is put back in the free list
- Disadvantage: mark and sweep can **fragment** memory
  - why is this a problem?
- Advantage: objects are **not moved** during GC
  - no need to update the pointers to objects
  - works for languages like C and C++ where it's difficult to distinguish pointers from data (more on this later)

Another Technique: Stop and Copy

Another Technique: Stop and Copy

- Memory is organized into two areas:

Another Technique: Stop and Copy

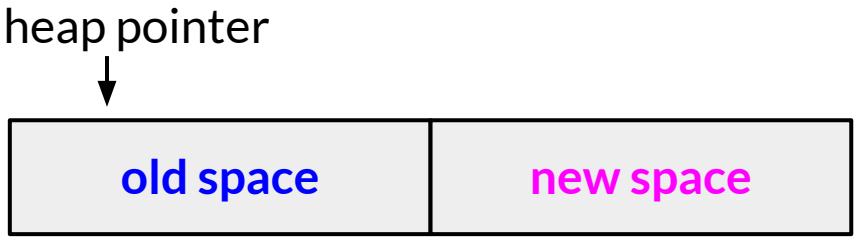
- Memory is organized into two areas:
  - ***Old space***: used for allocation

Another Technique: Stop and Copy

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

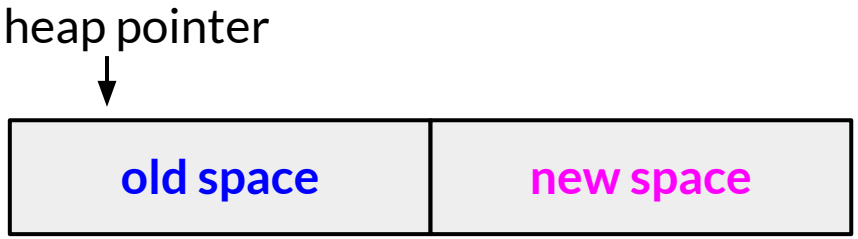
Another Technique: Stop and Copy

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



Another Technique: Stop and Copy

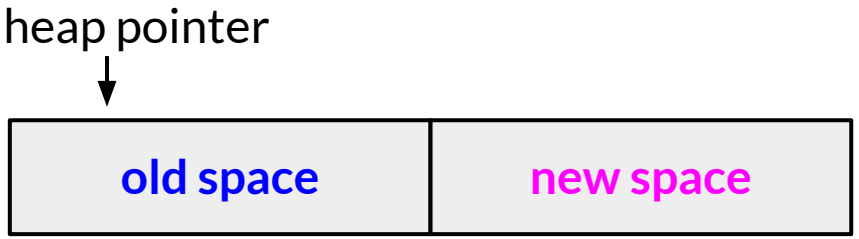
- Memory is organized into two areas:
  - ***Old space***: used for allocation
  - ***New space***: used as a reserve for the GC



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

Another Technique: Stop and Copy

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



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

Stop and Copy: Intuition

Stop and Copy: Intuition

- Starts when the old space is full

Stop and Copy: Intuition

- Starts when the old space is full
- Copies all reachable objects from old space into new space

Stop and Copy: Intuition

- Starts when the old space is full
- Copies all reachable objects from old space into new space
  - **garbage is left behind**

Stop and Copy: Intuition

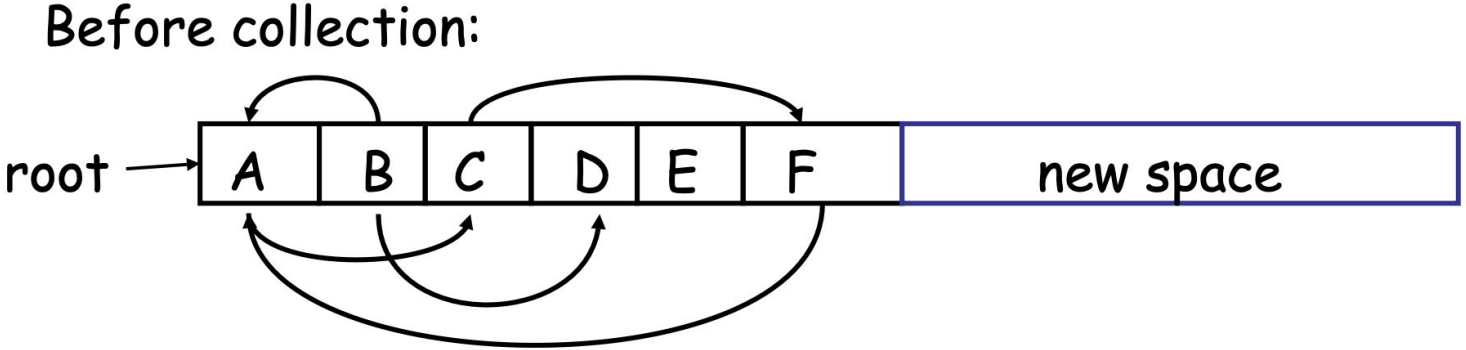
- Starts when the old space is full
- Copies all reachable objects from old space into new space
  - **garbage is left behind**
  - after the copy phase the new space uses less space than the old one before the collection

Stop and Copy: Intuition

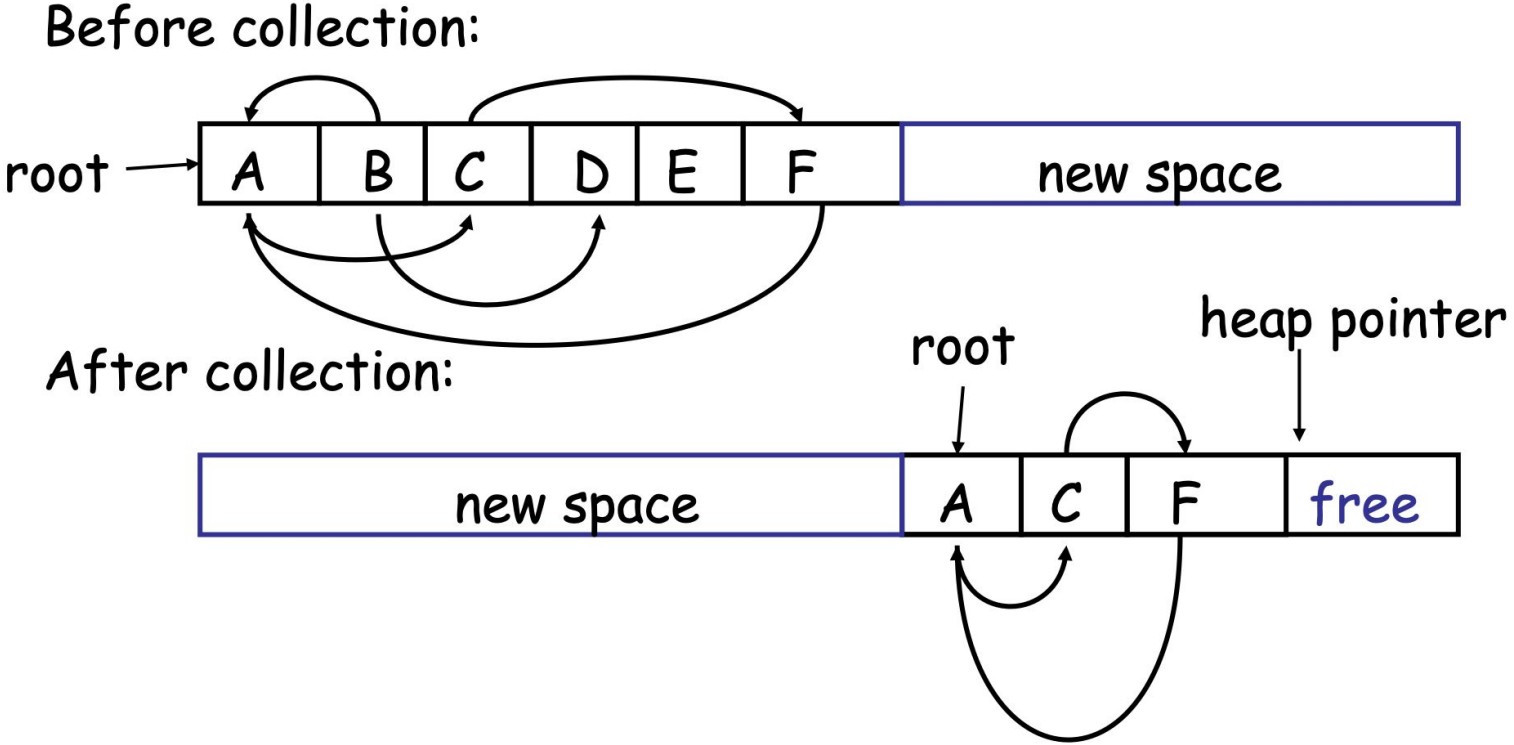
- Starts when the old space is full
- Copies all reachable objects from old space into new space
  - **garbage is left behind**
  - after the copy phase the new space uses less space than the old one before the collection
- After the copy the roles of the old and new spaces are **reversed** and the program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

Stop and Copy: Simple Example

Stop and Copy: Simple Example



Stop and Copy: Simple Example



Stop and Copy: Implementation

Stop and Copy: Implementation

- We need to find all the reachable objects
  - Just as in mark and sweep

Stop and Copy: Implementation

- We need to find all the reachable objects
  - Just as in mark and sweep
- As we find a reachable object we copy it into the new space

Stop and Copy: Implementation

- We need to find all the reachable objects
  - Just as in 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***!

Stop and Copy: Implementation

- We need to find all the reachable objects
  - Just as in 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

Stop and Copy: Implementation

- We need to find all the reachable objects
  - Just as in 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

Stop and Copy: Implementation

- We need to find all the reachable objects
  - Just as in 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
  - How can we identify forwarding pointers?

Stop and Copy: Implementation

- We need to find all the reachable objects
  - Just as in 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
  - How can we identify forwarding pointers?
- We also still have the issue of how to implement the traversal without using extra space

Stop and Copy: Implementation Trick

- Both problems can be solved via the following trick:

Stop and Copy: Implementation Trick

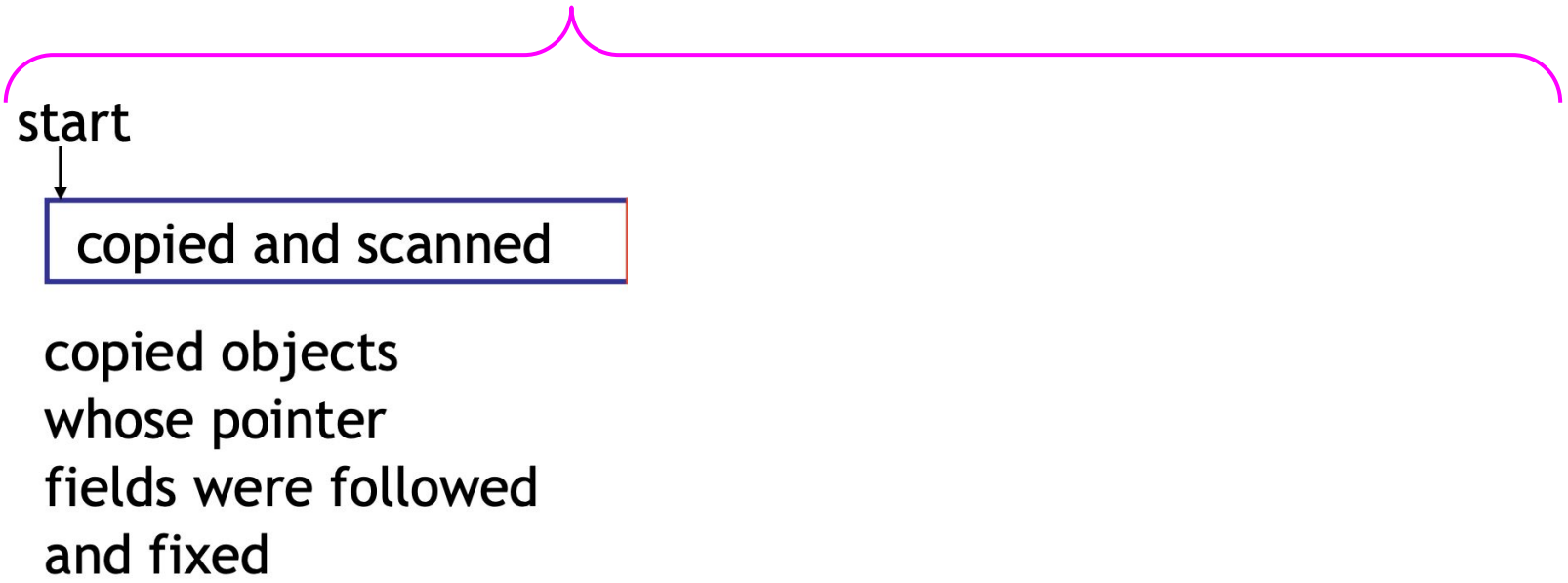
- Both problems can be solved via the following trick:
  - partition the new space into **three** contiguous regions:

Stop and Copy: Implementation Trick

- Both problems can be solved via the following trick:
  - partition the **new space** into **three** contiguous regions:

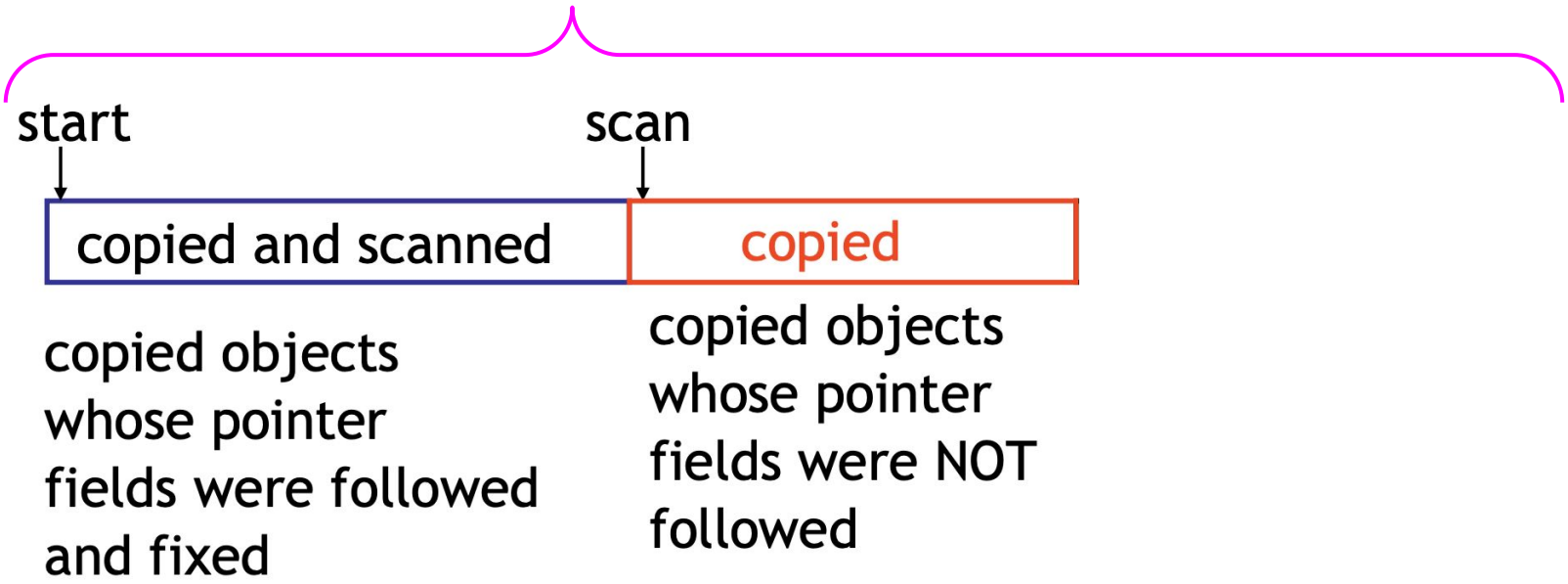
Stop and Copy: Implementation Trick

- Both problems can be solved via the following trick:
  - partition the **new space** into **three** contiguous regions:



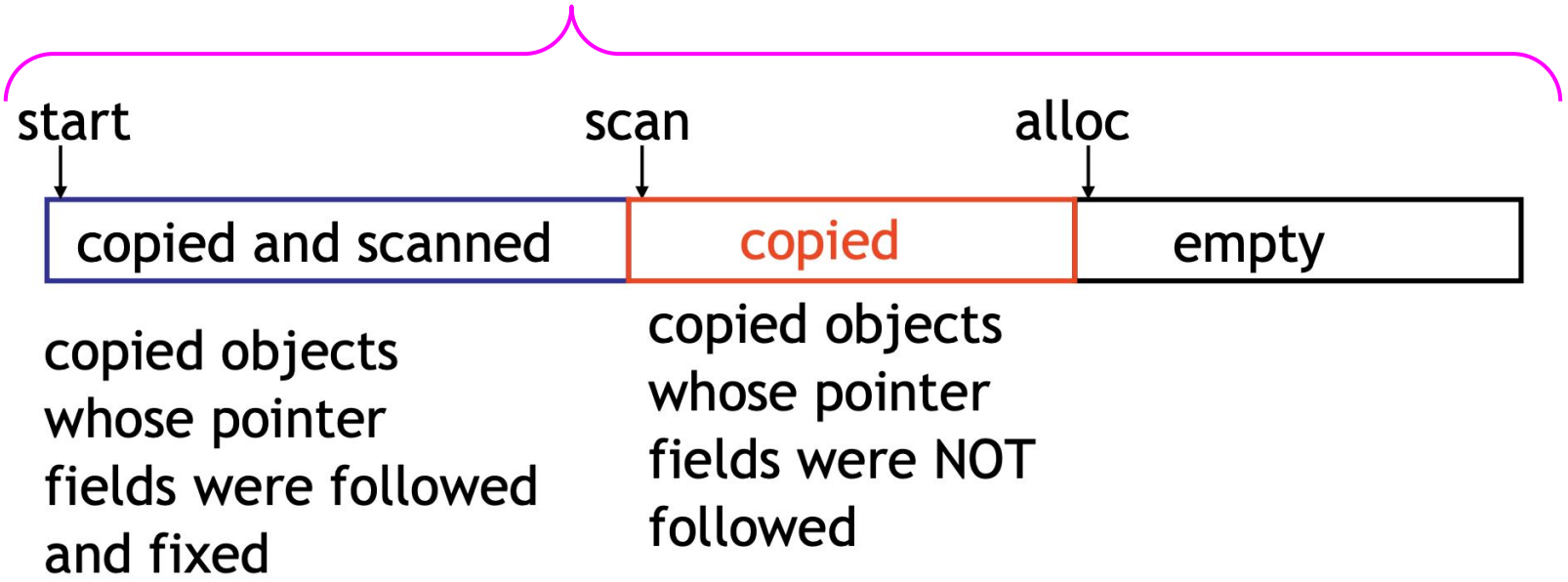
Stop and Copy: Implementation Trick

- Both problems can be solved via the following trick:
  - partition the **new space** into **three** contiguous regions:



Stop and Copy: Implementation Trick

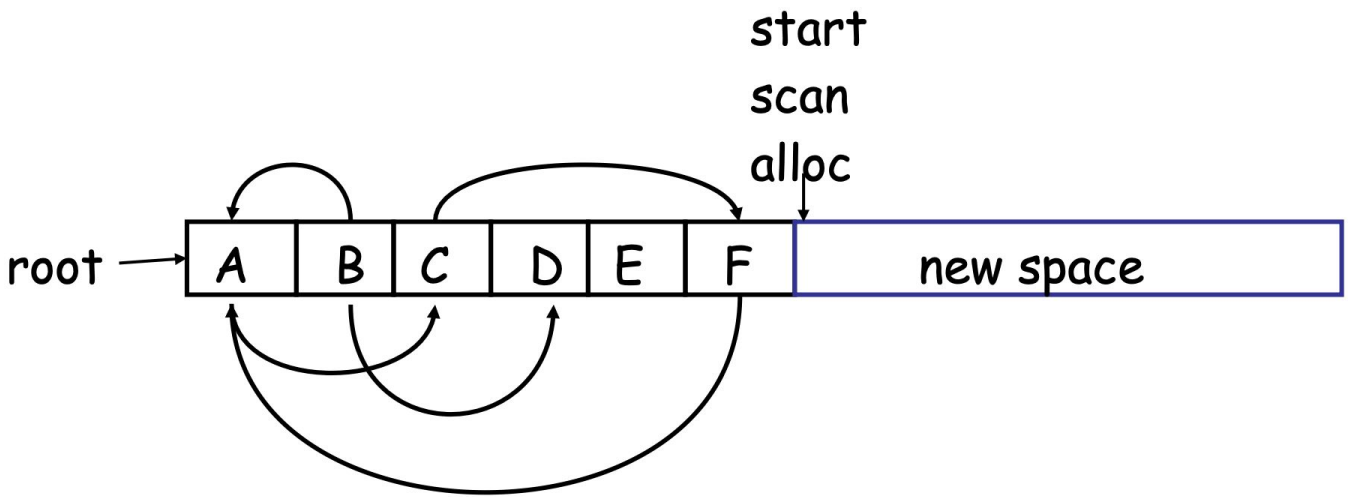
- Both problems can be solved via the following trick:
  - partition the **new space** into **three** contiguous regions:



Stop and Copy: Detailed Example

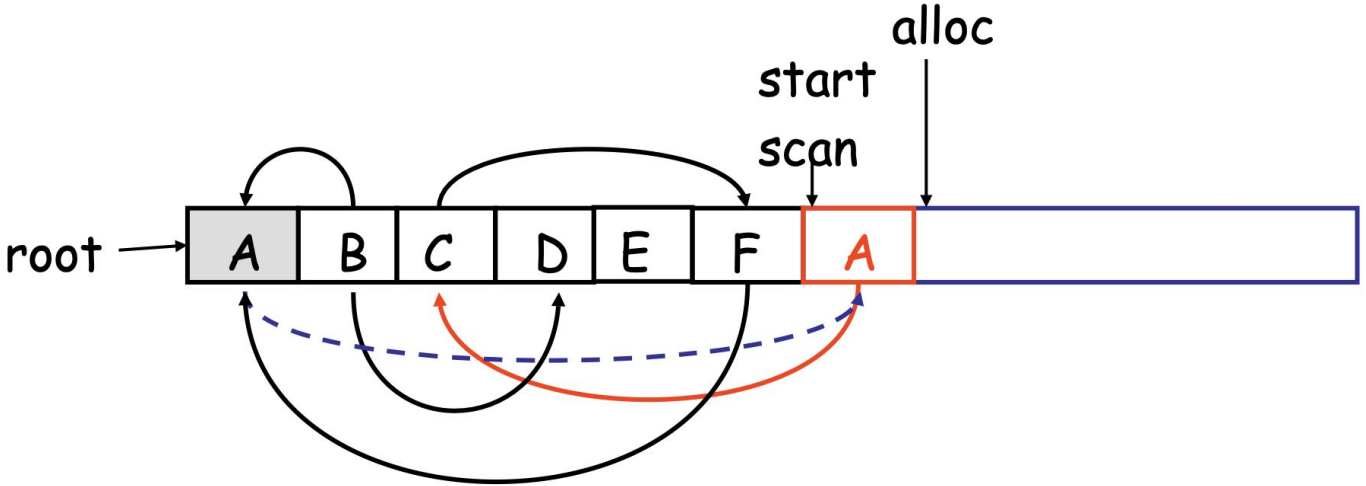
Stop and Copy: Detailed Example

- Before garbage collection:



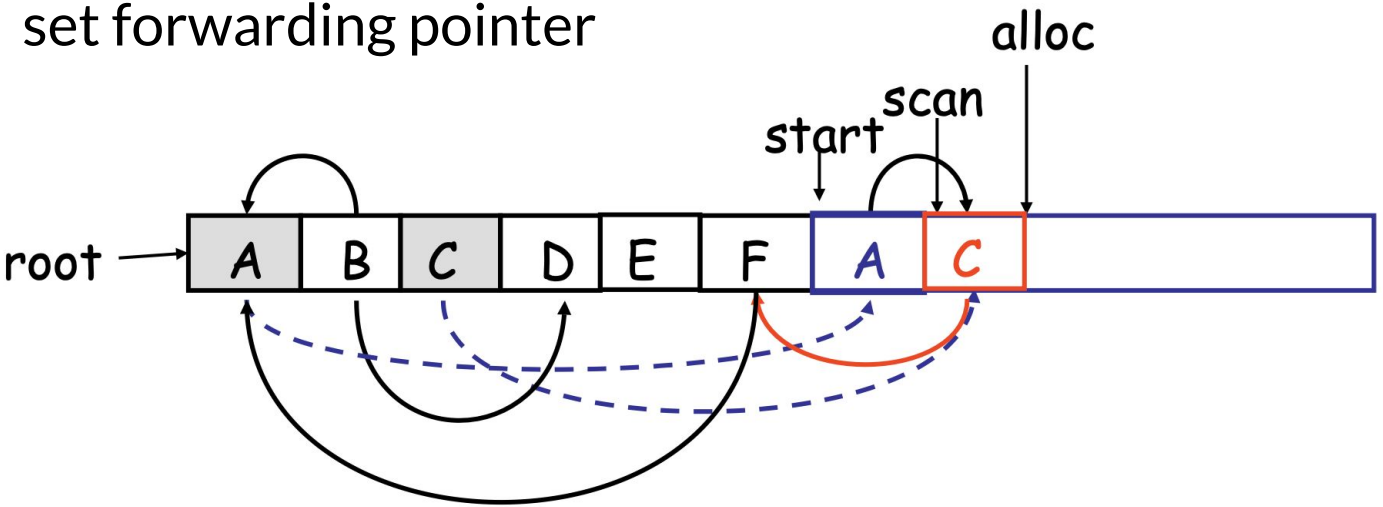
Stop and Copy: Detailed Example

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



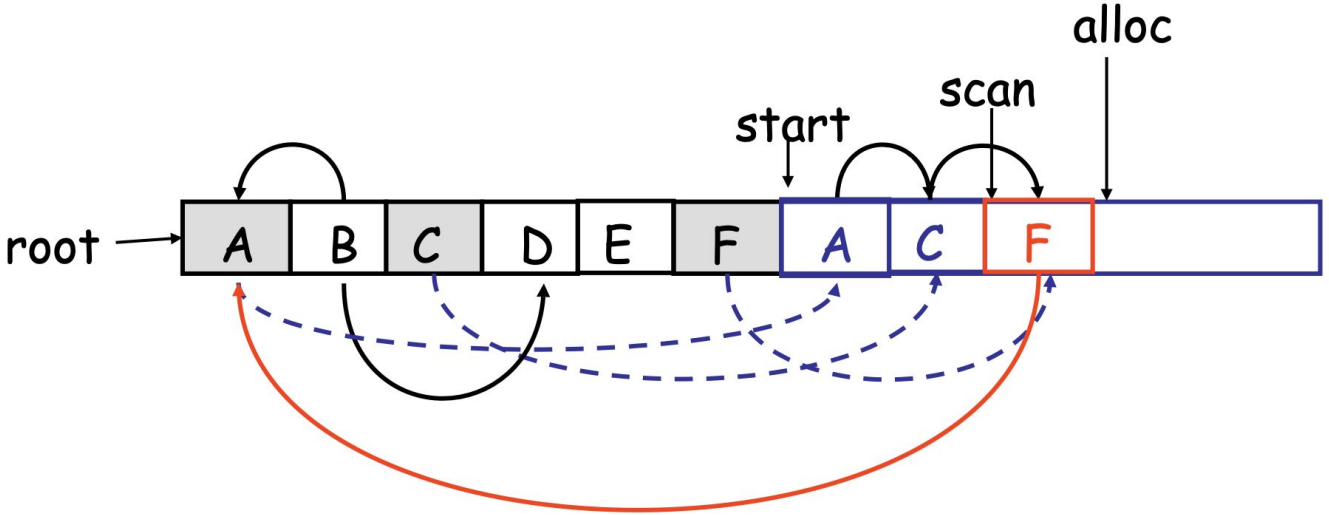
Stop and Copy: Detailed Example

- 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



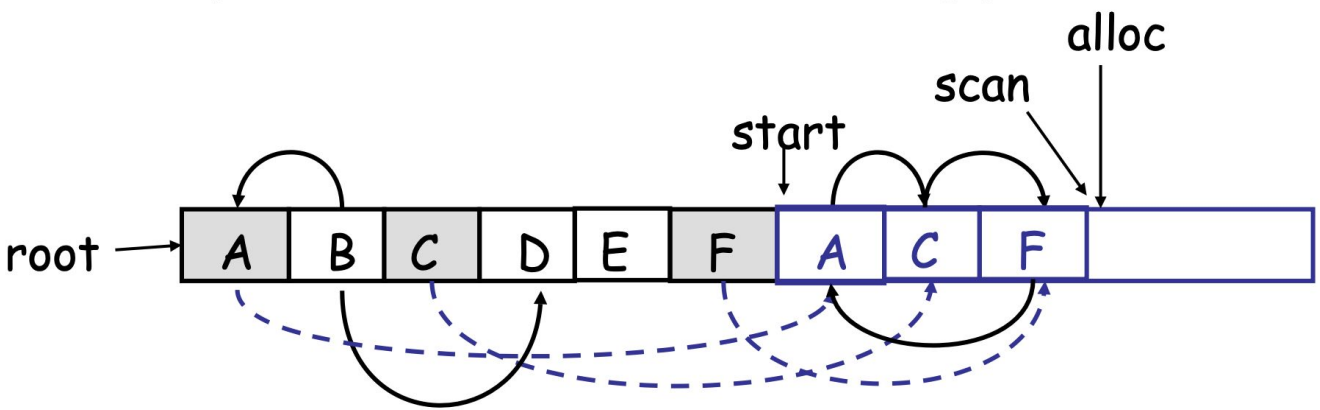
Stop and Copy: Detailed Example

- Step 3: Follow the pointer in the next unscanned object (C)
  - copy the pointed objects (F in this case)



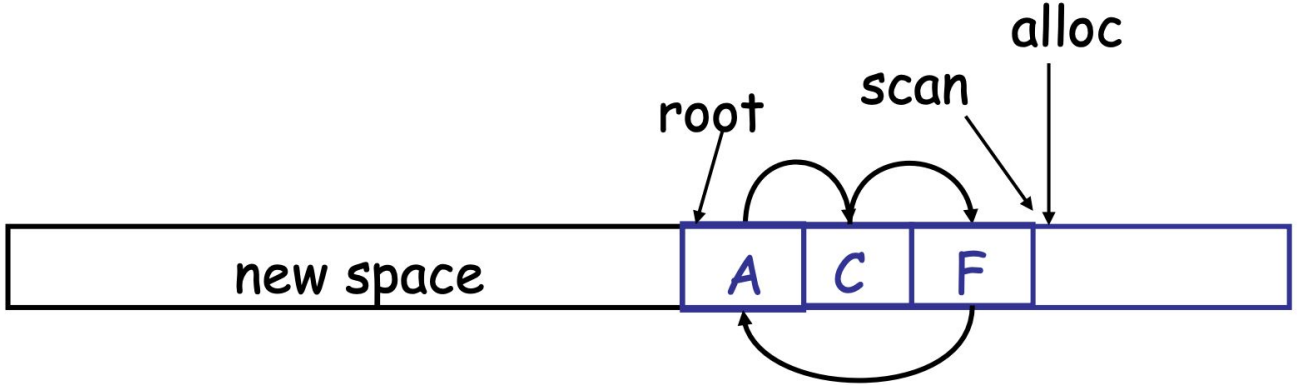
Stop and Copy: Detailed Example

- 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



Stop and Copy: Detailed Example

- Step 5: Since scan caught up with alloc, we are done
- Now, we swap the role of the two spaces and then resume the program



Stop and Copy: Algorithm

Stop and Copy: Algorithm

```
while scan != alloc do
        let O be the object at the 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
        rof
        increment scan pointer to the next object
od
```

Stop and Copy: Details

Stop and Copy: Details

- As with mark and sweep, we must be able to tell **how large** an object is when we scan it
  - Is this a problem in Cool?

Stop and Copy: Details

- As with mark and sweep, we must be able to tell **how large** an object is when we scan it
  - Is this a problem in Cool?
- We also need to know **where the pointers are** inside the object
  - How hard is this?

Stop and Copy: Details

- As with mark and sweep, we must be able to tell **how large** an object is when we scan it
  - Is this a problem in Cool?
- We also need to know **where the pointers are** inside the object
  - How hard is this?
- We must also copy any objects pointed to by the stack **and update** pointers in the stack

Stop and Copy: Details

- As with mark and sweep, we must be able to tell **how large** an object is when we scan it
  - Is this a problem in Cool?
- We also need to know **where the pointers are** inside the object
  - How hard is this?
- We must also copy any objects pointed to by the stack **and update** pointers in the stack
  - This can be an **expensive** operation in practice…

Stop and Copy: Evaluation

Stop and Copy: Evaluation

- Stop and copy is generally believed to be the **fastest** GC technique

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++, ...

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

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

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

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 languages like C or C++ it is **impossible** to identify the contents of objects in memory

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 languages like C or C++ it is **impossible** to identify the contents of objects in memory
  - e.g., how can you tell that 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)?

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 languages like C or C++ it is **impossible** to identify the contents of objects in memory
  - e.g., how can you tell that 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 we can be **conservative**:

Conservative Garbage Collection

- But we can be **conservative**:
  - If a memory word “looks like” a pointer, it is considered to be a pointer

Conservative Garbage Collection

- But we can be **conservative**:
  - If a memory word “looks like” a pointer, it is considered to be a pointer
    - e.g., if it is aligned (what does this mean?)

Conservative Garbage Collection

- But we can be **conservative**:
  - If a memory word “looks like” a pointer, it is considered to be a pointer
    - e.g., if it is aligned (what does this mean?)
    - it must point to a valid address in the data segment

Conservative Garbage Collection

- But we can be **conservative**:
  - If a memory word “looks like” a pointer, it is considered to be a pointer
    - e.g., if it is aligned (what does this mean?)
    - it must point to a valid address in the data segment
  - All such pointers are followed and we **overestimate** the reachable objects

Conservative Garbage Collection

- But we can be **conservative**:
  - If a memory word “looks like” a pointer, it is considered to be a pointer
    - e.g., if it is aligned (what does this mean?)
    - 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

Conservative Garbage Collection

- But we can be **conservative**:
  - If a memory word “looks like” a pointer, it is considered to be a pointer
    - e.g., if it is aligned (what does this mean?)
    - 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?

Conservative Garbage Collection

- But we can be **conservative**:
  - If a memory word “looks like” a pointer, it is considered to be a pointer
    - e.g., if it is aligned (what does this mean?)
    - 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 o
  pointers to them
  - What if what we thoug
    number?

Thus we **cannot** use stop-and-copy GC for languages with raw pointer fields that are indistinguishable from data at run time

Reference Counting

Reference Counting

- Rather that wait for memory to be exhausted, try to collect an object **when there are no more pointers to it**

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

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***

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

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
  - How **expensive** is that?

Reference Counting: Implementation

Reference Counting: Implementation

- new returns an object with a reference count of 1

Reference Counting: Implementation

- 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

Reference Counting: Implementation

- `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
- Code generated for every assignment `x <- y` must be changed:

Reference Counting: Implementation

- `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
- Code generated for 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: Implementation

- `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
- Code generated for 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
```

- (That's basically it!)

Reference Counting: Evaluation

- Advantages:

Reference Counting: Evaluation

- Advantages:
  - Easy to implement

Reference Counting: Evaluation

- Advantages:
  - Easy to implement
  - Collects garbage incrementally without large pauses in the execution
    - Why would we care about that?

Reference Counting: Evaluation

- Advantages:
  - Easy to implement
  - Collects garbage incrementally without large pauses in the execution
    - Why would we care about that?
- Disadvantages:

Reference Counting: Evaluation

- Advantages:
  - Easy to implement
  - Collects garbage incrementally without large pauses in the execution
    - Why would we care about that?
- Disadvantages:
  - Manipulating reference counts at each assignment is **very slow**

Reference Counting: Evaluation

- Advantages:
  - Easy to implement
  - Collects garbage incrementally without large pauses in the execution
    - Why would we care about that?
- Disadvantages:
  - Manipulating reference counts at each assignment is **very slow**
  - Cannot collect **circular structures**

Garbage Collection: Evaluation

Garbage Collection: Evaluation

- Automatic memory management avoids some serious storage bugs

Garbage Collection: Evaluation

- Automatic memory management avoids some serious storage bugs
- But it **takes away control** from the programmer

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

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

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

Garbage Collection: Other Approaches

Garbage Collection: Other Approaches

- **Concurrent**: allow the program to run while the collection is happening

Garbage Collection: Other Approaches

- **Concurrent**: allow the program to run while the collection is happening
- **Generational**: do not scan long-lived objects at every collection (infant mortality)
  - This approach, in particular, has seen wide adoption in practice (Java)

Garbage Collection: Other Approaches

- **Concurrent**: allow the program to run while the collection is happening
- **Generational**: do not scan long-lived objects at every collection (infant mortality)
  - This approach, in particular, has seen wide adoption in practice (Java)
- **Parallel**: several collectors working in parallel

Garbage Collection: Other Approaches

- **Concurrent**: allow the program to run while the collection is happening
- **Generational**: do not scan long-lived objects at every collection (infant mortality)
  - This approach, in particular, has seen wide adoption in practice (Java)
- **Parallel**: several collectors working in parallel
- **Real-Time / Incremental**: no long pauses

Garbage Collection: IRL

Garbage Collection: IRL

- **Python** uses **Reference Counting**
  - Because of “extension modules”, they deem it too difficult to determine the root set
  - Has a special separate cycle detector
- **Perl** does **Reference Counting** + cycles

Garbage Collection: IRL

- **Python** uses **Reference Counting**
  - Because of “extension modules”, they deem it too difficult to determine the root set
  - Has a special separate cycle detector
- **Perl** does **Reference Counting** + cycles
- **Ruby** does **Mark and Sweep**

Garbage Collection: IRL

- **Python** uses **Reference Counting**
  - Because of “extension modules”, they deem it too difficult to determine the root set
  - Has a special separate cycle detector
- **Perl** does **Reference Counting** + cycles
- **Ruby** does **Mark and Sweep**
- **OCaml** does **(generational) Stop and Copy**
- **Java** does **(generational) Stop and Copy**
- **Node.js** does **(generational) Stop and Copy**

Automatic Memory Management: Summary

Automatic Memory Management: Summary

- An **automatic memory management** system deallocates objects when they are no longer used and reclaims their storage space.

Automatic Memory Management: Summary

- An **automatic memory management** system deallocates objects when they are no longer used and reclaims their storage space.
- We must be **conservative** and only free objects that won't be used later.

Automatic Memory Management: Summary

- An **automatic memory management** system deallocates objects when they are no longer used and reclaims their storage space.
- We must be **conservative** and only free objects that won't be used later.
- Garbage collection scans the heap from a set of roots to find **reachable** objects. We saw three algorithms:

Automatic Memory Management: Summary

- An **automatic memory management** system deallocates objects when they are no longer used and reclaims their storage space.
- We must be **conservative** and only free objects that won't be used later.
- Garbage collection scans the heap from a set of roots to find **reachable** objects. We saw three algorithms:
  - **Mark and Sweep** uses a per-object mark bit to track which objects are reachable.

Automatic Memory Management: Summary

- An **automatic memory management** system deallocates objects when they are no longer used and reclaims their storage space.
- We must be **conservative** and only free objects that won't be used later.
- Garbage collection scans the heap from a set of roots to find **reachable** objects. We saw three algorithms:
  - **Mark and Sweep** uses a per-object mark bit to track which objects are reachable.
  - **Stop and Copy** has low overhead but stalls the program and requires language support

Automatic Memory Management: Summary

- An **automatic memory management** system deallocates objects when they are no longer used and reclaims their storage space.
- We must be **conservative** and only free objects that won't be used later.
- Garbage collection scans the heap from a set of roots to find **reachable** objects. We saw three algorithms:
  - **Mark and Sweep** uses a per-object mark bit to track which objects are reachable.
  - **Stop and Copy** has low overhead but stalls the program and requires language support
  - **Reference Counting** stores the number of pointers to an object with that object and frees it when that count reaches zero

Course Announcements

- As some of you have noticed, the PA4 leaderboard results are subject to **timing variance**
  - In particular, sometimes you “get lucky” and your compiler’s code appears to be much, much faster than the reference
  - There’s not much I can do about this, unfortunately.
  - We will run each final submission several times and take the **median** Q score (not the best), so if you see a big speedup without doing anything, you should assume it is ephemeral
- PA4c1 is due on Monday
- I will not hold office hours next Wednesday