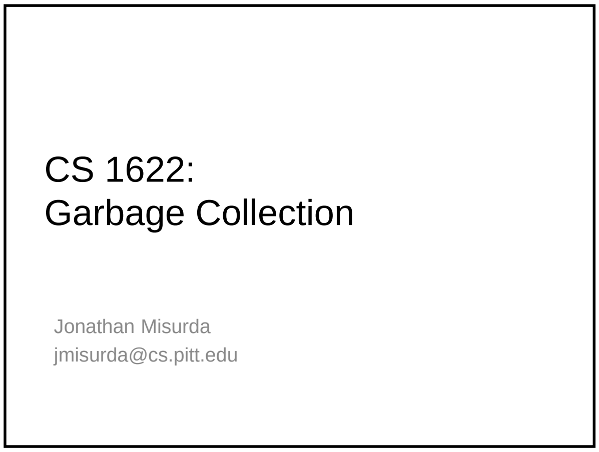
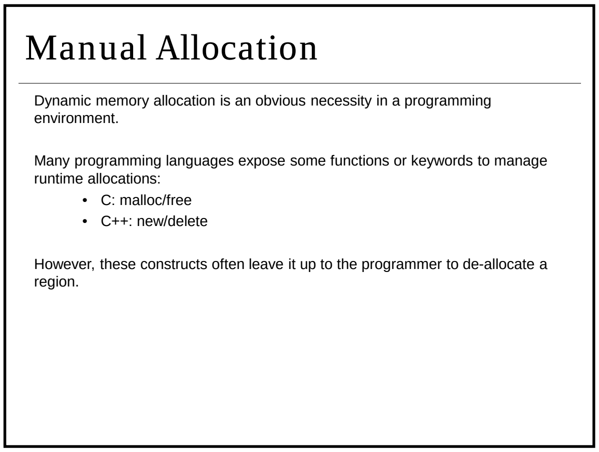


CS 1622:
Garbage Collection

Jonathan Misurda
jmisurda@cs.pitt.edu



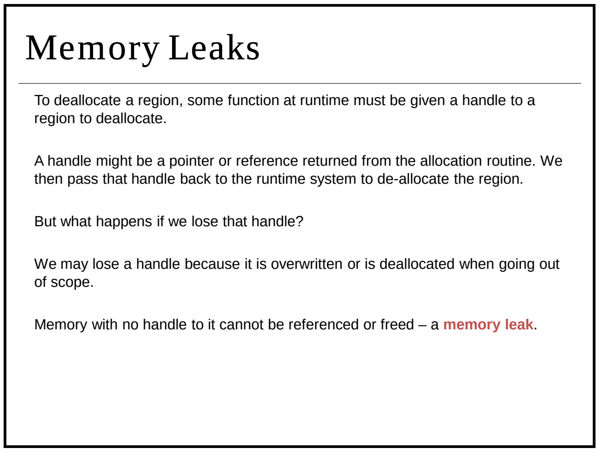
Manual Allocation

Dynamic memory allocation is an obvious necessity in a programming environment.

Many programming languages expose some functions or keywords to manage runtime allocations:

- C: malloc/free
- C++: new/delete

However, these constructs often leave it up to the programmer to de-allocate a region.



Memory Leaks

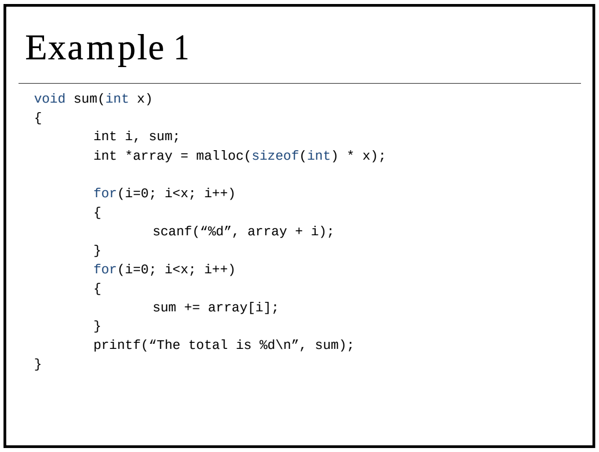
To deallocate a region, some function at runtime must be given a handle to a region to deallocate.

A handle might be a pointer or reference returned from the allocation routine. We then pass that handle back to the runtime system to de-allocate the region.

But what happens if we lose that handle?

We may lose a handle because it is overwritten or is deallocated when going out of scope.

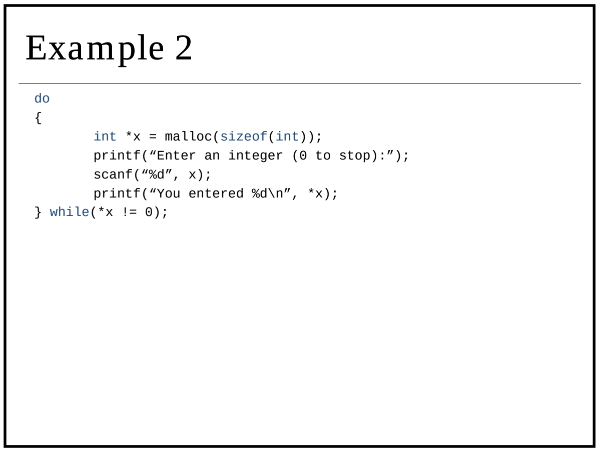
Memory with no handle to it cannot be referenced or freed – a **memory leak**.



Example 1

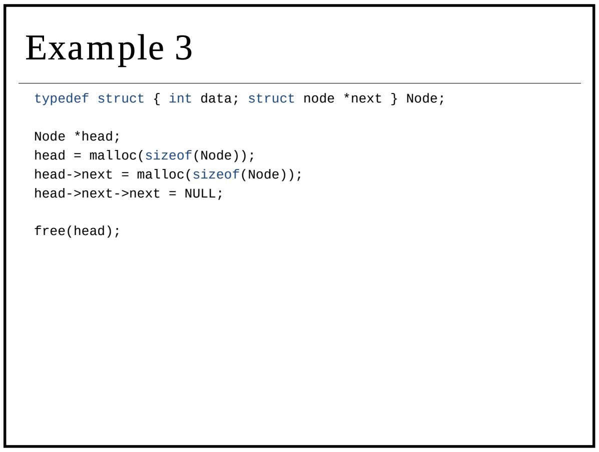
```
void sum(int x)
{
        int i, sum;
        int *array = malloc(sizeof(int) * x);

        for(i=0; i<x; i++)
        {
                scanf(“%d”, array + i);
        }
        for(i=0; i<x; i++)
        {
                sum += array[i];
        }
        printf(“The total is %d\n”, sum);
}
```



Example 2

```
do
{
        int *x = malloc(sizeof(int));
        printf(“Enter an integer (0 to stop):”);
        scanf(“%d”, x);
        printf(“You entered %d\n”, *x);
} while(*x != 0);
```



Example 3

```
typedef struct { int data; struct node *next } Node;

Node *head;
head = malloc(sizeof(Node));
head->next = malloc(sizeof(Node));
head->next->next = NULL;

free(head);
```

Garbage Collection

It can be hard to find memory leaks and they are easy mistakes to make.

A memory leak may eventually cause a program to terminate/crash due to being out of memory.

Can we automate memory de-allocation instead of relying on the programmer to do it properly?

- **Garbage Collection**

First we need to define garbage

- Must automatically detect that the program will never need a dynamically-allocated memory region again.

And then find some efficient way to make it go away.

Garbage

Garbage can be defined using the ideas we have already presented. **Garbage** is a region of memory with no way to find it:

- i.e., we've leaked it

The garbage collection depends on the ability to determine if an object is **reachable**.

But if we've lost all handles to it, how do we know what to reclaim?

In general, this can be an impossible task. But we may be able to augment the runtime system or compiler to make the task possible in most cases.

Reachability

Named object: something that had a name at compile time
Nameless object: something that was referenced only by address at runtime

```
struct Node *head = malloc(sizeof(struct node));
```

The pointer head is a named objects, the struct node on the heap is a nameless object.

An object x is **reachable** iff:

- A named object contains a handle to x
- Another reachable object y contains a handle to x

Garbage Collection

In general, garbage collection algorithms all operate similarly:

1. Allocate memory as needed at runtime
2. When space "runs out":
    a. Compute what might be used again by finding reachable objects
    b. Free space unused by the objects found in the previous step

We find reachable objects by starting with a set of **root** pointers. These are named objects that are held in registers, globals, or on the stack.

This means that we need to know the layout of objects so we can identify pointers versus other values.

- Is 0x080407d4 an address or an int?

Reference Counting

1. For every object (region of dynamically allocated memory):
    a) Retain an internal counter
    b) Increment when a reference is made to it
    c) Decrement when a reference is lost to it

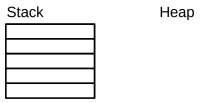
2. When counter is zero, free.

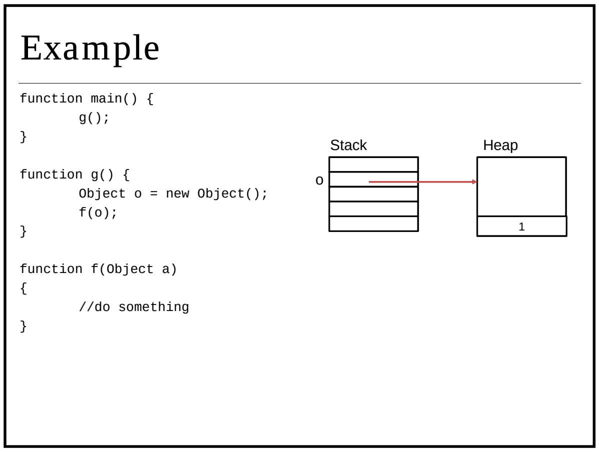
Example

```
function main() {
        g();
}

function g() {
        Object o = new Object();
        f(o);
}

function f(Object a)
{
        //do something
}
```



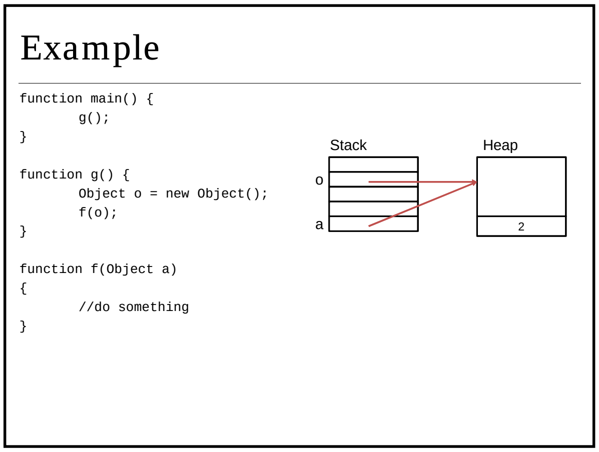


Example

```
function main() {
        g();
}

function g() {
        Object o = new Object();
        f(o);
}

function f(Object a)
{
        //do something
}
```

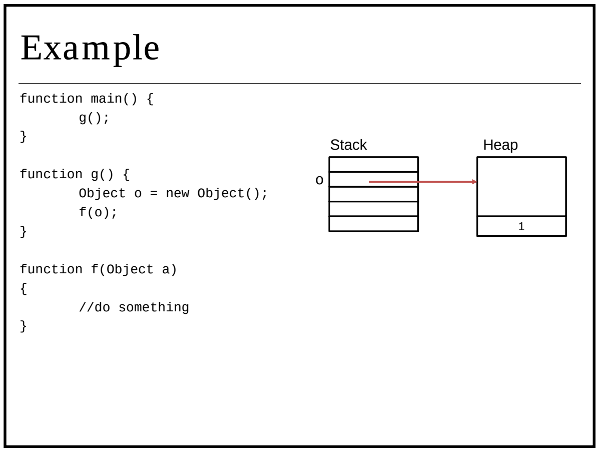


Example

```
function main() {
        g();
}

function g() {
        Object o = new Object();
        f(o);
}

function f(Object a)
{
        //do something
}
```

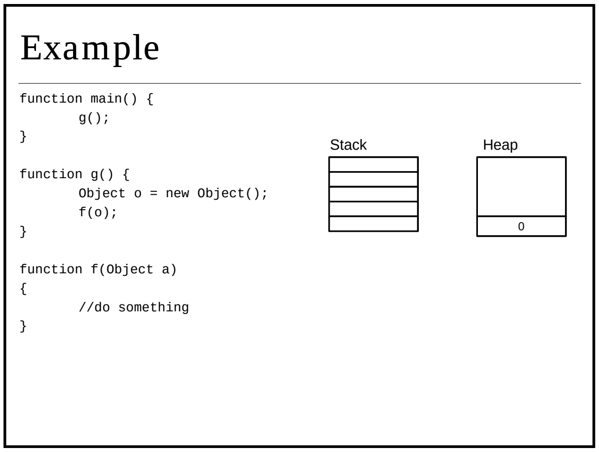


Example

```
function main() {
        g();
}

function g() {
        Object o = new Object();
        f(o);
}

function f(Object a)
{
        //do something
}
```

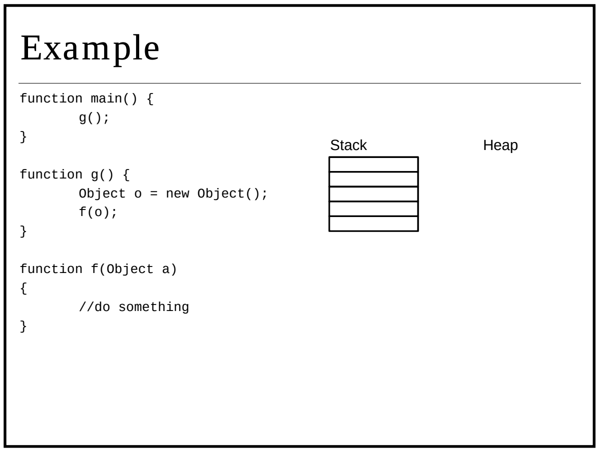


Example

```
function main() {
        g();
}

function g() {
        Object o = new Object();
        f(o);
}

function f(Object a)
{
        //do something
}
```

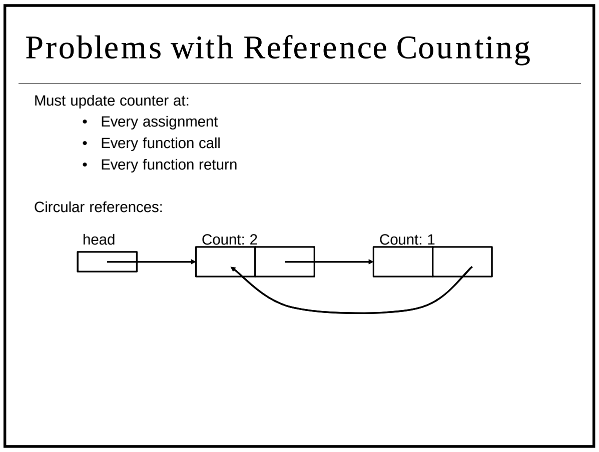


Example

```
function main() {
        g();
}

function g() {
        Object o = new Object();
        f(o);
}

function f(Object a)
{
        //do something
}
```

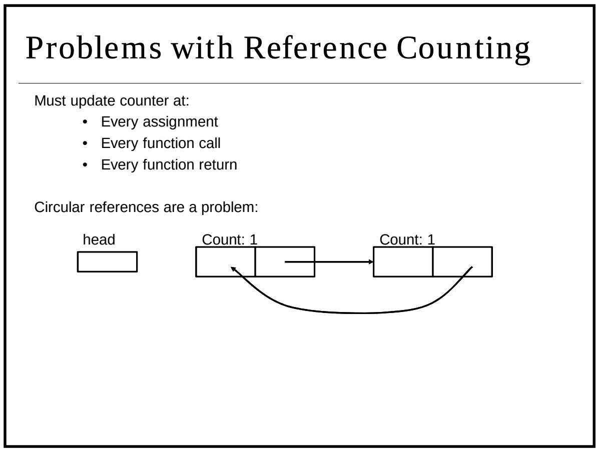


Problems with Reference Counting

Must update counter at:

- Every assignment
- Every function call
- Every function return

Circular references:



Problems with Reference Counting

Must update counter at:

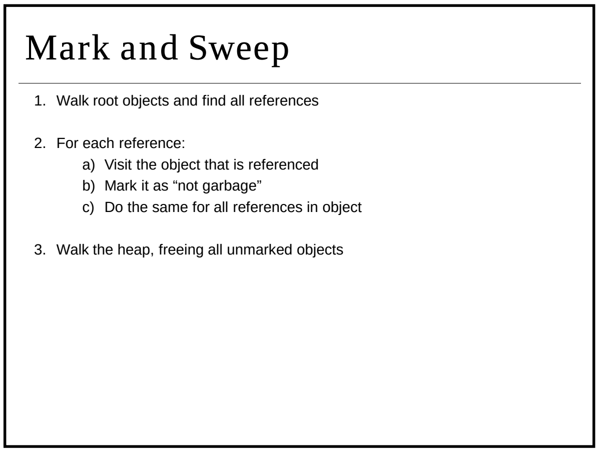
- Every assignment
- Every function call
- Every function return

Circular references are a problem:

head

Count: 1

Count: 1

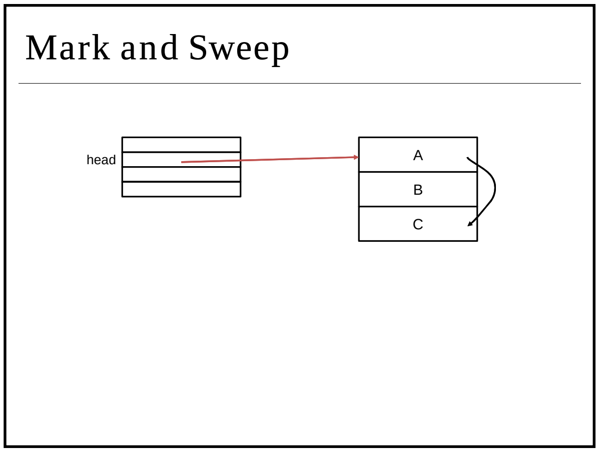


Mark and Sweep

1. Walk root objects and find all references

2. For each reference:
   a) Visit the object that is referenced
   b) Mark it as “not garbage”
   c) Do the same for all references in object

3. Walk the heap, freeing all unmarked objects



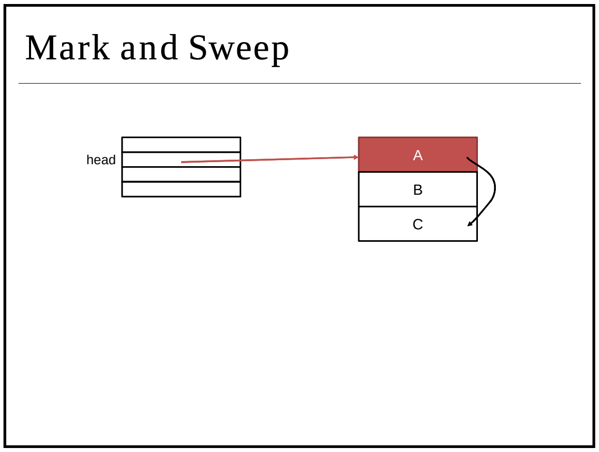
Mark and Sweep

head

A

B

C



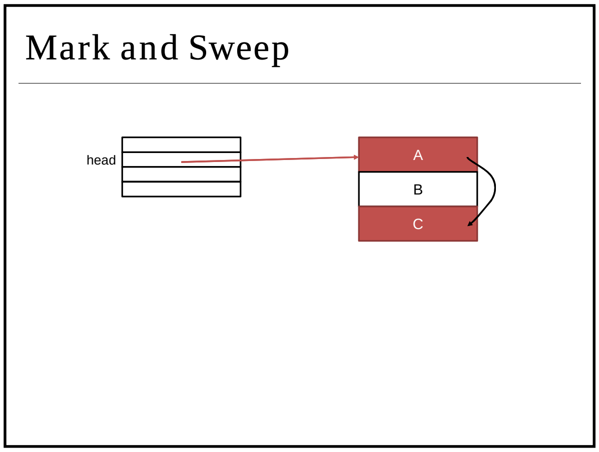
Mark and Sweep

head

A

B

C



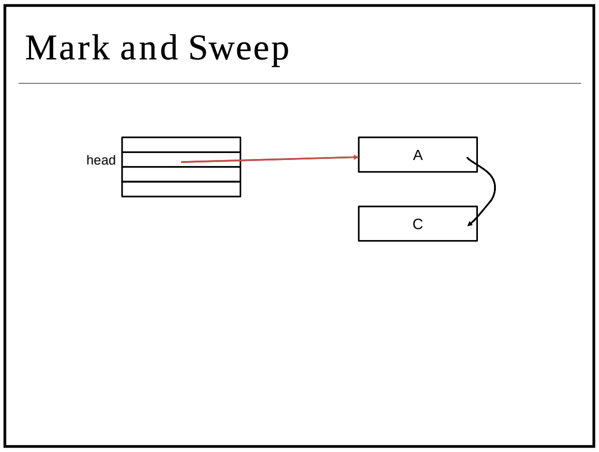
Mark and Sweep

head

A

B

C



Mark and Sweep

head

A

C

Advantages and Disadvantages

No longer need to modify reference counter during program execution.

However, Mark and Sweep is a stop-the-world collector.

- We cannot do this while the objects are being used.

External fragmentation is a considerable problem.

DFS potentially requires a large stack in a high-degree graph.

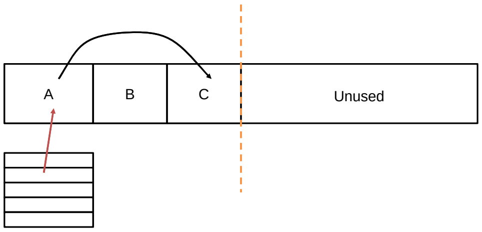
Copying Collectors

Try to avoid issues of external fragmentation by compacting used space.

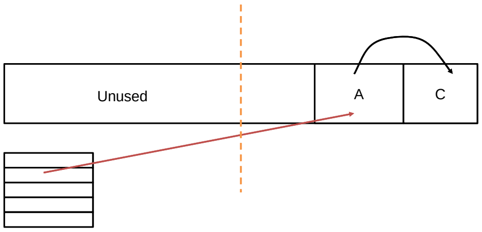
1. Divide memory into halves
2. Only allocate from first half
3. When half is (nearly) full
   a) Walk root objects and recursively copy every object to unused half

The most basic copying collector is called Stop and Copy or the SemiSpace collector

Stop and Copy



Stop and Copy



Problems

Slow to collect (another stop-the-world algorithm).

Moving objects hurts cache performance.

Wasteful of memory.

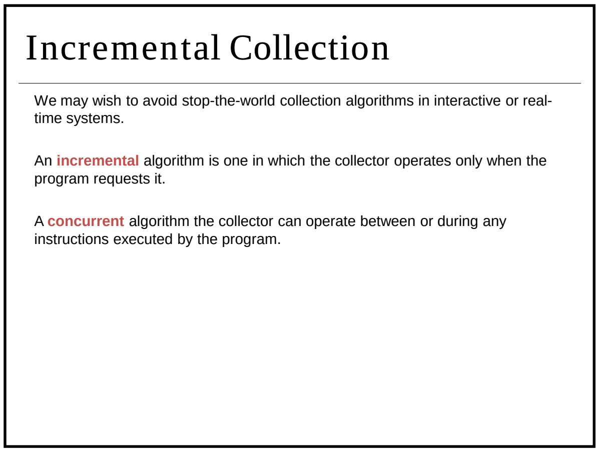
All pointers need to be updated to point to new location

- Alternatively use a layer of indirection called a table of contents

Generational Garbage Collectors

Most objects have a short lifetime, so if an object makes it through a GC without being collected, don't try to collect it again for a while.

1. Divide the heap into several partitions (generations)
2. Objects are allocated from the youngest generation
3. When a generation is full:
   a) move live objects to an older generation

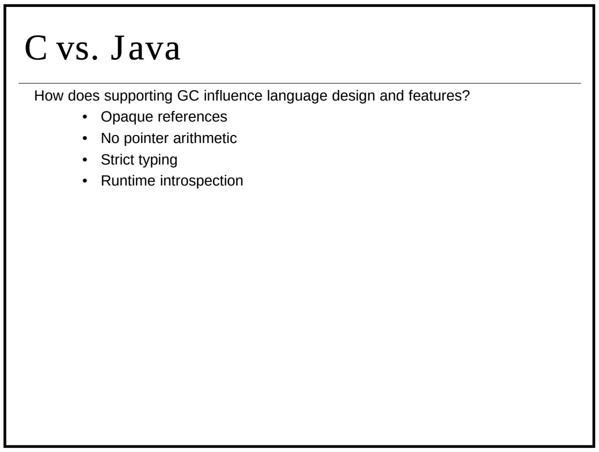


Incremental Collection

We may wish to avoid stop-the-world collection algorithms in interactive or real-time systems.

An **incremental** algorithm is one in which the collector operates only when the program requests it.

A **concurrent** algorithm the collector can operate between or during any instructions executed by the program.



C vs. Java

How does supporting GC influence language design and features?

- Opaque references
- No pointer arithmetic
- Strict typing
- Runtime introspection