

# COS 217: Introduction to Programming Systems

Indirection

Pointers, Arrays, Strings, Command Line Arguments,  
Structures, and Dynamic Memory

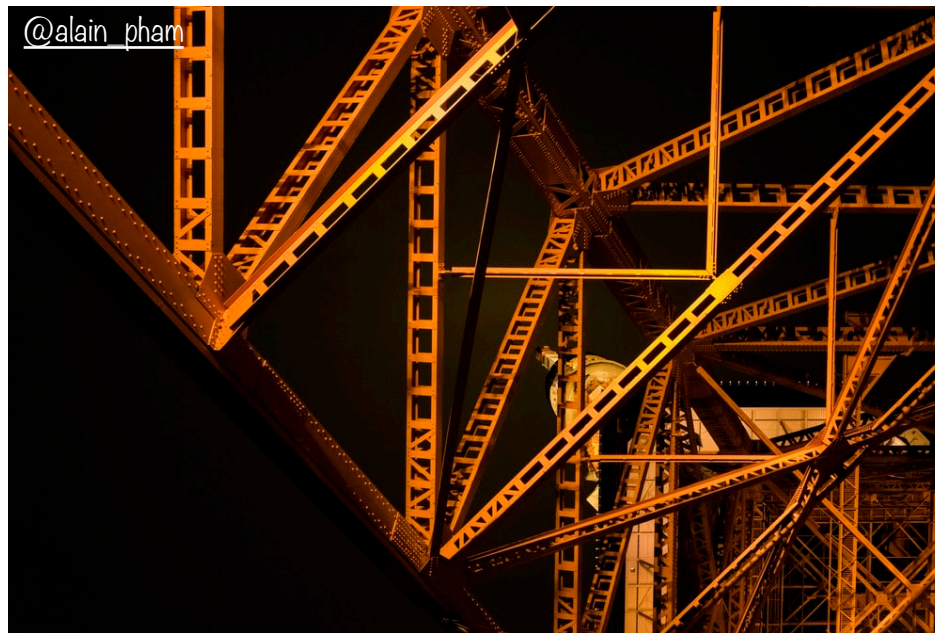


**PRINCETON UNIVERSITY**



# Indirection

- Pointers
- Arrays
  - Accessing array elements through pointers, use in pointer arithmetic
  - Array names are not pointers
- Strings as arrays of characters (terminated by NULL char)
- Command line arguments
  - argv: array of variable no. of pointers, each to variable-length string
  - argc: the number of these pointers
- Structures
- Indirection and Dynamic Memory



# INDIRECTION AND C STRUCTURES



# Why Structures

- Useful to have data structures and types for collections of elements; e.g. arrays
- In an array, every element is of the same type (e.g. ints, pointers, characters)
- What about a data structure for collections of elements of different types?
  - Flexible records for (related) data, such as student ID, name, age, home address, ...

```
Enum {MAX_NAME = 64, MAX_HOME_ADDR = 256}
```

```
struct SRec {  
    int ID ;  
    char name[NAX_NAME];  
    int age_in_yr;  
    char home_address[MAX_HOME];  
    float GPA;  
};
```



# C Struct

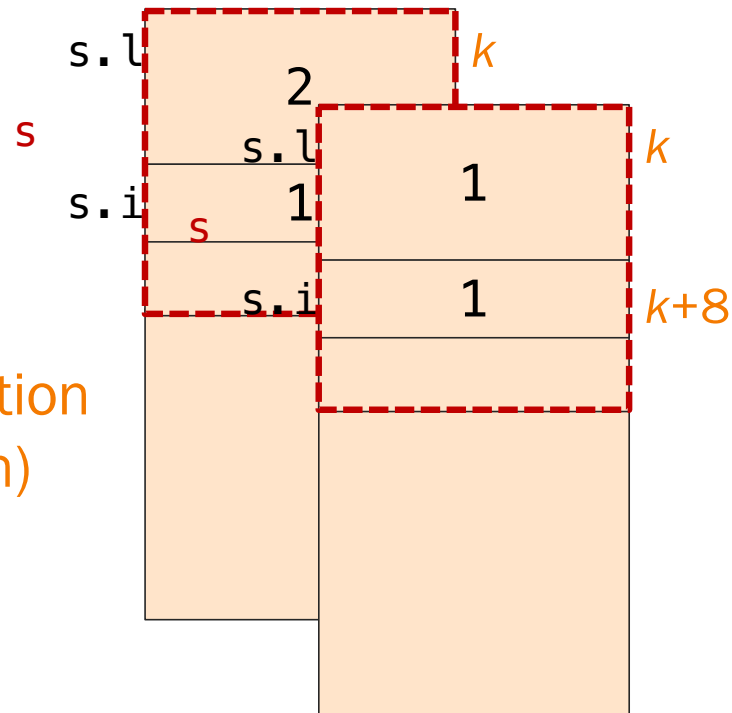
```
struct S {  
    long l;  
    int i;  
};
```

Type

```
struct S s = {2L, 1};
```

Variable Declaration  
(and Initialization)

```
s.l = s.i;
```





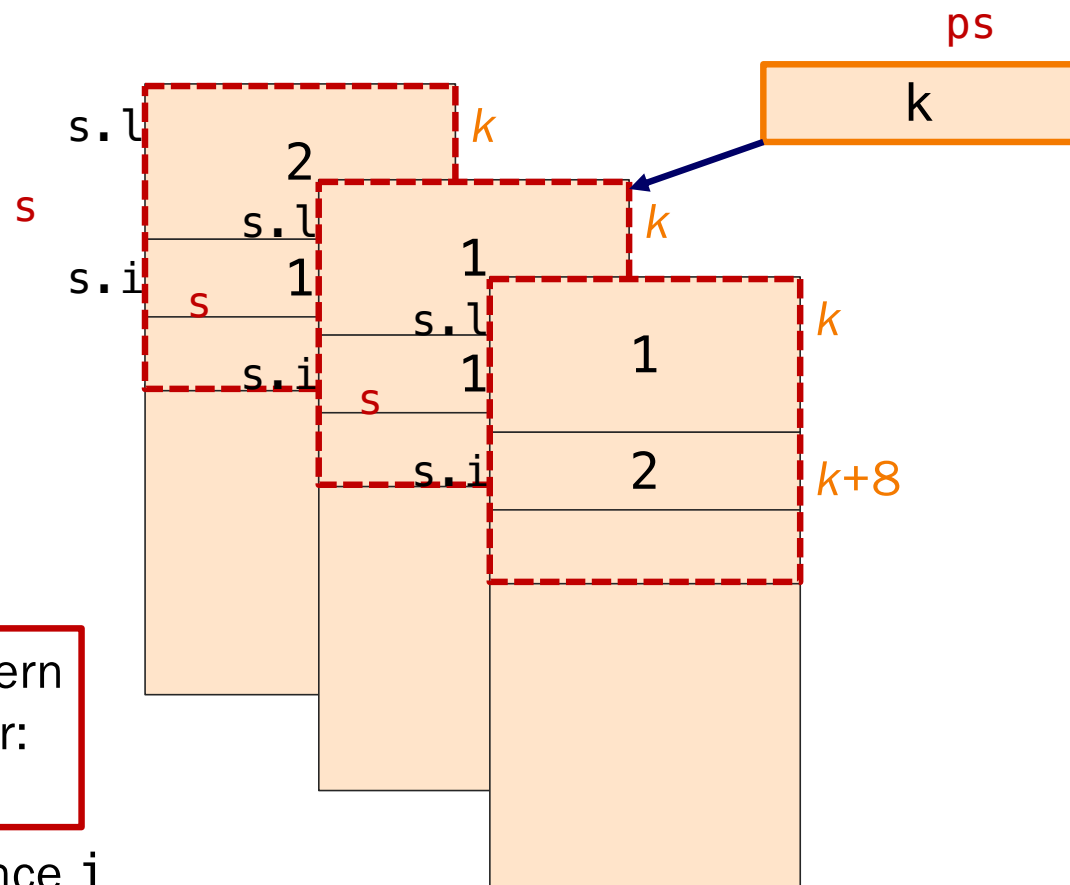
# C Struct

```
struct S {  
    long l;  
    int i;  
};  
  
struct S s = {2L, 1};  
struct S *ps = &s;  
  
s.l = s.i;  
  
(*ps).i *= 2;
```

This is such a common pattern  
that it has its own operator:

ps->i

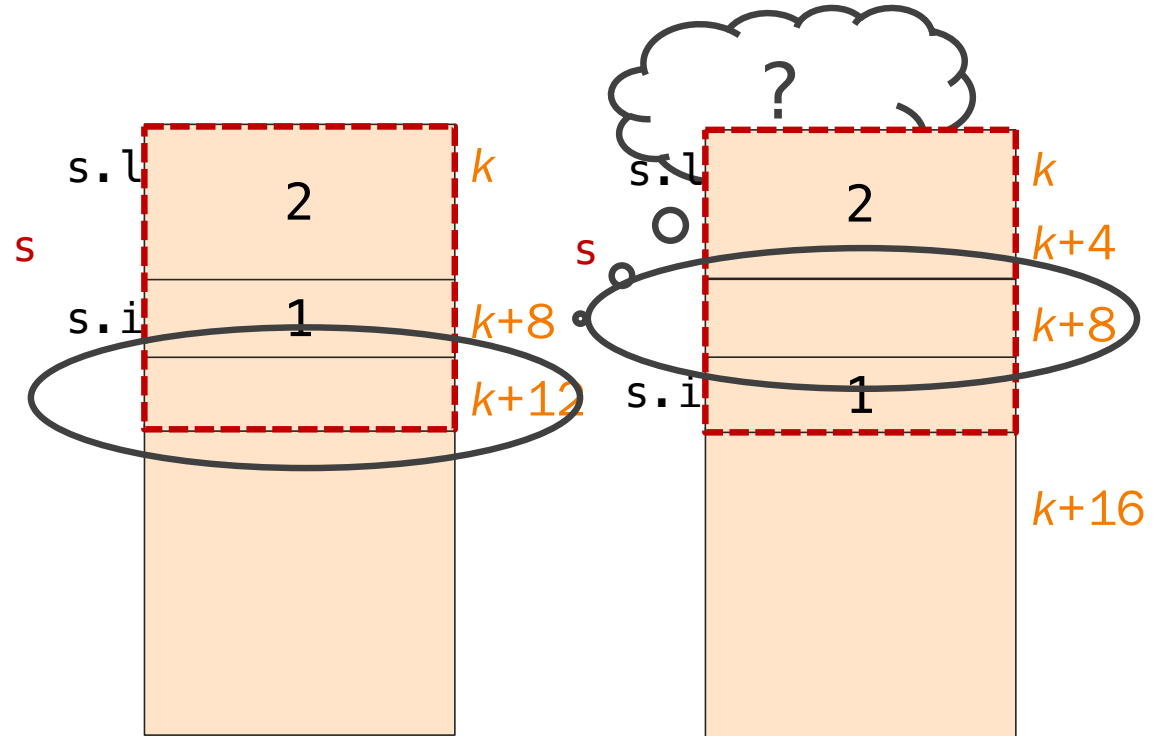
At least three ways to reference i





# Interface and Implementation: Padding in Structs

```
struct S {  
    long l;  
    int i;  
};  
  
struct S s = {2L, 1};
```



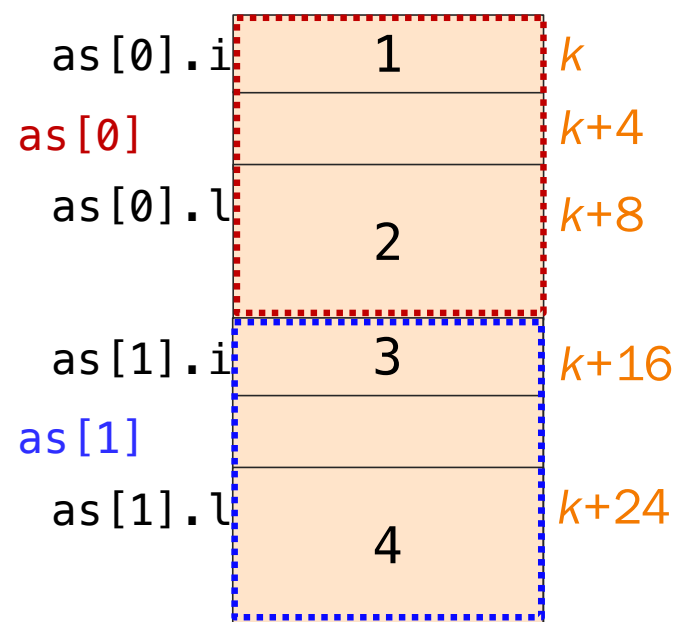
- So, use the interface given (`ps->l`, `ps->i`), don't try to know the implementation



# Arrays of Structs

```
struct S {
    int i;
    long l;
};

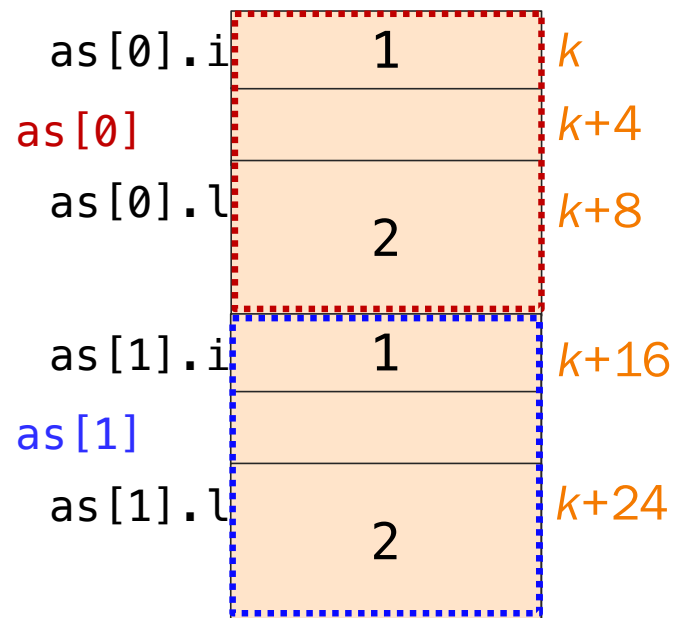
struct S as[2] =
    { {1, 2L}, {3, 4L} };
```





# Structs and Assignments

```
struct S {  
    int i;  
    long l;  
};  
  
struct S as[2] =  
    { {1, 2L}, {3, 4L} };  
  
as[1] = as[0];
```



- Assigning one struct variable to another makes a “deep copy” (copies the values)
  - Unlike with arrays, for historical reasons



# Structs and Functions

Passing structs to and returning them from functions behave differently than arrays

- Passing a struct to a function passes it by value, not by reference (pointer)
  - Makes a deep copy: The called function gets its own copy of the passed structure
  - Unlike with arrays, where what is passed is the address of the array (a pointer)
- A function can return a struct
  - Unlike with arrays



# DYNAMIC MEMORY





## Why, Though? Isn't Life Hard Enough?

- So far, all memory we've used was known at compile time (static)
  - Except when we didn't have to manage it, as in `argv[]`
  - Read inputs, but into known and preallocated variables
- This is often not feasible; memory needs are often dependent on runtime inputs, parameters, or state
  - E.g. Arbitrary number of inputs: no. of students records to track is input

How many records are being entered?



Need to allocate memory for that number of records

- E.g. Reading from a resource (file, network, etc.): How big is it?
- E.g. Creating new nodes in a tree until a threshold value is met



# Dynamically Managed Memory

Memory allocated at run-time based on state at that point

The data we've seen so far goes into three memory sections:

Text

- Program machine language code

RODATA

- Read-only data, e.g. string literals

Stack

- Activation records (aka "stackframes"):  
a function call's params and local variables



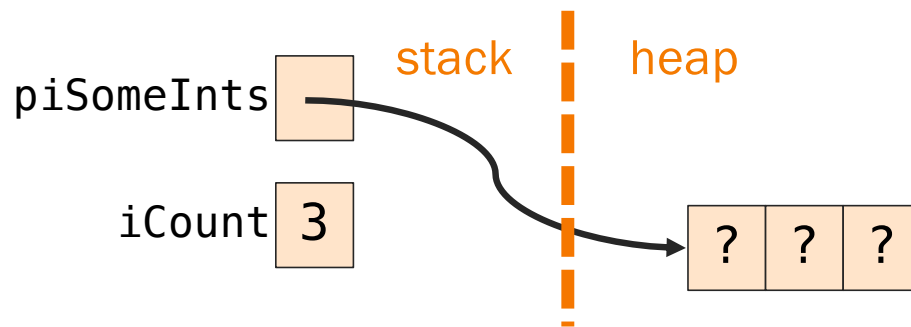
Now, a 4th: the "Heap":  
dynamically allocated storage

Accessed via pointers (on stack etc.)



# Your New Friends: `malloc`, `calloc` and `free`

```
int iCount;  
int *piSomeInts;  
printf("How many ints?");  
scanf("%d", &iCount);  
piSomeInts =  
    malloc(iCount * sizeof(int));
```



Use `sizeof` for portability

`malloc()` doesn't initialize data to 0

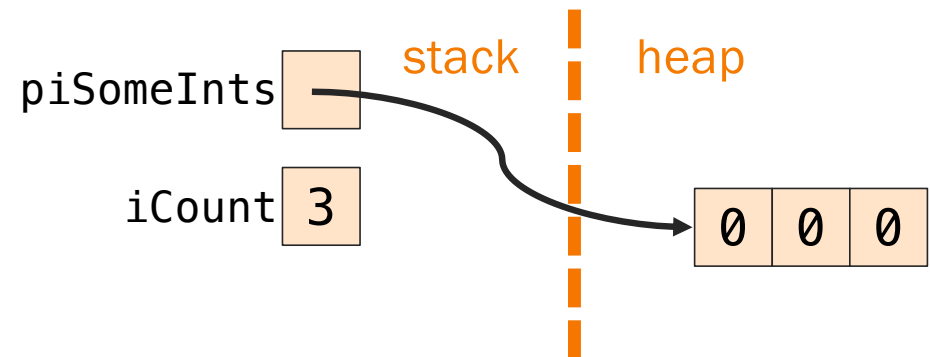


## Your New Friends: malloc, calloc and free

```
int iCount;
int *piSomeInts;
printf("How many ints?");
scanf("%d", &iCount);
piSomeInts =
    malloc(iCount * sizeof(int));
```

malloc() is very flexible, and gives you a chunk of memory. If you know you're allocating for a number of items of some kind: calloc()

```
int iCount;
int *piSomeInts;
printf("How many ints?");
scanf("%d", &iCount);
piSomeInts =
    calloc(iCount, sizeof(int));
```

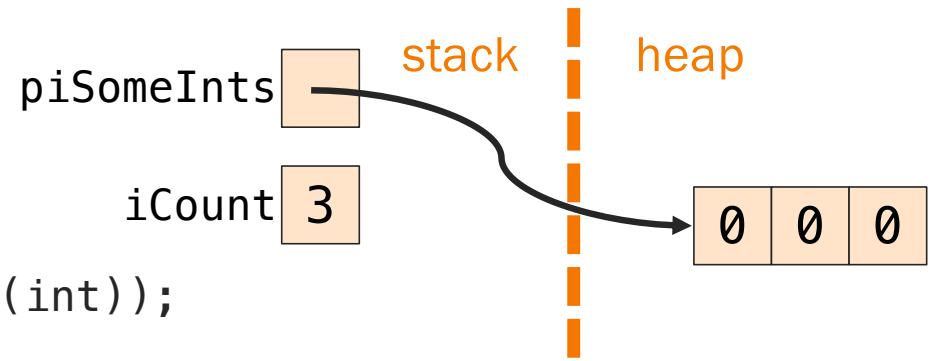


calloc() initializes data to 0



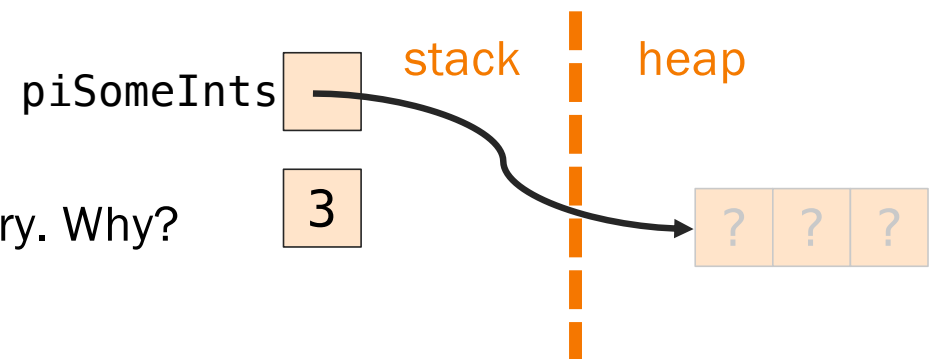
# Your New Friends: malloc, calloc and free

```
int iCount;
int *piSomeInts;
printf("How many ints?");
scanf("%d", &iCount);
piSomeInts = calloc(iCount, sizeof(int));
```



What if you no longer need the memory?  
What do you do in Java?

```
free(piSomeInts);
```



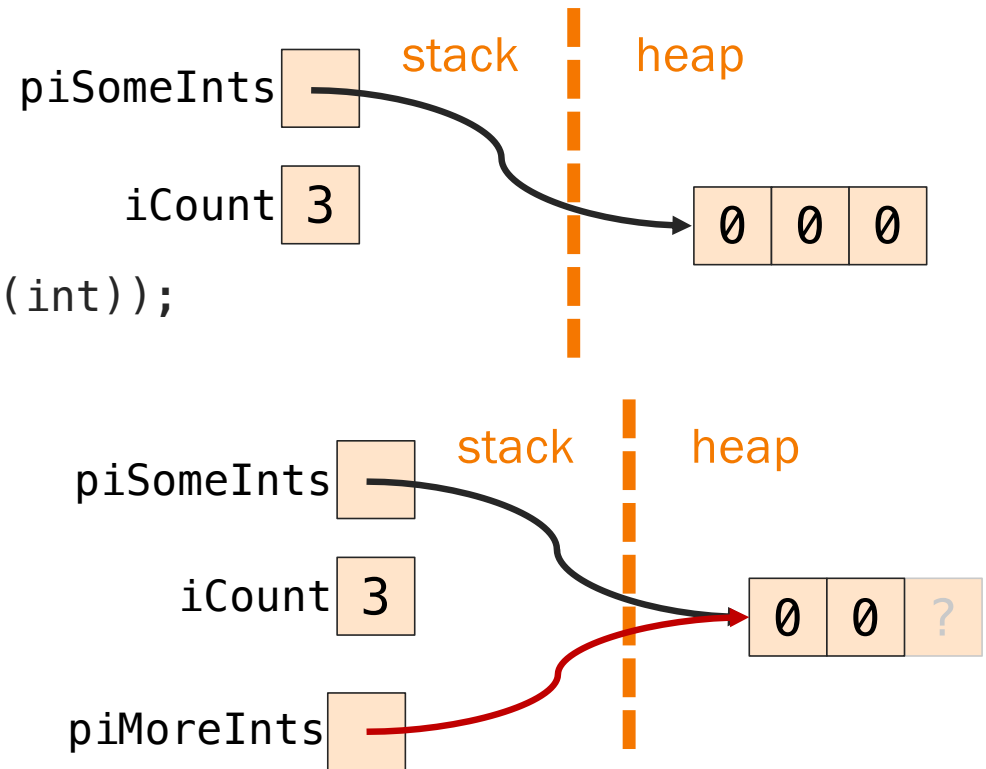
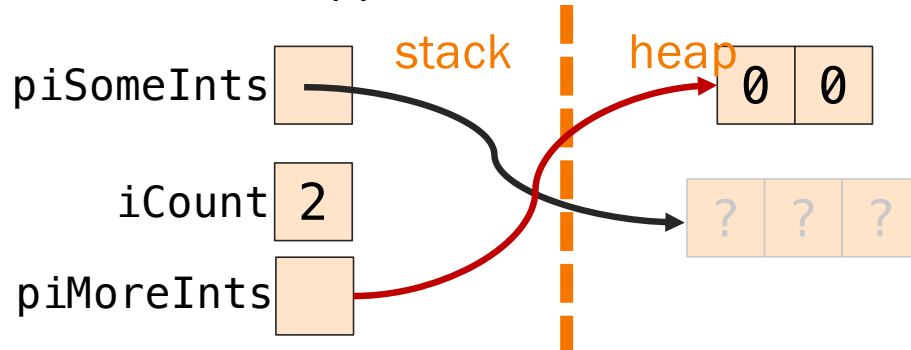
Freed memory can be reused by system  
`piSomeInts` keeps pointing to the memory. Why?  
Hmmm.... "Dangling pointer"  
What about the contents of the memory?



# I Need Less, not Nothing: realloc

```
int iCount;
int *piSomeInts, *piMoreInts;
printf("How many ints?");
scanf("%d", &iCount);
piSomeInts = calloc(iCount, sizeof(int));
piMoreInts = realloc(piSomeInts,
    (iCount-1)*sizeof(int));
```

This could happen too:

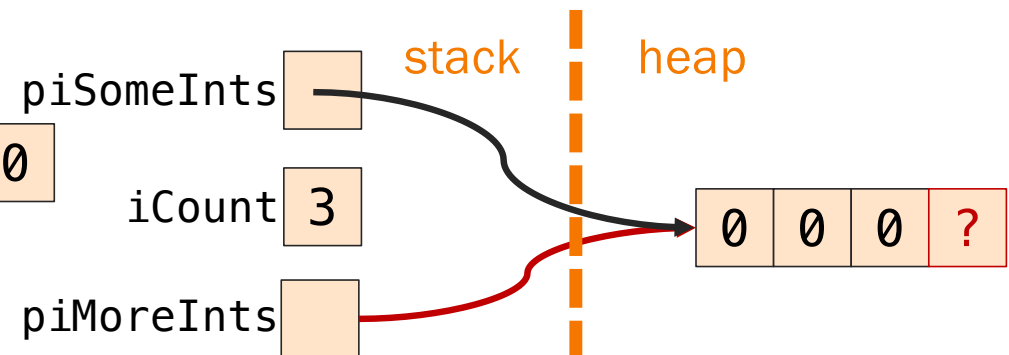
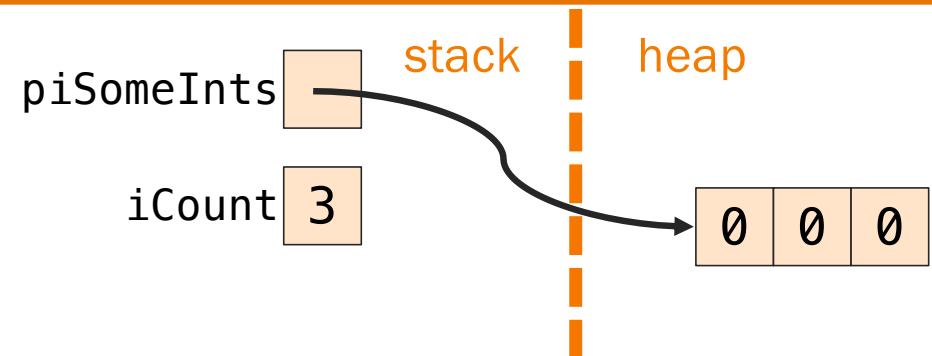
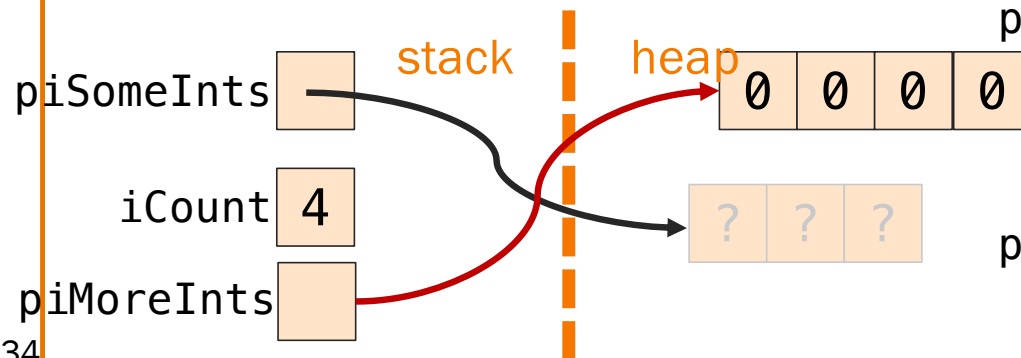


Why a new pointer?



# I Need to Add More: realloc

```
int iCount;
int *piSomeInts, *piOtherInts;
printf("How many ints?");
scanf("%d", &iCount);
piSomeInts = calloc(iCount,
    sizeof(int));
piOtherInts = realloc(piSomeInts,
    (iCount+1)*sizeof(int));
```





Sarah Kilian

# DYNAMIC MEMORY DISASTERS

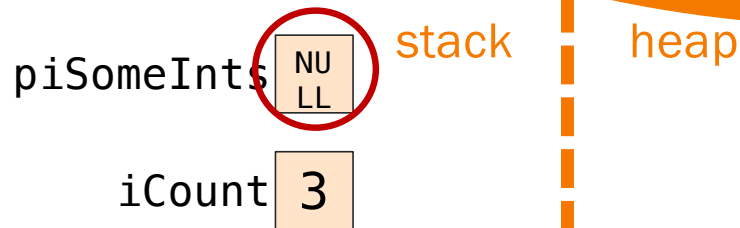




# What Could Go Wrong (malloc, calloc)?

```
int iCount;  
int *piSomeInts;  
printf("How many ints?");  
scanf("%d", &iCount);  
piSomeInts = calloc(iCount, sizeof(int));  
if(piSomeInts == NULL)...  
piSomeInts[0] = ...
```

What if someone calls calloc with a -ve number for iCount?





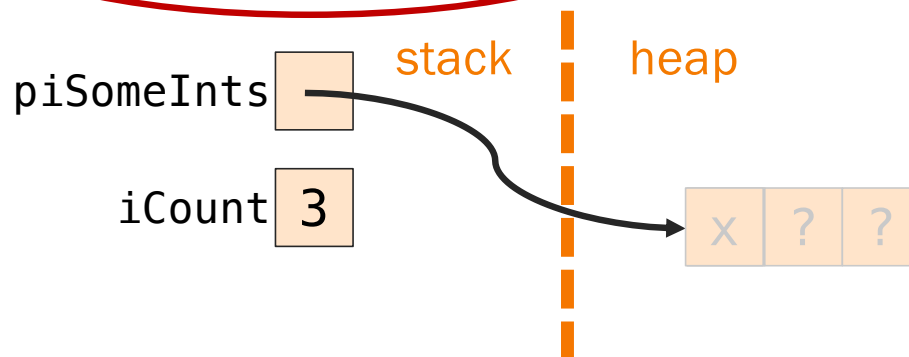
# What Could Go Wrong (free)?

```
int iCount;  
int *piSomeInts;  
printf("How many ints?");  
scanf("%d", &iCount);  
piSomeInts = calloc(iCount, sizeof(int));  
free(piSomeInts);
```

`piSomeInts[0] = x;`  
`free(piSomeInts);`

What happens when you use pointer after freeing it?

What happens when you free pointer after freeing it?

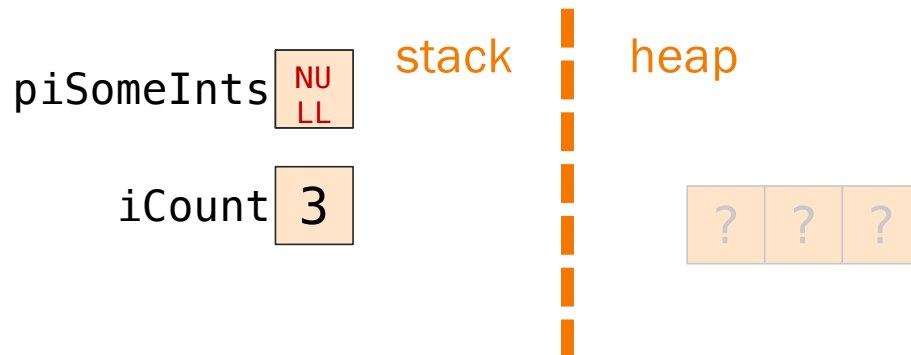




# What Could Go Wrong (free)?

```
int iCount;
int *piSomeInts;
printf("How many ints?");
scanf("%d", &iCount);
piSomeInts = calloc(iCount, sizeof(int));
free(piSomeInts);
piSomeInts = NULL;
piSomeInts[0] = x;
free(piSomeInts);
```

Will crash. But this is a bug, so that's good  
No double-free, since free does nothing for NULL pointer

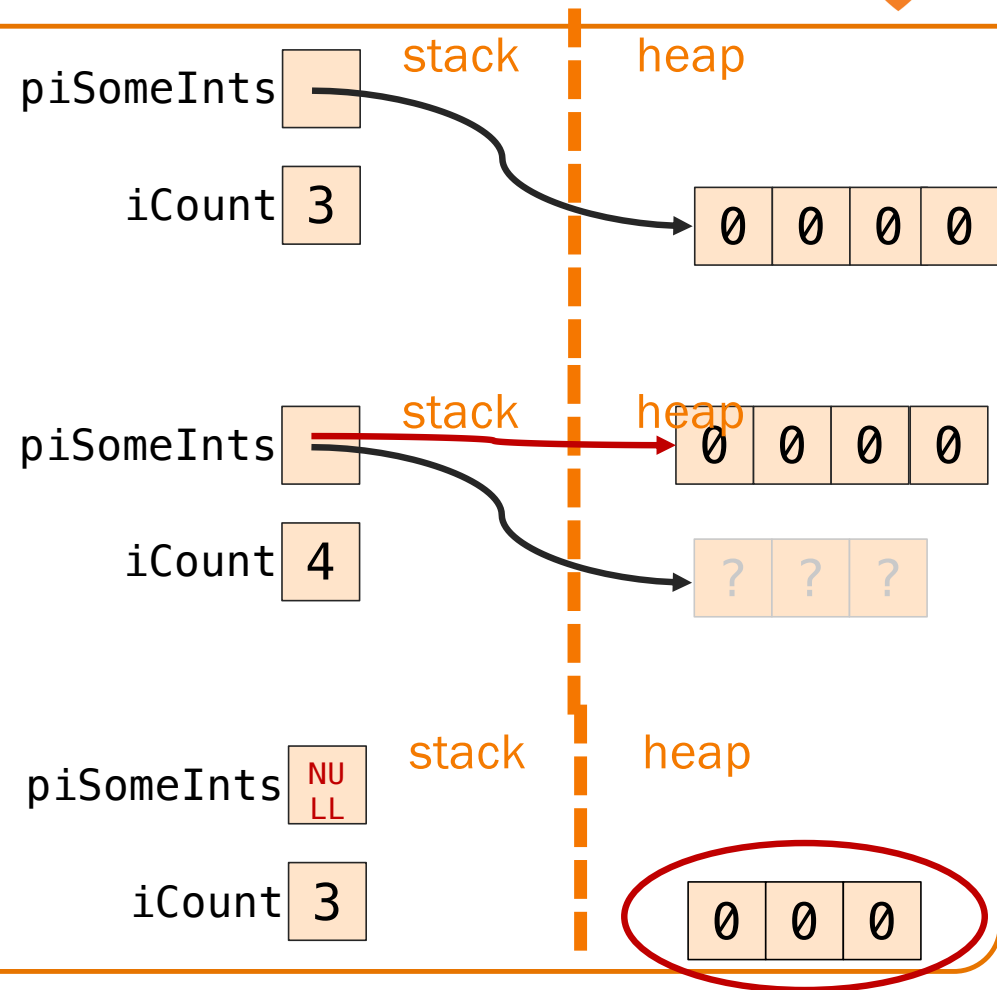




# What could go wrong: realloc

```
int iCount;
int *piSomeInts, *piMoreInts;
printf("How many ints?");
scanf("%d", &iCount);
piSomeInts = calloc(iCount,
    sizeof(int));
piSomeInts = realloc(piSomeInts,
    (iCount+1)*sizeof(int));
if(piSomeInts == NULL)...
```

Check result for NULL before dereference  
Regardless, if realloc fails: memory leak

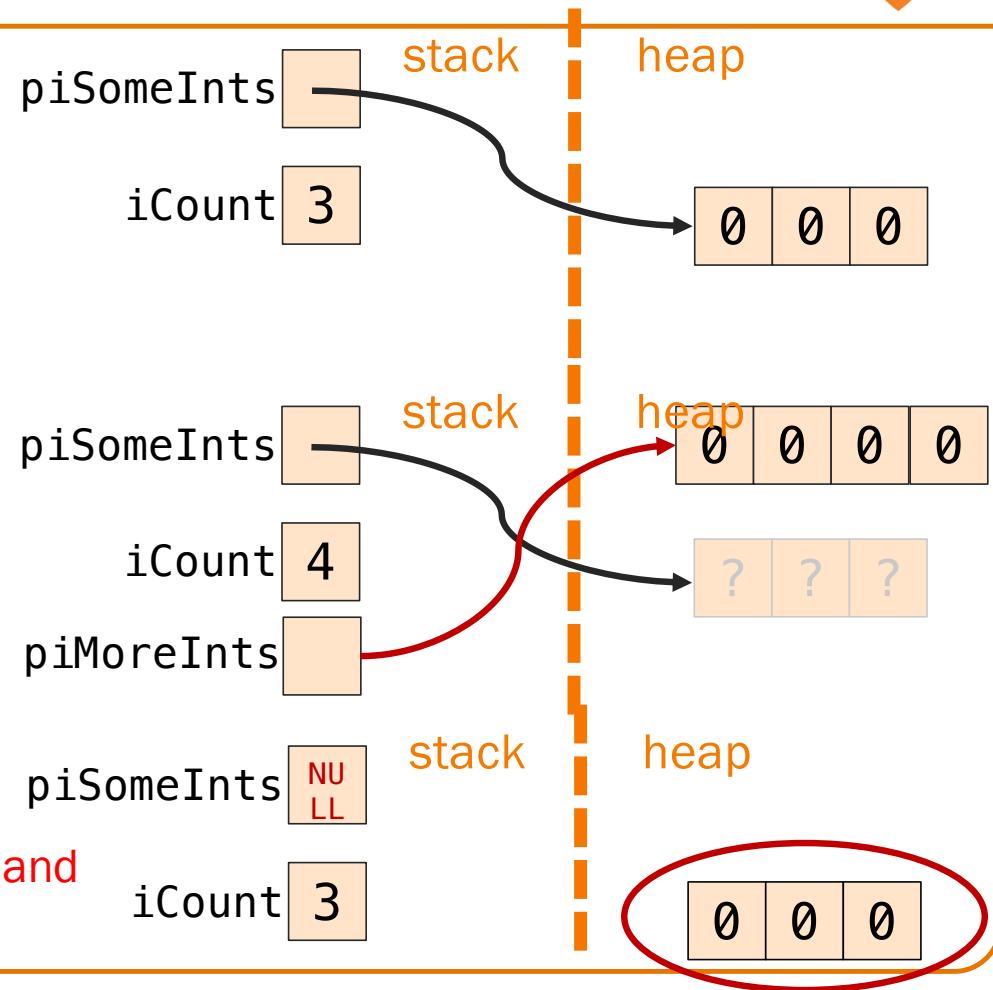




# What could go wrong: realloc

```
int iCount;
int *piSomeInts, *piMoreInts;
printf("How many ints?");
scanf("%d", &iCount);
piSomeInts = calloc(iCount,
    sizeof(int));
piMoreInts = realloc(piSomeInts,
    (iCount+1)*sizeof(int));
if(piMoreInts != NULL) {
    piSomeInts = piMoreInts;
    piMoreInts = NULL; }
```

Check result for NULL before dereference  
Regardless, if realloc fails: memory leak  
Solution: realloc to temp pointer, check NULL, and  
only then update original pointer

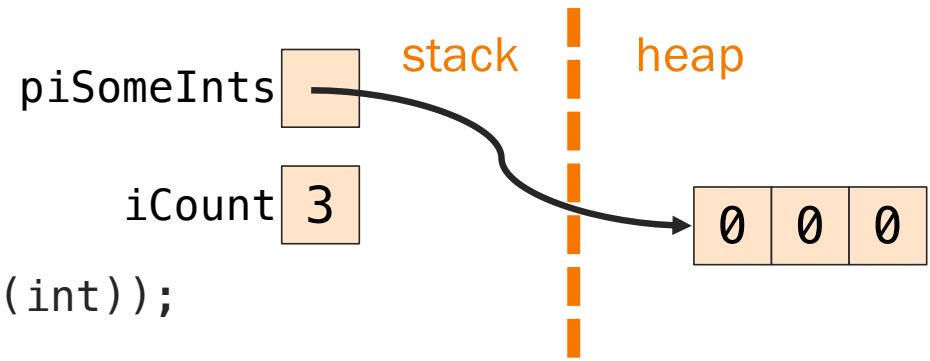




# What could go *really* wrong: realloc

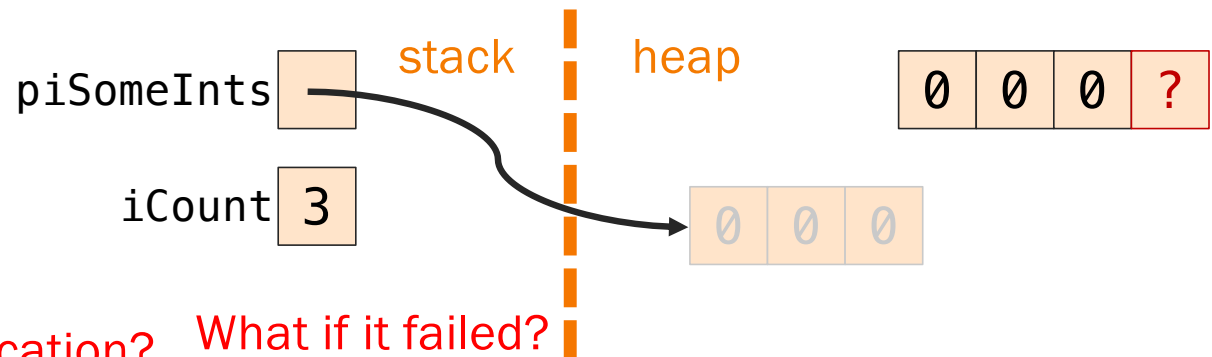
```
int iCount;
int *piSomeInts, *piMoreInts;
printf("How many ints?");
scanf("%d", &iCount);
piSomeInts = calloc(iCount, sizeof(int));
```

```
realloc(piSomeInts,
        (iCount+1)*sizeof(int));
if(piSomeInts == NULL)...
```



If `realloc` changed location:

- Memory Leak
- Dangling Pointer
- Likely eventual double free



What if `realloc` didn't change location? What if it failed?

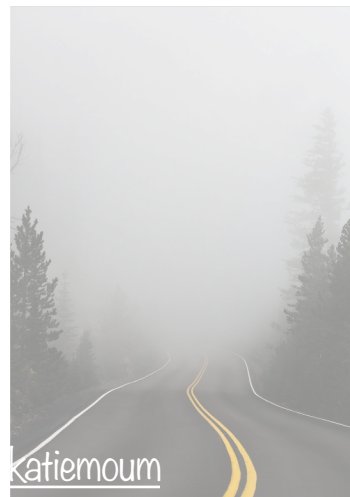


# Assignments Using Dynamic Memory

Assignment 2 does **NOT** use dynamic memory

- Assignments 3 and 4 will use it extensively

**DO NOT** use `{m, c, re}alloc` and `free` on A2



# Sample Exam Problem (Fall 2020 – 14 points / 80)



For the statements in each part of this question, indicate one or more appropriate statuses from this list:

ML - Memory Leak: aka garbage creation

BD - Bad Dereference: derefs NULL or a pointer to memory that was never allocated or has already been freed

IF - Improper Free: frees a pointer to memory that was never allocated or has already been freed

OK - Okay: exhibits no dynamic memory problem

If different statuses could result depending on the result of a call to malloc, calloc, or realloc, then list all possible statuses. You do NOT have to delineate the cases in which each would result.

Each part of this question is independent from the others, but you should assume for each that:

1. p is a char pointer pointing to k bytes that have been allocated in the heap, at least one of which is '\0'.
2. q is a char pointer

a) `strcpy(calloc(strlen(p)+1, sizeof(char)), p);`

b) `for(i=0; i<k; i++) free(p+i);`

c) `free(p); printf("%ul\n", p);`

d) `free(p++);`

e) `q = p; free(q); printf("%s", p);`

f) `free(p); p=NULL; free(p);`

g) `p = realloc(p, 2*k);`

# Sample Exam Problem (Fall 2020 – 22 points /80)



Consider the following program that contains 9 numbered location (0 through 8) :

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
int main(① int argc, ② char** argv){
    ③ int a[10] = {-1, 0, 1};
    ④ double x = 10.75;
    ⑤ double* px = &x;
    ⑥ char* s;
    ⑦ char* f = ⑧ "¥"¥s¥"¥n";
    s = ⑨ calloc(*px, sizeof(*s));
    printf(f, s);
    return strlen(s);
}
```

- how many bytes are allocated, and in which section of memory, for the expression immediately following each callout. Assume this is using gcc217 on armlab, and that the `calloc` call does not return `NULL`.
- What does this program print to standard output?
- How would this program's return value change if callout 8 were replaced with `malloc(x*sizeof(*s))` ; (Assume that, like `calloc`, `malloc` does not return `NULL`.)