Garbage Collection

CS449 Spring 2016

Manual Allocation

- Dynamic memory allocation
  - malloc/free
  - new/delete

- Leave it up to the programmer to de-allocate a region

Memory Leaks

- Given a handle to a region
  - A pointer or reference
  - Pass that handle back to de-allocate
- What happens if we lose that handle?
  - Overwritten
  - Out of scope
- Memory cannot be referenced or freed – a *memory leak*

Example I

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

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

Example III

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

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

free(head);
```

Garbage Collection

- Can be hard to find memory leaks
- Are easy mistakes to make
- May eventually cause a program to crash due to being out of memory

- So let's automate memory de-allocation
  - *Garbage Collection*

Now you have two problems

- How do we define garbage?
  - Must automatically detect that the program will never need a memory region again
- How do we collect it?
  - Want some efficient way to make it go away

Garbage

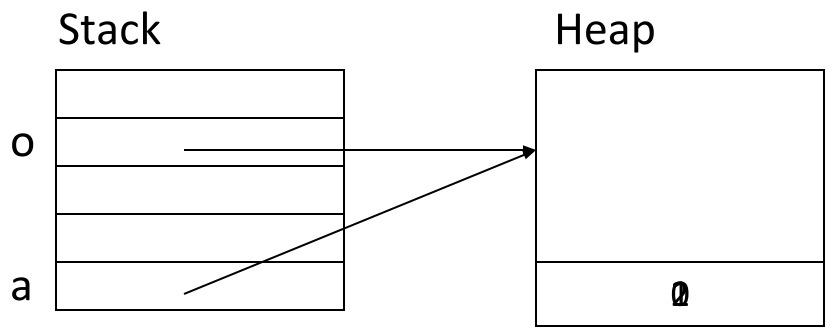
- A region of memory with no way to find it
  - i.e., we've leaked it
- But if we've lost all references (pointers) to it, how do we know what to reclaim?

Reference Counting

- For every object (region of dynamically allocated memory)
  - Retain an internal counter
  - Increment when a reference is made to it
  - Decrement when a reference is lost to it
- When counter is zero, free

Example

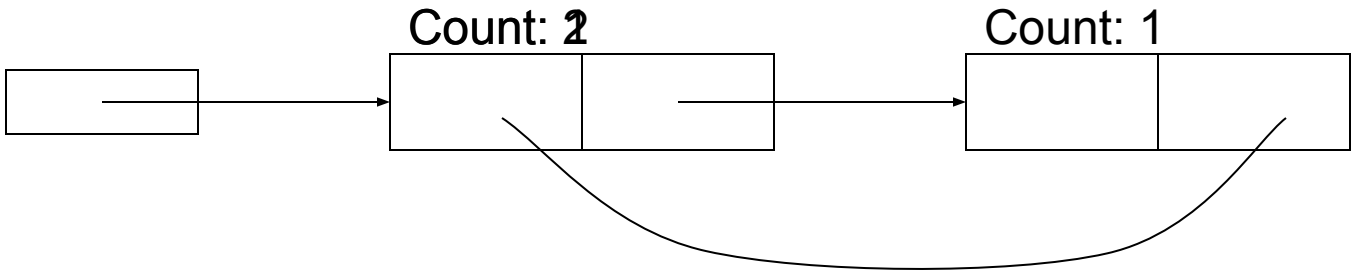
```
function main() {
    g();
}

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

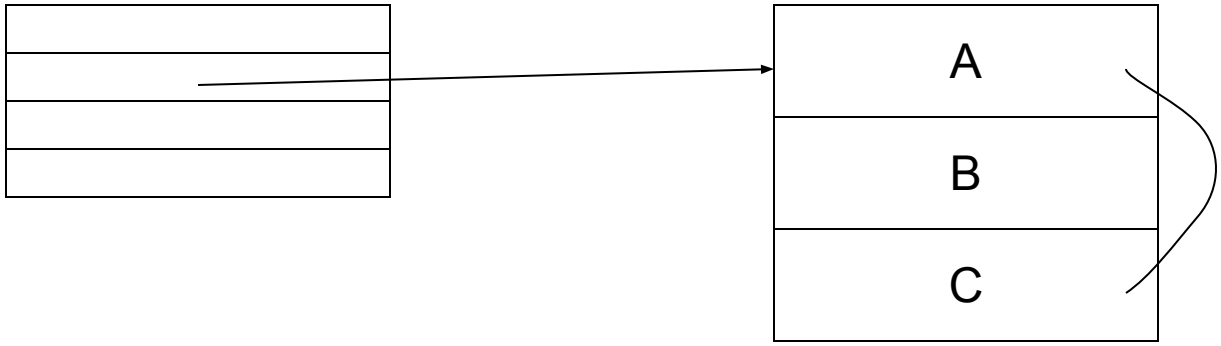
- Circular references



Mark and Sweep

- Maintain a list of all references
- For each reference:
  - Visit the object that is referenced
  - Mark it as “not garbage”
  - Do the same for all references in object
- Walk the heap, freeing all unmarked objects

Mark and Sweep



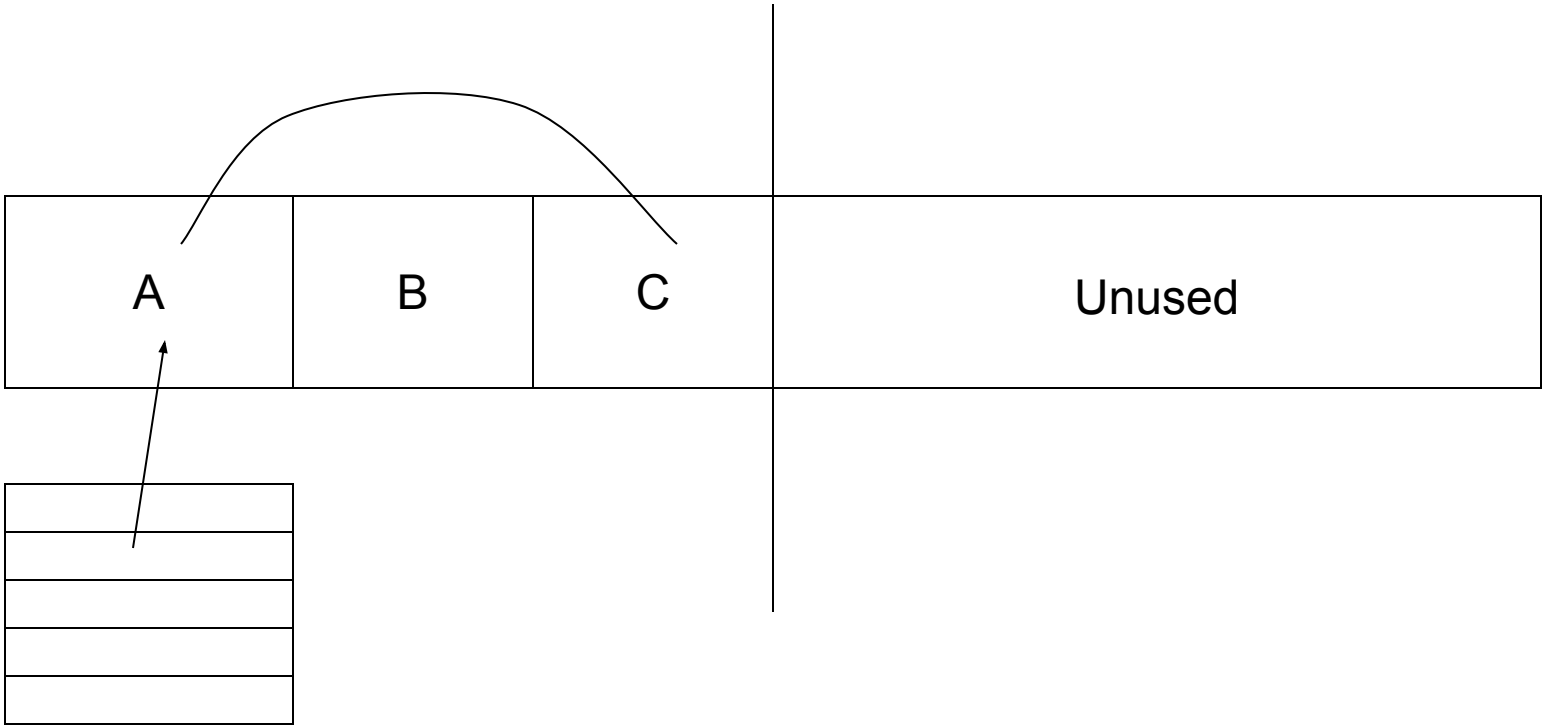
Advantages and Disadvantages

- No longer need to increment reference counter
- Cycles still can be problematic
- External Fragmentation

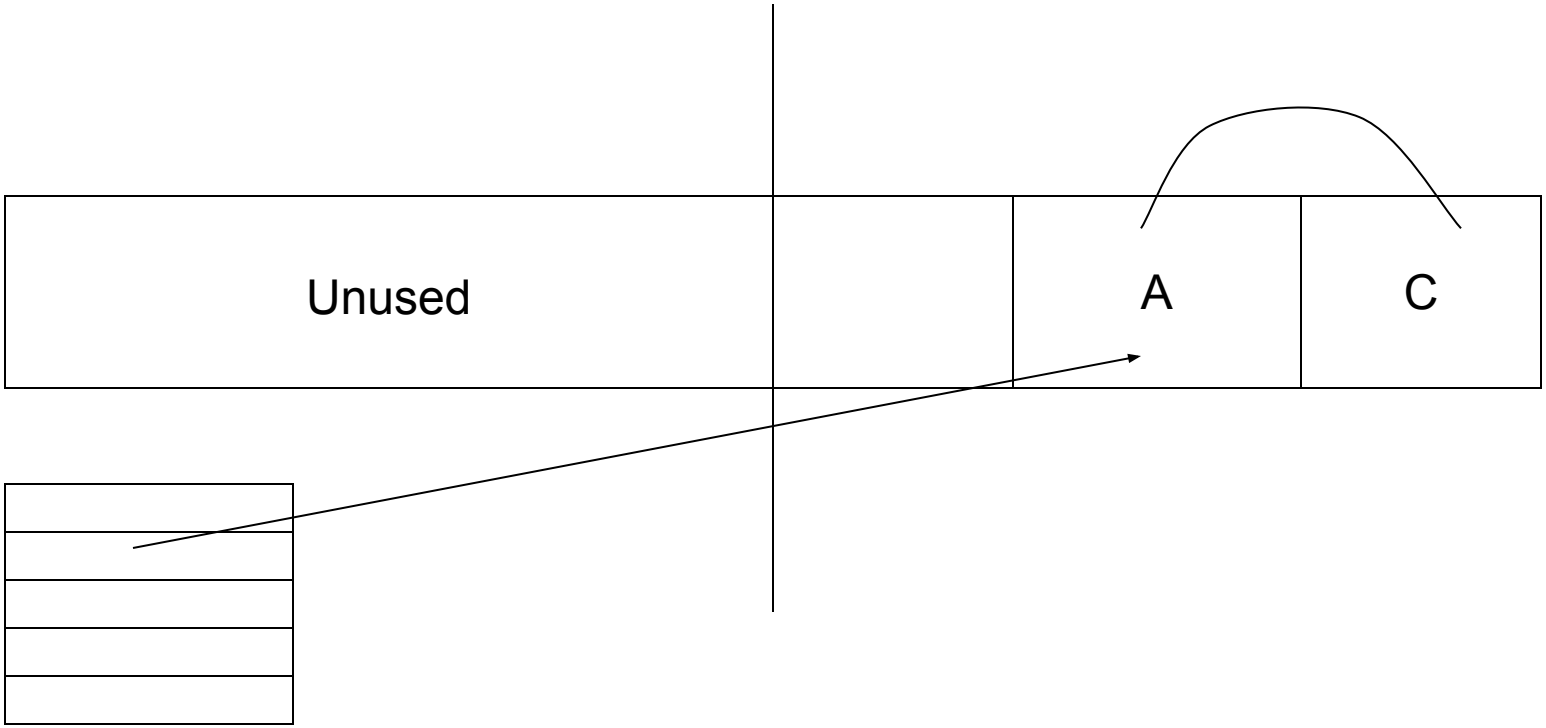
Copying Collectors

- Try to avoid issues of fragmentation
- Divide memory into halves
- Only allocate from first half
- When half is nearly full
  - Walk stack and recursively copy every object to unused half

Stop and Copy



Stop and Copy



Problems

- Slow
- 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*

C vs. Java

- How does supporting GC influence language design and features?
  - Opaque references
  - No pointer arithmetic
  - Strict typing